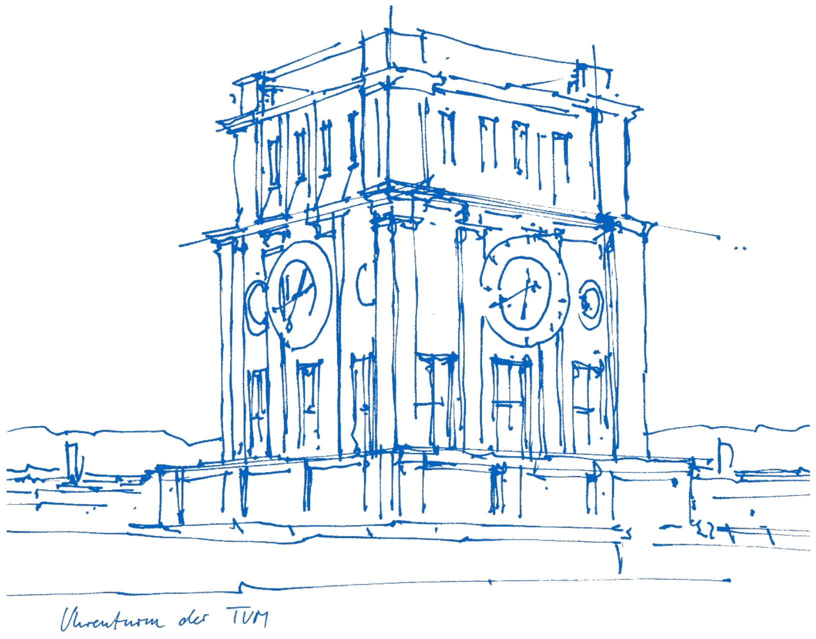
Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Lecture 4: OpenMP Beyond the Basics

- Seg. 15: Reductions in OpenMP
- Seg. 16: Correctness Issues in OpenMP
- Seg. 17: Loop Transformations
- Seg. 18: The OpenMP Memory Model
- Seg. 19: OpenMP Tasking
- Seg. 20: OpenMP Wrapup

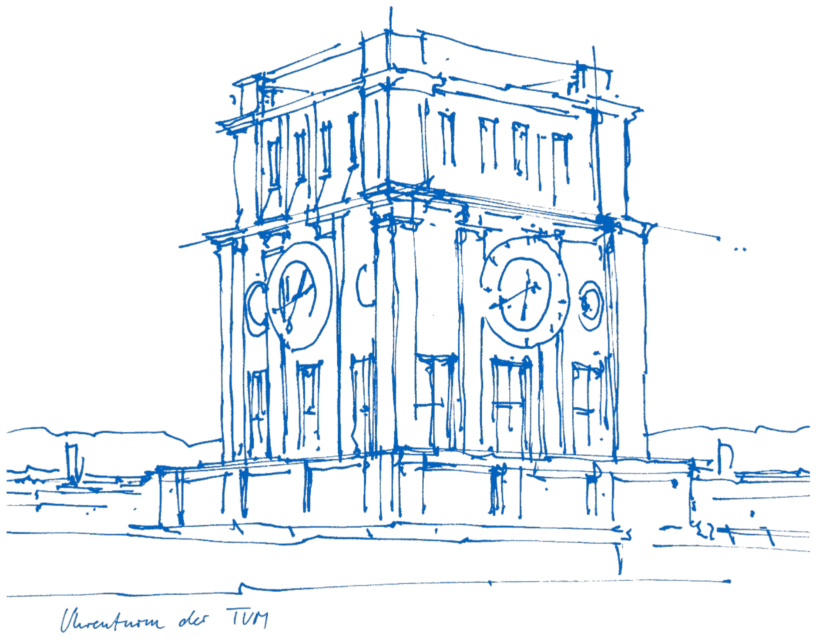


Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Lecture 4: OpenMP Beyond the Basics

- Seg. 14: Synchronization
- Seg. 15: Reductions in OpenMP
- Seg. 16: Correctness Issues in OpenMP
- Seg. 17: Loop Transformations
- Seg. 18: The OpenMP Memory Model
- Seg. 19: OpenMP Tasking
- Seg. 20: OpenMP Wrapup



Summary: OpenMP 101

OpenMP was created to standardize the programming of shared memory systems

- First standard in 1997, currently at OpenMP 5.2
- Goals were easy of use, simplicity and portability

Key concepts

- Parallel regions
- Worksharing through parallel for loops
- Additional clauses to control distribution, synchronization, …
- Options to control data visibility/sharing

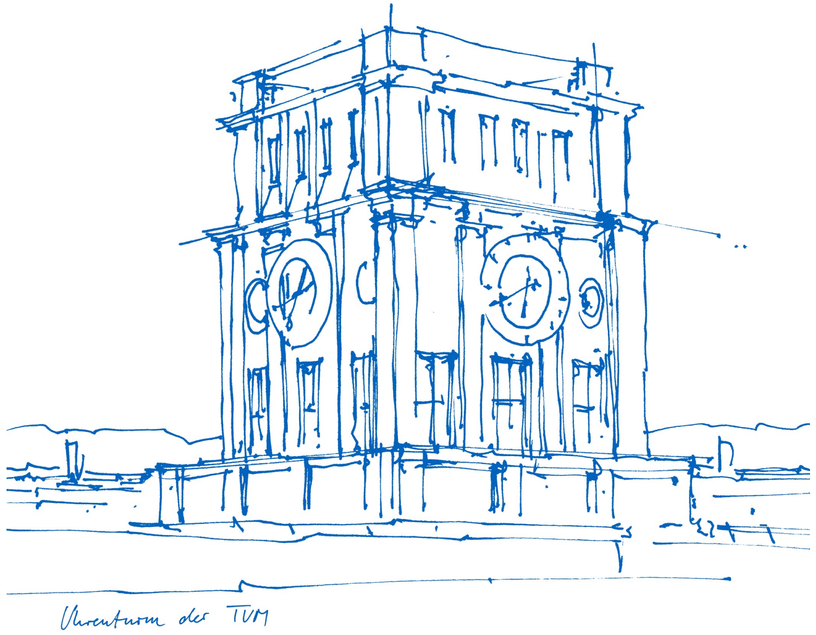
Programmer responsibility

- Ensure no loop dependencies exist
- Ensure the right variables or private or shared
- Ensure the necessary synchronization is added

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 14
Synchronization



Data Synchronization

As with Pthreads, we need options to synchronize data accesses

- Communication is through shared variables
- Synchronization has to be separate

OpenMP offers a wide range of pragma based options

- Barriers
- Masked regions
- Single regions
- Critical sections
- Atomic statements
- Ordering construct

Additional runtime routines for locking

Barrier

Key synchronization construct

- Synchronizes all the threads in a team
- Each thread waits until all other threads in the team have reached the barrier

Each parallel region has an implicit barrier at the end

- Synchronizes the end of the region
- Can be switched off by adding `nowait`

Additional barriers can be added when needed

```
#pragma omp barrier
```

Warning

- Can cause load imbalance
- Use only when really needed

Master Region

```
#pragma omp master
        block
```

A master region enforces that only the master executes the code block

- Other threads skip the region
- No synchronization at the beginning of region

Possible uses

- Printing to screen
- Keyboard entries
- File I/O

Warning:
Deprecated

Masked Region

```
#pragma omp masked
        block
```

The **masked** construct (without arguments) declares a region that only the **primary** thread executes the code block

- Other threads skip the region
- No synchronization at the beginning of region

Possible uses

- Printing to screen
- Keyboard entries
- File I/O

Masked Region

```
#pragma omp masked [filter(integer-expression)]
          block
```

The **masked** construct (with arguments) declares a region that only the threads execute that are specified in the integer expression (relative to the current team)

- Other threads skip the region
- No synchronization at the beginning of region

Possible Uses

- Designate a different thread than the primary one for I/O
- Possibly combined with a “nowait” → can create overlap

Single Region

```
#pragma omp single [parameter]
        block
```

A single region enforces that only a (arbitrary) single thread executes the code block
- Other threads skip the region
- Implicit barrier synchronization at the end of region (unless `nowait` is specified)

Possible uses
- Initialization of data structures

Critical Section

```
#pragma omp critical [(Name)]
        block
```

Mutual exclusion
- A critical section is a block of code
- Can only be executed by only one thread at a time.
- Compare to Pthreads locks

Critical section name identifies the specific critical section
- A thread waits at the beginning of a critical section until its available
- All unnamed critical directives map to the same name

Keep in mind
- Critical section names are global entities of the program
- If a name conflicts with any other entity, program behavior is unspecified
- Avoid long critical sections for performance reasons

Atomic Statements

```
#pragma ATOMIC
     expression-stmt
```

The ATOMIC directive ensures that a specific memory location is updated atomically

Has to have the following form:

- x binop= expr
- x++ or ++x
- x-- or --x

- where x is an lvalue expression with scalar type
- and expr does not reference the object designated by x

Equivalent to using critical section to protect the update

Useful for simple/fast updates to shared data structures

- Avoids locking
- Often implemented directly by native instructions

Simple Runtime Locks

In addition to pragma based options, OpenMP also offers runtime locks

- Same concept as Pthread mutex
- Locks can be held by only one thread at a time.
- A lock is represented by a lock variable of type omp_lock_t.

Operations

omp_init_lock(&lockvar)	initialize a lock
omp_destroy_lock(&lockvar)	destroy a lock
omp_set_lock(&lockvar)	set lock
omp_unset_lock(&lockvar)	free lock
logicalvar = omp_test_lock(&lockvar)	check lock and possibly set lock

returns true if lock was set by the executing thread.

Example: Simple Lock

```
#include <omp.h>
int id;
omp_lock_t lock;

omp_init_lock(lock);
#pragma omp parallel shared(lock) private(id)
{
  id = omp_get_thread_num();
  omp_set_lock(&lock); //Only a single thread writes
    printf(“My Thread num is: %d”, id);
  omp_unset_lock(&lock);

  while (!omp_test_lock(&lock))
    other_work(id);         //Lock not obtained
  real_work(id);            //Lock obtained
  omp_unset_lock(&lock);    //Lock freed
}
omp_destroy_lock(&lock);
```

locked

locked

Nestable Locks

Similar to simple locks

But, nestable locks can be set multiple times by a single thread.

- Each set operation increments a lock counter
- Each unset operation decrements the lock counter

If the lock counter is 0 after an unset operation, lock can be set by another thread

Separate routines for nestable locks

Look them up ☺

Ordered Construct

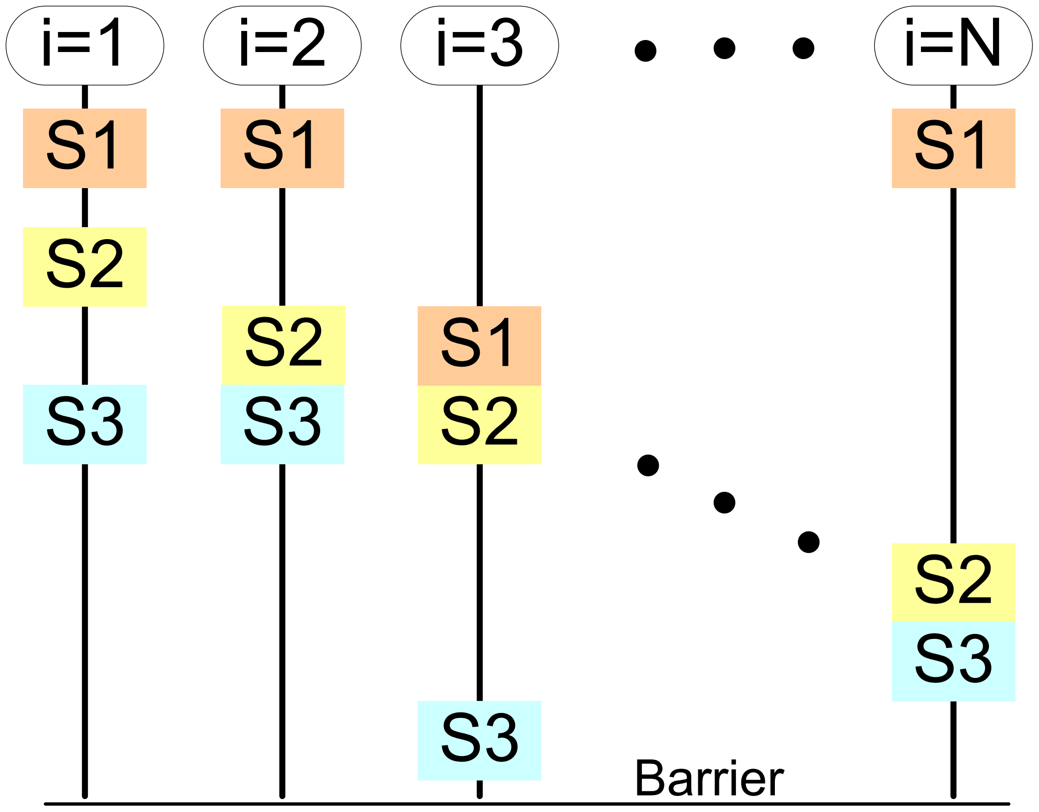
```
#pragma omp for ordered
    for (...)
     { ...
       #pragma omp ordered
       { ... }
       ...
     }
```

Construct must be within the dynamic extent of an `omp for` construct with an ordered clause.

Ordered constructs are executed strictly in the order in which they would be executed in a sequential execution of the loop.

Example: ordered clause

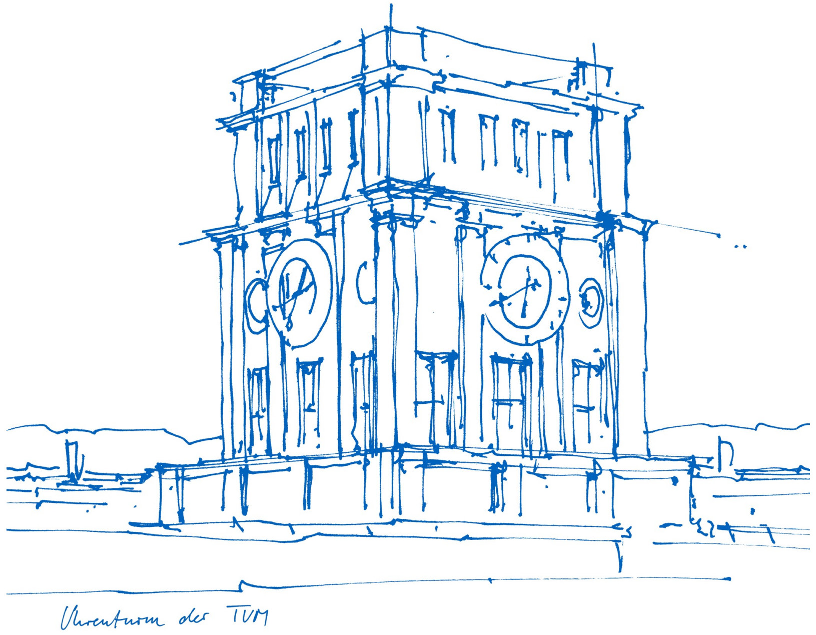
```
#pragma omp for ordered
for (...)
 { S1
   #pragma omp ordered
   { S2 }
   S3
 }
```



Lecture IN-2147
Parallel Programmin

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 15
OpenMP Reductions



Reductions

Synchronizing at the end of parallel regions is often not enough

- Need to communicate results of the region
- Aggregate data at primary thread
- Inform next parallel region

Reductions

- Perform operations on results from all threads
- Aggregate results
- Make it available to primary thread

Problem

- Potentially costly and time sensitive operation
- Hard to implement by oneself
- Should be part of the base language

OpenMP Reductions

OpenMP offers a special clause to specify reductions

```
        reduction(operator: list)
```

This clause performs a reduction …
… on the variables that appear in *list*,
… with the operator *operator*.
… across the values “at thread end“/“last iteration“ of ***each thread (!)***

Variables must be shared scalars
operator is one of the following:
`+, *, -, &, ^, |, &&, ||`

Reduction variable can only appear in statements with the following form:

- `x = x operator expr`
- `x binop= expr`
- `x++, ++x, x--, --x`

User defined reductions are an option in newer versions of OpenMP

Example: Reduction

```
int a=0;
#pragma omp parallel for reduction(+: a)
for (int j=0; j<4; j++)
{
  a = j;
}
printf("Final Value of a=%d\n", a);
```

OMP_NUM_THREADS=4

Final value of a:

0+1+2+3 = 6

OMP_NUM_THREADS=2

Final value of a:

1+3 = 4 (in most cases, assuming static loop scheduling)

Example: Reduction (cont.)

```
int a=0;
#pragma omp parallel for reduction(+: a)
for (int j=0; j<100; j++)
{
  a = j;
}
printf("Final Value of a=%d\n", a);
```

OMP_NUM_THREADS=4

Final value of a:

24+49+74+99 = 246

Thread 0:	0-24	last value is 24
Thread 1:	25-49	last value is 49
Thread 2:	50-74	last value is 74
Thread 3:	75-99	last value is 99

Example: Reduction (cont.)

```
int a=0;
#pragma omp parallel for reduction(+: a)
for (int j=0; j<100; j++)
{
  a += j;
}
printf("Final Value of a=%d\n", a);
```

Final value of a:

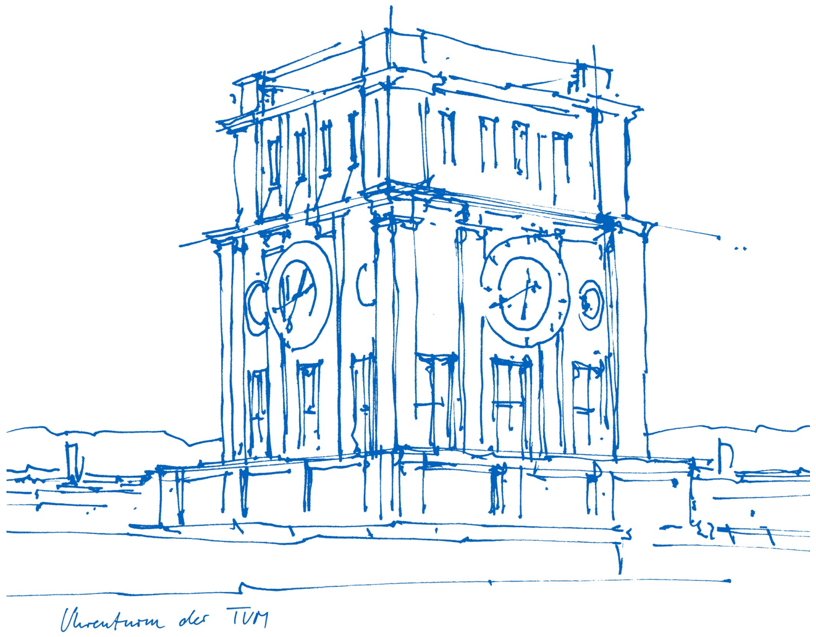
$0+1+2+\ldots+99 = (99*100)/2 = 4950$

(for any number of threads)

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 16
Correctness Issues in OpenMP



Correctness Issues in OpenMP

Programmer responsibility

- Ensure no loop dependencies exist
- Ensure the right variables or private or shared
- Ensure the necessary synchronization is added

Correctness Issue 1: Data Races

- Unsynchronized, conflicting accesses to shared variables
- Can lead to unexpected and incorrect results
- Hard to detect and fix

Correctness Issue 2: Loop dependencies

- Prevent parallelization of loops (or lead to races)
- Require careful analysis
- Can in some cases be fixed by transforming loops

Correctness Issue 3: Aliasing

- Do two separate data structures in memory?

Correctness Issue 1 / Dealing with Races

Two threads access the same shared variable

- At least one thread modifies the variable
- The accesses are concurrent, i.e. unsynchronized

Leads to non-deterministic behavior

- Depends on timing of accesses

Example:

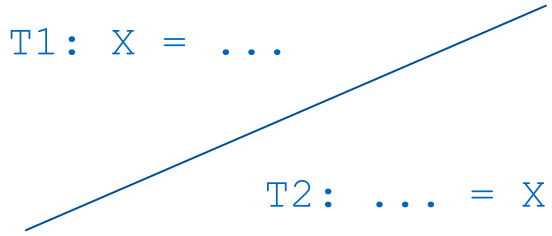
```
static double farg1,farg2;
#define FMAX(a,b) (farg1=(a),farg2=(b),farg1>farg2?farg1:farg2)

#pragma omp parallel for shared(bar, foo, THRESH)
for (x=0; x<1000;x++)
   double T = FMAX(0.1111*foo*bar[x],THRESH);
    <work with value T>
```

Types of Races

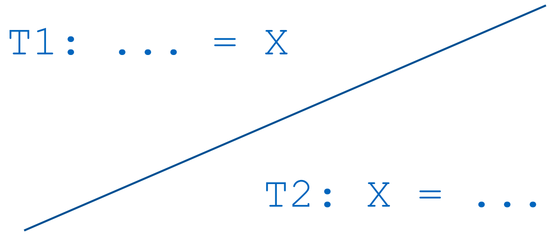
Three different categories of races

Read after Write (RAW)



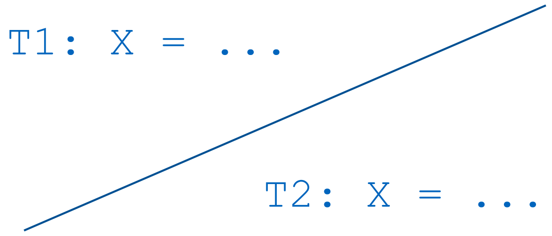
```
T1: X = ...
                T2: ... = X
```

Write after Read (WAR)



```
T1: ... = X
                T2: X = ...
```

Write after Write (WAW)



```
T1: X = ...
                T2: X = ...
```

Races can be benign

- Can deal with old or new value at a read
- Know the same value gets written

In most cases, we need to either introduce additional synchronization

- Add happens before relationship
- Examples: mutexes, barriers, …

Or we need to apply the right privatization

Race Detection Tools

Races are hard to find with traditional approaches

- Non-deterministic
- Debugging changes timing

Use dedicated race detection tools

- Static and/or dynamic instrumentation of all memory accesses
- Tracking synchronization
- Detection of unsynchronized accesses to the same memory

Examples of tools (not exhaustive)

- Helgrind (open source)
  - Dynamic, based on valgrind tool suite
- Intel Inspector (commercial)
  - Part of Intel's development suite
- Thread Sanitizer (open source)
  - Static instrumentation via LLVM
- Archer (open source)
  - Combines Thread Sanitizer's approach with OpenMP semantics

Correctness Issue 2 / Dealing with Dependencies

OpenMP requires the programmer to guarantee independence of iterations

Which of those codes are parallelizable?

```
Do i=1,n
 A(i)=5*B(i)+A(i)
Enddo
```

```
Do i=1,n
 A(i-1)=5*B(i)+A(i)
Enddo
```

```
Do i=1,n
 tmp=5*B(i)
 A(i)=tmp
Enddo
```

Answer: it depends!

Need to understand data dependencies

- Relationship between iterations
- Relationship between variables/arrays

Types of Data Dependencies

I(S)	= set of memory locations read (input)
O(S)	= set of memory locations written (output)

True Dependence / Flow Dependence	Anti Dependence	Output Dependence
$O(S1) \cap I(S2) \neq \emptyset$	$I(S1) \cap O(S2) \neq \emptyset$	$O(S1) \cap O(S2) \neq \emptyset$
$(S1\ \delta\ S2)$	$(S1\ \delta^{-1}\ S2)$	$(S1\ \delta^{o}\ S2)$
The output of S1 overlaps with the input of S2	The input of S1 overlaps with the output of S2	The output of S1 and S2 write to same location

```
S1: X = ...
         ...
S2: ... = X
```

```
S1: ... = X
         ...
S2: X = ...
```

```
X = ...
X = ...
```

Loop Dependencies

Loop Independent Dependencies

All dependencies are with iterations
No dependencies across iterations

Example:

```
for (i = 0; i < 4; i++)
    S1: b[i] = 8;
    S2: a[i] = b[i] + 10;
```

Iterations can be executed in parallel

Loop Carried Dependencies

Dependencies across iterations
Computation in one iteration depends on data written in another

Example:

```
for (i = 0; i < 4; i++)
    S1: b[i] = 8;
    S2: a[i] = b[i-1] + 10;
```

Iterations cannot be (easily/directly) executed in parallel

Correctness Issue 3 / Dealing Aliasing

Question: which variables refer to the same physical location?

- Simple for individual variables
- Harder for pointers
- Really hard for C-style pointers
- Impossible to statically determine for arbitrary pointer structures

```
Foo(a,b,n)
 Do i=1,n
  A(i)=5*B(i)+A(i)
 Enddo
```

```
int A[100],B[100];

Foo(A,B,100);
Foo(&A[1],B,100);
C=A+5; Foo(A,C,100);
Foo(A,A,100);
```

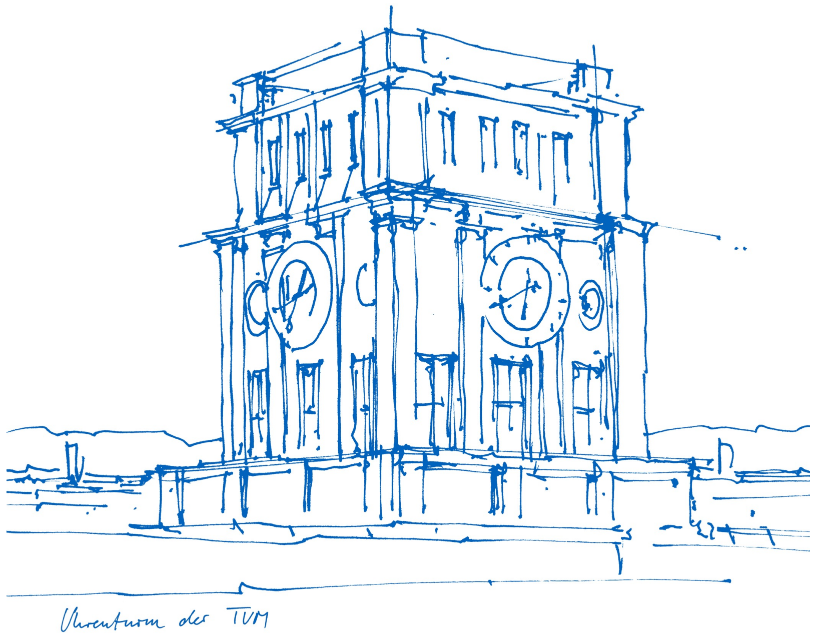
Options

- Programmer has to know data structures and know about aliases
- Library developer may have to assert conditions in the API
- Dynamic inspector/executor tests
- Compiler annotations to assert non-aliasing

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 17
Loop Transformations



Program Order vs Dependence

The sequential order imposed by the program is too restrictive
- Just one way to write the problem
- Possibly hides possible parallelism

Only need to uphold all dependencies to guarantee program correctness
- Look at partial order of all dependences
- Aside from that re-orderings are legal

Code transformations can help, but they have to be safe

– A reordering transformation **preserves a dependence** if it preserves the relative execution order of the source and sink of that dependence.

– Consequence: any reordering transformation that preserves every dependence in a program leads to an **equivalent** computation.

– A transformation is said to be **valid** for the program to which it applies if it preserves all dependences in the program.

Transformation 1: Loop Interchange

```
do l=1,10                            do l=1,10
  do m=1,10                            do k=1,10
    do k=1,10                            do m=1,10
      A(l,m,k)=A(l,m,k)+B     ≡            A(l,m,k)=A(l,m,k)+B
    enddo                                enddo
  enddo                                enddo
enddo                                enddo
```

A loop interchange is safe for a perfect loop nest, if
we only have loop independent dependencies

A loop interchange is safe for a perfect loop nest, if
the direction of all dependences is in a way that
all updates are done in the same way despite reordering
→ look at: dependency vectors and their directions

Assuming no aliasing

Loop Interchange (2)

```
do l=1,10
   do m=1,10
      do k=1,10
         A(l+1,m+2,k+3)=A(l,m,k)+B
      enddo
   emddo
enddo
```

$$\equiv$$

```
do l=1,10
   do k=1,10
      do m=1,10
         A(l+1,m+2,k+3)=A(l,m,k)+B
      enddo
   enddo
enddo
```

Assuming no aliasing

Transformation 2: Loop Distribution / Loop Fission

Transforms loop-carried dependences into loop-independent dependences.

```
do j=2,n
S1:  a(j)= b(j)+2
S2:  c(j)= a(j-1) * 2
enddo
```

$\Rightarrow$

```
do j=2,n
S1:  a(j)= b(j)+2
enddo

do j=2,n
S2:  c(j)= a(j-1) * 2
enddo
```

Safety of loop distribution

- Two sets of statements in a loop nest can be distributed into separate loop nests, if no dependence cycle exists between those groups.
- Dependences carried by outer non-distributed loops can be ignored.
- The order of the new loops has to preserve the dependences among the statement sets.

To consider

- It generates multiple parallel loops, thus decreasing granularity.
- Barrier synchronization might be required between the generated loops.

Transformation 3: Loop Fusion

Combine two loops

```
do i=1,n
  a(i)= b(i)+2
enddo

do i=1,n
  c(i)= d(i+1) * a(i)
enddo
```

→

```
do i=1,n
  a(i)= b(i)+2
  c(i)= d(i+1) * a(i)
enddo
```

Loop fusion is the dual transformation to fission. It combines subsequent loops.

- Loop fusion increases granularity (possibly reduces overhead)
- Loop fusion may eliminate the need for a barrier
- Loop fusion can introduce loop carried dependencies

Possible Pitfalls for Loop Fusion (Sequentially)

Incorrect Loop Fusion

```
do i=1,n
S1:  a(i)= b(i)+2
enddo

do i=1,n
S2:  c(i)= d(i+1) * a(i+1)
enddo
```

```
do i=1,n
S1:  a(i)= b(i)+2
S2:  c(i)= d(i+1) * a(i+1)
enddo
```

Correct Loop Fusion

```
do i=1,n
S1:  a(i)= b(i)+2
enddo

do i=1,n
S2:  c(i)= d(i+1) * a(i-1)
enddo
```

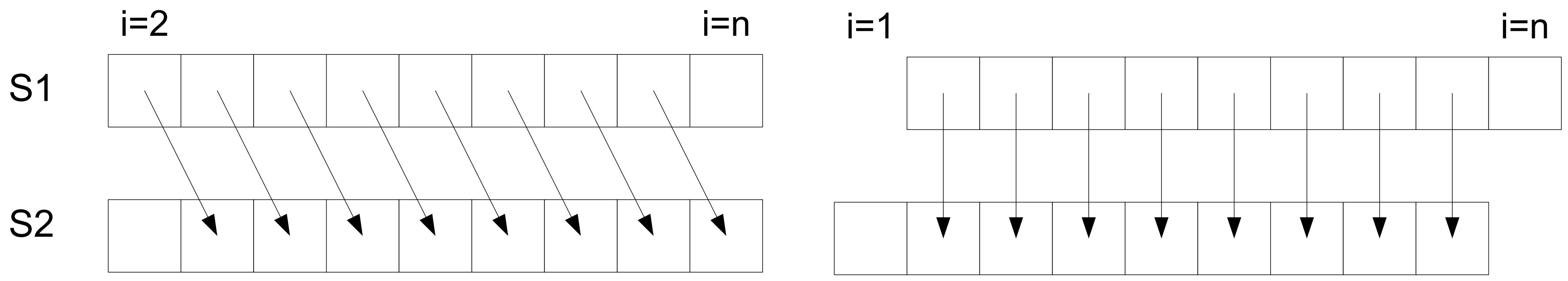
```
do i=1,n
S1:  a(i)= b(i)+2
S2:  c(i)= d(i+1) * a(i-1)
enddo
```

Transformation 4: Loop Alignment

Loop alignment changes a carried dependence into an independent dependence.

```
do i=2,n
S1:   a(i)= b(i)+2
S2:   c(i)= a(i-1) * 2
enddo
```

Idea: Shift of computations into other iterations.
Extra iterations and tests are overhead.



Loop Alignment

Loop alignment changes a carried dependence into an independent dependence.

```
do i=2,n
S1:  a(i)= b(i)+2
S2:  c(i)= a(i-1) * 2
enddo
```

$\Rightarrow$

```
do i=1,n
S1:  if (i>1) a(i)= b(i)+2
S2:  if (i<n) c(i+1)= a(i) * 2
enddo
```

First and last iteration can be peeled of:

```
c(2)=a(1) * 2
do i=2,n-1
S1:  a(i)= b(i)+2
S2:  c(i+1)= a(i) * 2
enddo
a(n)=b(n) + 2
```

In general, we can eliminate all carried dependences in a single loop if

– Loop contains no recurrence (dependence cycle) and
– Distance of each dependence is a constant independent of the loop index

Summary Loop Transformations

Loops that do not carry any dependences can be parallelized

The outermost loop should be parallelized, if possible

- Parallel inner loops can be moved outside by loop interchange.

Transformations can eliminate carried dependences

Examples:

- Loop fission/distribution
- Loop alignment

Transformations can improve efficiency

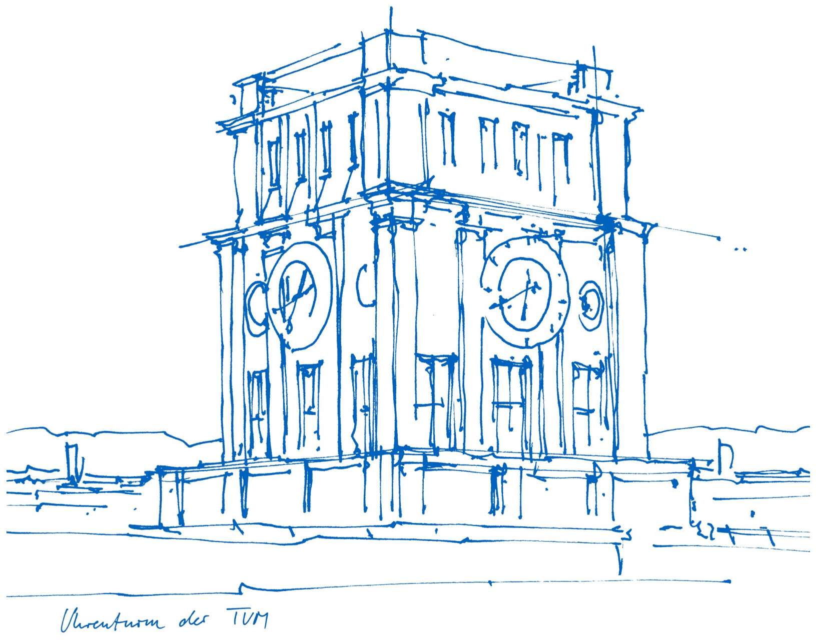
Examples:

- Loop fusion
- Loop interchange

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 18
The OpenMP Memory Model



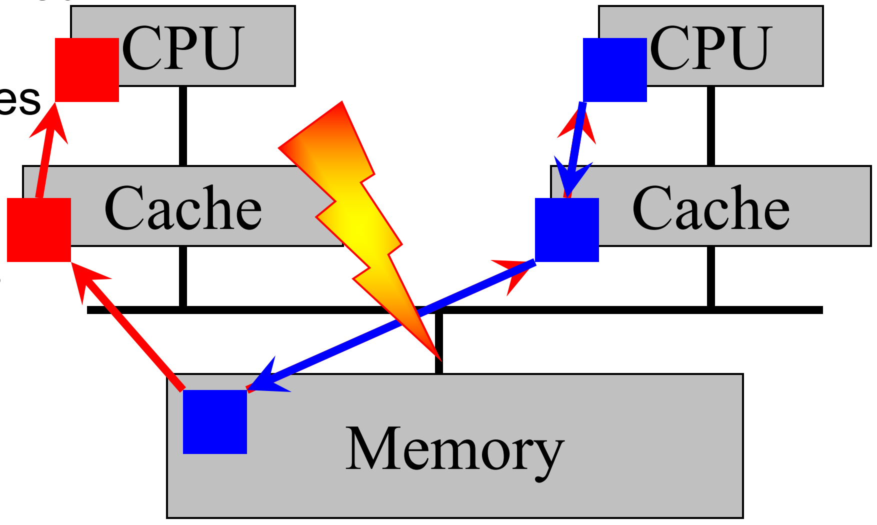
Memory Models

Hardware shared memory means concurrent accesses to shared state
- Multiple hardware threads work independently
- Question: which updates are seen when by which thread?

Two fundamental concepts
- Memory/Cache Coherency: reasoning about updates to one memory location
- Memory Consistency: reasoning about updates to several memory locations

A system is coherent if
- Program order for loads/stores are preserved
- All stores eventually become visible
- All processors see the same order of writes

Establishing coherency
- On cores: implemented within processors
- Cross cores: two main protocol families for cache coherency
  - Snoop-based protocols
  - Directory-based protocols



Memory Consistency

What can we expect when we read from and write to memory

- What is the relative ordering?
- When will updates be available?

Think of memory consistency as a contract between the programmer and the system

Most intuitive for the programmer: **Sequential Consistency**

[Lamport] "A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program"

Reference: Adve, S. V. and K. Gharachorloo, " Shared memory consistency models: A tutorial," IEEE Computer, December 1996.

Consequences of Sequential Consistency

SC is natural from a programmer's perspective

Why is the program order requirement hard?

- Network can reorder
- Use of write-buffers
- Compiler optimizations

Why is the in-sequence/atomicity requirement hard?

- A write has to take place instantly with respect to all processors
- Can lead to a system-wide serialization of writes

We need to relax consistency requirements

- Necessary to achieve well performing systems
  - Increase the level of concurrency
- Compensate with synchronization primitives
  - Need them anyway for proper synchronization of contents
  - Can implicitly execute consistency operations

Create the "illusion" of sequentially consistency

Relaxed Consistency Models

Processor Consistency (should probably be HW Threads Consistency by now)

- Writes by any thread are seen by all threads in the order they were issued
- But: different threads may see a different order

Weak Consistency

- Data operations vs. synchronization operations
- Synchronization operations are sequentially consistent
- When a synchronization operation is issued, the memory pipeline is flushed

Release Consistency

- Further subdivide synchronization operations into “acquire” and “release”
  - Matches the idea of ”acquiring a lock” and “releasing a lock”
- Before accessing a variable, all acquire operations have to be complete
- Before completing a release, all read and write operations have to be complete
- All acquires and releases have to be sequentially consistent

The OpenMP Memory Model

OpenMP uses a relaxed consistency similar to weak consistency

- Temporarily, the view on memory from different threads can be inconsistent
- Memory state is made consistent at particular constructs
- Programmer need to be aware of this and synchronize additionally as needed

Memory synchronization points (or flush points)

- Entry and exit of parallel regions
- Barriers (implicit and explicit)
- Entry and exit of critical regions
- Use of OpenMP runtime locks
- Every task scheduling point (see Segment 19)

However, note the following are NOT synchronization points:

- Entry and exit of work sharing regions
- Entry and exit of masked regions

Implementing Manual Synchronization

```
Thread 1                        Thread 2

a = foo();                      while (flag);
flag = 1;                       b = a;
```

What can go wrong?

- Compiler can reorder memory accesses
- Compiler can keep variables in registers

OpenMP's Flush Directive

```
#pragma omp flush [(list)]
```

Synchronizes data of the executing thread with main memory

- E.g., copies in register or cache
- It does not update implicit copies at other threads

Operates on variables in the given list

- If no list is specified, all shared variables accessible in the region

Semantics:

- Load/stores executed before the flush have to be finished
- Load/stores following the flush are not allowed to be executed early

Implementing Manual Synchronization

Thread 1

```
a = foo();
#pragma omp flush
flag = 1;
#pragma omp flush
```

Thread 2

```
while (flag)
{
   #pragma omp flush
}
#pragma omp flush
b = a;
```

Implementing Manual Synchronization

Thread 1

```
a = foo();
#pragma omp flush(a,flag)
flag = 1;
#pragma omp flush(flag)
```

Thread 2

```
while (flag)
{
  #pragma omp flush(flag)
}
#pragma omp flush(a,flag)
b = a;
```

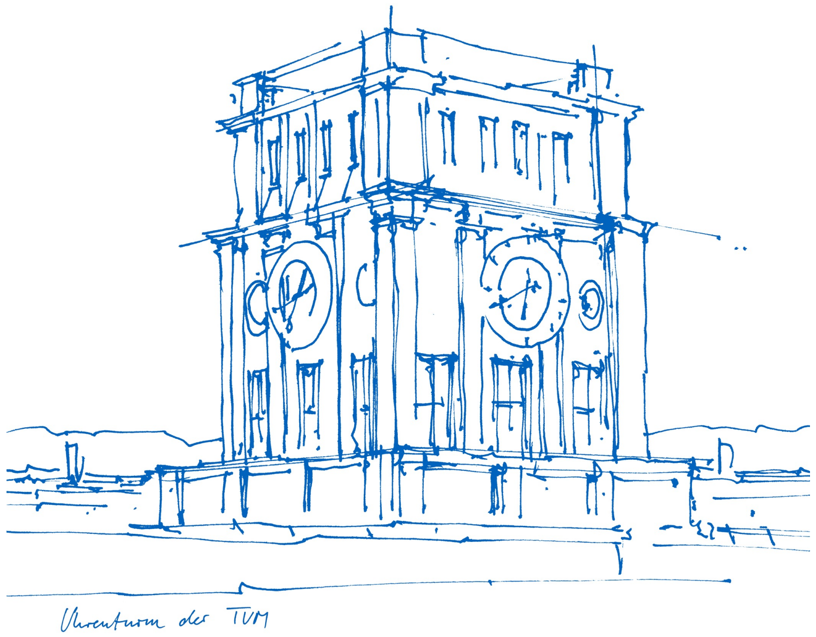
Practical implications

- Necessary for any “manual” synchronization
- Mainly to avoid problems from optimizations
  - Reorderings, register allocations
- Hardware memory model itself in most cases stronger than relaxed consistency
  - Codes may run, but could be incorrect and hence non-portable!
  - Flushes are rarely variable specific (but it could help → performance portability)

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 19
OpenMP Tasking



Drawbacks of Work Sharing

Main concept for parallelism discussed so far: Work Sharing

- Distribute work in for loops to threads
- In many cases: static distribution
- In many cases: equal work distributed

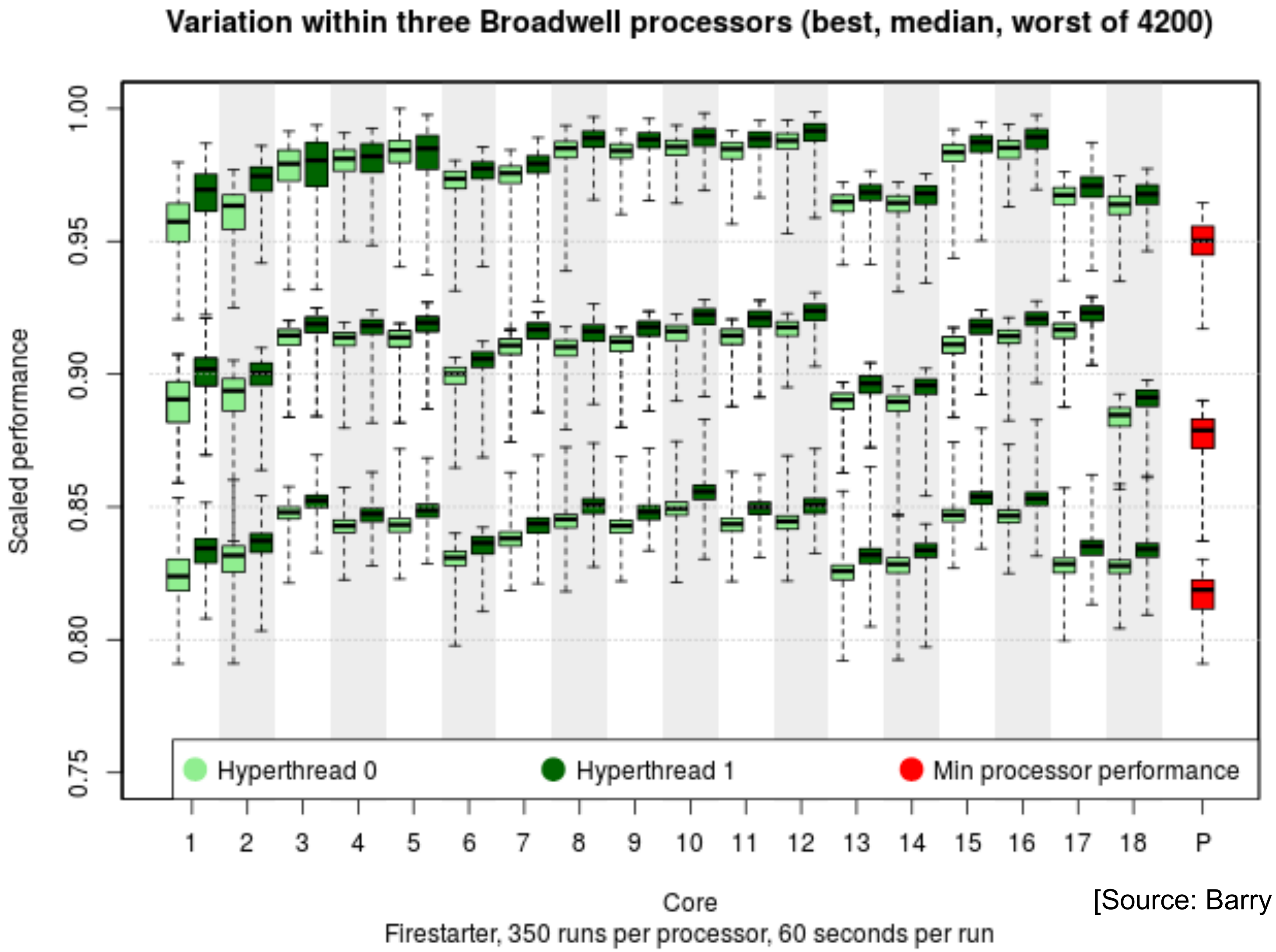
Drawback 1: possible imbalance caused by workload

- Static distribution of equal chunks cannot tolerate load imbalance
- Dynamic distribution can help, but overhead due to fine grained scheduling

Drawback 2: possible imbalance caused by machine

- Differences between nodes, threads, runs, …
- Machines are no longer homogeneous

Processor Variability



[Source: Barry Rountree, LLNL]

Drawbacks of Work Sharing

Main concept for parallelism discussed so far: Work Sharing

- Distribute work in for loops to threads
- In many cases: static distribution (needed for NUMA)
- In many cases: equal work distributed

Drawback 1: possible imbalance caused by workload

- Static distribution of equal chunks cannot tolerate load imbalance
- Dynamic distribution can help, but overhead due to fine grained scheduling

Drawback 2: possible imbalance caused by machine

- Differences between nodes, threads, runs, …
- Machines are no longer homogeneous

Drawback 3: limited programming flexibility

- Limited (in OpenMP) to for loops or coarse grained sections
- Hierarchical parallelism not easy to follow and optimize

Explicit Tasking (since OpenMP 3.0)

Main concept: users create tasks

- Independent pieces of work
- Guaranteed to be executable in any order

OpenMP runtime system schedules the tasks as needed

- Typically maintains one or more task queues
- Tasks distributed to threads
  - Dispatched to open thread as they became available
  - Thread becomes free when task finishes

Tied vs. Untied tasks

- Tied: once a task starts it will remain on the same thread
  - Default behavior
  - Easy to reason about
- Untied: tasks can move to a different thread
  - Execution can be interrupted and the task moved
  - Advantage: more flexibility and better resource utilization

The OpenMP Task Construct

Explicit creation of tasks

```
#pragma omp parallel
{
  #pragma omp single {
  for ( elem = l->first; elem; elem = elem->next)
     #pragma omp task
         process(elem)
  }
// all tasks are complete by this point
}
```

Task scheduling
- Tasks can be executed by any thread in the team

Barrier
- All tasks created in the parallel region have to be finished.

Tasking Syntax in OpenMP

```
#pragma omp task [clause list]
{ ... }
```

Select clauses

`if (scalar-expression)`

- FALSE: Execution starts immediately by the creating thread
- The suspended task may not be resumed until the new task is finished.

untied

- Task is not tied to the thread starting its execution.
- It might be rescheduled to another thread.

`Default (shared|none), private, firstprivate, shared`

- Default is firstprivate.

priority(value)

- Hint to influence order of execution
- Must not be used to rely on task ordering

Example: Tree Traversal

```
struct node {
  struct node *left;
  struct node *right;
};

void traverse( struct node *p ) {
  if (p->left)
    #pragma omp task // p is firstprivate by default
    traverse(p->left);
  if (p->right)
    #pragma omp task // p is firstprivate by default
    traverse(p->right);
  process(p);
}

#pragma omp parallel
#pragma omp single
   traverse(root);
```

Task Wait and Task Yield

```
#pragma omp taskwait
{ ... }
```

Waits for completion of immediate child tasks

– Child tasks: Tasks generated since the beginning of the current task

```
#pragma omp taskyield
{ ... }
```

The taskyield construct specifies that the current task can be suspended

- Explicit task scheduling point

Implicit task scheduling points

– Task creation
– End of a task
– Taskwait
– Barrier synchronization

Task Dependencies (since OpenMP 4.0)

Defines in/out dependencies between tasks

- `Out`: variables produced by this task
- `In`: variables consumed by this task
- `Inout`: variables is both in and out
- Influences scheduling order

Implemented as clause for task construct

```
#pragma omp task depend(dependency-type: list)
{ ... }
```

Example:

```
#pragma omp task shared(x, ...) depend(out: x) // T1
      preprocess_some_data(...);
#pragma omp task shared(x, ...) depend(in: x)  // T2
      do_something_with_data(...);
#pragma omp task shared(x, ...) depend(in: x)  // T3
      do_something_independent_with_data(...);
```

Performance Considerations for Tasking

Advantages

- Implicit load balancing
- Simple programming model
- Many complications and bookkeeping pushed to runtime

Consideration 1: Task granularity

- Fine grained allow for more resource utilization
  - But also cause more overhead
- Coarse grained tasks reduce overhead
  - But also cause schedule fragmentation
- Choosing the right granularity is up to the programmer!

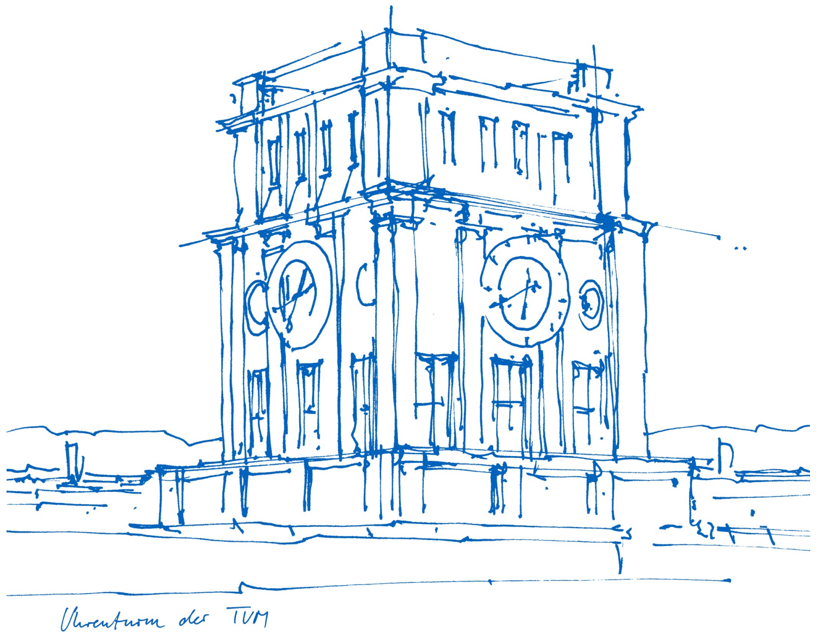
Consideration 2: NUMA optimization

- Each tasks can run anywhere
  - Programmer cannot influence this
- Modern runtimes provide optimizations
  - NUMA aware scheduling

Lecture IN-2147
Parallel Programming

Martin Schulz
Technical University of Munich
Department of Computer Engineering

Segment 20
OpenMP Wrapup



Important Runtime Routines

Runtime offers additional routines to control behavior

Determine the number of threads for parallel regions
- `omp_set_num_threads(count)`

Query the maximum number of threads for team creation
- `numthreads = omp_get_max_threads()`

Query number of threads in the current team
- `numthreads = omp_get_num_threads()`

Query own thread number (0..n-1)
- `iam = omp_get_thread_num()`

Query the number of processors
- `numprocs = omp_get_num_procs()`

Relevant Environment Variables (ICVs)

`OMP_NUM_THREADS=4`

- Number of threads in a team of a parallel region

`OMP_SCHEDULE=”dynamic” OMP_SCHEDULE=”GUIDED,4“`

- Selects scheduling strategy to be applied at runtime
- Schedule clause in the code takes precedence

`OMP_DYNAMIC=TRUE`

- Allow runtime system to determine the number of threads

`OMP_NESTED=TRUE`

- Allow nesting of parallel regions
- If supported by the runtime

Runtime routines to query and set ICVs

Advanced Features

SIMD Directives

– Guest Lecture on SIMD by Dr. Michael Klemm later this semester

Offload to ”other” devises

- Extensive set of primitives for data and synchronization management
- Major usage: GPU Pogramming (towards the end of the semester)

Advanced memory management and placement

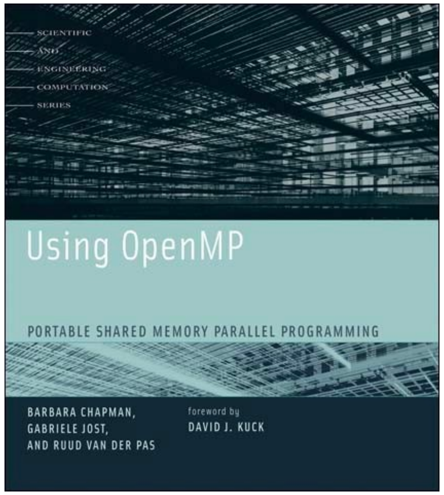
Loop transformation directives

Extensive runtime libraries

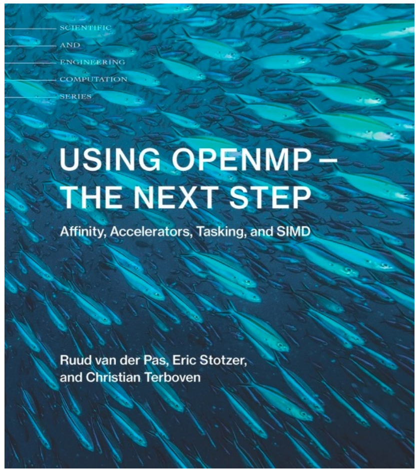
Tool interfaces for debugging and performance analysis

https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

Background Reading on OpenMP (in addition to the standard itself)


SCIENTIFIC AND ENGINEERING COMPUTATION SERIES
Using OpenMP
PORTABLE SHARED MEMORY PARALLEL PROGRAMMING
BARBARA CHAPMAN, GABRIELE JOST, AND RUUD VAN DER PAS
foreword by DAVID J. KUCK

Using OpenMP
Portable Shared Memory Parallel Programming
By Barbara Chapman, Gabriele Jost and Ruud van der Pas
MIT Press


SCIENTIFIC AND ENGINEERING COMPUTATION SERIES
USING OPENMP – THE NEXT STEP
Affinity, Accelerators, Tasking, and SIMD
Ruud van der Pas, Eric Stotzer, and Christian Terboven

Using OpenMP – The Next Step
(OpenMP beyond v2.5)
Ruud van der Pas, Eric Stotzer,
Christian Terboven
MIT Press

DE GRUYTER
GRADUATE
Michael Klemm, Jim Cownie
HIGH PERFORMANCE PARALLEL RUNTIMES
DESIGN AND IMPLEMENTATION

High Performance Parallel Runtimes
Design and Implementation
By Michael Klemm and Jim Cownie
De Gruyter Textbook