Introduction to OpenMP

Christian Terboven, Dirk Schmidl
IT Center, RWTH Aachen University
Member of the HPC Group
{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**

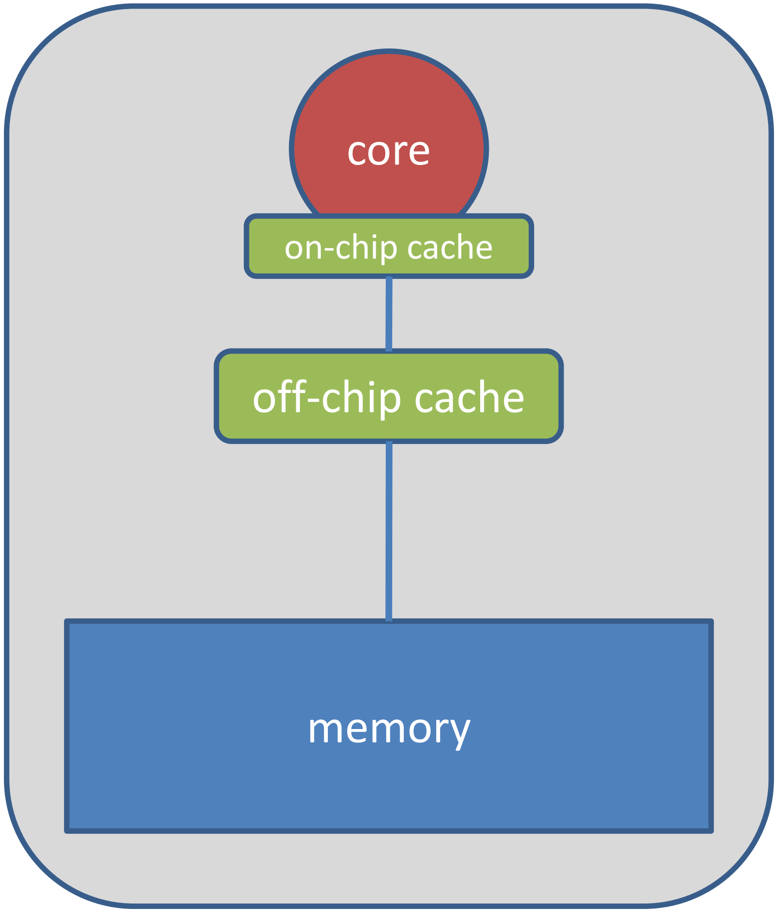
http://www.OpenMP.org

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

Multi-Core System Architecture

Single Processor System (dying out)

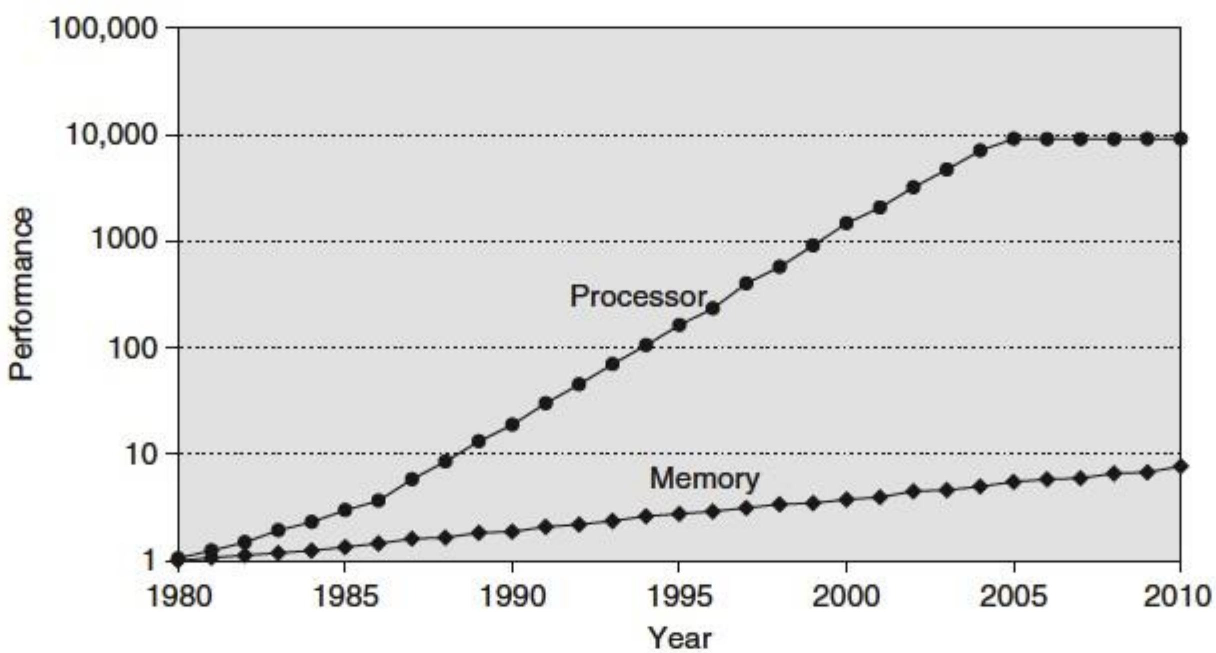
- **CPU is fast**
  - → Order of 3.0 GHz
- **Caches:**
  - → Fast, but expensive
  - → Thus small, order of MB
- **Memory is slow**
  - → Order of 0.3 GHz
  - → Large, order of GB



- **A good utilization of caches is crucial for good performance of HPC applications!**

Memory Bottleneck

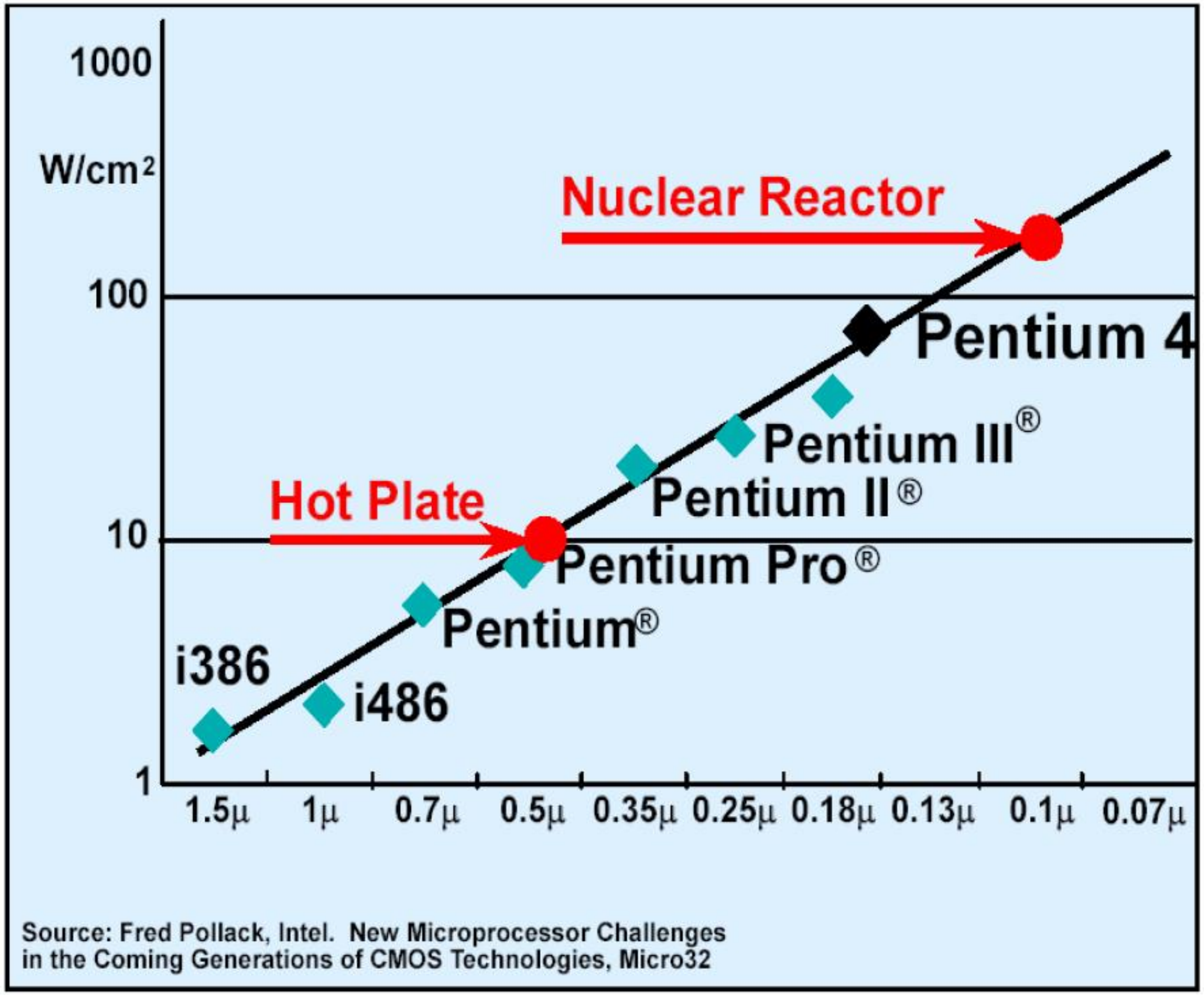
- **There is a growing gap between core and memory performance:**
  - → memory, since 1980: 1.07x per year improvement in latency
  - → single core: since 1980: 1.25x per year until 1986, 1.52x p. y. until 2000, 1.20x per year until 2005, then no change on a *per-core* basis



→ Source: John L. Hennessy, Stanford University, and David A. Patterson, University of California, September 25, 2012

Why is there no 4.0 GHz x86 CPU?

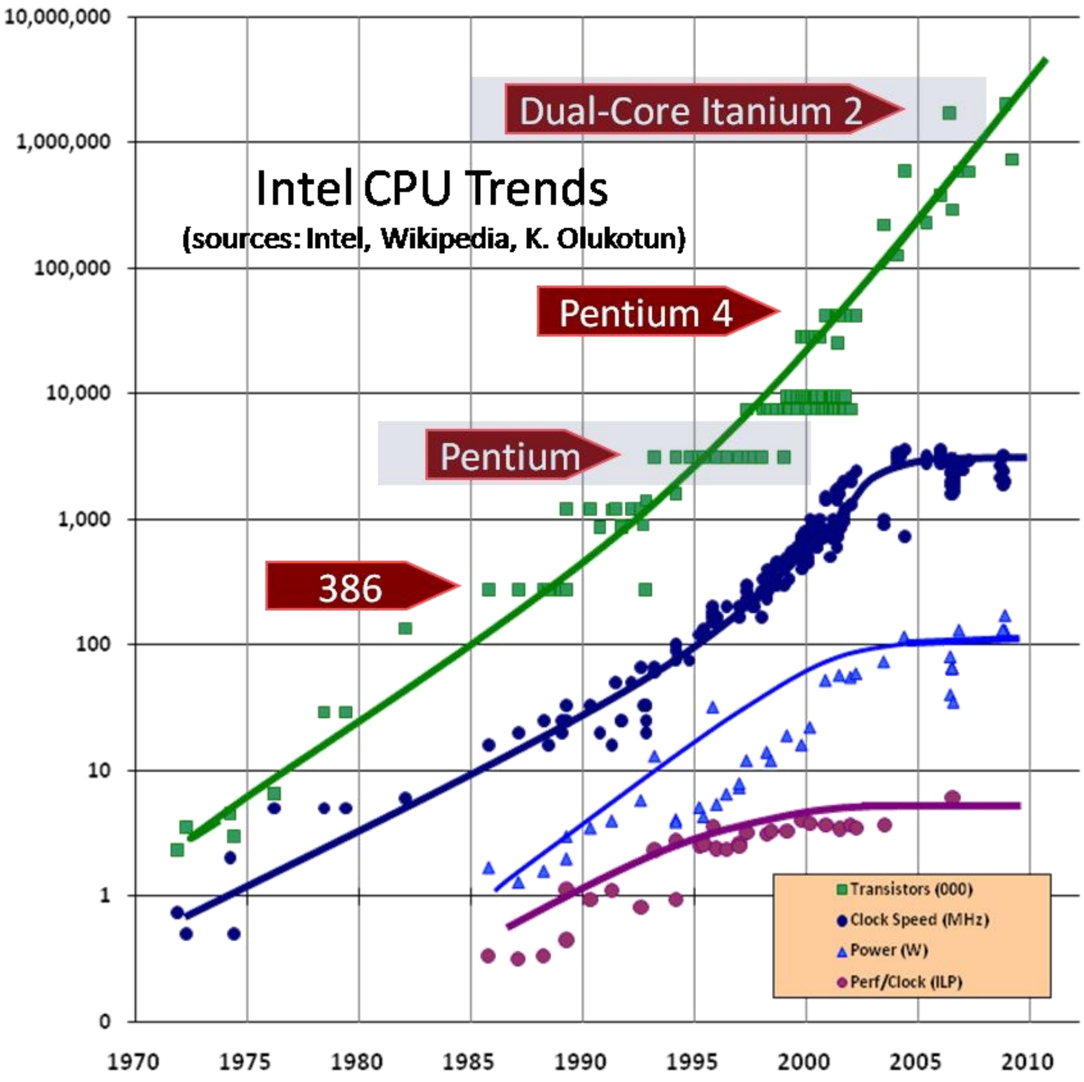
- **Because that beast would get too hot!**



Fast clock cycles make processor chips more expensive, hotter and more power consuming.

C. Terboven | IT Center der RWTH Aachen University

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!

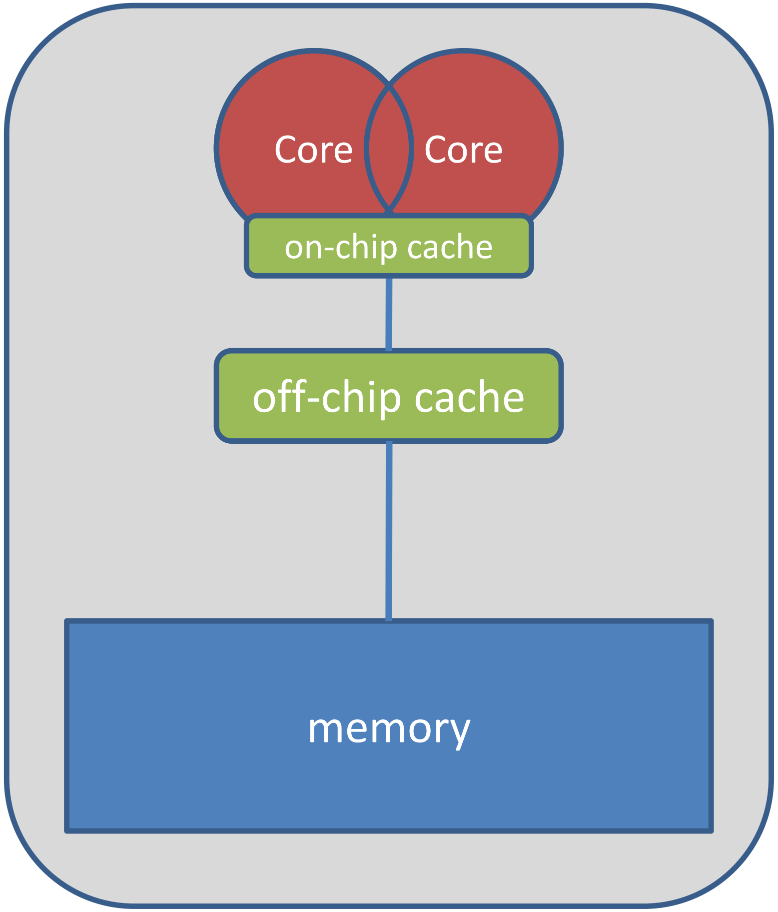
Instead, we will see many more cores per chip!

Source: Herb Sutter

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

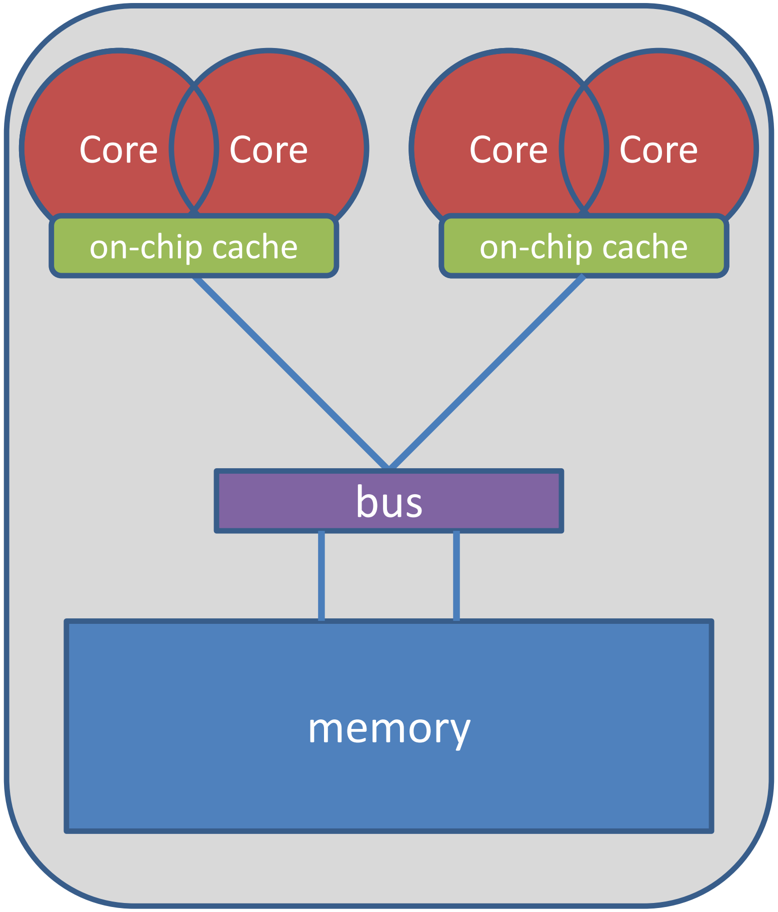
Dual-Core Processor System

- **Since 2005/2006 Intel and AMD are producing dual-core processors for the mass market!**
- **In 2006/2007 Intel and AMD introduced quad-core processors.**
- **→ Any recently bought PC or laptop is a multi-core system already!**



Example for a SMP system

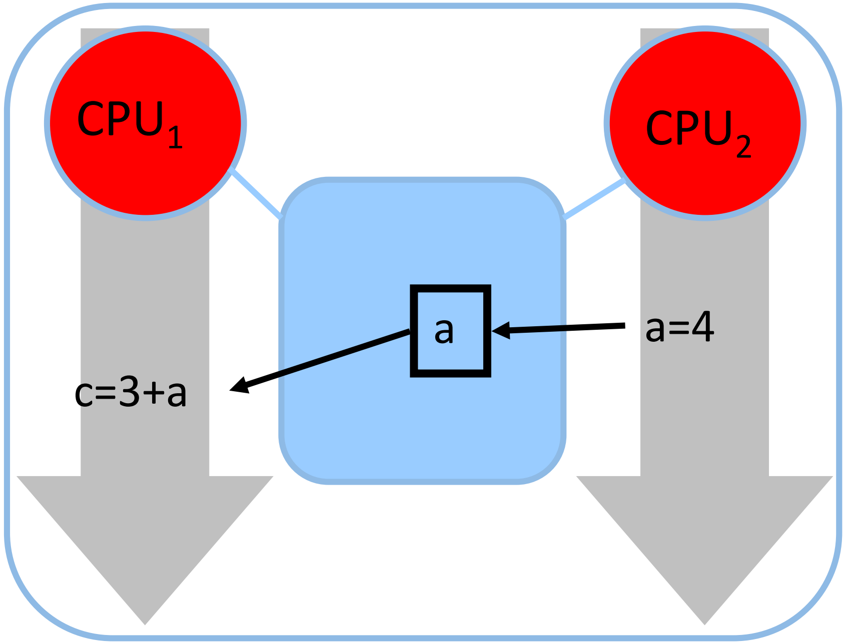
- **Dual-socket Intel Woodcrest (dual-core) system**
  - → Two cores per chip, 3.0 GHz
  - → Each chip has 4 MB of L2 cache on-chip, shared by both cores
  - → No off-chip cache
  - → Bus: Frontsidebus

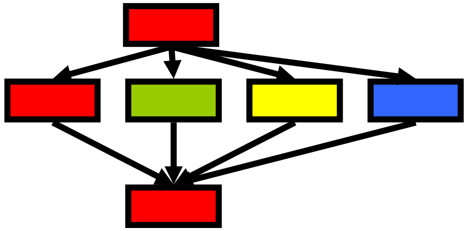


- **SMP: Symmetric Multi Processor**
  - → Memory access time is uniform on all cores
  - → Limited scalabilty

Shared Memory Parallelization

- **Memory can be accessed by several threads running on different cores in a multi-socket multi-core system:**



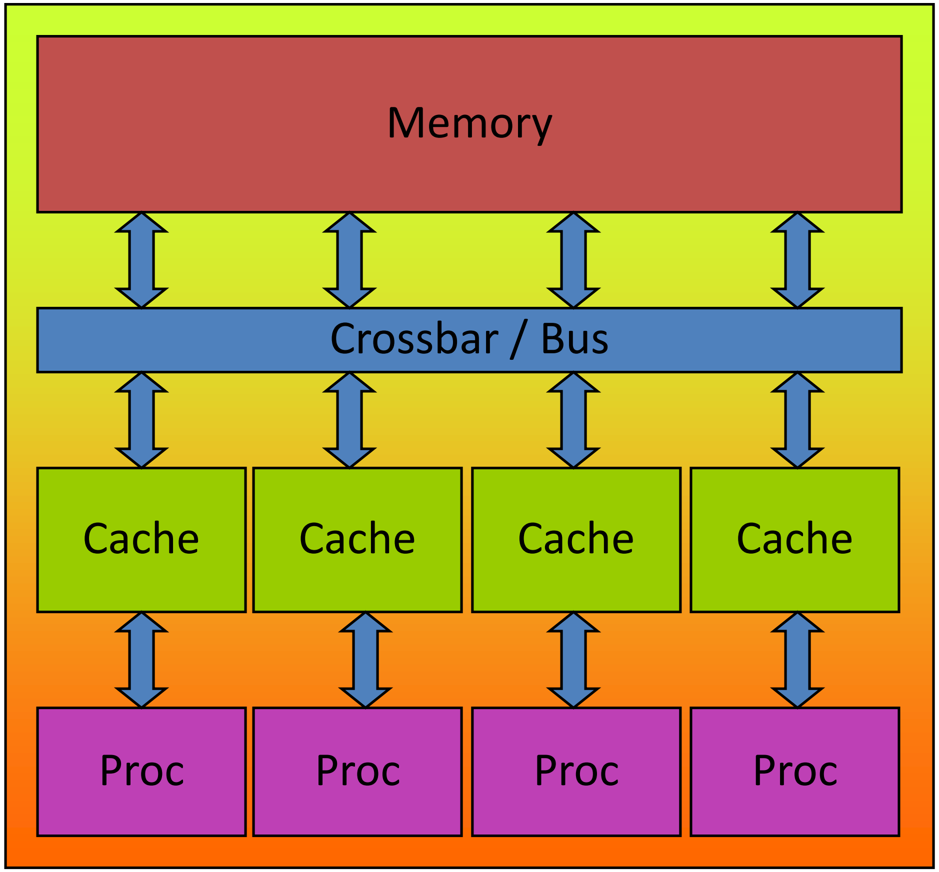


Look for tasks that can be executed simultaneously (task parallelism)

OpenMP Overview
&
Parallel Region

OpenMP‘s machine model

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



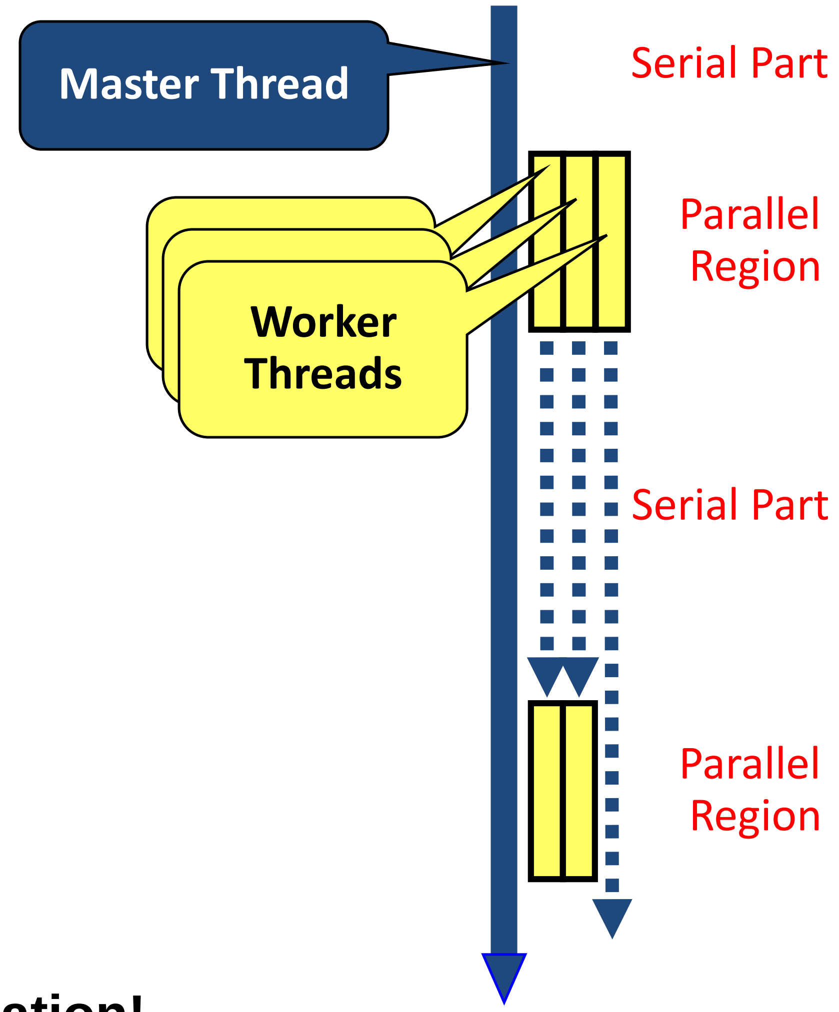
All processors/cores access a shared main memory.

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++

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

Fortran

```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.

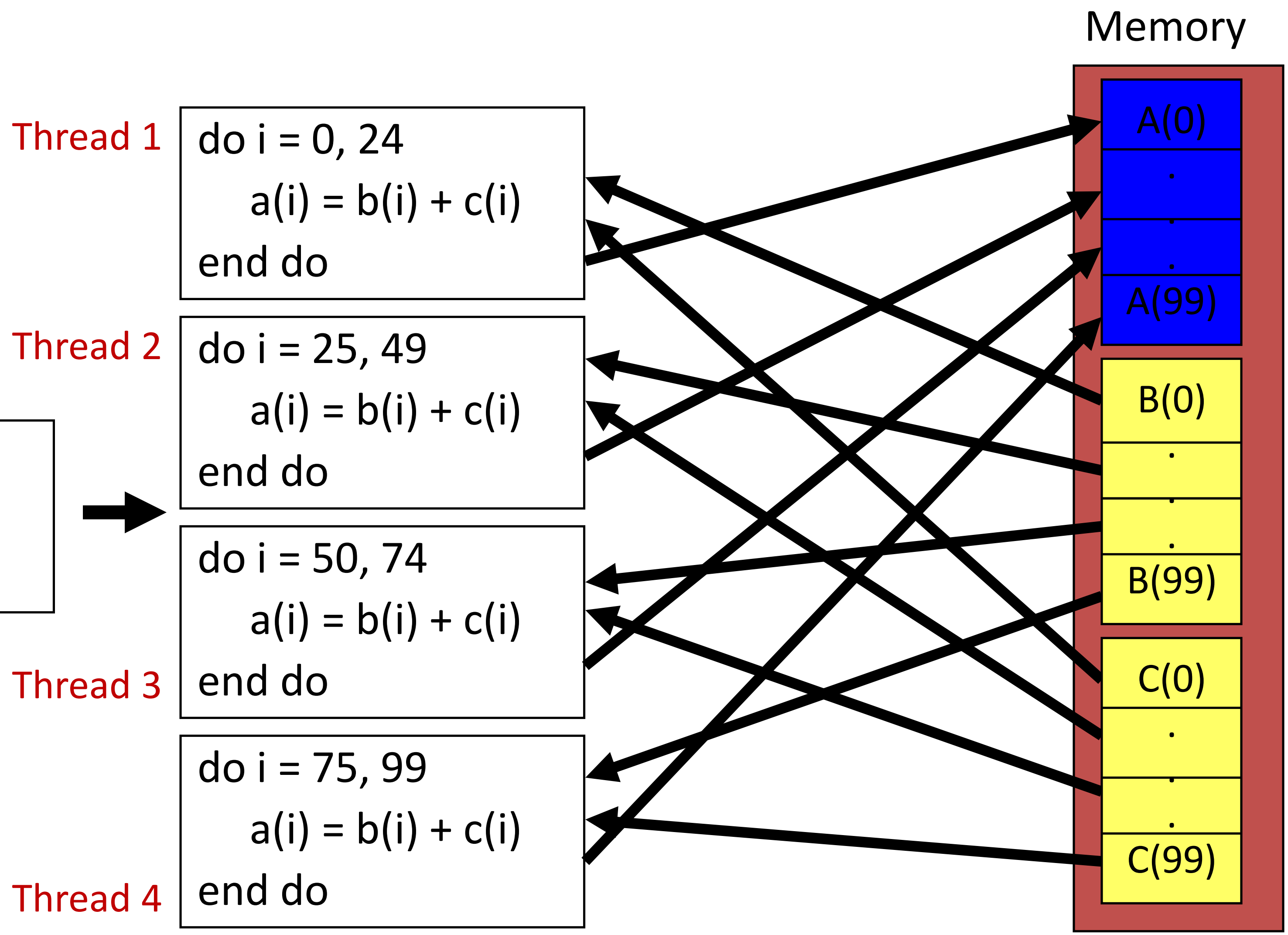
- **Loops often account for most of a program's runtime!**

Worksharing illustrated

Pseudo-Code
Here: 4 Threads

Serial

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



Vector Addition

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)

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),`

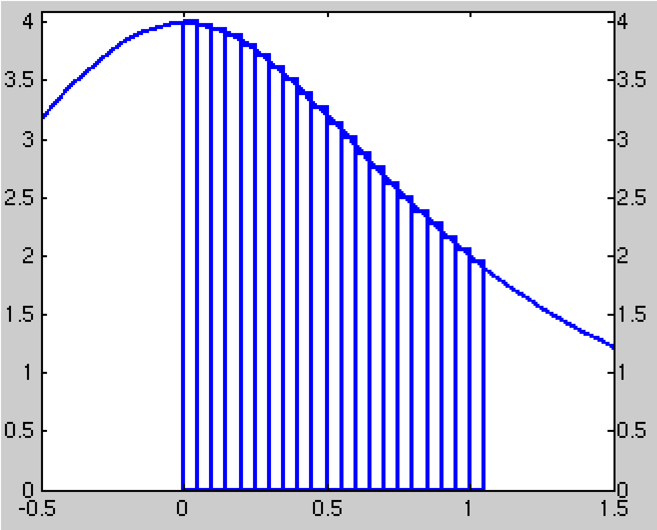
`^ (0), min (largest number), max (least number)`

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;
}
```

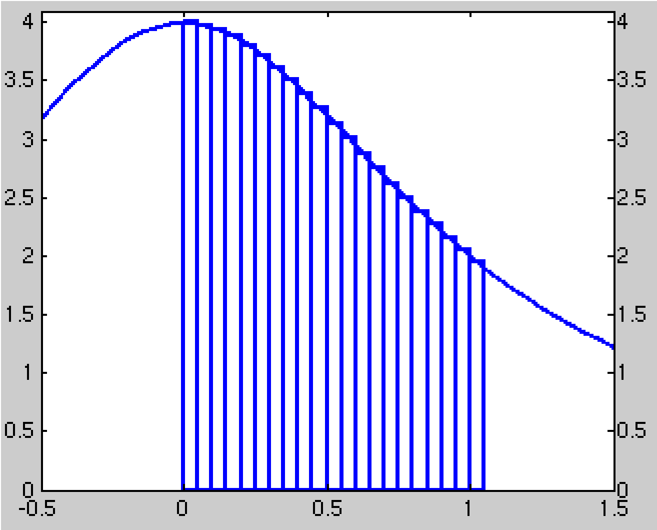
$$\pi = \int_0^1 \frac{4}{1+x^2}$$



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 ...

Correctness Checking Tools

Race Condition

- **Data Race: the typical OpenMP programming error, when:**
  - → two or more threads access the same memory location, and
  - → at least one of these accesses is a write, and
  - → the accesses are not protected by locks or critical regions, and
  - → the accesses are not synchronized, e.g. by a barrier.
- **Non-deterministic occurrence: e.g. the sequence of the execution of parallel loop iterations is non-deterministic and may change from run to run**
- **In many cases *private* clauses, *barriers* or *critical regions* are missing**
- **Data races are hard to find using a traditional debugger**
  - → Use the *Intel Inspector XE*

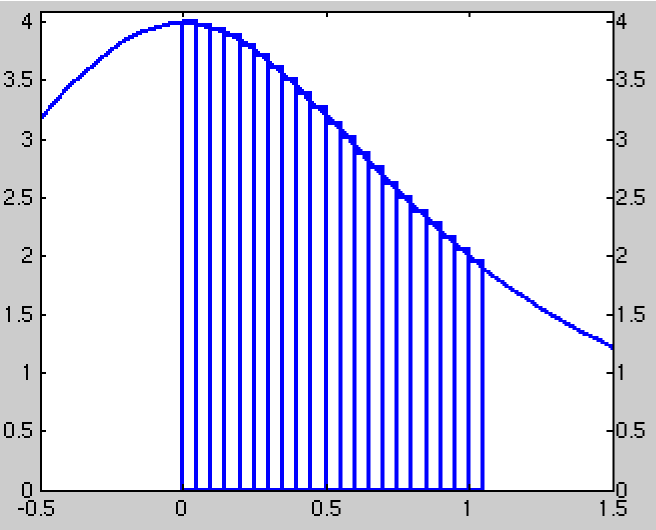
Intel Inspector XE

- **Detection of**
  - → Memory Errors
  - → Dead Locks
  - → Data Races
- **Support for**
  - → Linux (32bit and 64bit) and Windows (32bit and 64bit)
  - → WIN32-Threads, Posix-Threads, Intel Threading Building Blocks and OpenMP

PI Example Code

```
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;
}
```

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



PI Example Code

```
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;
}
```

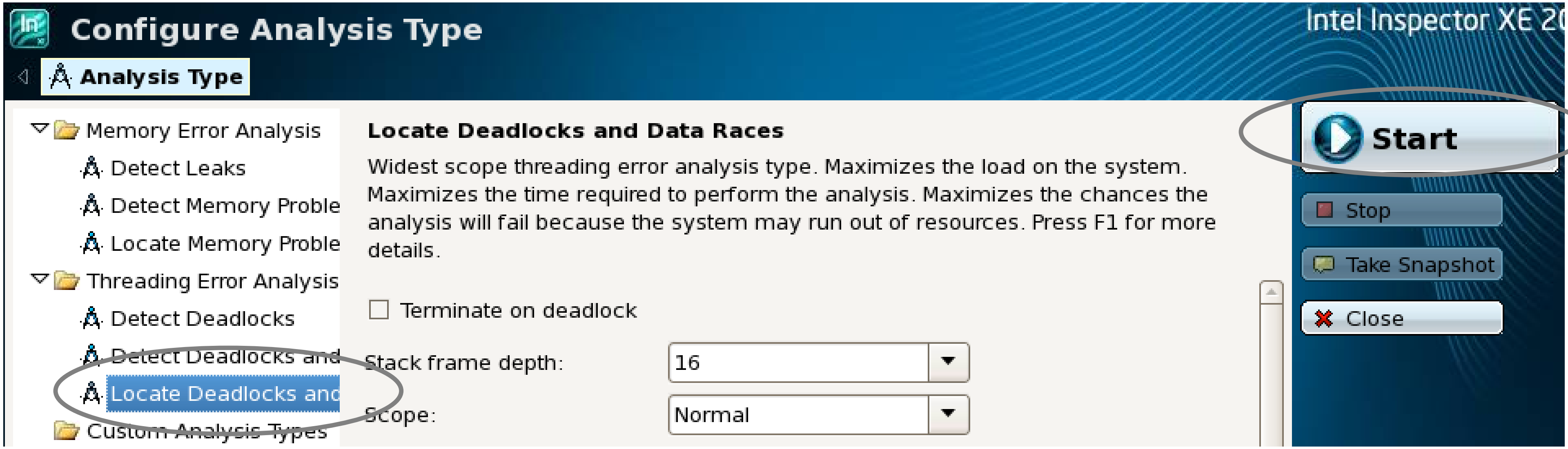
What if we would have forgotten this?

Inspector XE – Configure Analysis

Threading Error Analysis Modes

1. Detect Deadlocks
2. Detect Deadlocks and Data Races
3. Locate Deadlocks and Data Races

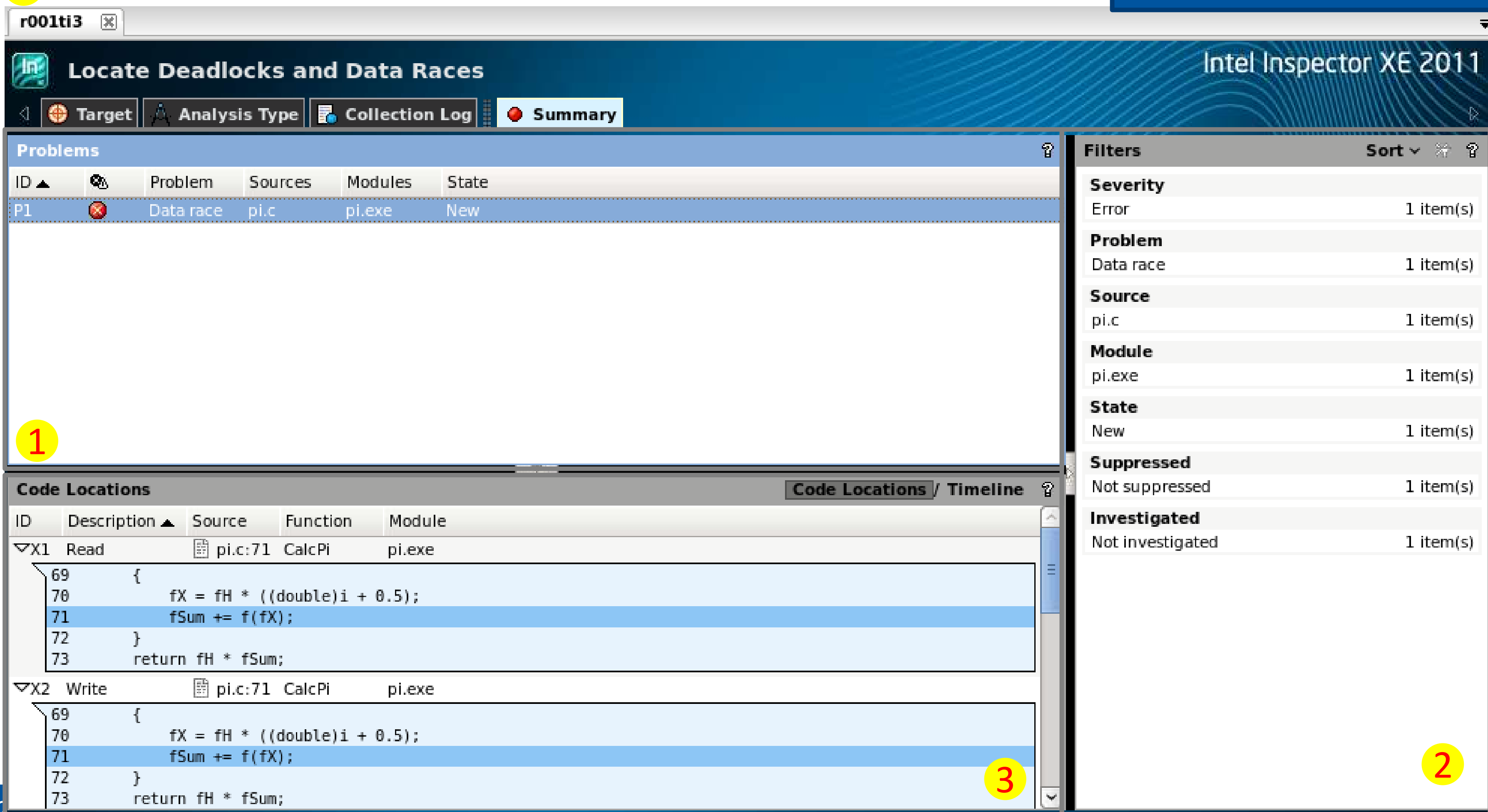
more details,
more overhead



Inspector XE – Results

1. detected problems
2. filters
3. code location

The missing reduction is detected.



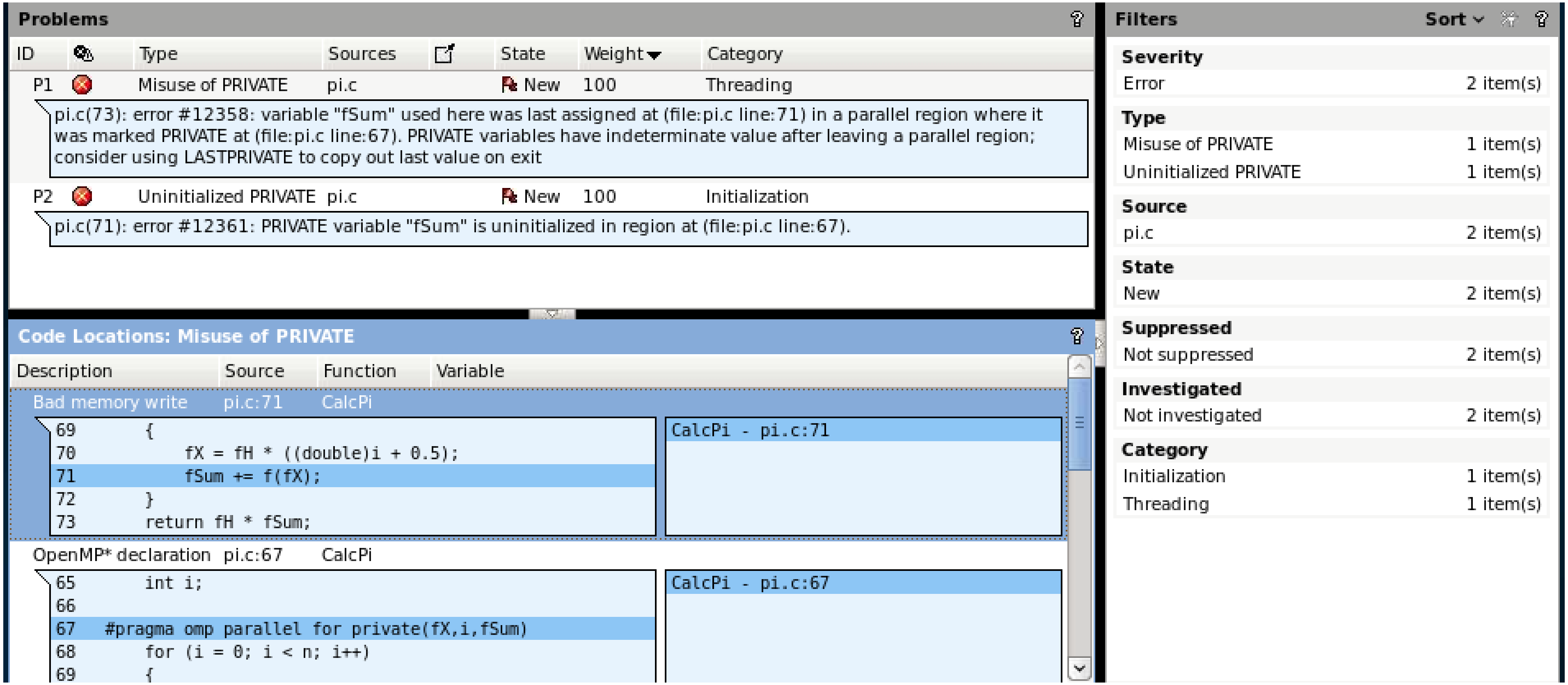
PI Example Code

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

What if we just made the variable private?

Inspector XE – Static Security Analysis

- At runtime no Error is detected!
- Compiling with the argument “-diag-enable sc-full” delivers:



- At compile-time this error can be found!

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!

We will later see how this can be done with tasks in a better way.

→ 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

User-defined Reductions

User Defined Reductions (UDRs)

- Use **`declare reduction`** directive to define operators
- **Operators used in reduction clause like predefined ops**

```
#pragma omp declare reduction (reduction-identifier :
typename-list : combiner) [initializer(initializer-expr)]
```

- **`reduction-identifier`** **gives a name to the operator**
  - → Can be overloaded for different types
  - → Can be redefined in inner scopes
- **`typename-list`** **is a list of types to which it applies**
- **`combiner`** **expression specifies how to combine values**
- **`initializer`** **specifies the operator's identity value**
  - → `initializer-expression` is an expression or a function

A simple UDR example

- **Declare the reduction operator**

```
#pragma omp declare reduction (merge : std::vector<int> :
   omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

- **Use the reduction operator in a `reduction` clause**

```
void schedule (std::vector<int> &v, std::vector<int> &filtered) {
  #pragma omp parallel for reduction (merge : filtered)
  for (std:vector<int>::iterator it = v.begin(); it < v.end();
it++)
    if ( filter(*it) )  filtered.push_back(*it);
```

- **Private copies created for a reduction are initialized to the identity that was specified for the operator and type**
  - → Default identity defined if `identity` clause not present
- **Compiler uses `combiner` to combine private copies**
  - → `omp_out` refers to private copy that holds combined value
  - → `omp_in` refers to the other private copy

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)

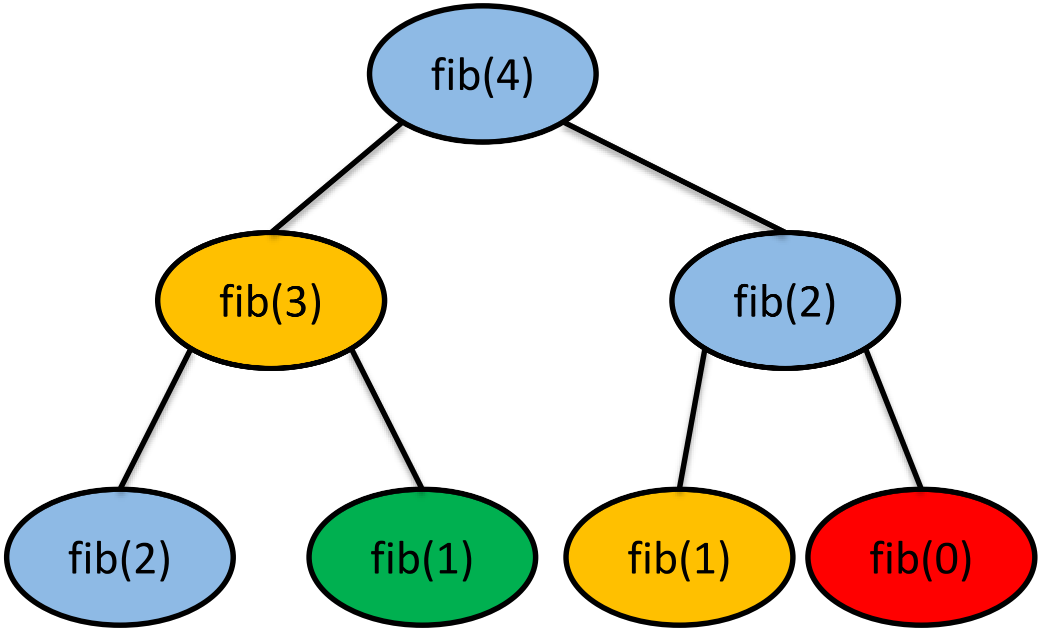
```
int main(int argc,
         char* argv[])
{
    [...]
    #pragma omp parallel
    {
        #pragma omp single
        {
            fib(input);
        }
    }
    [...]
}
```

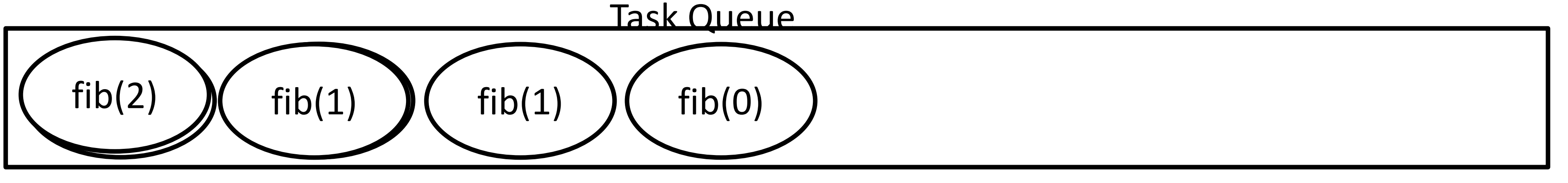
```
int fib(int n)   {
    if (n < 2) return n;
    int x, y;
    #pragma omp task shared(x)
    {
        x = fib(n - 1);
    }
    #pragma omp task shared(y)
    {
        y = fib(n - 2);
    }
    #pragma omp taskwait
        return x+y;
}
```

- **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

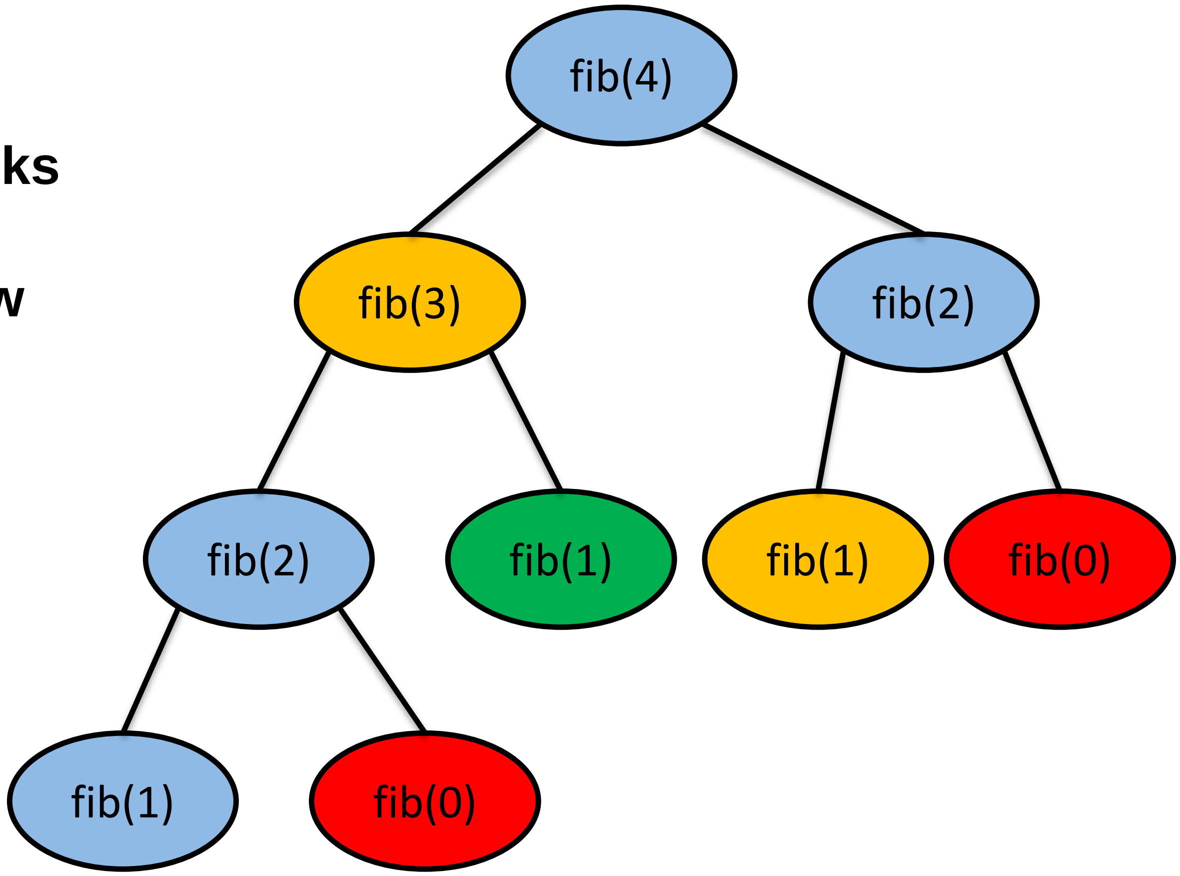
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks





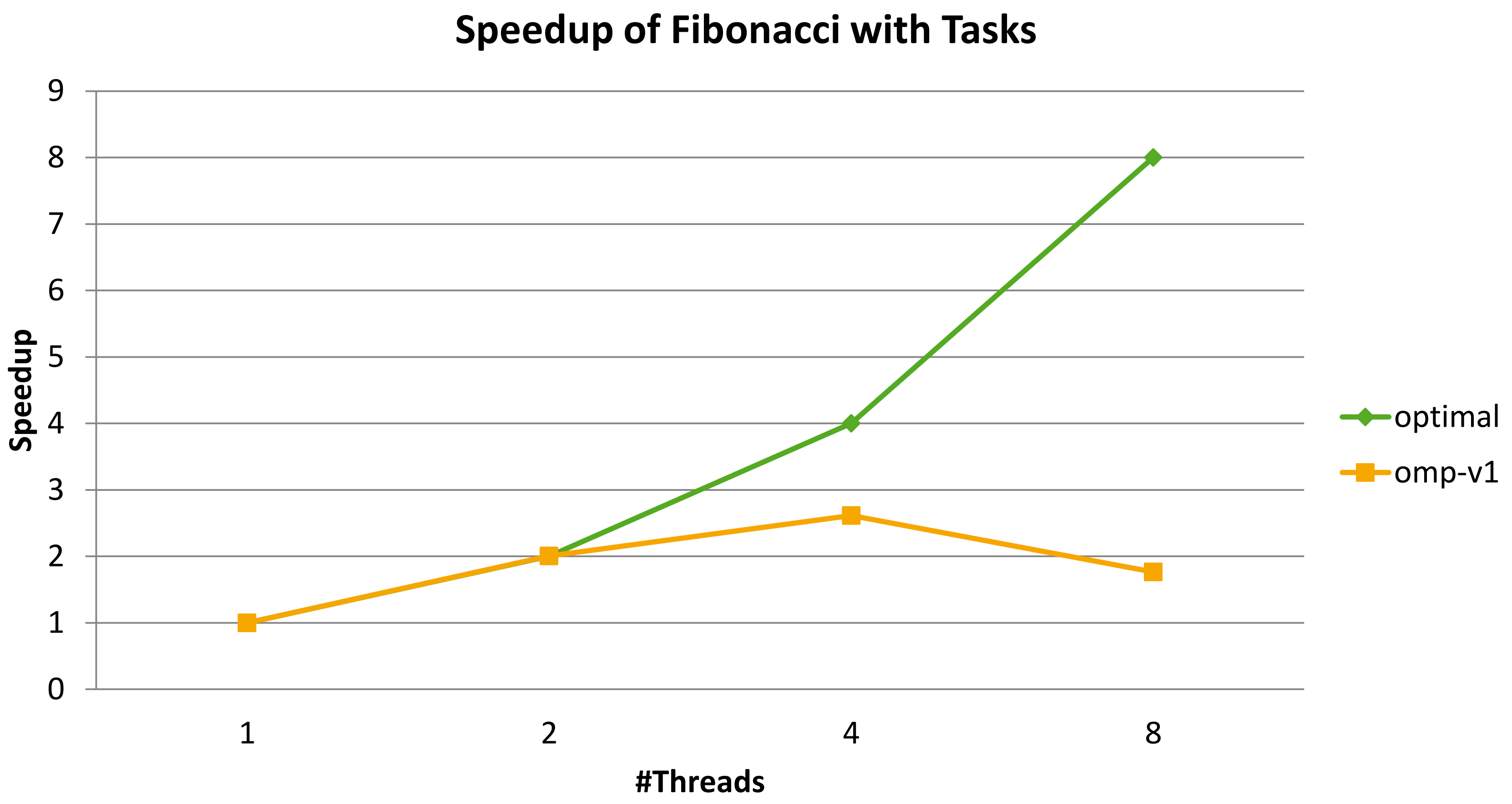
Fibonacci Illustration

- 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)

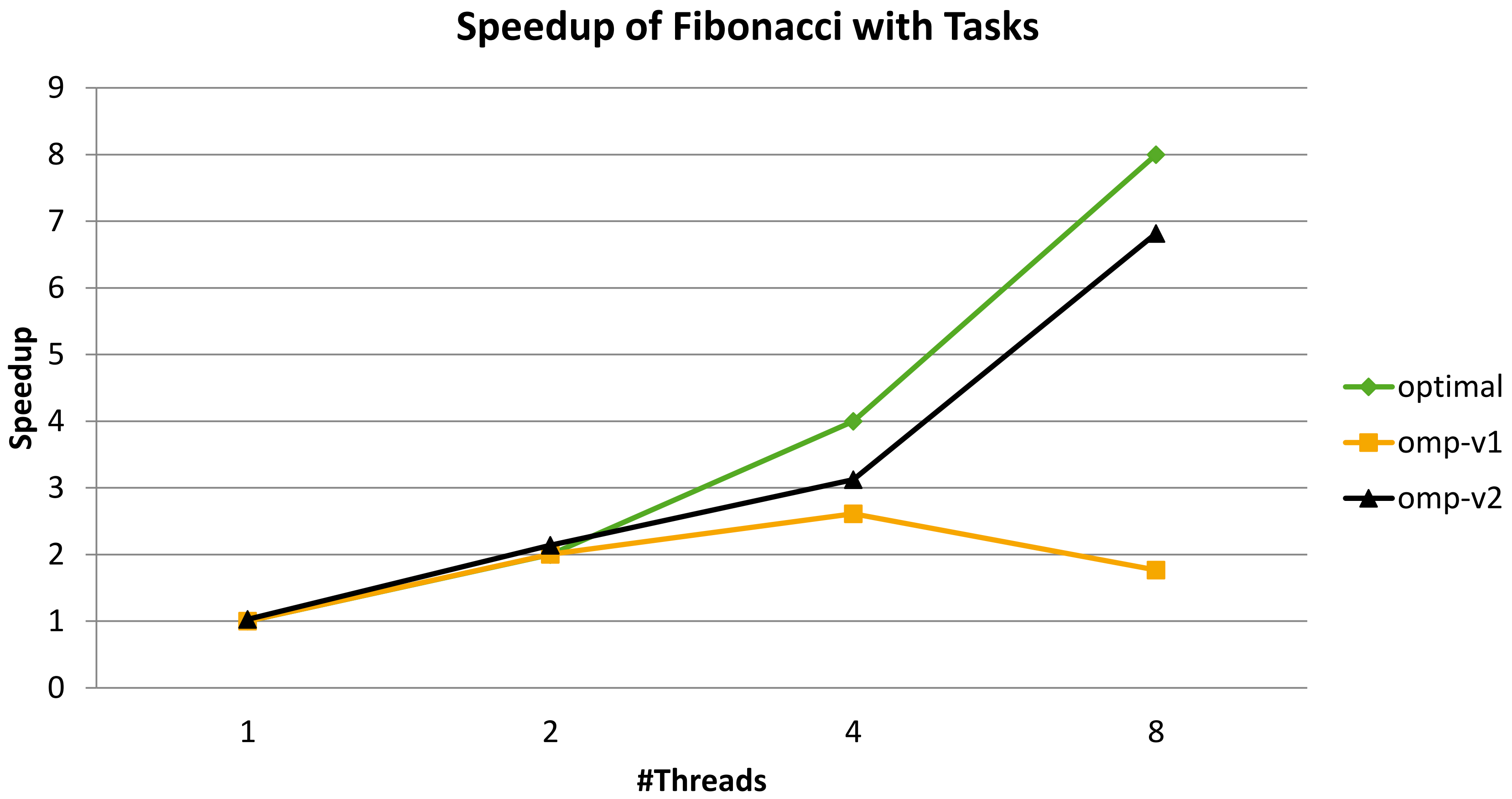
- **Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached**

```
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)

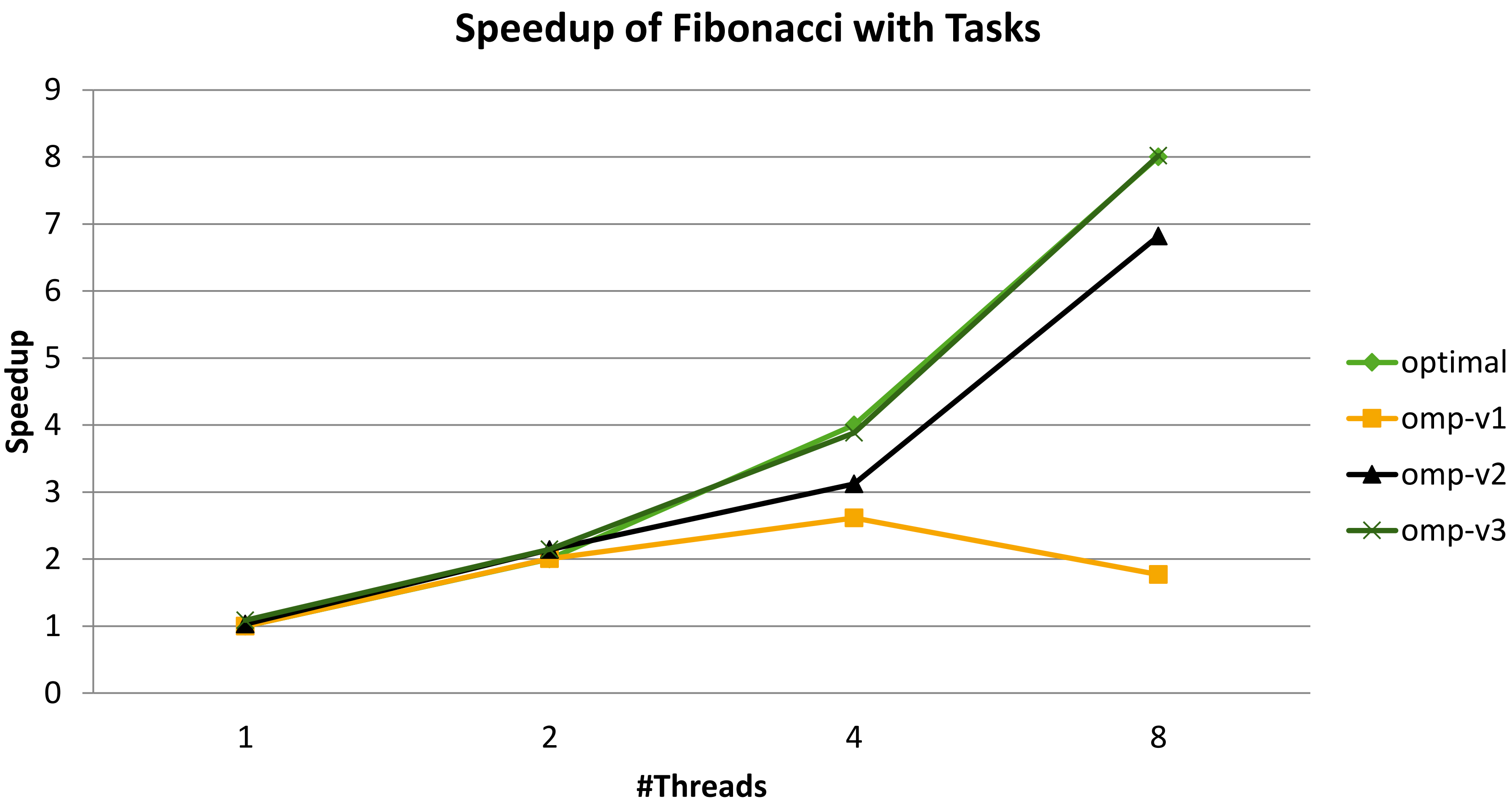
- **Improvement: Skip the OpenMP overhead once a certain `n` is reached (no issue w/ production compilers)**

```
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

More Environment Variables

OpenMP Environment Variables (1/2)

- **`OMP_NUM_THREADS`: Controls how many threads will be used to execute the program.**

- **`OMP_SCHEDULE`: If the schedule-type *runtime* is specified in a schedule clause, the value specified in this environment variable will be used.**

- **`OMP_DYNAMIC`: The OpenMP runtime is allowed to smartly guess how many threads might deliver the best performance. If you want full control, set this variable to *false*.**

- **`OMP_NESTED`: Most OpenMP implementations require this to be set to *true* in order to enabled nested Parallel Regions. Remember: Nesting Worksharing constructs is not possible.**

OpenMP Environment Variables (2/2)

- **Define interaction with system environment:**
  - → Env. Var. OMP_MAX_NESTED_LEVEL + API functions
    - → Controls the maximum number of active parallel regions
  - → Env. Var. OMP_THREAD_LIMIT + API functions
    - → Controls the maximum number of OpenMP threads
  - → Env. Var. OMP_STACKSIZE
    - → Controls the stack size of child threads
  - → Env. Var. OMP_WAIT_POLICY
    - → Control the thread idle policy:
      - → active: Good for dedicated systems (e.g. in batch mode)
      - → passive: Good for shared systems

Questions?