HPC.NRW

Introduction to OpenMP

Dr. Christian Terboven

Tasking

Dr. Christian Terboven

Introduction to OpenMP

Recursive approach to compute Fibonacci

HPC.NRW

- Fibonacci numbers
  - Form a sequence $\boldsymbol{F_n}$ such that each number is the sum of the two preceding
  - $\boldsymbol{F_0 = 0, F_1 = 1}$
  - $\boldsymbol{F_n = F_{n-1} + F_{n-2}}$ (for $n > 1$)

```
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

HPC.NRW

- Deferring (or not) a unit of work (executable for any member of the team)
  - Always attached to a structured block

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

```fortran
!$omp task [clause[[,] clause]...]
…structured-block…
!$omp end task
```

Where clause:

→ private(list),	Data Environment
→ firstprivate(list),	Data Environment
→ shared(list)	Data Environment
→ default(shared | none)	Data Environment
→ in_reduction(r-id: list) ≥ 5.0	Data Environment
→ untied	

→ if(scalar-expression)	Cutoff Strategies
→ mergeable	Cutoff Strategies
→ final(scalar-expression)	Cutoff Strategies
→ depend(dep-type: list)	Dependencies
→ priority(priority-value)	

Tasks in OpenMP: Data Scoping

- Some rules from *Parallel Regions* apply:
  - Static and Global variables are shared
  - Automatic Storage (local) variables are private
  - Task variables are `firstprivate` unless `shared` in the enclosing context
    - Only `shared` attribute is inherited
    - Exception: Orphaned Task variables are `firstprivate` by default!

First version parallelized with Tasking (omp-v1)

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

```c
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 responsible for creating the two initial work tasks
- Taskwait is required, as otherwise x and y would be lost

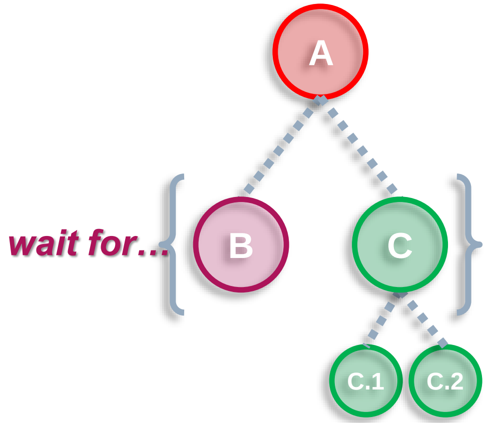
`taskwait` directive

- The taskwait directive (shallow task synchronization)
  - It is a stand-alone directive

```
#pragma omp taskwait
```

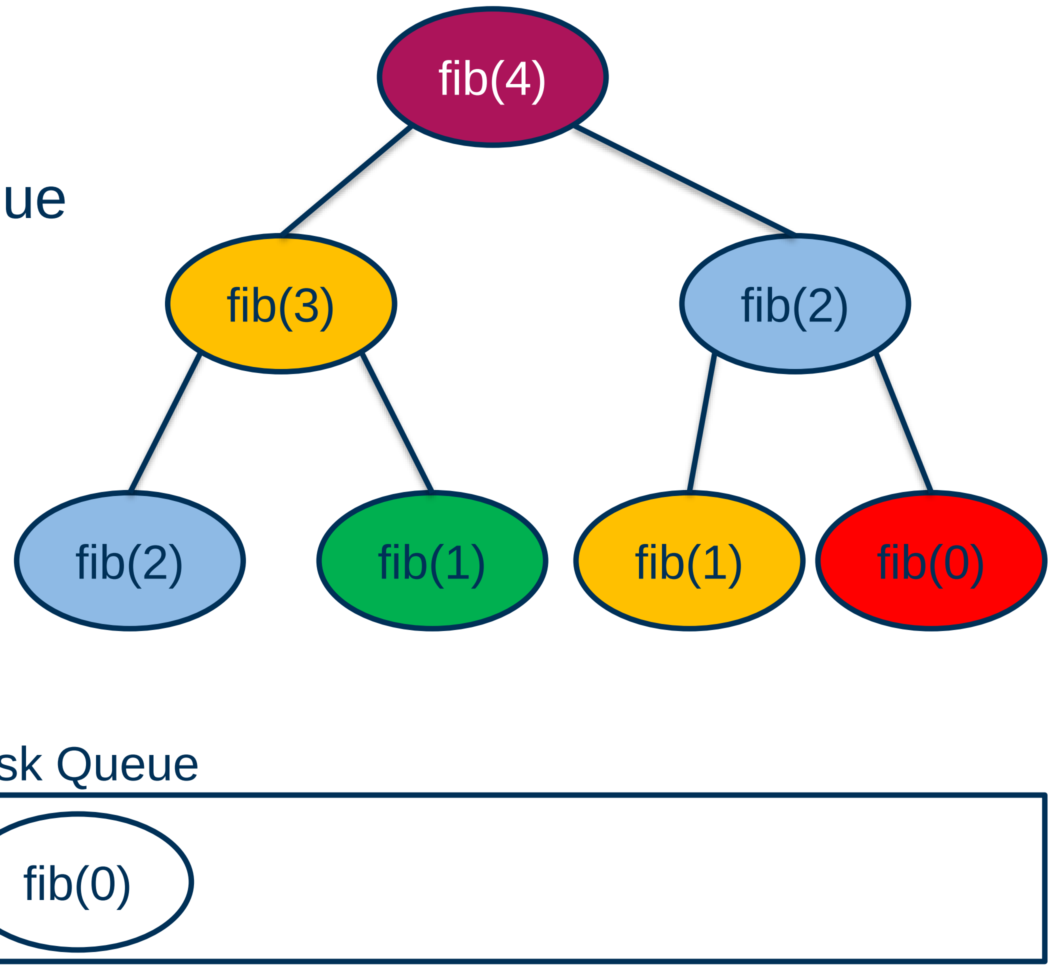
  - wait on the completion of child tasks of the current task; just direct children, not all descendant tasks; includes an implicit task scheduling point (TSP)

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task                    : A
    {
        #pragma omp task                : B
        { … }
        #pragma omp task                : C
        { … #C.1; #C.2; …}
        #pragma omp taskwait
    }
} // implicit barrier will wait for C.x
```



Fibonacci Illustration

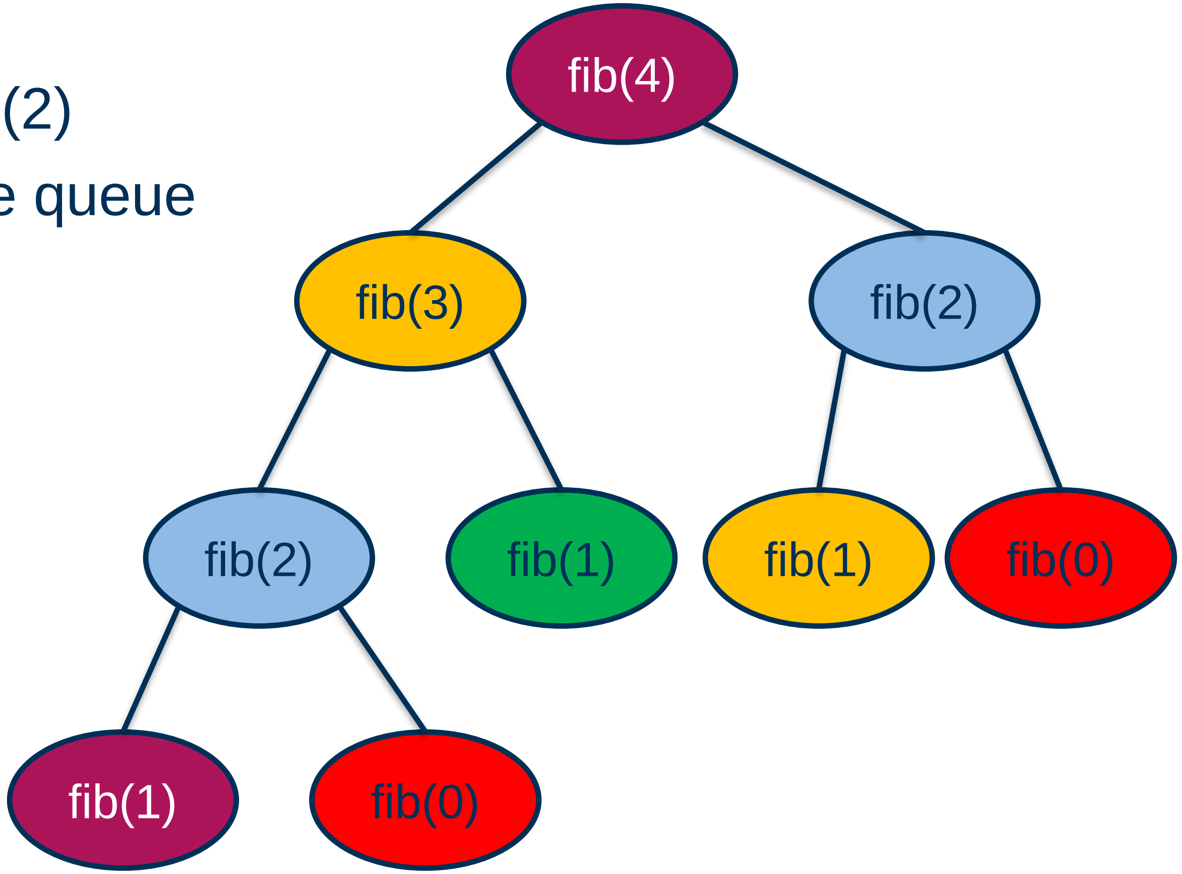
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



Fibonacci Illustration

HPC.NRW

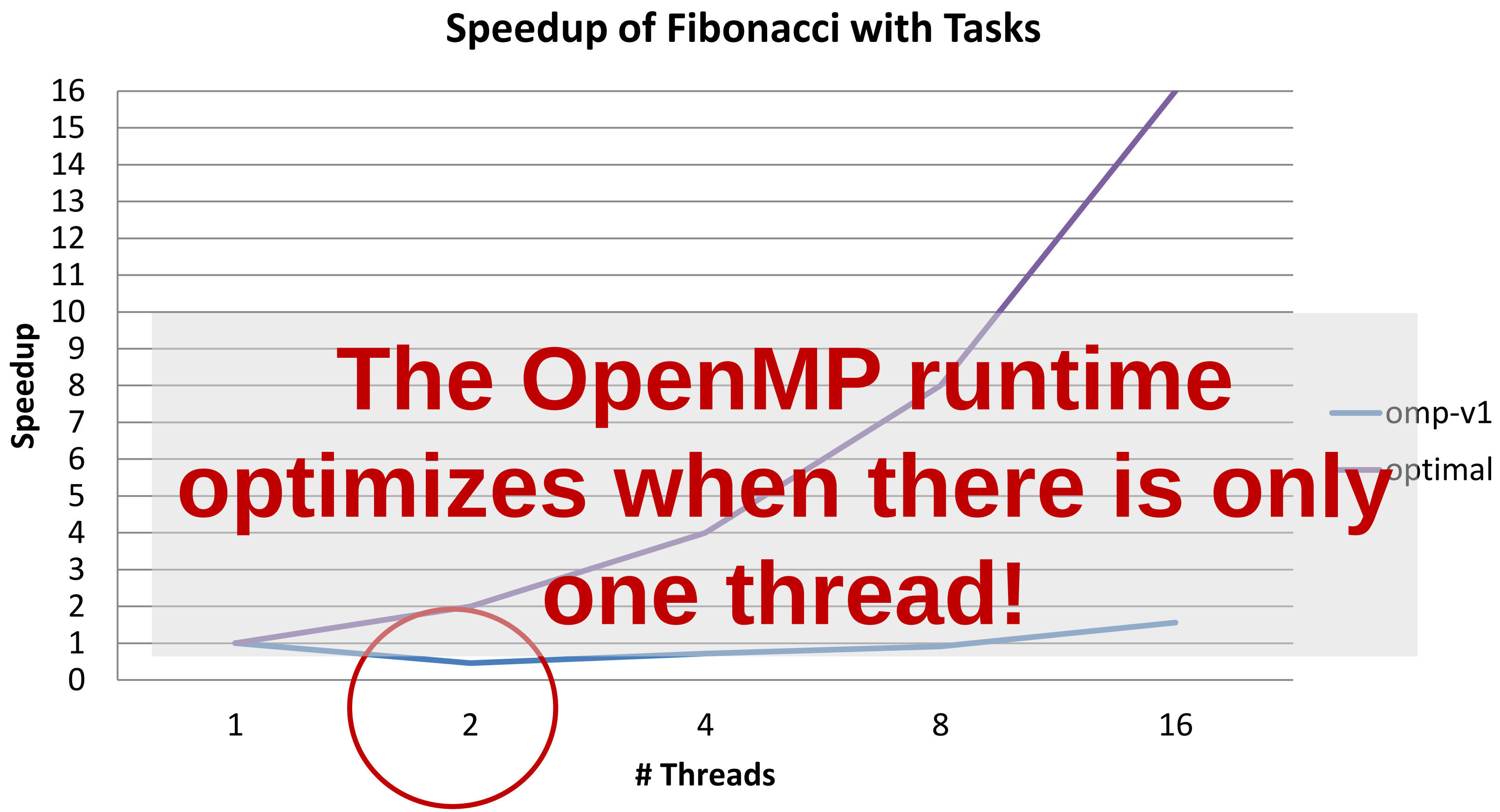
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- …



Scalability measurements (1/3)

HPC.NRW

– Overhead of task creation prevents scalability!



The OpenMP runtime optimizes when there is only one thread!

`if` Clause

HPC.NRW

- The if clause of a task construct
  - allows to optimize task creation/execution
  - reduces parallelism but also reduces the pressure in the runtime’s task pool
  - for “very” fine grain tasks you may need to do your own (*manual*) if

```
#pragma omp task if(expression)
{structured-block}
```

- If the *expression* of the “if” clause evaluates to false
  - the encountering task is suspended
  - the new task is executed immediately
  - the parent task resumes when the task finishes

- This is known as *undeferred* task

Improved parallelization with Tasking (omp-v2)

HPC.NRW

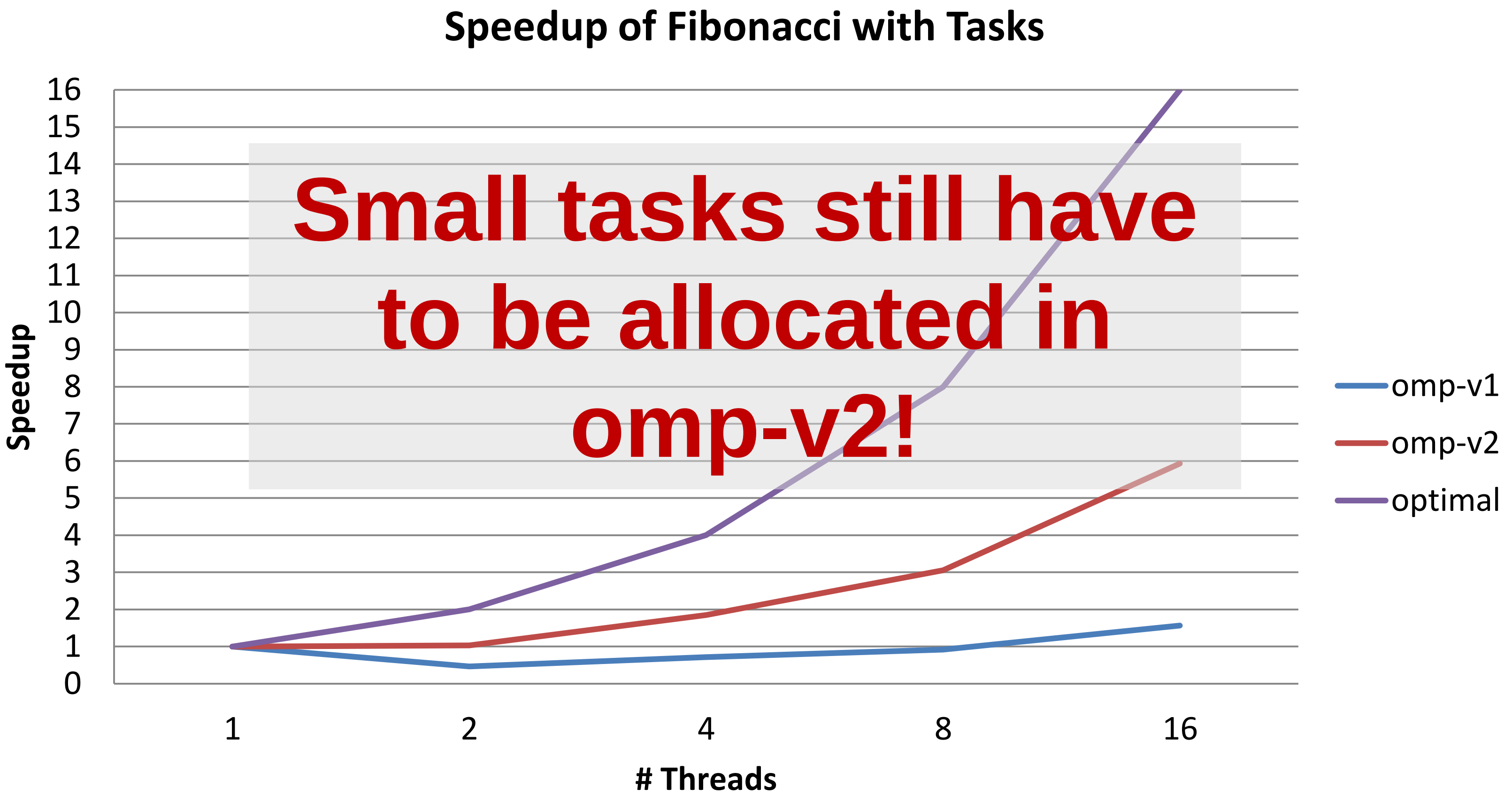
– Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached

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

```c
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;
}
```

– Speedup is better, but still not great

login-t, E5-2650 v4, 2x 12 cores @ 2.20 GHz
Intel Compiler 16.0.2, fib(45) = 1134903170



Improved parallelization with Tasking (omp-v3)

HPC.NRW

– 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);
        }
    }
    [...]
}
```

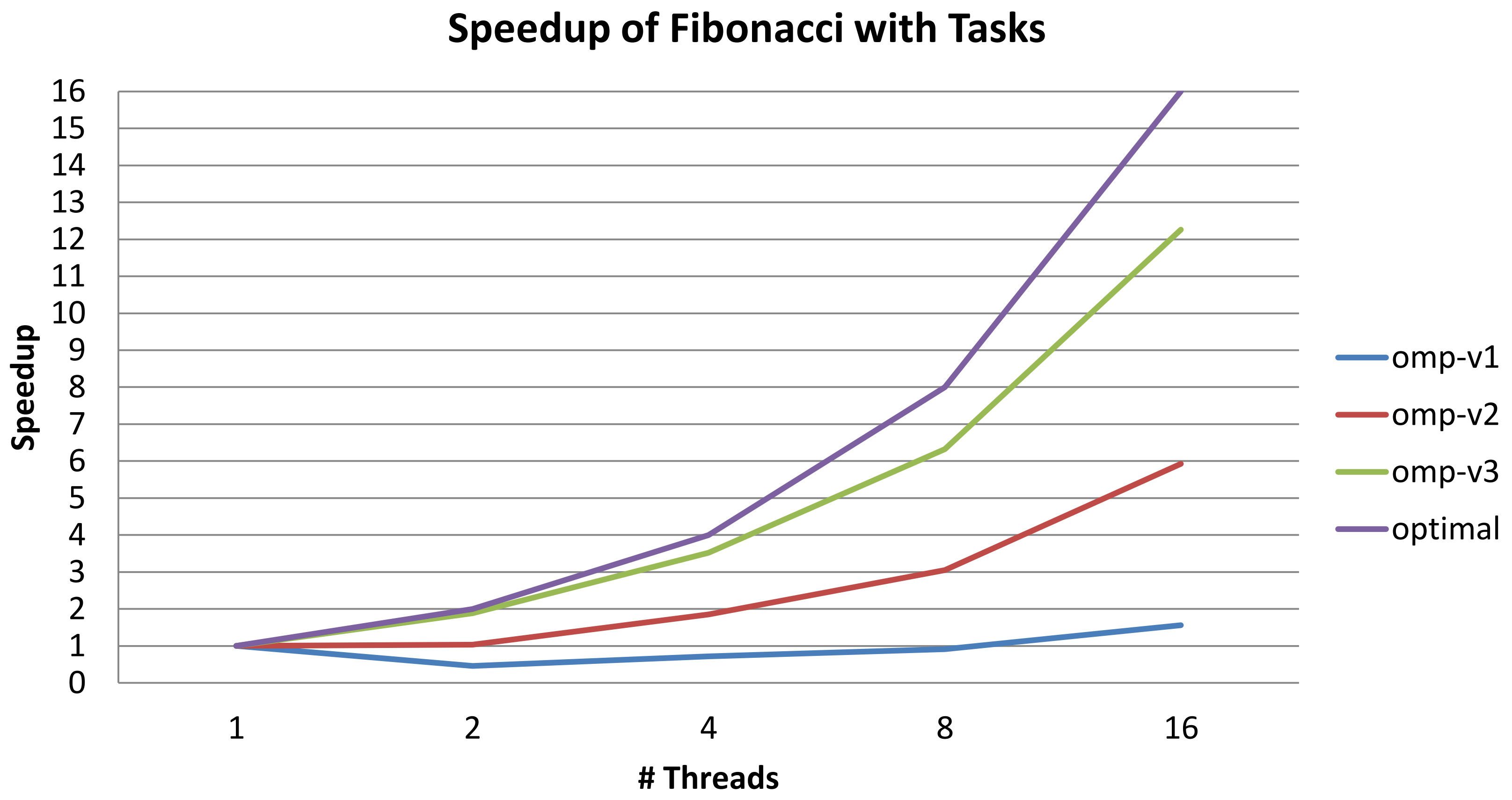
```
int fib(int n)   {
    if (n < 2) return n;
    if (n <= 30)
        return serfib(n);
    int x, y;
    #pragma omp task shared(x)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
        return x+y;
}
```

Scalability measurements (3/3)

HPC.NRW

– Looks promising…

login-t, E5-2650 v4, 2x 12 cores @ 2.20 GHz
Intel Compiler 16.0.2, fib(45) = 1134903170

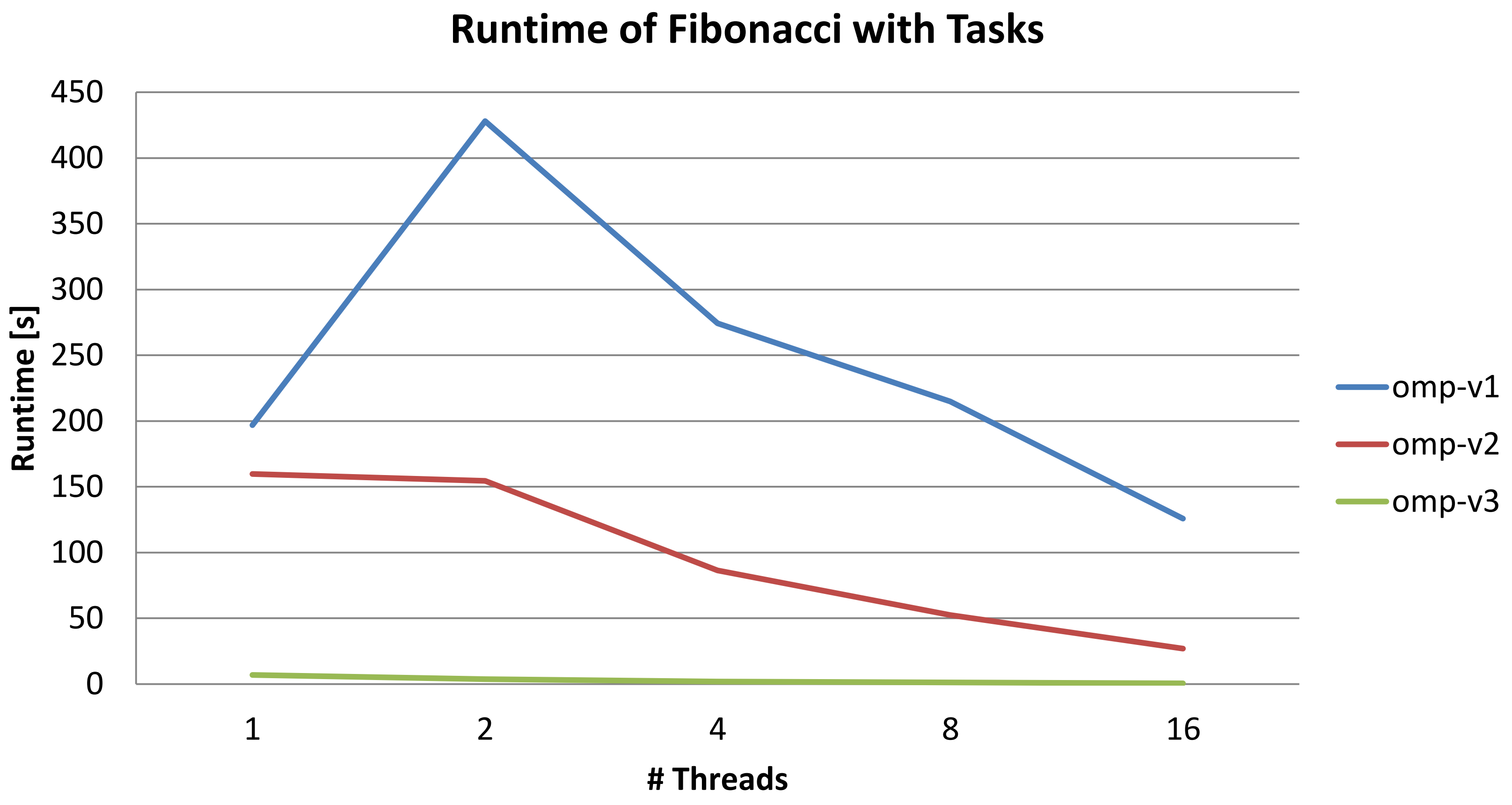


Runtime measurements (1/2)

HPC.NRW

– First two versions were slow because of overhead!

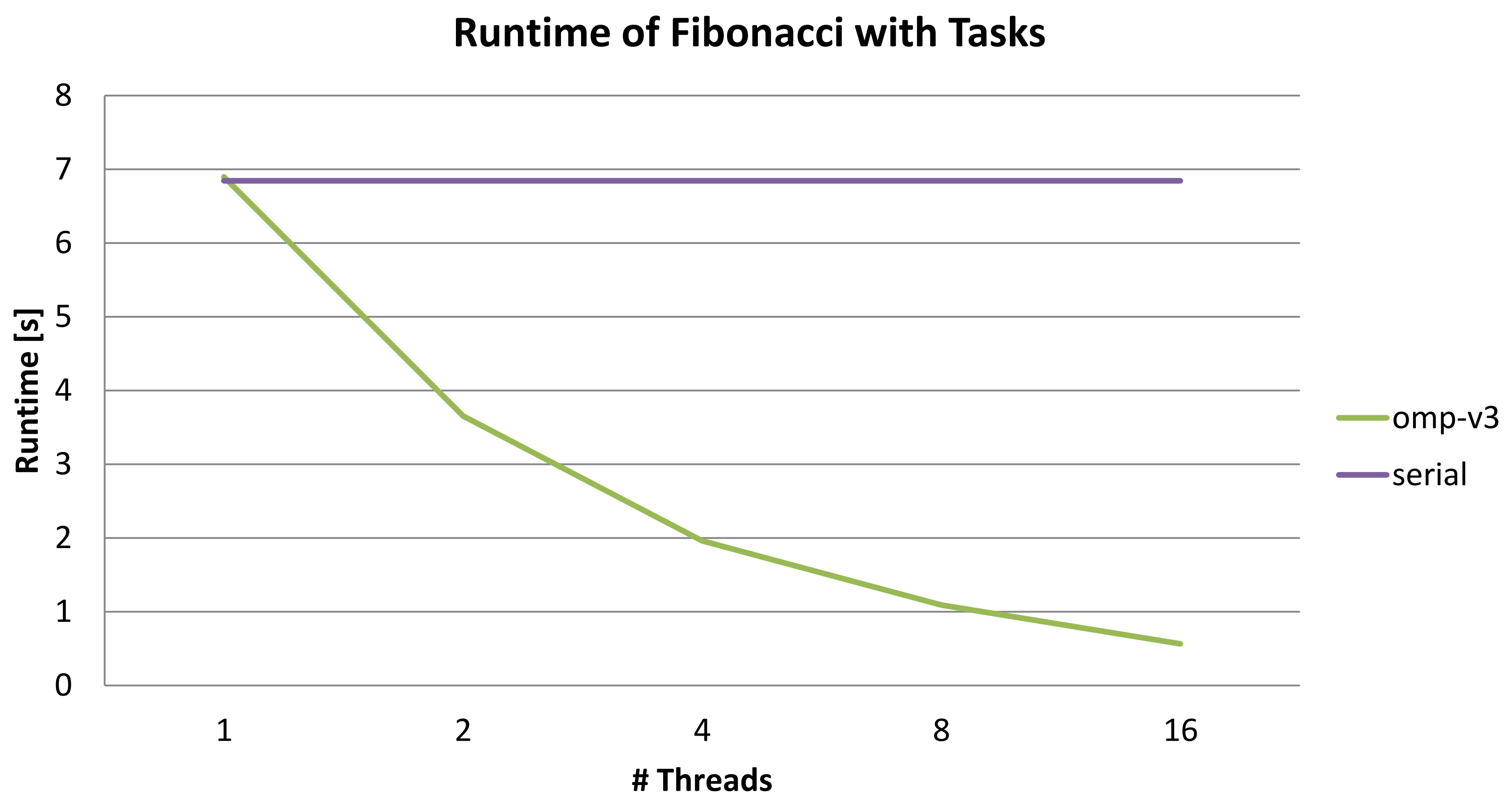
login-t, E5-2650 v4, 2x 12 cores @ 2.20 GHz
Intel Compiler 16.0.2, fib(45) = 1134903170



Runtime measurements (2/2)

- Third version is comparable to serial version w/o OpenMP ☺

login-t, E5-2650 v4, 2x 12 cores @ 2.20 GHz
Intel Compiler 16.0.2, fib(45) = 1134903170



Tasking Overheads

- Typical overheads in task-based programs are:
  - Task creation: populate task data structure, add task to task queue
  - Task execution: retrieve a task from the queue (may including work stealing)

- If tasks become too fine-grained, overhead becomes noticeable
  - Execution spends a higher relative amount of time in the runtime
  - Task execution contributing to runtime becomes significantly smaller

- A rough rule of thumb to avoid (visible) tasking overhead
  - OpenMP tasks: 80-100k instructions executed per task
  - TBB tasks: 30-50k instructions executed per task
  - Other programming models may have another ideal granularity!

Threads vs Tasks

- Threads do not compose well
  - Example: multi-threaded plugin in a multi-threaded application
  - Composition usually leads to oversubscription and load imbalance

- Task models are inherently composable
  - A pool of threads executes all created tasks
  - Tasks from different modules can freely mix

- Task models make complex algorithms easier to parallelize
  - Programmers can think in concurrent pieces of work
  - Mapping of concurrent execution to threads handled elsewhere
  - Task creation can be irregular (e.g., recursion, graph traversal)

Sometimes You’re Better off with Threads…

- Some scenarios are more amenable for traditional threads
  - Granularity too coarse for tasking
  - Isolation of autonomous agents

- Static allocation of parallel work is typically easier with threads
  - Controlling allocation of work to cache hierarchy

- Graphical User Interfaces (event thread + worker threads)

- Request/response processing, e.g.,
  - Web servers
  - Database servers

Questions?