Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → Basic concept
   - → Scoping
   - → Synchronization
   - → Correctness Checking Tools
   - → Runtime Library
   - → Tasking
   - → Load Balancing
   - → False Sharing
   - → NUMA Architectures
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

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**

OpenMP™

http://www.OpenMP.org

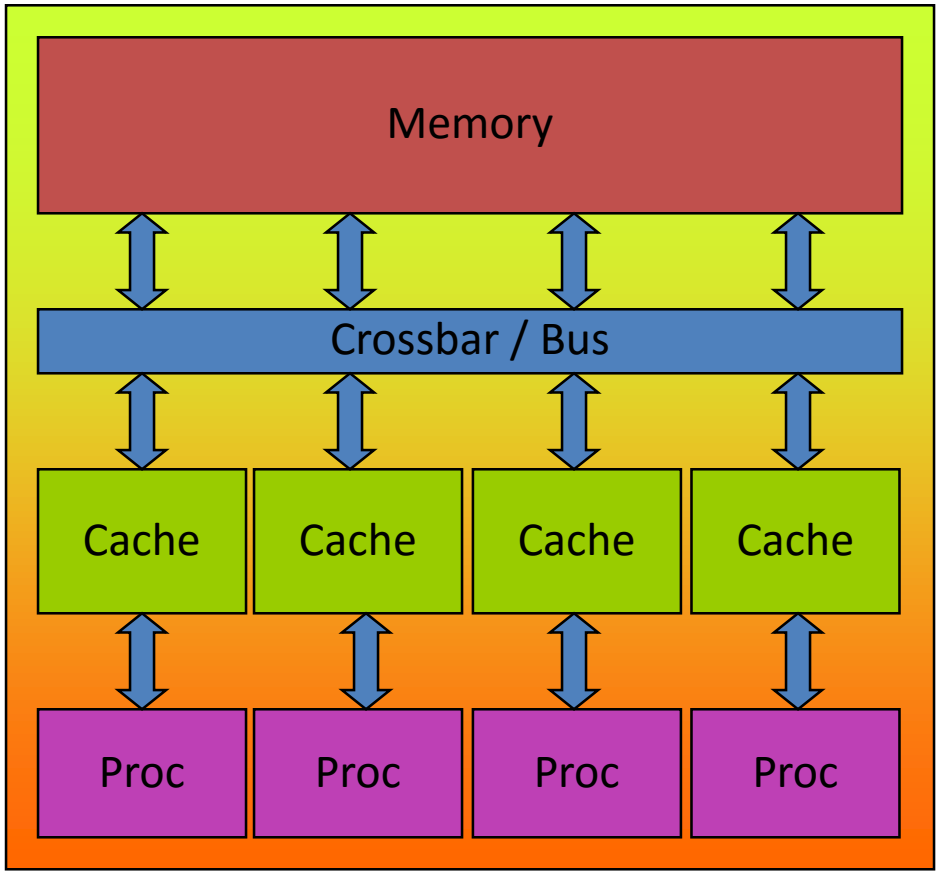
RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006.

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → **Basic concept**
   - → Scoping
   - → Synchronization
   - → Correctness Checking Tools
   - → Runtime Library
   - → Tasking
   - → Load Balancing
   - → False Sharing
   - → NUMA Architectures
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

OpenMP‘s machine model

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



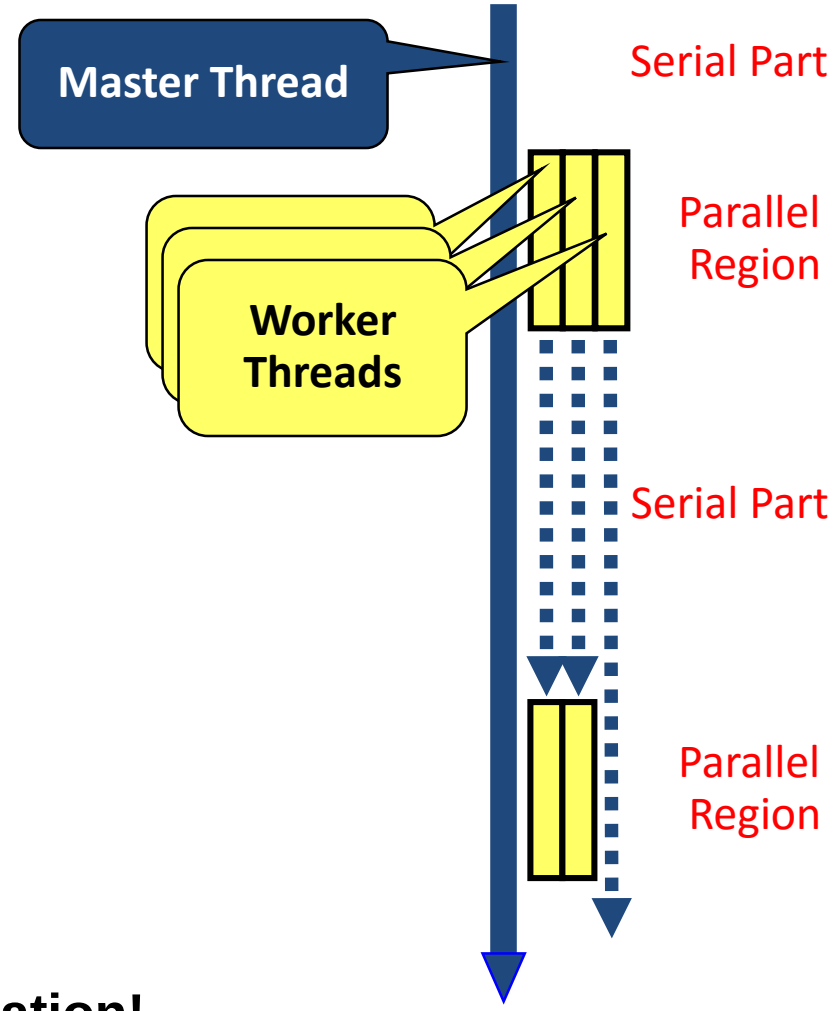
All processors/cores access a shared main memory.

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

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

- **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

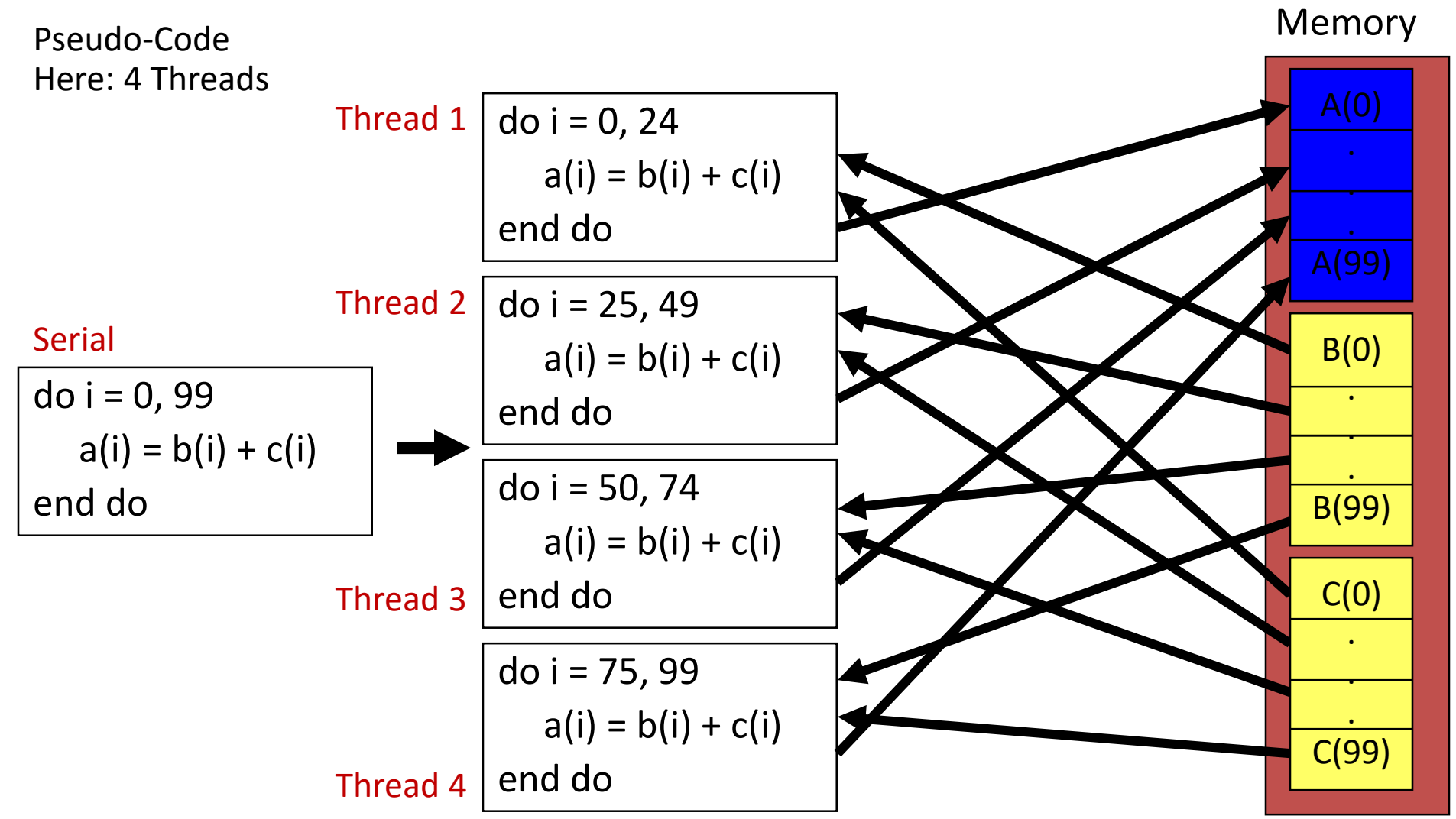
```
do i = 25, 49
    a(i) = b(i) + c(i)
end do
```

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
```



Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → Basic concept
   - → **Scoping**
   - → Synchronization
   - → Correctness Checking Tools
   - → Runtime Library
   - → Tasking
   - → Load Balancing
   - → False Sharing
   - → NUMA Architectures
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

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)
```

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

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → Basic concept
   - → Scoping
   - → **Synchronization**
   - → Correctness Checking Tools
   - → Runtime Library
   - → Tasking
   - → Load Balancing
   - → False Sharing
   - → NUMA Architectures
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Synchronization Overview

- **Example: Summing up Vector Elements**
- **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];  }
}
```

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;
#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++

```
#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),`

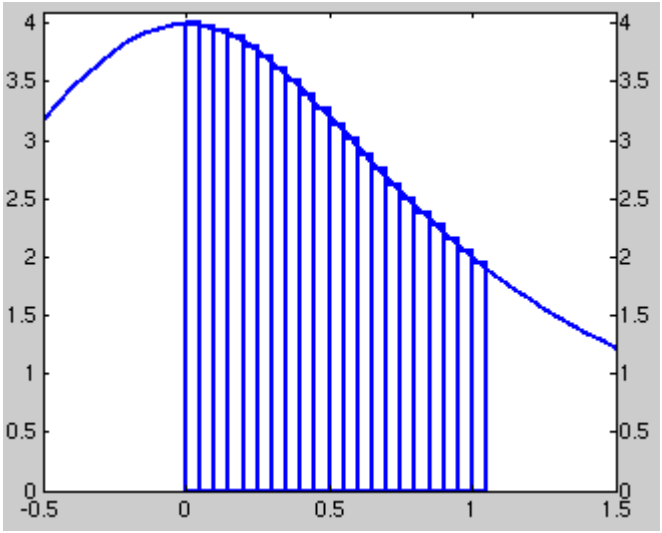
`& (~0), | (0), && (1), || (0),`

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

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;
}
```

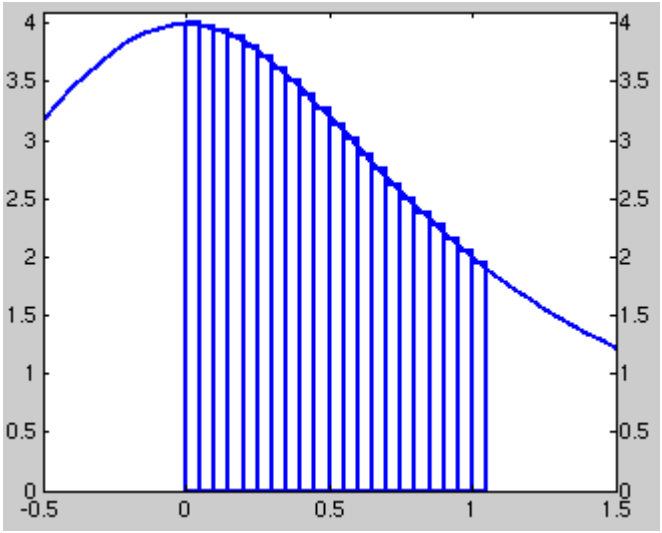
$$\pi = \int_0^1 \frac{4}{1+x^2}$$



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 ...

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 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…

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.**

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

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → Basic concept
   - → Scoping
   - → Synchronization
   - → **Correctness Checking Tools**
   - → Runtime Library
   - → Tasking
   - → Load Balancing
   - → False Sharing
   - → NUMA Architectures
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

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*

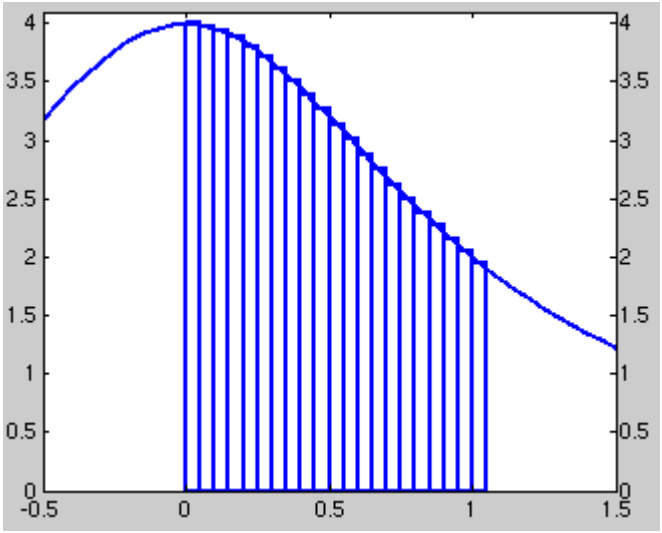
Intel Inspector XE

- **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;
}
```

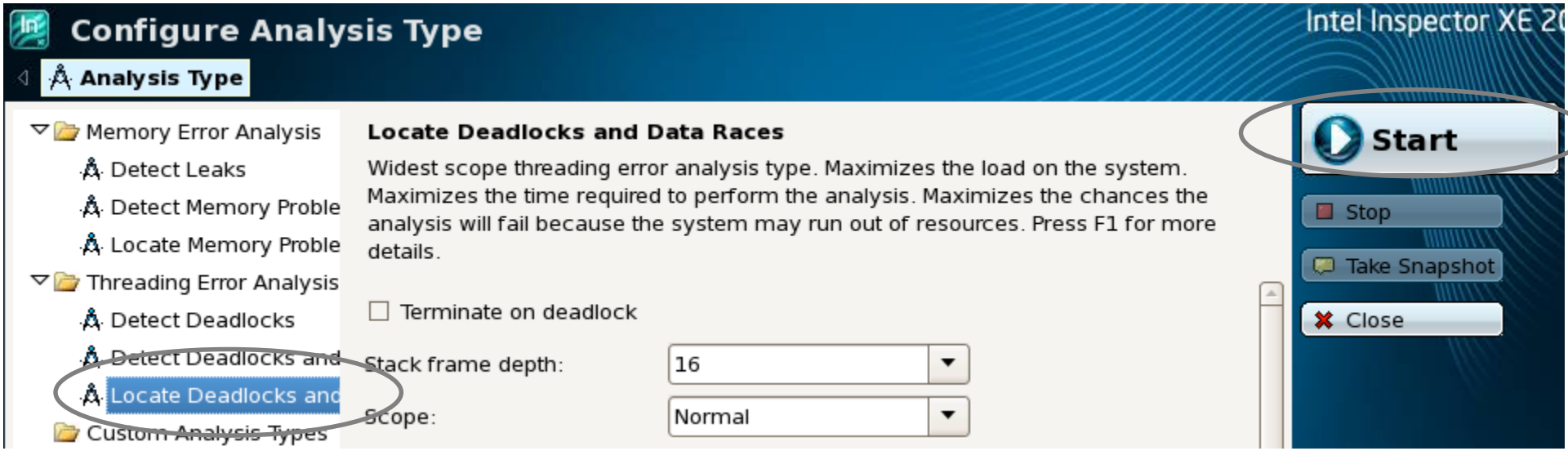
What if we would have forgotten this?

Inspector XE – Configure Analysis

Threading Error Analysis Modes

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

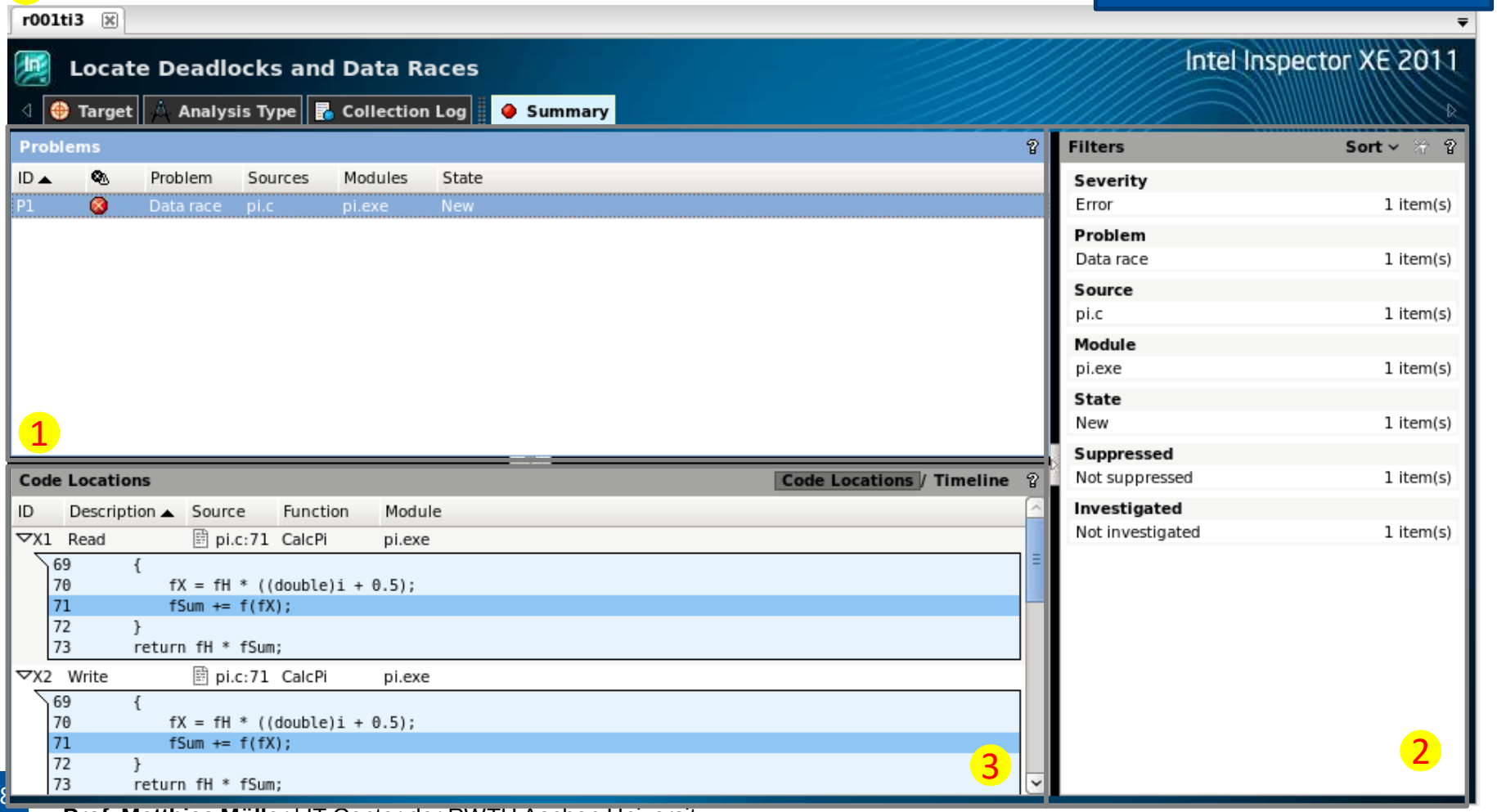
more details,
more overhead



Inspector XE – Results

1. detected problems
2. filters
3. code location

The missing reduction is detected.



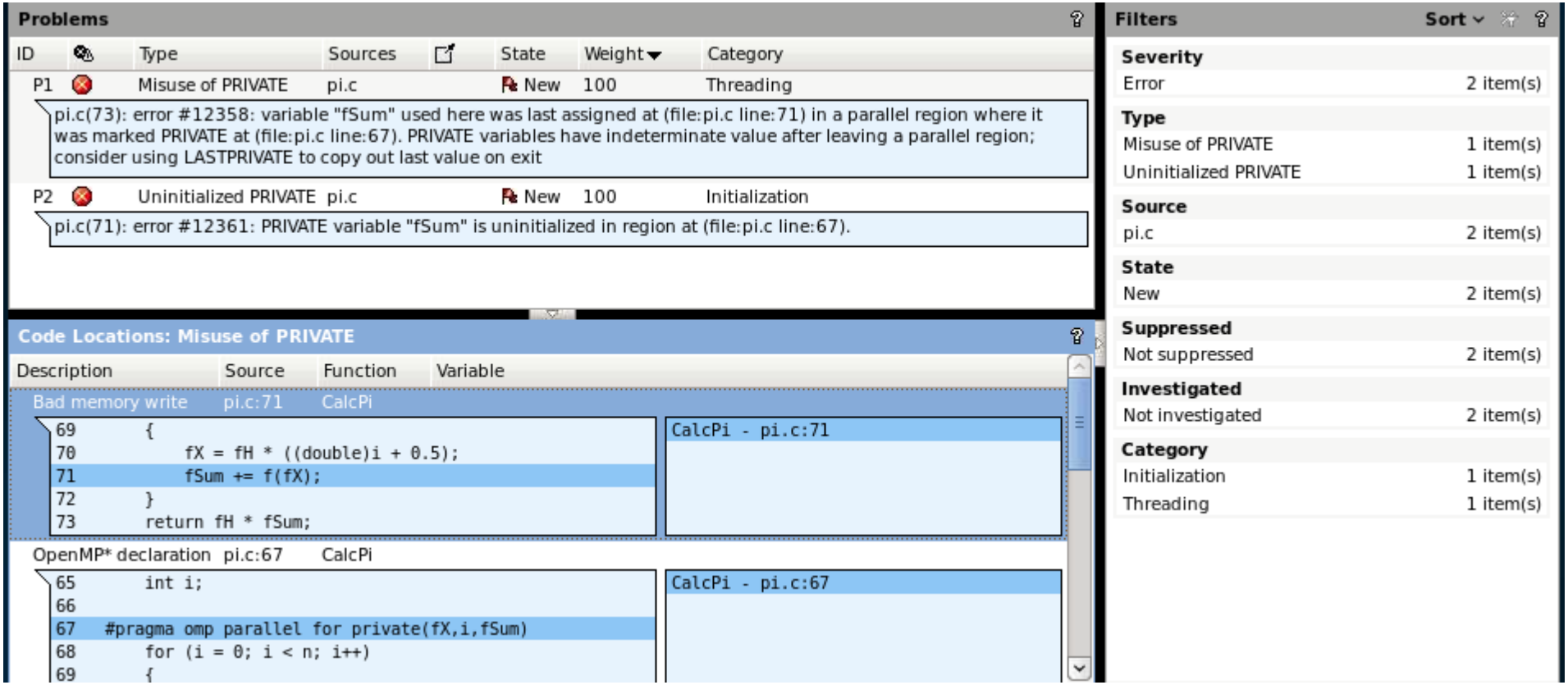
PI Example Code

```
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!

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → Basic concept
   - → Scoping
   - → Synchronization
   - → Correctness Checking Tools
   - → **Runtime Library**
   - → Tasking
   - → Load Balancing
   - → False Sharing
   - → NUMA Architectures
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

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.**

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → Basic concept
   - → Scoping
   - → Synchronization
   - → Correctness Checking Tools
   - → Runtime Library
   - → **Tasking**
   - → Load Balancing
   - → False Sharing
   - → NUMA Architectures
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

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)

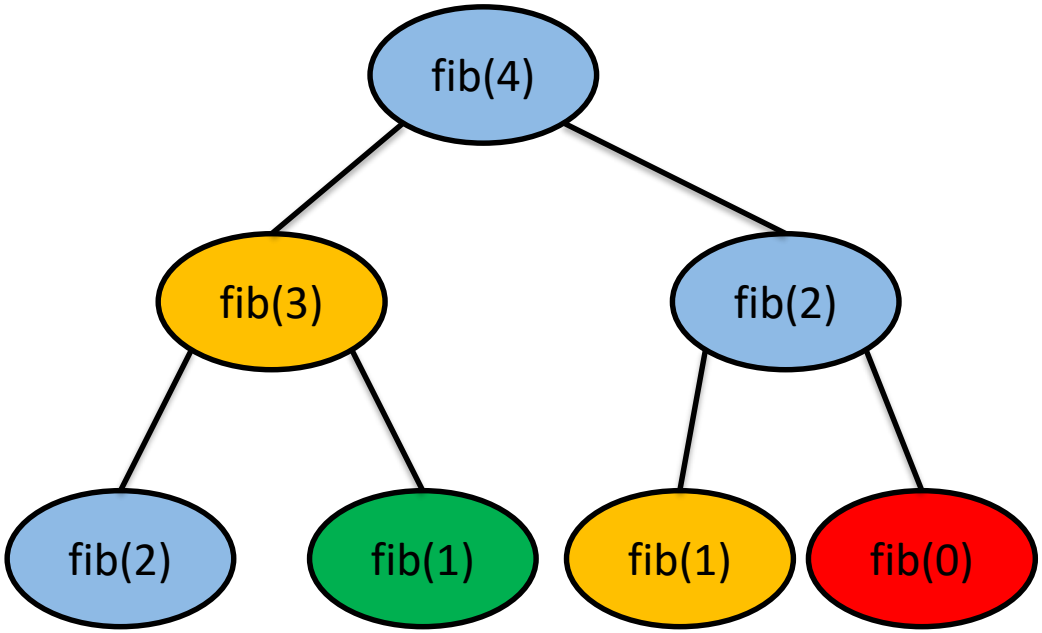
```
int main(int argc,
         char* argv[])
{
   [...]
   #pragma omp parallel
   {
        #pragma omp single
        {
                fib(input);
        }
   }
   [...]
}
```

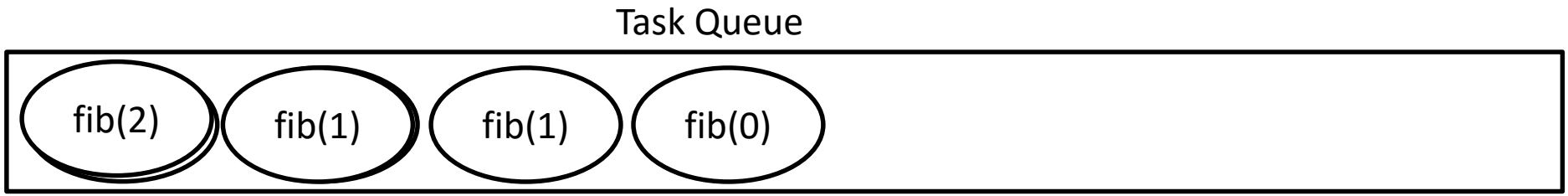
```
int fib(int n)   {
   if (n < 2) return n;
   int x, y;
   #pragma omp task shared(x)
   {
        x = fib(n - 1);
   }
   #pragma omp task shared(y)
   {
        y = fib(n - 2);
   }
   #pragma omp taskwait
        return x+y;
   }
```

- 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

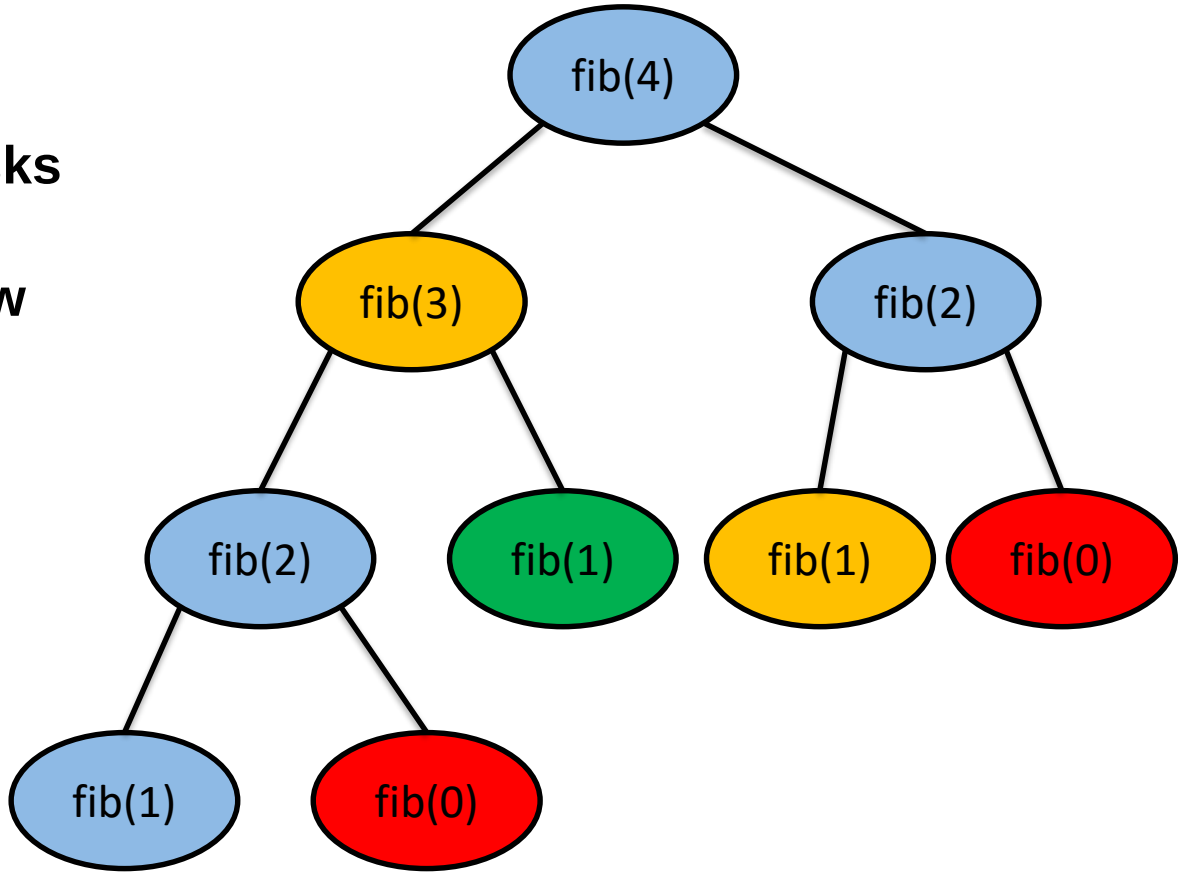
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks





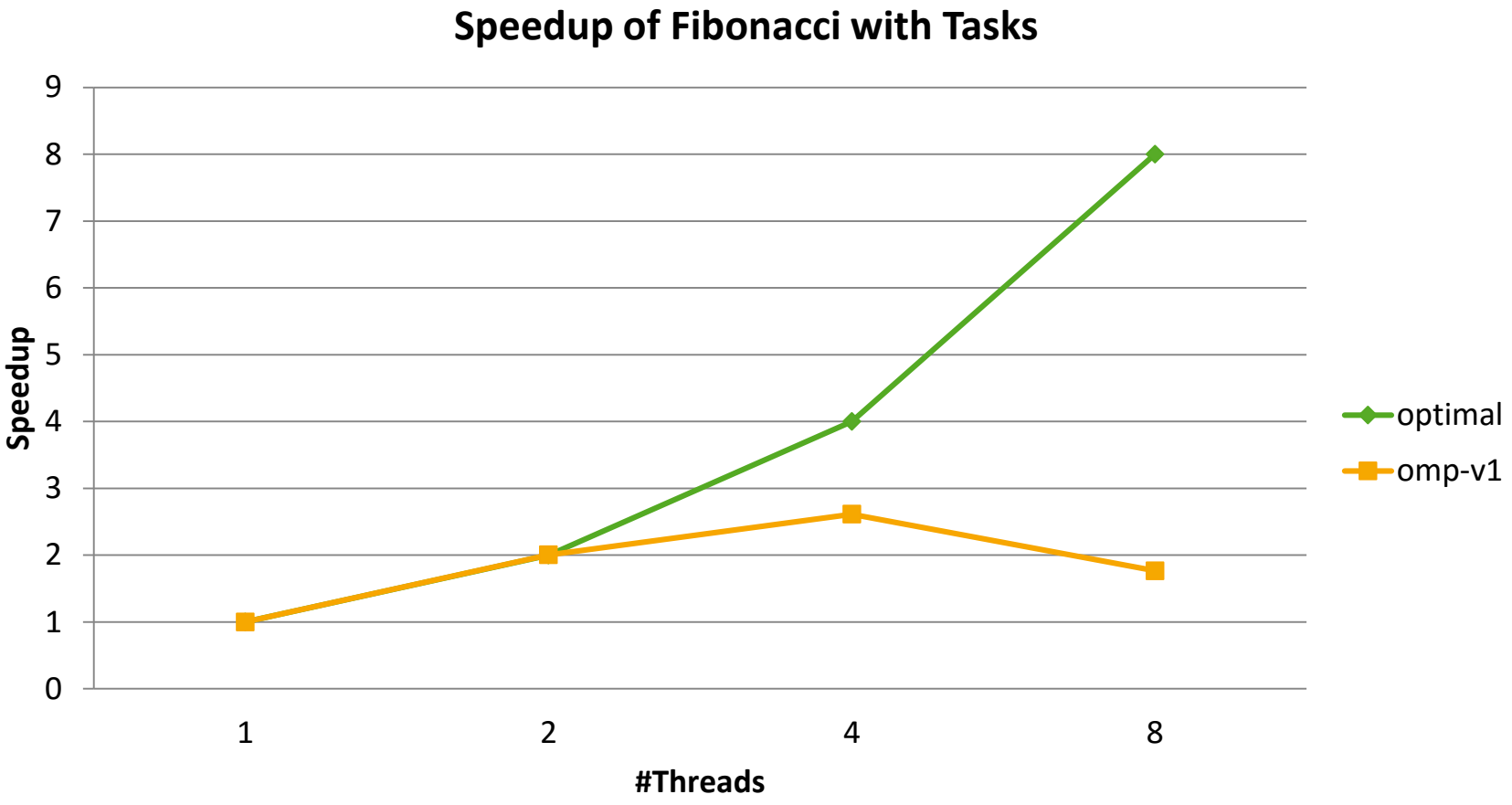
Fibonacci Illustration

- 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!**



Improved parallelization with Tasking (omp-v2)

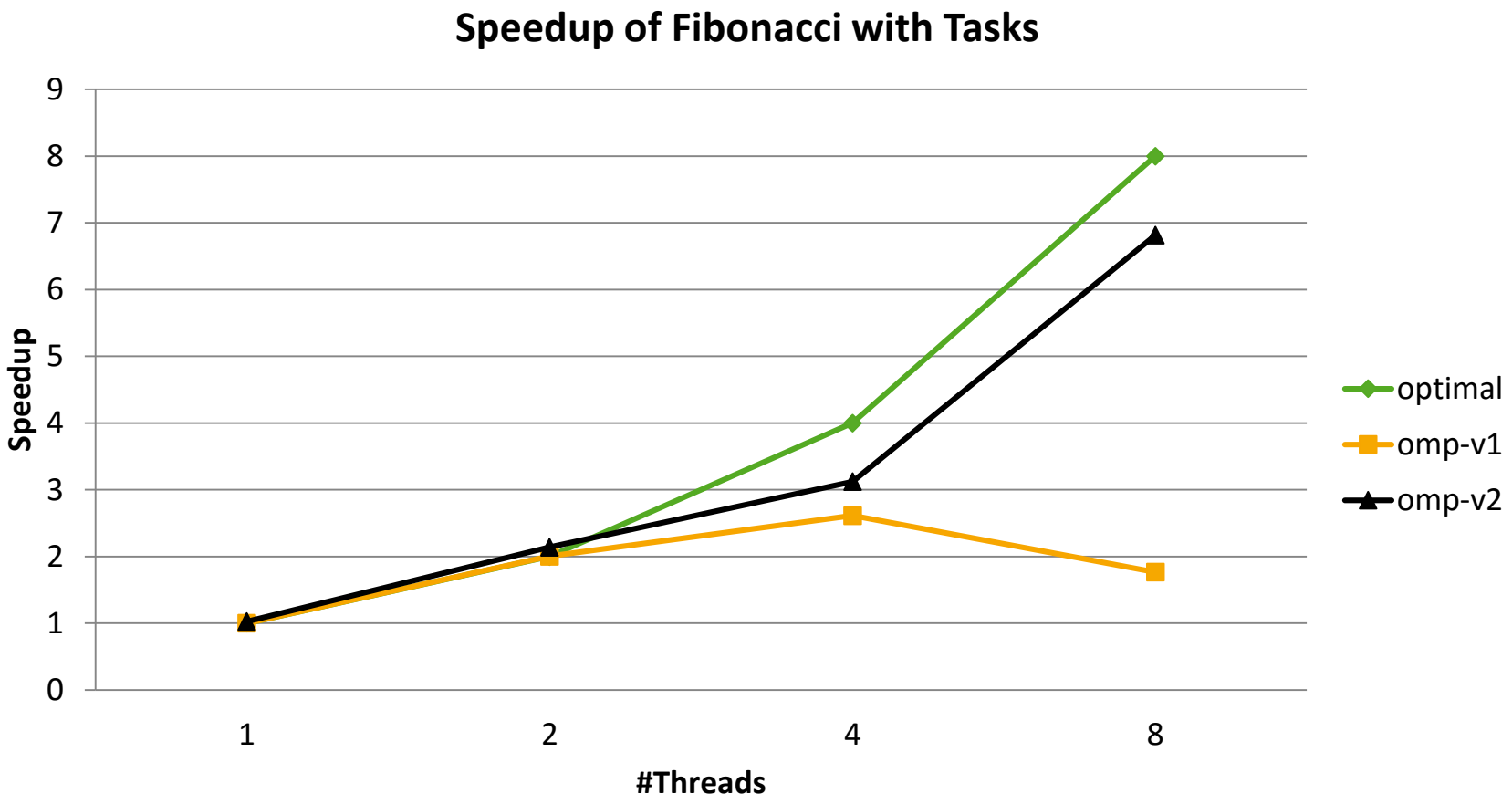
- **Improvement: Don‘t create yet another task once a certain (small enough) `n` is reached**

```
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)

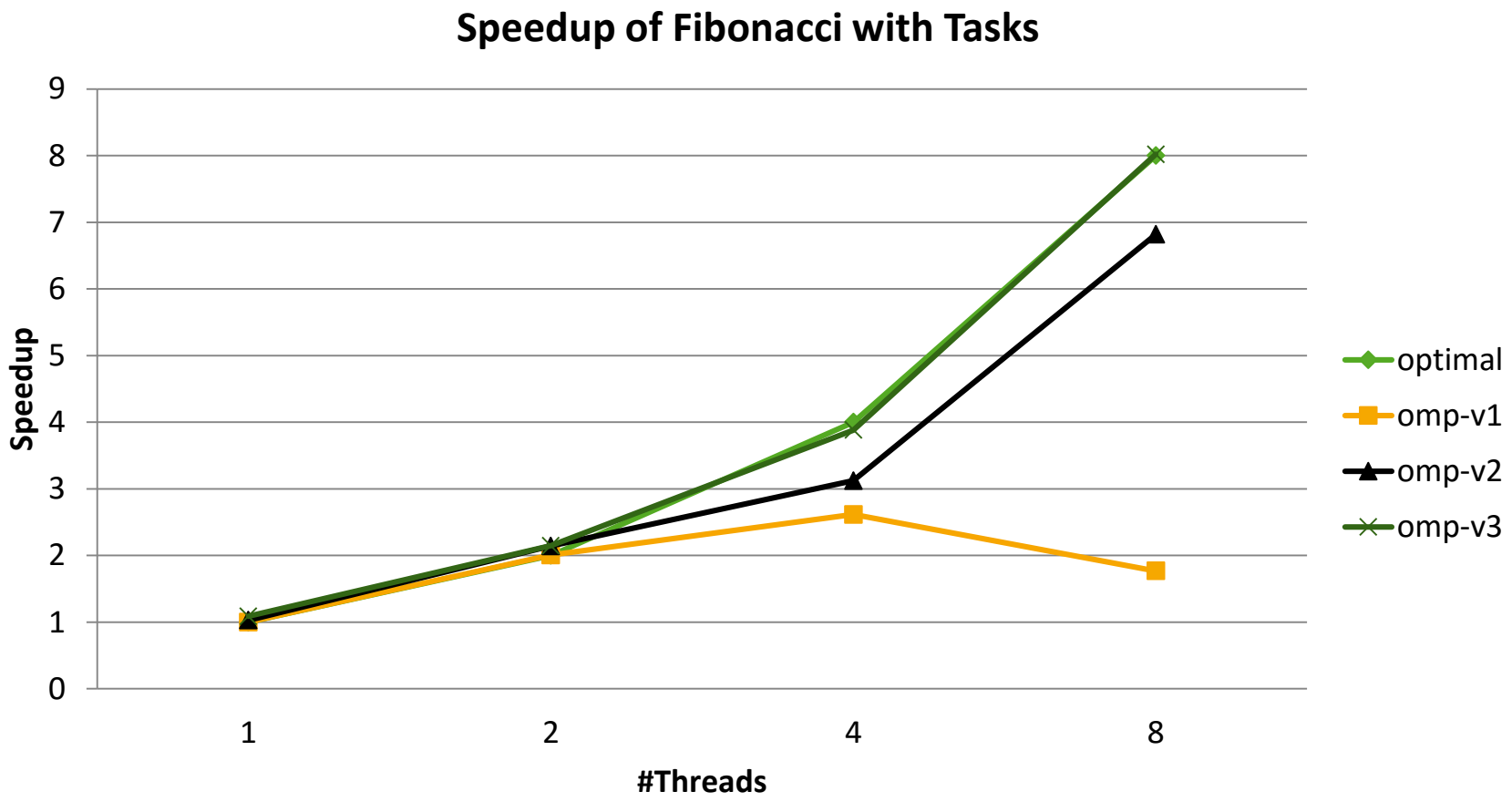
- **Improvement: Skip the OpenMP overhead once a certain `n` is reached (no issue w/ production compilers)**

```
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
```

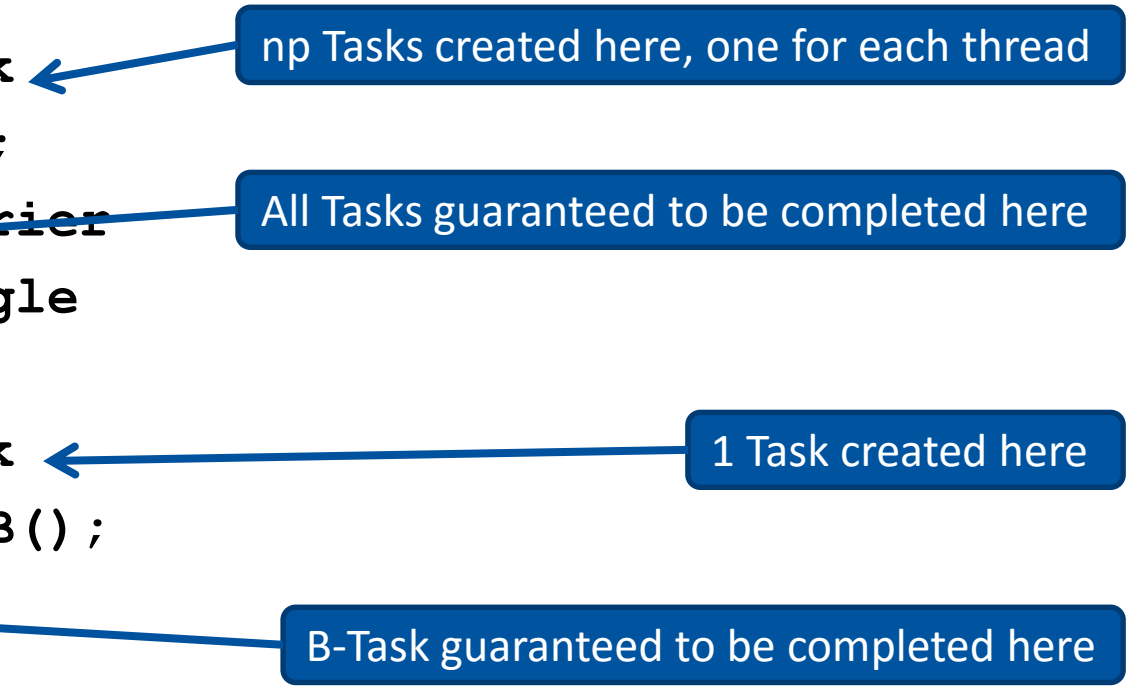
- **Task barrier: `taskwait`**
  - → Encountering Task suspends until child tasks are complete
    - → Only direct childs, not descendants!

```
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();
   }
}
```



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)

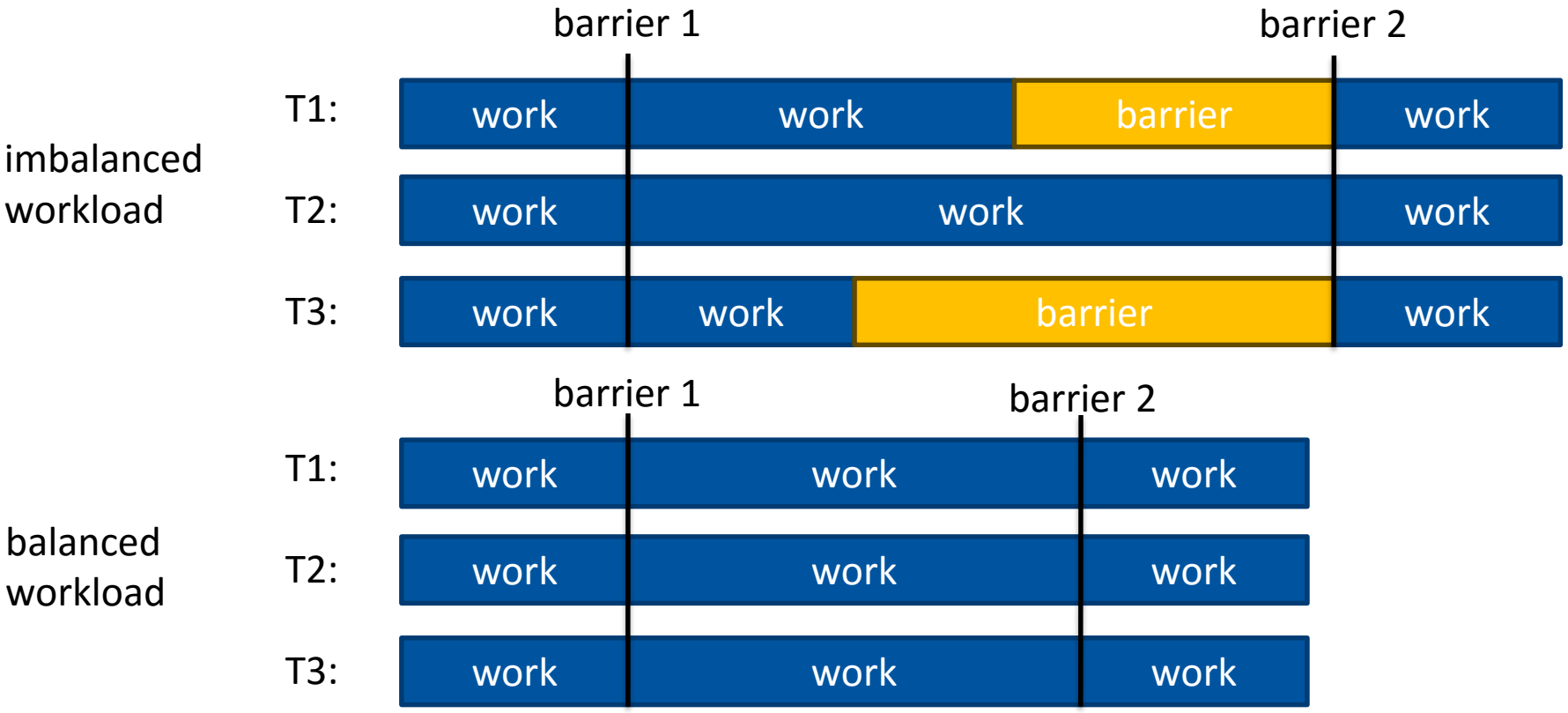
- **Define interaction with system environment:**
  - → Env. Var. OMP_MAX_NESTED_LEVEL + API functions
    - →Controls the maximum number of active parallel regions
  - → Env. Var. OMP_THREAD_LIMIT + API functions
    - →Controls the maximum number of OpenMP threads
  - → Env. Var. OMP_STACKSIZE
    - →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

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → Basic concept
   - → Scoping
   - → Synchronization
   - → Correctness Checking Tools
   - → Runtime Library
   - → Tasking
   - → **Load Balancing**
   - → False Sharing
   - → NUMA Architectures
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Load imbalance

- **Load imbalance occurs in a parallel program**
  - → when multiple threads synchronize at global synchronization points
  - → and these threads need a different amount of time to finish the calculation.



Influencing the For Loop Scheduling

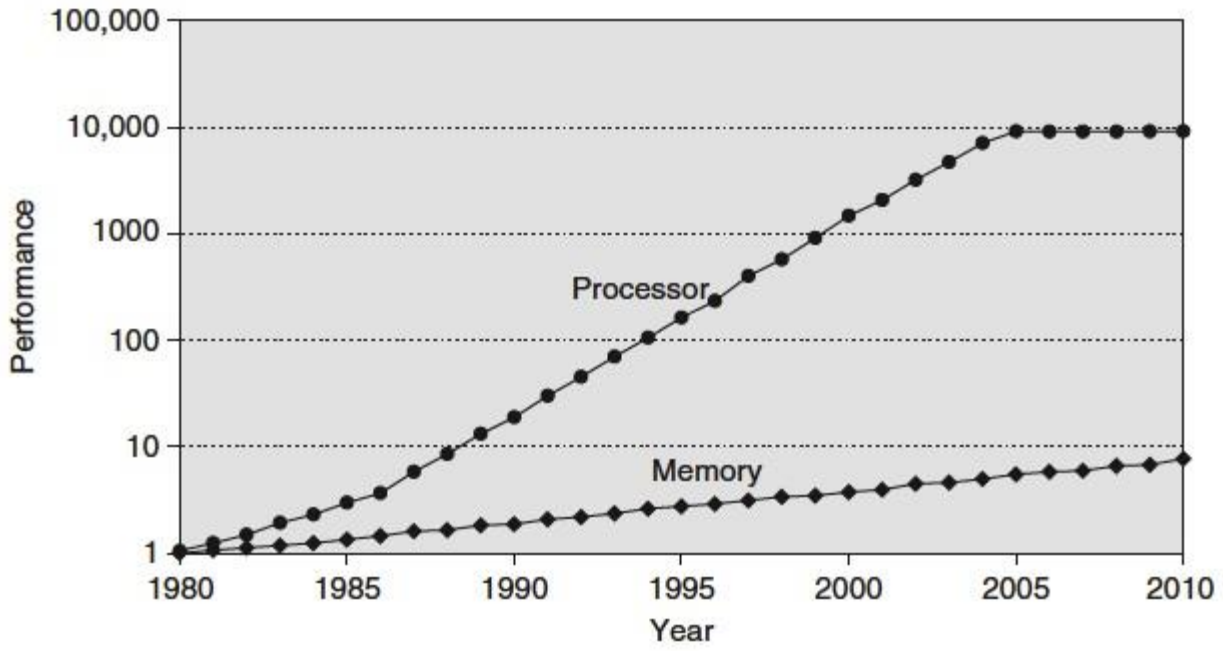
- ***for*-construct: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the *schedule* clause:**
  - → `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not 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)`.**

Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → Basic concept
   - → Scoping
   - → Synchronization
   - → Correctness Checking Tools
   - → Runtime Library
   - → Tasking
   - → Load Balancing
   - → **False Sharing**
   - → NUMA Architectures
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Memory Bottleneck

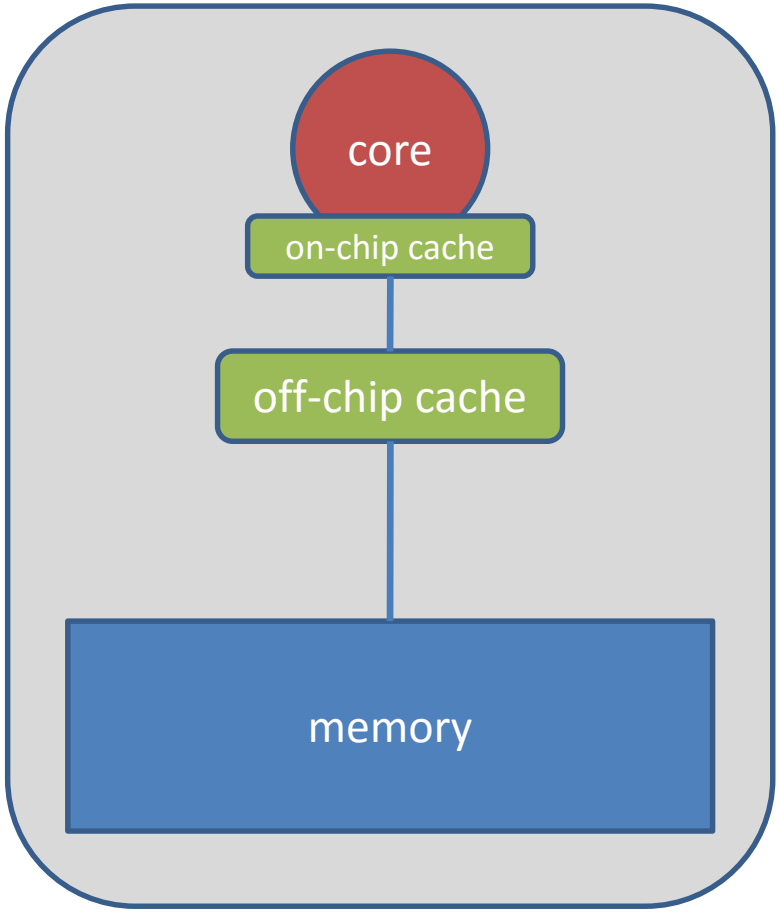
- **There is a growing gap between core and memory performance:**
  - → memory, since 1980: 1.07x per year improvement in latency
  - → single core: since 1980: 1.25x per year until 1986, 1.52x p. y. until 2000, 1.20x per year until 2005, then no change on a *per-core* basis



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

Caches

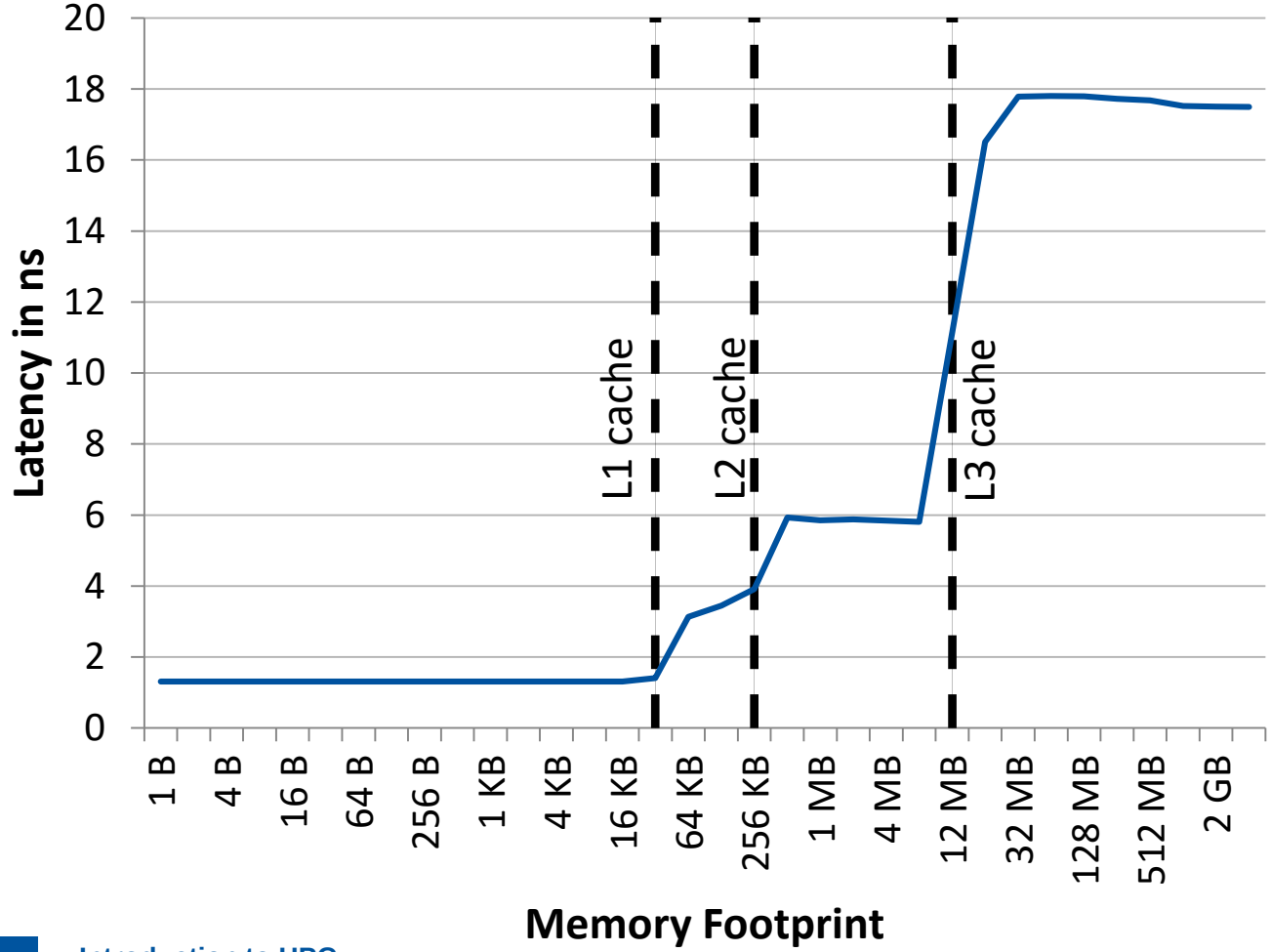
- **CPU is fast**
  - → Order of 3.0 GHz
- **Caches:**
  - → Fast, but expensive
  - → Thus small, order of MB
- **Memory is slow**
  - → Order of 0.3 GHz
  - → Large, order of GB



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

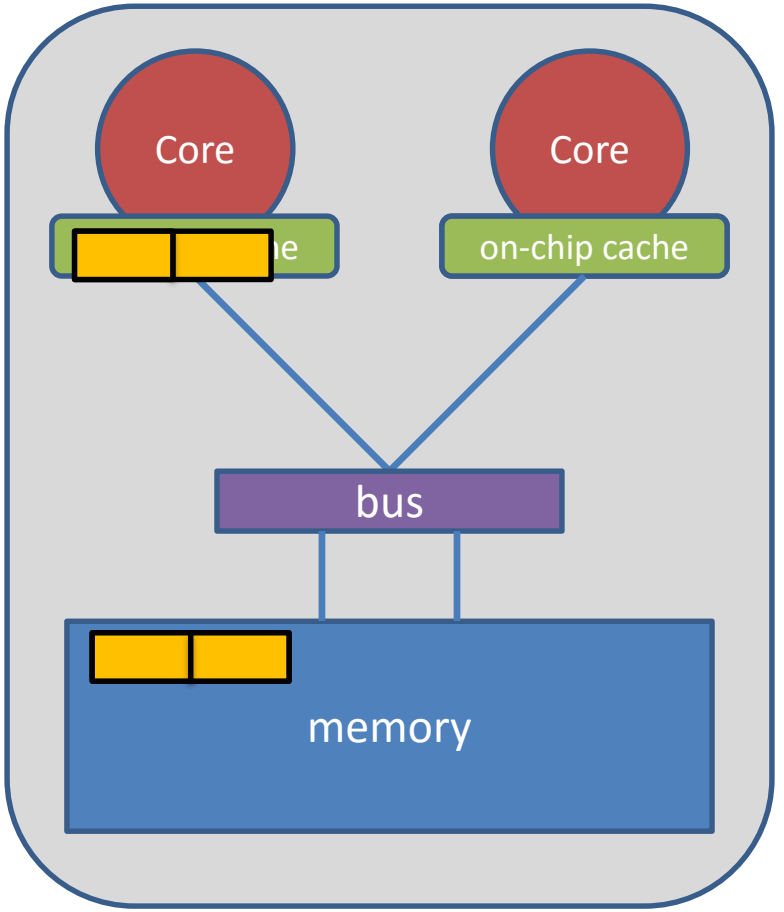
Visualization of the Memory Hierarchy

- **Latency on the Intel Westmere-EP 3.06 GHz processor**



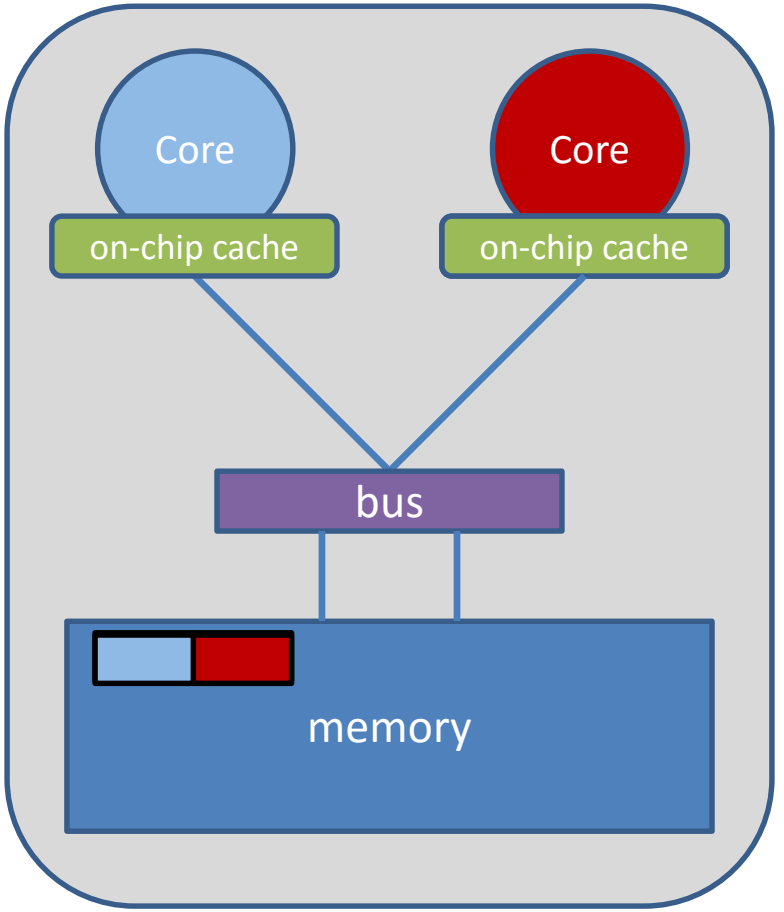
Data in Caches

- **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**



Summing up vector elements again
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
```

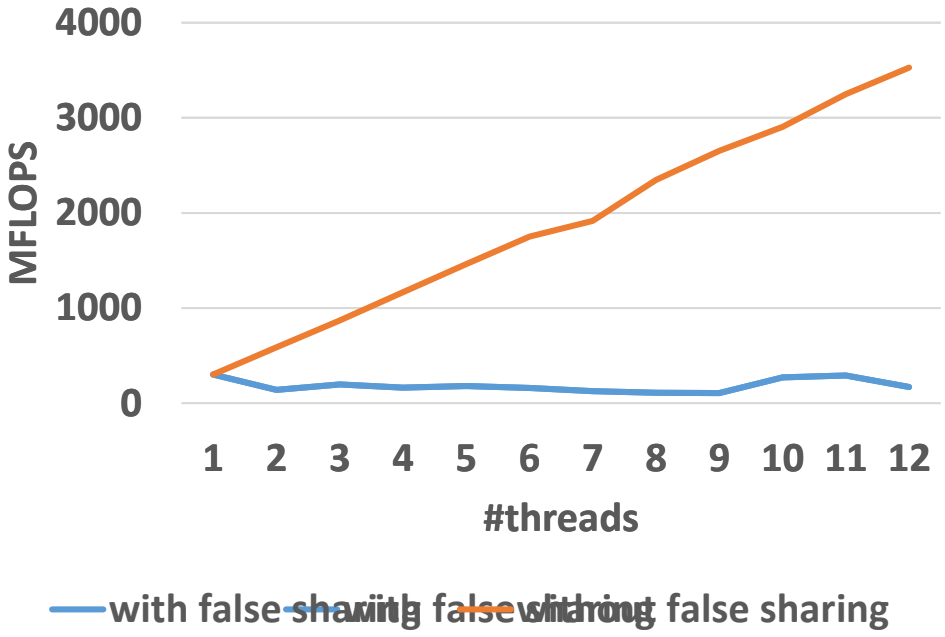
```
do i = 75, 99
   s = s + a(i)
end do
```

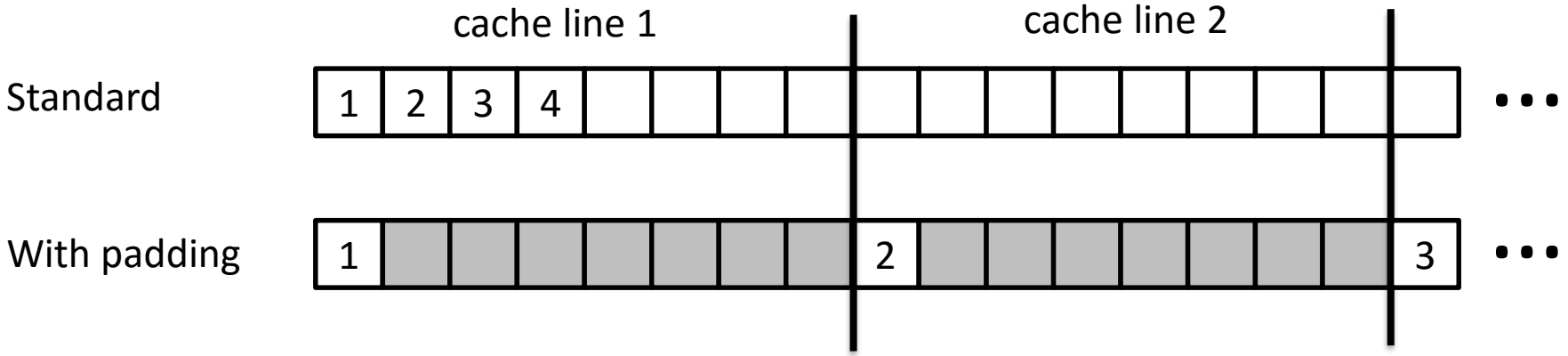
False Sharing

```
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];
}
```

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





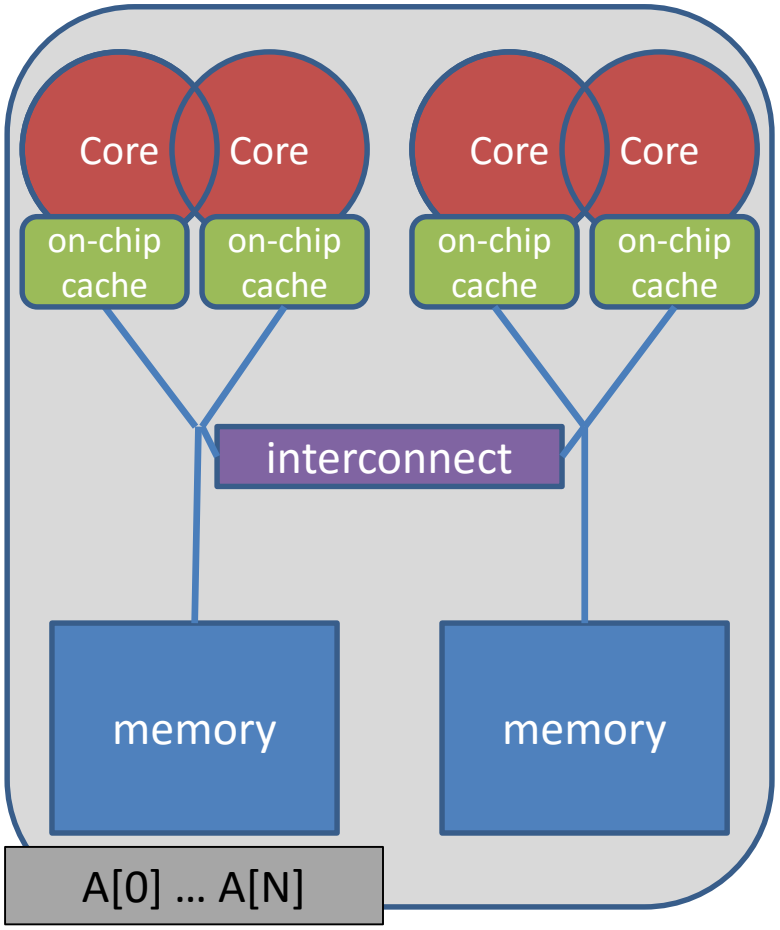
Outline

1. Why supercomputers?
2. Modern processors
3. Basic optimization techniques for serial code
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. **Shared-memory programming with OpenMP**
   - → Basic concept
   - → Scoping
   - → Synchronization
   - → Correctness Checking Tools
   - → Runtime Library
   - → Tasking
   - → Load Balancing
   - → False Sharing
   - → **NUMA Architectures**
8. Distributed-memory programming with MPI
9. Hybrid programming (MPI + OpenMP)
10. Parallel algorithms
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

How To Distribute The Data ?

```
double* A;
A = (double*)
    malloc(N * sizeof(double));
```

```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



About Data Distribution

- **Important aspect on cc-NUMA systems**
  - → If not optimal, longer memory access times and hotspots
- **OpenMP does not provide support for cc-NUMA**
- **Placement comes from the Operating System**
  - → This is therefore Operating System dependent

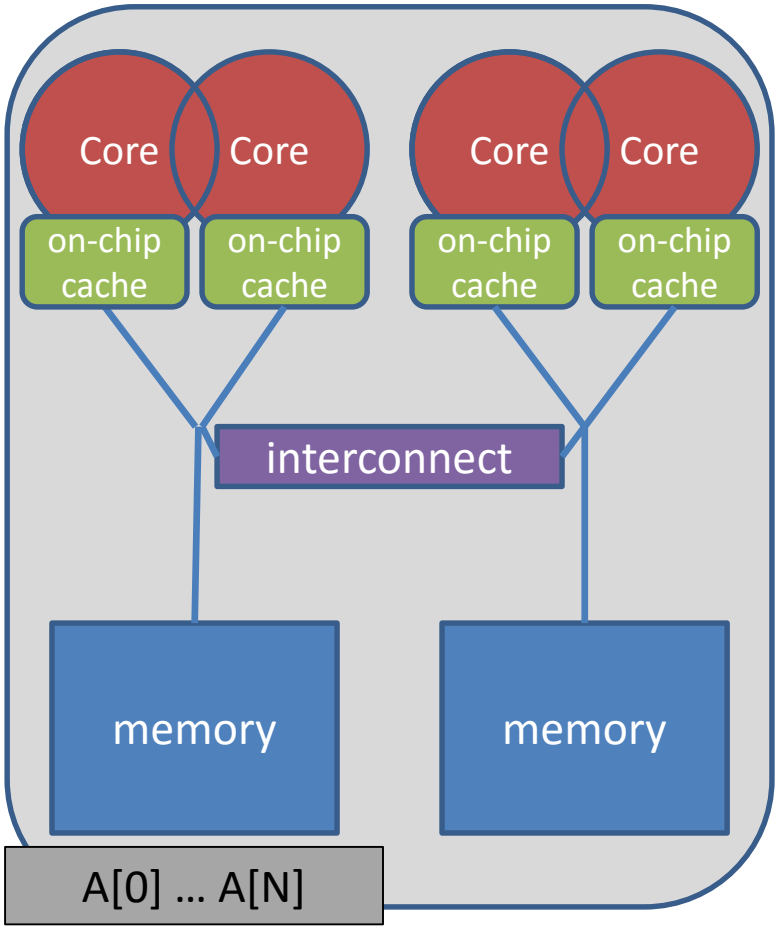
- **Windows, Linux and Solaris all use the “First Touch” placement policy by default**
  - → May be possible to override default (check the docs)

Non-Uniform Memory Arch.

- **Serial code: all array elements are allocated in the memory of the NUMA node containing the core executing this thread**

```
double* A;
A = (double*)
    malloc(N * sizeof(double));
```

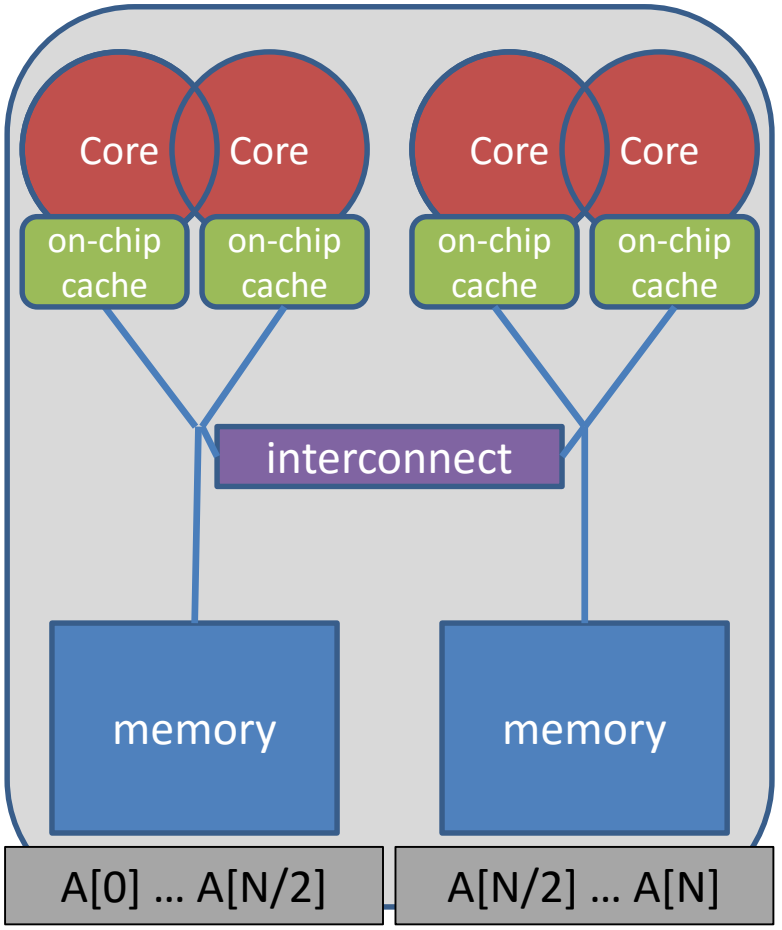
```
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



Non-Uniform Memory Arch.

- First Touch w/ parallel code: all array elements are allocated in the memory of the NUMA node containing the core executing the thread initializing the respective partition

```
double* A;
A = (double*)
    malloc(N * sizeof(double));

omp_set_num_threads(2);

#pragma omp parallel for
for (int i = 0; i < N; i++) {
   A[i] = 0.0;
}
```



Get Info on the System Topology

- **Before you design a strategy for thread binding, you should have a basic understanding of the system topology. Please use one of the following options on a target machine:**
  - → Intel MPI's `cpuinfo` tool
    - → `module switch openmpi intelmpi`
    - → `cpuinfo`
    - → Delivers information about the number of sockets (= packages) and the mapping of processor ids used by the operating system to cpu cores.
  - → hwlocs' `hwloc-ls` tool
    - → `hwloc-ls`
    - → Displays a graphical representation of the system topology, separated into NUMA nodes, along with the mapping of processor ids used by the operating system to cpu cores and additional info on caches.

Decide for Binding Strategy

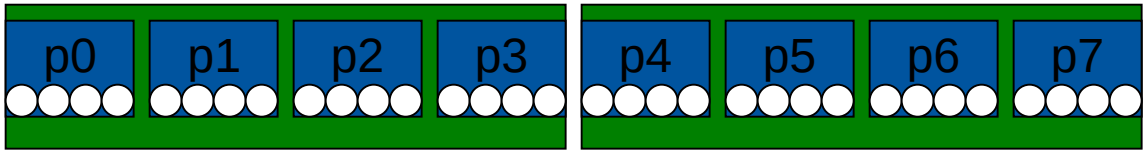
- **Selecting the „right“ binding strategy depends not only on the topology, but also on the characteristics of your application.**
  - → Putting threads far apart, i.e. on different sockets
    - →May improve the aggregated memory bandwidth available to your application
    - →May improve the combined cache size available to your application
    - →May decrease performance of synchronization constructs
  - → Putting threads close together, i.e. on two adjacent cores which possibly shared some caches
    - →May improve performance of synchronization constructs
    - →May decrease the available memory bandwidth and cache size
- **If you are unsure, just try a few options and then select the best one.**

OpenMP 4.0: Places + Binding Policies (1/2)

- **Define OpenMP Places**
  - → set of OpenMP threads running on one or more processors
  - → can be defined by the user, i.e. `OMP_PLACES=cores`
- **Define a set of OpenMP Thread Affinity Policies**
  - → SPREAD: spread OpenMP threads evenly among the places
  - → CLOSE: pack OpenMP threads near master thread
  - → MASTER: collocate OpenMP thread with master thread
- **Goals**
  - → user has a way to specify where to execute OpenMP threads for
  - → locality between OpenMP threads / less false sharing / memory bandwidth

Places

- **Assume the following machine:**



→ 2 sockets, 4 cores per socket, 4 hyper-threads per core

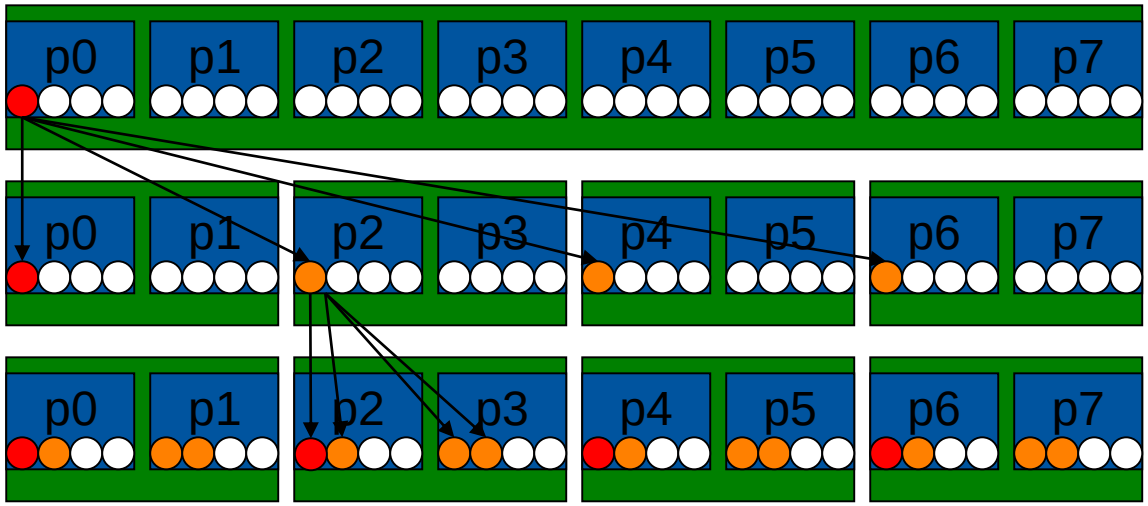
- **Abstract names for OMP_PLACES:**
  - → threads: Each place corresponds to a single hardware thread on the target machine.
  - → cores: Each place corresponds to a single core (having one or more hardware threads) on the target machine.
  - → sockets: Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

OpenMP 4.0: Places + Binding Policies (2/2)

- **Example's Objective:**
  - → separate cores for outer loop and near cores for inner loop
- **Outer Parallel Region: proc_bind(spread), Inner: proc_bind(close)**
  - → spread creates partition, compact binds threads within respective partition

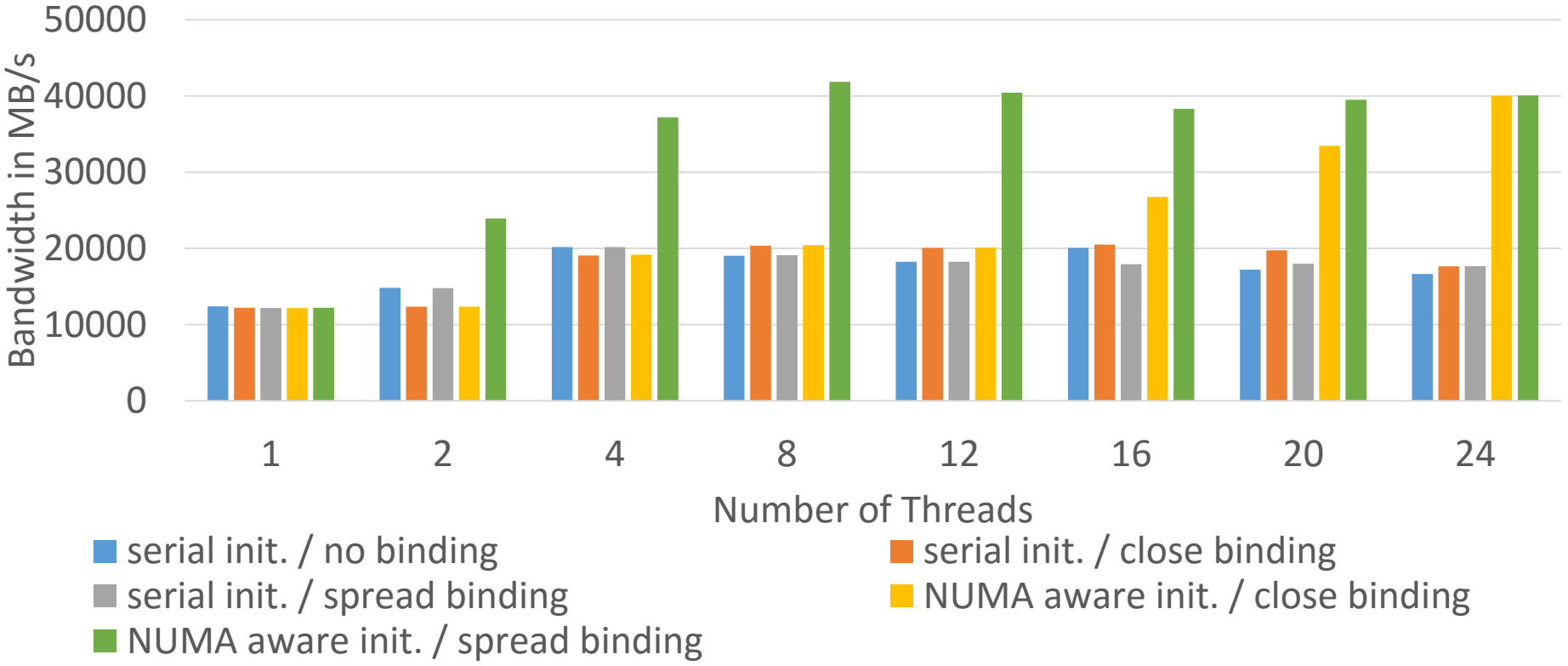
```
OMP_PLACES=(0,1,2,3), (4,5,6,7), ... = (0-3):8:4    = cores
#pragma omp parallel proc_bind(spread)
#pragma omp parallel proc_bind(close)
```

- **Example**
  - → initial
  - → spread 4
  - → close 4



Serial vs. Parallel Initialization

- **Performance of OpenMP-parallel STREAM vector assignment measured on 2-socket Intel® Xeon® X5675 („Westmere“) using Intel® Composer XE 2013 compiler with different thread binding options:**

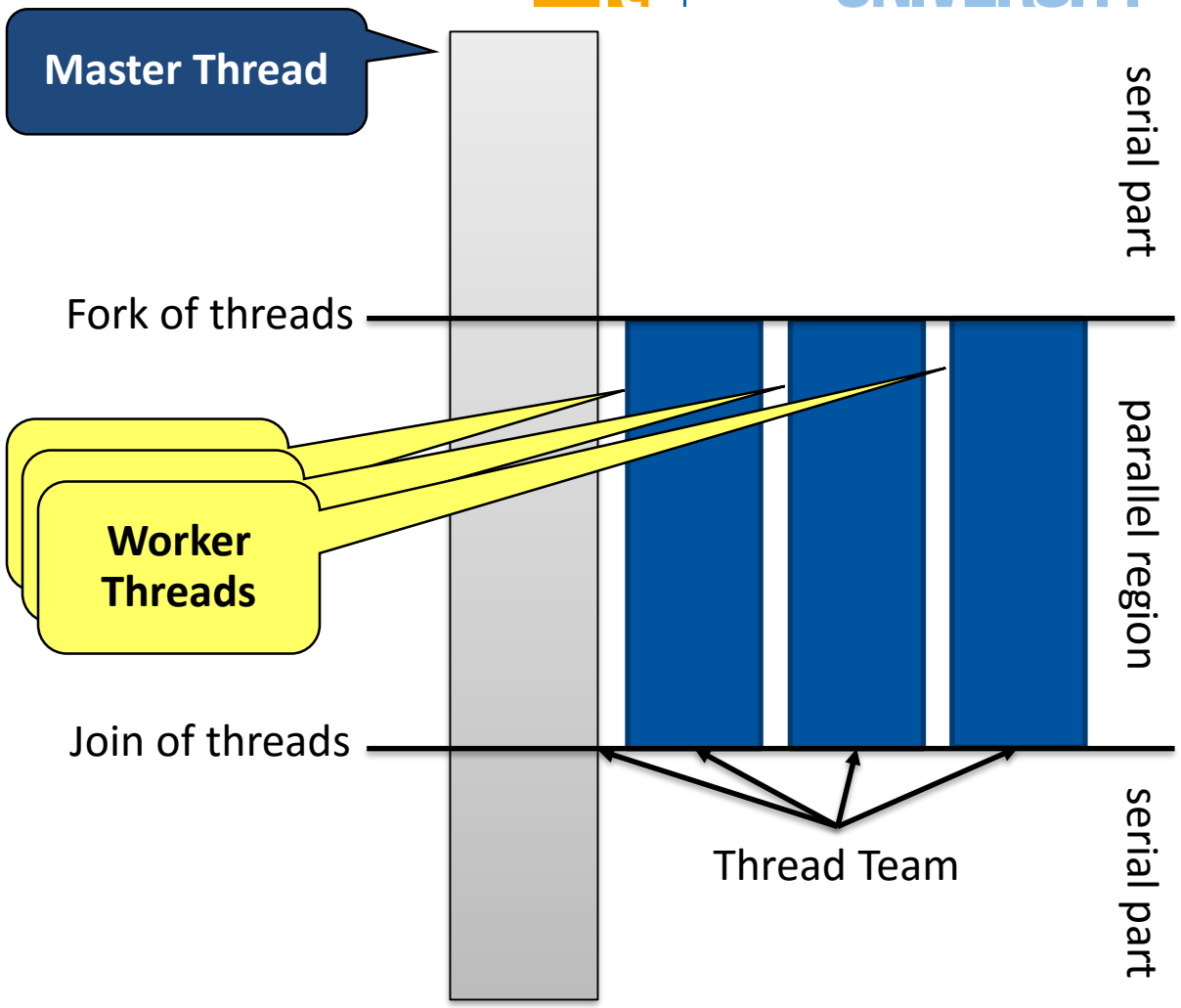


What you have learnt

- **Basic principle of OpenMP**
  - → Execution model
  - → Parallel region + worksharing constructs
- **Scoping**
  - → Data sharing clauses
- **Synchronization**
  - → Critical section
  - → Reduction clause
  - → Team and Task-Barriers
- **Runtime library**
  - → Important functions

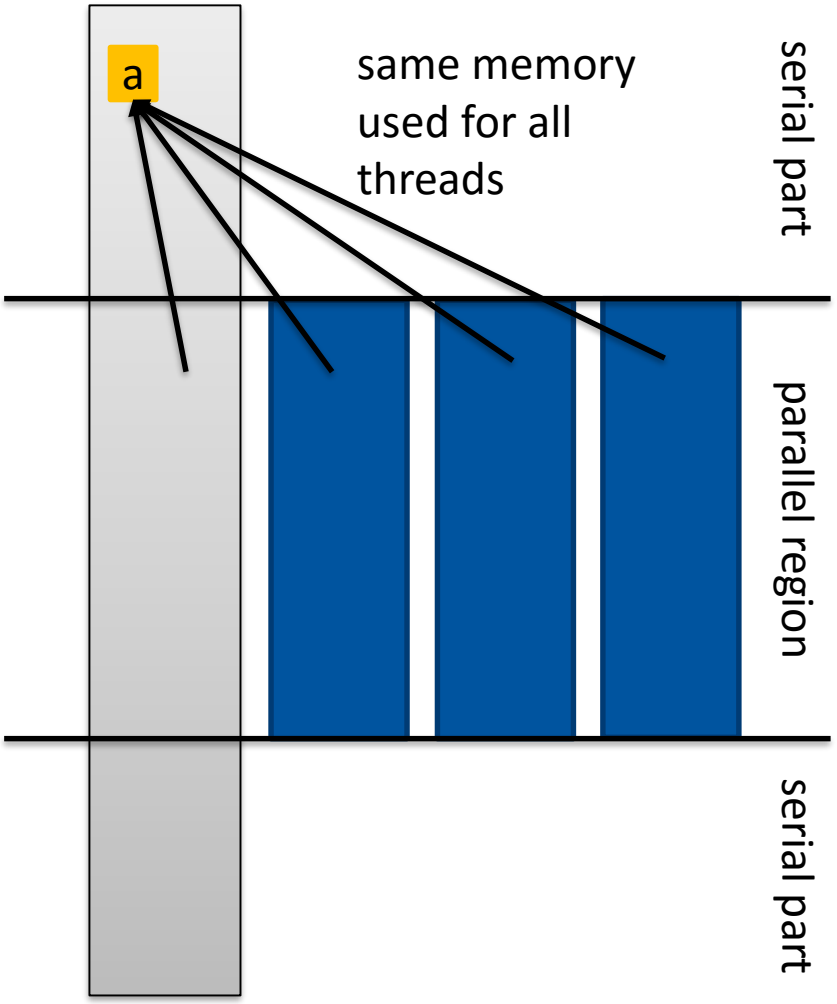
What you have learnt
Fork-Join Execution Model

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



What you have learnt
Data Sharing Attributes (1/3)

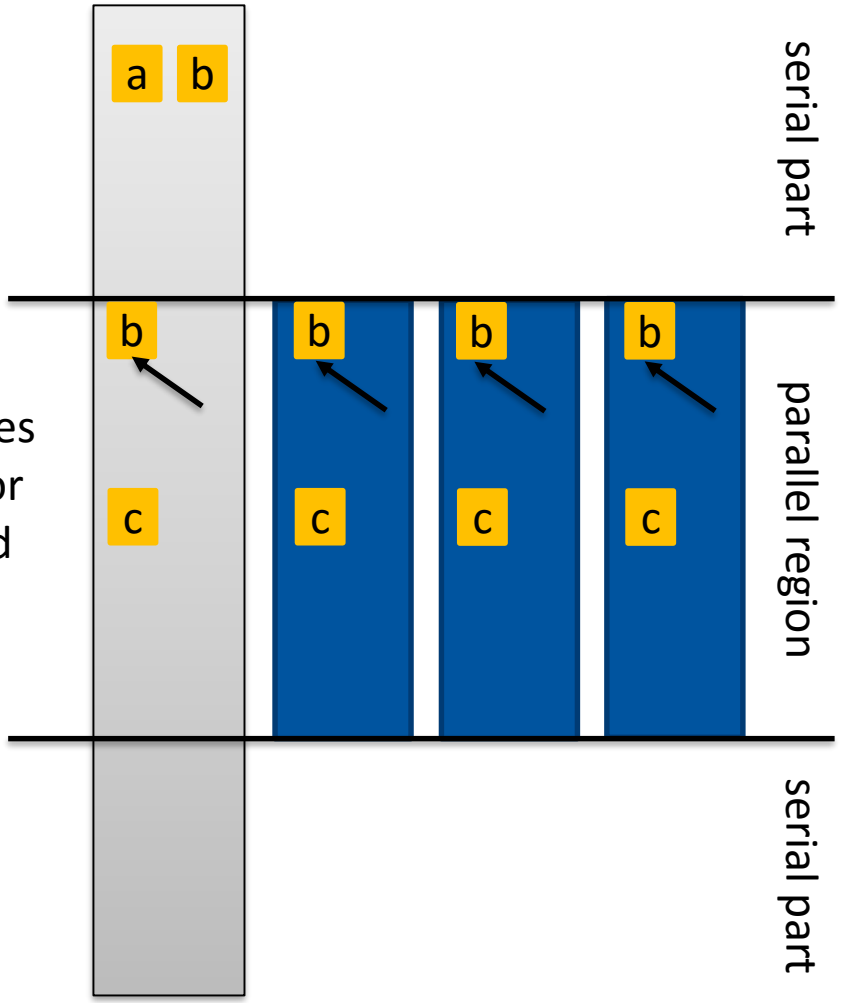
```
int a;
.
.
#pragma omp parallel shared(a)
{
.
.
.
.
}
.
.
.
```



What you have learnt
Data Sharing Attributes (2/3)

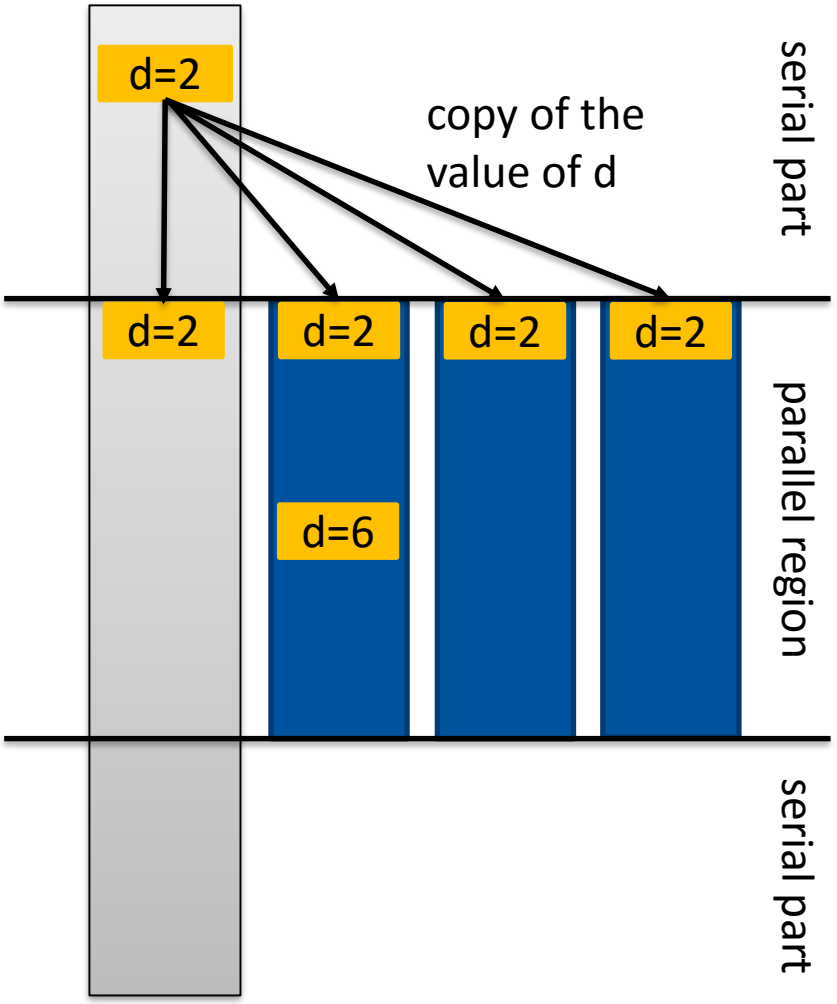
```
int a,b;
.
#pragma omp parallel shared(a) //
  private(b)
{
.
int c;
.
}
.
.
.
```

uninitialized
private copies
of b and c for
every thread



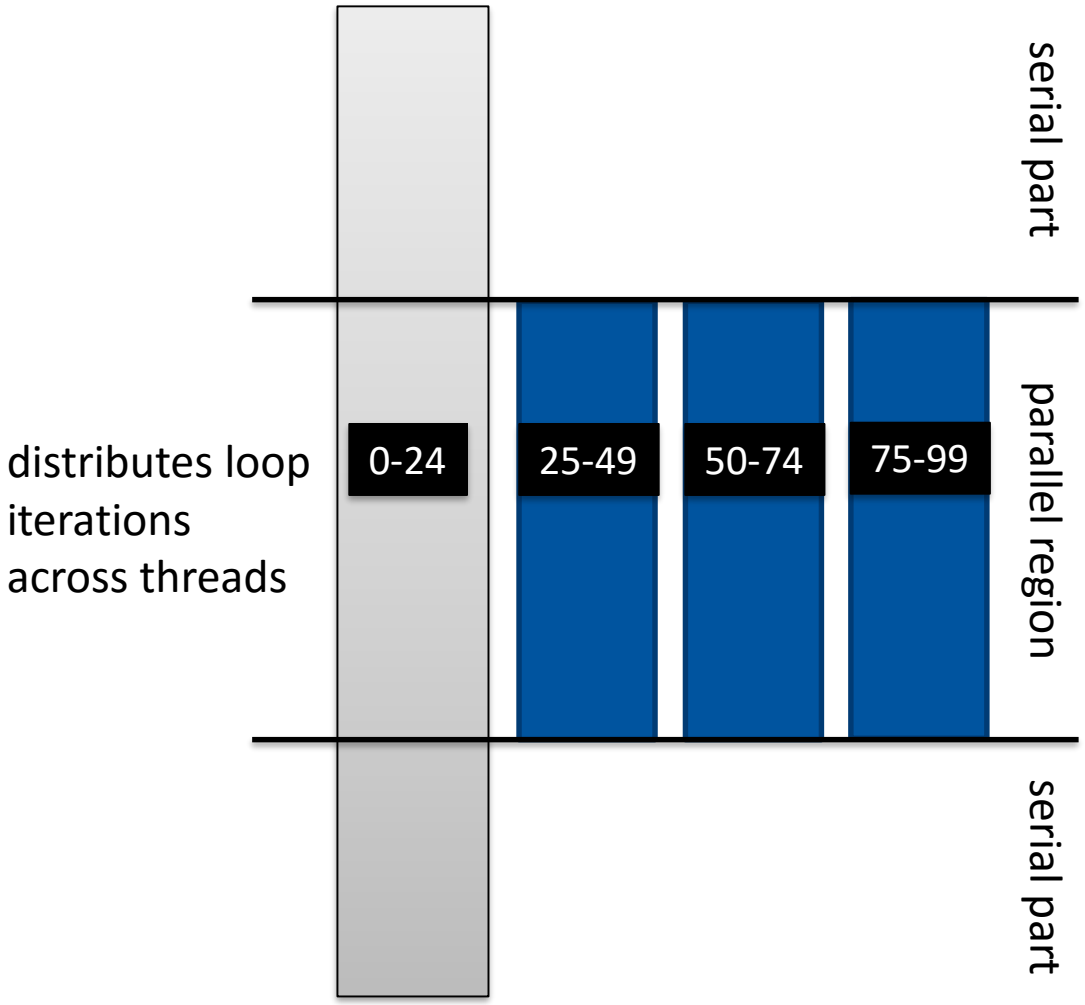
What you have learnt
Data Sharing Attributes (3/3)

```
int d=2;
.
.
#pragma omp parallel firstprivate(d)
{
#pragma omp single
{d=6;}
.
}
.
.
.
```



What you have learnt For Worksharing

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



What you have learnt
Parallelizable Loops

- **Loop iterations must be independent to parallelize a loop!**

No loop dependencies => parallelizable

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

Loop dependencies => **not** parallelizable

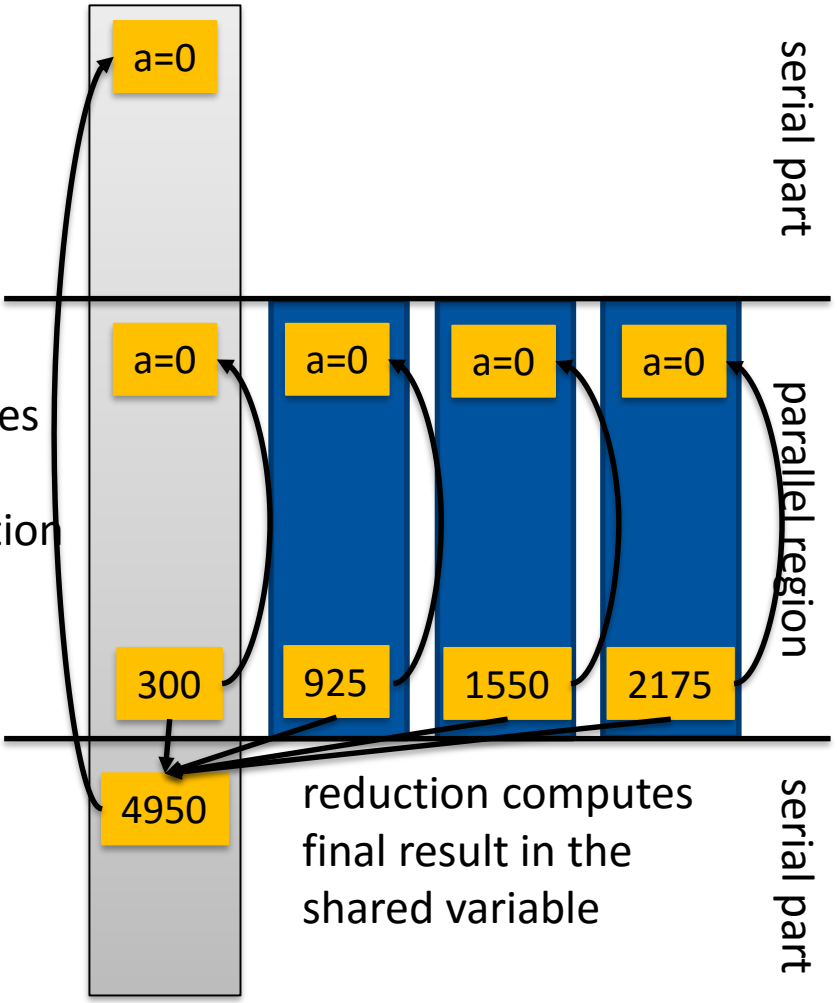
```
#pragma omp parallel for
for ( i=1 ; i<100 ; i++ ){
        a[i] = a[i] + a[i-1];
}
```

- **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

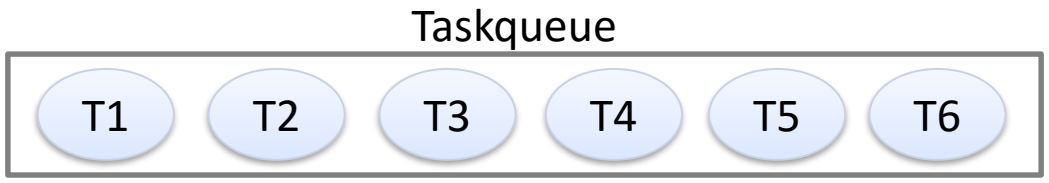
What you have learnt
Reduction Operations

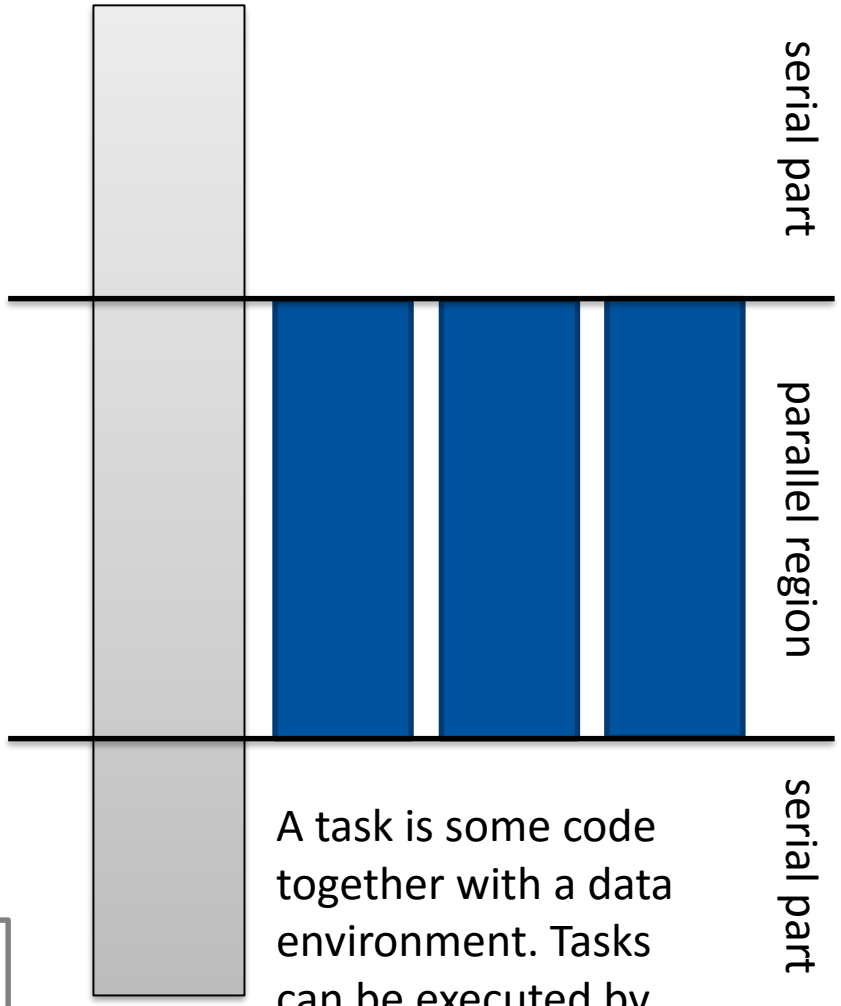
```
int a=0;
.
.
#pragma omp parallel
#pragma omp for reduction(+:a)
for (int i=0; i<100; i++)
{
        a+=i;
}
.
.
.
```



What you have learnt
Tasks

```
#pragma omp parallel
#pragma omp single
while (work()){
        #pragma omp task
        {
        …
        }
}// implicit barrier here
.
.
```





A task is some code together with a data environment. Tasks can be executed by any thread in any order.