Introduction to OpenMP

Christian Terboven, Dirk Schmidl

IT Center, RWTH Aachen University
{terboven,schmidl}@itc.rwth-aachen.de

History

- **De-facto standard for Shared-Memory Parallelization.**

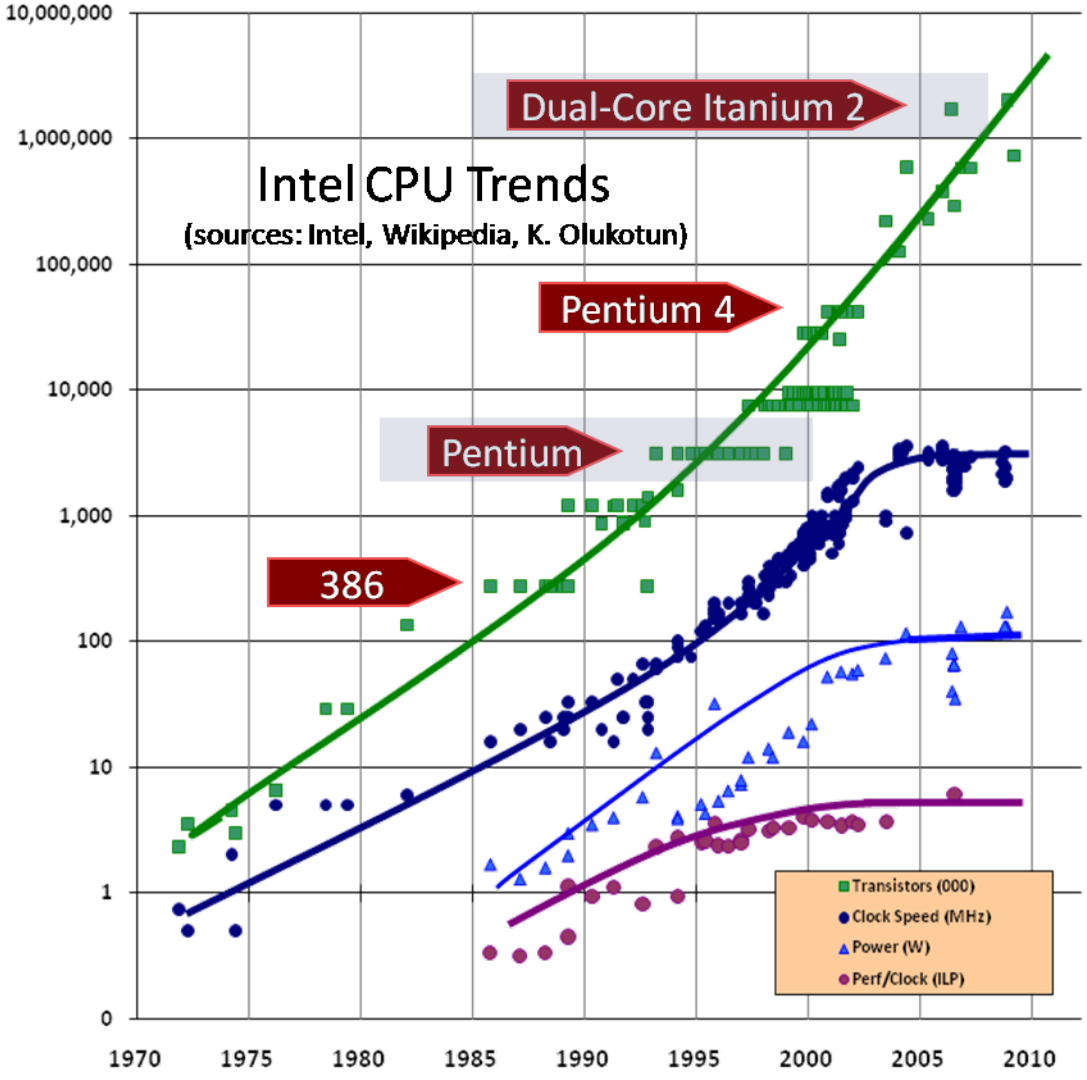
- **1997: OpenMP 1.0 for FORTRAN**
- **1998: OpenMP 1.0 for C and C++**
- **1999: OpenMP 1.1 for FORTRAN (errata)**
- **2000: OpenMP 2.0 for FORTRAN**
- **2002: OpenMP 2.0 for C and C++**
- **2005: OpenMP 2.5 now includes both programming languages.**
- **05/2008: OpenMP 3.0 release**
- **07/2011: OpenMP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

http://www.OpenMP.org

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

Multi-Core System Architecture

Moore‘s Law still holds!



The number of transistors on a chip is still doubling every 24 months …

… but the clock speed is no longer increasing that fast!

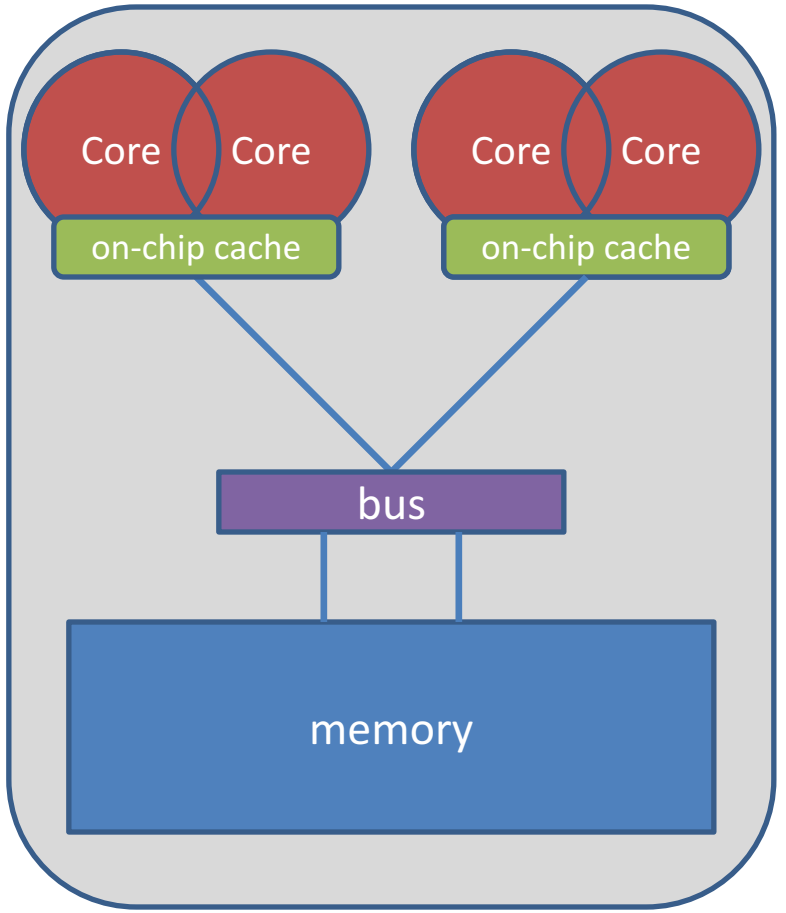
Instead, we will see many more cores per chip!

Source: Herb Sutter

www.gotw.ca/publications/concurrency-ddj.htm

Example for a SMP system

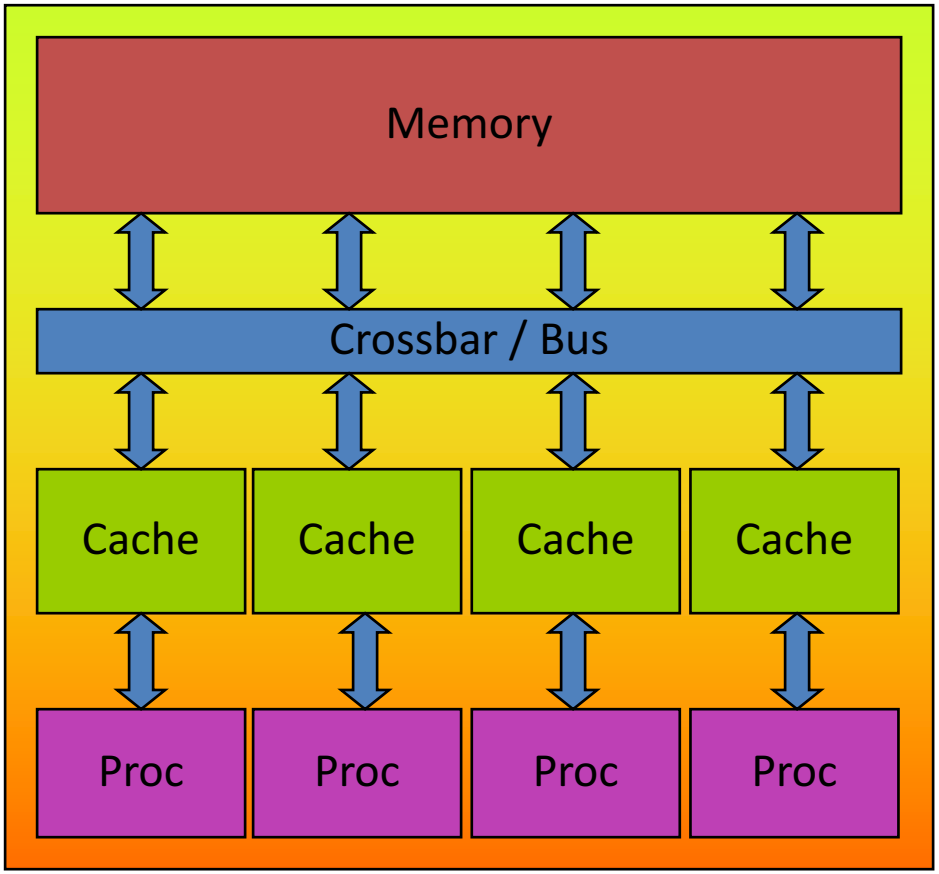
- **Dual-socket Intel Woodcrest (dual-core) system**
  - → Two cores per chip, 3.0 GHz
  - → Each chip has 4 MB of L2 cache on-chip, shared by both cores
  - → No off-chip cache
  - → Bus: Frontsidebus
- **SMP: Symmetric Multi Processor**
  - → Memory access time is uniform on all cores
  - → Limited scalabilty



OpenMP Overview
&
Parallel Region

OpenMP‘s machine model

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



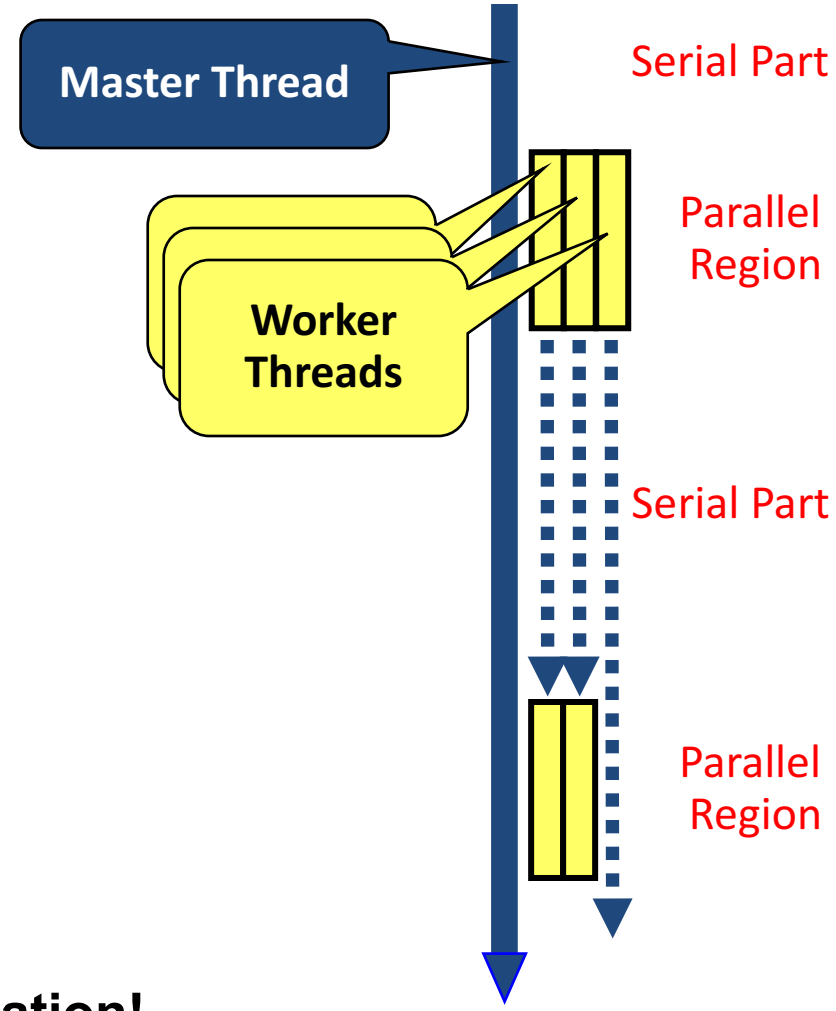
All processors/cores access a shared main memory.

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

Parallelization in OpenMP employs multiple threads.

OpenMP Execution Model

- 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!



Parallel Region and Structured Blocks

- **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

Starting OpenMP Programs on Linux

- 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

For Worksharing

- **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!**

Worksharing illustrated

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

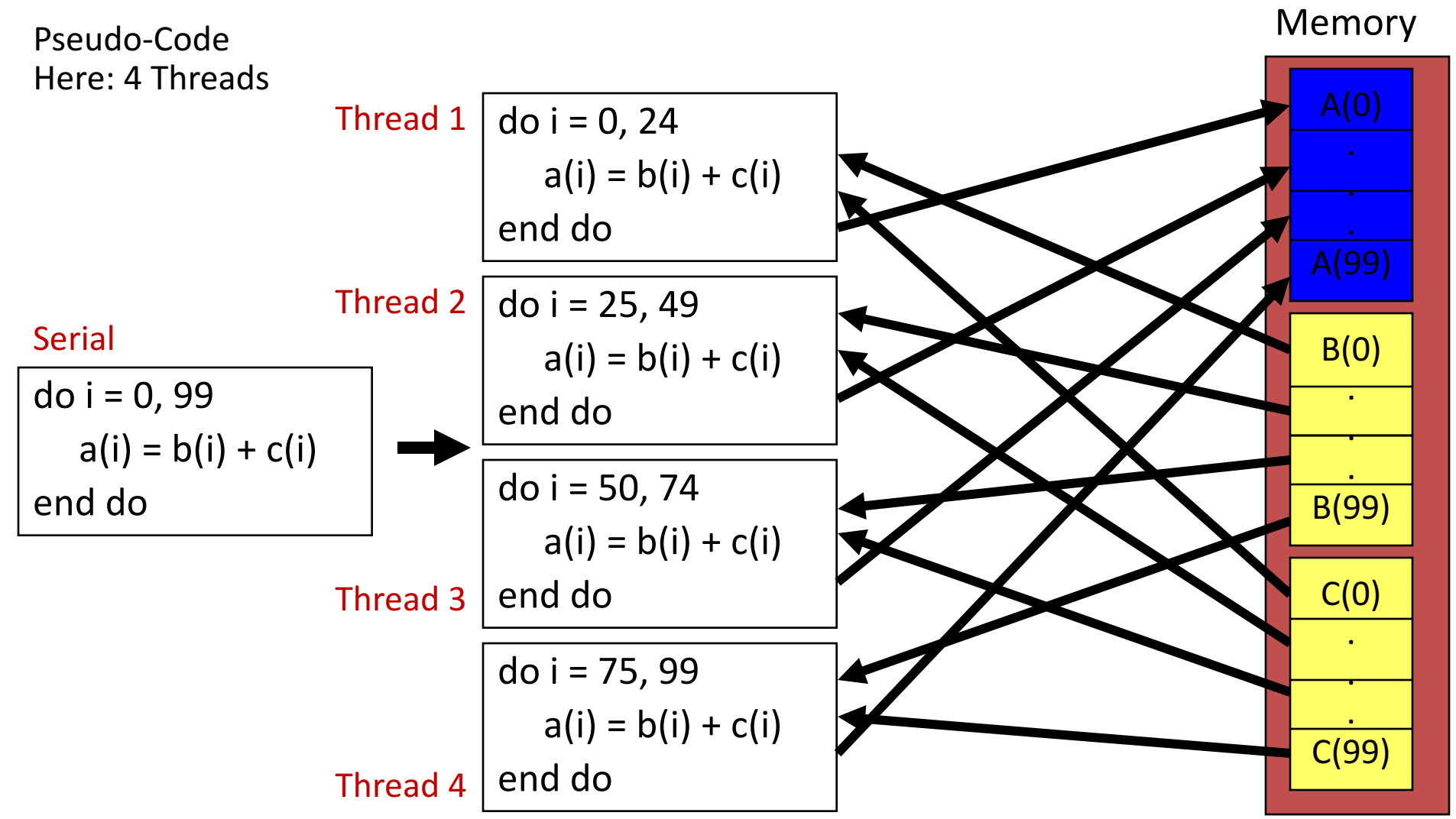
```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

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
```



Vector Addition

Influencing the For Loop Scheduling

- ***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)`.**

Synchronization Overview

- **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).**

Synchronization: Critical Region

- **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

Scoping Rules

- **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, *firstprivate* for Tasks.
  - → 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*

Privatization of Global/Static Variables

- **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)
```

Privatization of Global/Static Variables

- **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)

Really: try to avoid the use of threadprivate and static variables!

C/C++

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

Fortran

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

The Barrier Construct

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
```

The Reduction Clause

- **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),`

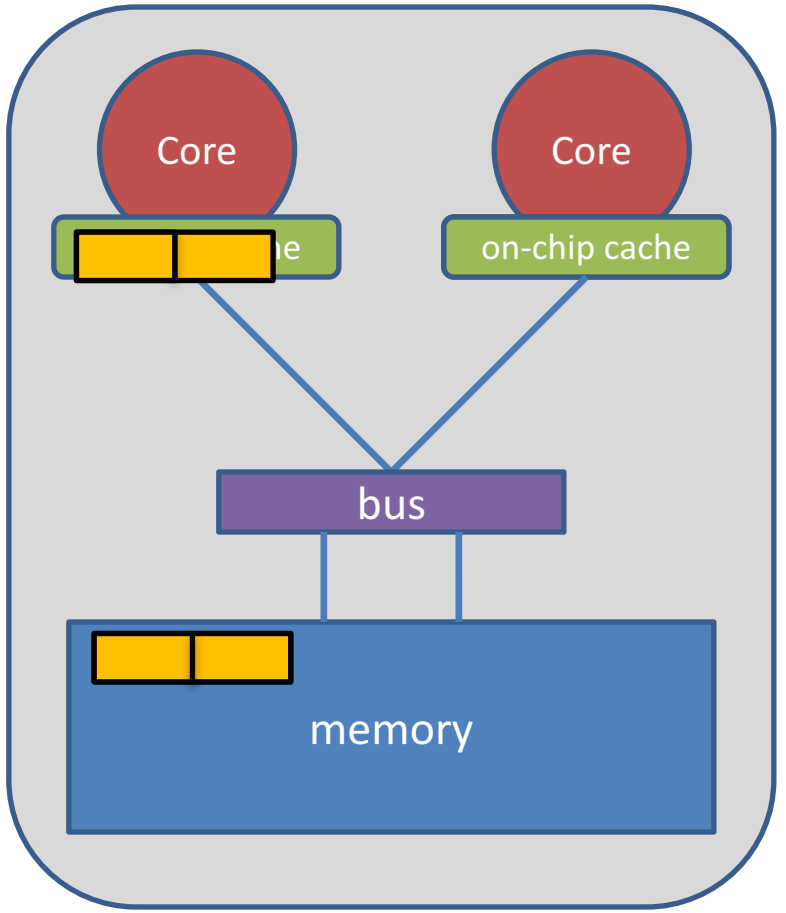
  `^ (0), min (largest number), max (least number)`

False Sharing

```
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];
}
```

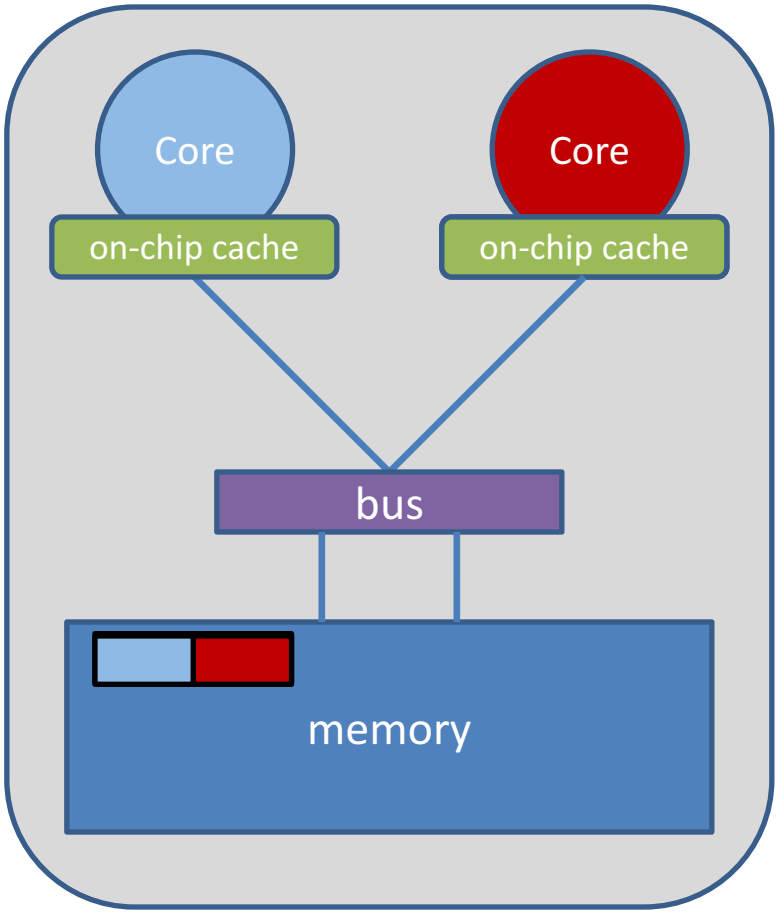
Data in Caches

- **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)



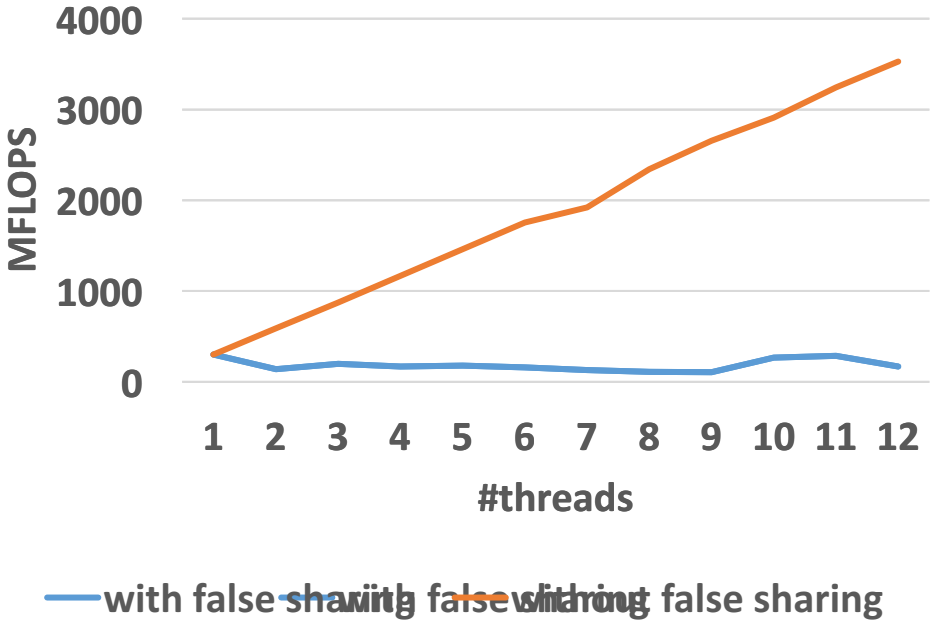
False Sharing

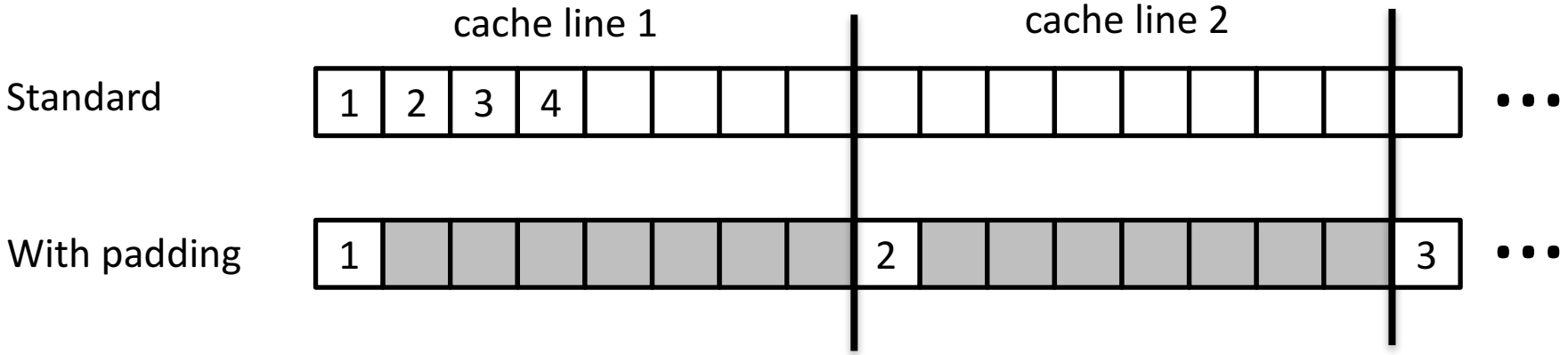
- **False Sharing occurs when**
  - → different threads use elements of the same cache-line
  - → one of the threads writes to the cache-line
- **As a result the cache line is moved between the threads, also there is no real dependency**
- **Note: False Sharing is a performance problem, not a correctness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

Example

PI

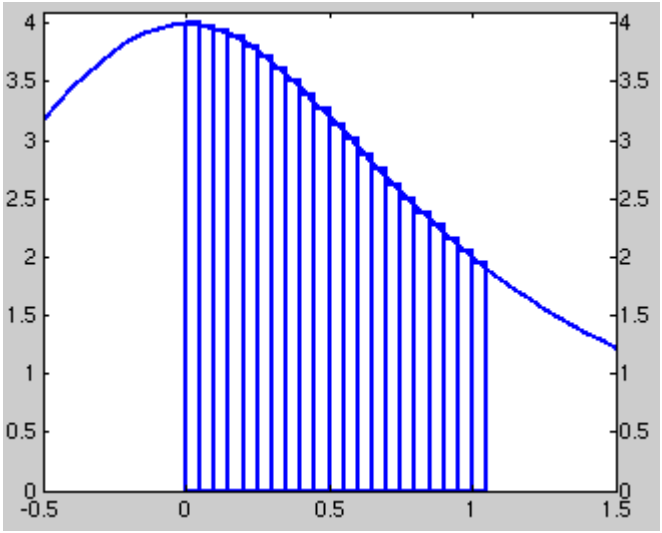
Example: Pi (1/2)

```
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;
}
```

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



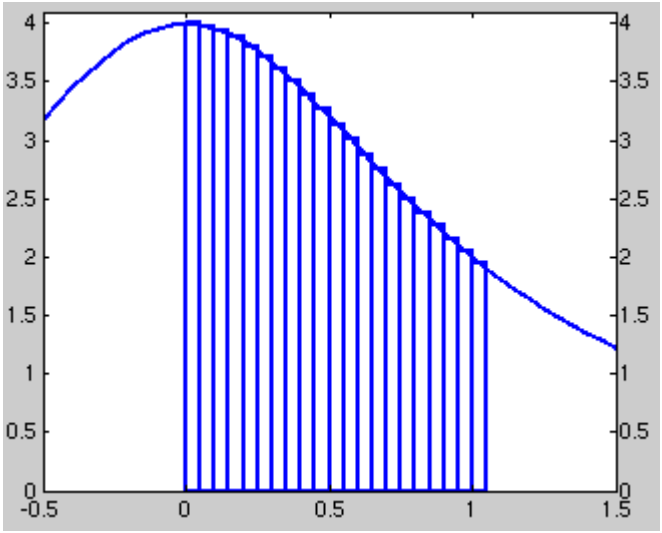
Example: Pi (1/2)

```
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;
}
```

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



Example: Pi (2/2)

- **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

The Single 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 on 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…

The Master Construct

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.**

Section and Ordered Construct

How to parallelize a Tree Traversal?

- How would you parallelize this code?

```
void traverse (Tree *tree)
{
    if (tree->left)    traverse(tree->left);

    if (tree->right)    traverse(tree->right);

    process(tree);
}
```

- One option: Use OpenMP‘s parallel sections.

The Sections Construct

C/C++

```
#pragma omp sections [clause]
{
  #pragma omp section
  ... structured block ...
  #pragma omp section
  ... structured block ...
 ...
}
```

Fortran

```
!$omp sections [clause]
  !$omp section
  ... structured block ...
  !$ omp section
  ... structured block ...
  ...
!$omp end sections
```

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

How to parallelize a Tree Traversal?!

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

```
   void traverse (Tree *tree)
   {
#pragma omp parallel sections
{
#pragma omp section
       if (tree->left)    traverse(tree->left);
#pragma omp section
       if (tree->right)    traverse(tree->right);
} // end omp parallel
       process(tree);
   }
```

Nested Parallel Regions

Barrier here!

- **Downsides of this option:**
  - → Unneccessary overhead and synchronization points
  - → Not always well supported (how many threads to be used?)

The ordered Construct

- **Allows to execute a structured block within a parallel loop in sequential order**
  - → In addition, an `ordered` clause has to be added to the *for* construct which any *ordered* construct may occur

```
#pragma omp parallel for ordered
for (i=0 ; i<10 ; i++){
        ...
        #pragma omp ordered
        {
                ...
        }
        ...
}
```

- **Use Cases:**
  - → Can be used e.g. to enforce ordering on printing of data
  - → May help to determine whether there is a data race

Runtime Library

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(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)

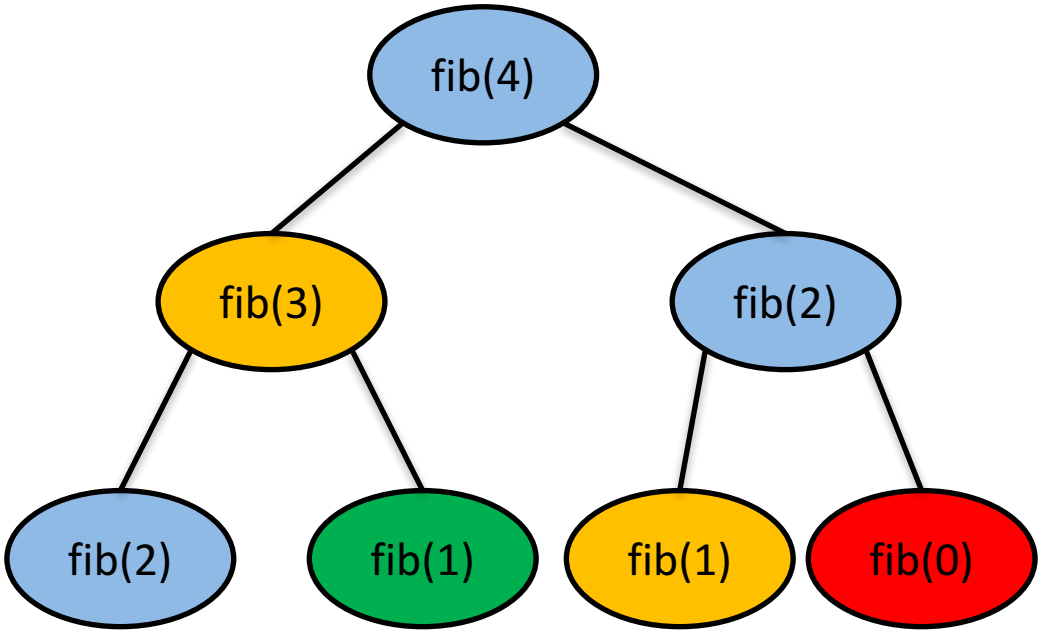
```
int main(int argc,
         char* argv[])
{
   [...]
   #pragma omp parallel
   {
        #pragma omp single
        {
                fib(input);
        }
   }
   [...]
}
```

```
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;
   }
```

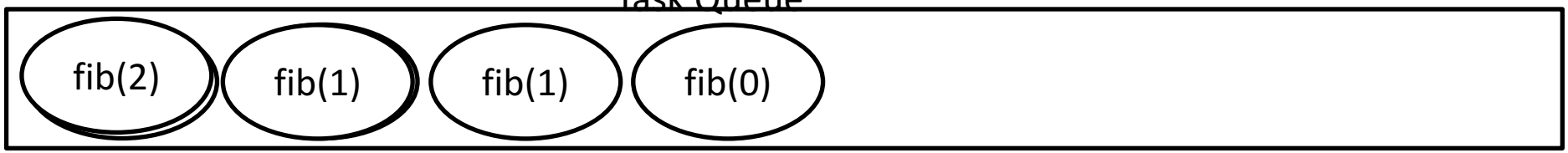
- Only one Task / Thread enters `fib()` from `main()`, it is responsable for creating the two initial work tasks
- Taskwait is required, as otherwise `x` and `y` would be lost

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

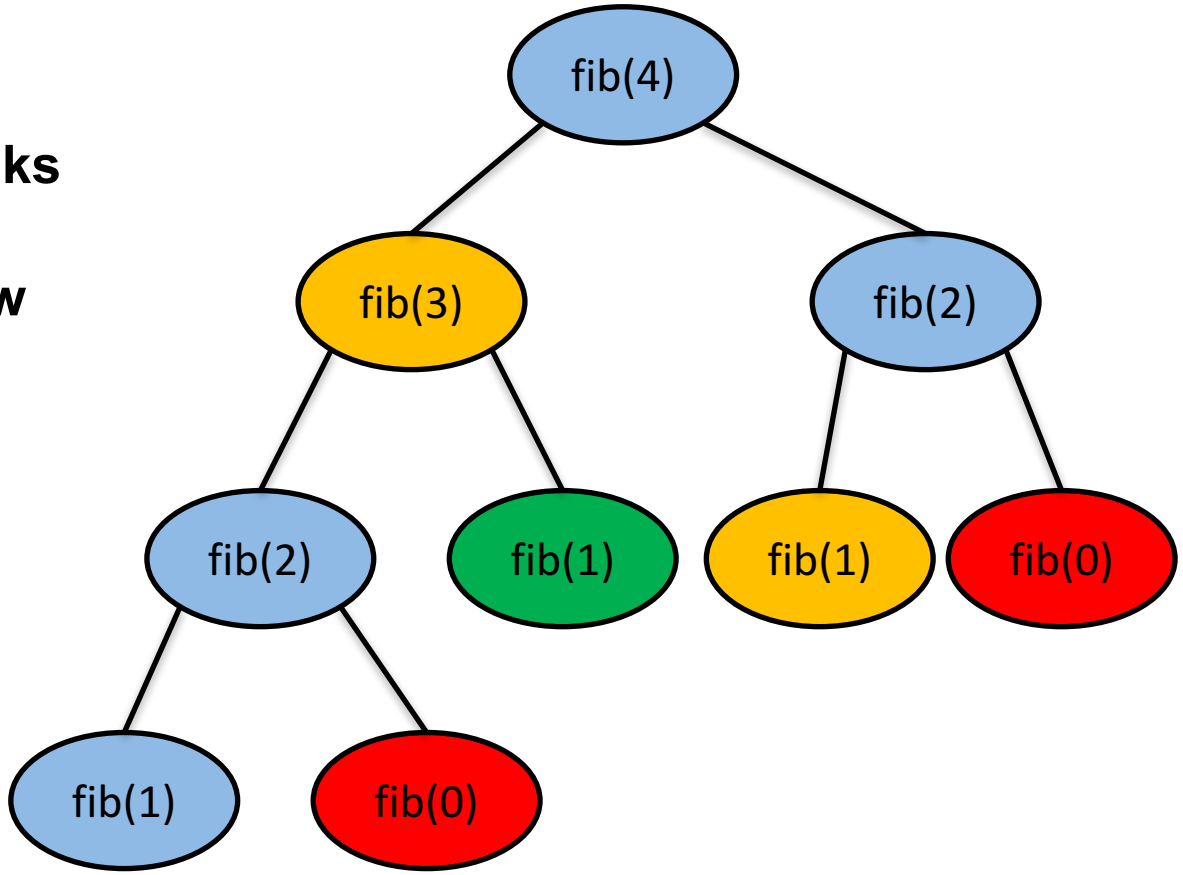


Task Queue



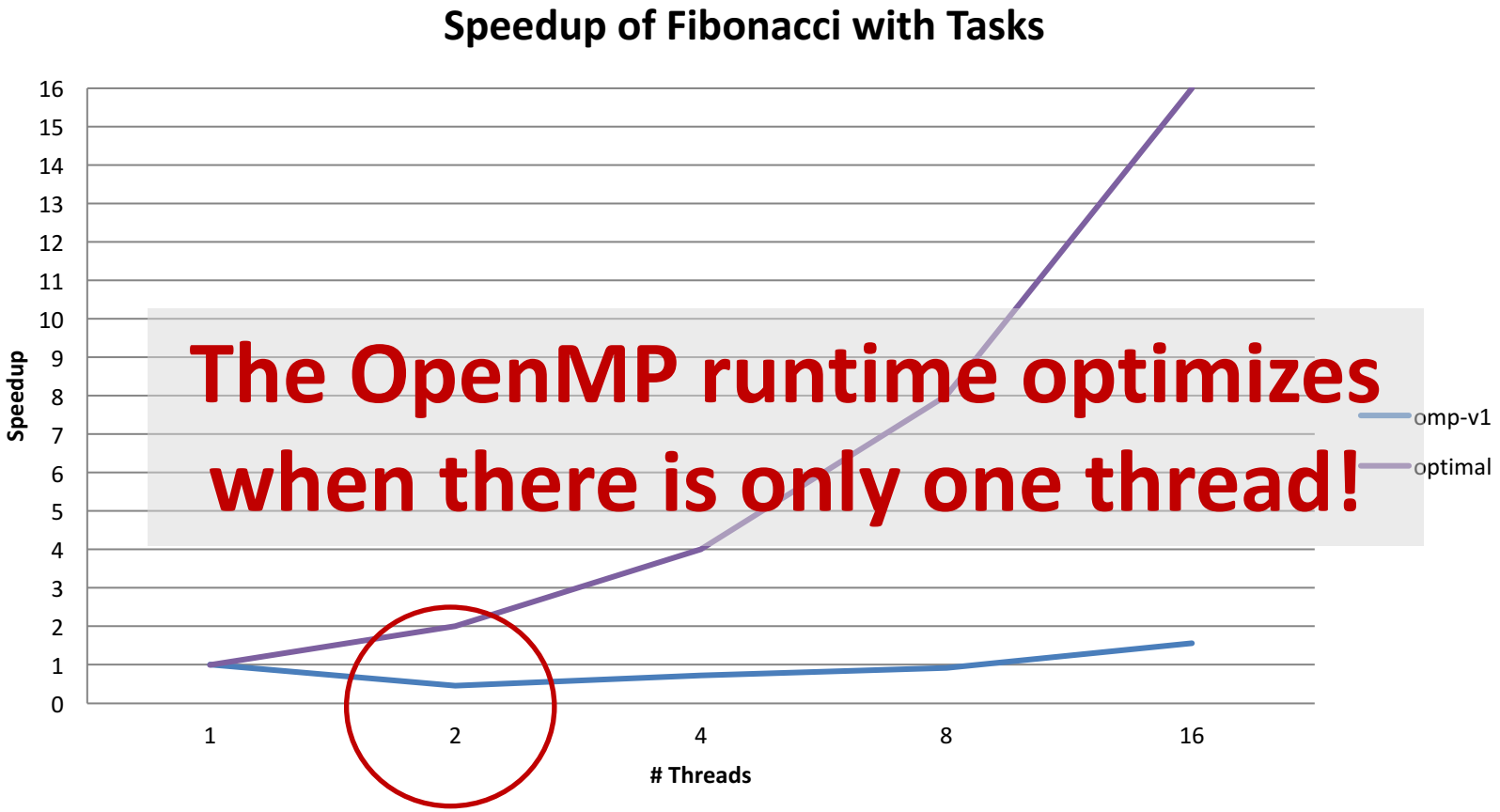
Fibonacci Illustration

- 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 scalability!**



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

`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, avoids queuing of small tasks

Improved parallelization with Tasking (omp-v2)

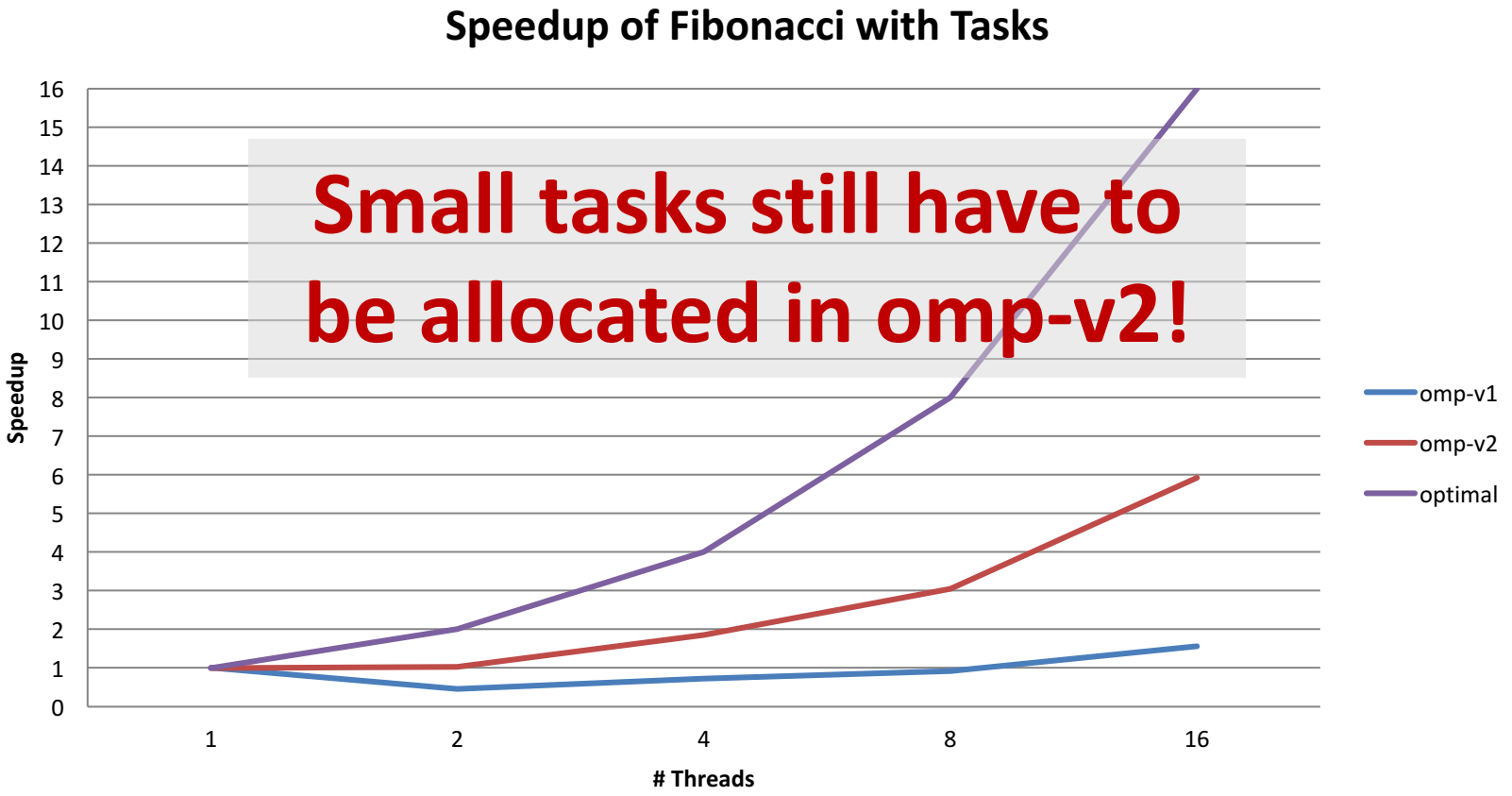
- **Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached**

```
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 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)

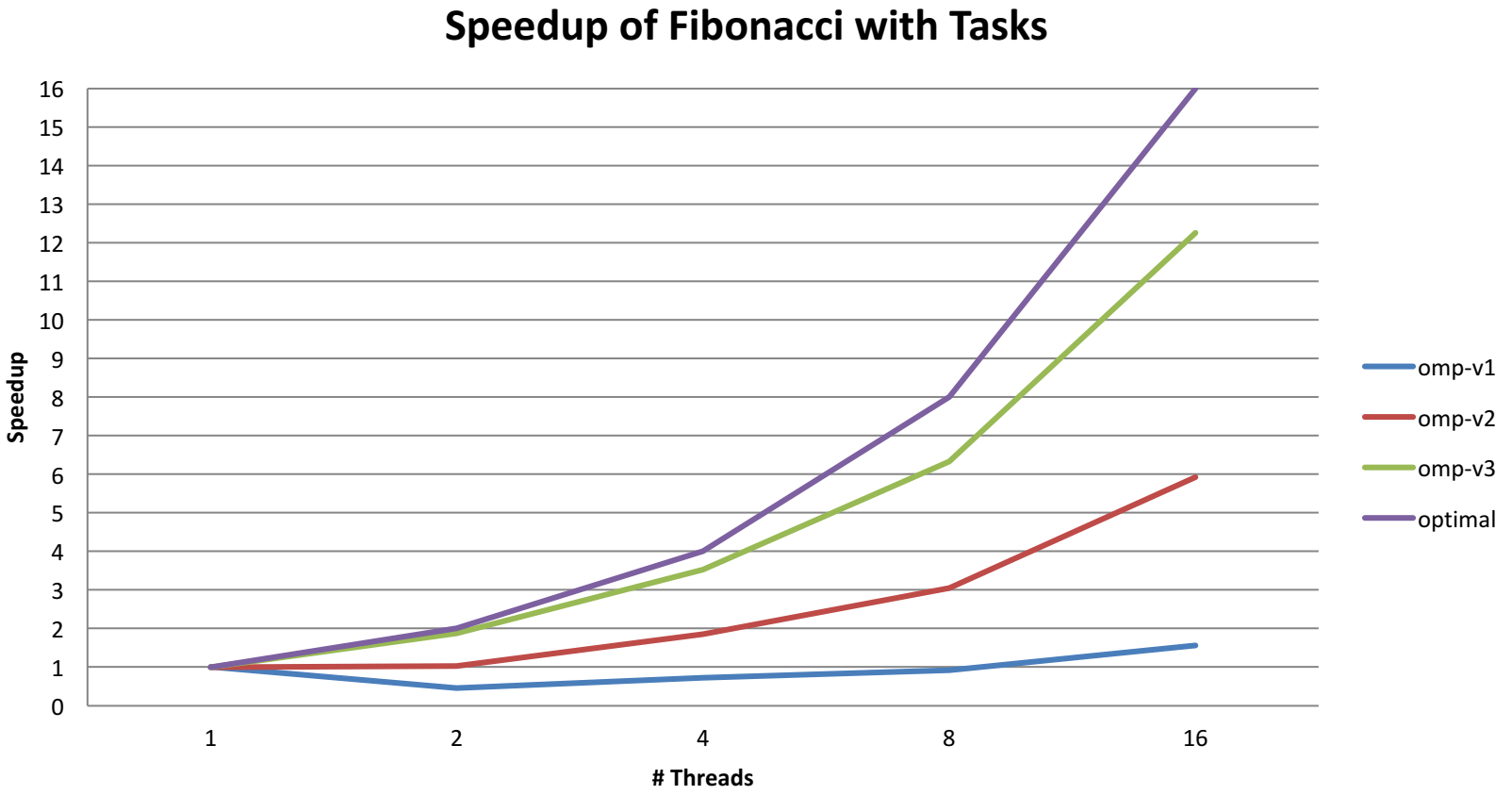
- **Improvement: Skip the OpenMP overhead once a certain `n` is reached**

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

```
int fib(int n)   {
   if (n < 2) return n;
   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)

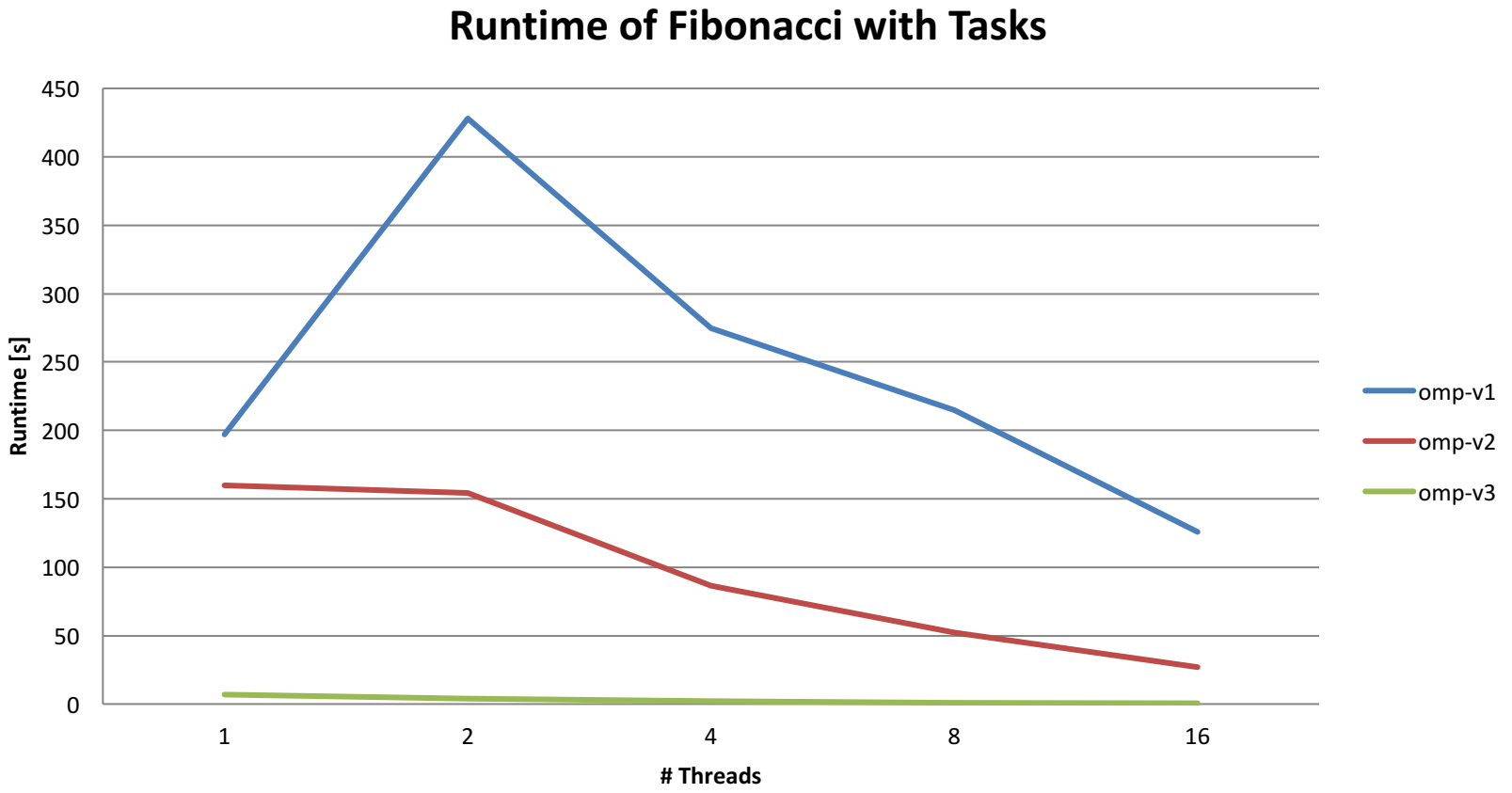
- **Looks promising…**



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

Runtime measurements (1/2)

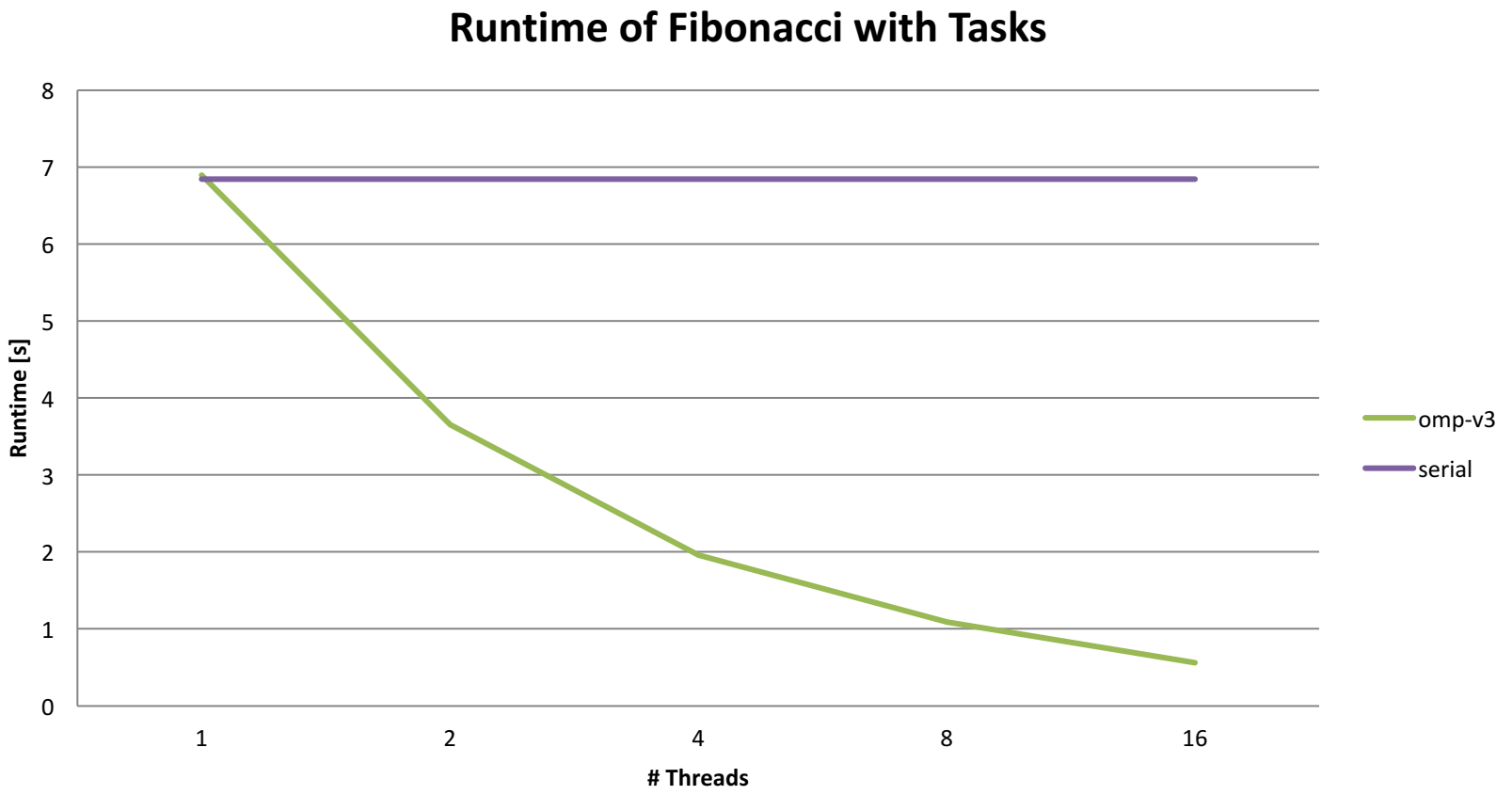
- First two versions were slow because of overhead!



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

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

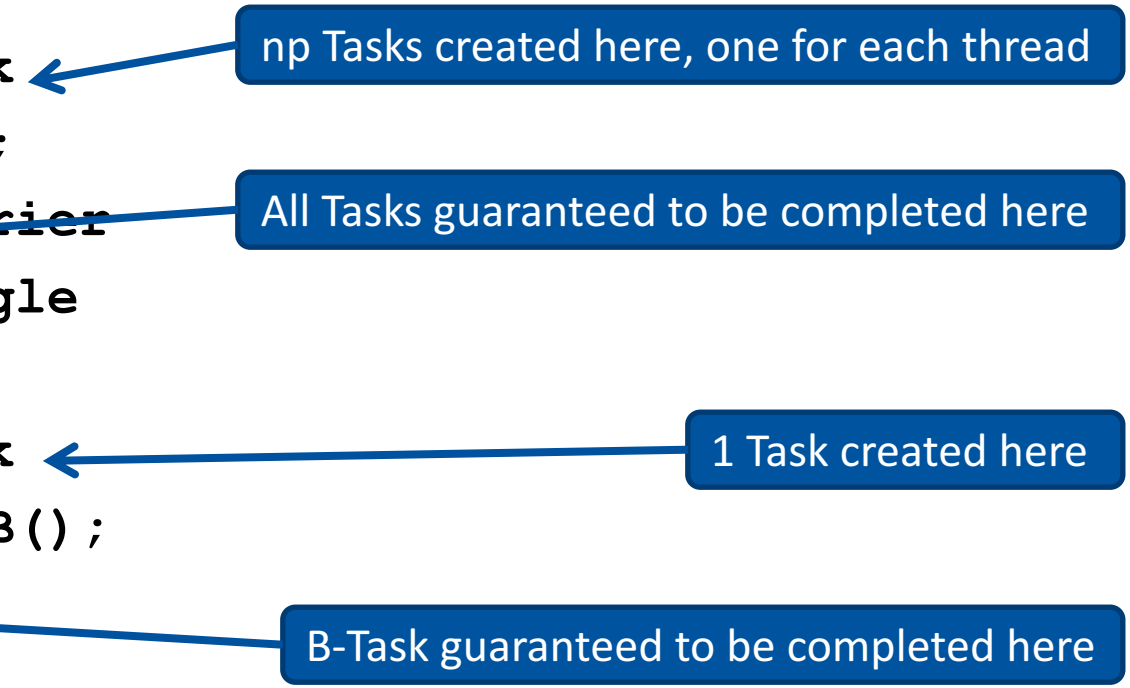
  → Only direct childs, not descendants!

```
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();
   }
}
```



Questions?