TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

Parallel Programming - OpenMP -

Prof. Dr. Felix Wolf

Technical University of Darmstadt

Outline

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

- Introduction
- Loop-level parallelism
- SPMD-style parallelism
- Synchronization
- Tasking

Introduction

Literature

- This course is based on
  - Parallel Programming in OpenMP by R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Meno, Morgan Kaufmann, 2000.
  - Open OpenMP Application Programming Interface Specification + Examples http://www.openmp.org/specifications/

- Also useful
  - The OpenMP Common Core: Making OpenMP Simple Again by Alice E. Koniges, Timothy G. Mattson, and Yun (Helen) He, MIT Press, 2019

What is OpenMP ?

Open specifications for Multi Processing

- Community standard of a shared-memory programming interface
  - Compiler directives to describe parallelism in the source code
  - Library functions
  - Environment variables
- Supports the creation of portable parallel programs
- Works with C/C++ and Fortran
- Current specification 6.0 (Nov 2024)
- http://www.openmp.org

Here, we cover only C/C++

A simple example


```
#include <omp.h>
#include <stdio.h>

int main(int argc, char* argv[])
{
  printf("Hello parallel world from thread:\n");

#pragma omp parallel
 {
   printf("%d\n", omp_get_thread_num());
 }
  printf("Back to the sequential world\n");
}
```

Output

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

```
> gcc -fopenmp main.c -o a.out
> export OMP_NUM_THREADS=4
> ./a.out
Hello parallel world from thread:
1
3
0
2
Back to sequential world
>
```

Pros & cons

Advantages

- Incremental parallelization
- Small increase in code size
- Directives keep code readable
- Possible to write code that compiles and runs with a normal compiler on single CPU
- Single memory address space

Disadvantages

- Limited scalability
- Requires special compiler
- Implicit communication hard to understand. Sometimes unclear
  - When communication occurs
  - How costly it is
- Danger of incorrect synchronization

Origin of OpenMP

- OpenMP is a community standard
  - From concept to adoption from July 1996 to October 1997
- Predecessors
  - Proprietary designs by some vendors (e.g., SGI, CRAY, SUN, IBM) with different sets of directives, very similar in syntax and semantics
  - Each used a unique comment or pragma notation for "portability”
  - Different unsuccessful attempts to standardize interface (PCF, ANSI X3H5)
- OpenMP was motivated by developer community
- Increasing interest in a truly portable solution

Evolution of OpenMP

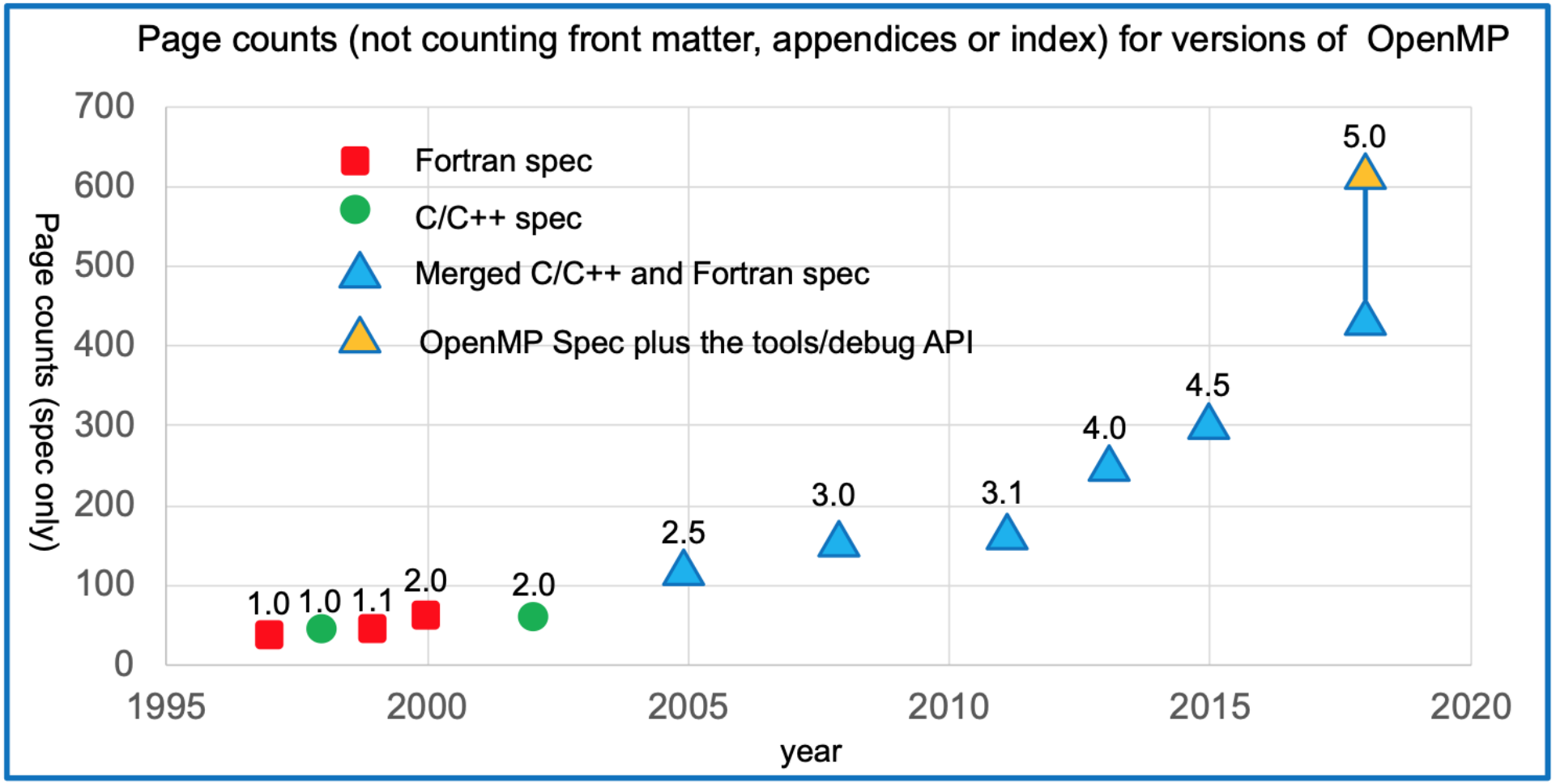
TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

Year	Version
1997	First API for Fortran V1.0
1998	First API for C/C++ V1.0
2000	Fortran V2.0
2002	C/C++ V2.0
2005	C/C++ and Fortran V2.5 ▪ Single standard for both languages ▪ Clarifications – in particular memory model
2008	V3.0 – Extensions including task parallelism

Year	Version
2013	V4.0 ▪ Extensions including SIMD parallelism and execution on accelerators ▪ Examples moved to a separate document
2015	V4.5
2018	V5.0 – Extended support for tasks and memory allocation
2020	V5.1
2021	V5.2
2024	V6.0 – Full support for C23, C++23, and Fortran 2023

Evolution of OpenMP



Source: Timothy G. Mattson, Yun He, Alice E. Koniges: The OpenMP Common Core, The MIT Press, 2019

Components of the API

- **Directives**
  - Instructional notes to any compiler supporting OpenMP
  - Pragmas in C/C++; source-code comments in Fortran
  - Express parallelism, including data environment and synchronization
- **Runtime library functions**
  - Examine and modify parallel execution parameters
  - Synchronization
- **Environment variables**
  - Preset parallel execution parameters

Directive syntax

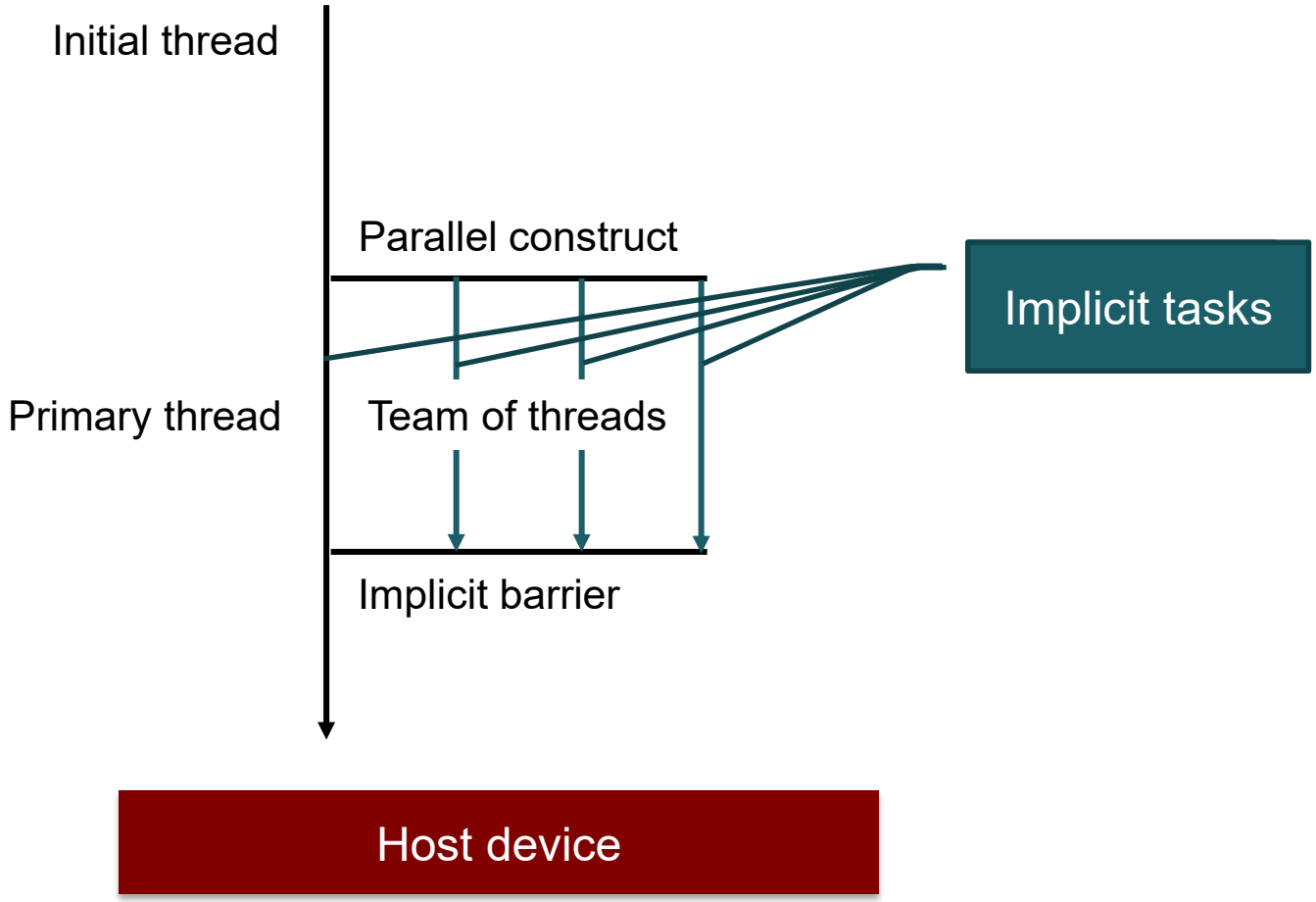
- OpenMP pragma (C/C++)

```
#pragma omp …
```

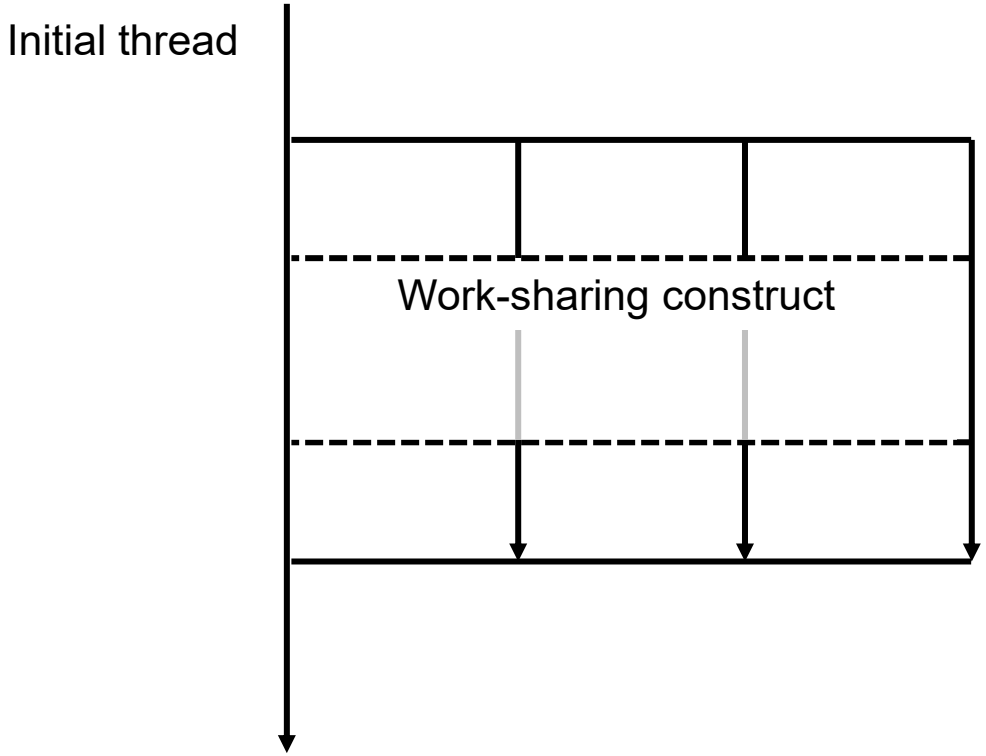
- Directives ignored by non-OpenMP compiler
- Conditional compilation with preprocessor macro name

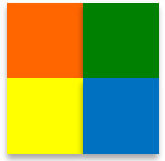
```
#ifdef _OPENMP
   iam = omp_get_thread_num();
#endif
```

Fork–join execution model



Work sharing

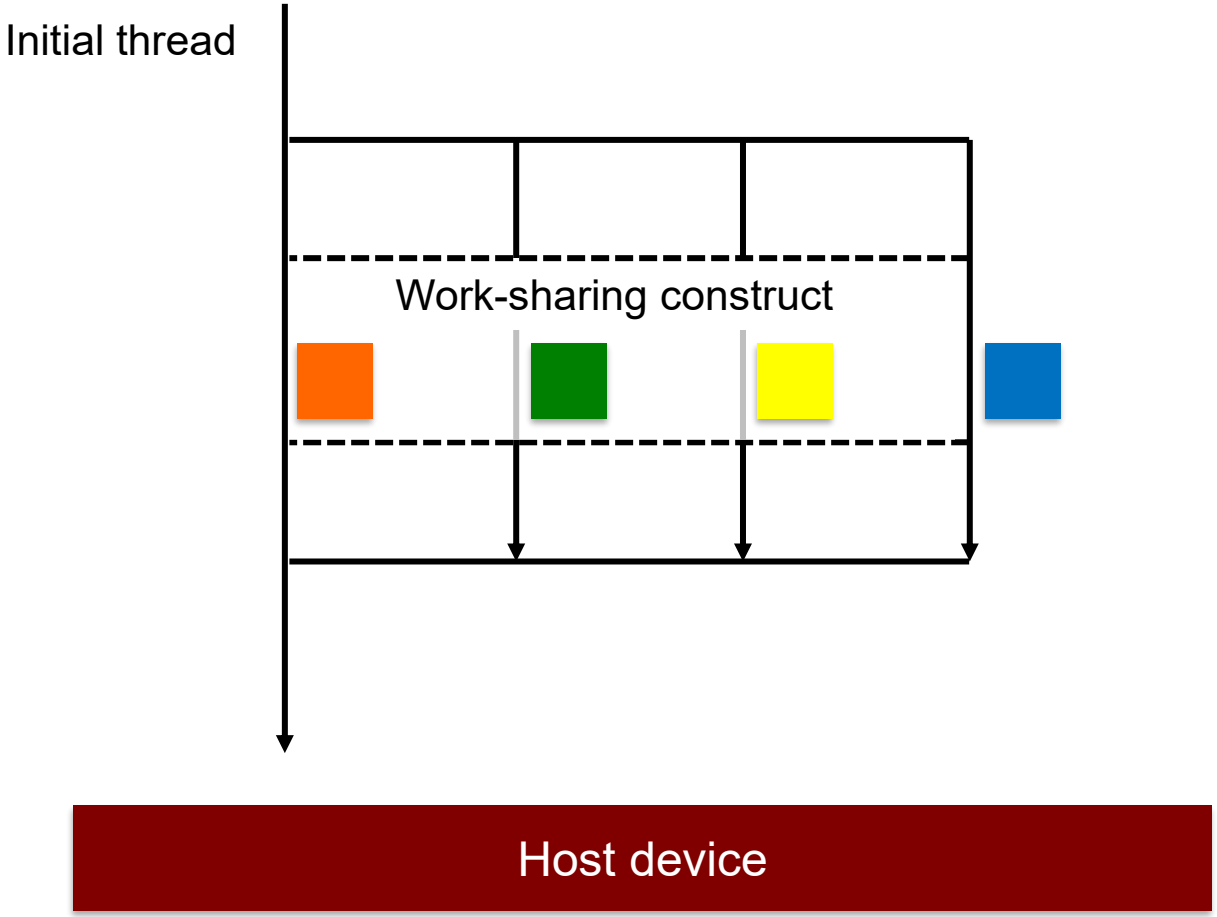




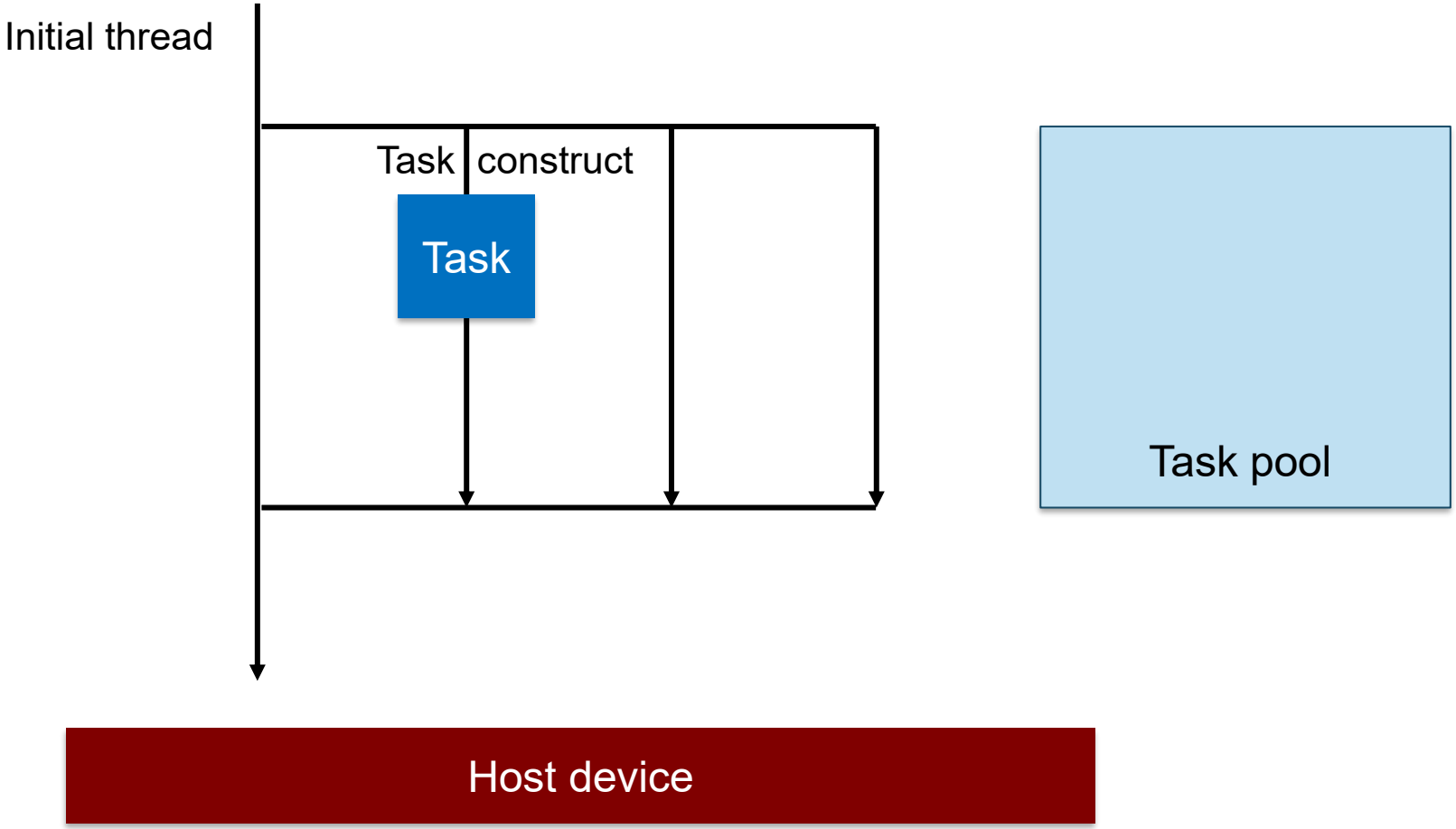
Work

Host device

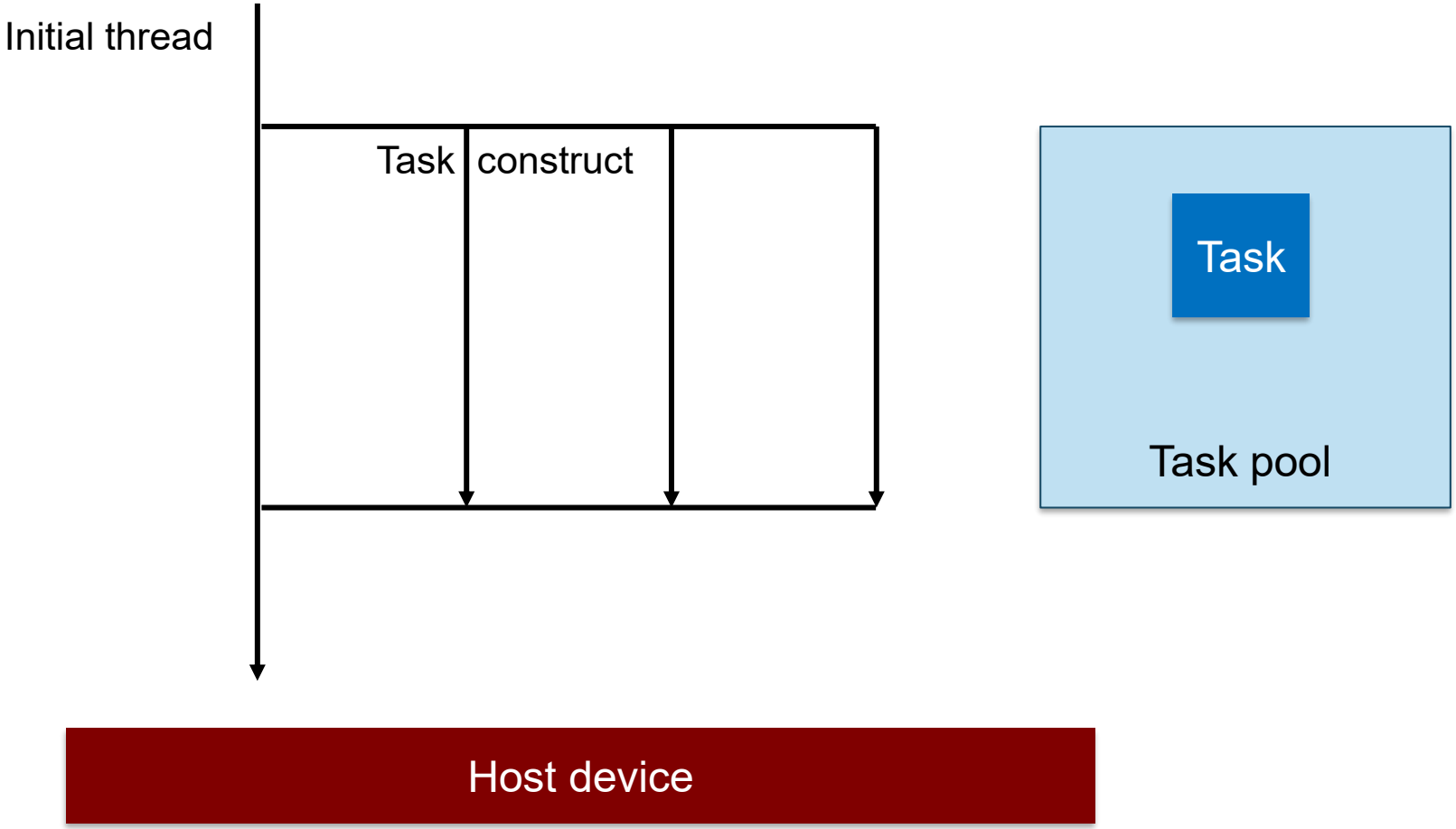
Work sharing



Tasking

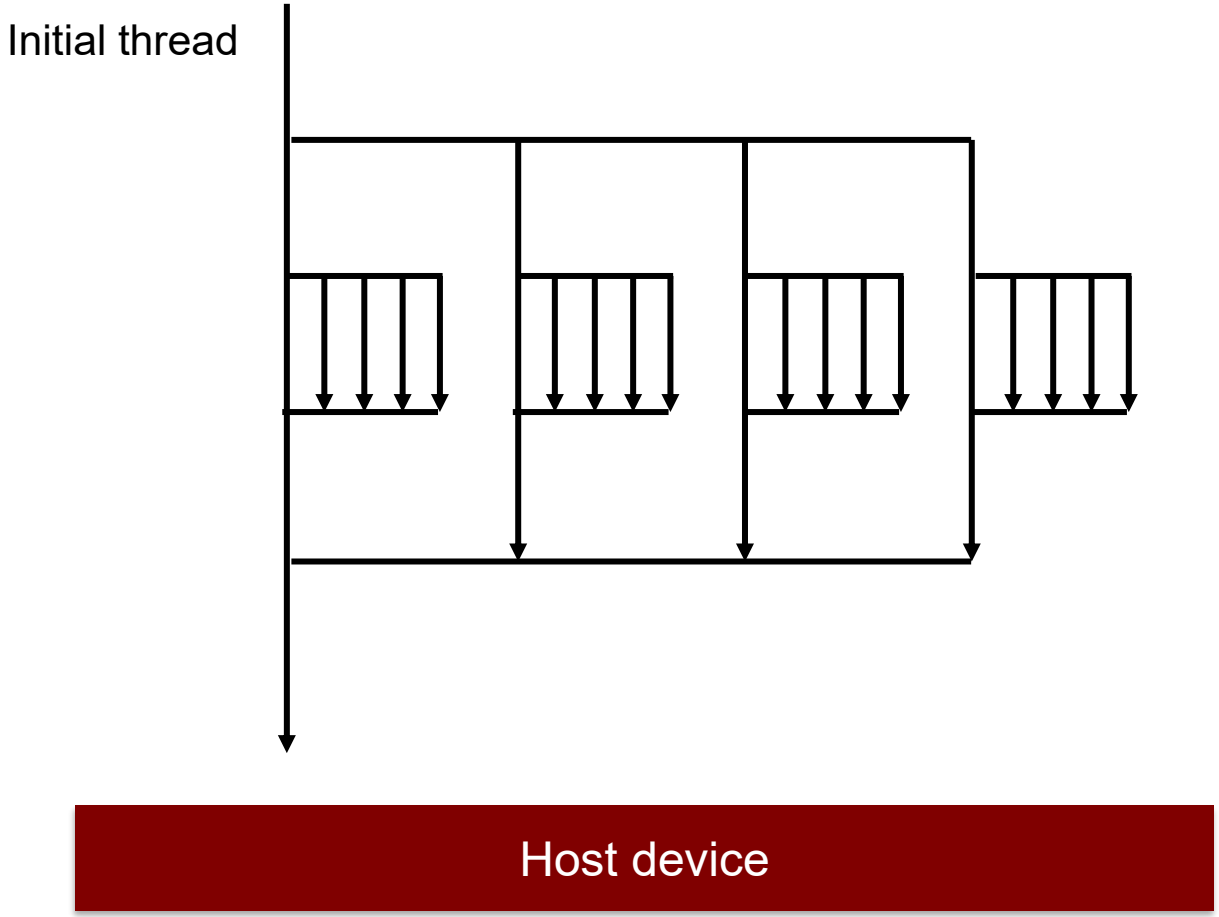


Tasking



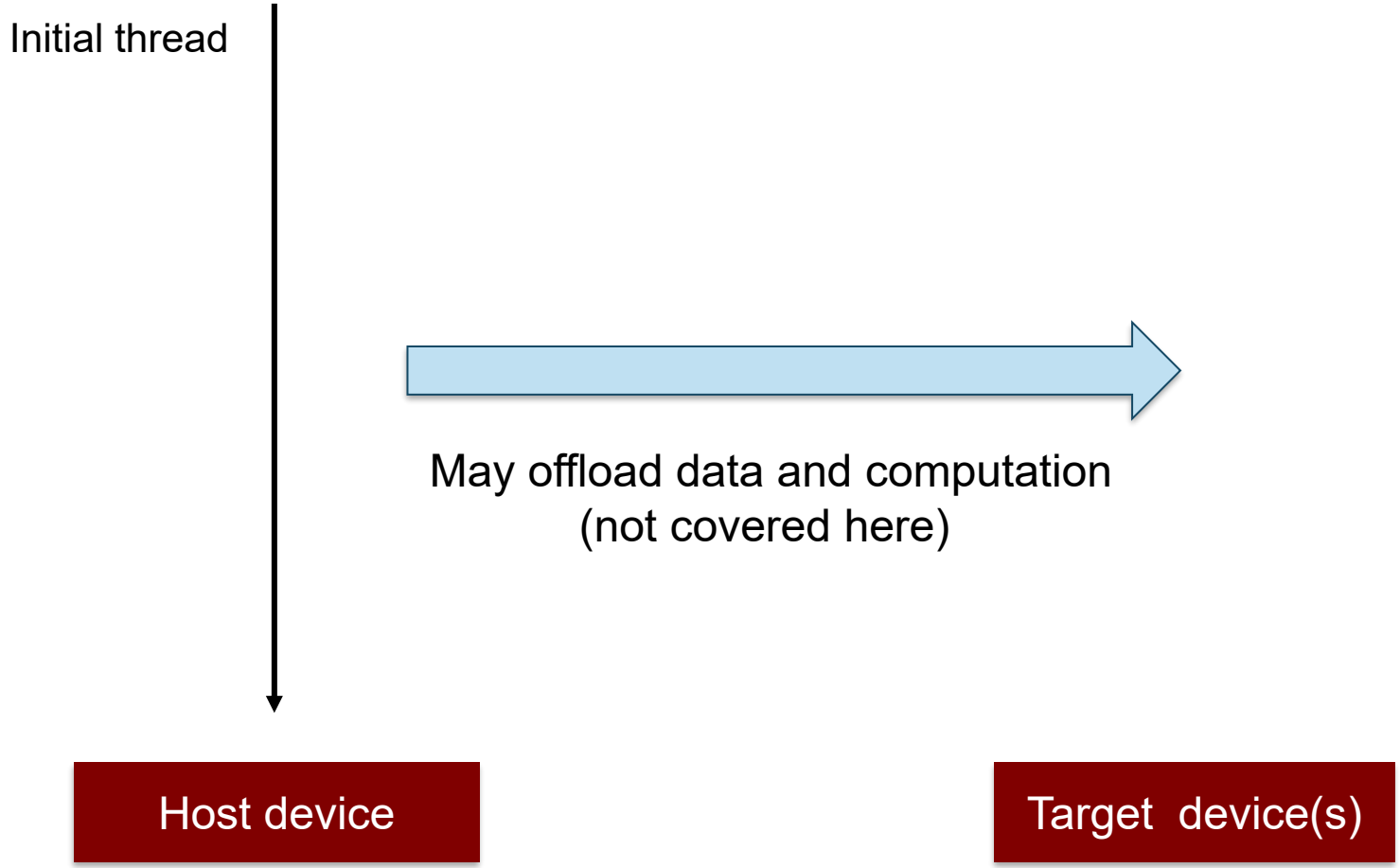
Nested parallelsim

Not covered



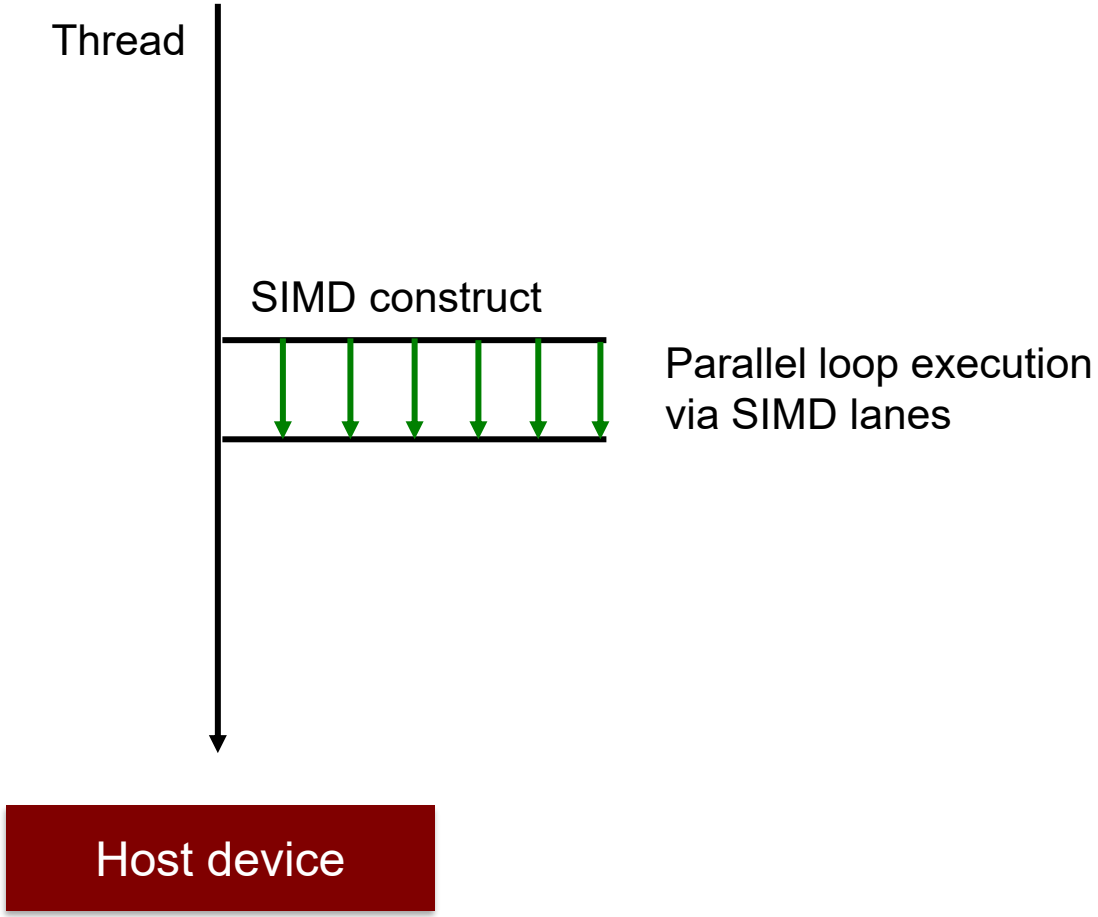
Offloading computations

Not covered



SIMD parallelsim

Not covered



OpenMP feature summary

Covered in course

- Loop-level work sharing
- SPMD-style parallelism
- Tasking

Not covered in this course

- Nested parallelism
- Offloading to accelerator
- SIMD parallelism
- Alternative work sharing methods

Multiply add or daxpy

(double-precision a*x plus y)

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

```
void daxpy(double z[], double a, double x[], double y, int n)
{
  /* for simplicity, we have y as a scalar variable */

  int i;

  for ( i = 0; i < n; i++ )
    z[i] = a * x[i] + y;
}
```

- No dependences
- Result of one iteration does not depend on results of others

Parallel daxpy

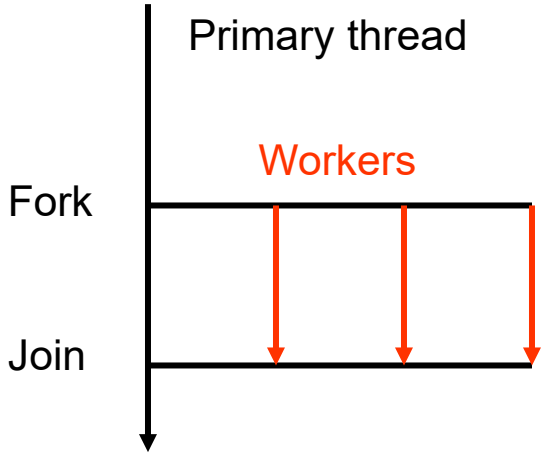

```
void daxpy(double z[], double a, double x[], double y, int n)
{
  /* for simplicity, we have y as a scalar variable */
  int i;

#pragma omp parallel for
  for ( i = 0; i < n; i++ )
    z[i] = a * x[i] + y;
}
```

- Parallel-for construct specifies concurrent execution of the loop
- Runtime system creates set of threads and distributes iterations

Runtime behavior

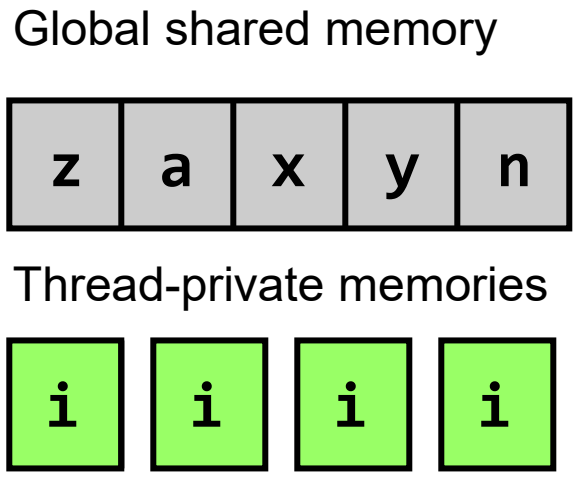

```
void daxpy(…)
{
  int i;
#pragma omp parallel for
  for ( i = 0; i < n; i++ )
    z[i] = a * x[i] + y;
}
```



- Each thread executes a distinct subset of the iterations
  - Distribution of iterations is not specified here

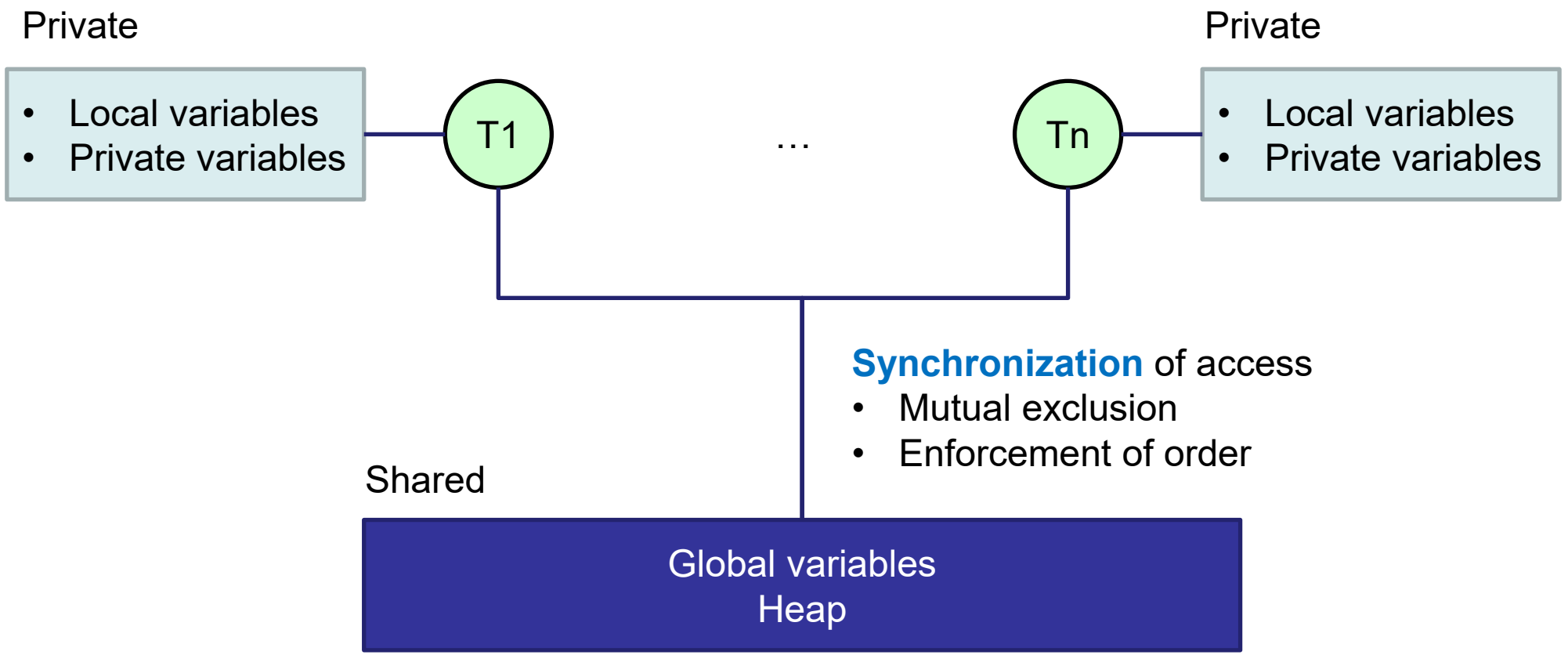
Communication and data sharing


```
void daxpy(…)
{
  int i;
#pragma omp parallel for
  for ( i = 0; i < n; i++ )
    z[i] = a * x[i] + y;
}
```



- Variables or array elements never updated concurrently – except for loop index
- Loop index is private by default – each thread has a private copy
- Update of array must be complete when primary thread resumes execution
  - Ensured by implicit barrier at the end of the construct

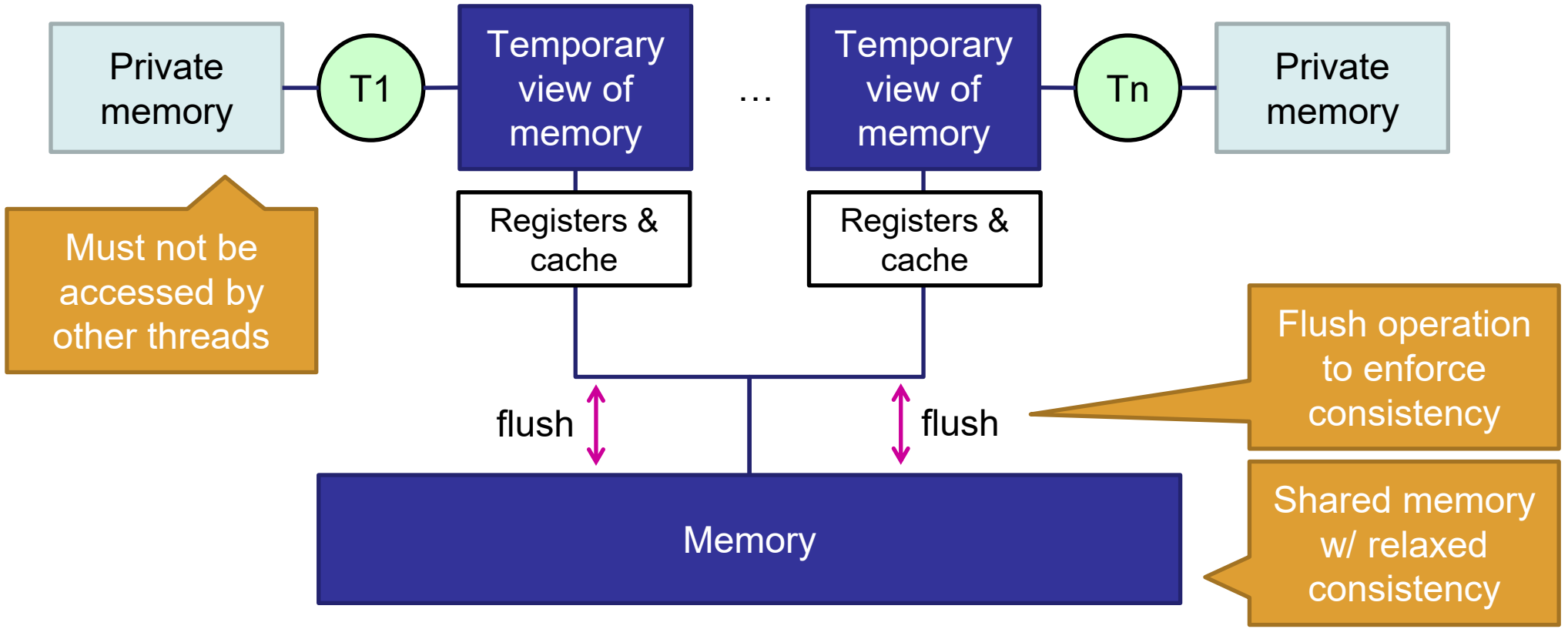
Data environment & synchronization



Control structures

- Create a team of threads executing concurrently (parallel construct)
- Distribute work among an existing team of threads (work-distribution constructs)
- Generate a task (task construct)
- Offload computation to a device (target construct)
- Generate vector code (SIMD construct)
- Synchronize threads

Memory model



Construct vs. region

Construct

- Lexical or static extent
- Code lexically enclosed

Region

- Dynamic extent
- Lexical extent plus code of subroutines called from within the construct

```
void subroutine();
{
  printf("Hello world!\n");
}

int main(int argc, char* argv[])
{
#pragma omp parallel
  {
    subroutine();
  }
}
```

Variable access

- Single access to a variable may be implemented with multiple loads or stores
  - Not guaranteed to be atomic
- Accesses to variables smaller than the implementation-defined minimum size or to C or C++ bit-fields may be implemented by reading, modifying, and rewriting a larger unit of memory
  - May interfere with updates of variables or fields in the same unit of memory

Internal control variables (ICVs)

- Control the behavior of an OpenMP program
- Examples
  - Number of threads to use for parallel execution
  - Scheduling strategy for parallel loops
- Initialized by the implementation
- User assigns values through
  - OpenMP environment variables
  - Calls to OpenMP API routines
  - Clauses in compiler directives
- Program can retrieve values only through API calls

Summary

- Fork–join execution model
- Runtime abstractions
  - Control structures
  - Data environment
  - Synchronization
- Relaxed memory model
- Internal control variables

Loop-level parallelism

Loop-level parallelism

- Executing the iterations of a loop concurrently
- Fine-grained – units of work are small
- Small, localized changes to the source code
- Loop-level parallelism in OpenMP
  - Worksharing-loop construct (`#pragma omp for`)
  - Loop construct (`#pragma omp loop`)
  - Maybe combined with parallel construct

```
void daxpy(…)
{
  int i;
#pragma omp parallel for
  for ( i = 0; i < n; i++ )
    z[i] = a * x[i] + y;
}
```

Shortcut for **for construct** nested inside a **parallel constuct**

Outline

- Parallel-for construct
- Data sharing
- Data dependences
- Scheduling strategies
- Summary

(Parallel) for construct


```
#pragma omp parallel for [ clause [[,] clause ] …] new-line
    for-loops
```

Can be loop nest

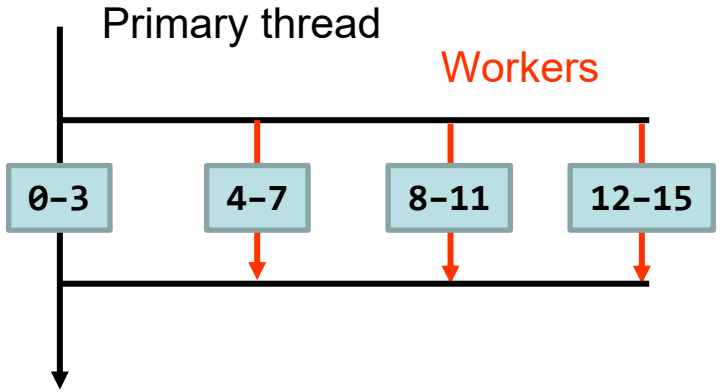
Loops immediately following the directive are parallelized

Is actually short cut for

```
#pragma omp parallel new-line
  {
#pragma omp for [ clause [[,] clause ] …] new-line
    for-loops
  }
```

Runtime behavior

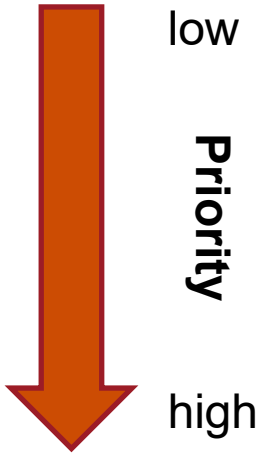

```
  int i;
#pragma omp parallel for
  for ( i=0; i<16; i++)
    […]
```



- Variables are either shared or private to each thread
- Implicit barrier at the end of the loop

How to specify the number of threads

`setenv OMP_NUM_THREADS n`	• Before program start
`omp_set_num_threads(n)`	• At runtime • Affects only subsequent parallel constructs
`num_threads` clause	• Individually for a construct




```
#pragma omp parallel for num_threads(4)
   […]
```

Canonical loop structure

- Allows number of iterations to be computed at loop entry
- Program must complete all iterations of the loop
  - No **break** statement
  - No exception thrown inside and caught outside the loop
- Exiting current iteration and starting next one possible
  - **continue** allowed
- Termination of the entire program inside the loop possible
  - **exit** allowed
- Details in standard

Loop nest

```
/* sum of a row */
#pragma omp parallel for
  for ( int i = 0; i < n; i++ ) {
    a[i][0] = 0;
    for ( int j = 1; j < m; j++ )
      a[i][0] = a[i][0] + a[i][j];
  }

/* smoothing function */
  for ( int i = 1; i < n; i++ ) {
#pragma omp parallel for
    for ( int j = 1; j < m - 1; j++ )
      a[i][j] = ( a[i-1][j-1] + a[i-1][j] + a[i-1][j+1] ) / 3.0;
  }
```

- (Parallel) for construct refers only to the loop immediately following it unless collapse clause is specified

Clauses

Purpose	Clause
Conditional parallelization	if
Number of threads	num_threads
Data sharing attributes	shared, private, default, firstprivate, lastprivate, reduction, linear
Copying thread-private variable values of the primary thread	copyin
Memory allocator for private variables	allocate
Thread affinity	proc_bind
Loop scheduling	schedule
Parallelization of nested loops	collapse
Execution order of loop iterations	ordered

Data sharing clauses

General rules

- Usually keyword plus comma separated list of variables
  - Example: **shared(x,y,z)**
- Reduction clause also specifies operation
  - Example: **reduction(+: x,y)**
- Clauses apply only to the the construct in which they appear (lexical extent)
- Variable must be visible in construct
- Can be used only for entire objects – but not for single components
  - Arrays, C/C++ struct, or class objects
  - Exception: static class variables in C++
- Variables can only appear in one clause
  - Exception: can appear in both firstprivate and lastprivate clause

Data sharing clauses

Shared vs private

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

Shared

- Variable is shared among all threads
- Single instance in shared memory
- All modifications update this single instance

- Caveat: pointers – only pointer itself is shared but not necessarily the object it points to

Private

- Each thread allocates a private copy
- References within (the lexical extent of) the construct read or write private copy
- Value undefined upon construct entry
- Exceptions: loop index variable & C++ class objects
- Value undefined upon construct exit

Default sharing semantics

= shared

Shared

- All variables visible upon entry of the construct
- Static C/C++ variables declared within the dynamic extent
- Heap allocated memory

Private

- Local variables declared within the region
- Loop index variable of the work-shared loop

Example


```
void caller(int a[], int n)
{
  int i, j, m = 3;

#pragma omp parallel for
  for (i = 0; i < n; i++) {
    int k = m;

    for (j = 1; j <= 5; j++ )
      callee(&a[i], &k, j);
  }
}
```

```
extern int c = 4;

void callee(int *x, int *y,
            int z)
{
  int ii;
  static int cnt;

  cnt++;
  for (ii = 0; ii < z; ii++)
    *x = *y + c;
}
```

Use of j and cnt is not safe

Sharing semantics

i	n	k	m	j	a	x	y	z	ii	cnt	*x	*y	c
prv	shd	prv	shd	shd	shd	prv	prv	prv	prv	shd	shd	prv	shd

Default sharing semantics

- Can be specified using default clause
  - **default(shared | firstprivate | private | none)**
- Default shared – is already the default
- Default firstprivate or private – makes the default of variables that are not explicitly classified by a data sharing attribute firstprivate or private
- Default none – all variables must be explicitly specified

Reduction clause


```
  sum = 0;

#pragma omp parallel for reduction(+:sum)
  for ( i = 0; i < n; i++ )
    sum = sum + b[i];
```

- Applications
  - Compute sum of array
  - Find the largest element
- Operator should be commutative & associative
- Reduction variables are initialized to the identity element

Predefined reduction operators

Operator	Initial Value
+	0
*	1
-	0
&	all bits on
|	0
^	0
&&	1
||	0
max	Least representable value in the reduction-list item type
min	Largest representable value in the reduction-list item type

Reduction clause


```
sum = 0;
#pragma omp parallel shared(sum) private(psum)
  {
    psum = 0;
#pragma omp for
    for ( i = 0; i < n; i++ )
      psum = psum + b[i];
#pragma omp critical
   sum = sum + psum;
  }
```

Possible translation using critical construct

- Not all valid operators are commutative & associative
  - Floating-point addition is not associative
  - Subtraction – can be rewritten using addition
  - Logical operators (&&, ||) in C(++) may not evaluate their right operand

Private variables

Initialization and finalization

- Default avoids copying
  - Undefined initial value of private copy upon loop entry
  - Undefined value of primary thread's copy after loop exit

Clause	Purpose
firstprivate	Initializes private copy to the value of primary thread's copy prior to entering the construct
lastprivate	Writes value of sequentially last iteration back to primary thread’s copy

- A variable can appear in both clauses
  - Exception to the rule that variable can appear in at most one clause

Private variables

Initialization and finalization (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

```
#pragma omp parallel for lastprivate(i)
  for ( i = 0; i < n-1; i++ )
      a[i] = b[i] + b[i+1];

a[i] = b[i];
```

- Firstprivate variables initialized once per thread – not once per iteration
- If a compound object was declared lastprivate, then elements not assigned in the sequentially last iteration have undefined value

Data dependences

- Parallelization must preserve the program's correctness
  - May be affected by data dependences
- They can exist
  - Between output and input data
  - Among intermediate results
  - On the order in which loop iterations are executed
- How to detect them?
- How to remove them?

Data dependences (2)

- A data dependence exists between two memory accesses if
  - They access the same memory location
  - At least one of them writes the location

```
for ( i = 1; i < n; i++ )
    a[i] = a[i] + a[i-1];
```

- Location can be anywhere – memory or file
- Data dependences in parallel programs may cause a race condition

Outcome of the program depends on the relative ordering of execution of operations on two or more threads

Example


```
#pragma omp parallel for
  for ( i = 1; i <= 2; i++ )
       a[i] = a[i] + a[i-1];
```

```
a[0] = 1;
a[1] = 1;
a[2] = 1;
```

Initial values

`a[2]` is computed using `a[1]`'s new value

```
a[0] = 1;
a[1] = 2;
a[2] = 3;
```

`a[2]` is computed using `a[1]`'s old value

```
a[0] = 1;
a[1] = 2;
a[2] = 2;
```

Are these two the only options?

Dependence detection

- Different loop iterations executed in parallel
- Same loop iteration executed in sequence
- Important for parallelization are loop-carried dependences
  - Dependences between different iterations of the same loop

No dependence	Dependence
• If location is only read • If location is accessed in only one iteration	• If location is accessed in more than one iteration and written in at least one of them

Dependence detection (2)

- Scalar variables are easy – well-defined name
- Arrays more difficult – array index may be computed at runtime

- **Rule of thumb**: loop can be parallelized if
  - All assignments are to arrays
  - Each element is assigned in at most one iteration
  - No iteration reads an element assigned by any other iteration

Dependence detection (3)

- Automatic detection (of their absence)
  - Use polyhedral compiler (e.g., PLUTO[1])
- Semi-automatic detection
  - Often array indices are linear expressions
  - Use index expressions and loop bounds to form system of linear inequalities
  - Use standard techniques to solve the system, e.g.,
    - Integer programming
    - Fourier–Motzkin projection
- Manual inspection

1) https://pluto-compiler.sourceforge.net

Dependence detection (4)

- Loop nests – usually the outermost loop is parallelized
  - Dependences might require consideration of multiple indices

```
/* this matrix multiply can be safely parallelized */
for ( i = 0; i < n; i++ )
   for ( j = 0; j < n; j++ ) {
      c[i][j] = 0;
      for ( k = 0; k < n; k++)
         c[i][j] = c[i][j] + a[i][k] * b[k][j];
   }
```

- Analysis should cover entire dynamic extent of a loop
  - Assignment to shared variables in subroutines
  - C/C++: global and static variables

Dependence classification

- Classifying a dependence to determine
  - Whether it needs to be removed
  - Whether it can be removed
  - Which techniques can be used to remove it
- Loop-carried vs. non-loop-carried dependences
  - Non-loop carried dependences not dangerous in many cases
  - Conditional assignment can cause a loop-carried dependence to look like a non-loop-carried dependence

```
x = 0;
for ( i = 0; i < n; i++ ) {
    if ( f(i) ) x = new;
    b[i] = x;
}
```

Dependence classification (2)

- Two accesses A1 and A2 (often two statements) of the same storage location
- A1 comes before A2 in a serial execution of the loop
- Four possibilities

	A1 = R	A1 = W
A2 = R	RAR = no dependence	RAW = flow dependence
A2 = W	WAR = anti-dependence	WAW = output dependence

R = read, W = write, A=after

Flow dependence

- A1 writes the location
- A2 reads the location

} RAW

- Result of A1 flows to A2 → flow dependence
- The two accesses cannot be executed in parallel

Anti-dependence

- A1 reads the location
- A2 writes the location

} WAR

- Reuse of the location instead of communication through the location
- Opposite of flow dependence → anti-dependence
- Parallelization
  - Give each iteration a private copy of the location
  - Initialize copy with value A1 would have read during serial execution

Output dependence

- A1 writes the location
- A2 writes the location

} WAW

- Only writing occurs → output dependence
- Parallelization
  - Make location private
  - Copy sequentially last value back to shared copy at the end of the loop

Example


```
1    for ( i = 1; i < n - 1; i++ ) {
2        x = d[i] + i;
3        a[i] = a[i+1] + x;
4        b[i] = b[i] + b[i-1] + d[i-1];
5        c[2] = 2 * i;
6    }
```

Memory location	Earlier access			Later access			Loop carried?	Kind of depend.
	Line	Iteration	r/w	Line	iteration	r/w		
x	2	i	write	3	i	read	no	flow
x	2	i	write	2	i+1	write	yes	output
x	3	i	read	2	i+1	write	yes	anti
a[i+1]	3	i	read	3	i+1	write	yes	anti
b[i]	4	i	write	4	i+1	read	yes	flow
c[2]	5	i	write	5	i+1	write	yes	output

Removing anti-dependences

- Anti-dependence between `a[i]` and `a[i+1]`
- Also x read and written in different iterations
- Parallelization
  - Privatize x
  - Create temporary array `a2`
- Overhead
  - Memory
  - Computation

```
for ( i = 0; i < n - 1; i++ ) {
  x = b[i] - c[i];
  a[i] = a[i+1] + x;
}
```

```
#pragma omp parallel for
  for ( i = 0; i < n - 1; i++ ) {
    a2[i] = a[i+1];
  }

#pragma omp parallel for private(x)
  for ( i = 0; i < n - 1; i++ ) {
    x = b[i] - c[i];
    a[i] = a2[i] + x;
  }
```

Removing output dependences

- Output dependence on d[1]
- Live-out locations d[1], x

```
for ( i = 0; i < n-1; i++ ) {
  x = b[i] - c[i];
  d[1] = 2 * x;
}
y = x + d[1] + d[2];
```

- Parallelization via lastprivate temporary

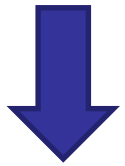
```
#pragma omp parallel for lastprivate(x, d1)
  for ( i = 0; i < n-1; i++ ) {
    x = b[i] - c[i];
    d1 = 2 * x;
  }
  d[1] = d1;
  y = x + d[1] + d[2];
```

Removing flow dependences

- A2 depends on result stored during A1
- Dependence cannot always be removed
- Three techniques
  - Reduction operations
  - Induction-variable elimination
  - Loop skewing

Reduction operations


```
for ( i = 0; i < n; i++ ) {
    x = x + a[i];
}
```



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

Induction-variable elimination

- Special case of reduction operations
- Value of reduction variable is simple function of loop index
- Uses of the variable can be replaced by simple expression containing the loop index variable

Example


```
for ( i = 2; i < n; i++ ) {
    fib[i] = fib[i-2] + fib[i-1];
}
```

Greetings to linear algebra:
https://math.hawaii.edu/~pavel/fibonacci

```
#pragma omp parallel for
for ( i = 2; i < n; i++ ) {
    double p1 = pow(1.0 + sqrt(5.0), i);
    double p2 = pow(1.0 - sqrt(5.0), i);
    double div = pow(2.0, i) * sqrt(5.0);
    fib[i] = (p1 - p2) / div;
}
```

Loop skewing

- Convert loop carried dependence into non-loop-carried one
- Shift (“skew”) access to a variable between iterations

```
for ( i = 1; i < n; i++ ) {
    b[i] = b[i] + a[i-1];
    a[i] = a[i] + c[i];
}
```

Loop-carried flow dependence
from read of `a[i-1]` to write of `a[i]`

```
b[1] = b[1] + a[0];
#pragma omp parallel for shared(a, b, c)
  for ( i = 1; i < n-1; i++ ) {
    a[i]   = a[i] + c[i];
    b[i+1] = b[i+1] + a[i];
  }
a[n-1] = a[n-1] + c[n-1];
```

Non-removable dependences

- Recurrences – difficult or impossible to parallelize

```
for ( i = 1; i < n; i++ ) {
    a[i] = (a[i-1] + a[i])/2;
}
```

  - Alternative (parallelizable) algorithm may exist (see Fibonacci numbers)
- Non-removable dependence in loop nest
  - Try to parallelize loop not involving the recurrence
- Fissioning
  - Splitting the loop into a parallelizable and non-parallelizable part
  - Parallelize only part of the loop

Non-removable dependences (2)

- Scalar expansion
  - Computation depends on scalar computed in each iteration – scalar computation cannot be parallelized
  - Compute array of all scalar values sequentially
  - Parallelize remaining part of the loop

Remarks

- When removing a dependence
  - Don’t violate other dependences
  - Remove new dependences introduced as a consequence of removing an old one as well
- Balance benefit against
  - Computational cost
  - Memory cost
- Dependence classification also applies to other forms of parallelism (e.g., coarse-grained parallelism)

Parallel overhead

- Compare benefit of parallelization to cost of
  - Creating a team of threads
  - Distributing the work among the team members
  - Synchronizing at the end of the loop
- Conditional parallelization using OpenMP if clause

```
#pragma omp parallel for if ( n > MIN_TRIP_COUNT )
  for ( i = 0; i < n; i++ )
    z[i] = a * x[i] + y;
```

Parallel overhead (2)

- Loop nests
  - Parallel overhead depends on the number of times a loop is reached
  - Inner loops are reached more often
  - Parallelizing outermost loop helps minimize parallel overhead
- If outermost loop cannot be parallelized (e.g., because of data dependences)
  - Source transformation: loop interchange
  - Respect data dependences
  - Transformation can change the order in which results are computed
  - Result of interchange may show different cache behavior

Scheduling

- The way loop iterations are distributed across the threads of a team is called schedule
- A loop is most efficient if all threads finish at about the same time
  - Threads should do the same amount of work / have the same load
- If each iteration requires the same amount of work, an even distribution of iterations will be most efficient
  - Default schedule of most implementations

```
#pragma omp parallel for
  for ( i = 0; i < n; i++ )
    z[i] = a * x[i] + y;
```

Variable load per iteration


```
#pragma omp parallel for
  for ( i = 0; i < n; i++ )
    if ( f(i) )
      do_big_work(i);
    else
      do_small_work(i);
```

- Even distribution of iterations may cause load imbalance
- Load imbalance causes synchronization delay at the end of the loop
  - Faster threads have to wait for slower threads
- Execution time will increase ➔ chose alternate scheduling strategy

Static vs. dynamic scheduling

Static

- Distribution (deterministically) at loop-entry time based on
  - Number of threads
  - Total number of iterations
  - Index of an individual iteration
- Low scheduling overhead ✓
- Less flexible ✗

Dynamic

- Distribution during execution of the loop
  - Each thread is assigned a subset of the iterations at the loop entry
  - After completion each thread asks for more iterations
- Synchronization overhead ✗
- Can easily adjust to load imbalance ✓

Schedule clause


```
schedule([modifier[, modifier]:]kind[,chunk_size])
```

- Distribution of iterations in chunks
- Chunks may have different **sizes**
- Chunks assigned either statically or dynamically
- Different assignment algorithms

- Kind
  - **static, dynamic, guided, auto, runtime**
- Modifier
  - **monotonic, nonmonotonic, simd**

Scheduling strategies (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

Static without chunk size

- One chunk of iterations per thread
- All chunks (nearly) equal size

Static with chunk size

- Chunks with specified size assigned in round-robin fashion

Dynamic

- Threads request new chunks dynamically
- Default chunk size is 1

Guided

- First chunk has implementation-dependent size
- Size of subsequent chunks decreases exponentially
- Chunks are assigned dynamically
- Chunk size specifies minimum size, default is 1

Runtime

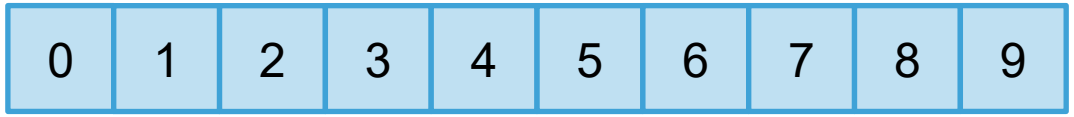
- Via environment variable or API call

Auto (no chunk size)

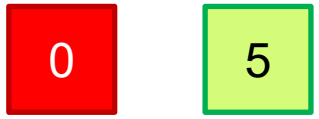
- Implementation decides

Example

- 10 iterations, 2 threads (red & green)



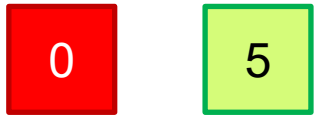
Example



- 10 iterations, 2 threads (red & green)

Static without chunk size

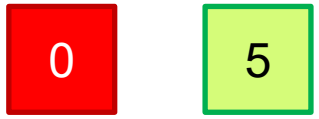
Example



- 10 iterations, 2 threads (red & green)

Static with chunk size = 1

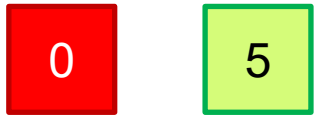
Example



- 10 iterations, 2 threads (red & green)

Dynamic with default chunk size (=1)

Example



- 10 iterations, 2 threads (red & green)

Guided

Correctness and scheduling

- Correctness of program must not depend on scheduling strategy
- Dependences might cause errors only under specific scheduling strategies
- Dynamic scheduling might produce wrong results only occasionally

Comparison

Static

- Cheap
- May cause load imbalance

Dynamic

- More expensive than static scheduling
- Small chunk size increases cost
- One synchronization per chunk
- Small chunk size can balance the load better

Guided

- Number of chunks increases only logarithmically with the number of iterations
- Most costly computation of chunk size

Runtime

- Allows testing of different scheduling strategies without recompilation

Summary loop-level parallelism

- Incremental parallelization of serial code using loop-level parallelism
- Potential hazards affecting correctness
  - Incorrect data sharing
  - Data dependences
- Scheduling strategy
  - Overhead vs. load balance
- No covered: thread affinity

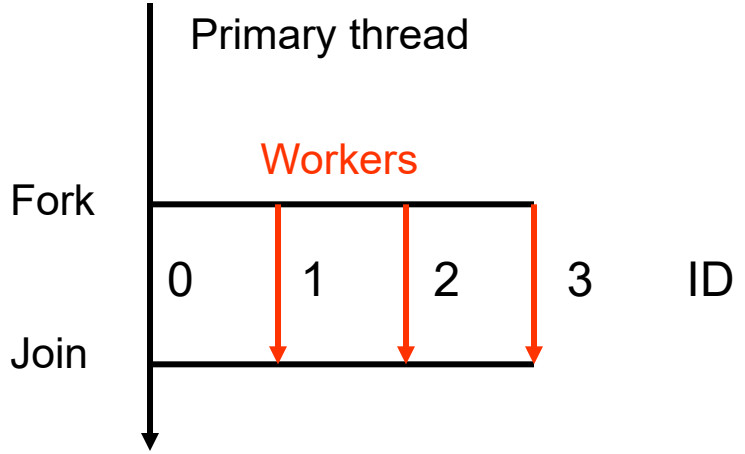
SPMD-style parallelism

SPMD-style parallelism

- Loop-level parallelism is a local concept
- Program usually consists of multiple loops and non-iterative constructs
- Need to parallelize larger portions of a program
- Two different ways of parallelism
  - Parallel construct provides SPMD-style replicated execution
    (**SPMD** = Single Program Multiple Data)
  - Work-distribution constructs distribute work across multiple threads

SPMD-style parallelism

- Single Program Multiple Data
- Same code executed by multiple threads
- Threads are enumerated
- Each thread can query its ID
- Different ID may lead to different control flows




```
if (omp_get_thread_num() == x) {
  do_something();
} else {
  do_something_else();
}
```

Parallel construct

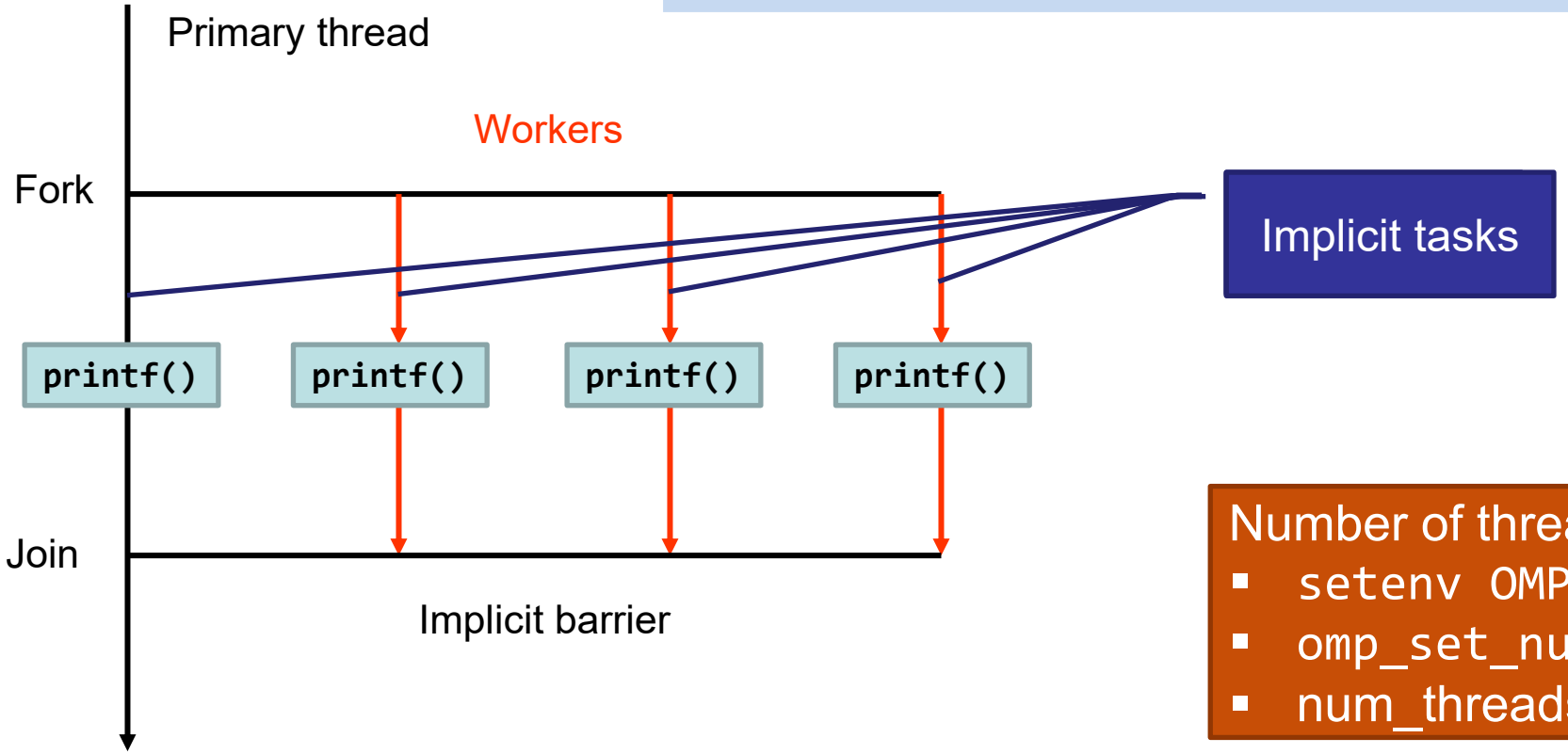
Fundamental construct that starts parallel execution

```
#pragma omp parallel [ clause [[,] clause ] …] new-line
        structured-block
```

Single entry at the top and a single exit at the bottom.

Execution model


```
#pragma omp parallel
  printf("Hello world!\n");
```



Number of threads can be set like before:

- `setenv OMP_NUM_THREADS n`
- `omp_set_num_threads(n)`
- num_threads clause

Parallel vs. parallel for

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

Parallel construct

- n outputs per thread
- Work replication

```
#pragma omp parallel
{
  int i;
  for ( i=0; i<n; i++)
    printf("Hello world!\n");
}
```

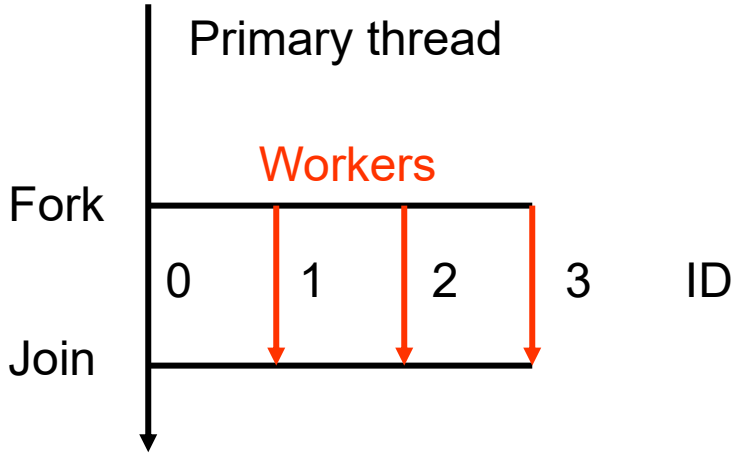
Parallel for construct

- n outputs total
- Work distribution

```
int i;
#pragma omp parallel for
  for ( i=0; i<n; i++)
    printf("Hello world!\n");
```

Thread numbers

- Within a **parallel** region, thread numbers uniquely identify each thread
- Thread numbers range from zero for the primary thread up to (team size - 1)
- A thread may obtain its own thread number using **`omp_get_thread_num()`**



Clauses (parallel construct)

Purpose	Clause
Conditional parallelization	if
Number of threads	num_threads
Data sharing attributes	shared, private, default, firstprivate, reduction
Copying thread-private variable values of the primary thread	copyin
Memory allocator for private variables	allocate
Thread affinity	proc_bind

Construct vs. region revisited

Construct

- Lexical or static extent
- Code lexically enclosed

Region

- Dynamic extent
- Lexical extent plus code of subroutines called from within the construct

```
void subroutine();
{
  printf("Hello world!\n");
}

int main(int argc, char* argv[])
{
#pragma omp parallel
  {
    subroutine();
  }
}
```

IMPORTANT: Data-sharing clauses refer only to construct

Private clause applies only to construct


```
int my_start, my_end;

void work() {    /* my_start and my_end are undefined */
  printf("My subarray is from %d to %d\n", my_start, my_end);
}

int main(int argc, char* argv[]) {
#pragma omp parallel private(my_start, my_end)
  {
    /* get subarray indices */
    my_start = get_my_start(omp_get_thread_num(), omp_get_num_threads());
    my_end   = get_my_end  (omp_get_thread_num(), omp_get_num_threads());
    work();
  }
}
```

Passing private variables as arguments


```
int my_start, my_end;

void work(int my_start, int my_end) {
  printf("My subarray is from %d to %d\n", my_start, my_end);
}

int main(int argc, char* argv[]) {
#pragma omp parallel private(my_start, my_end)
  {
    /* get subarray indices */
    my_start = […]
    my_end   = […]
    work(my_start, my_end);
  }
}
```

Cumbersome for long call paths

Threadprivate directive

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

```
int my_start, my_end;
#pragma omp threadprivate(my_start, my_end)

void work() {
  printf("My subarray is from %d to %d\n", my_start, my_end);
}

int main(int argc, char* argv[]) {
#pragma omp parallel
  {
    my_start = […]
    my_end   = […]
    work();
  }
}
```

Threadprivate directive (2)

- Makes a variable private to a thread across the entire program
- Initialized once prior to the first reference to that copy
- Value persists across multiple parallel regions
  - Exceptions: dynamic threads and change of the number of threads
- Thread with same thread number will have same copy
- Thread-private variables cannot appear in any data-sharing clauses except for copyin or copyprivate
- Many special rules

copyin clause

- Copies value of primary thread's copy to every thread-private copy when entering a parallel construct
- A variable in a copyin clause must be a threadprivate variable

```
int c;
#pragma omp threadprivate(c)

int main(int argc, char* argv[]) {
  c = 2;
#pragma omp parallel copyin(c)
  {
    /* c has value 2 in all thread-private copies */
    […] = c;
  }
}
```

Assignment of work in parallel regions

- Domain decomposition
  - Assignment based on thread number and total number of threads
- Work-distribution constructs
  - For construct – division of loop iterations
  - Parallel sections construct – distribution of distinct pieces of code
  - Single construct – identification of code that needs to be executed by a single thread only
- Task construct
  - Defines unit of work to be assigned to a thread

} Not covered

Domain decomposition


```
#pragma omp parallel private(nthreads, iam, chunk, start, end)
  {
    nthreads = omp_get_num_threads();
    iam      = omp_get_thread_num();
    chunk    = (n + (nthreads - 1))/nthreads;
    start    = iam * chunk;
    end      = n < (iam + 1) * chunk ? n : (iam + 1) * chunk;
    for ( i = start; i < end; i++ )
      do_work(i);
  }
```

- `omp_get_num_threads()` returns the number of threads in the team
- `omp_get_thread_num()` returns individual thread identifier in {0,…,n-1}

For construct


```
#pragma omp for [ clause [[,] clause ] …] new-line
   for-loops
```

- Divides iterations of the following loop among the threads in a team
- Subset of the clauses of the parallel for construct
  - Exception: nowait clause - disables the implicit barrier at the end
- Equivalent behavior

Collapse clause

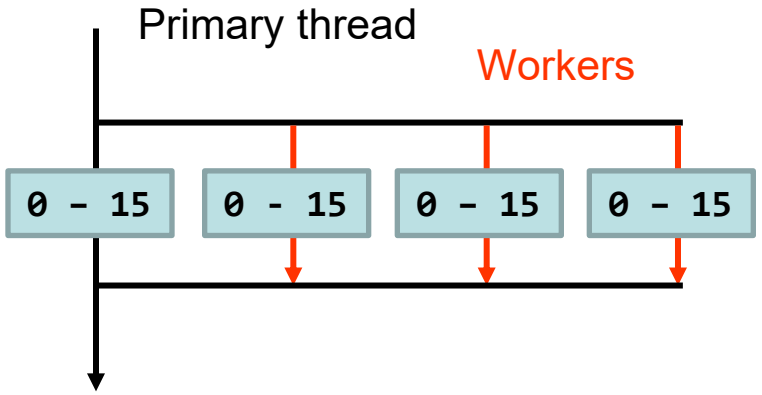
- Parameter specifies number of associated nested loops
- The iterations of all associated loops are collapsed into one larger iteration space to be divided according to the schedule clause
- The sequential execution of the iterations in all loops determines the order of iterations in the collapsed space

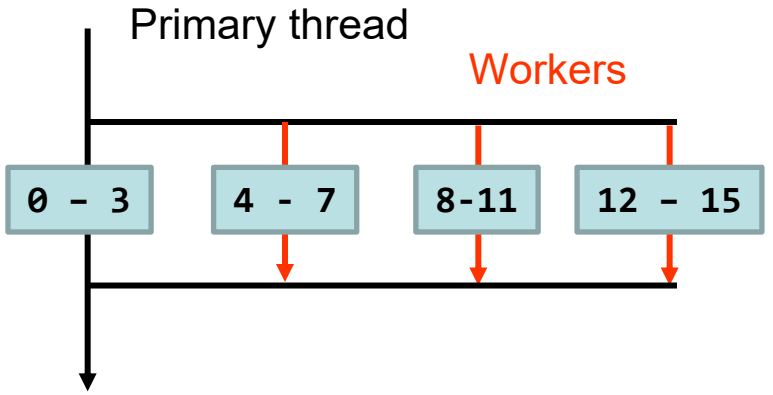
```
#pragma omp for collapse(2) schedule(static,1)
  for (i=0; i<2; i++)
     for (j=0; j<5; j++)
          a[i][j] = b[i][j];
}
```

0:0	0:1	0:2	0:3	0:4	1:0	1:1	1:2	1:3	1:4

Combined i:j iteration space

Replicated vs. divided execution






```
#pragma omp parallel
  {
    int i;
    for ( i=0; i<16; i++)
      […]
  }
```

```
#pragma omp parallel
  {
    int i;
#pragma omp for
    for ( i=0; i<16; i++)
      […]
  }
```

Parallel for construct


```
#pragma omp parallel for [ clause [[,] clause ] …] new-line
    for-loops
```

Is actually short cut for

```
#pragma omp parallel new-line
  {
#pragma omp for [ clause [[,] clause ] …] new-line
    for-loops
  }
```

Suppressing replication

- Some tasks cannot be executed by multiple threads
  - Example: file I/O operations, MPI calls
- Single construct requires that enclosed code is executed by a single thread only

```
#pragma omp single [ clause [[,] clause ] …] new-line
     structured-block
```

- Clauses: allocate, private, firstprivate, copyprivate, nowait

Writing intermediate result to a file

- An arbitrary thread is chosen to perform the operation
- Correctness must not depend on the selection of a particular thread
- Remaining threads wait for the selected thread to finish the I/O operation at the implicit barrier unless there is a nowait

```
#pragma omp parallel
  {
#pragma omp for
    for(i=0; i<n; i++)
      […]
#pragma omp single
    write_intermediate_result();
#pragma omp for
    for(i=0; i<n; i++)
      […]
  }
```

Copyprivate clause


```
double x;
#pragma omp threadprivate(x)

void init() {
  double a;
#pragma omp single copyprivate(a,x)
 {
   a = […];
   x = […];
 }
  use_values(a, x);
}
```

- Broadcasts values acquired by a single thread directly to all instances of the private variables in the other threads
- Helpful if using a shared variable is difficult, e.g., in recursion requiring a different variable at each level

Why is single a work-distribution construct?

- Enclosed code is executed exactly once, that is, it is not replicated
- It must be reached by all threads in a team
- All threads must reach all work-distribution constructs in the same order, including the single construct
- Has implicit barrier
- Has nowait clause

Restrictions

- Contents of work-distribution construct must have block structure
  - Single entry at the top and a single exit at the bottom
- Each work-distribution construct and barrier must be encountered by all threads in a team or by none at all

```
#pragma omp parallel
  {
    if (omp_get_thread_num() != 0) /* illegal */
#pragma omp for
      for (i=0; i<n; i++)
        […];
  }
```

Nesting of work-distribution constructs


```
#pragma omp for
  for (i=0; i<n; i++)
    {
#pragma omp for                    /* illegal */
       for (j=0; j<m; j++)
         a = […];
    }
```

- Nesting of work-sharing constructs is illegal in OpenMP
  - A thread executing inside a work-distribution construct executes its portion of work alone, therefore further distribution of work pointless
- Parallelization of nested loops – via collapse clause, nested parallel regions, or manual parallelization
- No explicit barrier in work-sharing constructs

Orphaned work-distribution constructs


```
void initialize(double a[], int n) {
  int i;
#pragma omp for
  for (i=0; i<n; i++)
    a[i] = 0.0;
}

int main(int argc, char* argv[]) {
  […]
#pragma omp parallel
  {
    initialize(a,n);
    […]
  }
}
```

Work-distribution constructs not in the lexical extent are called **orphaned**

Behavior when reached

From within a parallel region

- Almost as when inside the lexical extent
- Sharing clauses of the surrounding parallel region apply only to its lexical extent
  - Sharing semantics in orphaned constructs may be different
- C/C++ global variables are shared by default

From within serial code

- Almost the same as without work-distribution directive
- Single serial thread behaves like a parallel team composed of only one primary thread
- Can safely be invoked from serial code – directive is essentially ignored

Summary SPMD-style parallelism

Loop-level parallelism

- Only work distribution
- Loop iterations must be independent
- Easier to understand and use
- Allows incremental parallelization
- Limited to loops

SPMD-style parallelism

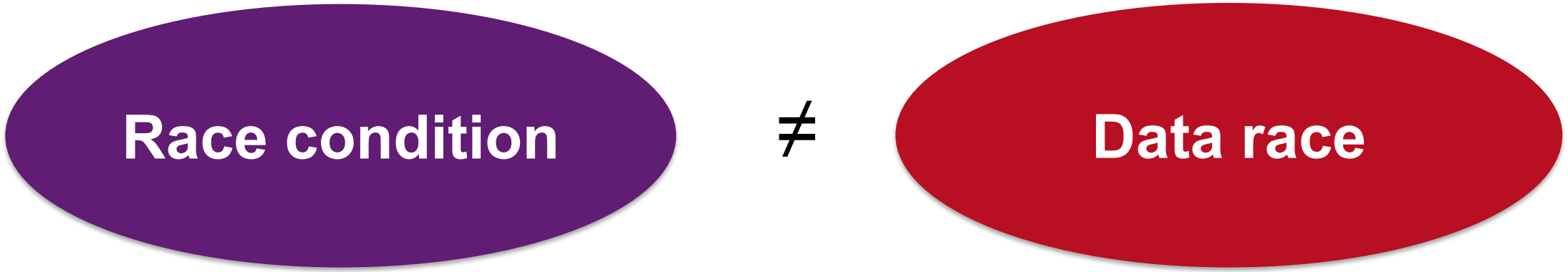
- Combination of replicated execution with work distribution
- More complex to use
- More generic
- Applicable to larger code regions
- Less parallel overhead for spawning threads and synchronization

Synchronization

Synchronization

- Implicit communication in shared-memory programming
  - Read/write operations on variables in shared address space
  - Requires coordination (i.e., synchronization)
- Two types
  - Mutual-exclusion synchronization
  - Event synchronization

Race condition and data race



- Unenforced relative timing of concurrent operations
- Results in **non-deterministic** behavior

- Non-atomic concurrent access (with at least one write) to memory location
- Results in **undefined** behavior

Race condition

Outcome of the program depends on the relative ordering of execution of operations on two or more threads

- Causes non-deterministic program behavior
  - Failure in programs expected to be deterministic
- Benign in programs where all possible outcomes are acceptable

Example


```
Thread 0

#pragma omp atomic write
  x = 1;
```

```
Thread 1

#pragma omp atomic write
  x = 2;
```

- Not deterministic but no atomicity violation

  ➔ race condition but no data race

Round-off errors

Sum of all elements in an array

```
  double sum = 0.0;
#pragma omp parallel for reduction(+:sum) schedule(dynamic,1)
  for ( i = 0; i < n; i++ ) {
    sum = sum + a[i];
  }
```

- Result may depend on thread scheduling
  - Floating-point addition is not associative
- May be acceptable – depending on the problem to solve

Data race

The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other.

- Behavior undefined
- Two expression evaluations **conflict** if one of them modifies a memory location and the other one reads or modifies the same memory location
- **Memory location** = either an object of scalar type, or a maximal sequence of adjacent bit-fields all having nonzero width

Find largest element in a list of numbers


```
  cur_max = MINUS_INFINITY;
#pragma omp parallel for
  for ( i = 0; i < n; i++ )
    if ( a[i] > cur_max )
      cur_max = a[i];
```

Find largest element in a list of numbers (2)

- Different results are possible – depending on the relative timing of operations
  ➔ race condition

```
Thread 0

read a[i]                    (12)
read cur_max                 (10)
if (a[i] > cur_max) (12 > 10)
  cur_max = a[i]             (12)
```

Update lost

```
Thread 1

read a[j]                    (11)
read cur_max                 (10)


if (a[j] > cur_max) (11 > 10)
  cur_max = a[j]             (11)
```

Value read or written

Find largest element in a list of numbers (3)


```
Thread 0

[…]
if (a[i] > cur_max)
  cur_max = a[i]
```

```
Thread 1

[…]
if (a[j] > cur_max)
  cur_max = a[j]
```

- The two threads may also update `cur_max` simultaneously

  ➔ data race = result undefined

Atomicity of variable accesses

A single access to a variable may be implemented with multiple load or store instructions, and hence is not guaranteed to be atomic with respect to other accesses to the same variable

Concurrent unsynchronized update of `cur_max` in example

- Will the result be one or the other or a mix of both?
- Perhaps unlikely but who guarantees that it will never happen?
- Example: ancient machine, future machine, long data type

Find largest element in a list of numbers (4)


```
#pragma omp parallel for reduction(max: cur_max)
  for ( i = 0; i < n; i++ )
    if ( a[i] > cur_max )
      cur_max = a[i];
```

Correct version – neither race condition nor data race

- Determine first local maxima and then yield global maximum
- Synchronization hidden inside the OpenMP implementation

Data race but no race condition


```
  found = 0;
#pragma omp parallel for
  for ( i = 0; i < n; i++ ) {
    if ( a[i] == item )
      found = 1;
  }
```

- All updates write the same value
- If item can be found, final value will likely be 1 regardless of concurrent updates
- If item cannot be found, final value will likely be 0
- According to C11, the result is undefined though

Removing data races using privatization


```
#pragma omp parallel for shared(a) private(b)
  for ( i = 0; i < n; i++ ) {
    b = f(i, a[i]);
    a[i] = a[i] + b;
  }
```

- Some variables are accessed by all the threads but not used to communicate between them
- Used as scratch storage within a thread
- Declare private to avoid data races

Synchronization mechanisms

Mutual exclusion

- Exclusive access to shared data
- Can be used to ensure
  - Only one thread has access to the data structure at a time
  - Accesses by multiple threads are interleaved

Event synchronization

- Signals the completion of some event from one thread to another
- Can be used to implement ordering between threads
- Mutual exclusion does not control the order in which shared data is accessed

Synchronization constructs

Mutual exclusion

- **Critical** – implements critical sections
- **Atomic** – efficient atomic update of a single memory location
- Runtime library **lock routines** – customized synchronization

Event synchronization

- **Barrier** – classical barrier synchronization
- **Taskwait / taskgroup** – waits on completion of task(s)
- **Ordered** – imposes sequential execution order
- **Flush** – enforces memory consistency

Not covered

Critical construct


```
#pragma omp critical [(name) [hint(hint-expression)]] new-line
    structured-block
```

- Provides mutual exclusion with respect to all critical sections in the program with the same (unspecified) name
- Hint may suggests a specific lock implementation
  - uncontended, contended, nonspeculative, speculative

Critical construct – example

- When encountering the construct, a thread waits until no other thread is executing inside
- No branches in/out of critical section allowed
- No fairness guaranteed
- Forward progress guaranteed
  - Eligible thread will always get access

```
  cur_max = MINUS_INFINITY;
#pragma omp parallel for
  for ( i = 0; i < n; i++ ) {
#pragma omp critical
    {
      if ( a[i] > cur_max )
        cur_max = a[i];
    }
  }
```

Preliminary test

- Previous example effectively serialized
- Frequently recommended solution
  is a preliminary test
  - `cur_max` mostly read and rarely written
- Danger – data race

- Better use reduction in this case

```
cur_max = MINUS_INFINITY;
#pragma omp parallel for
  for ( i = 0; i < n; i++ ) {
    if ( a[i] > cur_max )
#pragma omp critical
      {
        if ( a[i] > cur_max )
          cur_max = a[i];
      }
  }
```

Named critical constructs

- No critical construct can be executed concurrently with another one
  - Leads to reduced parallelism
  - Sometimes exclusive access to all critical sections together too restrictive
- Local lock instead of global lock needed
- Named critical constructs partition critical constructs into subsets that can be executed concurrently

Named critical constructs (2)


```
void critical_example(float *x, float *y) {

  int ix_next, iy_next;

  #pragma omp parallel shared(x, y) private(ix_next, iy_next)
  {
    #pragma omp critical (xaxis)
      ix_next = dequeue(x);
    work(ix_next, x);

    #pragma omp critical (yaxis)
      iy_next = dequeue(y);
    work(iy_next, y);
  }
}
```

Nesting of critical constructs

- Lexical nesting – why?
- Dynamic nesting – can easily lead to deadlock
- Make sure that all threads execute nested critical constructs in the same order

Thread 1

```
void foo() {
#pragma omp critical (A)
  {
     bar();
  }
}
```

Thread 2

```
void bar() {
#pragma omp critical (B)
  {
     foo();
  }
}
```

Atomic operations

- Many systems provide hardware instructions supporting the atomic update of a single memory location
- Instructions have exclusive access to the location during the update
  - No additional locking required – therefore better performance
- OpenMP atomic directive can give programmer access to these atomic hardware operations
- Alternative to critical construct, but only in a limited set of cases

Atomic construct


```
#pragma omp atomic [clause[[,] clause] ... ] new-line
          expression-statement or structured-block
```

Atomic clauses
read
write
update
No clause – equivalent to update

- Types of clauses – atomic, extended-atomic, or memory-order
- Permitted form of expression statement depends on clause
- Further variant with structured block = short sequence of statements to update and read a value or vice versa (not covered)

Example

Searching for an item in a list

```
  found = 0;
#pragma omp parallel for
  for ( i = 0; i < n; i++ ) {
    if ( a[i] == item )
#pragma omp atomic write
      found = 1;
  }
```

Atomic vs. critical construct

Single update

- Atomic usually never worse than critical construct
- Some implementations may choose to implement the atomic directive using a critical section

Multiple updates

- Would require multiple atomic constructs
- Critical section – one synchronization (+)
- Atomic directive – allows overlap (+)
- Usually smaller synchronization overhead for a single critical construct

Guideline

- Atomic construct to update a single or a few locations
- Critical construct to update several locations
- Do not mix critical and atomic constructs for synchronizing conflicting accesses

Runtime library lock routines


```
omp_lock_t lck;
int id;

omp_init_lock(&lck);

#pragma omp parallel shared(lck) private(id)
{
  id = omp_get_thread_num();
  omp_set_lock(&lck);
  /* only one thread at a time can execute this printf */
  printf("My thread id is %d.\n", id);
  omp_unset_lock(&lck);
}

omp_destroy_lock(&lck);
```

Runtime library lock routines (2)


```
omp_lock_t lck;
int id;
omp_init_lock(&lck);

#pragma omp parallel shared(lck) private(id)
{
  id = omp_get_thread_num();
  while (! omp_test_lock(&lck)) {
    /* do something else */
    skip(id);
  }
  /* we now have the lock and can do the work */
  work(id);
  omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Runtime library lock routines (3)

Functionality

- Lock initialization
- Lock testing
- Lock acquisition
- Lock release
- Lock destruction

More flexible

- No block structure required
- Lock variable can be determined dynamically
- Allows doing computation while waiting
- Nestable locks also provided

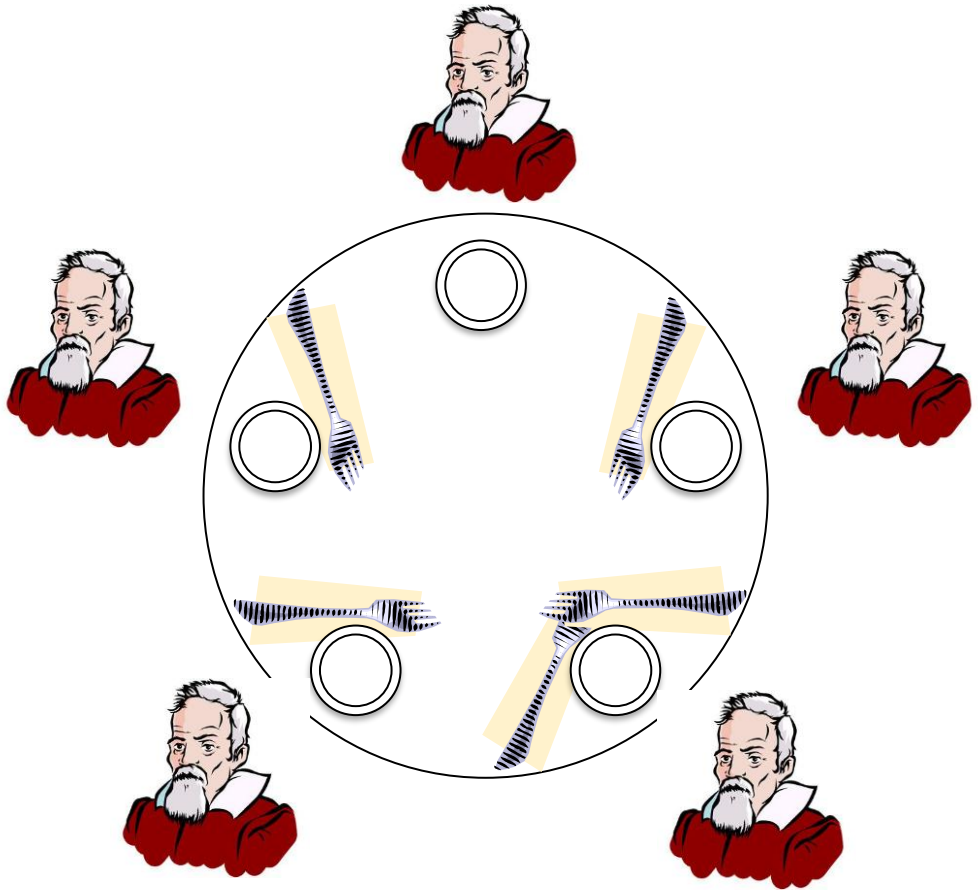
Nestable locks

```
struct exvar {
  int val;
  omp_nest_lock_t lock;
};

void incr(struct exvar* ev) {
  omp_set_nest_lock(&(ev->lock));
  ev->val++;
  omp_unset_nest_lock(&(ev->lock));
}

void times2plus1(struct exvar* ev) {
  omp_set_nest_lock(&(ev->lock));
  ev->val = ev->val * 2;
  incr(ev);
  omp_unset_nest_lock(&(ev->lock));
}
```

Dining philosophers



- Each philosopher thinks for a random period of time until he or she gets hungry
- Then attempts to grab two forks – one from the left and one from the right
- Adjacent philosophers can never eat simultaneously

Danger of deadlock

Deadlock

A deadlock is a situation where two or more threads are waiting indefinitely for an event that can be caused only by one of the waiting threads

- Can occur when multiple threads simultaneously try to set multiple locks
- Example - two threads T_1 and T_2 and two locks la and lb

  - Thread T_1 first sets la and then lb
  - Thread T_2 first sets lb and then la

- Deadlock occurs if both set the first lock before they try to set the second
- Can be avoided by setting locks in fixed order (e.g., always la then lb)

Relaxed memory model

OpenMP programs that

- Do not use non-sequentially consistent atomic directives,
- Do not rely on the accuracy of a false result from omp_test_lock and omp_test_nest_lock, and
- Correctly avoid data races

behave as though operations on shared variables were simply interleaved in an order consistent with the order in which they are performed by each thread

- The relaxed consistency model is invisible for such programs

Atomic construct – sequential consistency

Memory-order clauses
acq_rel
acquire
relaxed
release
seq_cst

Global variables

```
int x = 0;
int y = 0;
int p = 0;
int q = 0;
```

Thread 0

```
#pragma omp atomic write seq_cst
    x = 1;
#pragma omp atomic write seq_cst
    y = 1;
```

Thread 1

```
while(q != 1) {
#pragma omp atomic read seq_cst
   q = y;
}
#pragma omp atomic read seq_cst
   p = x;
assert(p == 1);
```

Barrier construct


```
#pragma omp barrier new-line
```

- Synchronizes the execution of all threads in a parallel region
- All the code before the barrier must have been completed by all the threads before any thread can execute any code past the barrier

Barrier construct – example


```
#pragma omp parallel private(index)
{
  index = generate_next_index();
  while ( index != 0 ) {
    add_index(index);
    index = generate_next_index();
  }
#pragma omp barrier
  index = get_next_index();
  while ( index != 0 ) {
    process_index(index);
    index = get_next_index();
  }
}
```

- All threads executing the parallel region must execute the barrier
- Cannot be placed inside work-distribution constructs
- Implicit barrier at the end of work-distribution constructs if there is no nowait clause

Summary

- Race conditions – result depends on relative timing of operations
- Data races – non-atomic access to single memory location
- Mutual exclusion
- Event synchronization

Synchronization constructs

- Critical
- Atomic
- Lock routines
- Barrier

Covered

- Ordered
- Flush

Not covered

- Taskwait / taskgroup

Later

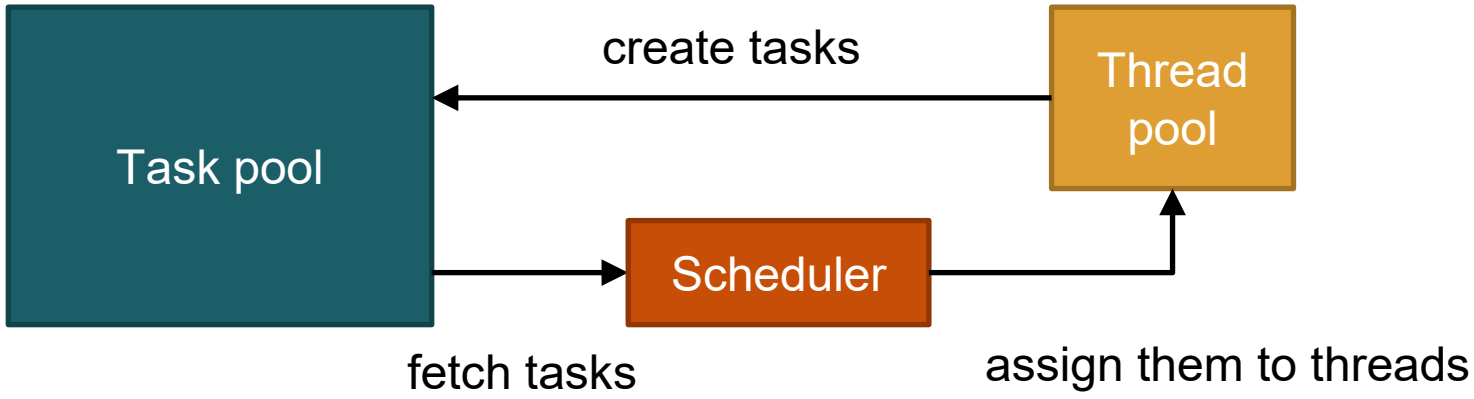
Tasking

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

Idea – separate problem decomposition from concurrency

- Decompose problem into a set of tasks and insert them into **task pool**
- Threads fetch them from there until all tasks are completed and task pool empty
- Task may create new tasks
- Advantage: good **load balance** if problem is over-decomposed



Example: Quicksort

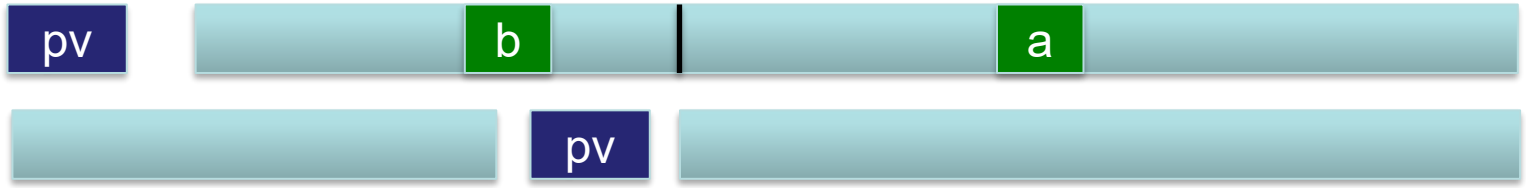
- Sorting algorithm
- Input – array of length n
- Data type with < relation
- Output – array sorted in ascending order
- Based on the principle of divide & conquer

Quicksort – step 1

- Select a pivot element pv
  - Common selection: Middle element, begin, end, or random

Quicksort – step 2

- Split array
  - Move all elements < pv to the left side
  - Move all elements >= pv to the right side

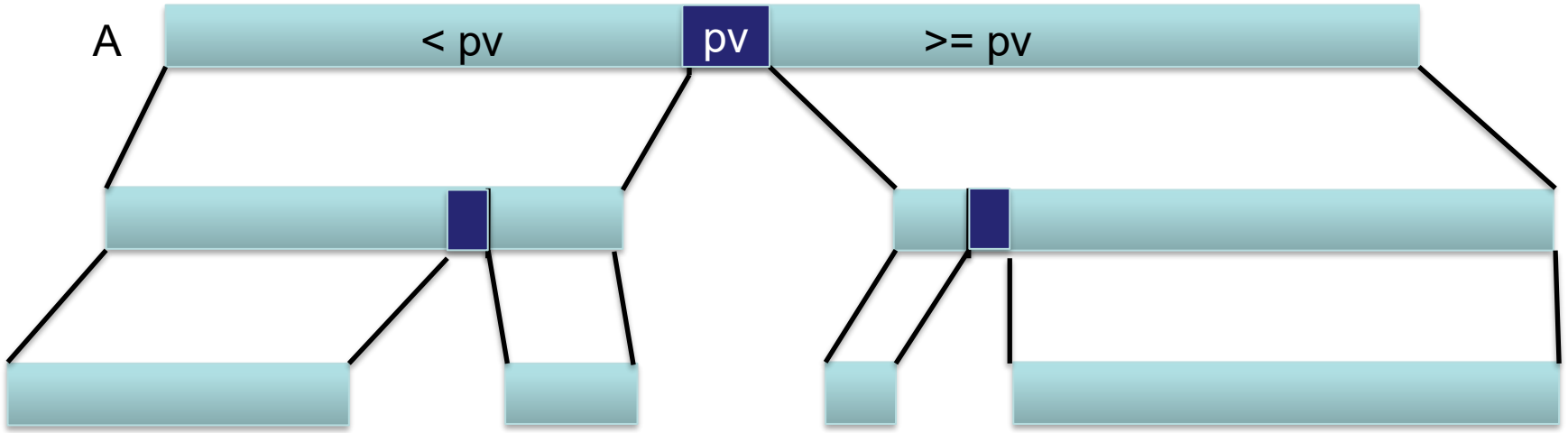


- Loop
  - Find the last element a < pv
  - Find the first element b >= pv
  - Swap a and b

Quicksort – recursion

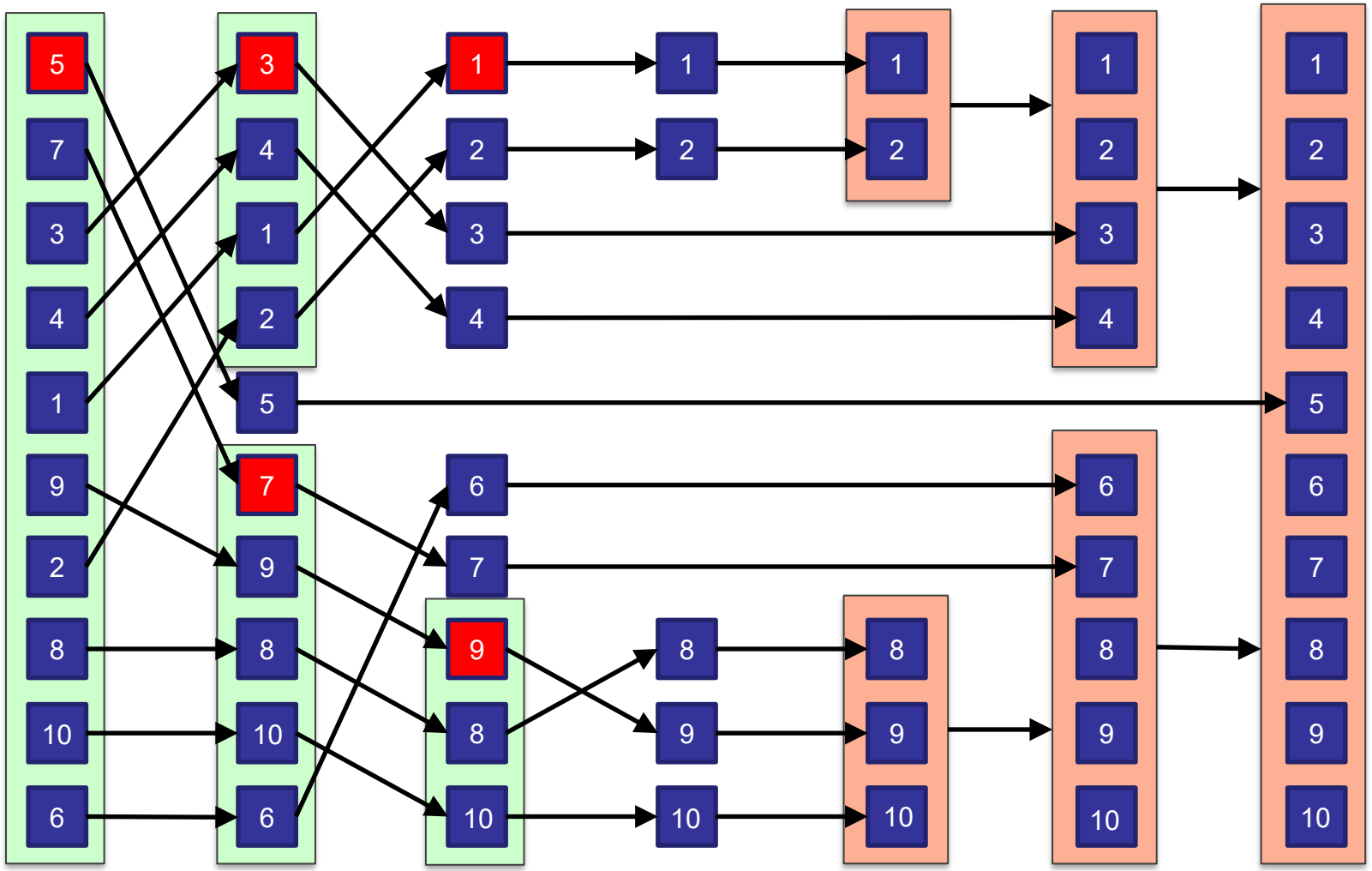
Sort both parts recursively

- Recursion stops if only one element is left
  - Arrays with one element are trivially sorted



[…]

Quicksort – example



Quicksort – serial version


```
void quicksort( int* A, int lo, int hi )
{
  if ( lo < hi ) {
    int p = partition( A, lo, hi );
    quicksort( A,  lo, p  );
    quicksort( A, p+1, hi );
  }
}

int partition(int* A, int lo, int hi)
{
  int p = A[lo], i = lo - 1, j = hi + 1;
  do {
    do i++; while ( A[i] < p );
    do j--; while ( A[j] > p );
    if ( i < j ) swap ( A, i, j );
  } while ( i < j );
  return j;
}
```

Task construct

- A task is a unit of work

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

- Task might be executed by any thread in parallel region
- Unspecified whether execution starts immediately or is deferred
- Tasks are executed in an undefined order unless the user specifies dependences

Quicksort with tasks


```
void quicksort_task( int* A, int lo, int hi )
{
  if ( lo < hi ) {
    int p = partition( A, lo, hi );

    #pragma omp task
      quicksort_task( A,  lo, p  );
    #pragma omp task
      quicksort_task( A, p+1, hi );
  }
}
```

Parallel quicksort

- Task constructs take only effect when called inside a parallel region
- Use single construct to initiate the sorting only once
  - Otherwise, each thread would initiate the sorting
- The implicit barrier at the end of the single construct ensures that all tasks are completed when the single construct is left

```
/* Assume A, lo, and hi are already initialized */
#pragma omp parallel
{
    #pragma omp single
      quicksort_task(A, lo, hi);
}
```

Task construct – clauses

Purpose	Clause
Conditional asynchronous execution	if (this task), final (for this task and subtasks)
Any thread can resume task after suspension	untied
Data sharing semantics	private, shared, firstprivate, in_reduction default
Dependences	depend
Mergeable task	mergeable
Execute tasks close to the storage location of variables	affinity
Memory allocator to be used to obtain storage for a list of variable	allocate
Hint for task execution order	priority

Explicit vs. implicit tasks

Explicit task	Implicit task
Task created by task construct	Task generated for each thread when a parallel construct is encountered during execution

Unification of these two phenomena under the umbrella “task”

makes standard more compact

Task synchronization – barrier

- A parallel construct can only complete if all explicit tasks are completed – because of the implicit barrier at the end

```
#pragma omp parallel
{
   #pragma omp task
   {
      #pragma omp barrier
   }
}
```

Deadlock

- Calling a barrier inside an explicit task leads to deadlock

Quicksort – synchronization


```
#pragma omp single nowait
   quicksort_task(A, lo, hi);
/* When using A here: A may not be sorted yet! */
```

- When using tasks, you must apply synchronization to ensure their completion before using their results
  - Don’t forget tasks inside function calls

```
#pragma omp single nowait
   quicksort_task(A, lo, hi);
#pragma omp barrier
/* You may use the sorted A now */
```

Task synchronization – taskwait

- The taskwait construct waits on all child tasks
  - But not on grand children (!)

```
#pragma omp task              // Task A
{
   #pragma omp task           // Task B
   {
      #pragma omp task        // Task C
      {
      }
   }
   #pragma omp taskwait
}
```

The taskwait waits on Task B, but not on Task C

Task synchronization – taskwait (2)


```
#pragma omp task          // Task A
{
   #pragma omp task       // Task B
   {
       #pragma omp task   // Task C
       {
       }
      #pragma omp taskwait
   }
   #pragma omp taskwait
}
```

- Use taskwait recursively to ensure waiting for all descendants
- Task A waits only on Task B – but Task B can only complete if Task C is complete

Quicksort – synchronization (2)


```
#pragma omp single nowait
   quicksort_task(A, lo, hi);
#pragma omp taskwait
/* When using A here: A may not be sorted yet! */
```

- Taskwait waits only for child tasks
  - It does not wait for recursively created tasks!
- Need to call taskwait recursively

With recursive taskwait


```
void quicksort_task( int* A, int lo, int hi )
{
  if ( lo < hi ) {
    int p = partition( A, lo, hi );

    #pragma omp task
      quicksort_task( A,  lo, p  );
    #pragma omp task
      quicksort_task( A, p+1, hi );
    #pragma omp taskwait
  }
}
```

Taskgroup

- Recursive task creation is a common pattern
  - Need to wait for the completion of all recursively created tasks
- taskwait does only wait for direct children
- barrier waits for all tasks – not appropriate for explicit tasks

```
#pragma omp taskgroup new-line
{
   structured-block
}
```

- At the end of the structured block taskgroup waits for all tasks created inside the structured block **and their descendants**

Tied vs. untied tasks

Tied task

- Default mode

```
#pragma omp task
```

- Can be suspended at scheduling points
- Task will be resumed by thread that suspended it
- The implicit task is always tied

Untied task

- Specified via untied clause

```
#pragma omp task untied
```

- Can be suspended at scheduling points
- Suspended task may be resumed by any thread in the team

Task scheduling

- Whenever a thread reaches a task scheduling point, it may switch between tasks
  - Begin executing a new task or resume execution of suspended task

- Implied task scheduling points
  - Point immediately following task generation
  - After last instruction of task region
  - Implicit and explicit barriers
  - taskwait, taskgroup constructs

- Explicit scheduling points
  - taskyield construct

Taskyield construct


```
#pragma omp taskyield
```

- Inserts an additional scheduling point
  - Allows the runtime system to suspend current task at this point
- Does not wait for any task

Data sharing in tasks

The data environment binds to the task, not the thread encountering the task

Semantics of variable on task construct	References to it inside the construct refer to…
shared	storage area of the original variable at the point the construct was encountered
private	a new uninitialized storage that is created when the task is executed
firstprivate	new storage with the same variable name that is created and initialized with the value of the variable when the tasks is encountered

Data sharing in tasks (2)

- Default for variables inherited from task creation context
  - shared if shared among all implicit tasks in the enclosing context
  - Otherwise firstprivate
- Variables created inside a task are private

Data sharing example 1


```
#pragma omp parallel
{
   int a, b, c;
#pragma omp task shared(a) firstprivate(b) // Task A
   {
       int d;
   }
}
```

Each implicit task has its own `a, b, c` (no sharing between implicit tasks) and creates its own Task A.
Between an implicit task and its Task A:

- `a` is shared (clause)
- `b` is firstprivate (clause)
- `c` is firstprivate (default)
- `d` is private to Task A (local variable)

No sharing between the explicit tasks

Data sharing example 2


```
#pragma omp parallel
{
   int a, b, c;
#pragma omp task shared(a,c)                    // Task B
   {
       int a, d;
   }
}
```

Each implicit task has its own `a, b, c` (no sharing between implicit tasks) and creates its own Task B.
Between an implicit task and its Task B:

- Task B has `a` private (local variable, shadows other variable)
- `b` is firstprivate (default)
- `c` is shared (clause)
- Task B has `d` private (local variable)

No sharing between explicit tasks

Data sharing example 3

TECHNISCHE UNIVERSITÄT DARMSTADT

Parallel Programming

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

All tasks (implicit and explicit) share a (both sharing clauses).
Each implicit task has its own b and c (sharing clause/local variable).

Between an implicit task and its Task C:

- b is shared (clause)
- c is firstprivate (default)

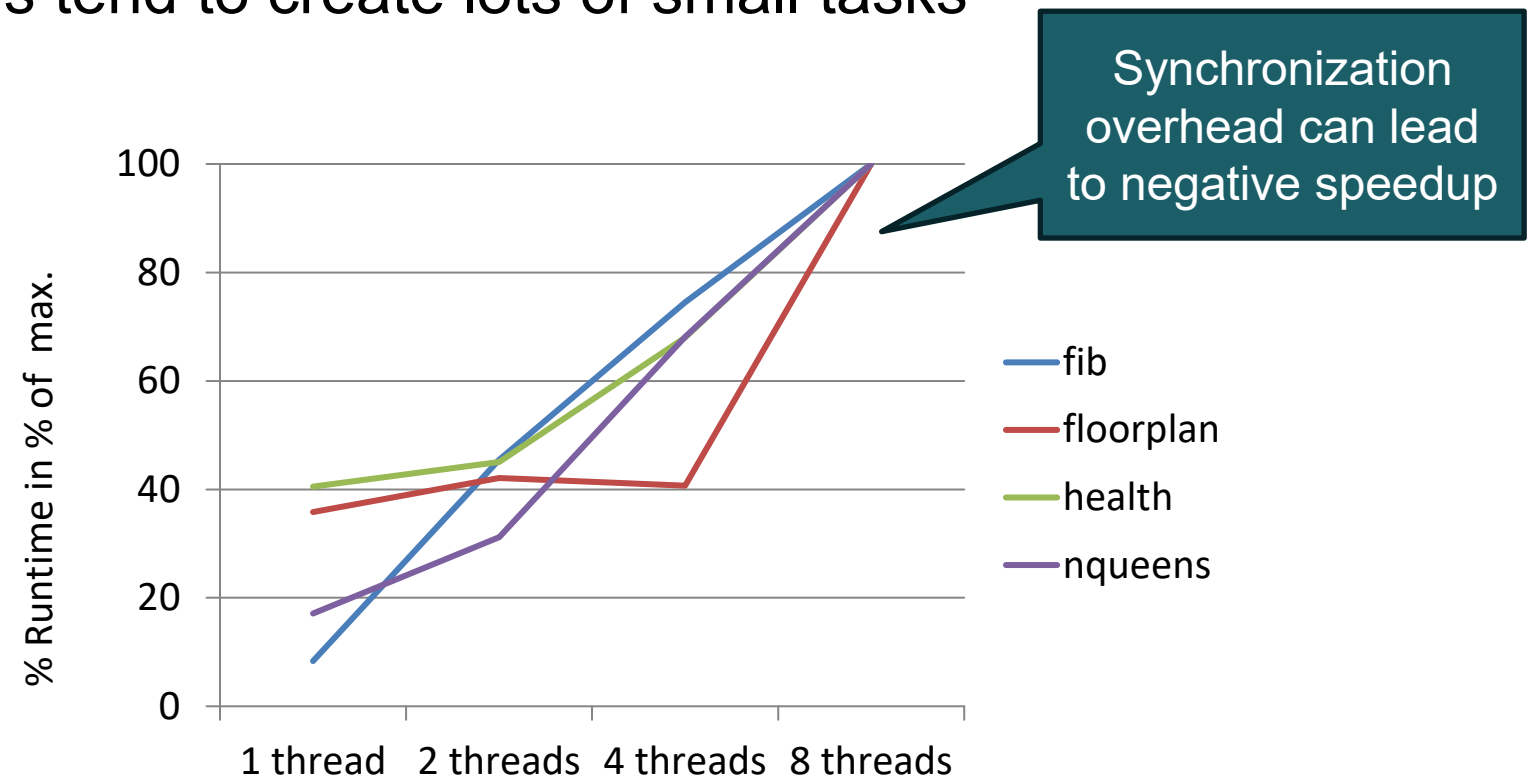
No sharing of variables among explicit tasks except for a

Task overhead

- Task creation and management incurs some overhead
  - Allocate memory for task data structures
  - Insert tasks into task pool
  - Synchronize access to the task pool during insertion and removal
- Much less than thread creation though

If tasks are too small…

- …task management may become a performance problem
- Especially recursive algorithms tend to create lots of small tasks on deeper recursion levels



if clause

- To avoid extensive overhead, create new task only if work exceeds a certain amount

```
#pragma omp task if(condition)
{
   structured-block
}
```

- Task is only created if condition evaluates to true
- If the condition evaluates to false, the task body is executed inside the current task. No new task is created
- Only one if clause per task allowed

Quicksort with if clause


```
void quicksort_task( int* A, int lo, int hi )
{
  if ( lo < hi ) {
    int p = partition( A, lo, hi );

    #pragma omp task if ( p - lo + 1 > MIN_VAL )
      quicksort_task( A,  lo, p  );
    #pragma omp task if ( hi - p > MIN_VAL )
      quicksort_task( A, p+1, hi );
    #pragma omp taskwait
  }
}
```

final clause


```
#pragma omp task final(condition)
{
   task body
}
```

- If condition evaluates to true, the new task becomes final
  - All task constructs appearing inside the task are ignored. Body is executed as part of the current task
- Purpose: optimization of if clause
  - If clause reevaluates the condition on every recursion level
  - Final clause avoids evaluation on subsequent levels
- Difference between if and final clause
  - Final clause creates **no** further tasks (beyond this task) if condition is true – but if clause may

Quicksort with final clause


```
void quicksort_task( int* A, int lo, int hi )
{
  if ( lo < hi ) {
    int p = partition( A, lo, hi );

    #pragma omp task final ( p - lo + 1 <= MIN_VAL )
      quicksort_task( A,  lo, p  );
    #pragma omp task final ( hi - p <= MIN_VAL )
      quicksort_task( A, p+1, hi );
    #pragma omp taskwait
  }
}
```

Task dependences

- Often, the output of one tasks is needed as input for another task
- Current synchronization mechanisms (barriers and taskwait) not powerful enough
  - Lead to (short) phases of task parallel execution followed by a synchronization phase
  - Increases synchronization overhead
  - Reduces the amount of exploitable concurrency

  } Limits use of tasks

- Solution – specify explicit dependences between tasks
  - RAW, WAR, WAW

Task dependences (2)

- Define which variables are read by a task
- Define which variables are written by a task
- If task A accesses a variable that was accessed by a formerly created sibling task B and one of the accesses is a write, then A depends on B
  - This also applies to two writes: The syntactically latter one depends on the first one !
- Runtime enforces dependences during execution

Typical, implementation dependent behavior, but not strictly required.

Task dependences (3)

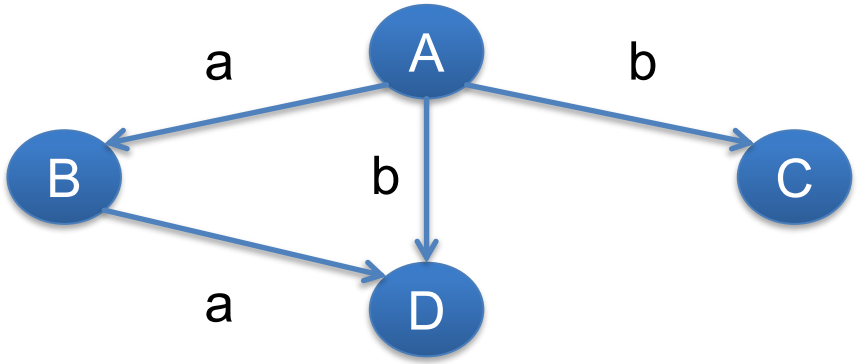

```
#pragma omp task depend(in|out|inout: variable-list)
```

Dependence type	Meaning
in	Variables read by the task
out	Variables written by the task
inout	Variables read and written by the task

- Establishes a synchronization of memory accesses performed by a dependent task with respect to accesses performed by the predecessor tasks
- However, the programmer must properly synchronize with respect to other concurrent accesses that occur outside of those tasks

Task dependences – example

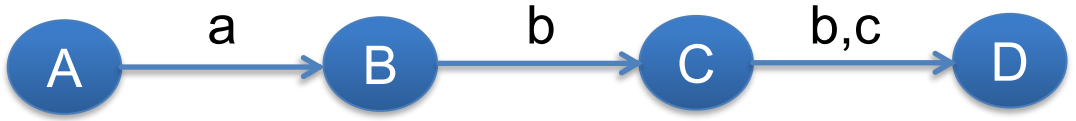
Execution order




```
int a, b;
#pragma omp task depend(out:a,b) shared(a,b) // Task A
{}
#pragma omp task depend(inout:a) shared(a)   // Task B
{}
#pragma omp task depend(in:b) shared(b)      // Task C
{}
#pragma omp task depend(in:a,b) shared(a,b)  // Task D
{}
```

WAW Task dependences - ordered

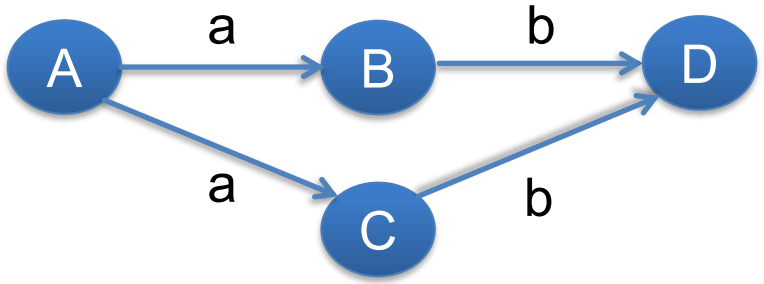
Execution order




```
int a, b, c;
#pragma omp task depend(out:a) shared(a,b) // Task A
{}
#pragma omp task depend(in:a) depend(out:b) shared(a,b)   // Task B
{}
#pragma omp task depend(inout:b) depend(out:c) shared(b,c)  // Task C
{}
#pragma omp task depend(in:b,c) shared(b,c)      // Task D
```

WAW Task dependences - unordered

Execution order



Typically: A,B,C;D

```
int a, b;
#pragma omp task depend(out:a) shared(a) // Task A
{}
#pragma omp task depend(in:a) depend(out:b) shared(a,b)   // Task B
{}
#pragma omp task depend(in:a) depend(out:b) shared(a,b)  // Task C
{}
#pragma omp task depend(in:b) shared(b)  // Task D
```

Dependences exist only between siblings

- Siblings are tasks that are created by the same parent task

```
#pragma omp task
{
  #pragma omp task depend(out:a) // Task A
    {}
}
#pragma omp task depend(in:a)    // Task B
{}
```

- No dependence between A and B
  - A is not a sibling of B

Summary tasking

- Tasking separates problem decomposition from concurrency
- Challenge to find the right task granularity
  - Good load balance if #tasks >> #threads
  - But significant overhead if tasks are too fine grained
- Task dependences enforce order between tasks
  - May limit exploitable parallelism