Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 8:

- Shared-Memory Programming with OpenMP
  - OpenMP Tasks

OpenMP: Tasks

OpenMP tasks provide a new parallel programming paradigm

- ❑ called the work-oriented paradigm
- ❑ based on the concept of a task pool

In this work-oriented paradigm, units of work (referred to as OpenMP tasks)

- ❑ are generated or "pushed" into the task pool by threads
- ❑ are retrieved or "pulled" off the task pool and executed by threads

OpenMP: Tasks

In classic OpenMP, threads are treated as a fundamental concept:

- ❑ The thread-centric paradigm
- ❑ The OpenMP standard *prior to 3.0* described semantics in terms of threads.

In the work-oriented paradigm, we focus on units of work referred to as tasks:

- ❑ We must now think how the code and data can be broken into units of work that can be executed in parallel, i.e., tasks.
- ❑ We can think of a task as
    - code
    - data (Data Environment)
    - internal control variables (ICVs) [implementation details]

packaged up as an independent schedulable unit.

OpenMP: Tasks

Threads are assigned to perform the work of each task:

- ❑Tasks may be deferred.
- ❑Tasks may be executed immediately.

Tasks enable irregular and recursive computational problems to be parallelized in a natural way, e.g.,

- ❑ Traverse a linked list while performing work on each node in parallel
- ❑ Implement parallel recursive algorithms

OpenMP: Tasks and Threads

- A task is a specific instance of Executable Code and its Data Environment that can be scheduled for execution by a thread.

- Each encountering thread creates a new instance of a task, e.g. code and data.
  - Tasks can be deferred ,i.e., they do not need to be executed immediately.
  - If a task is deferred, some thread in the team executes the task at some time later.
  - The encountering thread that generated a task is not necessarily the same thread that will eventually execute the task.

- Tasks bind to the corresponding innermost enclosing parallel regions.

OpenMP: Task Creation

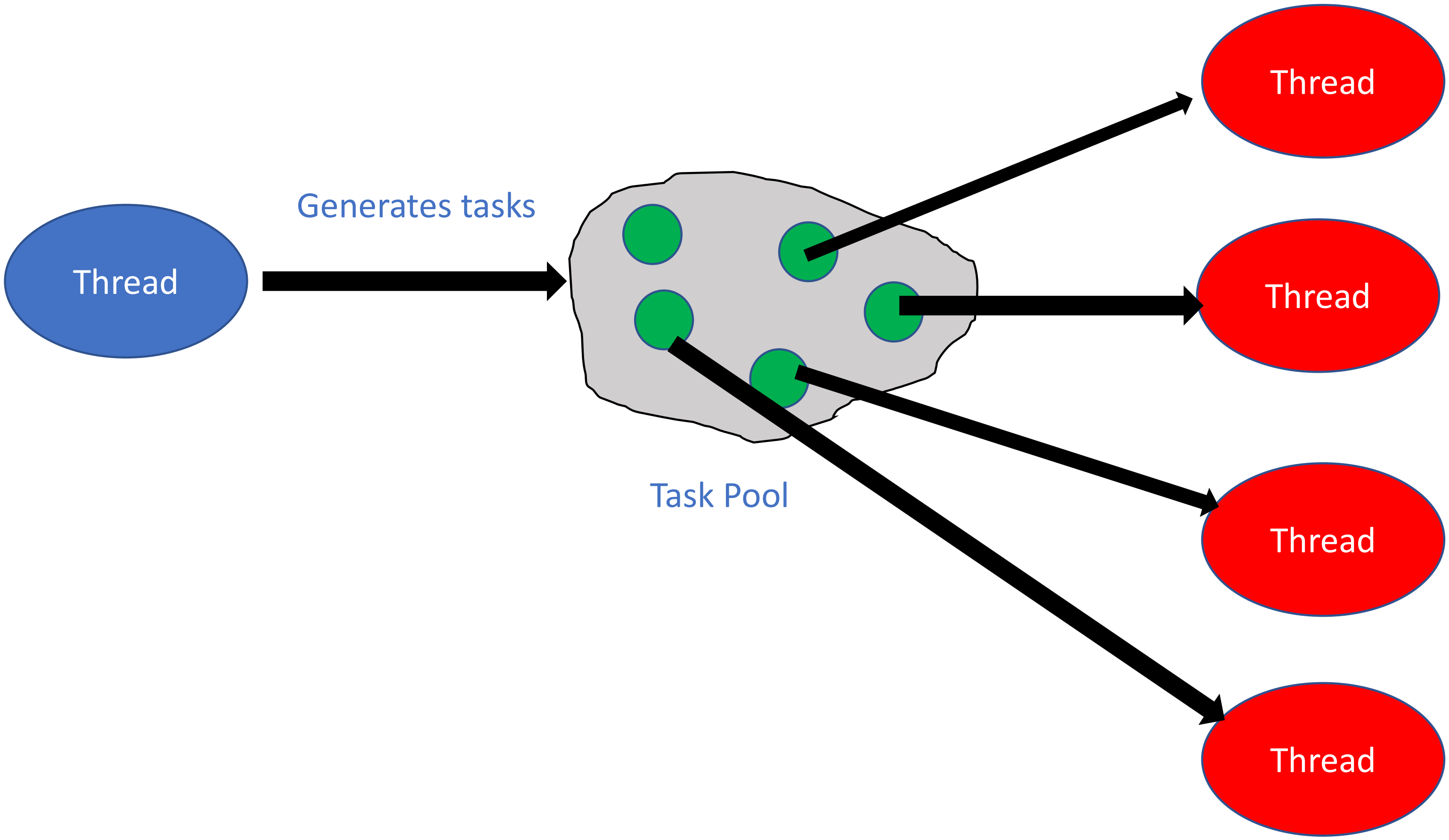
Tasks are created when

- ❑ when a #pragma omp parallel is reached
  - implicit tasks are created per thread
- ❑ when a #pragma omp task is encountered
  - an explicit task is created
- ❑ when a #pragma omp taskloop is encountered
  - explicit tasks per chunk are created
- ❑ when a #pragma omp target is encountered
  - A target task is created

C/C++

```
#pragma omp task [clause]
... structured block ...
```

OpenMP: Task Creation and Execution



OpenMP: Task Creation and Execution

```
#pragma omp parallel num_threads(2)
{
    #pragma omp task //Task A
    {
        #pragma omp critical
        std::cout << "Thread " << omp_get_thread_num() << " executes Task A." << std::endl;
    }

    #pragma omp task //Task B
    {
        #pragma omp critical
        std::cout << "Thread " << omp_get_thread_num() << " executes Task B." << std::endl;
    }

}/*--- End of Parallel Region ---*/
```

```
Thread 0 executes Task A.
Thread 0 executes Task B.
Thread 0 executes Task A.
Thread 1 executes Task B.
```

- Each thread in #pragma omp parallel creates a new task.
- It is non-deterministic which threads are to execute which tasks.

OpenMP: Task Creation and Execution

```
#pragma omp parallel num_threads(2)
{
    #pragma omp task //Task A
    {
        #pragma omp critical
        std::cout << "Thread " << omp_get_thread_num() << " executes Task A." << std::endl;
    }

    #pragma omp task //Task B
    {
        #pragma omp critical
        std::cout << "Thread " << omp_get_thread_num() << " executes Task B." << std::endl;
    }

}/*--- End of Parallel Region ---*/
```

- Each thread inside #pragma omp parallel pushes two tasks to the task pool, i.e., four tasks are pushed to the task pool in total.
- There is an implicit barrier at the end of #pragma omp parallel, which acts as a task synchronization construct.

OpenMP: Task Creation and Execution

```
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    {
        #pragma omp task //Task A
        {
            #pragma omp critical
            std::cout << "\tThread " << omp_get_thread_num() << " executes Task A." << std::endl;
        }

        #pragma omp task //Task B
        {
            #pragma omp critical
            std::cout << "\tThread " << omp_get_thread_num() << " executes Task B." << std::endl;
        }
    }/*--- End of Single Construct ---*/

}/*--- End of Parallel Region ---*/
```

```
Thread 1 executes Task A.
Thread 0 executes Task B.
```

- One of the two threads generates Task A and Task B.
- It is non-deterministic which threads are to execute which tasks.

OpenMP: Task Creation and Execution

```
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    {
        #pragma omp task //Task A
        {
            #pragma omp critical
            std::cout << "\tThread " << omp_get_thread_num() << " executes Task A." << std::endl;
        }

        #pragma omp task //Task B
        {
            #pragma omp critical
            std::cout << "\tThread " << omp_get_thread_num() << " executes Task B." << std::endl;
        }
    }/*--- End of Single Construct ---*/

}/*--- End of Parallel Region ---*/
```

- One of the two threads inside #pragma omp parallel is in charge of enqueuing tasks to the task pool.
- There is an implicit barrier at the end of #pragma omp single, which acts as a task synchronization construct.
- The *nowait* clause can be used on #pragma omp single to remove the implicit barrier.

OpenMP: Task Execution Model

Task Creation:

OpenMP tasks can be created using the following task creation patterns:

- ❑ single task creator
  - with #pragma omp single
- ❑ multiple task creators
- ❑ nested tasks

Task Execution:

All threads in the team are candidates to execute tasks:

- ❑ deferred
- ❑ Immediately executed

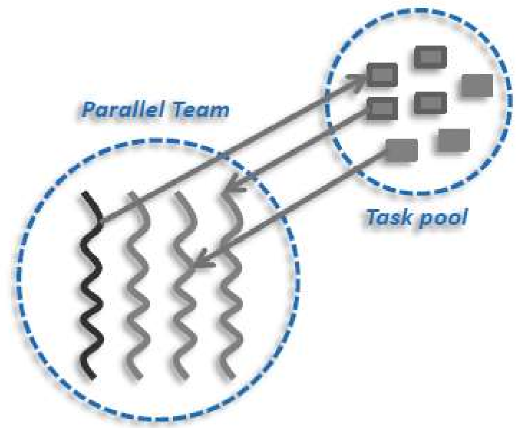


Figure Credit: C. Terboven PPCES 2021

OpenMP: Task Clauses

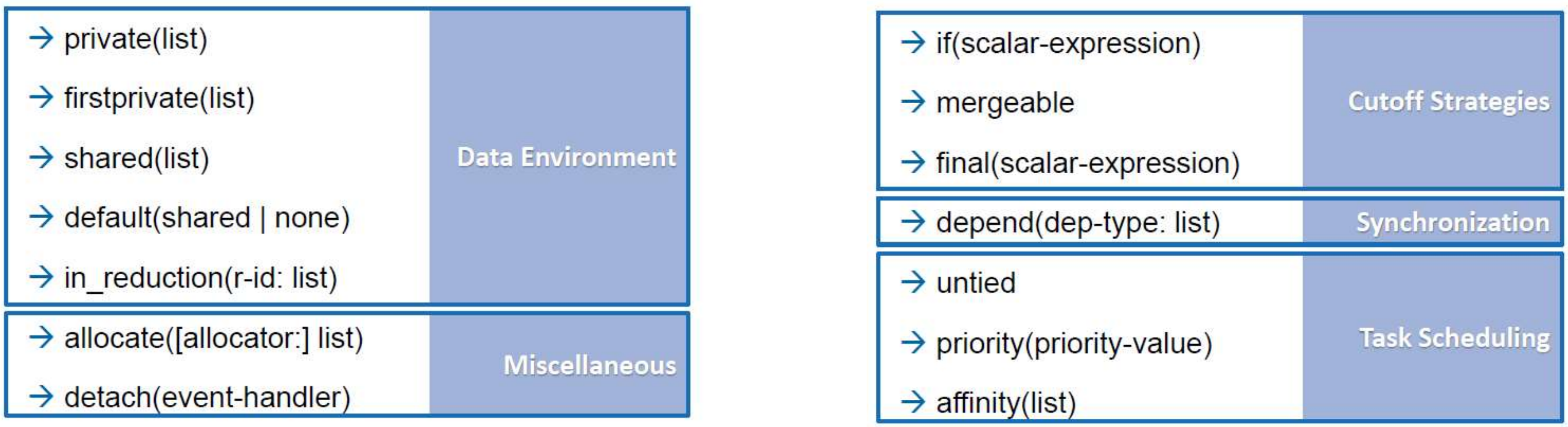


Figure Credit: C. Terboven PPCES 2021

OpenMP: Data Environment

- The Data Environment consists of all the variables associated with the execution of a given task.
  - Because tasks can be *deferred*, the data is *"captured"* at task creation.

- The semantics of data scopes are adapted to deferred execution as follows:
  - For a *shared* variable, the reference to the *shared* variable within the task is to the memory location to that name at the time where the task was created.
  - For a *private* variable, the reference to the *private* variable within the task is to new uninitialized storage that is created when the task is executed.
  - For a *firstprivate* variable, the reference to the *firstprivate* is to new storage that is created and initialized with the value of the existing memory of that name when the task is created.
    - Capturing data structures with *firstprivate* captures only the pointer but not the pointed data.

OpenMP: Data Environment

Data-Scoping Rules: Most rules from #pragma omp parallel still apply.

Pre-determined data-sharing rules for #pragma omp task:

- ❑ Static and global variables are *shared*.
- ❑ Static local variables are shared.
- ❑ Automatic (local) variables are *private*.
- ❑ *Threadprivate* variables are *threadprivate*.

The data-sharing attributes of variables that are not listed in any data-sharing clauses of #pragma omp task and are not pre-determined according to the rules above, are implicitly determined as follows:

Implicit data-sharing rules for #pragma omp task:

- ❑ The *shared* attribute is lexically inherited.
- ❑ In any other case, the variable is *firstprivate*.

OpenMP: Data Environment

```
int a = 1;

void foo()
{
    int b = 2, c=3;
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            //a is shared:          a = 1
            //b is firstprivate:    b = undefined
            //c is shared:          c = 3
            //d is firstprivate:    d = 4
            //e is private:         e = 5
        }
    }/*--- End of Parallel Region ---*/
}
```

- ❑ In #pragma omp parallel, *c* is shared by default
  - In #pragma omp task, the data scoping of *c* is lexically inherited, i.e., *c* is ***shared***.
- ❑ In #pragma omp parallel, *a* is shared by default since it is a global variable.
  - In #pragma omp task, the data scoping of *a* is lexically inherited, i.e., *a* is ***shared***.

OpenMP: Task Scheduling

Tasks can be either *tied* or *untied*:

- Tasks are *tied* by default unless the *untied* clause is explicitly placed on #pragma omp task.
- Tasks remain *tied* to the threads that first execute them.

A *tied* task is always executed by the same thread:

- The *tied* task must be executed by the same thread throughout its entire life cycle.
- The thread that executes the *tied* task does not have to be the same thread that created the task.

Tied tasks can restrict scheduling and may, therefore, negatively impact performance.

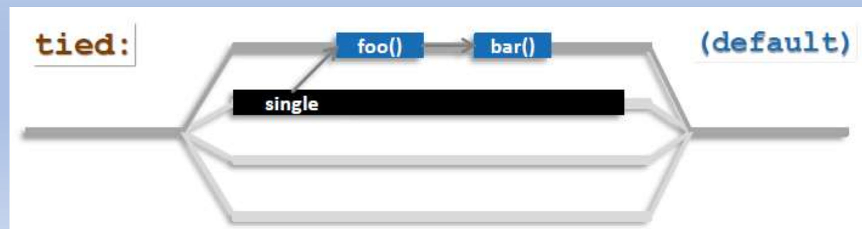


Figure Credit: C. Terboven PPCES 2021

OpenMP: Task Scheduling

In addition to explicit tasks specified with #pragma omp task, OpenMP has also the notion of implicit tasks.

An implicit task is one that is implicitly created when a parallel construct specified with #pragma omp parallel is encountered.

- Each implicit task is assigned to a different thread in the team and remains *tied* to that thread throughout the entire life cycle of the task.

OpenMP: Task Scheduling

The scheduling constraint can be relaxed using the *untied* clause:

- ❑ An *untied* task can migrate between any threads in the team.
- ❑ An *untied* task must be avoided when used with *thread-centric features* such as
  - Thread ID
  - Threadprivate Data [See]
- ❑ The OpenMP runtime has greater flexibility in scheduling tasks.
  - can potentially provide better load-balancing

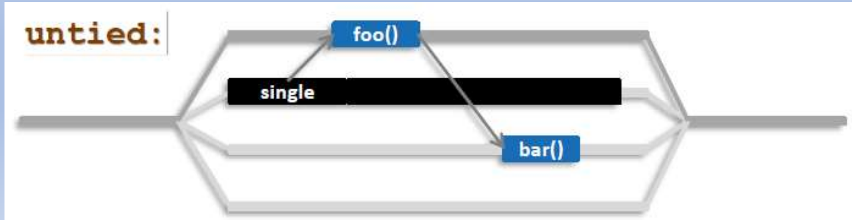


Figure Credit: C. Terboven PPCES 2021

OpenMP: Task Scheduling

Task Scheduling Points (TSP):

C/C++

```
#pragma omp taskyield
```

- ❑ Tasks can be suspended and resumed at TSPs.
- ❑ Implicit TSPs are included at the following locations:
  - Task creation
    - the point of encountering #pragma omp task
  - Task synchronization
    - The point of encountering #pragma omp taskwait
    - The end of #pragma omp taskgroup
    - The completion point of a task
    - The point of encountering an implicit or an explicit barrier
- ❑ Explicit TSPs can be specified using #pragma omp taskyield.

OpenMP: Task Scheduling

At a TSP, a thread that is executing a task *A* may

- temporarily suspend the current task *A* and
- switch to execute a different task *B*
  - It can either start or resume this other task *B*.

OpenMP: Task Scheduling

The taskyield directive explicitly specifies that the current task can be suspended in favor of a different task.

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task untied
    {
        foo();
        #pragma omp taskyield
        bar()
    }
}
```

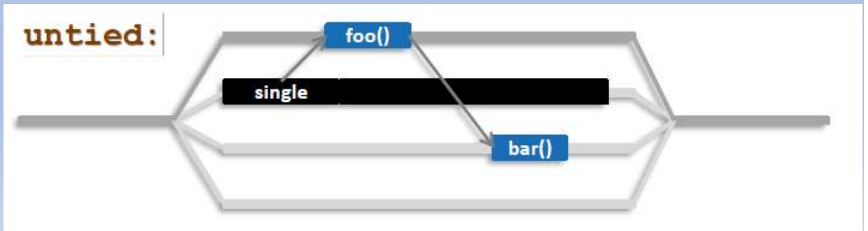


Figure Credit: C. Terboven PPCES 2021

OpenMP: Task Scheduling

Caveat with threadprivate and thread-specific information:

- When a thread encounters a TSP, the OpenMP implementation may choose to suspend the current task and schedule the thread to work on another task.
  - This implies that the value of a *threadprivate* variable or other thread-specific information, e.g. Thread ID, can potentially change across a TSP.
- If the suspended task is a *tied* task, then the thread that resumes executing the task is guaranteed to be the same thread that suspended it.
  - The Thread ID will remain the same after the task is resumed.
- If the suspended task is an *untied* task, the thread that resumes executing the task may be different from the thread that suspended it.
  - Both the Thread ID and the value of a *threadprivate* variable before and after the TSP may be different.

OpenMP: Task Scheduling

Let's sum up the concept of tiedness and untiedness.

❑When a thread encounters a TSP, it may do one of the following:

- it may begin execution of a *tied* task bound to the current team
- it may resume execution of any suspended *tied* task, bound to the current team, to which it is *tied*.
- it may begin execution of an *untied* task bound to the current team
- it may resume execution of any suspended *untied* task bound to the current team

❑If more than one of the above choices is available, it is unspecified as to which option will be chosen by the OpenMP runtime.

OpenMP: Task Synchronization

- ❑ All tasks created by any thread of the current team are guaranteed to have completed at a barrier (implicit or explicit).
  - ❑Explicit barrier : #pragma omp barrier
  - ❑Implicit barrier : at the end of

    #pragma omp parallel

    #pragma omp for

    #pragma omp single

    etc.
- ❑ A task that encounters #pragma omp taskwait is suspended until all child tasks complete.
  - applies only to child tasks, not all descendant tasks !!!
  - provides shallow task synchronization.

C/C++

```
#pragma omp taskwait
```

OpenMP: Task Synchronization

```
1  int main(int argc,
2           char* argv[])
3  {
4      [...]
5      #pragma omp parallel
6      {
7          #pragma omp single
8          {
9              fib(input);
10         }
11     }
12     [...]
13 }
```

```
14 int fib(int n)   {
15     if (n < 2) return n;
16     int x, y;
17     #pragma omp task shared(x)
18     {
19         x = fib(n - 1);
20     }
21     #pragma omp task shared(y)
22     {
23         y = fib(n - 2);
24     }
25     #pragma omp taskwait
26         return x+y;
27 }
```

- Only one thread executing #pragma omp single is responsible for creating tasks.
- This thread creates the two initial tasks (line 17 and 21).
- #pragma omp taskwait (line *25*) is used to wait for the two tasks to complete. Otherwise, *x* and *y* would get lost.

OpenMP: Task Synchronization

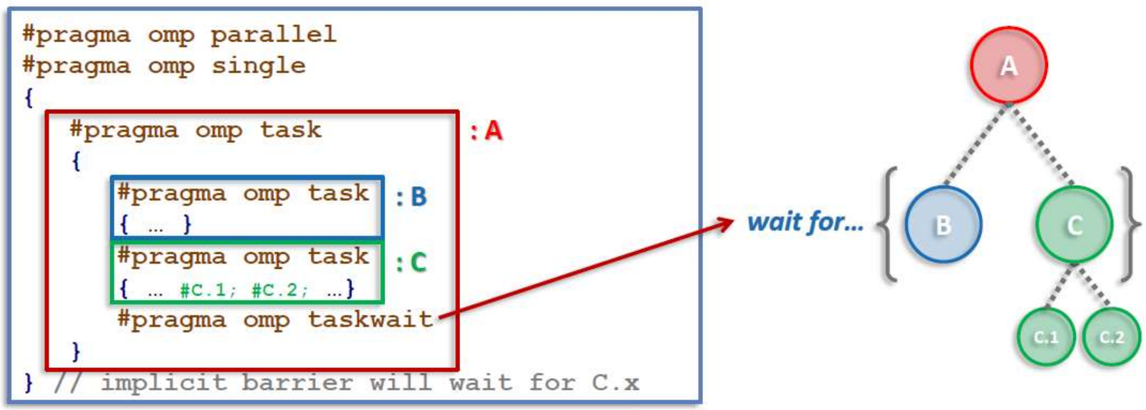


Figure Credit: C. Terboven PPCES 2021

- Task *C.1 and C.2* are not affected:
  - They can execute past #pragma omp taskwait without the need to wait.
  - #pragma omp taskwait only waits for Task *B* and *C.*
- All tasks including *C.1* and *C.2* are guaranteed to have completed at the implicit barrier of the #pragma omp single.

OpenMP: Task Synchronization

Deep Task Synchronization:

- ❑ Tasks can be synchronized across all descendant levels can be synchronized with #pragma omp taskgroup.
  - provides deep task synchronization.
- ❑ The clause allowed is the *task_reduction* clause:
  - of the form *task_reduction(reduction-identifier:list-items)*

```
#pragma omp taskgroup [clause[[,] clause]...]
{structured-block}
```

OpenMP: Task Synchronization

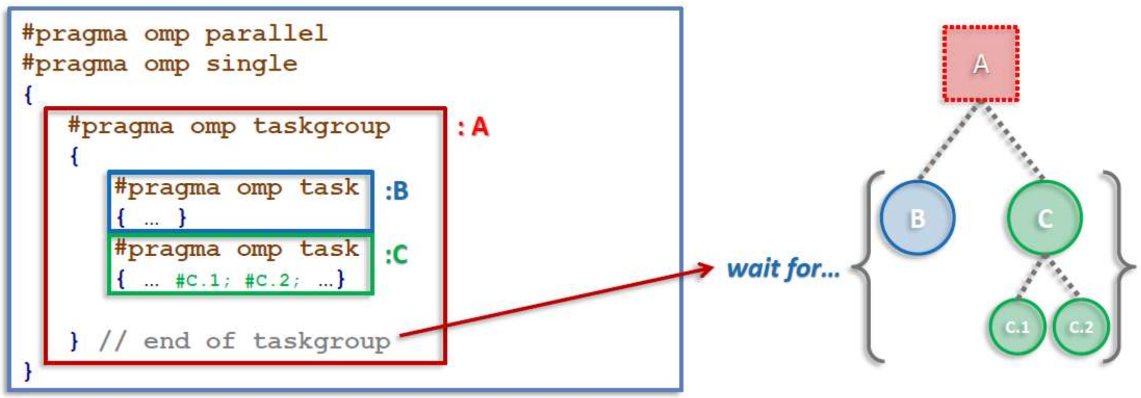


Figure Credit: C. Terboven PPCES 2021

- Task *C.1 and C.2* are affected:
  - They can not execute past #pragma omp taskwait without the need to wait.
- All tasks, namely, *B*, *C*, *C.1* and *C.2* are guaranteed to have completed at the end of the #pragma omp taskgroup.

OpenMP: Task Reduction

```
int res = 0;
node_t* node = NULL;
...
#pragma omp parallel
{
  #pragma omp single
  {
    #pragma omp taskgroup task_reduction(+: res)
    { // [1]
      while (node) {
        #pragma omp task in_reduction(+: res) \
                   firstprivate(node)
        { // [2]
          res += node->value;
        }
        node = node->next;
      }
    } // [3]
  }
}
```

Figure Credit: C. Terboven PPCES 2021

- The taskgroup-scoping reduction *task_reduction* clause
  - registers a new reduction at [1].
  - computes and make the final result available after [3].
- The task *in_reduction* clause
  - allows the task to participle in the reduction operation [2]
- introduced in OpenMP 5.0

OpenMP: Cut-Off Strategies

For recursively defined divide-and-conquer algorithms, a *cut-off* can be defined to avoid *task explosion*.

Mechanisms:

- ❑Programmed cut-off mechanism by explicitly calling a function that does not create tasks
- ❑The *if* clause
- ❑The *final* clause
- ❑The *mergeable* clause

OpenMP: Programmed Cut-Off

```
float sum(const float* a, size_t n)
{
  // base cases
  if(n==0) return 0;
  if(n==1) return*a;

  // recursive case
  size_t half=n/2;
  return sum(a,half) + sum(a+half,n-half);
}
```

OpenMP: Programmed Cut-Off

```
float sum(const float* a, size_t n)
{
  // base cases
  if(n == 0) return 0;
  if(n == 1) return *a;

  // recursive case
  size_t half = n/2;
  float x, y;
  #pragma omp parallel
  #pragma omp single nowait
  {
    #pragma omp task shared(x)
    x = sum(a,half);
    #pragma omp task shared(y)
    y = sum(a+half,n-half);
    #pragma omp taskwait
    x += y;
  }
 return x;
}
```

- The OpenMP code creates tasks recursively.
- Problem:
  - The code creates too many small tasks, i.e., *n* is small.
  - Inefficient
- Solution:
  - stop creating tasks at some *cut-off threshold*

OpenMP: Programmed Cut-Off

```
#define CUTOFF 100  // arbitrary

static float parallel_sum(const float *a,
  size_t n){
  // base case
  if (n <= CUTOFF){
    return serial_sum(a, n);
  }
  // recursive case
  float x, y;
  size_t half = n / 2;
  #pragma omp task shared(x)
  x = parallel_sum(a, half);
  #pragma omp task shared(y)
  y = parallel_sum(a + half, n - half);
  #pragma omp taskwait
  x += y;
  return x;
}
```

```
float sum(const float *a, size_t n){
  float r;
  #pragma omp parallel
  #pragma omp single nowait
  r = parallel_sum(a, n);
  return r;
}

static float serial_sum(const float *a,
  size_t n){
  float x = 0.0;
  for(int i=0; i<n; i++) {
    x += a[i];
  }
  return x;
}
```

Drawback: separate versions of sum !!!

- ❑ *serial_sum*
- ❑ *parallel_sum*

OpenMP: If Clause

- If the *if* clause evaluates to *false*
  - the encountering task, i.e., *parent*, is suspended
  - the new task, i.e., *child*, is executed immediately
    - The new task is denoted as an *"undeferred"* task
  - the parent task resumes once the new task has completed
- This feature can be used to improve the performance by avoiding queuing tasks that are too small.
  - The computational complexity of such small tasks may be considered too small and not worth the overhead for queuing them in the task pool.
- The *if* clause does not affect the child tasks created by the encountering task.
- This feature can be useful for debugging task-based OpenMP applications.

OpenMP: Final Clause

- The *final* clause prevents the creation of further tasks if condition evaluates to *true*.

- The purpose is to reduce overhead, which will also limit the degree of parallelism by preventing too many tasks from being created.

- The new task is created and executed normally
  - However, in its context, its nested tasks will be executed immediately by the same threads
  - The undeferred nested tasks are denoted as *"included"* tasks.

OpenMP: Final Clause

```
#pragma omp task final(e)
{
  #pragma omp task
  { … }
  #pragma omp task
  { … #C.1; #C.2 … }
  #pragma omp taskwait
}
```

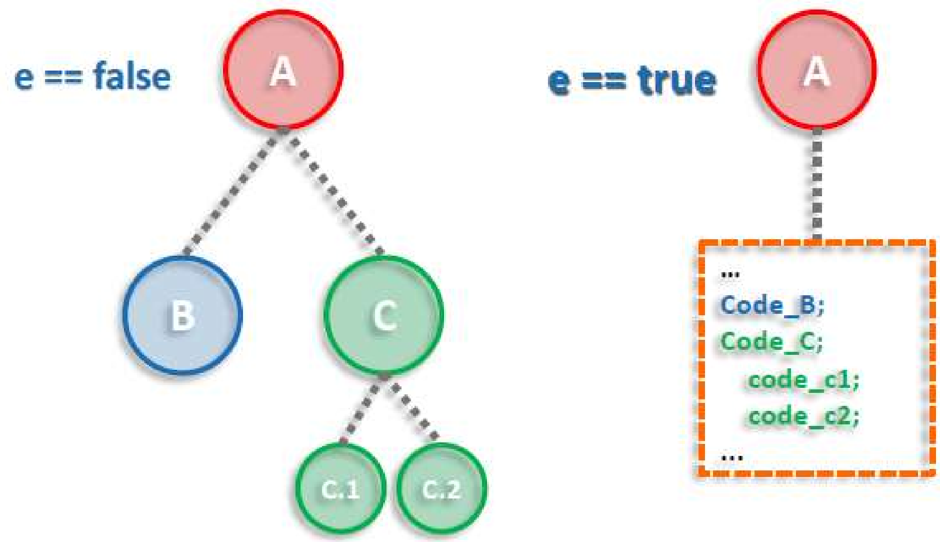


Figure Credit: C. Terboven PPCES 2021

OpenMP: Mergeable Clause

- A task can be created mergeable if
  - it is an *undeferred* task **OR**
  - it is an *included* task
- Its purpose is to
  - reduce OpenMP runtime overhead
  - inline the Data Environment to enable efficient data sharing
- Often used with the *final* clause

- No commercial OpenMP implementations have taken advantage of *mergeable* yet.

Reference

[1] *Ruud van der Pas, Eric Stotzer, and Christian Terboven. 2017. Using OpenMP -- The Next Step: Affinity, Accelerators, Tasking, and SIMD (1st. ed.). The MIT Press.*

[2] *Christian Terboven. 2021. Programming OpenMP. PPCES 2021.*