

# Introduction to OpenMP

Christian Terboven

IT Center, RWTH Aachen University  
[terboven@itc.rwth-aachen.de](mailto:terboven@itc.rwth-aachen.de)

- **De-facto standard for Shared-Memory Parallelization.**
- **1997: OpenMP 1.0 for FORTRAN**
- **1998: OpenMP 1.0 for C and C++**
- **1999: OpenMP 1.1 for FORTRAN (errata)**
- **2000: OpenMP 2.0 for FORTRAN**
- **2002: OpenMP 2.0 for C and C++**
- **2005: OpenMP 2.5 now includes both programming languages.**
- **05/2008: OpenMP 3.0**
- **07/2011: OpenMP 3.1**
- **07/2013: OpenMP 4.0**
- **11/2015: OpenMP 4.5**



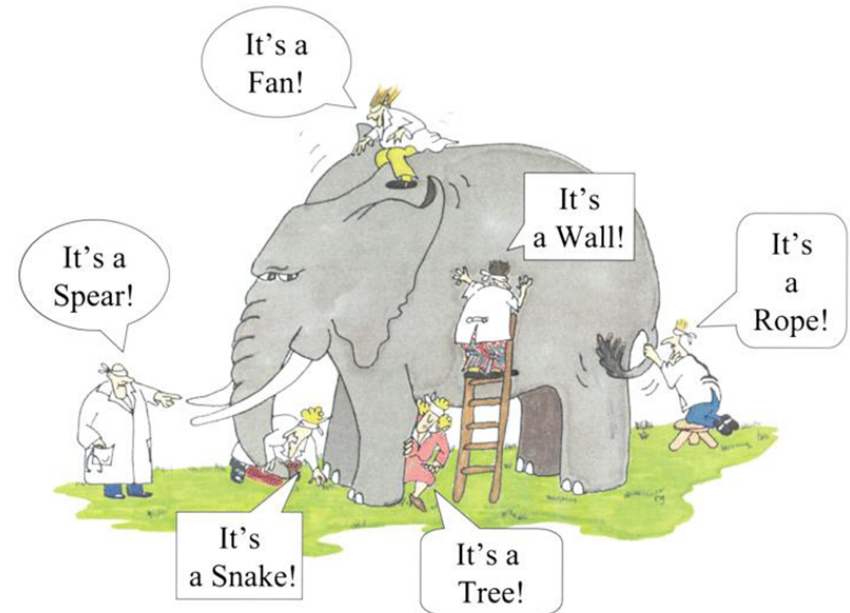
<http://www.OpenMP.org>

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.

# What is OpenMP?

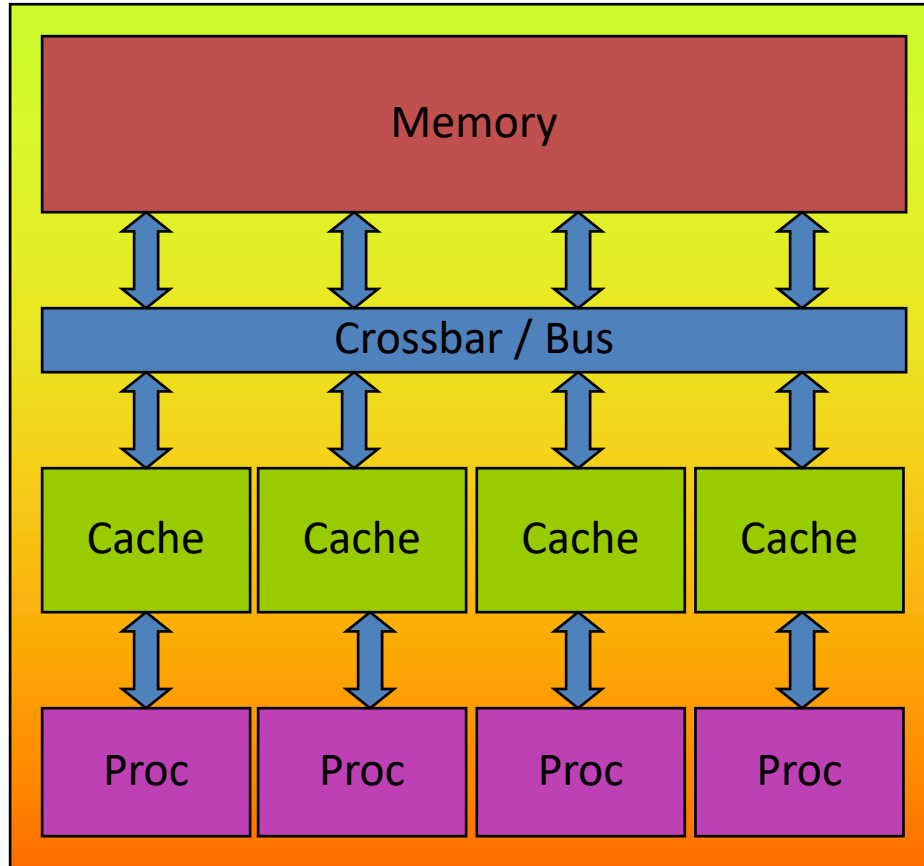


- De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran
- Compiler Directives, Runtime routines and Environment variables



# OpenMP Overview & Parallel Region

## ■ OpenMP: Shared-Memory Parallel Programming Model.

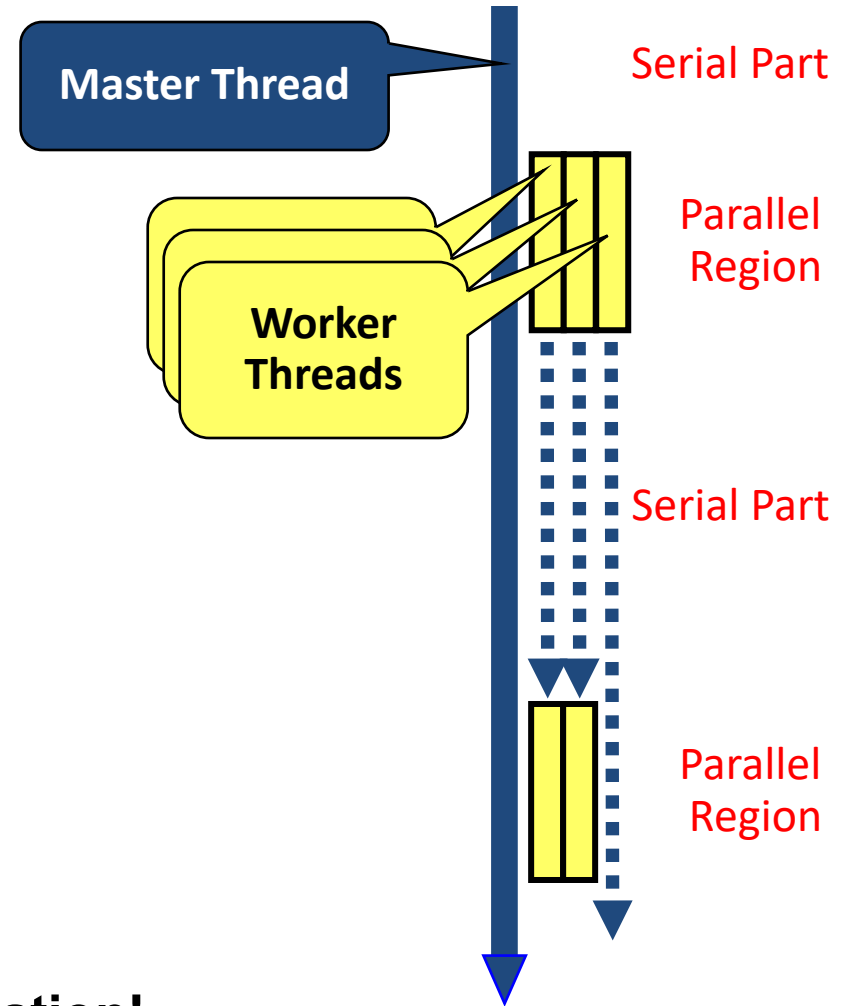


**All processors/cores access a shared main memory.**

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

**Parallelization in OpenMP employs multiple threads.**

- OpenMP programs start with just one thread: The *Master*.
- *Worker* threads are spawned at *Parallel Regions*, together with the Master they form the *Team* of threads.
- In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.
- Concept: *Fork-Join*.
- Allows for an incremental parallelization!



## ■ The parallelism has to be expressed explicitly.

C/C++

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

Fortran

```
!$omp parallel
...
    structured block
...
!$omp end parallel
```

## ■ **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 / exit)

## ■ **Specification of number of threads:**

- ▶ Environment variable:  
OMP\_NUM\_THREADS=...
- ▶ Or: Via `num_threads` clause:  
add `num_threads(num)` to the  
parallel construct

# Hello OpenMP World



# Hello orphaned OpenMP World

- From within a shell, global setting of the number of threads:

```
export OMP_NUM_THREADS=4  
./program
```

- From within a shell, one-time setting of the number of threads:

```
OMP_NUM_THREADS=4 ./program
```

# For Worksharing Construct

- 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 for  
for (i = 0; i < 100; i++)  
{  
    a[i] = b[i] + c[i];  
}
```

Fortran

```
INTEGER :: i  
!$omp do  
DO i = 0, 99  
    a[i] = b[i] + c[i]  
END DO
```

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

- Loops often account for most of a program's runtime!

Pseudo-Code  
Here: 4 Threads

Serial

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

Thread 1

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

Thread 2

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

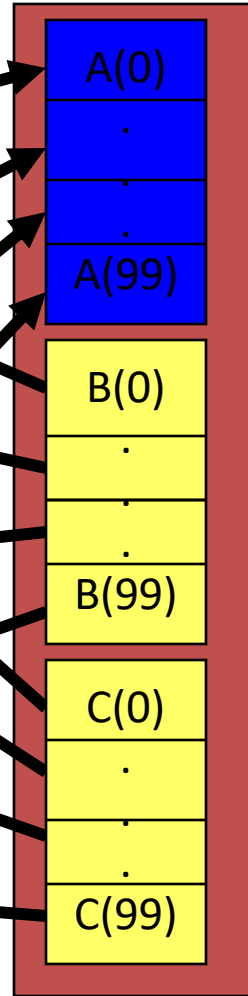
Thread 3

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

Thread 4

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

Memory



# Vector Addition

- **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)`.**

## ■ 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, int s = 0;

#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).



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

# Data Scoping

- **Managing the Data Environment is the challenge of OpenMP.**
- **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
  - General default is *shared* for Parallel Region.
  - 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*

## ■ 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)
```

Fortran

```
SAVE INTEGER :: i  
!$omp threadprivate(i)
```

## ■ 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)
```

Fortran

```
SAVE INTEGER :: i  
!$omp threadprivate(i)
```

# The Barrier Construct

## ■ OpenMP `barrier` (implicit or explicit)

→ Threads wait until all threads of the current *Team* have reached the barrier

C/C++

```
#pragma omp barrier
```

## ■ All worksharing constructs contain an implicit barrier at the end

# Back to our bad scaling example

C/C++

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



# It's your turn: Make It Scale!



```
#pragma omp parallel
```

```
{
```

```
#pragma omp for
```

```
for (i = 0; i < 99; i++)
```

```
{
```

```
    s = s + a[i];
```

```
}
```

```
} // end parallel
```

```
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
```

- 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++

```
int i, s = 0;

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

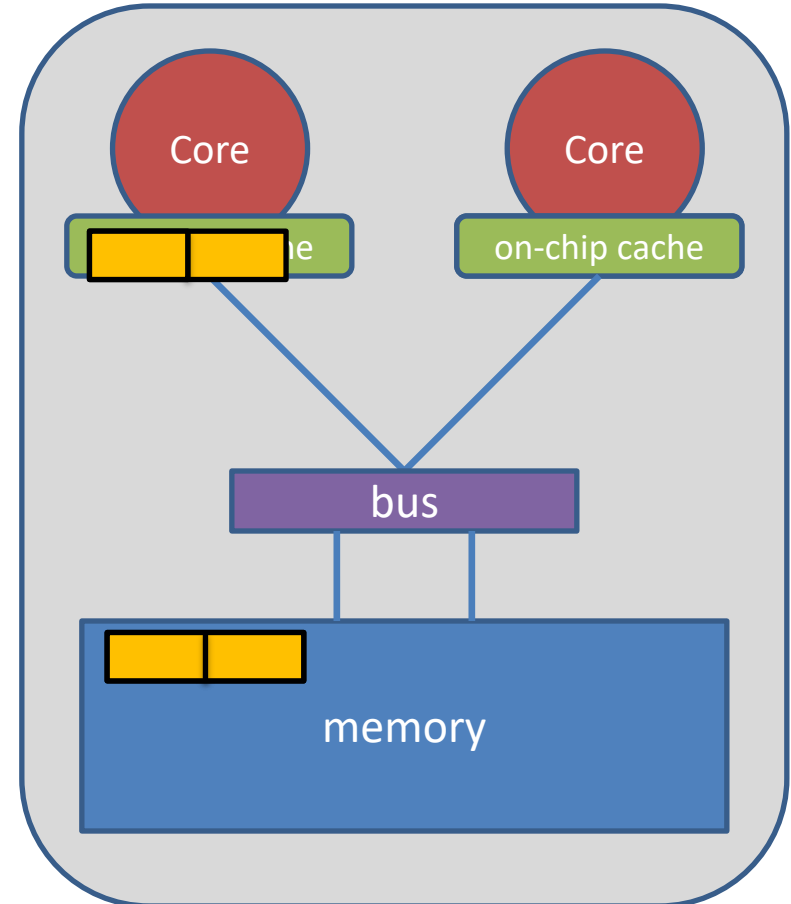
→ Possible reduction operators with initialization value:

`+` (0), `*` (1), `-` (0), `&` (~0), `|` (0), `&&` (1), `||` (0),

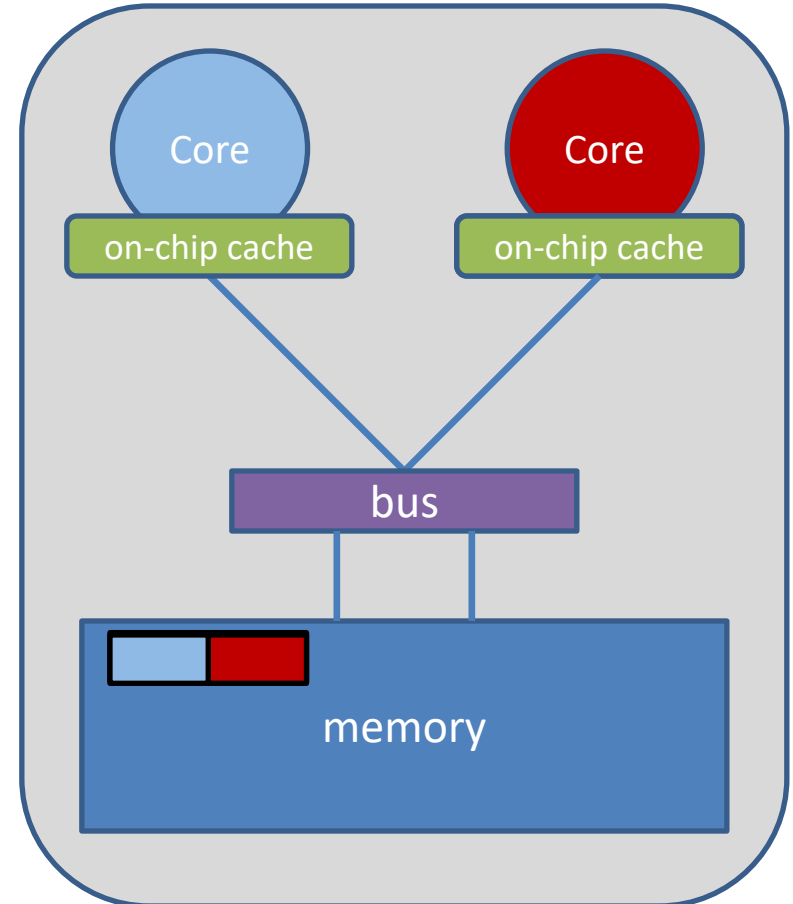
`^` (0), `min` (largest number), `max` (least number)

```
double s_priv[nthreads];  
  
#pragma omp parallel num_threads(nthreads)  
{  
    int t=omp_get_thread_num();  
  
    #pragma omp for  
    for (i = 0; i < 99; i++)  
    {  
        s_priv[t] += a[i];  
    }  
} // end parallel  
for (i = 0; i < nthreads; i++)  
{  
    s += s_priv[i];  
}
```

- When data is used, it is copied into caches.
- The hardware always copies chunks into the cache, so called *cache-lines*.
- This is useful, when:
  - the data is used frequently (temporal locality)
  - consecutive data is used which is on the same cache-line (spatial locality)

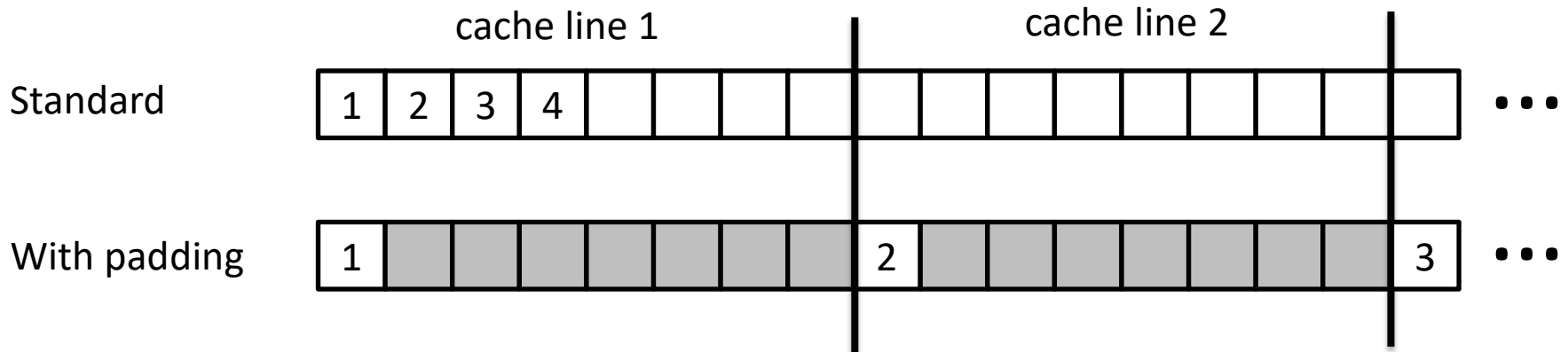
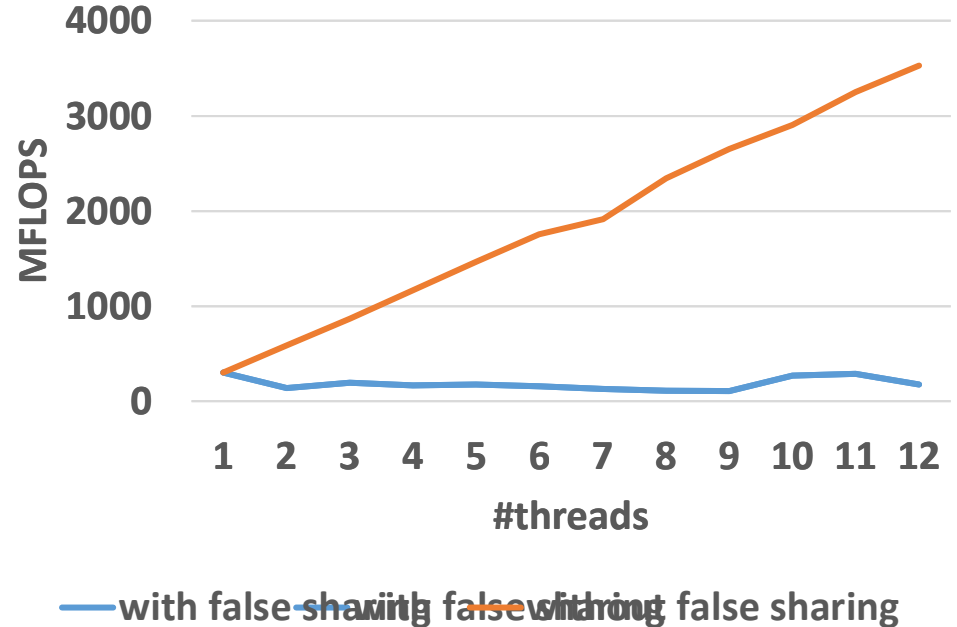


- **False Sharing occurs when**
  - different threads use elements of the same cache-line
  - one of the threads writes to the cache-line
- **As a result the cache line is moved between the threads, also there is no real dependency**
- **Note: False Sharing is a performance problem, not a correctness issue**



# False Sharing

- no performance benefit for more threads
- Reason: false sharing of `s_priv`
- Solution: padding so that only one variable per cache line is used



# False Sharing avoided



```
double s_priv[nthreads * 8];

#pragma omp parallel num_threads(nthreads)
{
    int t=omp_get_thread_num();

    #pragma omp for
    for (i = 0; i < 99; i++)
    {
        s_priv[t * 8] += a[i];
    }
} // end parallel
for (i = 0; i < nthreads; i++)
{
    s += s_priv[i * 8];
}
```

# PI



## Example: Pi (1/2)

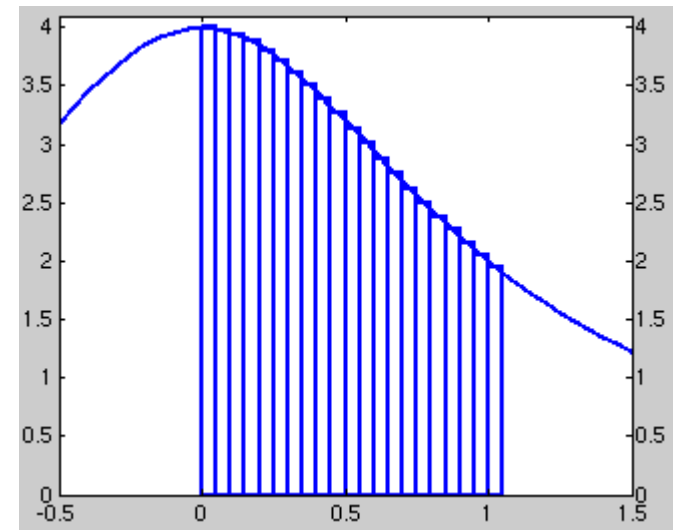


```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

```
double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

    #pragma omp parallel for
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



## Example: Pi (1/2)

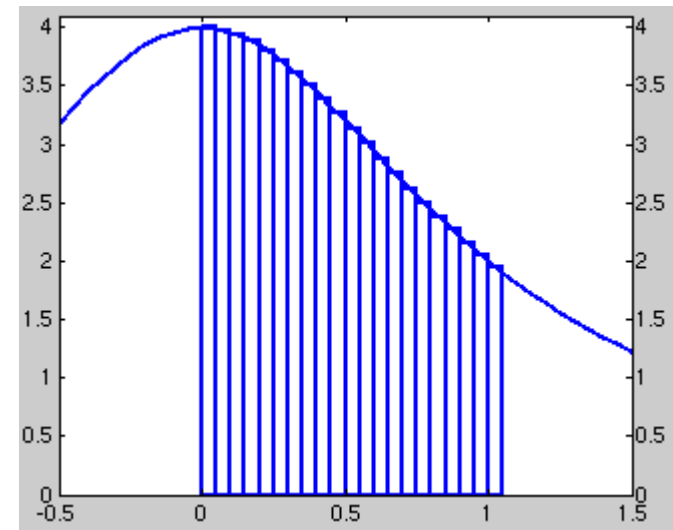


```
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}
```

```
double CalcPi (int n)
{
    const double fH  = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

    #pragma omp parallel for private(fX,i) reduction(+:fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

$$\pi = \int_0^1 \frac{4}{1+x^2}$$



### ■ Results:

# Threads	Runtime [sec.]	Speedup
1	1.11	1.00
2		
4		
8	0.14	7.93

### ■ Scalability is pretty good:

- About 100% of the runtime has been parallelized.
- As there is just one parallel region, there is virtually no overhead introduced by the parallelization.
- Problem is parallelizable in a trivial fashion ...

# Single and Master Construct

C/C++

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

Fortran

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

- The `single` construct specifies that the enclosed structured block is executed by only one thread of the team.

→ It is up to the runtime which thread that is.

- Useful for:

→ I/O

→ Memory allocation and deallocation, etc. (in general: setup work)

→ Implementation of the single-creator parallel-executor pattern as we will see now...

C/C++

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

Fortran

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

- The `master` construct specifies that the enclosed structured block is executed only by the master thread of a team.
- Note: The master construct is no worksharing construct and does not contain an implicit barrier at the end.

# 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.



# Tasking

# Recursive approach to compute Fibonacci



```
int main(int argc,  
         char* argv[])  
{  
    [...]  
    fib(45);  
    [...]  
}
```

```
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

- Deferring (or not) a unit of work (executable for any member of the team)

→ Always attached to a structured block

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

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

- Where clause:

→ private(list),

→ firstprivate(list),

→ shared(list)

→ default(shared | none)

→ in\_reduction(r-id: list)  $\geq 5.0$

→ untied

Data Environment

→ if(scalar-expression)

→ mergeable

→ final(scalar-expression)

→ depend(dep-type: list)

→ priority(priority-value)

Cutoff Strategies

Dependencies

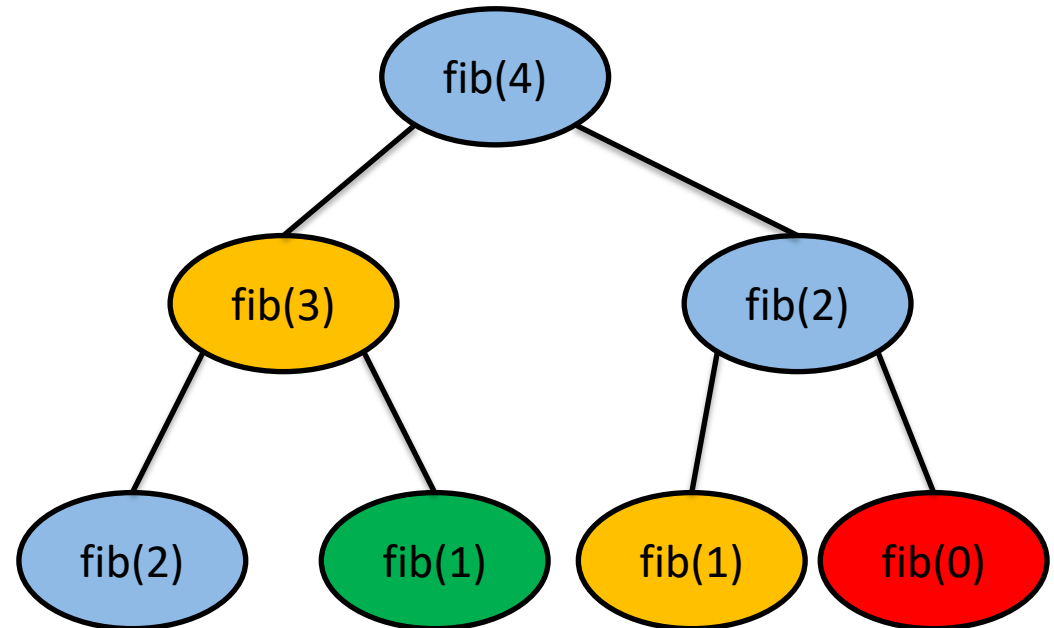
- **Some rules from *Parallel Regions* apply:**
  - Static and Global variables are shared
  - Automatic Storage (local) variables are private
  
- **If shared scoping is not derived by default:**
  - Orphaned Task variables are `firstprivate` by default!
  - Non-Orphaned Task variables inherit the `shared` attribute!
  - Variables are `firstprivate` unless `shared` in the enclosing context

```
int main(int argc,
        char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(45);
        }
    }
    [...]
```

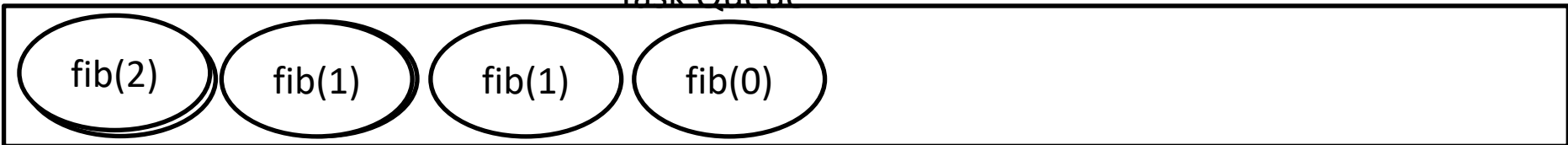
```
int fib(int n)    {
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
    return x+y;
}
```

- **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**

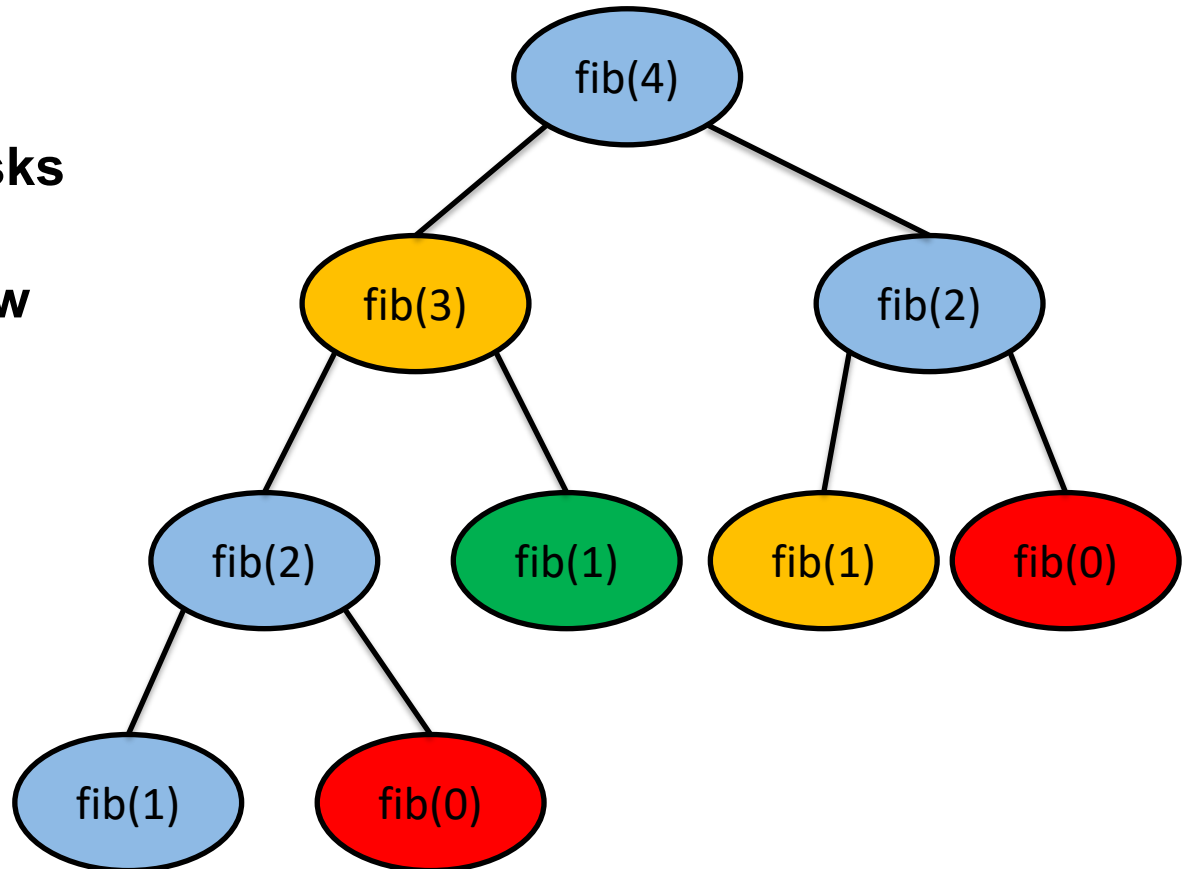
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks



Task Queue

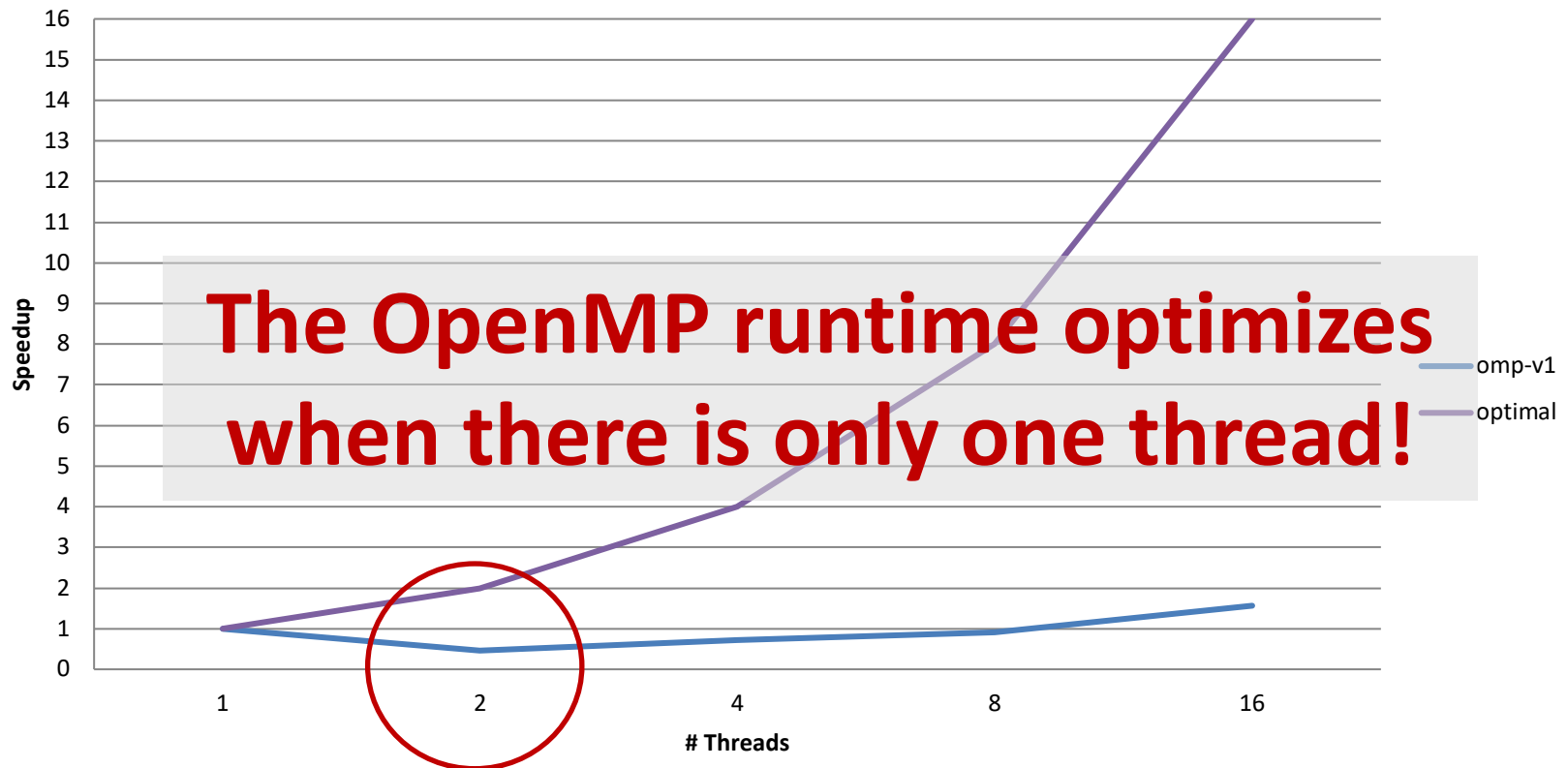


- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- ...



## ■ Overhead of task creation prevents scalability!

Speedup of Fibonacci with Tasks





## ■ 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 *undelayed* task

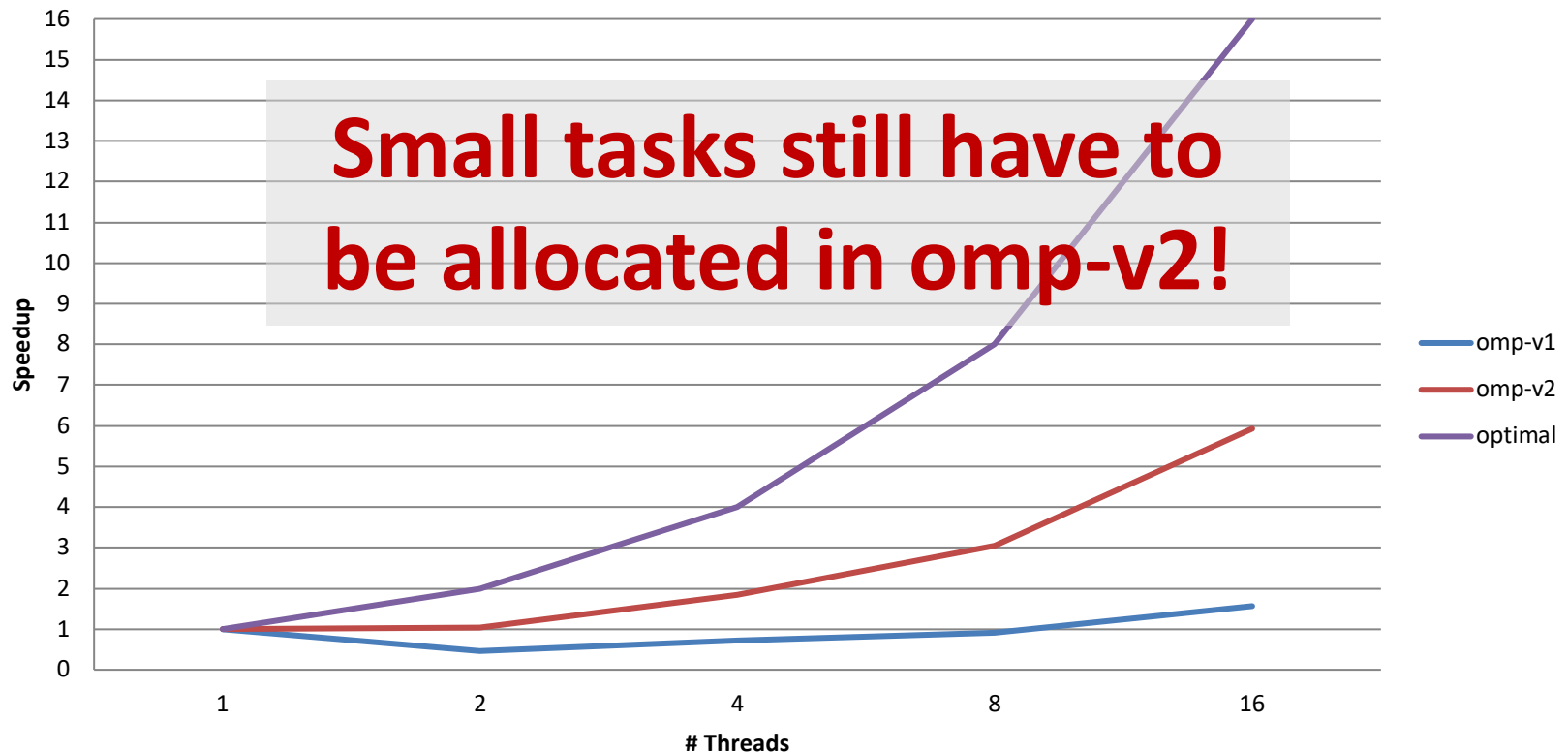
- **Improvement: Don't create yet another task once a certain (small enough)  $n$  is reached**

```
int main(int argc,
        char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(45);
        }
    }
    [...]
}
```

```
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

Speedup of Fibonacci with Tasks



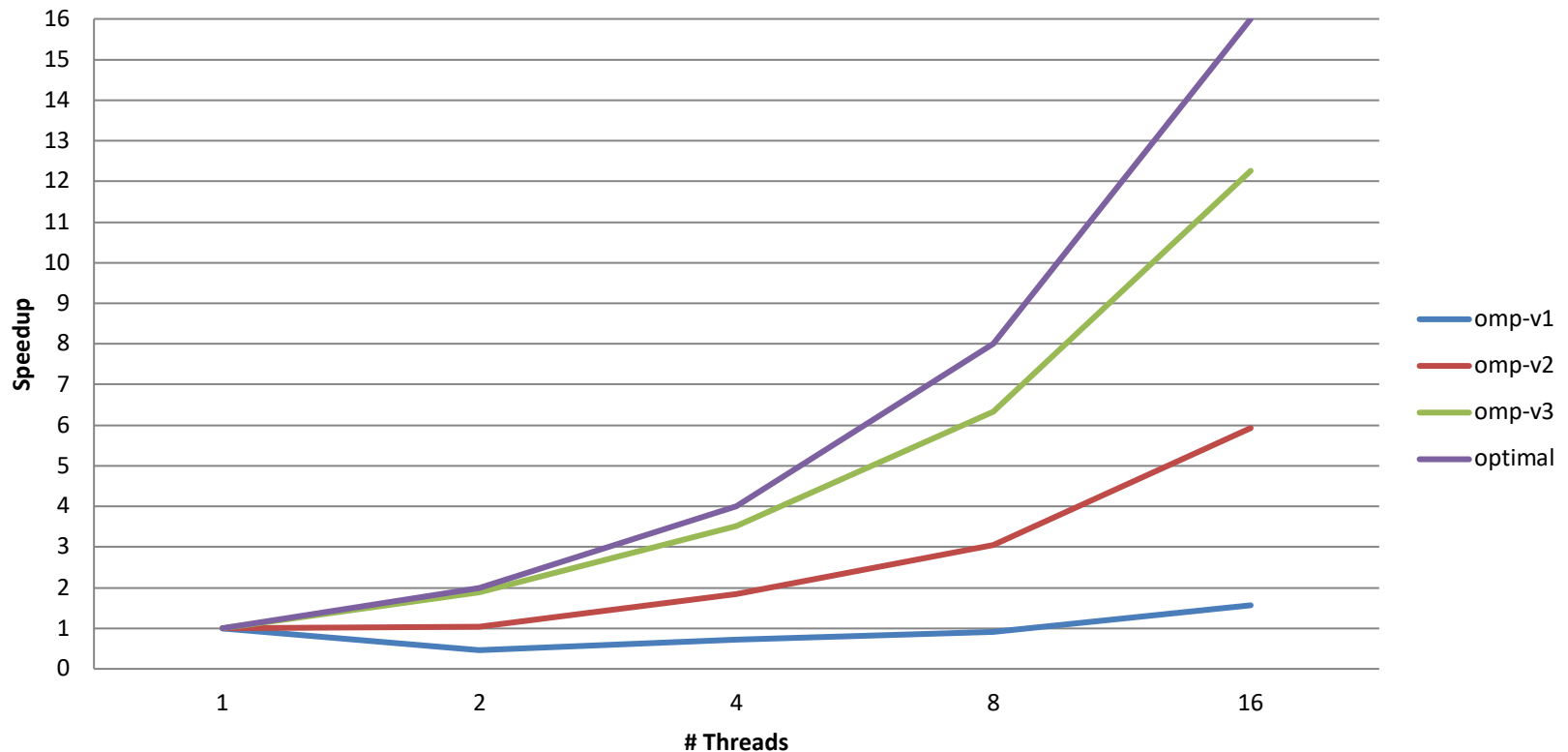
## ■ Improvement: Skip the OpenMP overhead once a certain *n* is reached

```
int main(int argc,
         char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(45);
        }
    }
    [...]
}
```

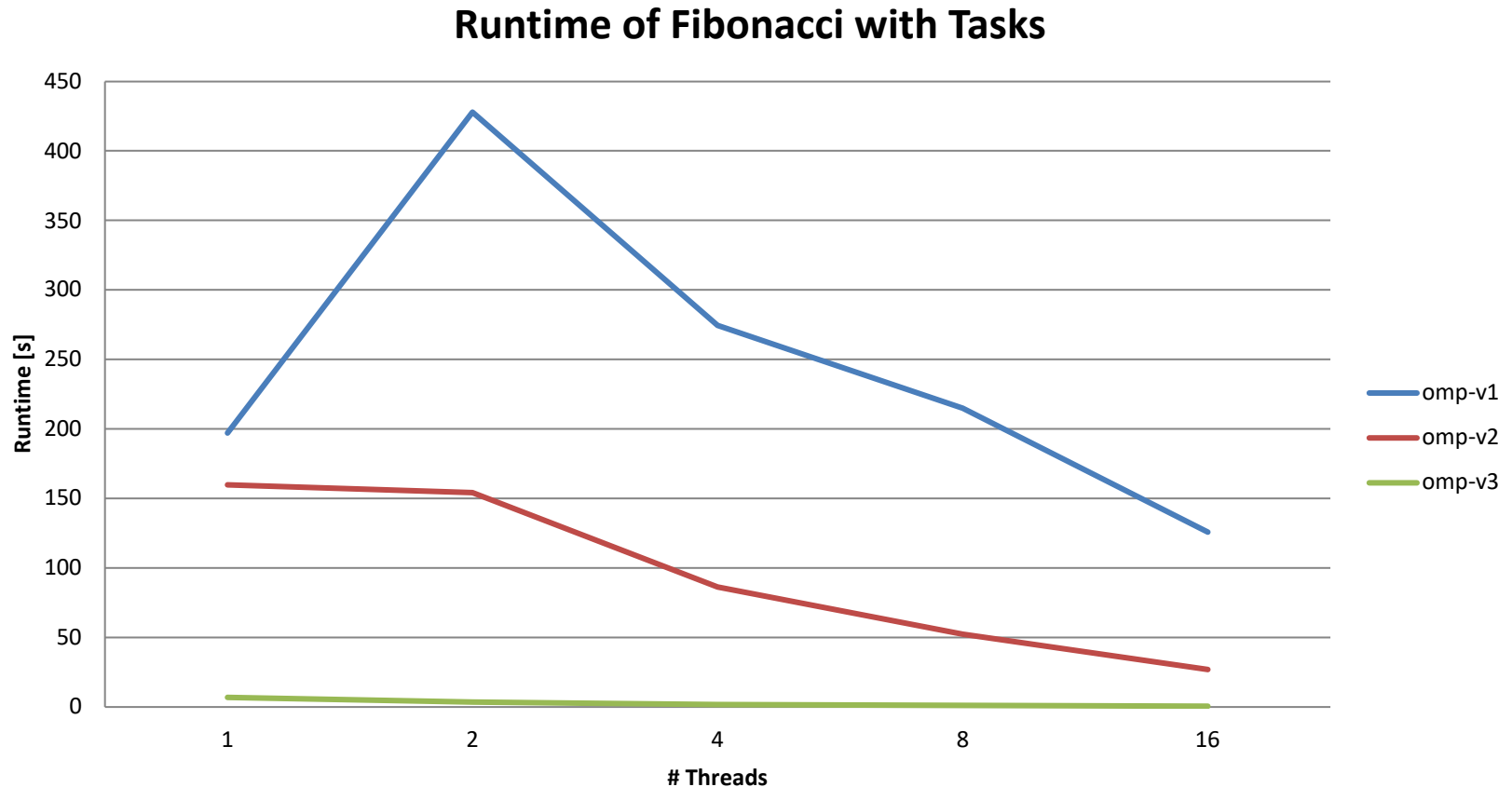
```
int fib(int n)    {
    if (n < 2) return n;
    if (n <= 30)
        return serfib(n);
    int x, y;
    #pragma omp task shared(x)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
    return x+y;
}
```

## ■ Looks promising...

Speedup of Fibonacci with Tasks

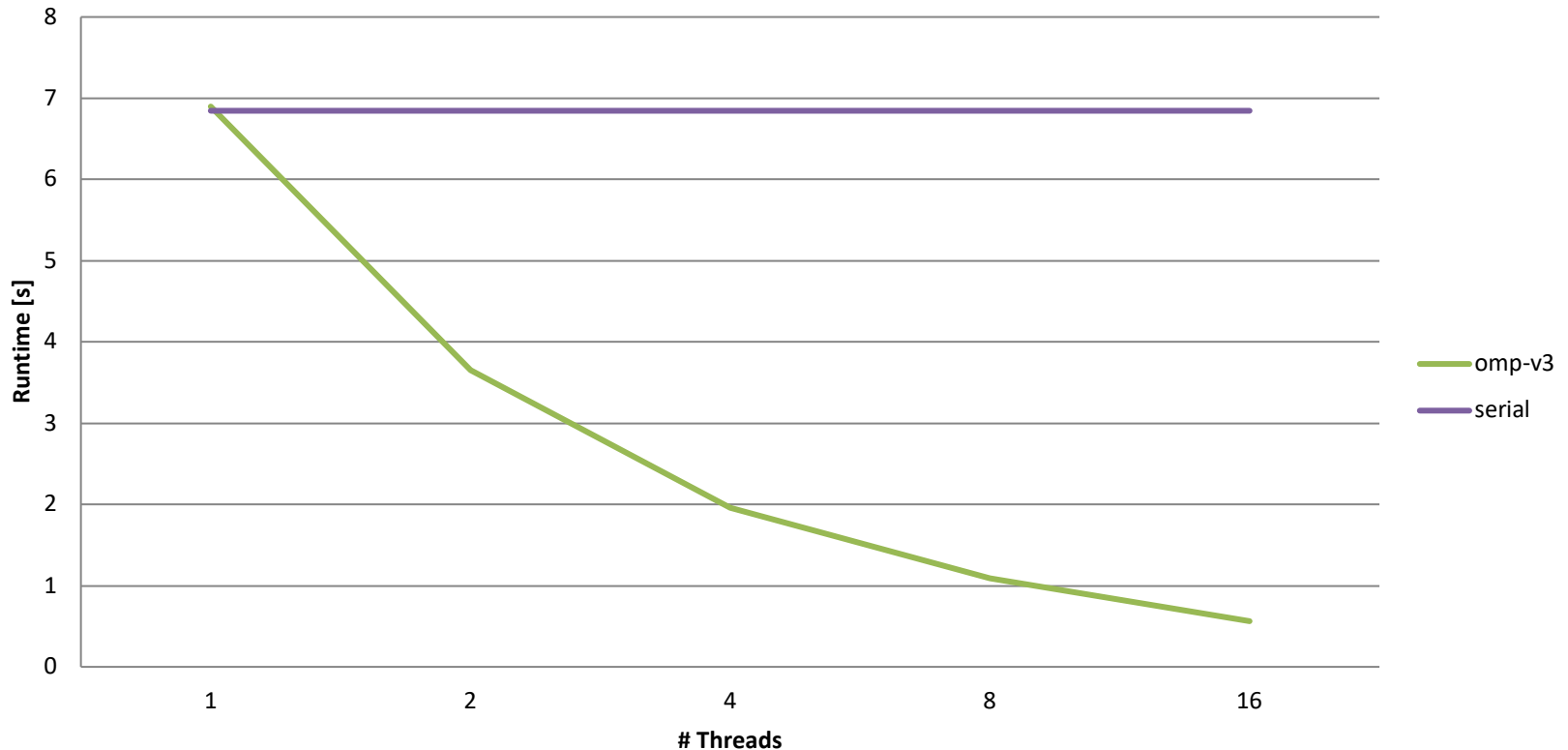


- First two versions were slow because of overhead!



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

Runtime of Fibonacci with Tasks



# 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

# Data Scoping

# 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

        }
    }
}
```

# Task Synchronization

## ■ 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 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

# Questions?