COMP 4901Q: High Performance Computing (HPC)

Lecture 6: OpenMP 3.0 and Tasking

Instructor: Shaohuai SHI (shaohuais@cse.ust.hk)

Teaching assistants: Mingkai TANG (mtangag@connect.ust.hk)

Yazhou XING (yxingag@connect.ust.hk)

Course website: https://course.cse.ust.hk/comp4901q/

Outline

- OpenMP 3.0 features
- Tasking

Before OpenMP 3.0

- Constructs worked well for many cases
  - But OpenMP's Big Brother had to see everything
    - Loops with a known length at run time
    - Finite number of parallel sections

- Didn't work well with certain common problems
  - Linked lists
  - Divide and conquer
  - Recursive algorithms

OpenMP 3.0 – Major New Features

- Tasking
  - Move beyond loops with generalized tasks and support complex and dynamic control flows
- Nested parallelism support
  - Better definition of and control over nested parallel regions, new APIs to determine nesting structure
- Enhanced loop schedules
  - Support aggressive compiler optimizations and better runtime control
- Loop collapse
  - Combine nested loops together to expose more concurrency

New Directives, APIs, Environment Vars

- Two new directives
  - Task
  - Taskwait
- Nine new API routines
  - omp_set_schedule, omp_get_schedule
  - omp_get_ancestor_thread_num, omp_get_team_size
  - omp_get_level, omp_get_active_level,
  - omp_get_thread_limit
  - omp_set_max_active_levels, omp_get_max_active_levels
- Four new environment variables
  - OMP_STACKSIZE, OMP_WAIT_POLICY
  - OMP_THREAD_LIMIT, OMP_MAX_ACTIVE_LEVELS

Enhancing Nest Parallelism Support

- Setting number of threads
  - With API *omp_set_num_threads*
    - only defined for the outermost level in 2.5 (use *num_threads*)
    - Allowed for all levels in 3.0
- Querying nest levels
  - New API routines for querying nest level, thread id and team size at each nest level
- Resource constraints
  - Maximum number of threads
    - OMP_THREAD_LIMIT, omp_get_thread_limit
  - Maximum number of nest levels
    - OMP_MAX_ACTIVE_LEVELS,
    - omp_set/get_max_active_levels

Schedule Kinds

- STATIC schedule and NOWAIT
  - The 2.5 spec does NOT guarantee the safe use of NOWAIT here. 3.0 clarifies the meaning of STATIC

- SCHEDULE(AUTO)
  - Allows an implementation to do anything it wants

- API routines for SCHEDULE(RUNTIME)
  - omp_set_schedule(kind,modifier)
  - omp_get_schedule(kind,modifier)

kind = {omp_sched_static, omp_sched_dynamic,
omp_sched_guided, omp_sched_auto}
modifier = chunk_size

Loop Collapse

- For perfectly nested loops

```
#pragma omp parallel for
   for (int i = 0; i < 4; i++)
   {
      for (int j = 0; j < 5; j++)
      {
         printf("Thread number is %d\n",
             omp_get_thread_num());
      }
   }
```

Only 4 threads in active state.

```
#pragma omp parallel for collapse(2)
   for (int i = 0; i < 4; i++)
   {
      for (int j = 0; j < 5; j++)
      {
         printf("Thread number is %d\n",
             omp_get_thread_num());
      }
   }
```

20 threads in active state!

Other Environment Variables

- OMP_WAIT_POLICY
  - Controls how threads behave at barriers and locks
  - Values
    - ACTIVE - improve application performance
    - PASSIVE - improve system responsiveness

- OMP_STACKSIZE
  - Controls the OpenMP thread stack size
  - Value in form of
    - size | sizeB | sizeK | sizeM | sizeG (default is K)

Summary of OpenMP 3.0 C/C++ Syntax

Directives

An OpenMP executable directive applies to the succeeding structured block or an OpenMP Construct. A "structured block" is a single statement or a compound statement with a single entry at the top and a single exit at the bottom.

The **parallel** construct forms a team of threads and starts parallel execution.

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
        structured-block
```

clause:
- **if** (*scalar-expression*)
- **num_threads** (*integer-expression*)
- **default** (**shared** | **none**)
- **private** (*list*)
- **firstprivate** (*list*)
- **shared** (*list*)
- **copyin** (*list*)
- **reduction** (*operator*: *list*)

The loop construct specifies that the iterations of loops will be distributed among and executed by the encountering team of threads.

```
#pragma omp for [clause[[,] clause] ... ] new-line
        for-loops
```

clause:
- **private** (*list*)
- **firstprivate** (*list*)
- **lastprivate** (*list*)
- **reduction** (*operator*: *list*)
- **schedule** (*kind*[, *chunk_size*])
- **collapse** (*n*)
- **ordered**
- **nowait**

The most common form of the for loop is shown below.

```
for(var = lb;
    var relational-op b;
    var += incr)
```

The **sections** construct contains a set of structured blocks that are to be distributed among and executed by the encountering team of threads.

```
#pragma omp sections [clause[[,] clause] ...] new-line
    {
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line
        structured-block ]
    ...
    }
```

clause:
- **private** (*list*)
- **firstprivate** (*list*)
- **lastprivate** (*list*)
- **reduction** (*operator*: *list*)
- **nowait**

The **single** construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task.

```
#pragma omp single [clause[[,] clause] ...] new-line
        structured-block
```

clause:
- **private** (*list*)
- **firstprivate** (*list*)
- **copyprivate** (*list*)
- **nowait**

Summary of OpenMP 3.0 C/C++ Syntax

Directives (continued)

The **task** construct defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

#pragma omp task *[clause[[,]clause] ...] new-line*
structured-block

clause: **if** (*scalar-expression*)
untied
default(shared | none)
private (*list*)
firstprivate (*list*)
shared (*list*)

The **master** construct specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.

#pragma omp master *new-line*
structured-block

The **critical** construct restricts execution of the associated structured block to a single thread at a time.

#pragma omp critical *[(name)] new-line*
structured-block

The **barrier** construct specifies an explicit barrier at the point at which the construct appears.

#pragma omp barrier *new-line*

The **taskwait** construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

#pragma omp taskwait *newline*

The **atomic** construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

#pragma omp atomic *new-line*
expression-stmt

expression-stmt: one of the following forms:
x binop = *expr*
x++
++*x*
x--
--*x*

Summary of OpenMP 3.0 C/C++ Syntax

Directives (continued)

The **flush** construct executes the OpenMP flush operation, which makes a thread's temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables.

#pragma omp flush *[(list)] new-line*

The **ordered** construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

#pragma omp ordered *new-line*
    structured-block

The **threadprivate** directive specifies that variables are replicated, with each thread having its own copy.

#pragma omp threadprivate*(list) new-line*

Summary of OpenMP 3.0 C/C++ Syntax

Clauses

Not all of the clauses are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive. Most of the clauses accept a comma-separated list of list items. All list items appearing in a clause must be visible.

Data Sharing Attribute Clauses

Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears.

default(shared|none);
Controls the default data-sharing attributes of variables that are referenced in a **parallel** or **task** construct.

shared(*list***);**
Declares one or more list items to be shared by tasks generated by a **parallel** or **task** construct.

private(*list***);**
Declares one or more list items to be private to a task.

firstprivate(*list***);**
Declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

lastprivate(*list***);**
Declares one or more list items to be private to an implicit task, and causes the corresponding original item to be updated after the end of the region.

reduction(*operator***:***list***);**
Declares accumulation into the list items using the indicated associative operator. Accumulation occurs into a private copy for each list item which is then combined with the original item.

Data Copying Clauses

These clauses support the copying of data values from private or thread-private variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.

copyin(*list***);**
Copies the value of the master thread's *threadprivate* variable to the *threadprivate* variable of each other member of the team executing the **parallel** region.

copyprivate(*list***);**
Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the **parallel** region.

Summary of OpenMP 3.0 C/C++ Syntax

Runtime Library Routines

Execution environment routines affect and monitor threads, processors, and the parallel environment. Lock routines support synchronization with OpenMP locks. Timing routines support a portable wall clock timer. Prototypes for the runtime library routines are defined in the file "omp.h".

Execution Environment Routines

void omp_set_num_threads(int *num_threads***);**
Affects the number of threads used for subsequent **parallel** regions that do not specify a **num_threads** clause.

int omp_get_num_threads(void);
Returns the number of threads in the current team.

int omp_get_max_threads(void);
Returns maximum number of threads that could be used to form a new team using a "parallel" construct without a "num_threads" clause.

int omp_get_thread_num(void);
Returns the ID of the encountering thread where ID ranges from zero to the size of the team minus 1.

int omp_get_num_procs(void);
Returns the number of processors available to the program.

int omp_in_parallel(void);
Returns *true* if the call to the routine is enclosed by an active **parallel** region; otherwise, it returns *false*.

void omp_set_dynamic(int *dynamic_threads***);**
Enables or disables dynamic adjustment of the number of threads available.

int omp_get_dynamic(void);
Returns the value of the *dyn-var* internal control variable (ICV), determining whether dynamic adjustment of the number of threads is enabled or disabled.

int omp_get_thread_limit(void)
Returns the maximum number of OpenMP threads available to the program.

void omp_set_max_active_levels(int *max_levels***);**
Limits the number of nested active **parallel** regions, by setting the *max-active-levels-var* ICV.

int omp_get_max_active_levels(void);
Returns the value of the *max-activelevels-var ICV*, which determines the maximum number of nested active **parallel** regions.

int omp_get_level(void);
Returns the number of nested **parallel** regions enclosing the task that contains the call.

int omp_get_ancestor_thread_num(int *level***);**
Returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread.

Summary of OpenMP 3.0 C/C++ Syntax

Runtime Library Routines (continued)

`int omp_get_team_size(int` *level*`);`

Returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

`int omp_get_active_level(void);`

Returns the number of nested, active **`parallel`** regions enclosing the task that contains the call.

Timing Routines

`double omp_get_wtime(void);`

Returns elapsed wall clock time in seconds.

`double omp_get_wtick(void);`

Returns the precision of the timer used by **`omp_get_wtime`**.

Summary of OpenMP 3.0 C/C++ Syntax

Environment Variables

Environment variable names are upper case, and the values assigned to them are case insensitive and may have leading and trailing white space.

`OMP_SCHEDULE` *type*`[,`*chunk*`]`
Sets the *run-sched-var* ICV for the runtime schedule type and chunk size. Valid OpenMP schedule types are **`static`**, **`dynamic`**, **`guided`**, or **`auto`**. *Chunk* is a positive integer.

`OMP_NUM_THREADS` *num*
Sets the *nthreads-var* ICV for the number of threads to use for **`parallel`** regions.

`OMP_DYNAMIC` *dynamic*
Sets the *dyn-var* ICV *for* the dynamic adjustment of threads to use for **`parallel`** regions. Valid values for *dynamic* are **`true`** or **`false`**.

`OMP_NESTED` *nested*
Sets the *nest-var* ICV to enable or to disable nested parallelism. Valid values for *nested* are true or false.

`OMP_STACKSIZE` *size*
Sets the *stacksize-var* ICV that specifies the size of the stack for threads created by the OpenMP implementation. Valid values for *size* (a positive integer) are *size*, *size***`B`**, *size***`K`**, *size***`M`**, *size***`G`**. If units **`B`**, **`K`**, **`M`** or **`G`** are not specified, size is measured in kilobytes (**`K`**).

`OMP_WAIT_POLICY` *policy*
Sets the *wait-policy-var* ICV that controls the desired behavior of waiting threads. Valid values for *policy* are **`active`** (waiting threads consume processor cycles while waiting) and **`passive`**.

`OMP_MAX_ACTIVE_LEVELS` *levels*
Sets the *max-active-levels-var* ICV that controls the maximum number of nested active **`parallel`** regions.

`OMP_THREAD_LIMIT` *limit*
Sets the *thread-limit-var* ICV that controls the maximum number of threads participating in the OpenMP program.

Summary of OpenMP 3.0 C/C++ Syntax

Details

Operators legally allowed in a reduction

Operator	Initialization value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Schedule types for the loop construct

static Iterations are divided into chunks of size *chunk_size*, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.

dynamic Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.

guided Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. The chunk sizes start large and shrink to the indicated *chunk_size* as chunks are scheduled.

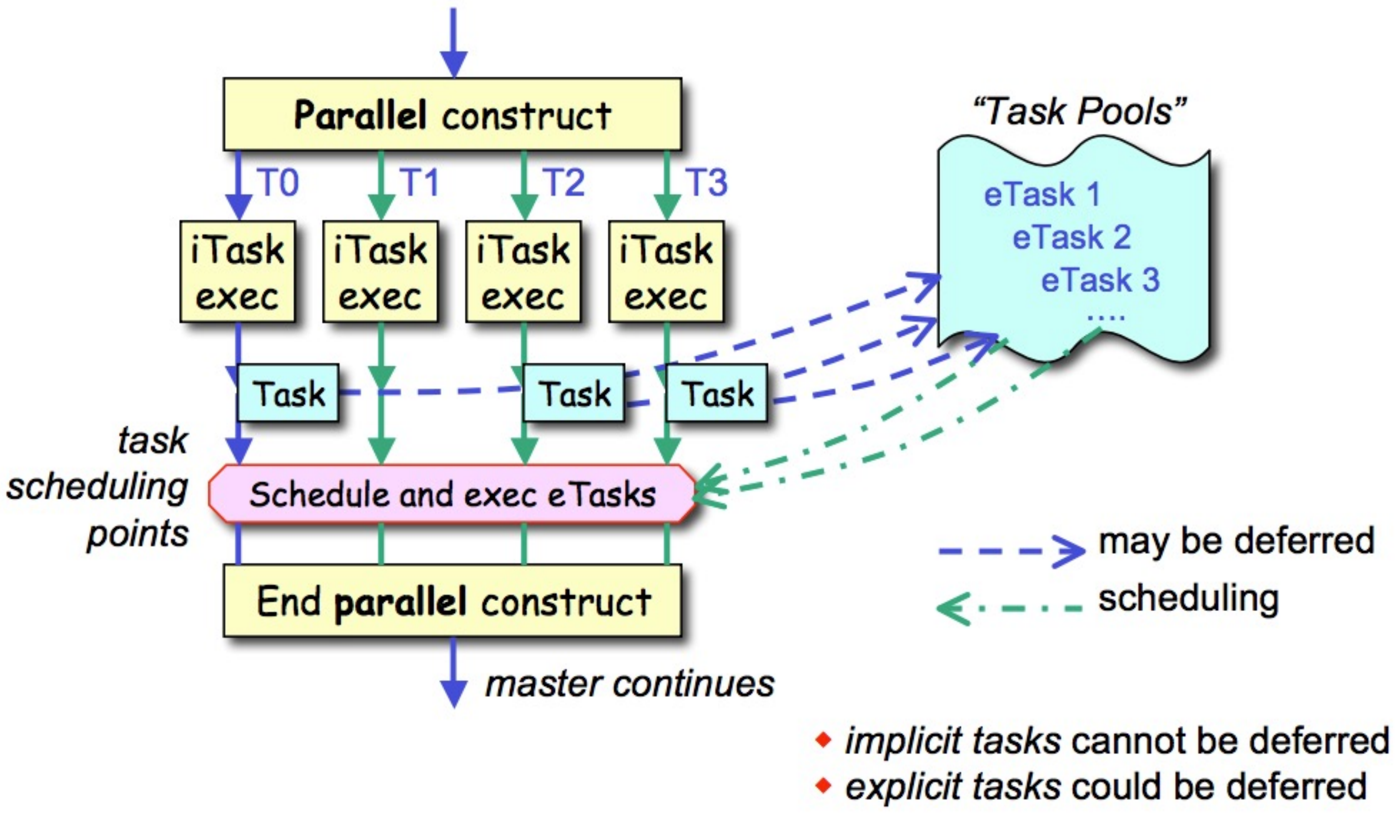
auto The decision regarding scheduling is delegated to the compiler and/or runtime system.

runtime The schedule and chunk size are taken from the run-sched-var ICV.

https://www.openmp.org/wp-content/uploads/OpenMP3.0-SummarySpec.pdf
https://www.openmp.org/wp-content/uploads/spec30.pdf

OpenMP 3.0 Task's View of Execution Model

- A task is a chunk of independent work
  - Different tasks can be executed simultaneously
- The run time system decides on the scheduling of the tasks
  - At certain points (implicit and explicit), tasks are guaranteed to be completed
  - **Implicit tasks** generated by the **parallel** directive
  - **Explicit tasks** generated by the **task** directive



OpenMP 3.0 Task

#pragma omp **task [*clause*…]**

{"this is my task"}

- Data environment is associated with tasks
  - Default **shared** for implicit tasks
  - Default **firstprivate** for explicit tasks (in most cases)
- Task synchronization
  - **taskwait** to synchronize child tasks of a generating task
  - Implicit or explicit **barriers** to wait for all explicit tasks
- Locks are owned by tasks
  - Set by a task, unset by the same task
- ***clause*** is as follows
  - if (scalar-expr)
  - untied
  - default(shared/none)
  - private(list)
  - firstprivate(list)
  - shared(list)

Task Data Scoping

- Some rules from Parallel Regions apply
  - Static and Global variables are shared
  - Automatic Storage (local) variables are private

- If **shared** scoping is not derived by default
  - Orphaned Task variables are firstprivate by default!
  - Non-Orphaned Task variables inherit the shared attribute!
  - Variables are firstprivate unless shared in the enclosing context

Task Data Scoping: Example

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

Task Synchronization

- At implicit or explicit **barrier**
  - #pragma omp barrier
  - Wait for all explicit tasks generated within the current parallel region to complete
- With **taskwait**
  - #pragma omp taskwait
  - Ensure all *child* tasks generated up to the point are complete
- Synchronizing two tasks in the middle
  - Use locks or the “flush” directive

Task Synchronization - Example

```
#pragma omp parallel num_threads(np)
{
#pragma omp task
    function_A();
#pragma omp barriar
#pragma omp single
   {
       #pragma omp task
       function_B();
   }
}
```

Task Scheduling

- Default: Tasks are tied to the thread that first executes them → not necessarily the creator. Scheduling constraints
  - Only the thread a task is tied to can execute it
  - A task can only be suspended at task scheduling points
  - Task creation, task finish, taskwait, barrier, taskyield
  - If task is not suspended in a barrier, executing thread can only switch to a direct descendant of all tasks tied to the thread

- Tasks created with the untied clause are never tied
  - Resume at task scheduling points possibly by different thread
  - But: More freedom to the implementation, e.g. load balancing

Implications from Task

- Good part
  - Data is associated with tasks. It forces programmers to think harder on data locality, potentially producing better code
  - Work stealing (possible) may improve performance
  - The model covers much boarder range of applications

- The catch
  - Programmers have to be very careful on the scope of variables and make sure they do not disappear or go out of scope before the end of a task execution
  - **threadprivate** data may not be preserved

Task Switching - the Hard Part

- Definition
  - The act of a thread switching its execution from one task to another task
- Action
  - Suspend or finish the current task
  - Resume or start another task (with constraints)
  - Can only occur at the **task scheduling points**
- Task scheduling points
  - Right after the task construct
  - At the end of a task region
  - At **taskwait**, implicit or explicit barrier
  - At implementation defined places (for untied tasks only)

Task Switching - the Bad Part

- Definition
  - A task suspended by one thread, but resumed by a different thread

- Tied vs Untied tasks
  - For a tied task, thread switching is not allowed (i.e., the task is tied to the same thread)
    - task without the "untied" clause, implicit tasks
  - For an untied task, thread switching is allowed
    - explicit task with the "untied" clause

- Implication
  - threadprivate data may not persist any longer
  - Locks can only be owned by tasks, not by threads

Task Switching - the Ugly Part

- For a task with **if(expr)** evaluated as false
  - The task (whether tied or untied) gets executed **immediately**

- For an **untied** task
  - No rules on how it gets scheduled
  - Rely on smart compiler/implementation for performance

- For a **tied** task not at a **barrier**
  - Subject to the Task Scheduling Constraint
    - Defined by a set of task regions currently tied to the thread
    - The task can be scheduled
      - if the set is empty, or
      - if the task is a descendant of every task in the set

Other New Terminology

- Task region
  - A region consisting of all code encountered during a task execution
- Descendant task
  - A child task of the task or one of its descendant tasks
- Ancestor thread
  - A parent thread or one of its parent thread’s ancestor thread
- Active parallel region
  - A parallel region with a team of more than one thread
- Inactive parallel region
  - A parallel region with a team of only one thread

Example 1

- Write a program that prints either "A race car" or "A car race" and maximize the parallelism

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
  printf("A ");
  printf("race ");
  printf("car ");
  printf("\n");
  return 0;
}
```

Output:
"A race car"

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
        printf("A ");
        printf("race ");
        printf("car ");
  }
  printf("\n");
  return 0;
}
```

Output:
"A A race race car car" or
"A race A car race car" or
...

Example 1 cont.

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
    #pragma omp single
    {
            printf("A ");
            printf("race ");
            printf("car ");
    }
  }
  printf("\n");
  return 0;
}
```

Output: (only 1 thread)
“A race car”

Example 1 cont.

- Tasks can be executed in arbitrary order

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
 #pragma omp parallel
 {
   #pragma omp single
   {
            printf("A ");
            #pragma omp task
            {printf("race "); }
            #pragma omp task
            {printf("car "); }
   }
 }
 printf("\n");
 return 0;
}
```

Output:
“A race car” or
“A car race”

Example 1 Extension

- Have the sentence end with “is fun to watch”

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
  #pragma omp parallel
  {
    #pragma omp single
    {
            printf("A ");
            #pragma omp task
            {printf("race "); }
            #pragma omp task
            {printf("car "); }
            printf(”is fun to watch ");
    }
  }
  printf("\n");
  return 0;
}
```

Tasks are executed at **a task execution point**!

Output:
“A is fun to watch race car” or
“A is fun to watch car race”

Example 1 Extension cont.

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[])
{
 #pragma omp parallel
 {
   #pragma omp single
   {
            printf("A ");
            #pragma omp task
            {printf("race "); }
            #pragma omp task
            {printf("car "); }
            #pragma omp taskwait
            printf(”is fun to watch ");
   }
 }
 printf("\n");
 return 0;
}
```

Output:
“A car race is fun to watch” or
“A race car is fun to watch”

Example 2 Fibonacci

```
#include<stdlib.h>
#include<stdio.h>
int fibo(int n) {
    if (n < 2) return n;
    return fibo(n-1)+fibo(n-2);
}
int main(int argc, char *argv[])
{
   fibo(input);
}
```

Example 2 Fibonacci cont.

- First version with Tasking (omp-v1)

```
int main(int argc, char *argv[])
{
#pragma omp parallel
  {
    #pragma omp single
    {
      fibo(input);
    }
  }
}
```

```
int fibo(int n) {
   if (n < 2) return n;
   int x, y;
#pragma omp task shared(x)
{
  x = fibo(n-1);
}
#pragma omp task shared(y)
{
 y = fibo(n-2);
}
#pragma omp taskwait
  return x+y;
}
```

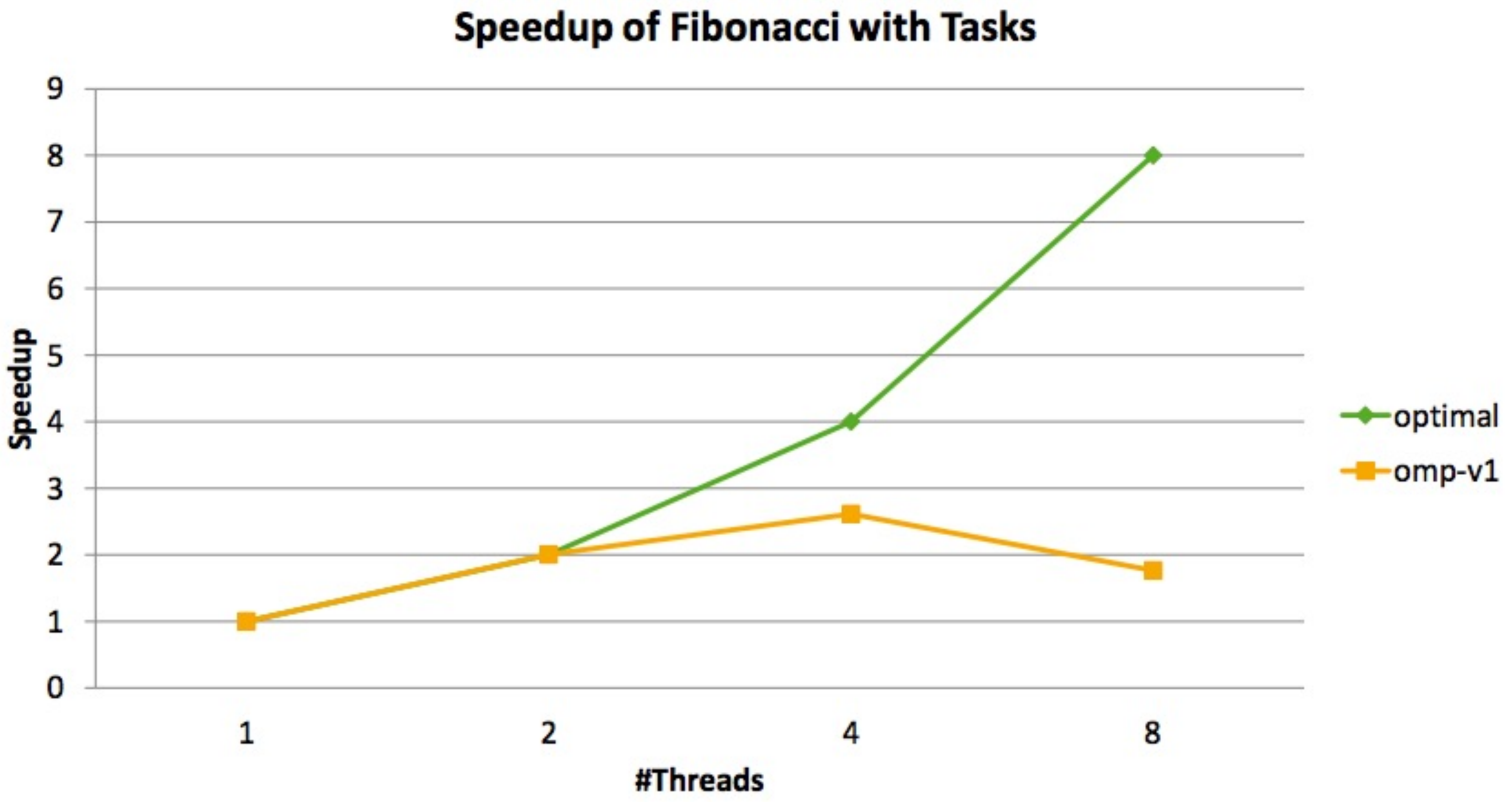
- Only one Thread enters fibo() from main(), it is responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would be lost

Example 2 Fibonacci cont.

- Overhead of task creation prevents better scalability!

```
int fibo(int n) {
   if (n < 2) return n;
   int x, y;
#pragma omp task shared(x)
{
  x = fibo(n-1);
}
#pragma omp task shared(y)
{
 y = fibo(n-2);
}
#pragma omp taskwait
  return x+y;
}
```

omp-v1



Example 2 Fibonacci cont.

- First version with Tasking (omp-v2)

```
int main(int argc, char *argv[])
{
#pragma omp parallel
  {
    #pragma omp single
    {
      fibo(input);
    }
  }
}
```

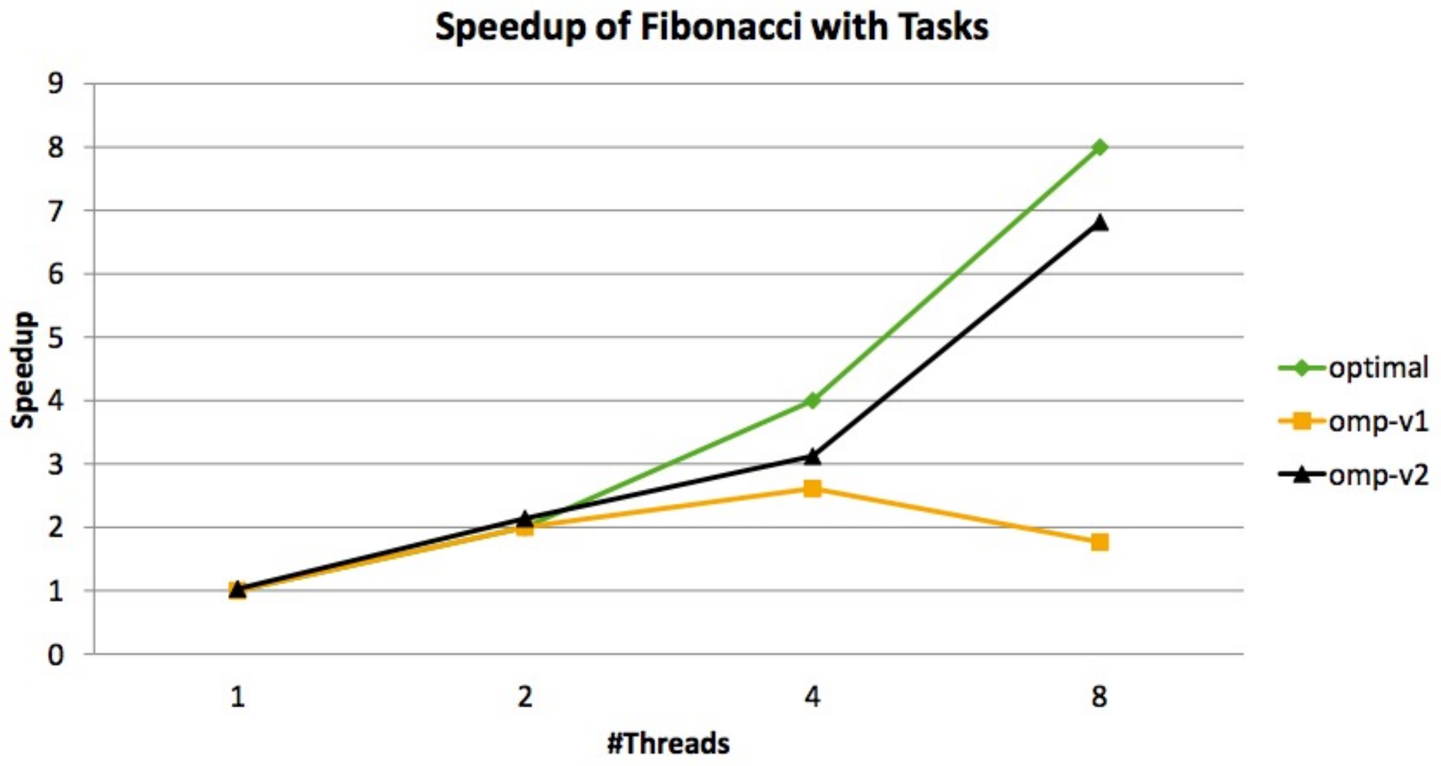
```
int fibo(int n) {
   if (n < 2) return n;
   int x, y;
#pragma omp task shared(x) \
     if (n > 30)
{
  x = fibo(n-1);
}
#pragma omp task shared(y) \
     if (n > 30)
{
 y = fibo(n-2);
}
#pragma omp taskwait
  return x+y;
}
```

Example 2 Fibonacci cont.

- Speedup is ok, but we still have some overhead when running with 4 or 8 threads

```
int fibo(int n) {
   if (n < 2) return n;
   int x, y;
#pragma omp task shared(x) \
     if (n > 30)
{
  x = fibo(n-1);
}
#pragma omp task shared(y) \
     if (n > 30)
{
 y = fibo(n-2);
}
#pragma omp taskwait
  return x+y;
}
```

omp-v2



Example 2 Fibonacci cont.

- First version with Tasking (omp-v3)

```
int main(int argc, char *argv[])
{
#pragma omp parallel
  {
    #pragma omp single
    {
      fibo(input);
    }
  }
}
```

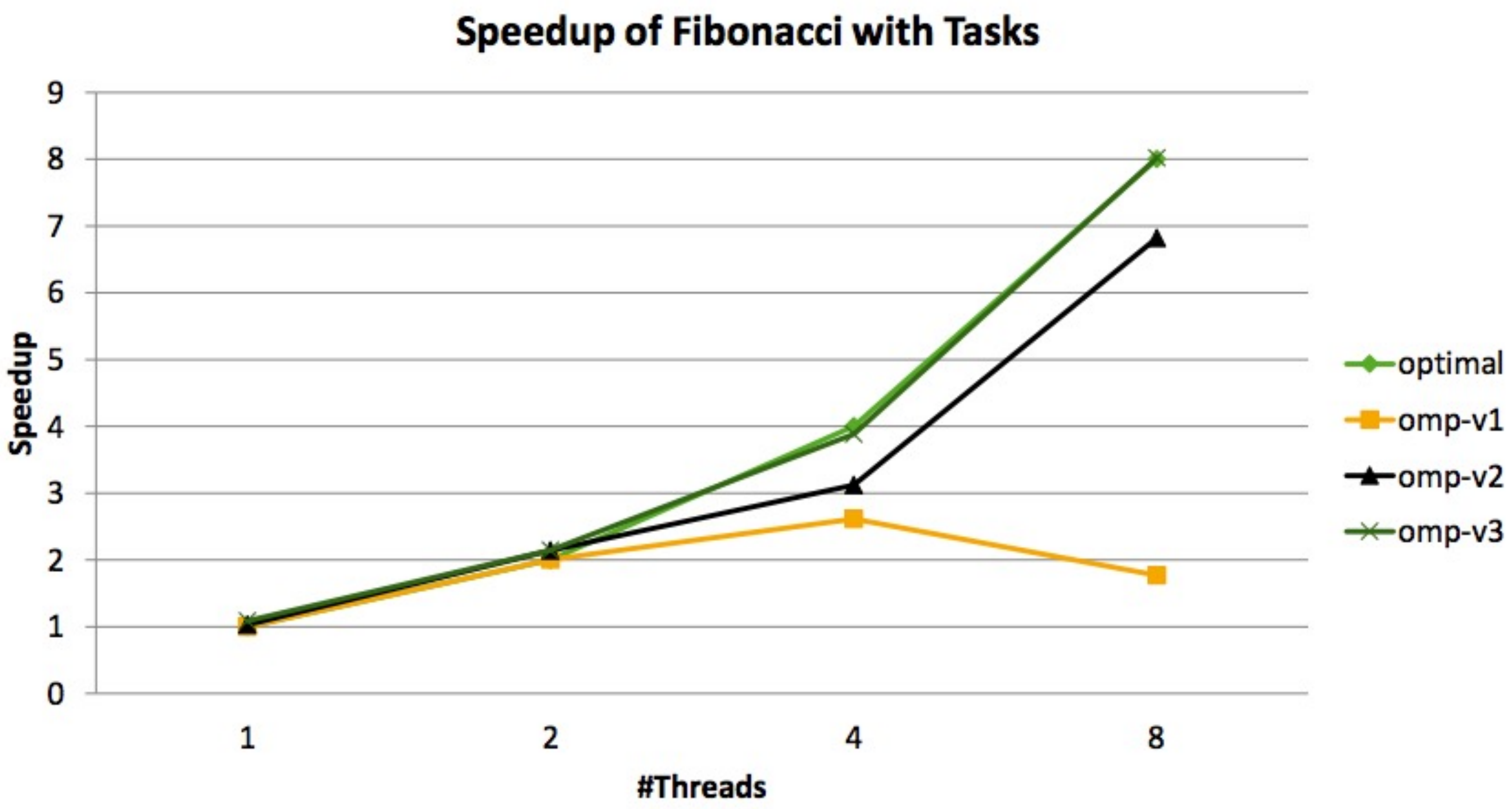
```
int fibo(int n) {
   if (n < 2) return n;
   if (n <= 30) return serial_fibo(n);
   int x, y;
#pragma omp task shared(x) \
     if (n > 30)
{
  x = fibo(n-1);
}
#pragma omp task shared(y) \
     if (n > 30)
{
  y = fibo(n-2);
}
#pragma omp taskwait
  return x+y;
}
```

Example 2 Fibonacci cont.

- It seems perfect!

```
int fibo(int n) {
   if (n < 2) return n;
   if (n <= 30) return serial_fibo(n);
   int x, y;
#pragma omp task shared(x) \
      if (n > 30)
{
   x = fibo(n-1);
}
#pragma omp task shared(y) \
      if (n > 30)
{
  y = fibo(n-2);
}
#pragma omp taskwait
   return x+y;
}
```

omp-v3



Example 3 - Linked List

- Hard to do before tasking:
  - First count number of iterations, then convert while loop to for loop
- Use task is simple
  - Use the single construct : one thread generates the tasks
  - All other threads execute the tasks as they become available

```
…
my_pointer = listhead;
while(my_pointer) {
  do_independent_work(my_pointer);
  my_pointer = my_pointer->next;
} // End of while loop
…
```

Example 3 - Linked List cont.

```
…
my_pointer = listhead;
#pragma omp parallel
{
   #pragma omp single
   {
         while(my_pointer) {
           #pragma omp task firstprivate(my_pointer)
           {do_independent_work(my_pointer); }
           my_pointer = my_pointer->next;
         } // End of while loop
    }
}
…
```

Summary

- Nest parallelism support
  - Better definition of and control over nested parallel regions, new APIs to determine nesting structure
- Enhanced loop schedules
  - Support aggressive compiler optimizations and better runtime control
- Loop collapse
  - Combine nested loops together to expose more concurrency
- Tasking
  - Move beyond loops with generalized tasks and support complex and dynamic control flows