OpenMP Overview

in 30 Minutes

Christian Terboven <terboven@rz.rwth-aachen.de>

06.12.2010 / Aachen, Germany

Stand: 03.12.2010

Version 2.3

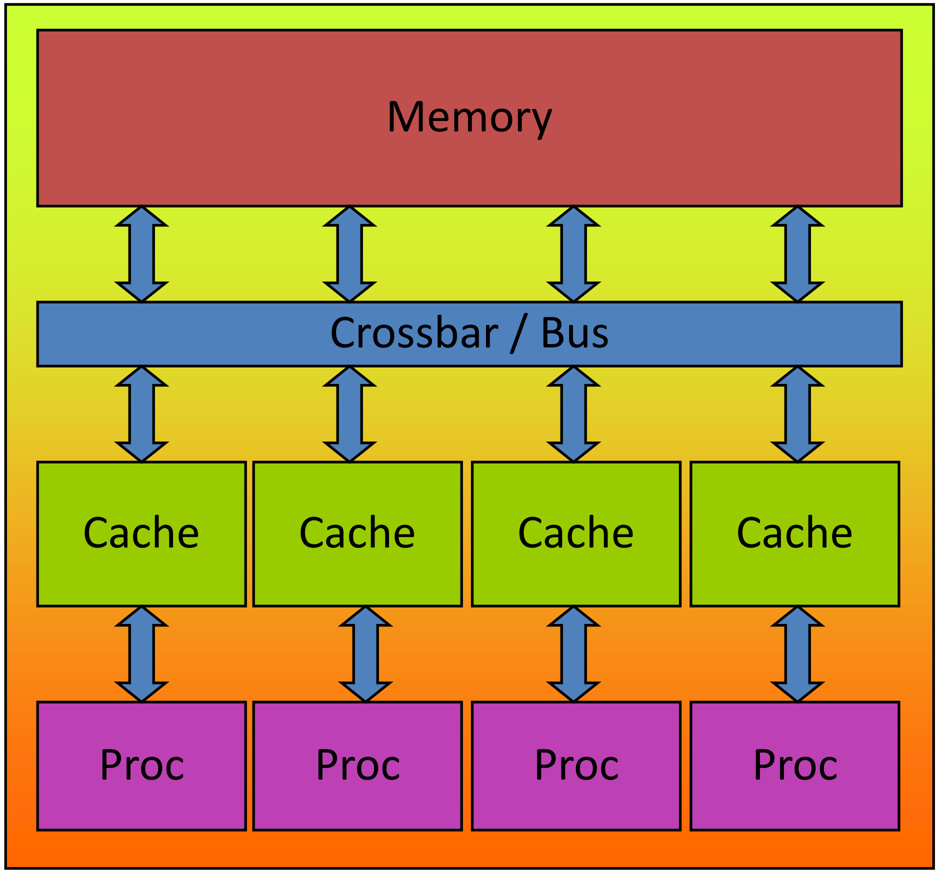
Agenda

- **OpenMP: Parallel Regions, Worksharing, Synchronization**
- **Example: Pi**
- **OpenMP: Tasking**
- **Example: Fibonacci**

OpenMP: Parallel Regions, Worksharing, Synchronization

OpenMP Overview (1/2)

- **OpenMP: Shared-Memory Parallel Programming Model.**

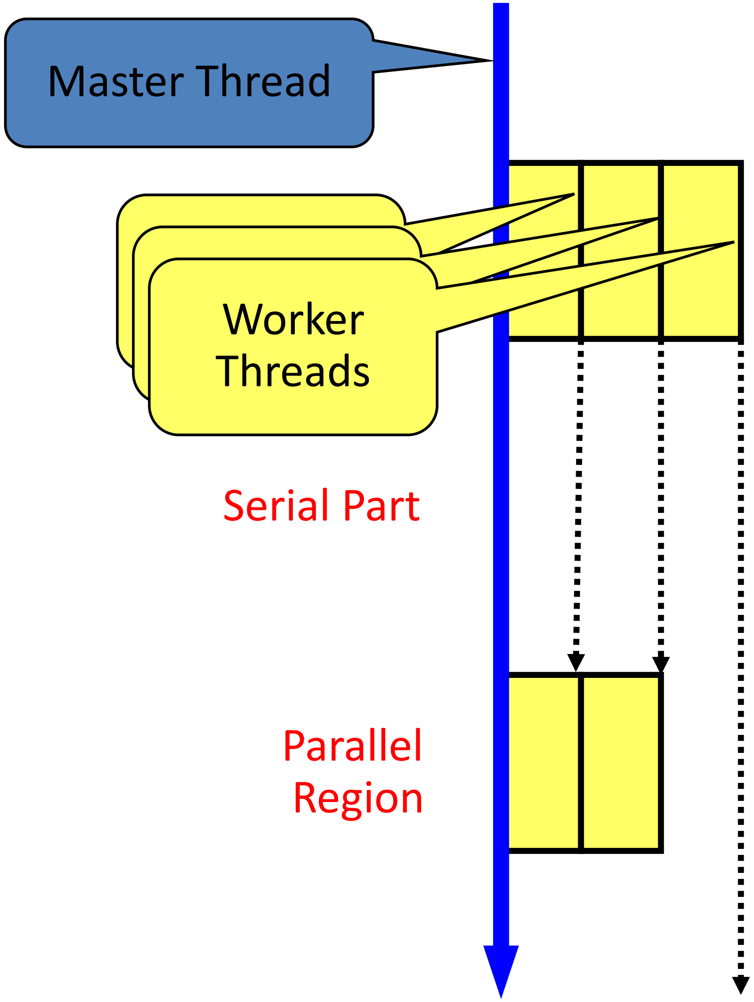


All processors/cores access a shared main memory.

Real architectures are more complex, as we will see later / as you just have seen.

Parallelization in OpenMP employs threads.

OpenMP Overview (2/2)



- **OpenMP programs start with just one thread: The *Master*.**
- ***Worker* threads are spawned at *Parallel Regions*. Together with the Master they form a *Team*.**
- **In between Parallel Regions the Worker threads are put to sleep.**
- **Concept: *Fork-Join*.**
- **Allows for an incremental parallelization!**

- **The parallelism has to be expressed explicitly.**

C/C++

```
#pragma omp parallel
{
    ...
    structured block
    ...
}
```

- ***Structured Block***
  - Exactly one entry point at the top
  - Exactly one exit point at the bottom
  - Branching in or out is not allowed
  - Terminating the program is allowed (abort)

- **Specification of number of threads:**
  - Environment variable:

    `OMP_NUM_THREADS=…`

  - Or: Via `num_threads` clause:

```
#pragma omp parallel \
        num_threads(num) {…}
```

Worksharing (1/2)

- **If only the *parallel* construct is used, each thread executes the Structured Block.**
- **Program Speedup: *Worksharing***
- **OpenMP‘s most common Worksharing construct: *for***

C/C++

```
int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   a[i] = b[i] + c[i];
}
```

  - Distribution of loop iterations over all threads in a Team.
  - Scheduling of the distribution can be influenced.

- **Loops often account for most of the program runtime!**

Worksharing (2/2)

- ***for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:**
  - `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.
  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.
  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.
- **Default on most implementations is `schedule(static)`.**

Scoping (1/2)

- **Challenge of Shared-Memory parallelization: Managing the Data Environment.**
- ***Scoping* in OpenMP: Dividing variables in *shared* and *private*:**
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - Default is *shared*
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for each thread
    - *firstprivate*: Initialization with Master‘s value
    - *lastprivate*: Value of last loop iteration is written back to Master
  - Static variables are *shared*

Scoping (2/2)

- **Global / static variables can be privatized with the *threadprivate* directive**
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword `__thread` (GNU extension)

C/C++

```
static int i;
#pragma omp threadprivate(i)
```

Synchronization (1/4)

- **Can all loops be parallelized with `for`-constructs? No!**
  - Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent. BUT: This test alone is not sufficient:

C/C++

```
int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   s = s + a[i];
}
```

- ***Data Race*: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).**

Synchronization (2/4)

Pseudo-Code
Here: 4 Threads

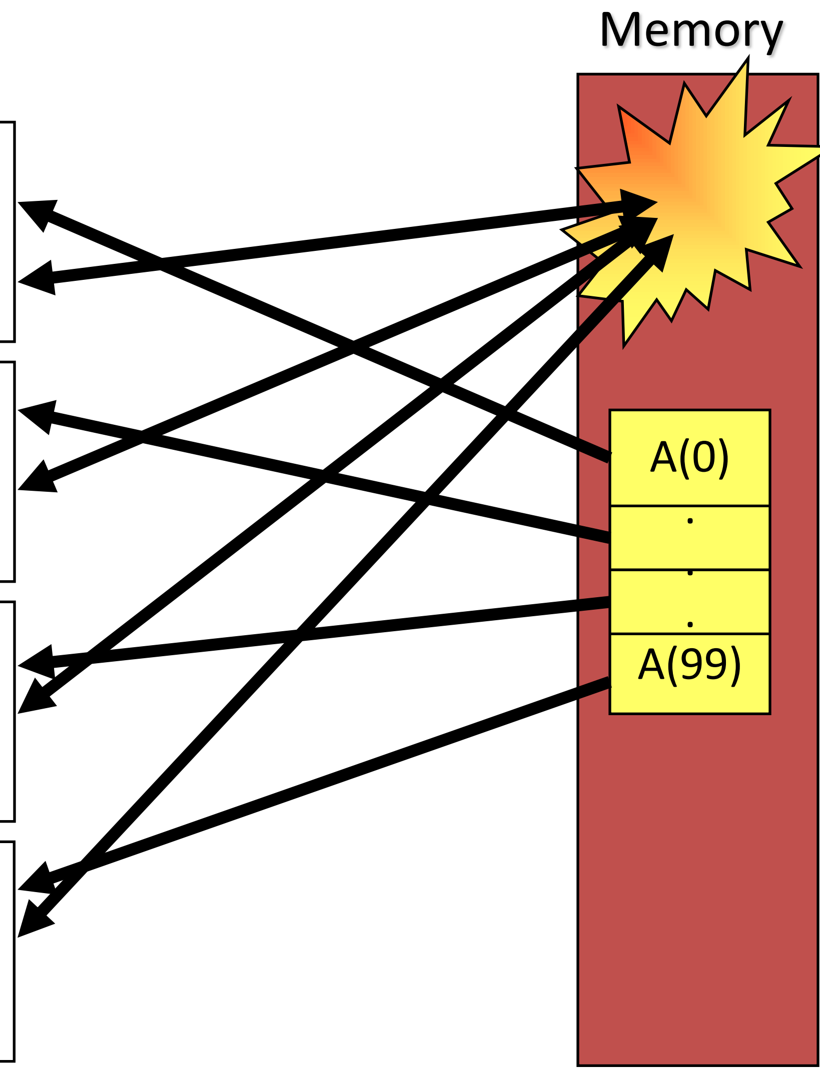
```
do i = 0, 99
    s = s + a(i)
end do
```

```
do i = 0, 24
    s = s + a(i)
end do
```

```
do i = 25, 49
    s = s + a(i)
end do
```

```
do i = 50, 74
    s = s + a(i)
end do
```

```
do i = 75, 99
    s = s + a(i)
end do
```



Synchronization (3/4)

- A *Critical Region* is executed by all threads, but by only one thread simultaneously (*Mutual Exclusion*).

C/C++

```
#pragma omp critical (name)
{
    ... structured block ...
}
```

- Do you think this solution scales well?

C/C++

```
int i;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{

#pragma omp critical
    {   s = s + a[i];   }
}
```

Synchronization (4/4): Reductions

- **In a *reduction*-operation the operator is applied to all variables in the list. The variables have to be *shared*.**
  - `reduction(operator:list)`
  - The result is provided in the associated reduction variable

C/C++

```
#pragma omp parallel for reduction(+:s)
for(i = 0; i < 99; i++)
{
   s = s + a[i];
}
```

  - Possible reduction operators: `*, -, &, |, &&, ||, ^`

Runtime Library

- **C and C++:**
  - If OpenMP is enabled during compilation, the preprocessor symbol `_OPENMP` is defined. To use the OpenMP runtime library, the header `omp.h` has to be included.
  - `omp_set_num_threads(int)`: The specified number of threads will be used for the parallel region encountered next.
  - `int omp_get_num_threads`: Returns the number of threads in the current team.
  - `int omp_get_thread_num()`: Returns the number of the calling thread in the team, the Master has always the id 0.
- **Additional functions are available, e.g. to provide locking functionality.**

Example: Pi

- Simple example: calculate Pi by integration

```
double f(double x) {
   return (double)4.0 / ((double)1.0 + (x*x));
}

void computePi() {
   double h = (double)1.0 / (double)iNumIntervals;
   double sum = 0, x;


   for (int i = 1; i <= iNumIntervals; i++) {
      x = h * ((double)i - (double)0.5);
      sum += f(x);
   }

   myPi = h * sum;
}
```

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

OpenMP: Tasking

How to parallelize a While-loop?

- **How would you parallelize this code?**

```
typedef list<double> dList;
dList myList;
/* fill myList with tons of items */

dList::iterator it = myList.begin();
while (it != myList.end())
{
     *it = processListItem(*it);
     it++;
}
```

- **One possibility: Create a fixed-sized array containing all list items and a parallel loop running over this array**
  Concept: Inspector / Executor

How to parallelize a While-loop!

- **Or: Use Tasking in OpenMP 3.0**

```
#pragma omp parallel
{
#pragma omp single
{
   dList::iterator it = myList.begin();
   while (it != myList.end())
   {
#pragma omp task
      {
        *it = processListItem(*it);
      }
        it++;
   }
}
}
```

- **All while-loop iterations are independent from each other!**

The task directive

C/C++

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

- **Each encountering thread 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*)

Task synchronization (1/2)

- **At OpenMP `barrier` (implicit or explicit)**
  - All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit
- **Task barrier: `taskwait`**
  - Encountering Task suspends until child tasks are complete
    - Only direct childs, not descendants!

C/C++

```
#pragma omp taskwait
```

Task synchronization (2/2)

- **Simple example of Task synchronization in OpenMP 3.0:**

```
#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

Tasks in OpenMP: Data Scoping

- **Some rules from *Parallel Regions* apply:**
  - Static and Global variables are shared
  - Automatic Storage (local) variables are private
- **If no `default` clause is given:**
  - Orphaned Task variables are `firstprivate` by default!
  - Non-Orphaned Task variables inherit the `shared` attribute!

  → Variables are `firstprivate` unless `shared` in the enclosing context
- **So far no verification tool is available to check Tasking programs for correctness!**

Example: Fibonacci

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.**

First version parallelized with Tasking (omp-v1)

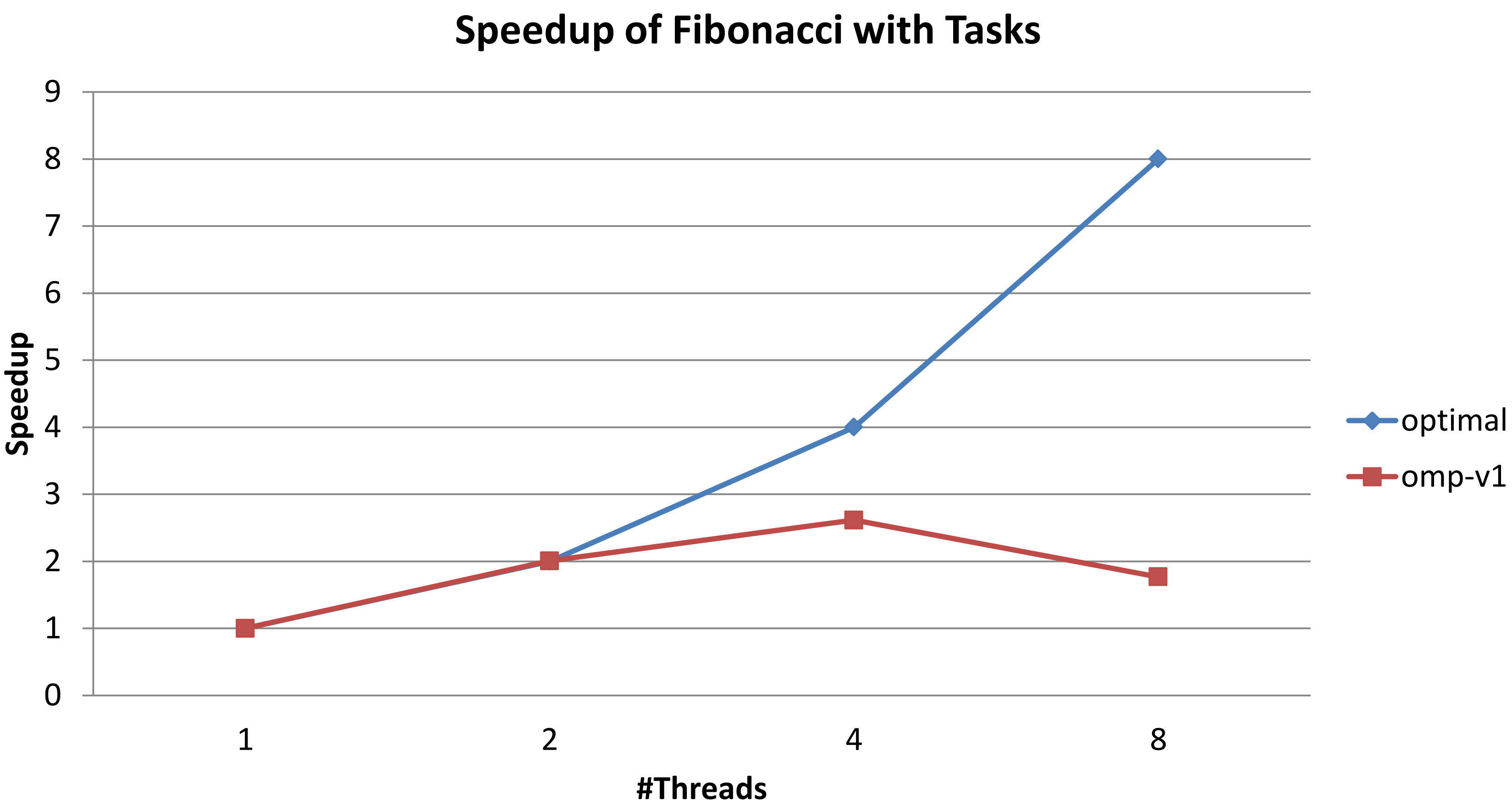
```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
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**

Scalability measurements (1/3)

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



Improved parallelization with Tasking (omp-v2)

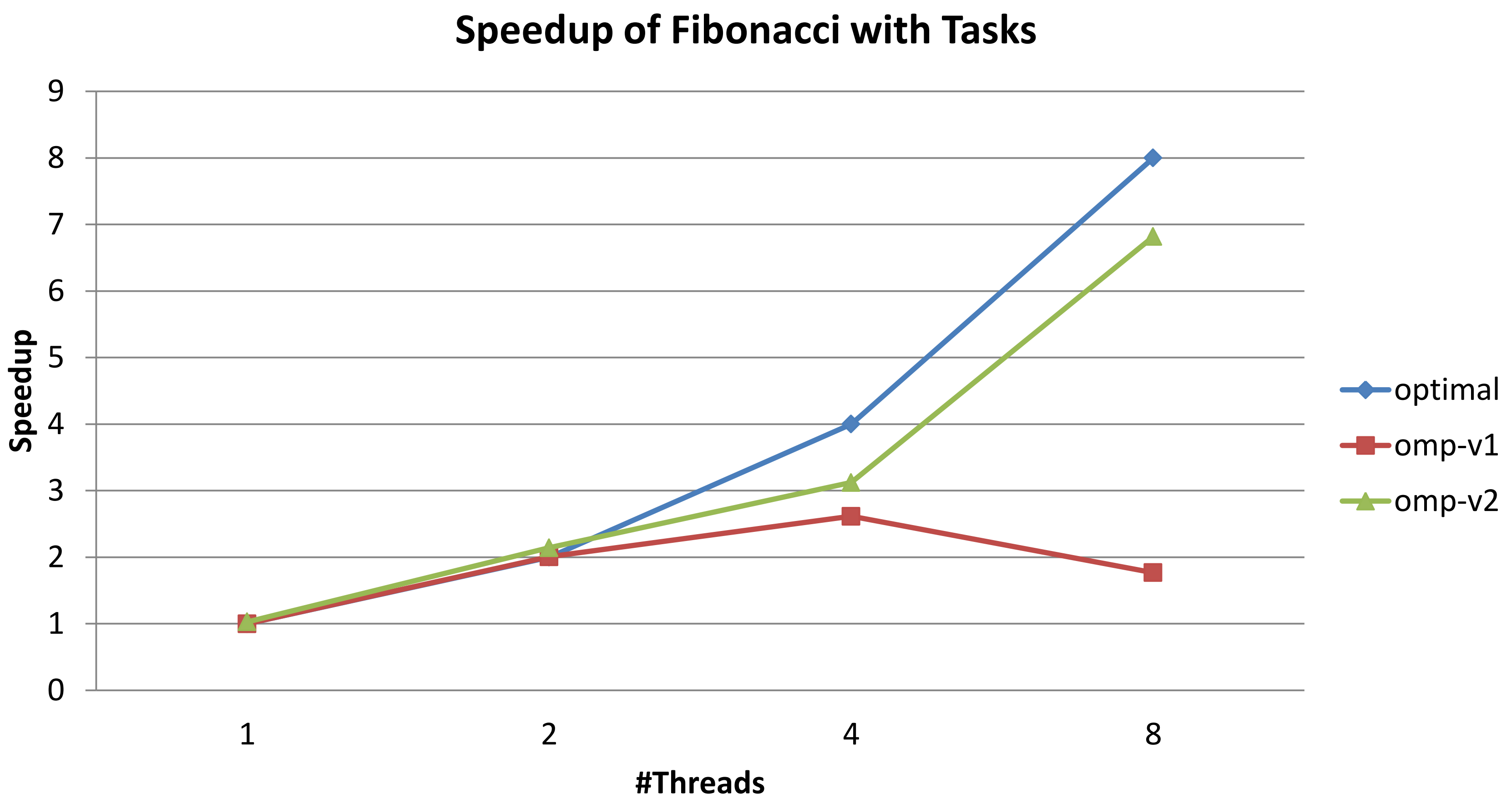
- **Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached**

```
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)

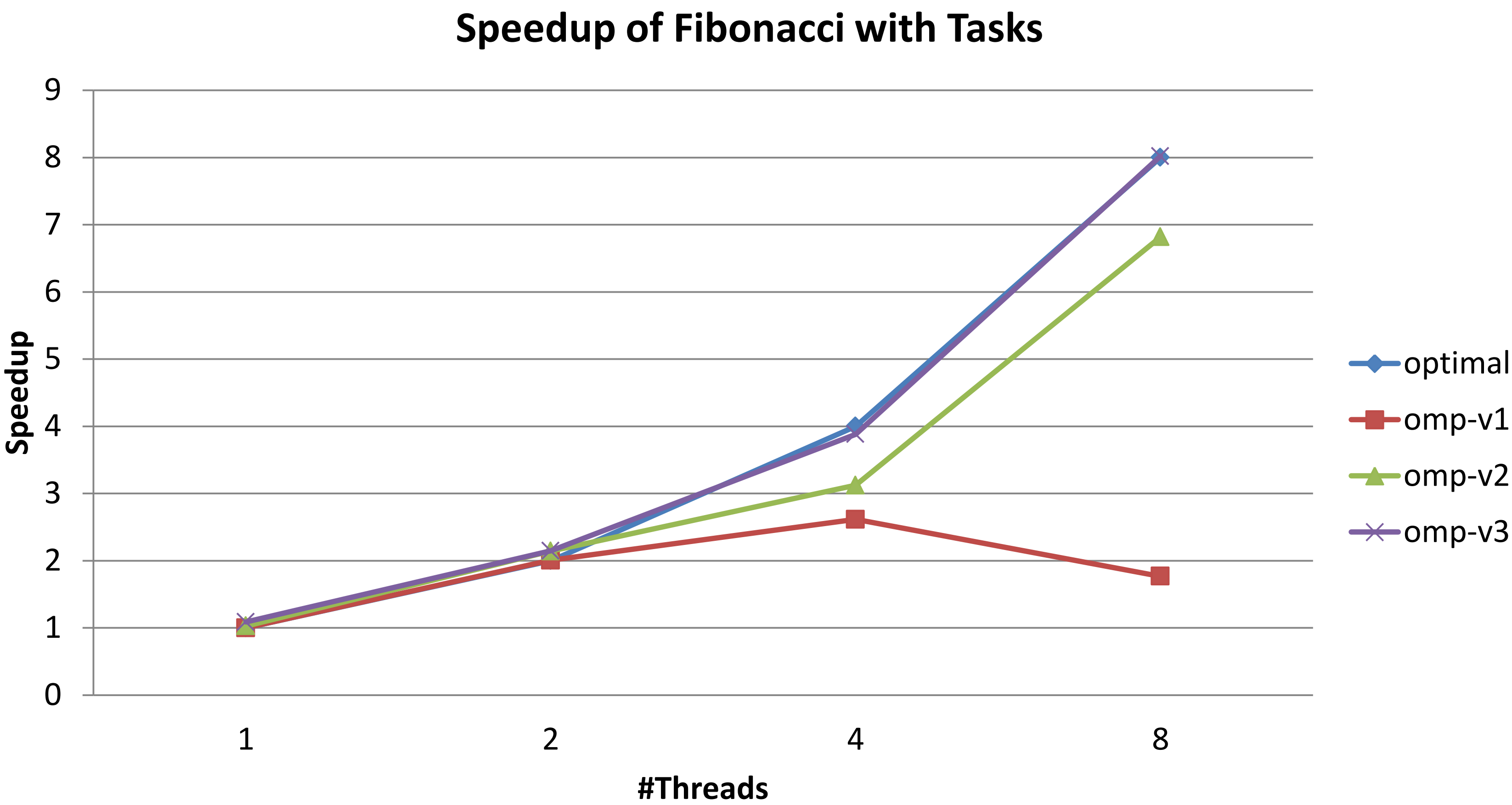
- **Improvement: Skip the OpenMP overhead once a certain `n` is reached (no issue w/ production compilers)**

```
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 ☺**



The End

Thank you for your attention.