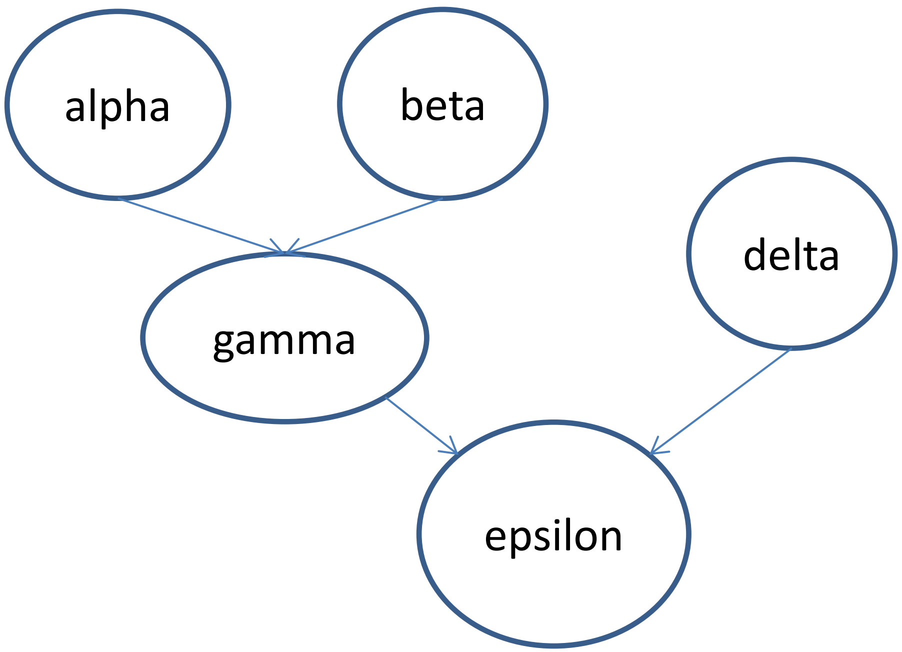
```
V=alpha();
W=beta();
X=gamma(v,w);
Y=delta();
printf("%g\n", epsilon(x,y));
```



Data dependence diagram

Functions alpha, beta, delta may be executed in parallel

Credits: Zhiliang Xu

Worksharing sections Directive

sections directive enables specification of task parallelism

- Sections construct gives a different structured block to each thread.

```
#pragma omp sections [clause list]
                     private (list)
                     firstprivate (list)
                     lastprivate (list)
                     reduction (operator: list)
                     nowait
{
#pragma omp section
    structured_block
#pragma omp section
    structured_block
}
```

Credits: Zhiliang Xu

```c
#include "omp.h"
#define N 1000
int main(){
   int i;
   double a[N], b[N], c[N], d[N];
   for(i=0; i<N; i++){
      a[i] = i*2.0;
      b[i] = i + a[i]*22.5;
   }
   #pragma omp parallel  shared(a,b,c,d) private(i)
   {
      #pragma omp sections nowait
      {
         #pragma omp section
            for(i=0; i<N;i++) c[i] = a[i]+b[i];
         #pragma omp section
            for(i=0; i<N;i++) d[i] = a[i]*b[i];
       }
   }
}
```

Two tasks are computed concurrently

By default, there is a barrier at the end of the sections. Use the "nowait" clause to turn of the barrier.

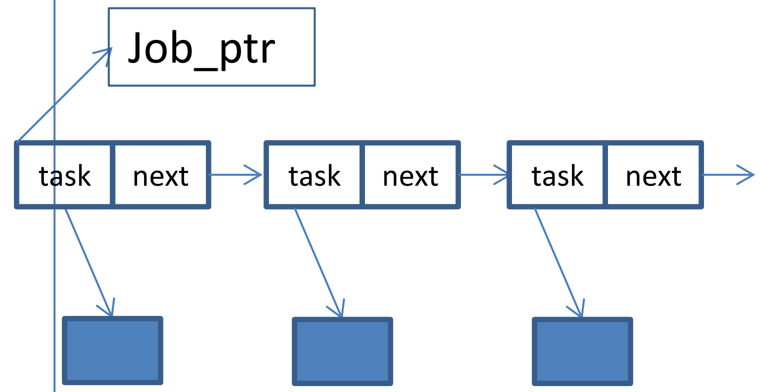
Credits: Zhiliang Xu

```
#include “omp.h”

#pragma omp parallel
{
#pragma omp sections
      {
         #pragma omp section
            v=alpha();
         #pragma omp section
            w=beta();
      }
#pragma omp sections
      {
         #pragma omp section
            x=gamma(v,w);
         #pragma omp section
            y=delta();
      }
      printf(“%g\n”, epsilon(x,y));
}
```

Credits: Zhiliang Xu

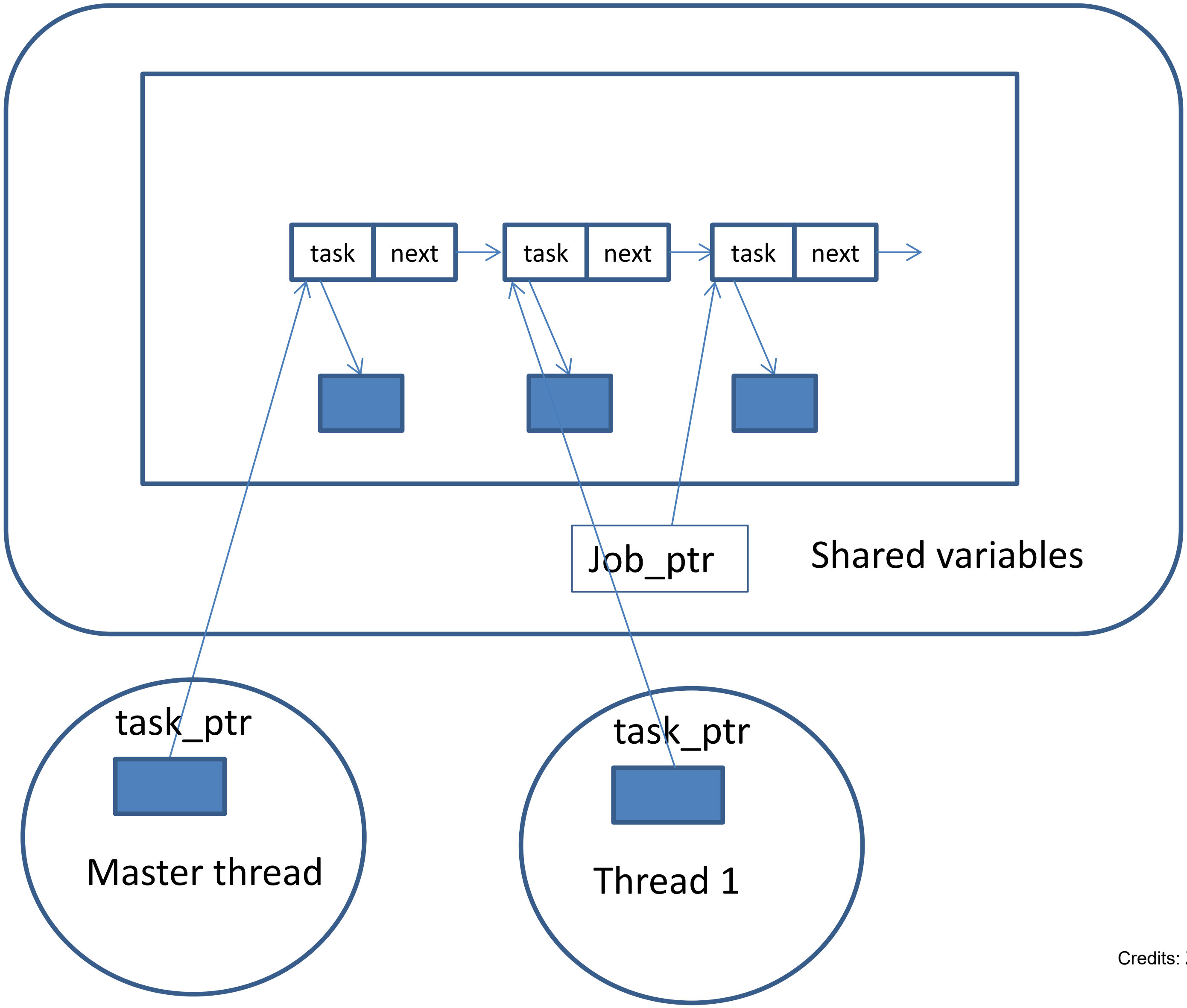
Code Fragment for Manager/Worker Model

```
int main(int argc, char argv[])
{
   struct job_struct  job_ptr;
   struct task_struct  *task_ptr;
   ...
   task_ptr = get_next_task(&job_ptr);
   while(task_ptr != NULL){
       complete_task(task_ptr);
       task_ptr = get_next_task(&job_ptr);
    }
    ...
}

struct task_struct  *get_next_task(struct job_struct  *job_ptr)
{
    struct task_struct  *answer;
    if(job_ptr == NULL) answer = NULL;
    else
     {
        answer = job_ptr->task;
        job_ptr = job_ptr->next;
     }
     return answer;
}
```



Credits: Zhiliang Xu

- Two threads complete the work



Credits: Zhiliang Xu

Tasking

OpenMP 3.0 and Tasks

Tasks allow to parallelize irregular problems

- Unbounded loops
- Recursive algorithms
- Manger/work schemes
- …

A task has

- **Code** to execute
- **Data** environment (It owns its data)
- **Internal control variables**
- An assigned thread that executes the code and the data

Two activities: packaging and execution

- Each encountering thread packages a new instance of a task (code and data)
- Some thread in the team executes the task at some later time

Credits: Zhiliang Xu

Recursive approach to compute Fibonacci

```
int main(int argc,
         char* argv[])
{
   [...]
   fib(input);
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
   int x = fib(n - 1);
   int y = fib(n - 2);
   return x+y;
}
```

- **On the following slides we will discuss three approaches to parallelize this recursive code with Tasking.**

The Task Construct

C/C++

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

Fortran

```
!$omp task [clause]
... structured block ...
!$omp end task
```

- **Each encountering thread/task creates a new Task**
  - → Code and data is being packaged up
  - → Tasks can be nested
    - → Into another Task directive
    - → Into a Worksharing construct
- **Data scoping clauses:**
  - → `shared`(*list*)
  - → `private`(*list*) `firstprivate`(*list*)
  - → `default`(*shared* | *none*)

Tasks in OpenMP: 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

First version parallelized with Tasking (omp-v1)

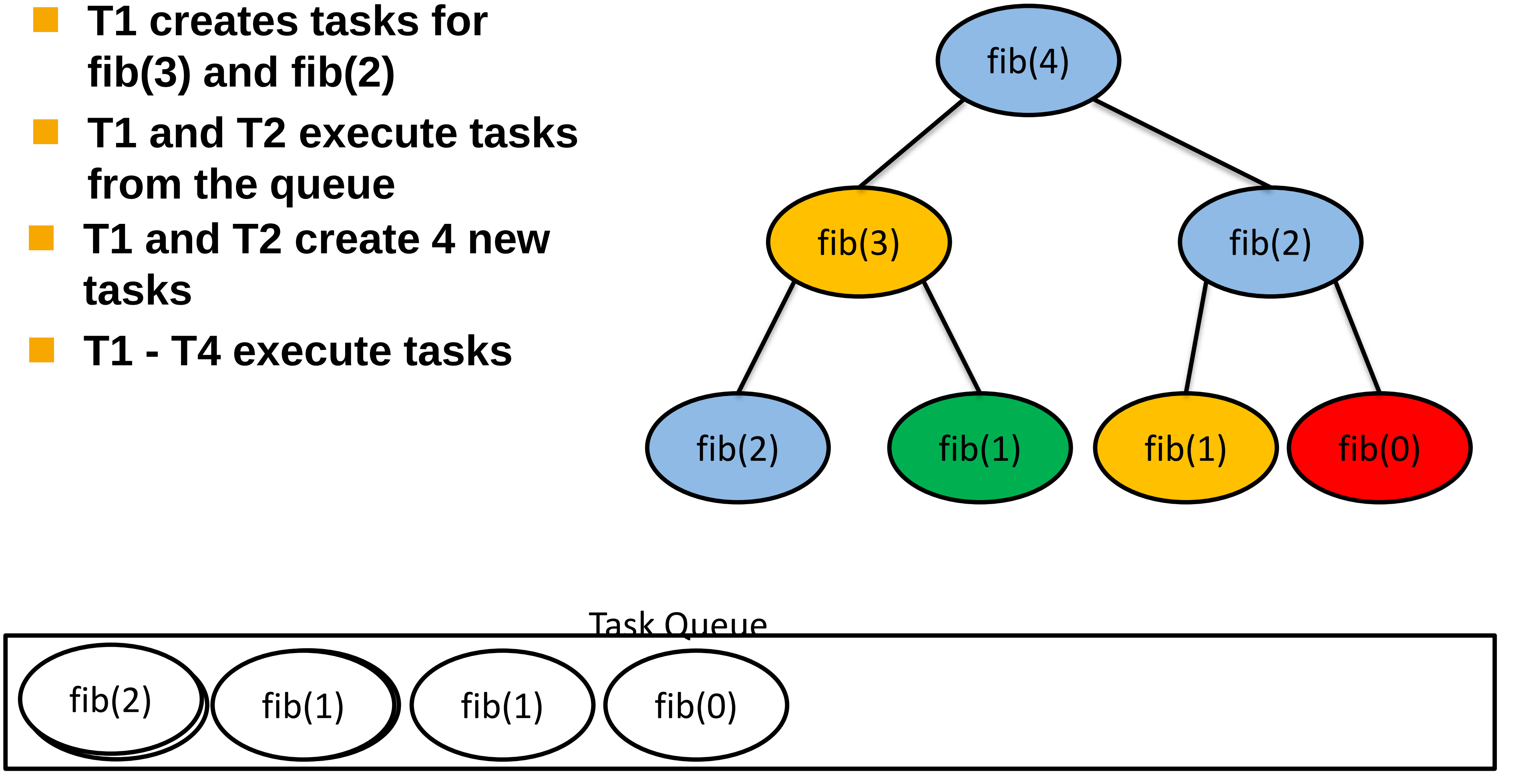
```
int main(int argc,
          char* argv[])
{
   [...]
   #pragma omp parallel
   {
         #pragma omp single
         {
                  fib(input);
         }
   }
   [...]
}
```

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

- **Only one Task / Thread enters `fib()` from `main()`, it is responsable for creating the two initial work tasks**
- **Taskwait is required, as otherwise `x` and `y` would be lost**

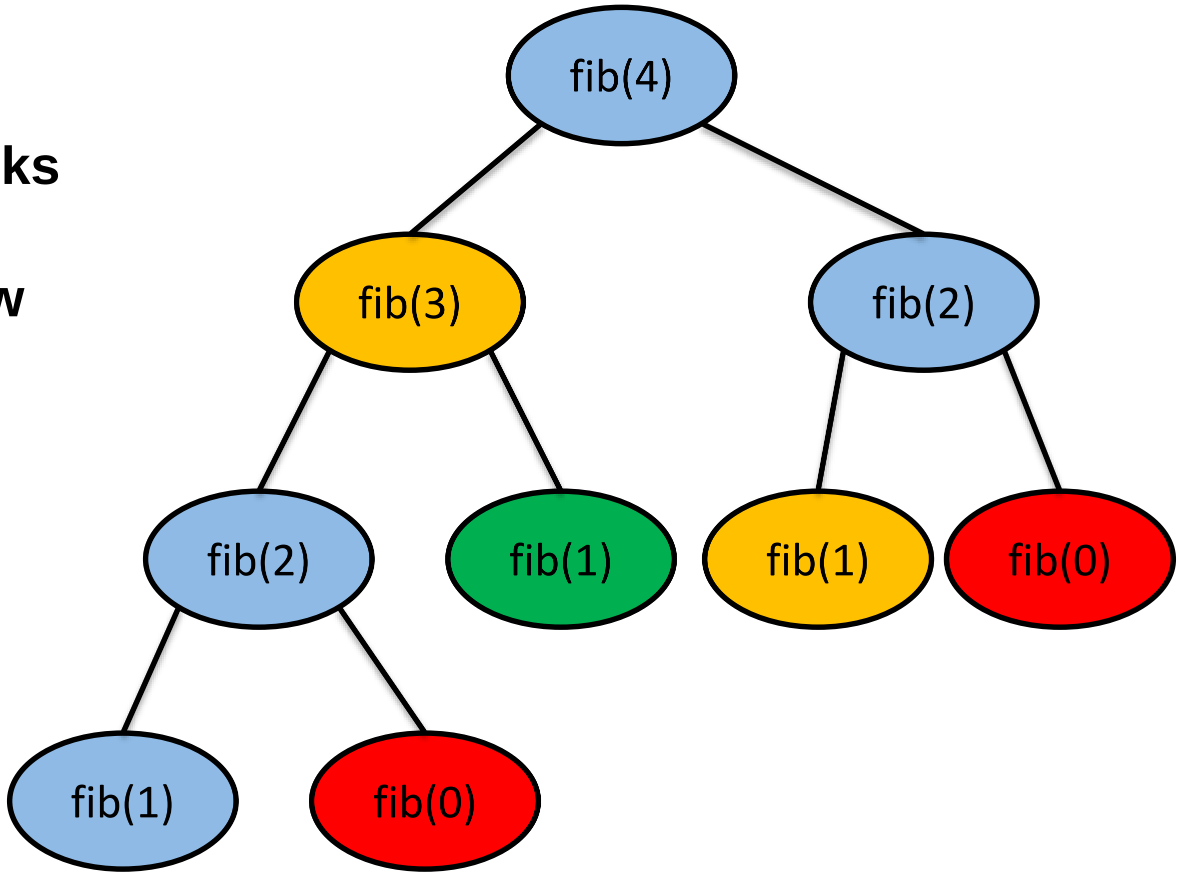
Fibonacci Illustration

- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



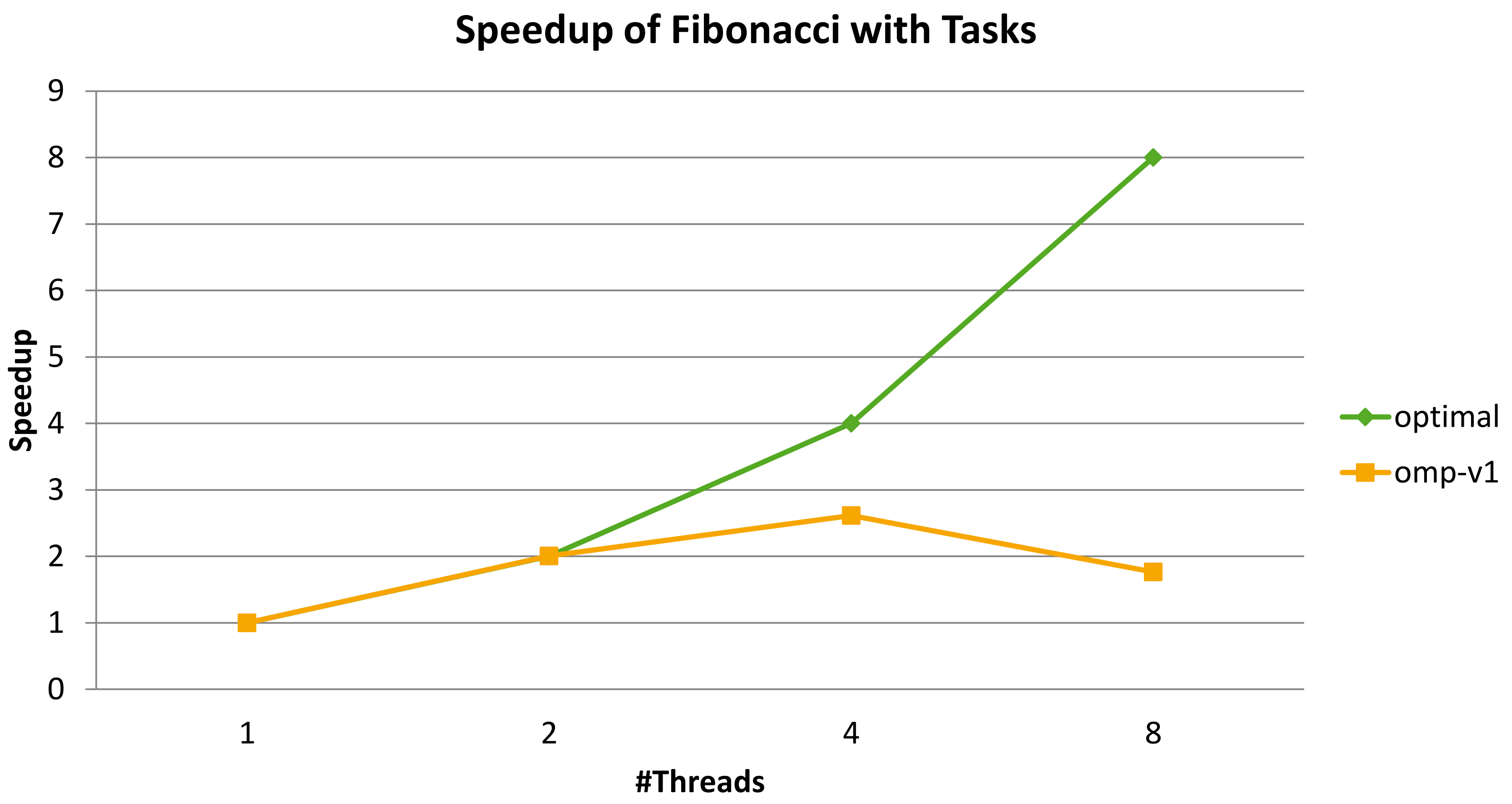
Fibonacci Illustration

- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- …



Scalability measurements (1/3)

- **Overhead of task creation prevents better scalability!**



`if` Clause

- **If the expression of an `if` clause on a task evaluates to `false`**
  - → The encountering task is suspended
  - → The new task is executed immediately
  - → The parent task resumes when the new task finishes
  - → Used for optimization, e.g., avoid creation of small tasks

Improved parallelization with Tasking (omp-v2)

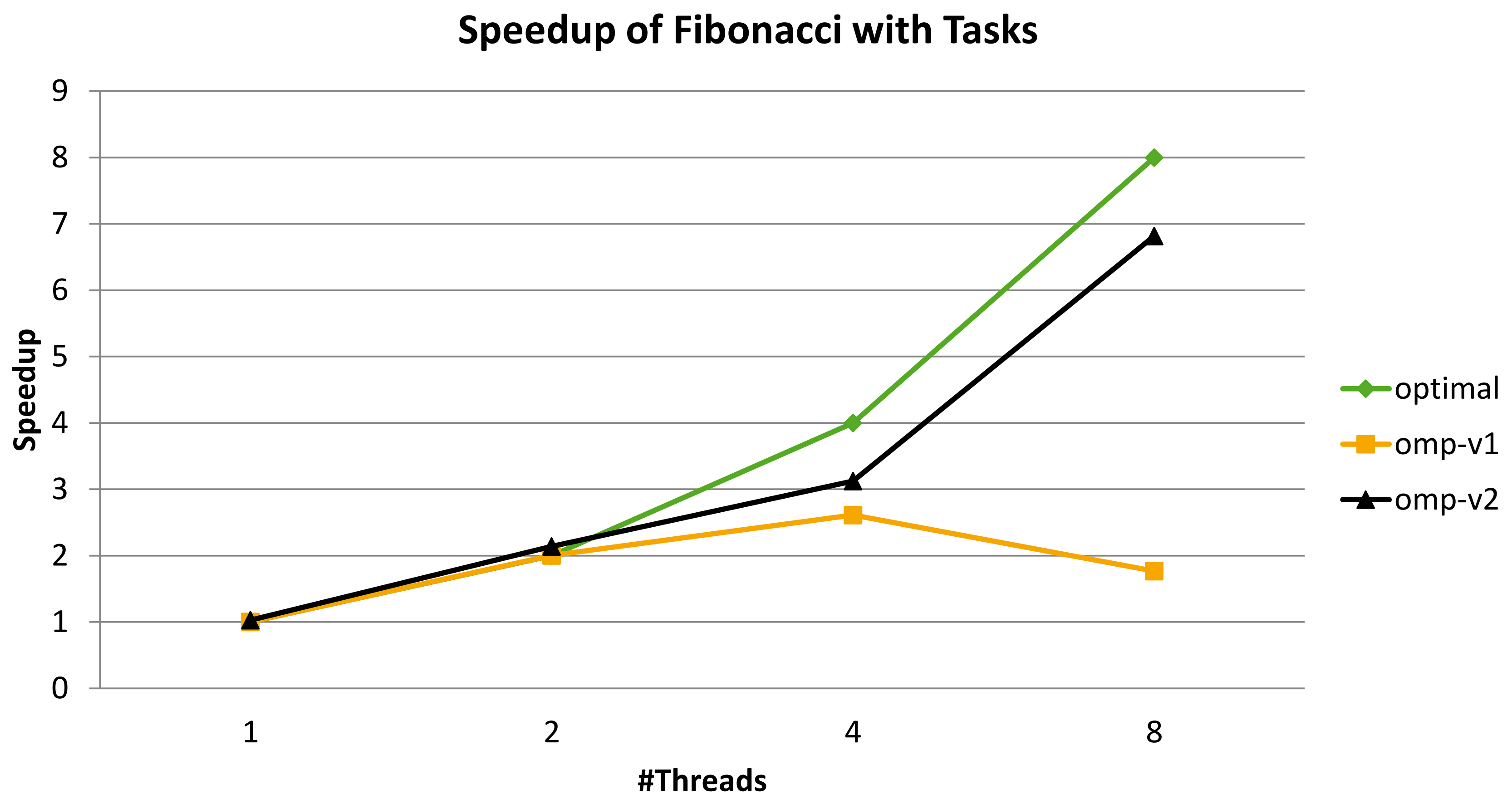
- **Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached**

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

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

Scalability measurements (2/3)

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



Improved parallelization with Tasking (omp-v3)

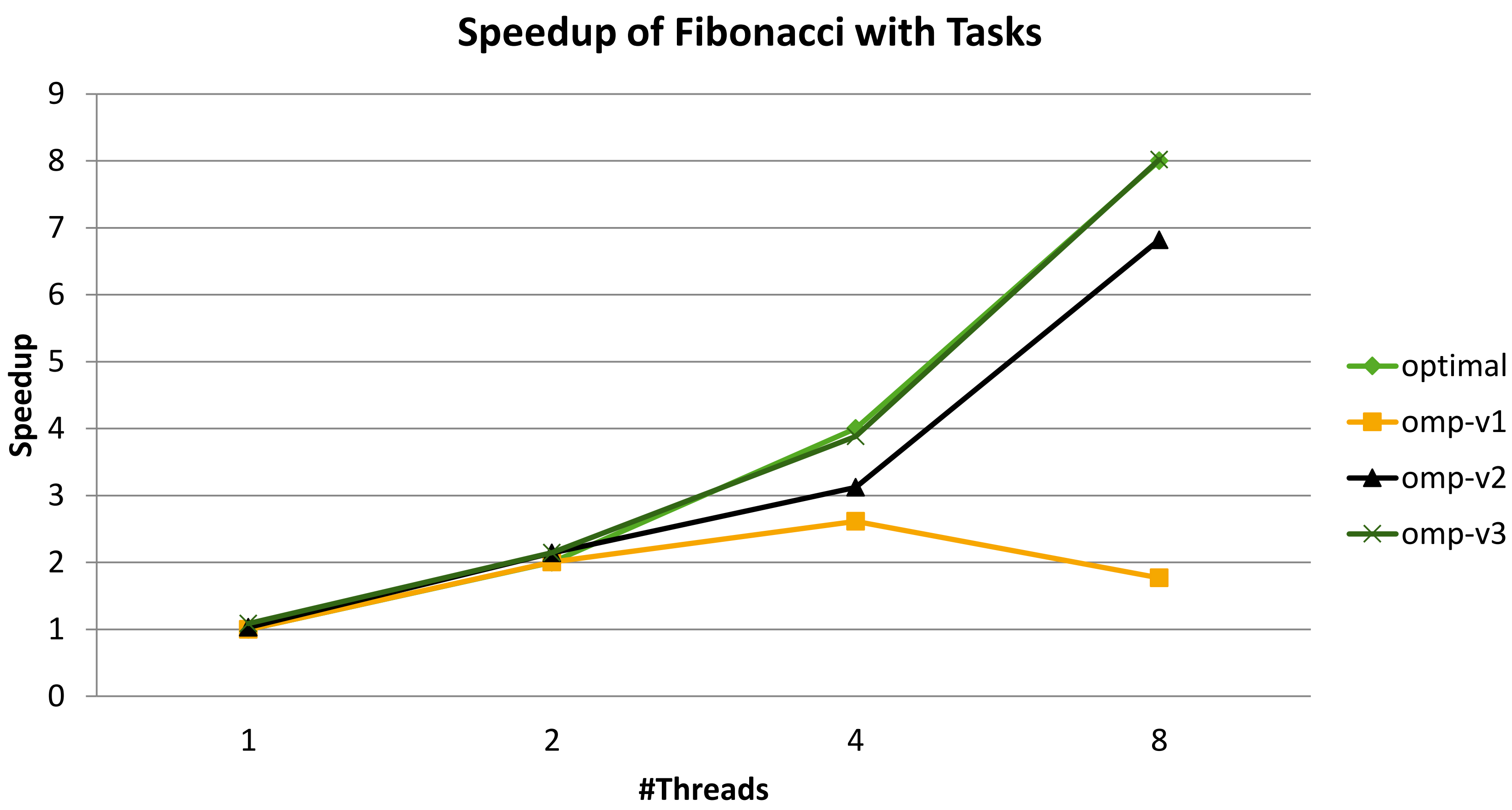
- **Improvement: Skip the OpenMP overhead once a certain `n` is reached (no issue w/ production compilers)**

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

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

Scalability measurements (3/3)

- **Everything ok now ☺**



Data Scoping Example (1/7)

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

               // Scope of a:
               // Scope of b:
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (2/7)

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

               // Scope of a: shared
               // Scope of b:
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (3/7)

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

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c:
                // Scope of d:
                // Scope of e:
} } }
```

Data Scoping Example (4/7)

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

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d:
               // Scope of e:
} } }
```

Data Scoping Example (5/7)

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

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d: firstprivate
               // Scope of e:
} } }
```

Data Scoping Example (6/7)

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

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c: shared
               // Scope of d: firstprivate
               // Scope of e: private
} } }
```

Hint: Use default(none) to be forced to think about every variable if you do not see clear.

Data Scoping Example (7/7)

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

                // Scope of a: shared,         value of a: 1
                // Scope of b: firstprivate,   value of b: 0 / undefined
                // Scope of c: shared,         value of c: 3
                // Scope of d: firstprivate,   value of d: 4
                // Scope of e: private,        value of e: 5
} } }
```

The Barrier and Taskwait Constructs

- **OpenMP `barrier` (implicit or explicit)**
  - → All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

```
C/C++
#pragma omp barrier
```

- **Task barrier: `taskwait`**
  - → Encountering Task suspends until child tasks are complete
    - → Only direct childs, not descendants!

```
C/C++
#pragma omp taskwait
```

Task Synchronization

- **Task Synchronization explained:**

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

np Tasks created here, one for each thread

All Tasks guaranteed to be completed here

1 Task created here

B-Task guaranteed to be completed here

Loop Collapse

- Allows parallelization of perfectly nested loops without using nested parallelism
- Compiler forms a single loop and then parallelizes this

```
{
   …
        #pragma omp parallel for collapse (2)
        for(i=0;i< N; i++)
        {
              for(j=0;j< M; j++)
             {
                 foo(A,i,j);
             }
         }
}
```

Credits: Zhiliang Xu