RWTH AACHEN UNIVERSITY

# Introduction to OpenMP

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

10.04.2013 / Darmstadt, Germany

Stand: 06.03.2013

Version 2.3

# 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
- 11/2012: OpenMP 4.0 RC1 + TR1
- 03/2013: OpenMP 4.0 RC2
- 05/2013: OpenMP 4.0 to be released


http://www.OpenMP.org

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

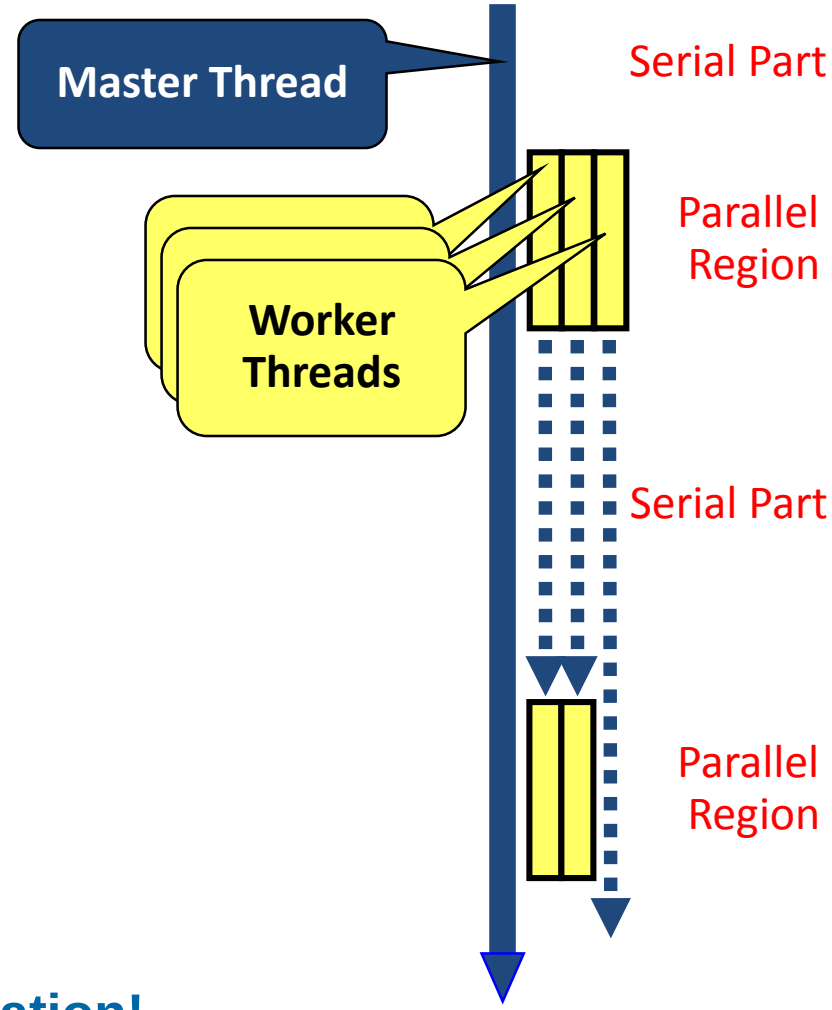
# Agenda

- Basic Concept: *Parallel Region*
- The *For* Construct
- The *Single* Construct
- The *Task* Construct
- *Scoping*: Managing the Data Environment
- The *Synchronization* and *Reduction* Constructs
- Runtime Library

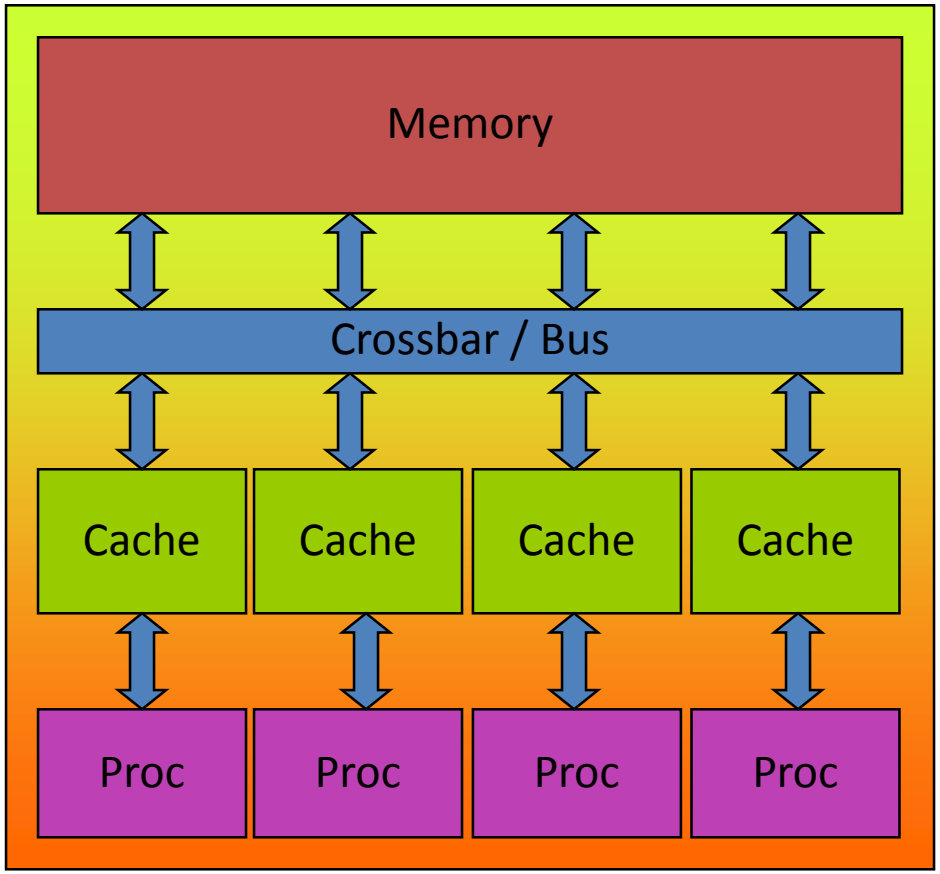
# Parallel Region

# 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!



- 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.

# 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

# 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
```

- Intel Compiler on Linux: asking for more information:

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

# Hello orphaned World

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

Fortran

```
INTEGER :: i
!$omp parallel 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

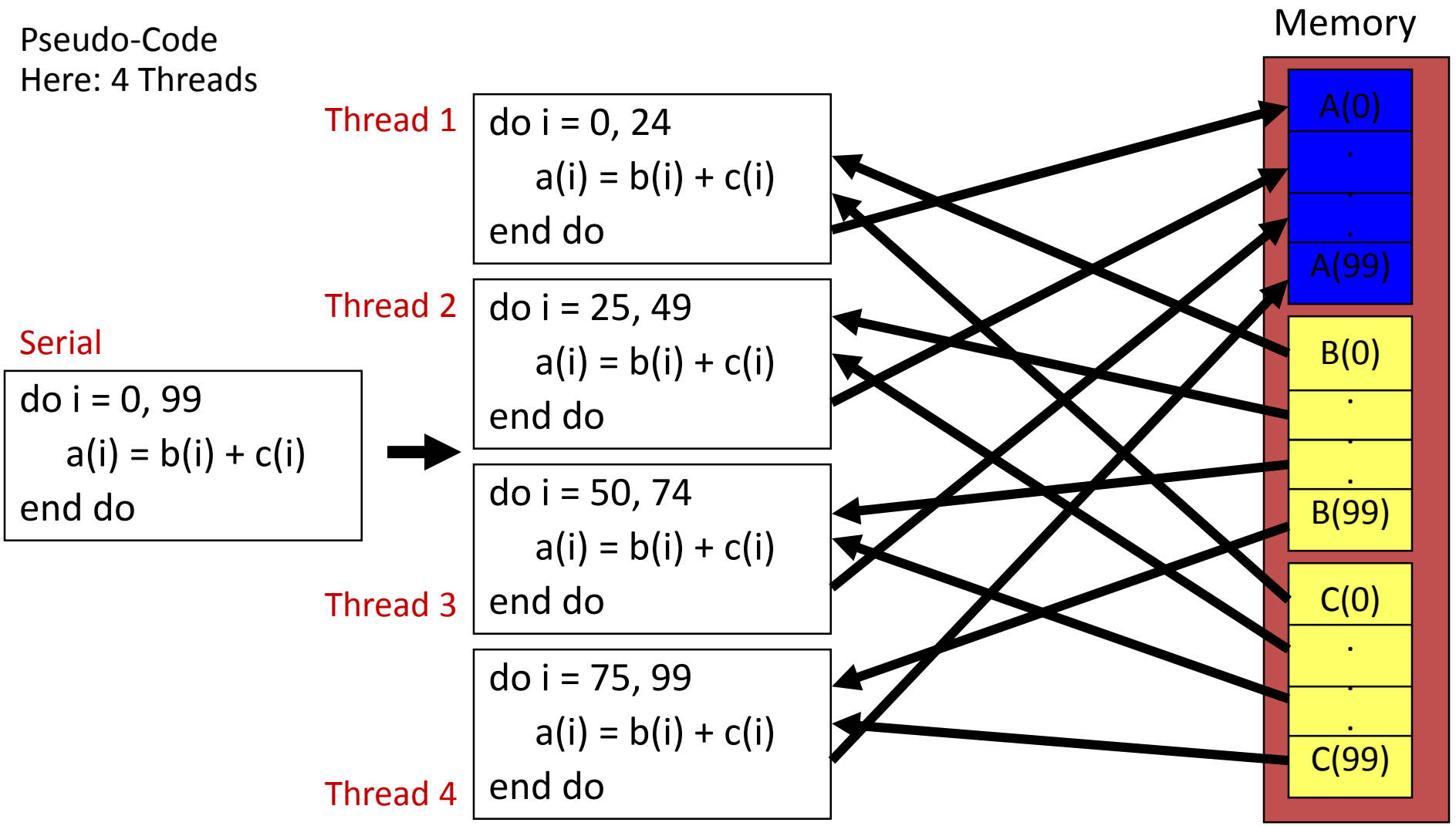
```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

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

# The Single 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…

# Task Construct

# How to parallelize a While-loop?

- **Can we parallelize this code with the For-Worksharing construct?**

```
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++;
}
```

- **No.**
- **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!

RWTH AACHEN UNIVERSITY

- **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++;
   }
}
}
```

This structured block will be executed by just one thread, the other threads will skip the block and jump right to it's end.

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

# 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*)

# 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)

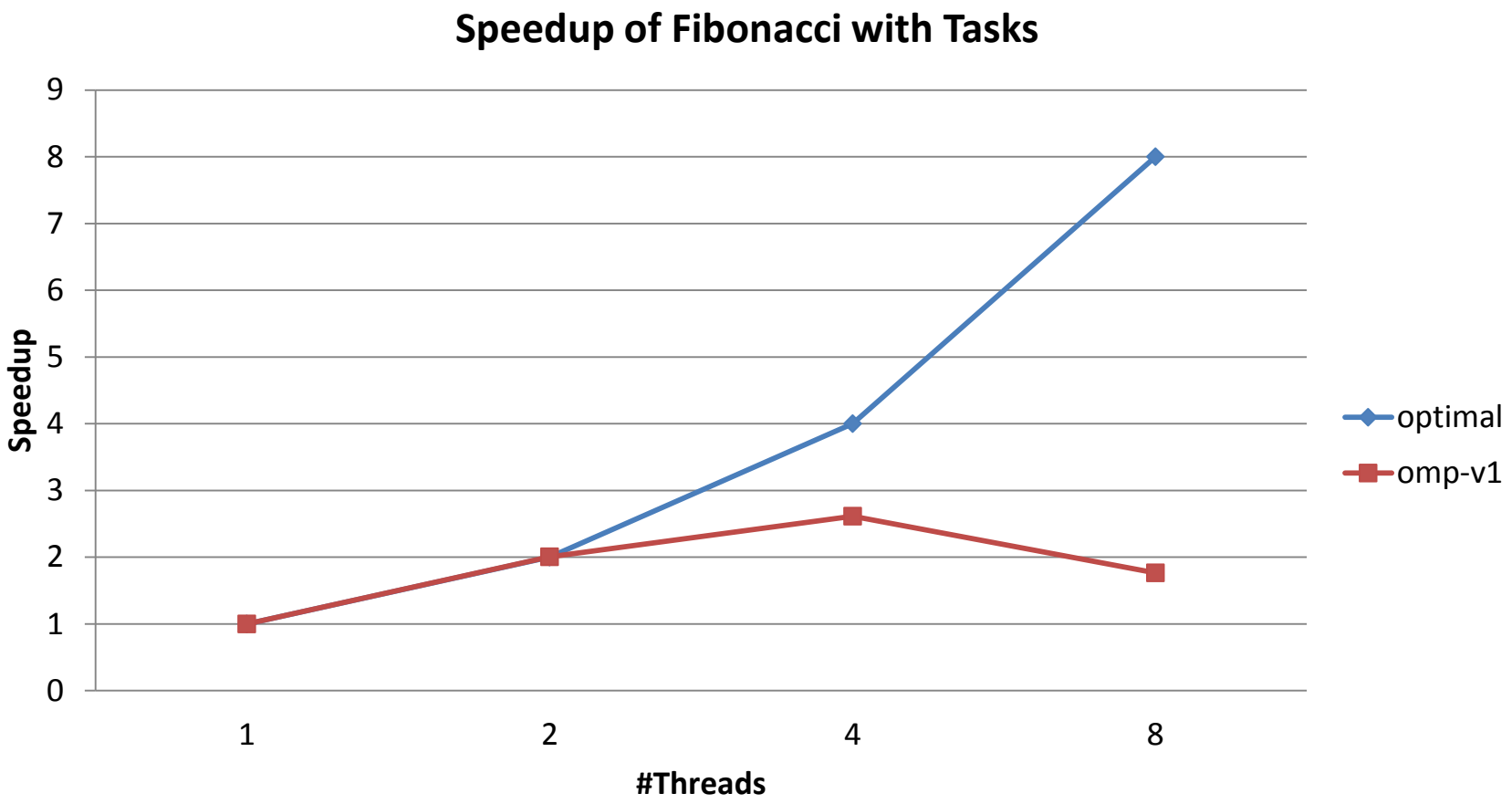
```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
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)

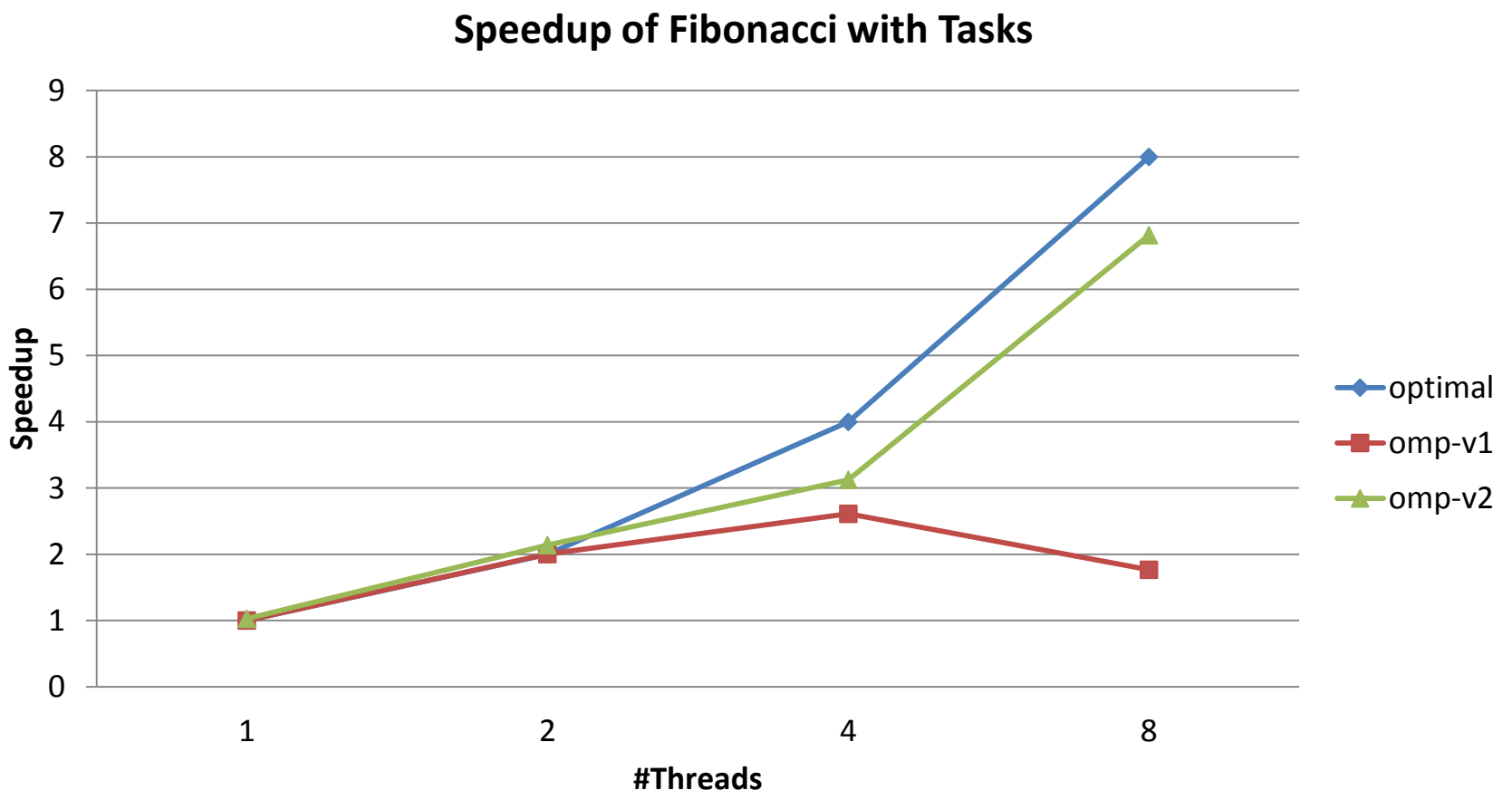
- **Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached**

```
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)

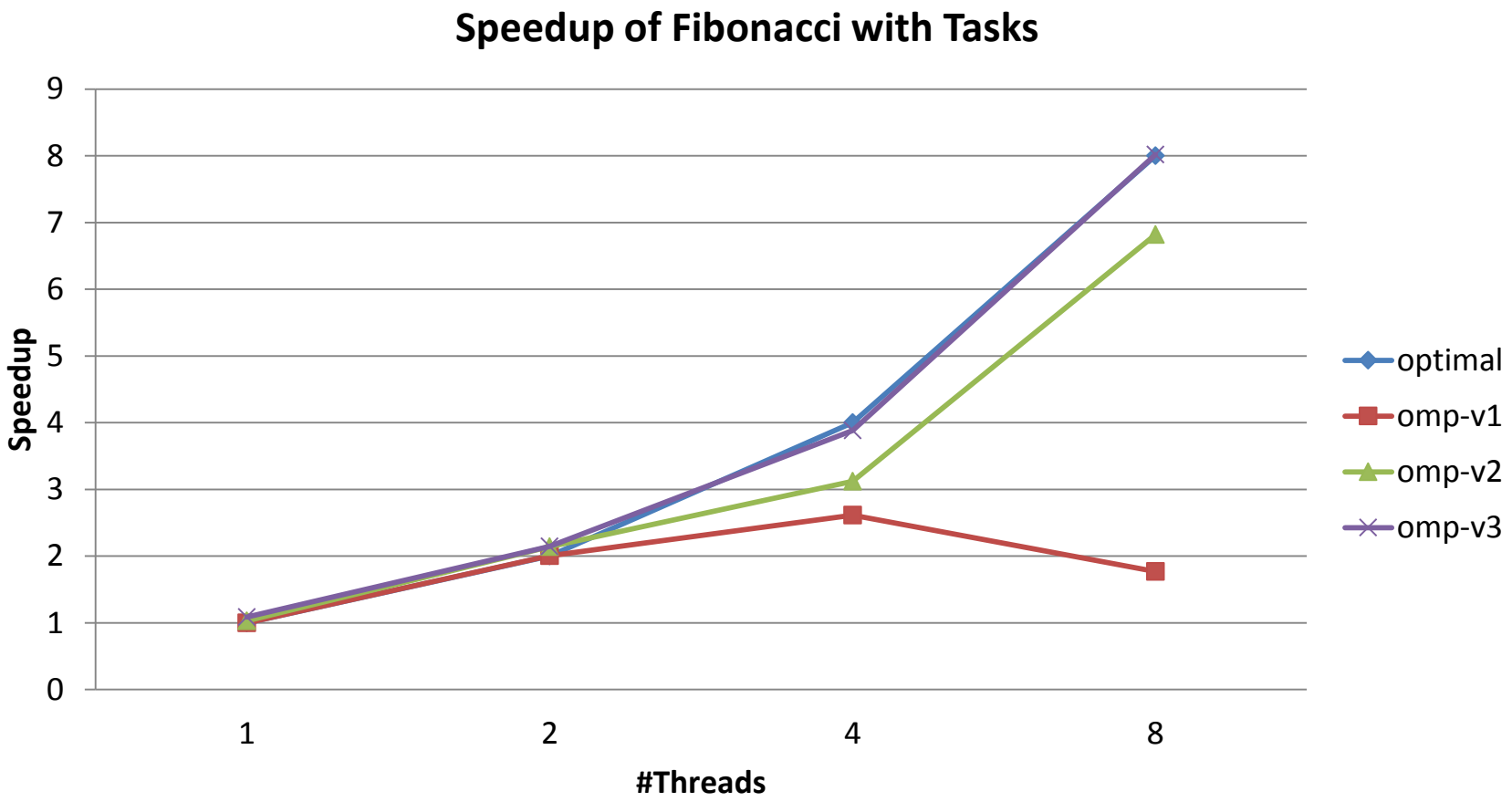
- **Improvement: Skip the OpenMP overhead once a certain `n` is reached (no issue w/ production compilers)**

```
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 ☺



# 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*, *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)
```

# 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
- **So far no verification tool is available to check Tasking programs for correctness!**

# Data Scoping with Tasking

# Data Scoping Example (1/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a:
               // Scope of b:
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

# Data Scoping Example (2/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a: shared
               // Scope of b:
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

# Data Scoping Example (3/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // Scope of a: shared
               // Scope of b: firstprivate
               // Scope of c:
               // Scope of d:
               // Scope of e:
} } }
```

# Data Scoping Example (4/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // 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;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
                int e;

                // 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;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
                int e;

                // Scope of a: shared
                // Scope of b: firstprivate
                // Scope of c: shared
                // Scope of d: firstprivate
                // Scope of e: private
} } }
```

# Data Scoping Example (7/7)

```
int a;
void foo()
{
   int b, c;
   #pragma omp parallel shared(b)
   #pragma omp parallel private(b)
   {
        int d;
        #pragma omp task
        {
               int e;

               // 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.

# Synchronization

- **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: 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;
#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++

```
#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)`

# 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
```

# Tasks in OpenMP: Scheduling

- **Default: Tasks are *tied* to the thread that first executes them → not neccessarily the creator. Scheduling constraints:**
  - Only the Thread a Task is tied to can execute it
  - A Task can only be suspended at a suspend point
    - Task creation, Task finish, `taskwait`, `barrier`
  - If Task is not suspended in a barrier, executing Thread can only switch to a direct descendant of all Tasks tied to the Thread
- **Tasks created with the `untied` clause are never tied**
  - No scheduling restrictions, e.g. can be suspended at any point
  - But: More freedom to the implementation, e.g. load balancing

# Task Synchronization

RWTH AACHEN UNIVERSITY

## 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

# Unsafe use of untied Tasks

- **Problem: Because tasks can migrate between threads at any point, thread-centric constructs can yield unexpected results**

- **Remember when using `untied` tasks:**
  - Avoid `threadprivate` variable
  - Avoid and use of thread-ids (i.e. `omp_get_thread_num()`)
  - Be careful with `critical region` and *locks*

- **Simple Solution:**
  - Create a tied task region with

```
#pragma omp task if(0)
```

# PI

# Example: Pi (1/2)

- **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;

#pragma omp parallel for reduction(+:sum) private(x)
   for (int i = 1; i <= iNumIntervals; i++) {
      x = h * ((double)i - (double)0.5);
      sum += f(x);
   }

   myPi = h * sum;
}
```

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

# Example: Pi (1/2)

- **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;

#pragma omp parallel for private(x) reduction(+:sum)
   for (int i = 1; i <= iNumIntervals; i++) {
      x = h * ((double)i - (double)0.5);
      sum += f(x);
   }

   myPi = h * sum;
}
```

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

# Example: Pi (2/2)

RWTH AACHEN UNIVERSITY

- **Results (with C++ version):**

| # 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 trival fashion ...

# 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.**

# The End

**Thank you for your attention.**

# Appendix A: make/gmake

RWTH AACHEN UNIVERSITY

- **make: “smart” utility to manage compilation of programs and much more**
  - automatically detects which parts need to be rebuild
  - general rules for compilation of many files
  - dependencies between files can be handled
- **Usage:**

```
make <target>  or  gmake <target>
```

- **Rules:**

```
target ... : prerequisites ...
        < tab > command
        < tab > ...
```

  - target: output file (or only a name)
  - prerequisites: input files (e.g. source code files)
  - command: action to be performed