TECHNISCHE UNIVERSITÄT DARMSTADT

System and Parallel Programming

Prof. Dr. Felix Wolf

SHARED MEMORY PROGRAMMING WITH OPENMP

Outline

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

Literature

- 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
  Version 4.5, November 2015 + Examples
  http://www.openmp.org/specifications/

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 creation of portable parallel programs
- Works with C/C++ and Fortran
- Current specification 4.5 (Nov 2015)
- http://www.openmp.org

Here, we cover only C/C++

A simple example

```c
#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

```
> 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 of serial code
- Small increase in code size
- Code readability maintained as directives within serial code
- Possible to write code that compiles and runs with a normal compiler on a 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

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
2013	V4.0 • Extensions including SIMD parallelism and execution on accelerators • Examples moved to a separate document
2015	V4.5

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++)

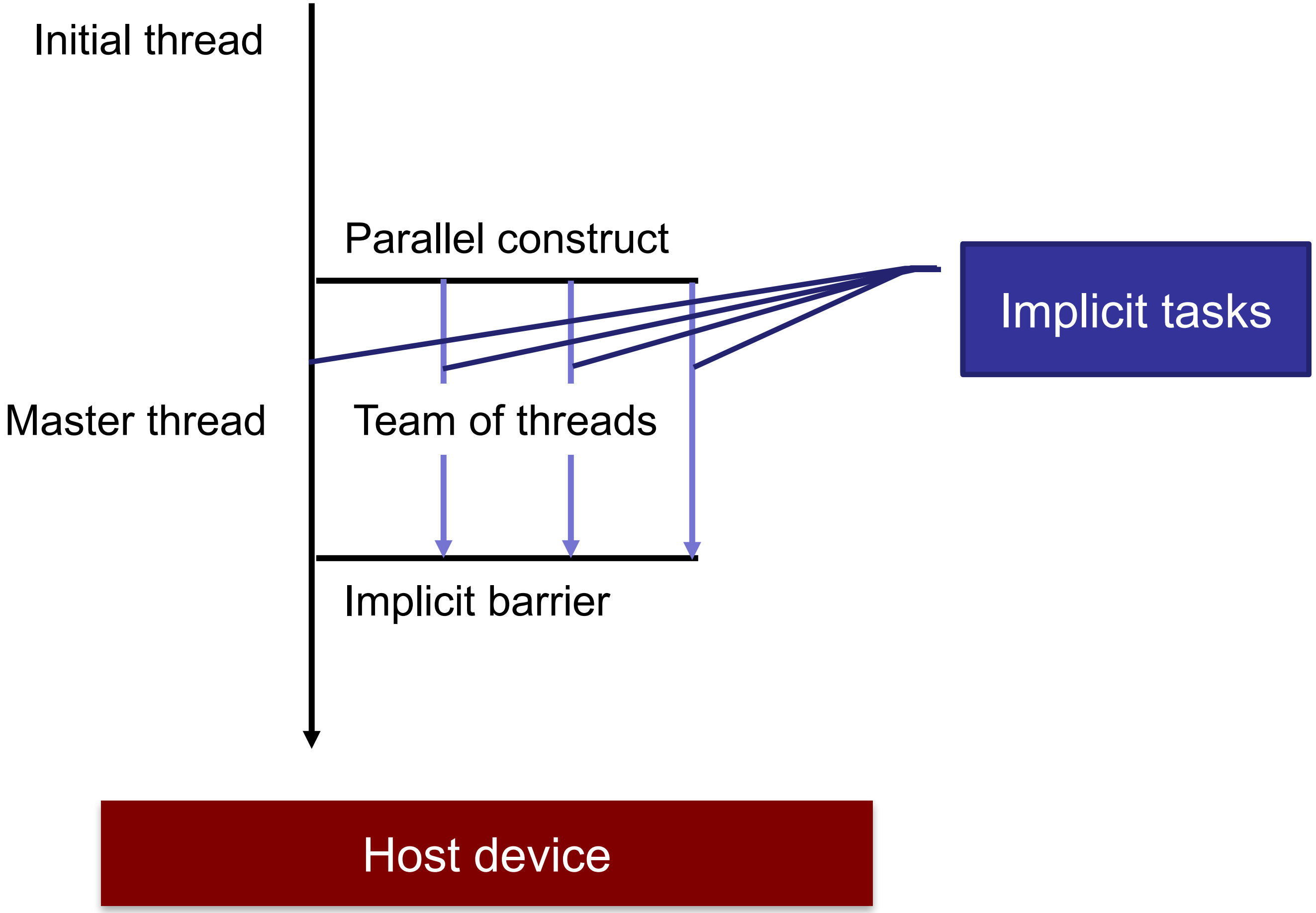
```
#pragma omp …
```

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

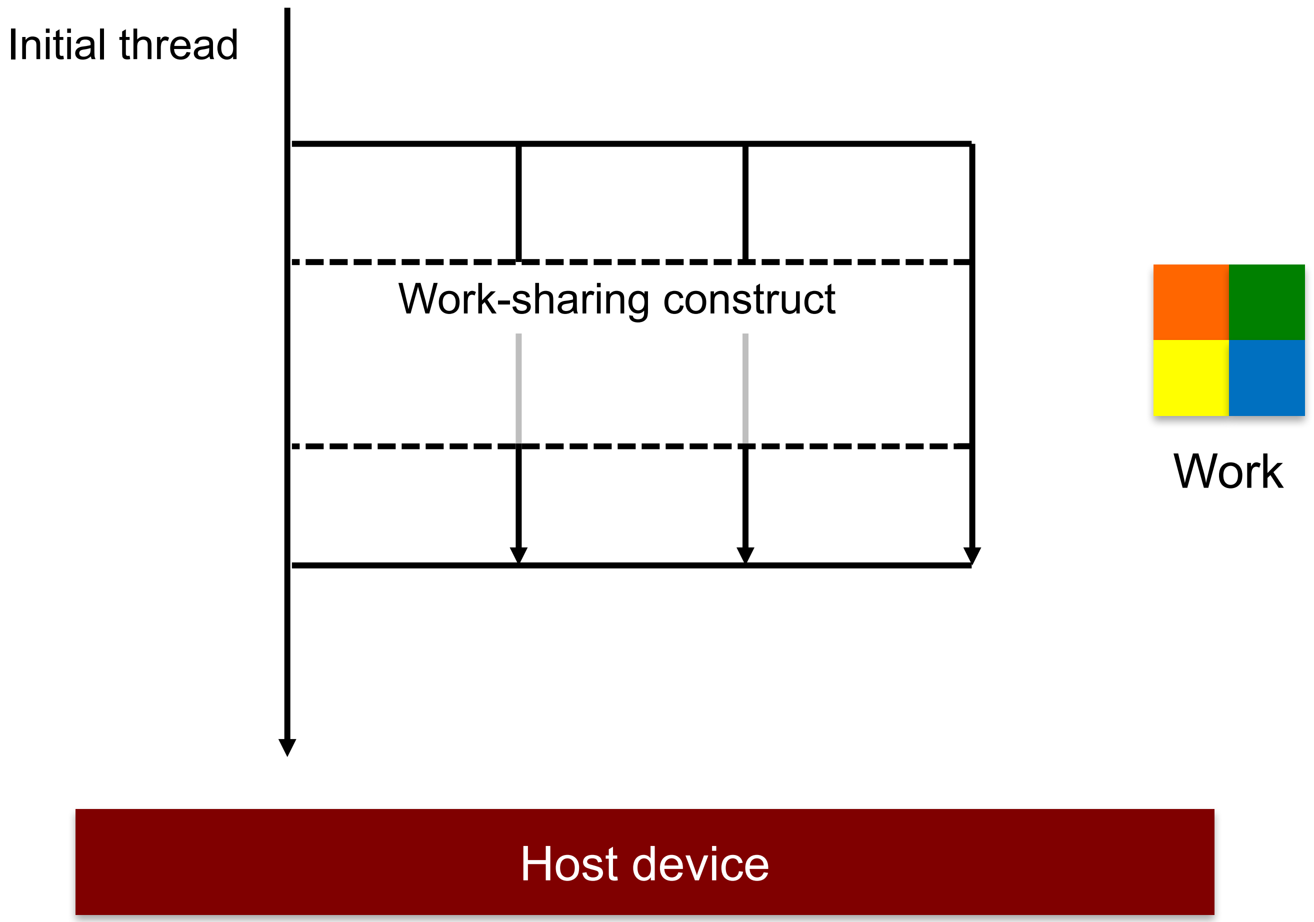
```
#ifdef _OPENMP
   iam = omp_get_thread_num();
#endif
```

Fork-join execution model

TECHNISCHE UNIVERSITÄT DARMSTADT

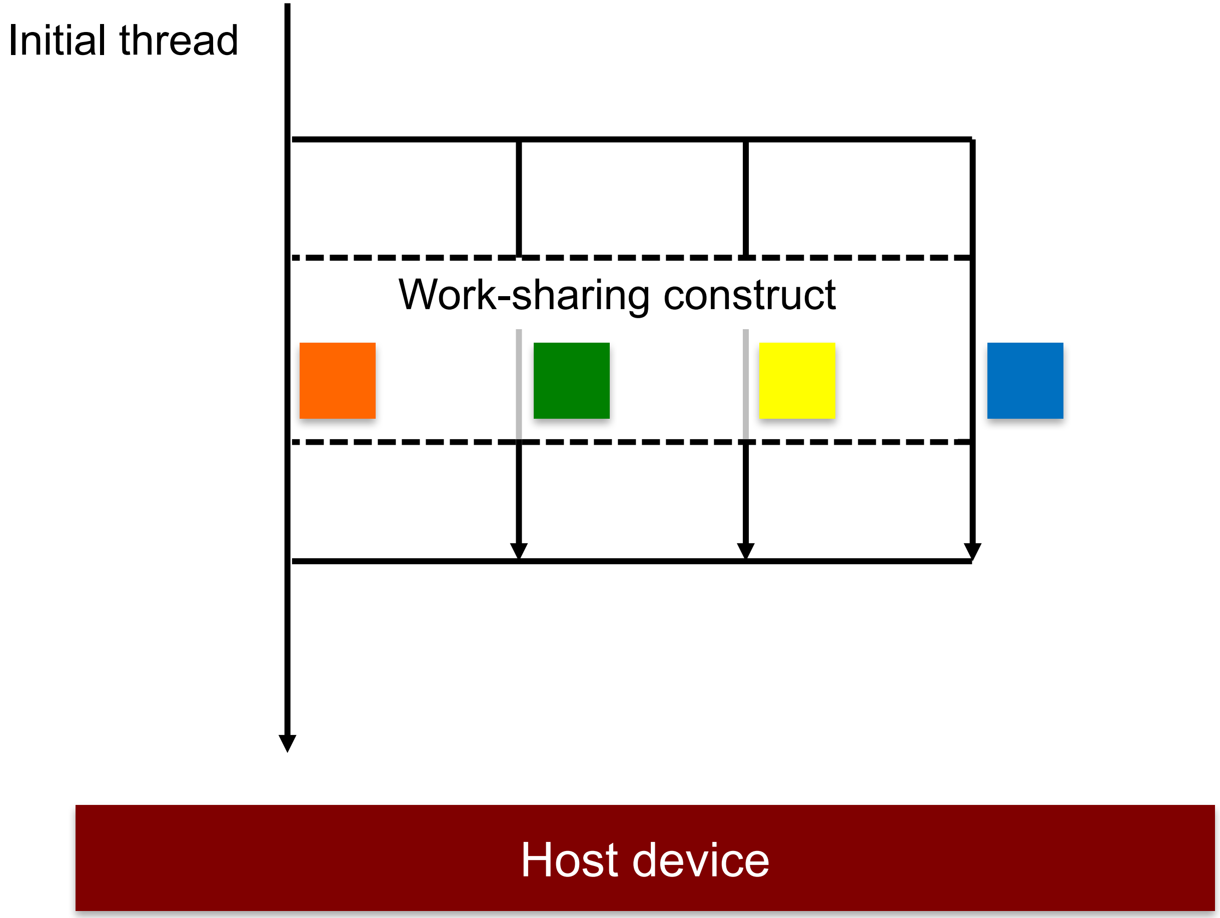


Work sharing

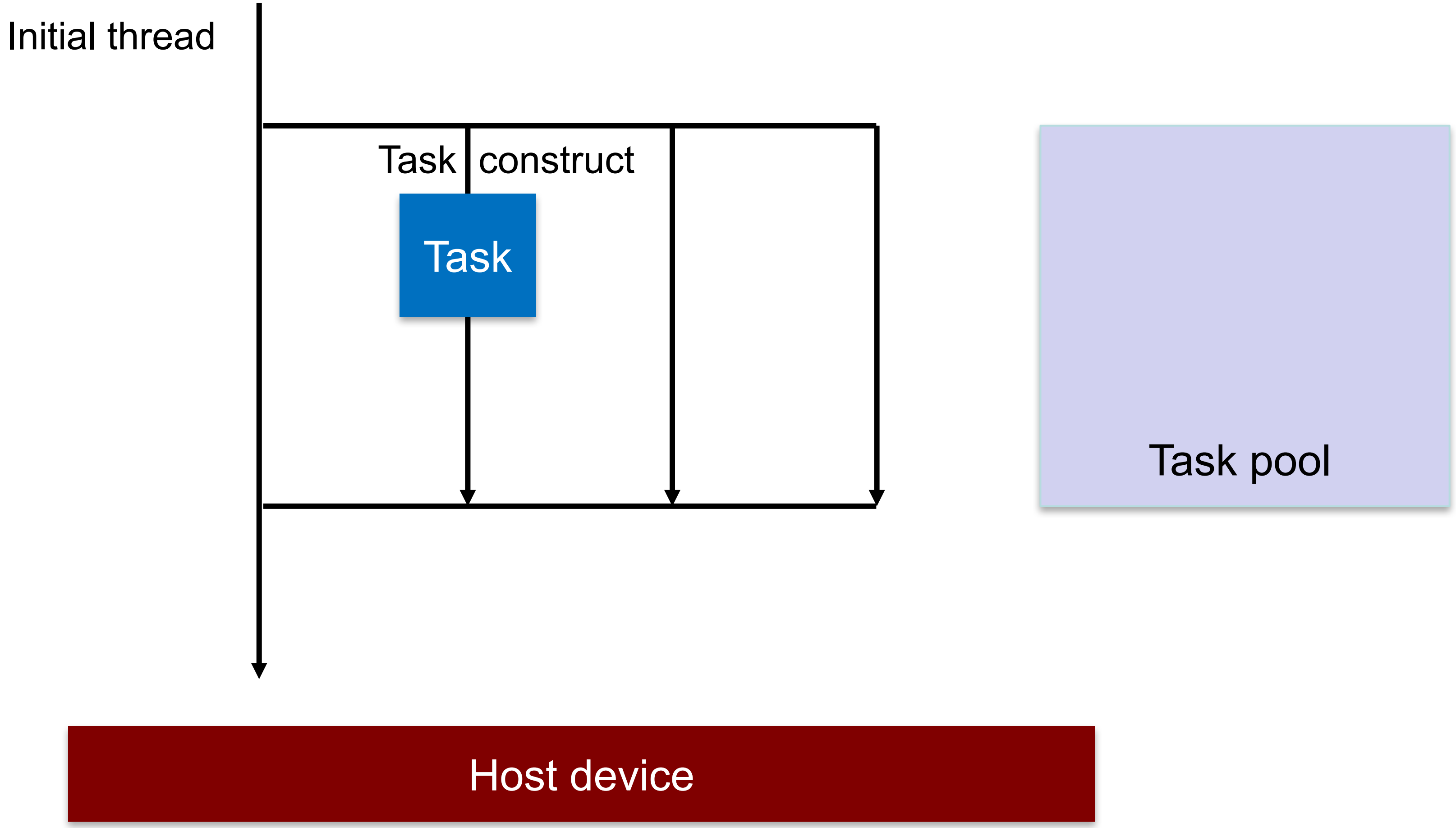


Work sharing

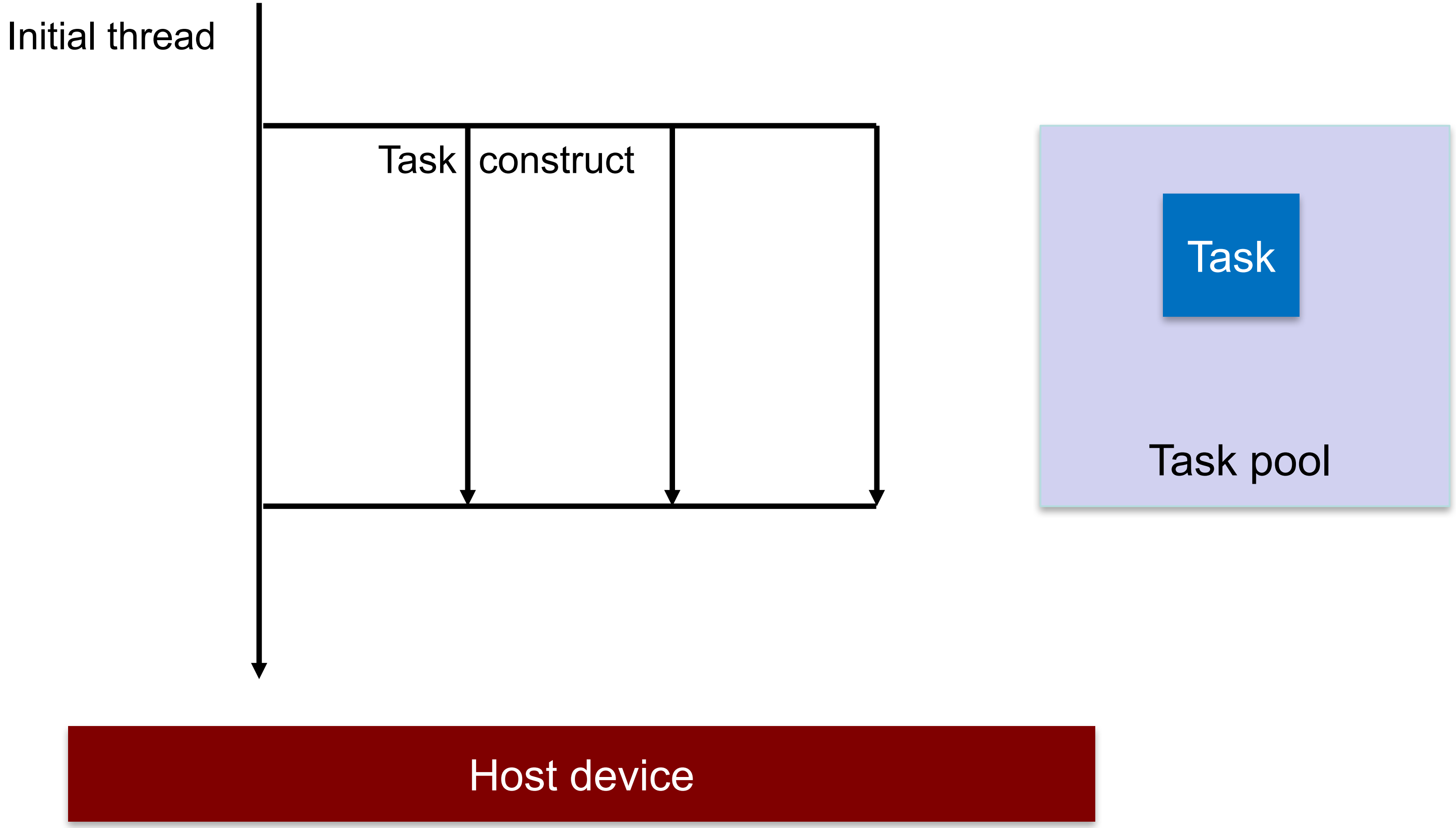
TECHNISCHE UNIVERSITÄT DARMSTADT



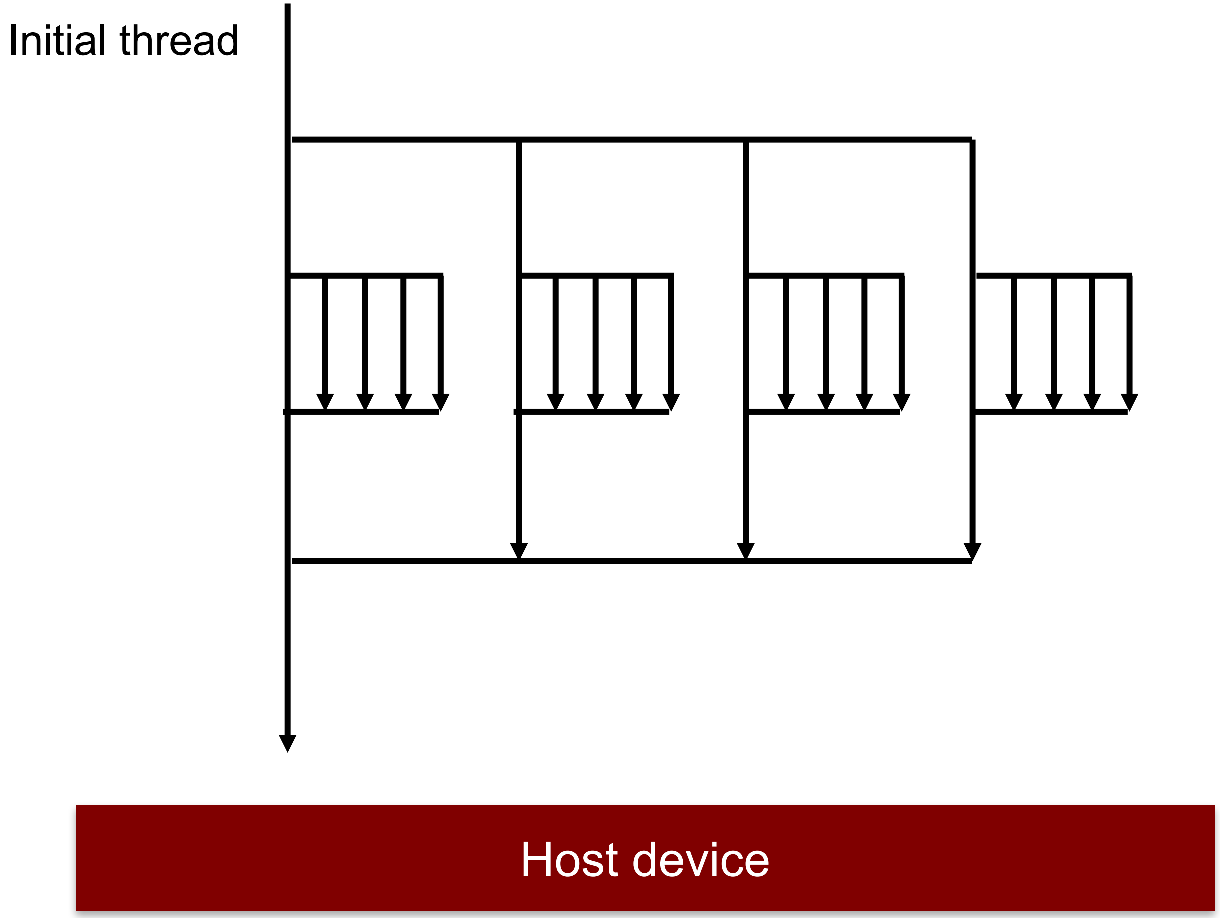
Tasking



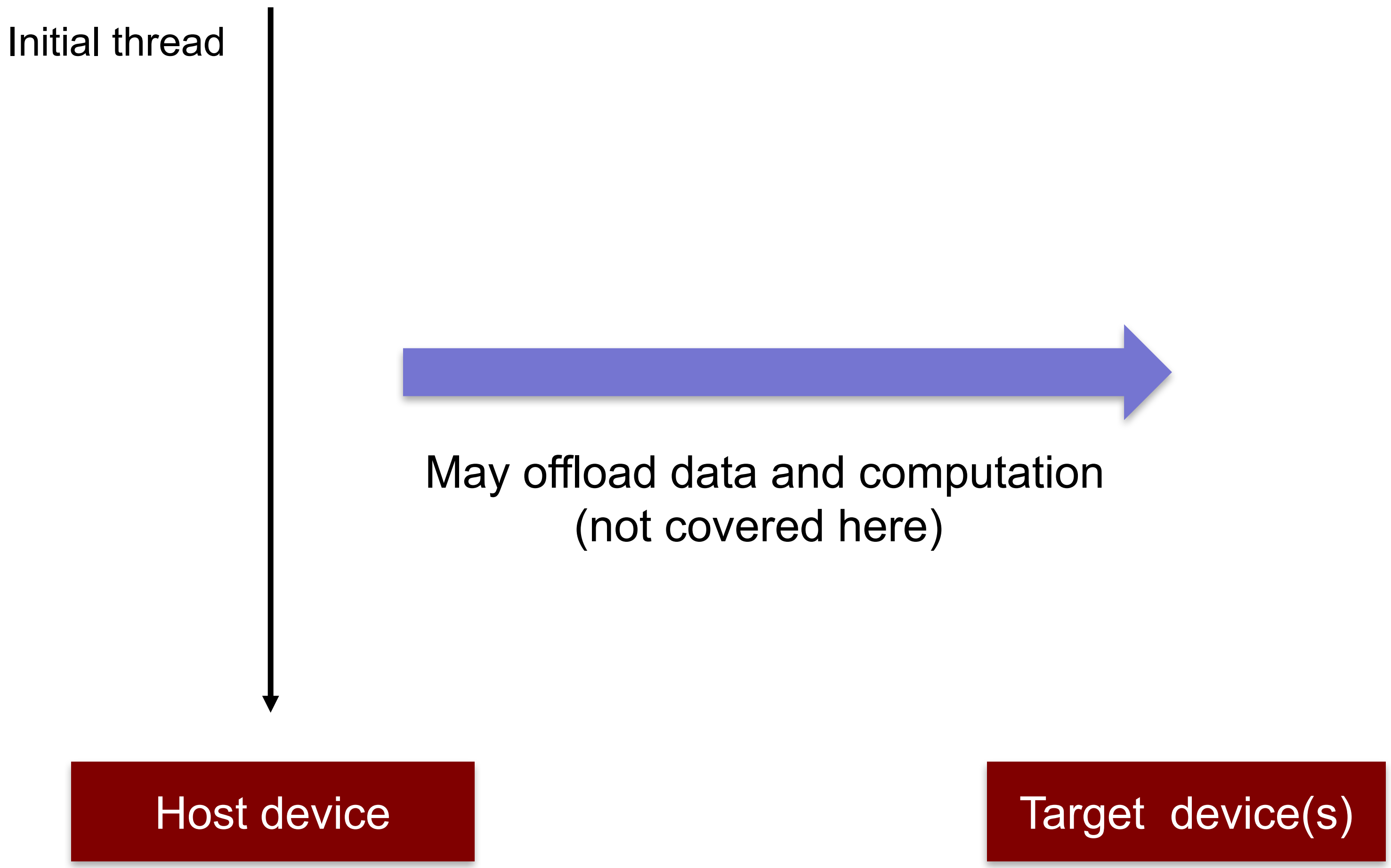
Tasking



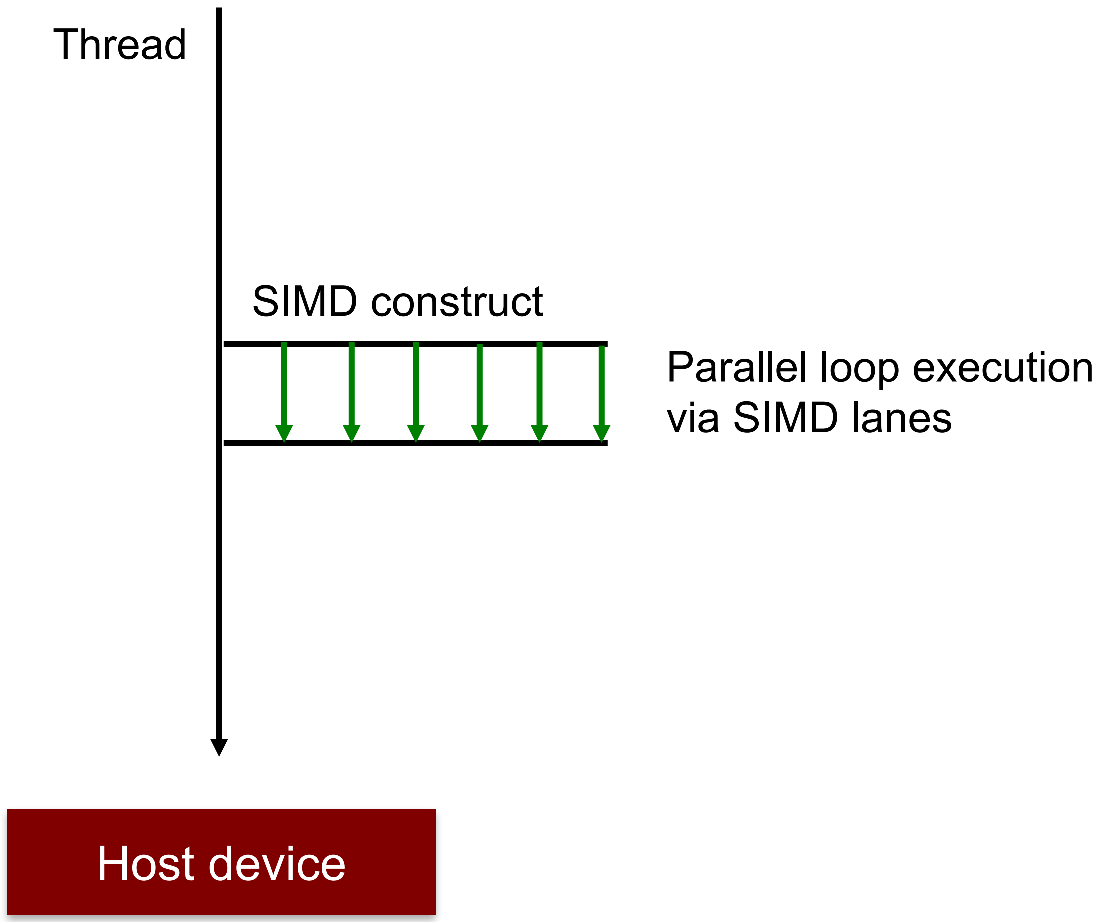
Nested parallelism



Offloading computations



SIMD parallelism



OpenMP feature summary

Covered in course

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

Not covered in course

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

Parallel control structures

- Fork new threads; pass execution control to sets of threads
- OpenMP has only a small set of control structures
- Three main functions
  - Creating a team of threads executing concurrently (parallel construct)
  - Dividing work among an existing set of parallel threads
    - Work-sharing constructs (e.g., loop-level parallelism)
  - Generating a task (task construct)
  - Offloading computation to a device (target construct)
- Note: SIMD transformation (simd construct) handled by compiler

Data environment

- Each thread has data environment
  - Global variables
  - Automatic variables within subroutines (allocated on the stack)
  - Dynamically allocated variables (allocated on the heap)
- Data environment of the initial thread exists for entire program
- Data environment of a worker thread
  - Own stack
  - All other variables are either shared or private
  - Can be specified on a per-variable basis using data scope clauses

Sharing semantics

- Shared semantics
  - Threads that access this variable access same storage location
  - Communication via reading from or writing to shared variables
- Private semantics
  - Multiple storage locations
  - One in each thread’s execution context
  - Inaccessible to other threads
- Reduction variables
  - In between shared and private
  - Subject to an arithmetic operation at the end of a parallel construct

Synchronization – two forms

Mutual exclusion

- Coordinates access to shared variables across multiple threads
- Ensures integrity in view of concurrent access
- Example: critical section – protects code section against concurrent access

Event synchronization

- Signals occurrence of event across multiple threads
- Example: barrier – point where each threads waits for all other threads to arrive

Multiply add or saxpy (single-precision a*x plus y)

```
void saxpy(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 saxpy

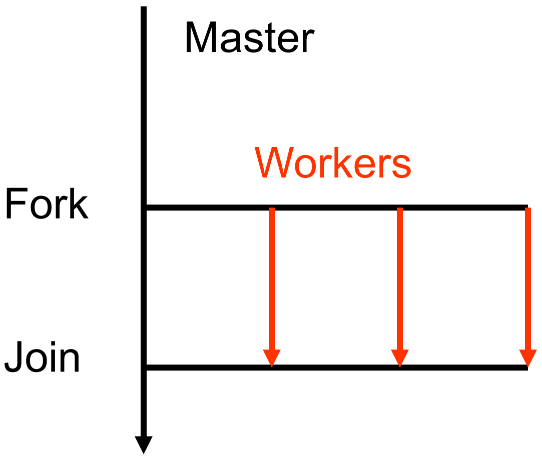
```
void saxpy(double z[], double a, double x[], double y,
           int n)
{
  /* for simplicity, we have y as a scalar variable */
  int i;

#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

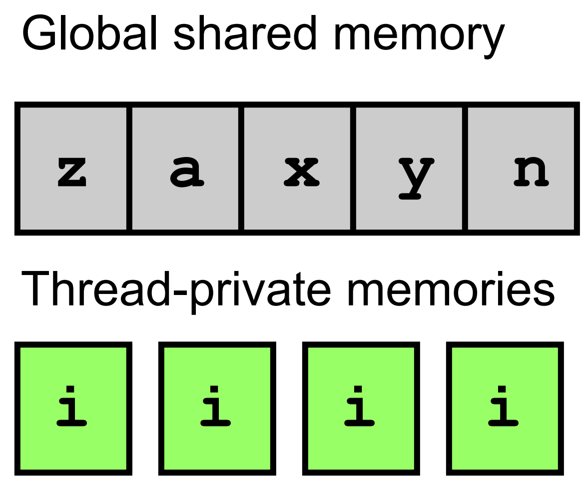
```
void saxpy(…)
{
  int i;
#pragma omp parallel for
  for ( i = 0; i < n; i++ )
    z[i] = a * x[i] + y;
}
```



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

Communication and data sharing

```
void saxpy(…)
{
  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 the loop index
- The loop index of the parallel for construct is private by default
  - Each thread has a private copy of the loop index

Synchronization

- Access to shared variables
  - All threads modify only distinct elements of the array
- Update of array must be complete when master resumes execution
  - Ensured by implicit barrier at the end of the parallel for construct

- Remarks
  - Example showed loop-level parallelism
  - Non-iterative code requires different kinds of parallelism

Construct vs. region

Construct

- Lexical or static extent
- Code lexically enclosed

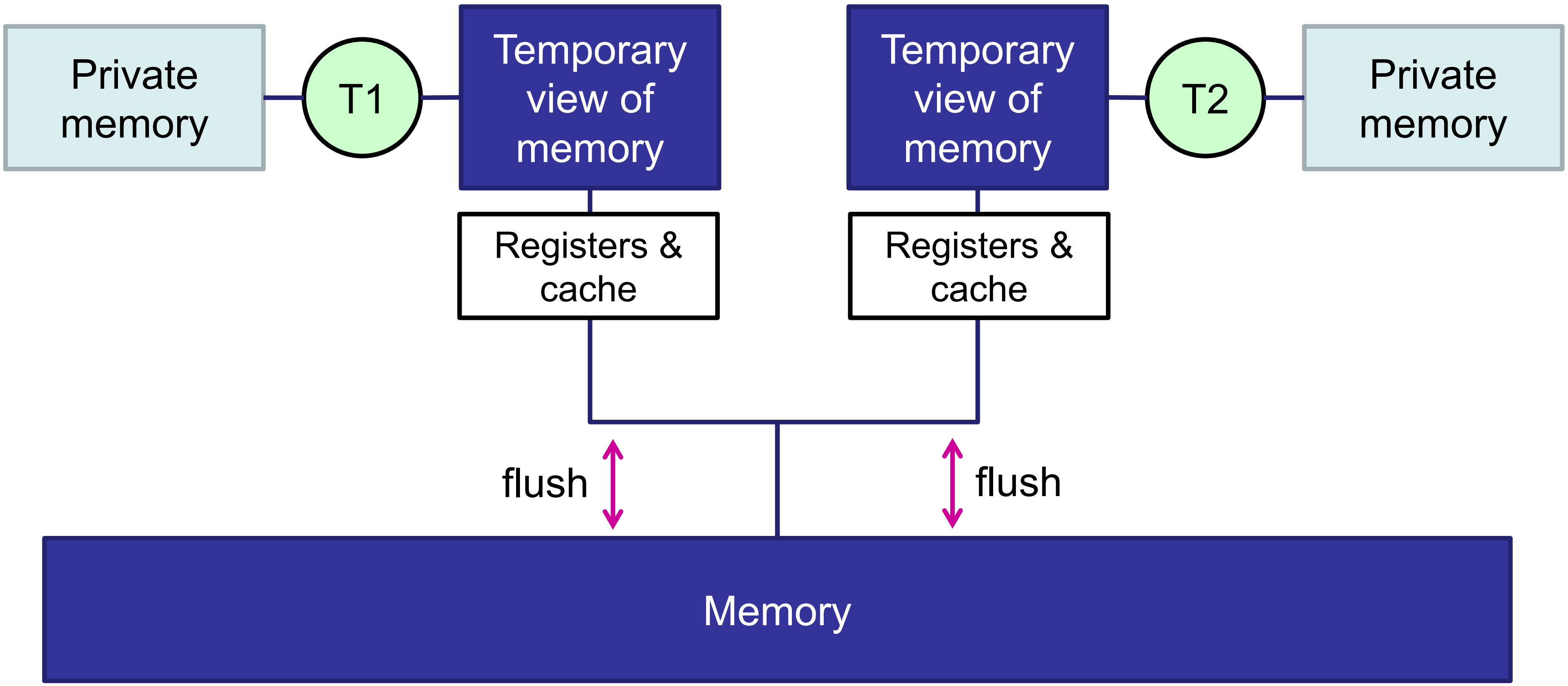
Region

- Dynamic extent
- Lexcial 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();
  }
}
```

Memory model



Memory model (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Shared memory with relaxed consistency

- All threads have access to memory
  - Each thread can have its own temporary view of memory
  - Any kind of intervening structure between thread and memory
    - Example: register, cache
  - Allows thread to avoid going to memory for every reference
  - Use flush operation to enforce consistency between temporary view and memory
- Each thread has also access to threadprivate memory
  - Must not be accessed by other threads

Memory model (3)

- A single access to a variable may be implemented with multiple load or store instructions
  - 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
- Store information such as
  - Number of threads to use for parallel execution
  - Scheduling strategy for parallel loops
- Initialized by the implementation itself
- User assigns values through
  - OpenMP environment variables
  - Calls to OpenMP API routines
- Program can retrieve values of ICVs only through API calls

Summary introduction

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

Loop-level parallelism

TECHNISCHE UNIVERSITÄT DARMSTADT

- Executing the iterations of a loop concurrently
- Fine-grained – units of work distributed across threads are small
- Incremental parallelization by adding directives
  - Small, localized changes to the source code
- Correctness
  - Parallel version should yield same results as serial version
- Loop-level parallelism in OpenMP
  - (parallel) loop construct

Loop-level parallelism – outline

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

Parallel loop construct

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

Can be loop nest

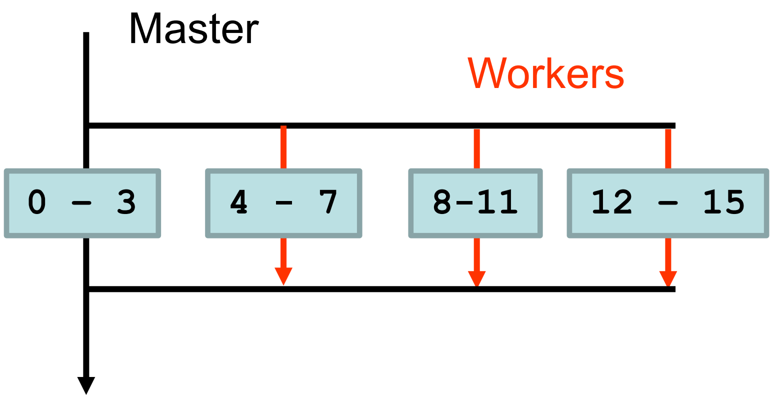
Loops immediately following the directive are parallelized

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++)
    […]
```



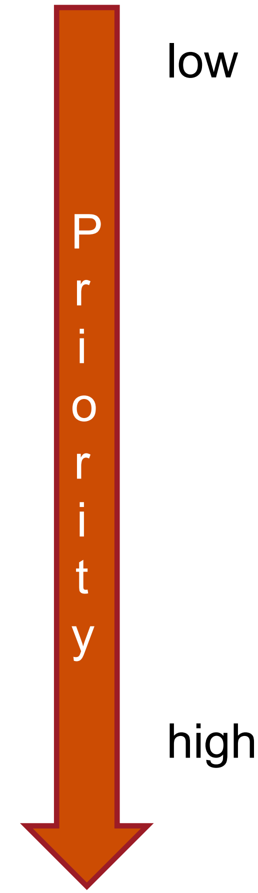
Runtime behavior (2)

- Outside the loop a single master thread executes serially
- When reaching the loop, the master creates a team with zero or more worker threads
- The loop is concurrently executed by all threads
- Each iteration is executed only once
- Each thread may execute more than one iteration
- 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`
  - If set before program start, the application will create teams of the given size
- `omp_set_num_threads(n)`
  - Adjustment of the team size at runtime
  - Affects only subsequent parallel constructs
- `numthreads` clause
  - Controls the number of threads for a particular parallel construct

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



Canonical loop structure

- Restrictions on loops to simplify compiler-based parallelization
- 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

Canonical loop structure (2)

for (*init-expr*; *test-expr*; *incr-expr*) *structured-block*			
init-expr	*var = lb* *integer-type var = lb* *random-access-iterator-type var = lb* *pointer-type var = lb*		
test-expr	*var relational-op b* *b relational-op var*		
incr-expr	*++var* *var++* *- - var*	*var - -* *var += incr* *var - = incr*	*var = var + incr* *var = incr + var* *var = var – incr*

Canonical loop structure (3)

for (*init-expr*; *test-expr*; *incr-expr*) *structured-block*	
var	• A variable of a signed or unsigned integer type • For C++, a variable of a random access iterator type • For C, a variable of a pointer type • If this variable would otherwise be shared, it is implicitly made private in the loop construct • Must not be modified during the execution of the *for-loop* other than in *incr-expr* • Unless the variable is specified **lastprivate** or **linear** on the loop construct, its value after the loop is unspecified
relational-op	< <= > >=
lb and *b*	Loop invariant expressions of a type compatible with the type of *var*
incr	A loop invariant integer expression

Loop nest

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

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

Clauses of the parallel loop construct

Clause	Purpose
if(*[*parallel :*] scalar-expression*)	Conditional parallelization
num_threads(*integer-expression*)	Number of threads
copyin(*list*)	Copying between thread-private variables
proc_bind(master | close | spread)	Thread affinity
schedule*[modifier[, modifier]*:*]kind*[,*chunk_size*])	Controls distribution of work across the team of threads
collapse(n)	Collapses nested loop into one larger iteration space to be parallelized
ordered*[* (n) *]*	Execution order of ordered constructs the same as in serial execution
Data sharing attribute clauses (next slide)	

Data sharing attribute clauses

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Clause	Purpose
private(*list*)	Specifies private semantics for a variable
shared(*list*)	Specifies shared semantics for a variable
default(shared | none)	Changes default semantics of variables (only parallel loop construct)
firstprivate(*list*)	Initialization of private variables
lastprivate(*list*)	Finalization of private variables
linear(*list[: linear-step]*)	Declares variable • to be private to a SIMD lane • to have a linear relationship with respect to the iteration space of a loop
reduction(op*: list*)	Specifies a reduction operation on a variable

General rules

- Most data sharing attribute clauses consist of keyword plus comma separated list of variables in parantheses
  - Example: **`shared(x,y,z)`**
  - Reduction clause also specifies operation, e.g., **`reduction(+: x,y)`**
- Clauses apply only to the the construct in which they appear
  - Not to the wider region
- Variable must be visible in construct

General rules (2)

- Clauses 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++
- A variable can only appear in one clause
  - Exception: can appear in both firstprivate and lastprivate clause

Shared clause

- 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 clause

- Each thread allocates a private copy of the variable from storage within the thread's private execution context
- References within (the lexical extent of) the parallel construct read or write the private copy
- Value undefined upon entry of the parallel construct
  - Exceptions: loop index variable & C++ class objects
- Value undefined upon exit of the parallel construct
- Size of the variable must be known to the compiler
  - No incomplete or reference types in C/C++

Default sharing semantics = shared

TECHNISCHE UNIVERSITÄT DARMSTADT

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

```c
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);
  }
}
```

```c
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;
}
```

Shared	Private
a, n, j, m, *x, c, cnt	i, k, x, y, z, ii, *y

Use of j and cnt is not safe

Default sharing semantics

- Can be specified using default clause
  - C/C++: **`default(shared | none)`**
- Default shared
  - Is already the default
- Default none
  - All variables must be explicitly specified

Reduction operations

```
  sum = 0;

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

- Syntax: **`reduction(`***op* **:** *var-list***`)`**
- Applications
  - Compute sum of array
  - Find the largest element
- Operator should be commutative-associative
- Reduction variables are initialized to the identity element
- Reduction on array element via scalar temporary

Predefined reduction operators (C/C++)

TECHNISCHE UNIVERSITÄT DARMSTADT

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

Behavior of 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 variable initialization and finalization

Default avoids copying

- Undefined initial value of private copy upon loop entry
- Undefined value of master's copy after loop exit

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

A variable can appear in both clauses

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

Private variable initialization and finalization (2)

```
#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
- C++ objects
  - Firstprivate initialization with copy constructor
  - Lastprivate finalization with copy assignment operator

Data dependences

- Parallelization must preserve the program’s correctness
- Data dependences may affect the program’s correctness
- 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)

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

```
#pragam omp prarallel 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;
```

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
  - Find two different values i,j of the loop index variable within the index range such that iteration i writes some element and j reads or writes the same element
- 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)

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

```
/* no dependence */
for ( i = 1; i < n; i += 2 )
    a[i] = a[i] + a[i-1];

/* no dependence */
for ( i = 0; i < n/2; i++ )
    a[i] = a[i] + a[i + n/2];

/* dependence */
for ( i = 0; i < n/2 + 1; i++ )
    a[i] = a[i] + a[i + n/2];

/* hard to decide */
for ( i = 1; i < n; i++ )
    a[i] = a[f(i)] + b[f(i)];
```

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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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; i++ ) {
  x = b[i] - c[i];
  d[1] = 2 * x;
}
y = x + d[1] + d[2];
```

- Parallelization via lastprivate temporary

```
#pragma omp parallel 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

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
- 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 this part of the loop

Non-removable dependences (2)

- Scalar expansion
  - Computation depends on scalar computed in each iteration
  - 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 a 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

TECHNISCHE UNIVERSITÄT DARMSTADT

Static

- Distribution is done (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 is done 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 ✔

Scheduling strategies

- Distribution of iterations occurs in chunks
- Chunks may have different sizes
- Chunks are assigned either statically or dynamically
- There are different assignment algorithms

- Schedule clause
  - **`schedule([modifier[, modifier]:]kind[,chunk_size])`**
- Kind
  - **`static, dynamic, guided, auto, runtime`**
- Modifier
  - **`monotonic, nonmonotonic, simd`**

Scheduling strategies (2)

TECHNISCHE UNIVERSITÄT DARMSTADT

Static without chunk size

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

Static with chunk size

- Chunks with specified size are assigned in round-robin fashion
- “Interleaved” schedule

Dynamic

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

Guided

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

Example

10 iterations, 2 threads (red & green)

0	1	2	3	4	5	6	7	8	9

Example

TECHNISCHE UNIVERSITÄT DARMSTADT

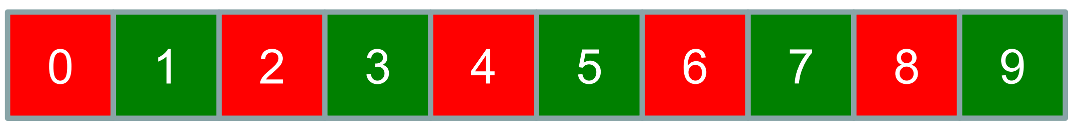
10 iterations, 2 threads (red & green)

Static without chunk size

Example

10 iterations, 2 threads (red & green)

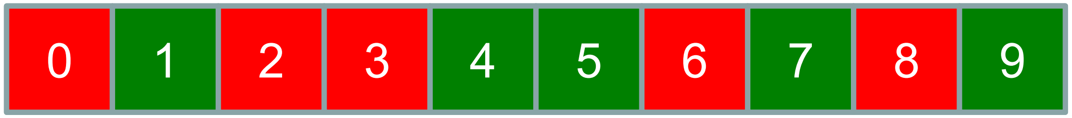
Static with chunk size = 1



Example

10 iterations, 2 threads (red & green)

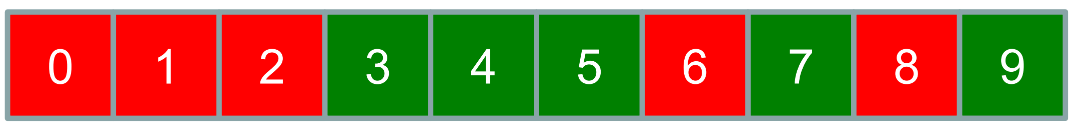
Dynamic with default chunk size (=1)



Example

10 iterations, 2 threads (red & green)

Guided



Scheduling strategies (3)

Auto (no chunk size)

- Gives implementation freedom to choose any possible mapping of iterations to threads

Runtime

- Scheduling strategy is determined by environment variable or API call
  - `setenv OMP_SCHEDULE` “*type[, chunksize]*”
  - `omp_set_schedule(...)`
- Otherwise scheduling implementation dependent

Correctness and scheduling

TECHNISCHE
UNIVERSITÄT
DARMSTADT

- 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

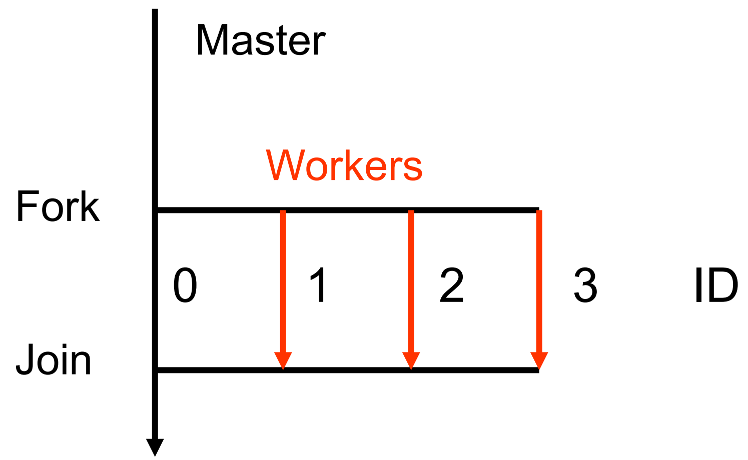
- Incremental parallelization of serial code using loop-level parallelism
- Potential hazards affecting correctness
  - Incorrect data sharing
  - Data dependences
- Scheduling strategy
  - Overhead vs. load balance
- No covered: thread affinity

SPMD-style parallelism and work-sharing

- 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-sharing constructs provide distribution of 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

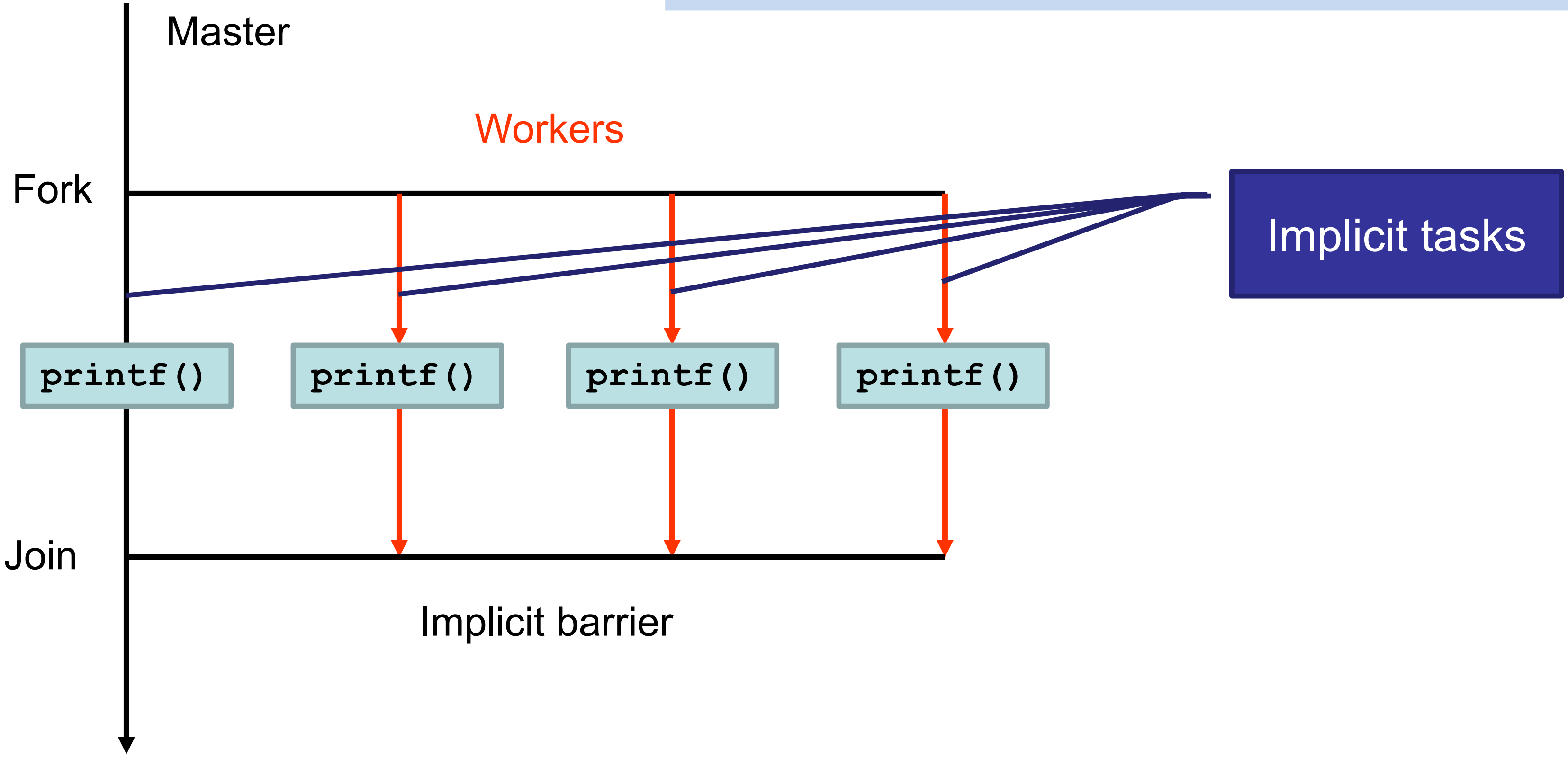
Fundamental construct that starts parallel execution

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

Execution model

TECHNISCHE
UNIVERSITÄT
DARMSTADT

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



Parallel vs. parallel for construct

TECHNISCHE UNIVERSITÄT DARMSTADT

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 directive

- n outputs total
- Work distribution

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

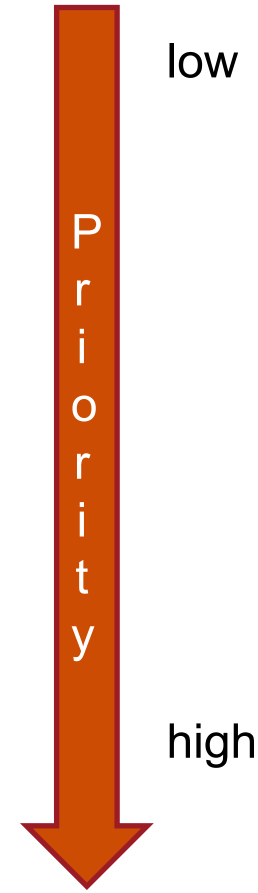
Restrictions

- Code encountered during a specific instance of the execution of a parallel construct is called parallel region
  - Must be a structured block
  - One or more statements
  - Entered at the top, left at the bottom
  - No branches into/out of the parallel region
  - Branches within the parallel region permitted
  - Program termination within parallel region permitted
    - C/C++: `exit()`

How to specify the number of threads

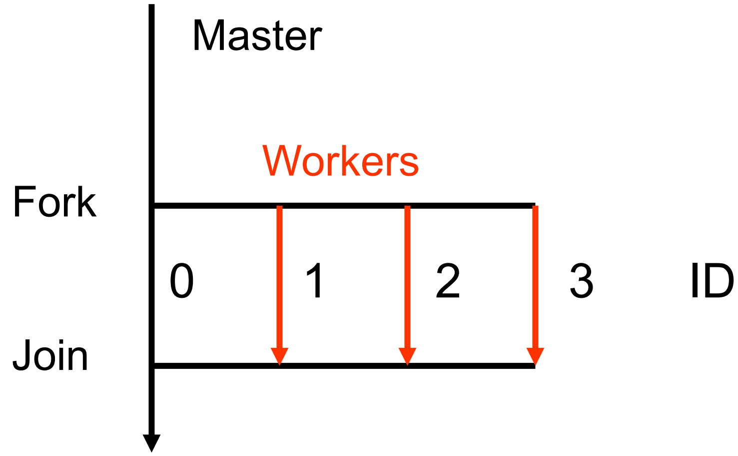
- `setenv OMP_NUM_THREADS n`
  - If set before program start, the application will create teams of the given size
- `omp_set_num_threads(n)`
  - Adjustment of the team size at runtime
  - Affects only subsequent parallel regions
- `numthreads` clause
  - Controls the number of threads for a particular parallel construct

```
#pragma omp parallel numthreads(4)
   […]
```



Thread numbers

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



Clauses

Clause	Purpose
if ([parallel :] *scalar-expression*)	Conditional parallelization
num_threads(*integer-expression*)	Number of threads
default(shared | none)	Data sharing attribute clauses
private(*list*)	
firstprivate(*list*)	
shared(*list*)	
reduction(op*: list*)	
copyin(*list*)	Copying between thread-private variables
proc_bind(master | close | spread)	Thread affinity

Construct vs. region revisited

TECHNISCHE UNIVERSITÄT DARMSTADT

Construct

- Lexical or static extent
- Code lexically enclosed

Region

- Dynamic extent
- Lexcial 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

```c
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

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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)
  {
    my_start = […]
    my_end   = […]
    work(my_start, my_end);
  }
}
```

Cumbersome for long call paths

Threadprivate directive

TECHNISCHE UNIVERSITÄT DARMSTADT

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

- Copies value of master 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

TECHNISCHE UNIVERSITÄT DARMSTADT

- Domain decomposition
  - Assignment based on thread number and total number of threads
- Work-sharing constructs
  - Loop 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

  Not covered
- Task construct
  - Defines unit of work to be assigned to a thread

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

Loop construct

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

Divides iterations of the following loop among the threads in a team

Clauses of the loop construct

- Subset of the clauses of the parallel loop construct
  - Exception: nowait clause - disables the implicit barrier at the end
- Equivalent behavior

Clauses of the loop construct

Clause	Purpose
private(*list*)	Specifies private semantics for a variable
firstprivate(*list*)	Initialization of private variables
lastprivate(*list*)	Finalization of private variables
linear(*list[: linear-step]*)	Related to SIMD execution
reduction(op*: list*)	Specifies a reduction operation on a variable
schedule*[modifier[, modifier]*:*]kind*[,*chunk_size*])	Controls distribution of work across the team of threads
collapse(n)	Collapses nested loop into one larger iteration space to be parallelized
ordered*[* (n) *]*	Execution order of ordered constructs the same as in serial execution
nowait	Disables the implicit barrier at the end

Collapse clause

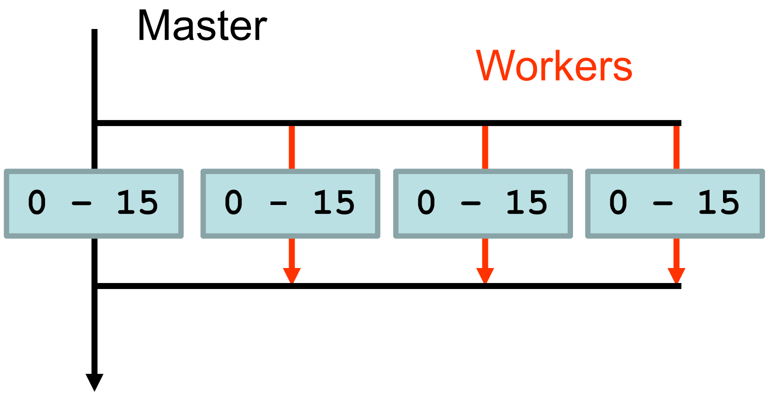
- Parameter specifies number of associated nested loops
- The iterations of all associated loops are collapsed into one larger iteration space to be divided according to the schedule clause
- The sequential execution of the iterations in all loops determines the order of iterations in the collapsed space

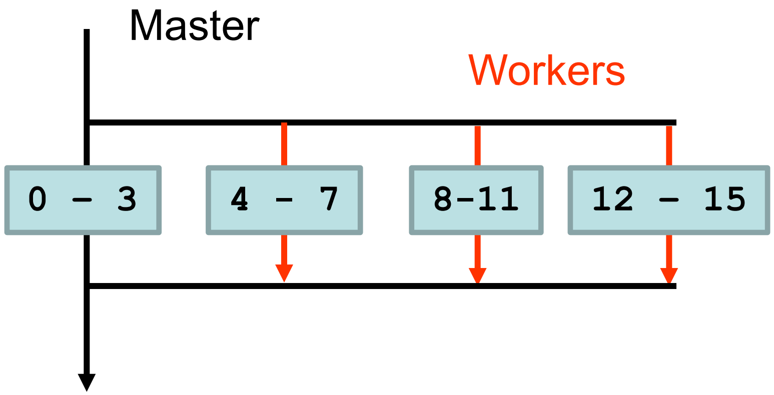
```
#pragma omp for collapse(2) schedule(static,1)
  for (i=0; i<2; i++)
     for (j=0; j<5; j++)
          a[i][j] = b[i][j];
}
```

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 loop 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: 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

```
#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++)
       […]
  }
```

- Remaining threads wait for the selected thread to finish the I/O operation at the implicit barrier unless there is a nowait

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-sharing 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-sharing constructs in the same order including the single construct
- Has implicit barrier
- Has nowait clause

Block structure and entry/exit of work-sharing constructs

- Contents of work-sharing constructs must have block structure
- Complete statements
- Block must be entered at the top
- Block must be left at the bottom
- No branching out of the block (e.g., return)
- Branches within the block are allowed
- Termination of the program within a block is allowed
  - C/C++: exit()

Collective execution of work-sharing constructs

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

- Each work-sharing region must be encountered by all threads in a team or by none at all
  - Exception: cancellation of innermost enclosing parallel region (not covered)
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team

Nesting of work-sharing 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-sharing construct executes its portion of work alone, therefore further division 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-sharing 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-sharing constructs not in the lexical extent are called **orphaned**

Behavior when reached

… from within a **parallel region**

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

Summary SPMD-style parallelism

TECHNISCHE UNIVERSITÄT DARMSTADT

Loop-level parallelism

- Only work-sharing
- Iterations of a loop must be independent
- Special case of a parallel region with one for directive
- Easier to understand and use
- Well-suited for incremental parallelization
- Limited to loops

SPMD-style parallelism

- Combination of replicated execution with work sharing
- More complex to use – requires prior identification of work-sharing opportunities
- Can be applied to more general and larger code regions
- Less parallel overhead for spawning threads and synchronization

Synchronization

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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 conditions and data races

TECHNISCHE UNIVERSITÄT DARMSTADT

Race condition

- Unenforced relative timing of concurrent operations
- Results in non-deterministic execution
- Failure in programs expected to be deterministic (e.g., scientific applications)

Data race

- Non-atomic concurrent access (with at least one write) to memory location
- Results in undefined behavior
- Earlier definitions often more general (shared data instead of memory location)
- Now precise definitions in C(++)11 standards

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
  x = 1;
```

```
Thread 0

#pragma omp atomic
  x = 2;
```

- Not deterministic but no atomicity violation

  ➔ race condition but no data race

Race condition due to round-off errors

Sum of all elements in an array

```
  double sum = 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 problem to solve

Data race (according to C11)

TECHNISCHE UNIVERSITÄT DARMSTADT

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)

TECHNISCHE UNIVERSITÄT DARMSTADT

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

  ➔ race condition

Value read or written

```
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)
```

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 in OpenMP

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

Correct version - neither race condition nor data race

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

- Determine first local maxima and 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 in OpenMP

TECHNISCHE UNIVERSITÄT DARMSTADT

Mutual exclusion

- Exclusive access to a shared data structure
- Can be used to ensure
  1. Only one thread has access to the data structure for the duration of the synchronization construct
  2. Accesses by multiple threads are interleaved at the granularity of the synchronization constructs

Event synchronization

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

OpenMP 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
- Ordered – imposes sequential order on the execution of the enclosed code section
- Master – code that should be executed only by the master thread
- Taskwait / taskgroup – waits on completion of task(s)
- Flush – enforces memory consistency

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 a 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

```
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];
      }
  }
```

- Better use reduction in this case

Named critical constructs

- Critical construct provides exclusive access with respect to all critical constructs in the entire program
- However, sometimes different critical constructs are used to protect different data structures and are therefore unrelated
- Still 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 - example

```
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

TECHNISCHE UNIVERSITÄT DARMSTADT

- Lexical nesting - why?
- Dynamic nesting – can easily lead to deadlock
- To avoid overhead, OpenMP does not provide special support for nested critical constructs
- If program contains nested critical constructs make sure that all threads execute them 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
  - Load-linked, store-conditional (LL/SC) on MIPS
  - Compare-and-exchange (CMPXCHG) on Intel x86
- 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 – example

Calculating a histogram

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

- Advantage of using the atomic directive
  - Multiple updates to different locations can occur concurrently
  - However, false sharing is possible if multiple shared variables reside on the same cache line

Atomic construct

```
#pragma omp atomic [seq_cst[,]] atomic-clause [[,] seq_cst] new-line
  expression-statement
```

```
#pragma omp atomic [seq_cst] new-line
  expression-statement
```

- Permitted form of expression statement depends on atomic clause
- Further variant with structured block = short sequence of statements to update and read a value or vice versa (not covered, details in standard)

Expression statement

If clause is **`read`**

```
v = x;
```

If clause is **`write`**

```
x = expr;
```

If clause is **`update`** or not present

```
x++;
++x;
x--;
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

If clause is **`capture`**

```
v = x++;
v = ++x;
v = x--;
v = --x;
v = x binop= expr;
v = x = x binop expr;
v = x = expr binop x;
```

binop

+, *, -, /, &, ^, |, <<, >>

Atomic clauses

Clause	Effect
read	Forces an atomic read
write	Forces an atomic write
update	Forces an atomic update
capture	Forces an atomic update while also capturing the original or final value
no clause	Equivalent to update
seq_cst	Forces the atomically performed operation to include an implicit flush operation

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 – example

```
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 – example (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

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);
}
```

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();
  }
}
```

In a serialized parallel region the barrier is trivially complete when the thread arrives at the barrier

- All threads executing the parallel region must execute the barrier
- Cannot be placed inside work-sharing constructs
- Implicit barrier at the end of work-sharing 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
- Master
- Flush
- Taskwait / taskgroup

Covered: Critical, Atomic, Lock routines, Barrier

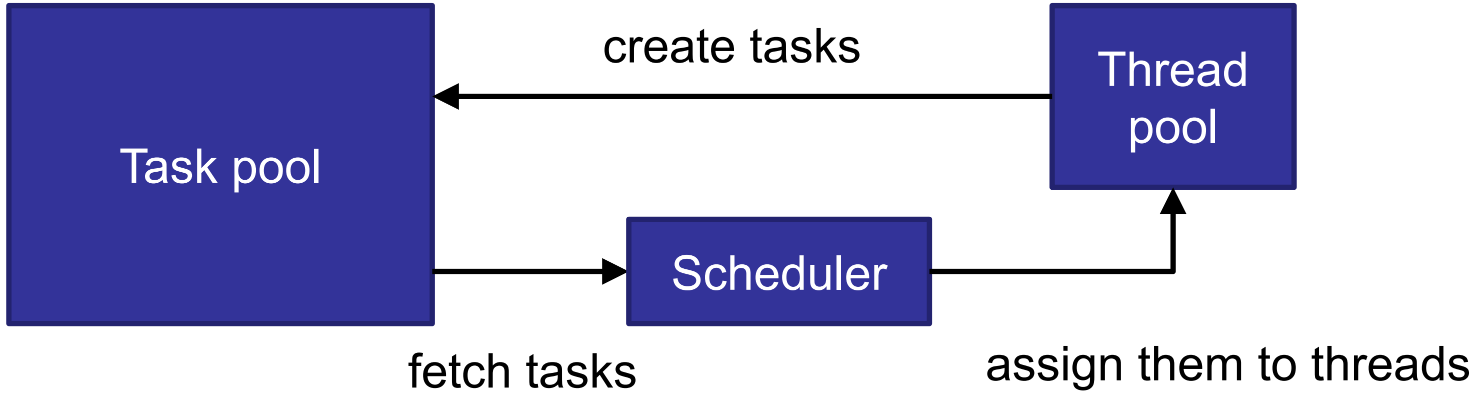
Not covered: Master, Flush

Later: Taskwait / taskgroup

Tasking

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. Note that a task may create new tasks
- Advantage: good load balance if problem is over-decomposed



Task construct

- Introduced with OpenMP specification 3.0
- 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

Clauses

Clause	Purpose
if([task:] *scalar-expression*)	Conditional asynchronous execution
final(*scalar-expression*)	Conditional asynchronous execution for this task and subtasks
untied	Any thread can resume task after suspension
default(shared | none)	Default data sharing semantics
mergeable	Mergeable task
private(*list*)	Specifies private semantics for a variable
firstprivate(*list*)	Initialization of private variables
shared(*list*)	Specifies shared semantics for a variable
depend(*dependence-type : list*)	
priority(*priority-value*)	Hint for task execution order

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

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

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 consruct

TECHNISCHE UNIVERSITÄT DARMSTADT

```
#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 attributes

TECHNISCHE UNIVERSITÄT DARMSTADT

- Variables inherited from task creation context are firstprivate
  - Caveat: default of parallel constructs is shared (!)
- The sharing attribute can be specified by explicit clauses
- Supports clauses similar to parallel construct:
  - **`shared(`*`variable-list`*`)`**
  - **`private(`*`variable-list`*`)`**
  - **`firstprivate(`*`variable-list`*`)`**
- Variables created inside a task are private

Data sharing example

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

a is shared among the implicit task and Task A. In Task B the local a is private

b is firstprivate in all tasks

c is firstprivate in Task A, but Task B shares c with the implicit task

d is private to Task A and Task B

Example: Quicksort

TECHNISCHE UNIVERSITÄT DARMSTADT

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

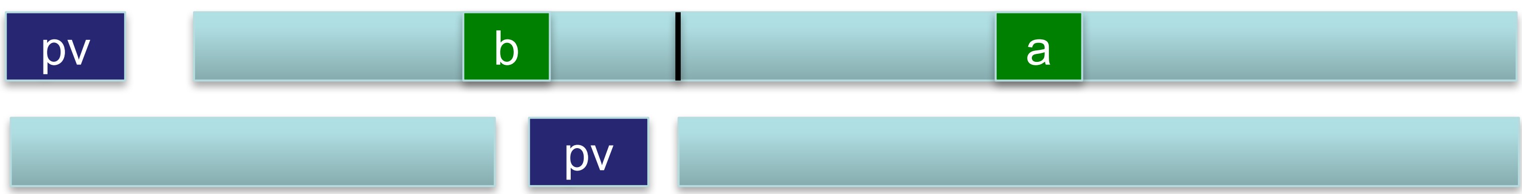
Quicksort – step 1

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

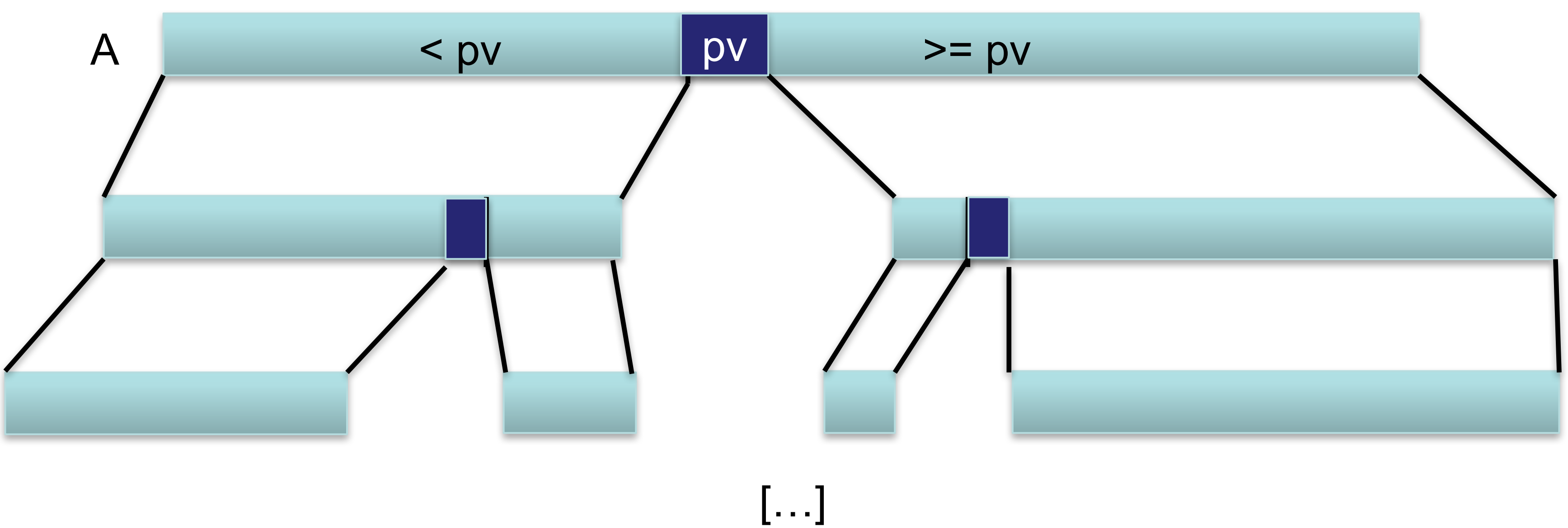


- Loop
  - Find the last element a < pv
  - Find the first element b >= pv
  - Swap a and b

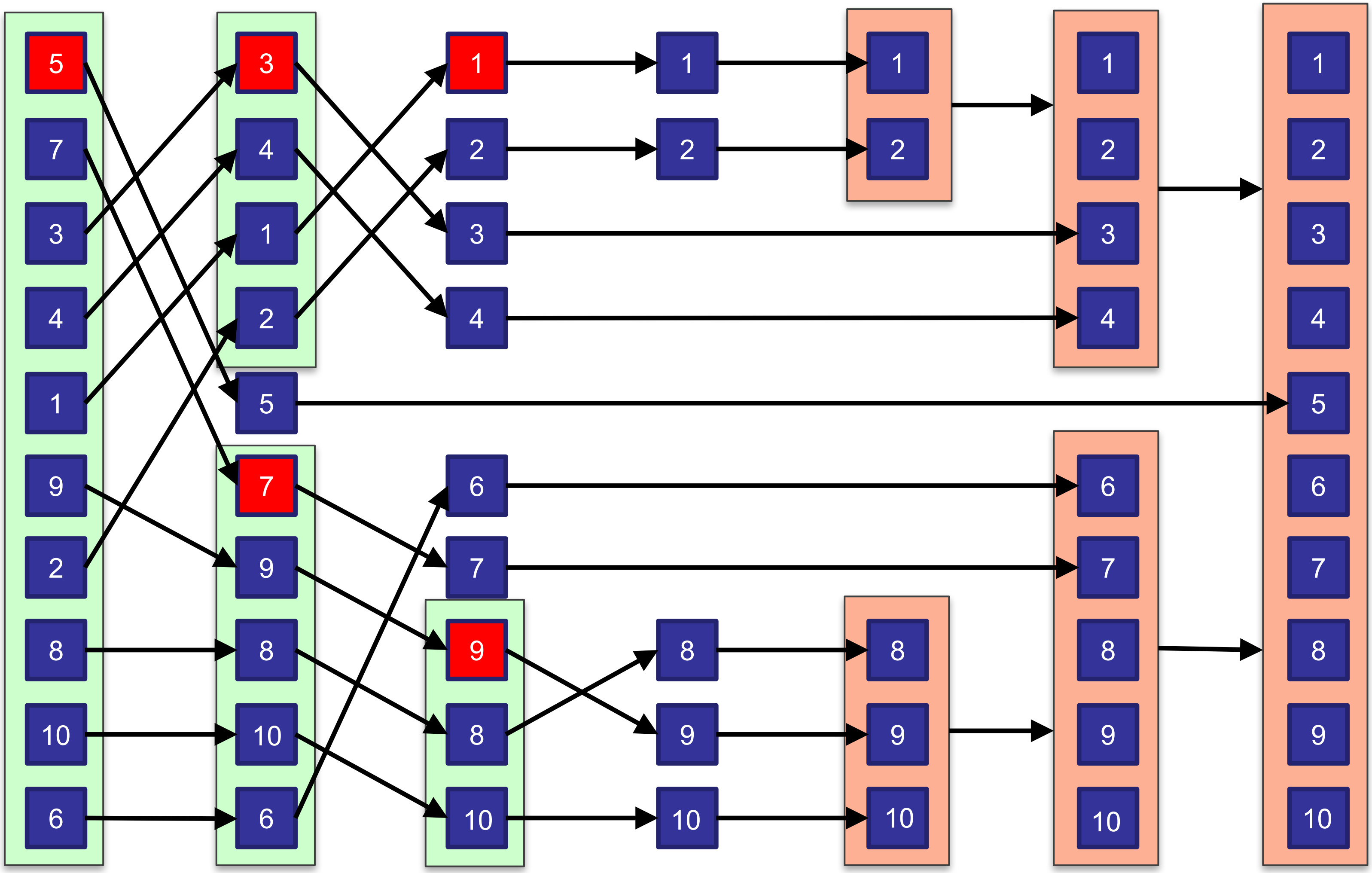
Quicksort – recursion

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 length )
{
  if ( length <= 1 ) return;
  int pv = A[length/2];
  int forw = 0;
  int backw = length-1;
  while ( forw < backw )
  {
     while ( forw < backw && A[forw] < pv ) forw ++;
     while ( forw < backw && A[backw] >= pv ) backw --;
     if ( A[forw] > A [backw] ) swap( A, forw, backw );
  }
  quicksort( A, forw );
  quicksort( &A[backw+1], length - backw - 1);
}
```

Quicksort with tasks

```
void quicksort_task( int* A, int length )
{
  if ( length == 1 ) return;
  int pv = A[length/2];
  int forwa = 0;
  int backw = length-1;
  while ( forw < backw )
  {
     while ( forw < backw && A[forw] < pv ) forw ++;
     while ( forw < backw && A[backw] >= pv ) backw --;
     if ( A[forw] > A [backw] ) swap( A, forw, backw );
  }

  #pragma omp task
     quicksort_task( A, forw );
  #pragma omp task
     quicksort_task( &A[backw+1], length - backw - 1 );
}
```

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

Quicksort – synchronization

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

Quicksort – synchronization (2)

```
#pragma omp single nowait
   quicksort_task(A, length);
#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

Quicksort with recursive taskwait

```
void quicksort_task( int* A, int length )
{
  if ( length == 1 ) return;
  int pv = A[length/2];
  int forwa = 0;
  int backw = length-1;
  while ( forw < backw )
  {
     while ( forw < backw && A[forw] < pv ) forw ++;
     while ( forw < backw && A[backw] >= pv ) backw --;
     if ( A[forw] > A [backw] ) swap ( A, forw, backw );
  }
  #pragma omp task
     quicksort_task ( A, forw );
  #pragma omp task
     quicksort_task ( &A[backw+1], length - backw - 1 );
  #pragma omp taskwait
}
```

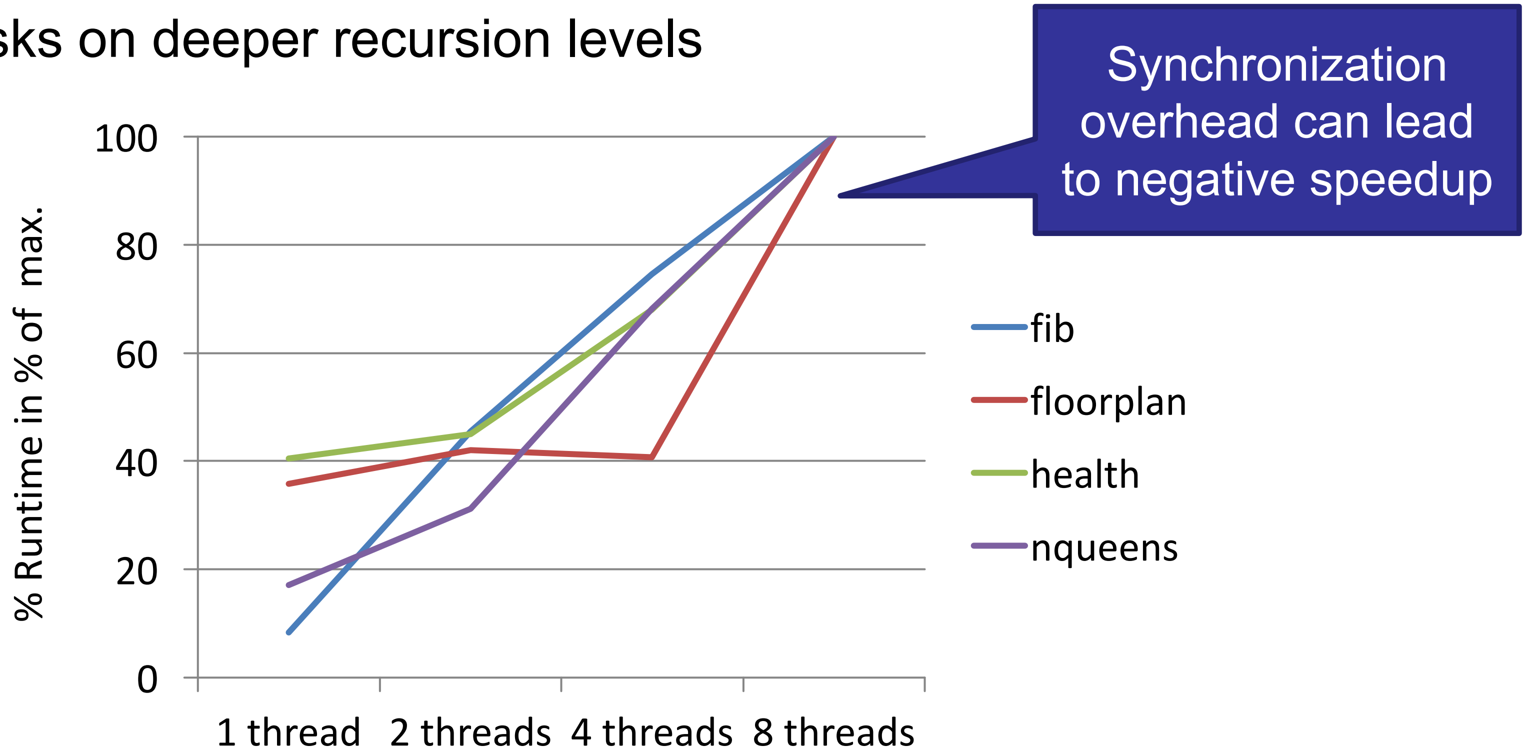
Task overhead

- 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 length )
{
  if ( length == 1 ) return;
  int pv = A[length/2];
  int forwa = 0;
  int backw = length-1;
  while ( forw < backw )
  {
     while ( forw < backw && A[forw] < pv ) forw ++;
     while ( forw < backw && A[backw] >= pv ) backw --;
     if ( A[forw] > A [backw] ) swap( A, forw, backw );
  }
  #pragma omp task if ( forw > MIN_VAL )
     quicksort_task( A, forw );
  #pragma omp task if ( length – backw – 1 > MIN_VAL )
     quicksort_task( &A[backw+1], length – backw – 1 );
  #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
  - The task 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 of condition on subsequent recursion levels
- Difference between if and final clause
  - Final clause creates no further tasks if condition is true but if clause does

Quicksort with final clause

```
void quicksort_task( int* A, int length )
{
  if ( length == 1 ) return;
  int pv = A[length/2];
  int forwa = 0;
  int backw = length-1;
  while ( forw < backw )
  {
     while ( forw < backw && A[forw] < pv ) forw ++;
     while ( forw < backw && A[backw] >= pv ) backw --;
     if ( A[forw] > A [backw] ) swap( A, forw, backw );
  }
  #pragma omp task final ( forw <= MIN_VAL )
     quicksort_task( A, forw );
  #pragma omp task final ( length - backw - 1 <= MIN_VAL )
     quicksort_task( &A[backw+1], length - backw - 1 );
  #pragma omp taskwait
}
```

Task dependences

- Often, the output of one tasks is needed as input for another task
  - Requires certain order of task execution
- Current synchronization mechanisms (barriers and taskwait) not powerful enough
  - Leads to (short) phases of task parallel execution followed by a synchronization phase
  - Limits use of tasks
  - Increases synchronization overhead
  - Reduces the amount of exploitable concurrency
- 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
- Runtime enforces dependences during execution

Task dependences (3)

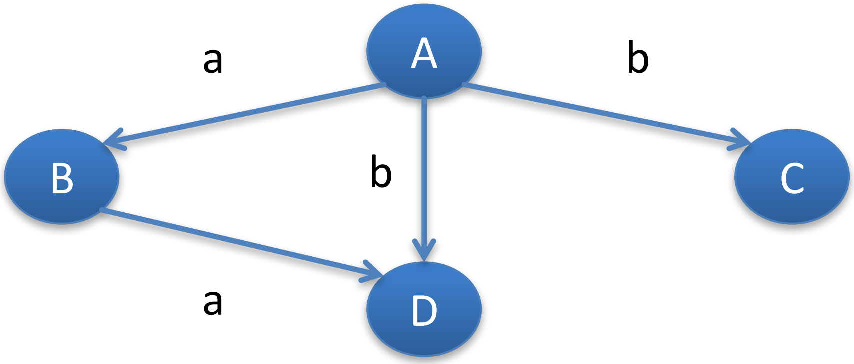
```
#pragma omp task depend(in|out|inout: variable-list)
```

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

Task dependences - example

```
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
{}
```

Execution order



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

Taskgroup

TECHNISCHE UNIVERSITÄT DARMSTADT

- 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

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