Introduction to OpenMP

R. Bader (LRZ)
G. Hager (RRZE)
V. Weinberg (LRZ)
How to build faster computers – a survey

1. Increase **performance / throughput of CPU core**
   a) Reduce cycle time, i.e. increase clock speed (Moore)
   b) Increase throughput, i.e. superscalar + SIMD

2. Improve **data access time**
   a) Increase cache size
   b) Improve main memory access (bandwidth & latency)

3. Use parallel computing (**shared memory**)
   a) Requires shared-memory parallel programming
   b) Shared/separate caches
   c) Possible memory access bottlenecks

4. Use parallel computing (**distributed memory**)
   “Cluster” of computers tightly connected
   a) Almost unlimited scaling of memory and performance
   b) Distributed-memory parallel programming
5. Use an accelerator with your compute node
   a) Requires offload of program regions as well as data (semantics may be limited)
   b) Host and accelerator memory are connected, but separate
      (Improvements are under way)
   c) Programming complexity is higher than for shared memory systems („heterogeneous parallel computing“)
Multi-core processor

It is not a faster CPU – it is a parallel computer on a chip.

Put multiple processors ("cores") on a chip which share resources (example shows a dual core that shares L2 cache and memory bandwidth)

Efficient use of all cores for a single application $\rightarrow$ programmer
… the party is over!

- Option 1 a) is not feasible any more, option 2 only in small increments

By courtesy of D. Vrsalovic, Intel
Paradigms supported by OpenMP – three faces of parallelism

Node Architecture

Threaded Parallelism
(multi-core, shared memory)

Vectorized execution (SIMD)

Offloaded execution (accelerators)

Focus of this course

Also discussed in this course

Not covered in this course

© 2010-19 LRZ/RRZE
Introduction to OpenMP
OpenMP and portability

- **Syntactic portability**
  - Directives / pragmas
  - Conditional compilation permits to mask API calls

- **Semantic portability**
  - Standardized across platforms → safe-to-use interface
  - Unsupported/unavailable hardware features → irrelevant directives will be ignored (you might need a special compiler for your devices …)

- **Performance portability**
  - Unfortunately, performance is not necessarily portable
  - Has traditionally been a problem (partly due to differences in hardware/architectural properties)
  - Becoming worse with recent hardware generations
Compatibility with Sequential Execution

Are semantics for sequential execution retained?
- yes, due to directive concept
- programmer may choose not to

Do memory accesses occur in the same order?
- no, due to relaxed memory consistency (performance feature!)

Are the same numeric results obtained for parallel execution?
- no associativity for model number operations
- parallel execution might reorder operations
  (programmer may need to enforce ordering for reproducibility and/or numeric stability)
OpenMP Standard

- **Responsible body:** OpenMP Architecture Review Board
  - Published OpenMP 5.0 in November 2018
- **Base languages**
  - Fortran (up to 2008)
  - C, C++
  - (Java is not a base language)
- **Resources:**
  - [http://www.openmp.org](http://www.openmp.org) (including standard documents)
  - [http://www.compunity.org](http://www.compunity.org)

- **Note:**
  - LRZ has become a member of the OpenMP ARB in March, 2019
OpenMP history
(courtesy Intel „The Parallel Universe“, issue 18)

Note the increase in the standard's size (OpenMP 5.0 has 666 pages)

Course Target:
Learn the most useful and therefore most commonly used features of OpenMP
Your program ...

Comment lines in source code

Some library routines are exposed to the programmer

Determine resource allocation and assignment, scheduling strategies, etc.

OS threads are executed concurrently on HW cores

© 2010-19 LRZ/RRZE

Introduction to OpenMP
A simple application

Fortran

program
  use m
  implicit none

  call f()

end program

module m
  implicit none
  contains
    subroutine f()
      print *, 'Hello'
    end subroutine
end module

C

#include <stdio.h>

int main() {
  f();
  return 0;
}

void f() {
  printf("Hello\n");
}
Parallel execution model

- **fork-join sequence**
  - can repeat, with differing thread counts
Adding a parallel region

**General form of directives:**

- **Fortran**
  ```fortran
  program
    use m
    implicit none
  !$omp parallel
  call f()
  !$omp end parallel
end program
  ```

- **C**
  ```c
  #include <stdio.h>
  int main() {
  #pragma omp parallel
  {
    f();
  }
  return 0;
  }
  ```

- clauses, if present, modify a directive’s semantics
- multiple clauses per directive are possible
- continuation lines are supported for long directives:
OpenMP structured block rules

**Fortran**
- statements between a beginning and ending directive pair
- GOTO into block is prohibited
- STOP, ERROR STOP

**C / C++**
- delineated by braces following a directive
- setjmp() into block is prohibited
- longjmp() and throw() outside block are prohibited
- exit()
subroutine f()
!$ use omp_lib
 integer :: me
 me = 0
!$ me =omp_get_thread_num()
 print *, 'Hello from thread ', me
end subroutine

#include <stdio.h>
#include <omp.h>
void f() {
    int me = 0;
#ifdef _OPENMP
    me = omp_get_thread_num();
#endif
    printf("Hello from thread %i\n",me);
}
Independent execution contexts

- As many independent function calls as there are threads
- Thread-individual memory management within function call
  - local variables (e.g., "me") are created in the thread-specific stack
  - malloc() or ALLOCATE create memory in the heap separately for each thread
- **Private variables**
  - associated with a particular thread are **inaccessible** by any other thread
  - **pro**: safe to use
  - **con**: communication is not possible (it is needed by many parallel algorithms), unnecessary replication of objects may happen.

- **Thread-individual stack limit**
  - control via environment variable (example: 100 MByte)
  - export `OMP_STACKSIZE=100M`
### OpenMP API

#### Classes of routines:
- Execution environment (36), **Locking (12)**, **Timing (2)**, **Device Memory (7)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Result type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_set_num_threads (int num_threads)</td>
<td>none</td>
<td>number of threads to be created for subsequent parallel region</td>
</tr>
<tr>
<td>omp_get_num_threads()</td>
<td>int</td>
<td>number of threads in currently executing region</td>
</tr>
<tr>
<td>omp_get_max_threads()</td>
<td>int</td>
<td>maximum number of threads that can be created for a subsequent parallel region</td>
</tr>
<tr>
<td>omp_get_thread_num()</td>
<td>int</td>
<td>thread number of calling thread (zero based) in currently executing region</td>
</tr>
<tr>
<td>omp_get_num_procs()</td>
<td>int</td>
<td>number of processors available</td>
</tr>
<tr>
<td>omp_get_wtime()</td>
<td>double</td>
<td>return wall clock time in seconds since some (fixed) time in the past</td>
</tr>
<tr>
<td>omp_get_wtick()</td>
<td>double</td>
<td>resolution of timer in seconds</td>
</tr>
</tbody>
</table>
Compiling and Running

- **Compilation:**
  - Fortran:
    ```
    f90 -fopenmp -o hello.exe hello.f90
    ```
  - C:
    ```
    cc -fopenmp -o hello.exe hello.c
    ```

- **Switch for OpenMP**
  - specific spelling is compiler-dependent
  - toggles both directives and conditional compilation
  - generates threaded code and links against OpenMP run time

- **Execution:**
  ```
  export OMP_NUM_THREADS=4
  ./hello.exe
  ```

- **Output for example program:**
  - Hello from 0
  - Hello from 1
  - Hello from 2
  - Hello from 3

ordering will vary between runs (asynchronous execution)

serial compilation may require stub library

Now: First exercise session
Simple work sharing, Scoping of Data, and Synchronization
Questions that now arise ...

- We know how to set up threading, but
  - how can a large work item be divided up among threads? (using the API for this works in principle, but is tedious)
  - what happens with objects that already exist before the parallel region starts?

- Example:
  - matrix-vector multiplication \( r = M \cdot x \) i.e. \( r_i = \sum_{j=1}^{n} M_{ij}x_j \)

\[
\begin{align*}
M & \rightarrow M \times 1 \rightarrow r
\end{align*}
\]

A bunch of scalar products
Concept of work sharing

- The idea is to split the work among threads

- Note that
  - all elements of $\mathbf{x}$ must be available to all threads
  - Matrix-Vector is often deployed iteratively $\Rightarrow r$ becomes $\mathbf{x}$ in the next iteration $\Rightarrow$ copying of data must be possible

- Consequence:
  - need for variables that are accessible to all threads $\Rightarrow$ "data sharing" is often a prerequisite for "work sharing"
  - a natural concept for a shared memory programming model
Sharing variables across threads

The "shared" clause
- implies that scalar s and array a both are accessible to all threads

Rules for concurrent accesses to a single object
- reads/writes or writes/writes by different threads are not permitted ("data races")

Note: updates to array a are OK because disjoint parts of object are updated

real :: s, a(200) Fortran

s = ...
$omp parallel shared(s,a)
select case (me)
case (0)
a(1:100) = ... * s
case (1)
a(101:200) = ... * (-s)
end select
$omp end end parallel
Data dependencies that prevent parallelization

- **Flow dependency ("read after write", RAW):**

  - $a = \ldots$
  - $b = a$
  - $c = b$

  Second instruction cannot execute concurrently with first.

- **Anti-dependency ("write after read", WAR):**

  - $b = a$
  - $a = \ldots$
  - ... = $a + \ldots$
  - $b = a$
  - $a^2 = \ldots$
  - ... = $a^2 + \ldots$

  Resolvable at cost of introducing a new variable ("name dependency").

- **Output dependency ("write after write", WAW):**

  - $a = \ldots$
  - $b = a$
  - $a = \ldots$
  - $b = a$
  - $a^2 = \ldots$

  After name dependency resolution, statements 1 and 3 can execute concurrently. Flow dependency remains.
Privatization

C

```c
a[k] = ...;
#pragma omp parallel \
    shared(a)
{ int i; float s;
  s = 0.0;
  for (i=...;i<...;i++) {
    s += a[i];
  }
}
```

example calculates thread-individual sums

useless, from a practical point of view. But bear with me - we'll fix this, eventually

Block-local variables in C/C++
- are automatically private

Note: One can expect the same behaviour for the Fortran 2008 BLOCK construct, but this is currently not specified in the OpenMP standard
Privatization with masking

### Implementation Example

```fortran
real :: s
real :: a(:)
integer :: i

s = ...
!
omp parallel private(s) &
!omp shared(a)

s = 0.0
do i = ..., ...
    s = s + a(i)
end do
!
omp end parallel

... = ... + s
```

- **Masking occurs**
  - for privatized variables declared outside the parallel region
- **Loop variables**
  - are always private

---

**Diagram Explanation**

- **Shared Variables**
  - Persist across parallel regions
  - Example: `a`

- **Private Variables**
  - Local to each thread
  - Example: `s`

**Execution Sequence**

- **Fork:**
  - Threads `T_0`, `T_1`, `T_2`, `T_3`

- **Join:**
  - Variables become undefined

---

- **Warning:**
  - If `s` were shared, the program would have a race condition.
  
  Would expect value from before parallel region to persist, but side effects are possible. For example, modification via a pointer (avoid this!).

---

© 2010-19 LRZ/RRZE

Introduction to OpenMP
Code for work-shared Matrix-Vector multiplication: The DO / FOR directive

### Serial

**Fortran**

```fortran
DO k = 1, n
  DO j = 1, n
    r(j) = r(j) + a(j, k) * x(k)
  END DO
END DO
```

**C**

```c
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r[j] = r[j] + a[k*n+j] * x[k];
  }
}
```

### OpenMP parallel

**Fortran**

```fortran
!$omp parallel
!$omp do
DO j = 1, n
  DO k = 1, n
    r(j) = r(j) + a(j, k) * x(k)
  END DO
END DO
!$omp end do
... = r(...)
!$omp end parallel
```

**C**

```c
#pragma omp parallel
{
  #pragma omp for
  for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
      r[j] = r[j] + a[k*n+j] * x[k];
    }
  }

  ... = r[...];
}
```

- `r, a, x` are shared by default
- `j, k` are private
- `r[j]` at line 9 is updated only by thread `j`
- `r[j]` at line 12 is updated by all threads
- `r[j]` at line 16 is updated by all threads

---

© 2010-19 LRZ/RRZE

Introduction to OpenMP
Further rules for work shared loops

- **Slicing of iteration space**
  - „loop scheduling“
  - default behaviour is implementation dependent
  - usually as equal as possible chunks of largest possible size, one chunk per thread

- **In the example,**
  - slicing is done as shown some slides earlier
  - loop order was switched to avoid having many synchronizations

- **Additional clauses**
  - on OMP DO / omp for will be discussed later

- **Restrictions on loop structure**
  - Trip count must be computable at entry to loop
  - **Disallow:**
    - C style loops modifying the loop variable in the loop body, or using a non-evaluable exit condition, or Fortran DO WHILE loop;
  - loop body must be a well-formed structured block with single entry and single exit point

- **Note:**
  - directive (by default) acts only on outermost enclosed loop

- Actually, we're caught between a rock and a hard place here...
Avoiding race conditions (1): mutual exclusion via the critical directive

**Fortran**

```fortran
real :: s, stot
real :: a(:)
integer :: i

stot = 0.0
!$omp parallel private(s) &
!$omp shared(a, stot)
  s = 0.0
!$omp do
do i = 1, size(a)
  s = s + a(i)
end do
!$omp end do
!$omp critical
  stot = stot + s
!$omp end critical
!$omp end parallel
```

- **Only one thread at a time can execute a critical region**
  - others must wait → code in region is **effectively serialized**
Dealing with race conditions through atomic updates

Properties of atomic operations
- the atomic directive applies only for a single update to a scalar shared variable of intrinsic type
- this way of updating can be done safely when executed concurrently (exception to the rules on race conditions!)
- otherwise, no synchronising effect imposed by semantics
- hardware atomic instructions available → likely more efficient than critical region

parallel array summation

C can use #pragma omp critical
Fortran can use !$omp atomic ...

float stot;
stot = 0.0;
#pragma omp parallel \
  shared(a,stot)
{
  int i; float s;
  s = 0.0;
#pragma omp for
  for (i=0;i<N;i++) {
    s += a[i];
  }
#pragma omp atomic update
  stot += s;
}

Introduction to OpenMP
The two kinds of memory in OpenMP

- Data accessed by can be shared or private
  - shared data – one instance of an entity available to all threads (in principle)
  - private data – each per-thread copy only available to thread that owns it

- Data transfer transparent to programmer

- Synchronization necessary for accessing shared data from different threads to avoid race conditions
  - implicit barrier
  - explicit directive
The firstprivate clause

real :: s

s = ...
!$omp parallel &
!$omp firstprivate(s)

... = ... + s

s = ...
!$omp end parallel
... = ... + s

• Extension of private:
  • value of master copy is transferred to private variables
  • restrictions: not a pointer, not assumed shape, not a subobject, master copy not itself private etc.
**The lastprivate clause**

```
real :: s
!
omp parallel
!omp do lastprivate(s)
doi = 1, n
    s = ...
end do
!omp end do
... = ... + s
!omp end parallel
```

- **When to use?**
  - as little as possible
  - legacy code

- **Extension of private:**
  - value from thread which executes last update in the serial code is transferred back to master copy
  - restrictions similar to `firstprivate`

---

© 2010-19 LRZ/RRZE

Introduction to OpenMP
Scoping clauses can be specified for
- parallel regions
- loop work sharing constructs

Defaults
- apply if no clause is specified
- may vary by construct, but for the above the following apply:
  - pre-existing objects are by default `shared`, except for loop variables, which are `private`.
  - objects declared inside the lexical or dynamic scope of the construct are `private`.

Recommendation:
- specify a `default(none)` clause on each directive that permits scoping:

  ```fortran
  !$omp parallel default(none) &
  !$omp shared(...) private(...) ...
  ...
  ```

  ```c
  #pragma omp parallel default(none) \
  shared(...) private(...) ...
  ...
  ```

  - this `forces` you to explicitly consider and specify scoping for all pre-existing objects

Now: Second exercise session
Reductions
Concept of Reduction

- **Seen in previous exercise:**
  - need for assembling partial results across threads
  - up to now: with critical region

- **OpenMP reductions:**
  - sometimes more efficient at scale
  - implementation tunings like
    
    reduce complexity from $O(n_{\text{threads}})$ to $O(\log_2(n_{\text{threads}}))$
  - always easier to understand and maintain

\[ s_i = s_i + \ldots \text{ on each thread} \]

\[ \text{want } \sum_i s_i \text{ here (not directly possible because } s \text{ is private)} \]

\[ \text{new concept is needed ...} \]
Example 1: Sum reduction in a parallel region

- **value of s after end of parallel region:** $s_{\text{incoming}} + \sum_i s_i$

**Note: multiple reductions are permitted**
Example 2: Sum reduction in a work shared region

```fortran
real :: s
s = 2.2
!$omp parallel shared(s)
    ... 
!$omp do reduction(:+s)
    do i = 1, n
        ... 
        s = s + ... 
    end do
!$omp end do
!$omp end parallel
... = ... * s

```

```c
float s;
    s = 2.2;
    #pragma omp parallel shared(s)
    {
        ...
        #pragma omp for reduction(:+s)
        for (i=0, i<n, i++) {
            ... 
            s += ... 
        }
    } 
... *= s;

```

- Value of `s` after end of worksharing region: $s_{\text{incoming}} + \sum_i s_i$
### Initial value of private reduction variables

- **Depends on operation**
- **Supported intrinsic operations:**

#### Fortran

<table>
<thead>
<tr>
<th>Operation</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>.and.</td>
<td>.true.</td>
</tr>
<tr>
<td>.or.</td>
<td>.false.</td>
</tr>
<tr>
<td>.eqv.</td>
<td>.true.</td>
</tr>
<tr>
<td>.neqv.</td>
<td>.false.</td>
</tr>
<tr>
<td>MAX</td>
<td>-HUGE(X)</td>
</tr>
<tr>
<td>MIN</td>
<td>HUGE(X)</td>
</tr>
<tr>
<td>IAND</td>
<td>all bits set</td>
</tr>
<tr>
<td>IEOR</td>
<td>all bits 0</td>
</tr>
<tr>
<td>IOR</td>
<td>all bits 0</td>
</tr>
</tbody>
</table>

#### C / C++

<table>
<thead>
<tr>
<th>Operation</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>&amp;</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td>smallest representable value</td>
</tr>
<tr>
<td>MIN</td>
<td>largest representable value</td>
</tr>
</tbody>
</table>
Array reductions

**Example**
- reduces complete array `b` and `m` elements of array `a`, elementwise
- uses regular Fortran array section notation

```
real :: a(*)
real :: b(n)

!$omp parallel reduction(+:b) &
!$omp reduction(*:a(1:m))
...
```

```
float *a;
float b[N];
#pragma omp parallel 
    reduction(+:b[:]) 
    reduction(*:a[0:m])
...
```

**General rules:**
- array section must be a **contiguous** object (→ no strides permitted)
- dynamic objects must be associated / allocated, and the status must not be modified for the private copies

```
[lower bound : upper bound]
```

```
[lower bound : length]
```

now also supported in C/C++!
User-defined reductions

- Using derived types

```fortran
type :: fraction
  integer :: numerator, denominator
end type
```

```c
typedef struct {
  int numerator, denominator;
} Fraction;
```

Add overloaded operators +, -, * etc. or even user-defined operators.

- And now we want to write

```fortran
type(fraction) :: af
af = ...
!$omp parallel reduction(+:af)
  ...
  af = af + ...
!$omp end parallel
```

```c
Fraction af;
af = ...;
#pragma omp parallel 
  reduction(+:af)
  {
    ...
    Fraction_sum(af, ...);
  }
```

Provide functions to add, etc.

- but the compiler will refuse to build it („+“ not known to OpenMP) unless further measures are taken ...
Declaring a user-defined reduction

---

### Combiner
- connects to operator implementation

**Fortran:** example defers to overloaded `+`, **C:** references `Fraction_add`

special OpenMP parameters `omp_in, omp_out` formally describe the two operands for each operation needed

### Initializer
- implements initial value setting for private copies

**Fortran:** uses (overloaded) structure constructor, **C** similar

special OpenMP parameter `omp_priv` formally describes private copy
More on Work Sharing

Loops and loop scheduling
Collapsing loop nests
Parallel sections
The schedule clause

**Default scheduling:**
- implementation dependent
- **typical:** largest possible chunks of as-equal-as-possible size ("static scheduling")

**User-defined scheduling:**

```
Fortran
!$OMP do schedule( static, [chunk] )
```

chunk: always a non-negative integer. If omitted, has a schedule dependent default value

1. **Static scheduling**
   - `schedule(static,10)`
   - minimal overhead (precalculate work assignment)
   - default chunk value: see left

2. **Dynamic scheduling**
   - after a thread has completed a chunk, it is assigned a new one, until no chunks are left
   - `schedule(dynamic,10)`
   - synchronization overhead
   - default chunk value is 1
3. Guided scheduling

- **Size of chunks in dynamic schedule**
  - too small → large overhead
  - too large → load imbalance

- **Guided scheduling**: *dynamically* vary chunk size.
  - Size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to chunk-size. (default: → 1)

- **Chunk size**:  
  - means minimum chunk size (except perhaps final chunk)
  - default value is 1

- both dynamic and guided scheduling are useful for handling **poorly balanced and unpredictable** workloads.
OpenMP Scheduling of simple for loops

**OMP_SCHEDULE=static**

- Chart 1: OMP_SCHEDULE=static,10

**OMP_SCHEDULE=dynamic,10**

- Chart 2: OMP_SCHEDULE=dynamic,10, out

**OMP_SCHEDULE=static,10**

- Chart 3: OMP_SCHEDULE=static,10 out

**OMP_SCHEDULE=guided,10**

- Chart 4: OMP_SCHEDULE=guided,10 out
4. Deferred scheduling

- **Decided at run time:**
  - auto (automatic scheduling)
    - programmer gives implementation the freedom to use any possible mapping.
  - runtime
    - schedule is one of the above or the previous two slides
    - determine by either setting OMP_SCHEDULE, and/or calling omp_set_schedule()
      (overrides env. setting)
    - find which is active by calling omp_get_schedule()

- **Examples:**
  - environment setting:
    ```
    export OMP_SCHEDULE='guided'
    export OMP_NUM_THREADS=4
    ./myprog.exe
    ```
  - call to API routine:
    ```
    omp_set_schedule(
      omp_sched_dynamic,4);
    #pragma omp parallel
    {
      #pragma omp for schedule(runtime)
      for (...) {
        ...
      }
    }
    ```

Final remarks on scheduling

- Please check your compiler documentation for implementation-dependent aspects

- An implementation may add its own scheduling algorithms
  - code using specific scheduling may be at a disadvantage
  - **recommendation:** Allow changing of schedule during execution

- If runtime scheduling is chosen and OMP_SCHEDULE is not set
  - execution starts with implementation-defined setting
Collapsing loop nests

Example: Two nested loops

```fortran
!$OMP do
  do k=1, kmax
    do j=1, jmax
      :
      end do
    end do
  end do
!$OMP end do
```

- assume kmax is 2, and jmax is 3
- then the workshared loop will scale to at most 2 threads

Therapy:

- use a collapse clause to improve scaling
- this flattens two (or more) loop nests into a single iteration space

Improved example:

```fortran
!$OMP do collapse(2)
  do k=1, kmax
    do j=1, jmax
      :
      end do
    end do
  end do
!$OMP end do
```

- slicing is performed on the virtual index $I_{coll}$:

```
<table>
<thead>
<tr>
<th>$I_{coll}$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>K</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

Restrictions:

- rectangular iteration space
- CYCLE/continue in innermost loop only
Collecting load imbalances at synchronization points

**Example:**

```
!$omp parallel
!$omp do reduction(+:tsum)
   do k=1, kmax
      tsum = tsum + foo(a, b, c)
   end do
!$omp end do
!$omp end parallel
```

**Assumptions** on code following the synchronization point:
- does not involve `tsum`
- has a load imbalance that is inverse to that of preceding code block
### nowait clause and explicit barrier directive

- **Reduce load imbalance**
  - by removing the barrier via the `nowait` clause

- **Assure code correctness**
  - may require explicit barrier directive before `tsum` (or other modified shared variable) is accessed

```fortran
!$omp parallel
!$omp do reduction(+:tsum)
    do k=1, kmax
        tsum = tsum + foo(a, b, c)
    end do
!$omp end do nowait
...
!$omp barrier
... = tsum ...
!$omp end parallel
```

- `nowait` clause prevents waiting in the barrier.
- `barrier` directive ensures code correctness.

---

**Diagram:**
- 

---

```c
#pragma omp for reduction(+:tsum) \
    nowait
{ ... }
```
Parallel sections

- Non-iterative work-sharing construct
  - distribute a static set of structured blocks

```fortran
!$OMP sections
!$OMP section
  ...
!$OMP section
  ...
!$OMP section
  ...
!$OMP end sections
```

- each block is executed **exactly once** by one of the threads in the team

- **Allowed clauses on sections:**
  - private, first/lastprivate, reduction, nowait

```c
#pragma omp sections
#pragma omp section
  {  
  
  }
#pragma omp section
  {  
  
  }

#pragma omp section
  {  
  
  }

// end sections
```
Parallel sections cont'd

- **Restrictions:**
  - *section* directive must be within lexical scope of *sections* directive, and directly enclosed (no interleaved language construct is permitted)

  - *sections* directive binds to innermost enclosing parallel region → only the threads executing the binding parallel region participate in the execution of the section blocks and the implicit barrier (if not eliminated with nowait)

- **Scheduling to threads**
  - implementation-dependent
  - if there are more threads than code blocks, excess threads wait at synchronization point

- **In modern OpenMP,**
  - *tasking* provides a much more flexible and scalable way to implement this and much more general patterns → will be treated tomorrow
**Execution:**
- only one thread of the team executes the statements in the block
- others go to the end of the block

**Synchronization**
- of all threads at end of **single** block
single directive syntax

- **Note:**
  - update of shared variables inside a single block is safe against subsequent accesses, due to synchronization at the end of that block.
Work sharing with single: the nowait clause

- Implement a self-written work scheduler
  - one possible scheme (of many), sketched only:

```fortran
... !$omp parallel
  do iw=1, nwork
  !$omp single
    ...
  !$omp end single nowait
  ...
  !$omp barrier
  end do !iw
  !$omp end parallel
```

- not the most efficient method
  → preferably use tasking (covered tomorrow); the single construct will be relevant in that context
Global variables and threading
Global variables and their default scope

- **Examples:**

```fortran
module myGlobals
    implicit none
    integer :: my_count
    real, allocatable :: a(:)
    ...
end module
```

- **Such variables by default have shared scope**
- **The same applies for variables with the SAVE (Fortran) or static (C) attribute**

**Imagination:**
- code using such memory is often **not thread-safe**, unless mutual exclusion mechanisms are used when accessing the objects
Privatizing global objects

- When program semantics requires that each thread work on its own copy, privatization is necessary
  - not exactly the same as private variables → separate syntax needed

- C:
  - #pragma omp threadprivate(list)
  - list is a comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types

- Fortran:
  - !$omp threadprivate(list)
  - list is a comma-separated list of named variables and named common blocks. Common block names must appear between slashes.

- Objects start out with master copy existing only
  - thread-private copies (with undefined values) spring into existence when the first parallel region is started
Further properties of threadprivate storage

- **Copyin clause**
  - broadcasts object values from master copy to thread-individual copies
  - works analogous to the firstprivate clause

```fortran
allocate( a(ndim) )
a(:) = ...
!$omp parallel copyin(a)
... = a(i) + ...
a(i) = ...
!$omp end parallel
```

- **Subsequent parallel regions:**
  - thread-individual copies retain their values (by thread) if
    1. second parallel region not nested inside first
    2. same number of threads is used
    3. no dynamic threading is used

**Note:** none of the potential violations of the above three rules are dealt with in this course

**Recommendations:**
- Avoid using global variables in the context of threading
- Use object-based design instead
... useful varia
The master construct

- **Only thread zero (from the current team) executes the enclosed code block**
  - there is **no implied barrier** either on entry to, or exit from, the master construct. Other threads continue **without synchronization**

- **Notes:**
  - Not all threads must reach the construct; if the master thread does not reach it, it will not be executed at all
  - this is not a work sharing construct, it only serves for execution control

```fortran
 !$omp master
 block
 !$omp end master
```

```c
#pragma omp master
{
 block
}
```
Combined constructs

- Certain combinations of constructs can be fused
  - the result is a single construct that behaves as if the two individual ones were tightly nested
  - may be more efficient due to reduced synchronization needs
  - is often easier to read
- Example: joint "parallel do" (C has "parallel for" here ...)

```
!$omp parallel
!$omp do
  do i=1, n
    ...
  end do
!$omp end do
!$omp end parallel

!$omp parallel do
  do i=1, n
    ...
  end do
!$omp end parallel do
```

- both variants have the same semantics
Conditional parallelism

- Put an "if" clause on a parallel region

\[
\text{Fortran:}
\begin{align*}
&!\text{omp parallel if (n > 8000)} \quad \ldots \quad !\text{omp end parallel} \\
&\text{process work item of size } O(n^p)
\end{align*}
\]

- specify a scalar logical argument
- may require manual tuning for properly dealing with thread count dependency etc.

- Specific uses:
  1. execute serially for small problem sizes (parallel overhead may reduce performance)
  2. suppress nested parallelism in a library routine:

\[
\begin{align*}
&\text{Fortran:}
&\#pragma \text{omp parallel if} \quad ( ! \text{omp_in_parallel() } ) \\
&\{ \quad \ldots \quad \}
\end{align*}
\]

Now: Third exercise session
OpenMP 4.0
SIMD (vectorization) directives

Optimization of innermost loop structures

Acknowledgment is due to M. Klemm (Intel)
SIMD - single instruction multiple data

- **Example:**
  - Sandy Bridge vector unit
  - 256 Bit SIMD
  - addition of 8 Byte words

- **Instruction capability**
  - 1 vector add and 1 vector mult per cycle → theoretical Peak 8 Flops/cycle (double precision)

- **LD/ST issue capability for Sandy Bridge**
  - 4 Words LD/cycle
  - 4 Words ST/(2 cycles)
  - performance boost depends on algorithm, including its temporal locality properties

- **More recent processors may have more advanced units**
  - more SIMD lanes
  - additional vector operations

Example:
- Sandy Bridge vector unit
- 256 Bit SIMD
- addition of 8 Byte words

Instruction capability
- 1 vector add and 1 vector mult per cycle → theoretical Peak 8 Flops/cycle (double precision)

LD/ST issue capability for Sandy Bridge
- 4 Words LD/cycle
- 4 Words ST/(2 cycles)
- performance boost depends on algorithm, including its temporal locality properties

More recent processors may have more advanced units
- more SIMD lanes
- additional vector operations
Before OpenMP 4.0 …

- … programmers had to rely on auto-vectorization,
  - or use **non-portable** extensions
    - programming models (e.g. Intel Cilk Plus)
    - intrinsics (e.g. `_mm_add_pd()`)  
    - compiler pragmas

```c
#pragma omp parallel for
#pragma vector always
#pragma ivdep
for (int i=0; i<N; i++) {
    a[i] = b[i] + …;
}
```

which may or may not get ignored by the compiler
OpenMP SIMD loop construct

- Vectorize a loop nest
  - cut into chunks that fit into a SIMD vector register
  - without parallelization of the loop body

- Syntax

  ```c
  #pragma omp simd [clause[[], clause], …]
  for loops
  ```

  ```fortran
  !$omp simd [clause[[], clause], …]
  do loops
  !$omp end simd
  ```
Simple example

- **Scalar product**

```c
void sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp simd reduction(+:sum)
    for (int k=0; k<n; k++) {
        sum += a[k] * b[k];
    }
}
```

- **Converts serial element-wise execution**

  to vectorized one:

  [Diagram showing vectorization process with architecture-specific vector length]
Data Sharing Clauses

▪ Existing ones adapted to SIMD-style execution
  • required for more complex loop bodies

▪ `private (var-list)`
  create uninitialized vectors for variables in var-list
  (loop iteration variables are private by default)

▪ `lastprivate (var-list)`
  copy last iteration value to variable at the end of the construct

▪ `reduction (op:var-list)`
  create private copies for variables in var-list and apply the reduction operation \( op \) at the end of the construct
Loop clauses (1)

- **safelen (length)**
  - maximum distance between iterations that can run concurrently without breaking any dependencies

- **linear (list[:linear-step])**
  - produce private copy of a variable that is in linear relationship with the loop iteration variable: \( x_i = x_{start} + (i - i_{start}) \times \text{linear-step} \)

```c
#pragma omp simd safelen(5)
for (int k=j; k<n; k++) {
    b[k] = a[k] * b[k-j];
}
```

- programmer assures \( j > 5 \)
- compiler can use a vector length of at most 6
- **aligned (list[:alignment])**
  - specifies that variables in the list are aligned, either by the specified integer value of alignment in units of bytes, or in implementation-specific manner

- **collapse(n)**
  - collapse iteration space of a SIMD loop nest
Parallelize and vectorize a loop nest

- distribute iteration space of loops across threads
- subdivide loop chunks to be processed in SIMD registers

Syntax

**C**

```c
#pragma omp for simd [clause[[], clause], ...]
```

**Fortran**

```fortran
!$omp do simd [clause[[], clause], ...]
```

```fortran
[!$omp end do simd]
```
void sprod(float *a, float *b, int n) {
    float sum = 0.0f;
    #pragma omp for simd reduction(+:sum)
    for (int k=0; k<n; k++) {
        sum += a[k] * b[k];
    }

assume invocation by all threads executing in a parallel region
Function vectorization

- **Function call inside SIMD region**

```c
float min(float a, float b) {
    return a < b ? a : b;
}

float distsq(float x, float y) {
    return (x - y)*(x - y);
}

void example() {
    #pragma omp for simd
    for (i=0; i<N; i++) {
        d[i] = min(
            distsq( a[i], b[i] ), c[i] );
    }
}
```

- **Therapy: explicitly declare for use in vectorized loops**
  - C/C++ syntax

```c
#pragma omp declare simd
function def. or decl.
```

  - Fortran syntax

```fortran
!$omp declare simd &
!$omp (proc-name-list)
```

- clauses are also supported
- causes generation of multi-version code by the compiler
vectorized versions of generated functions are shown

```c
#pragma omp declare simd
defloat min(float a, float b) {
  return a < b ? a : b;
}

#pragma omp declare simd
defloat distsq(float x, float y) {
  return (x - y)*(x - y);
}

devoid example() {
  #pragma omp for simd
  for (i=0; i<N; i++) {
    d[i] = min(
      distsq( a[i], b[i] ), c[i] );
  }
}
```

```c
vec8 min_v(vec8 a, vec8 b) {
  return a < b ? a : b;
}

vec8 distsq_v(vec8 x, vec8 y) {
  return (x - y)*(x - y);
}
```

```c
vec8 vd = min_v(
  distsq_v (va, vb), vc );
```

no SIMD directives permitted inside vectorized functions!
Clauses applicable for declare simd

- **simdlen (length)**
  generate function to support supplied vector length

- **uniform (argument-list)**
  argument has a constant value between iterations of invoking loop

- **inbranch vs. notinbranch**
  function always / never called from inside an if statement

- **linear (list[:linear-step])**
- **aligned (list[:alignment])**
- **reduction (op:var-list)**

© 2010-19 LRZ/RRZE
Final remarks on SIMD

- Case studies on vectorizable applications:
  - show performance improvements of factor 1.5 – 4.3 compared to auto-vectorized code
  - you may not be as successful, but a 20% performance improvement for 45 min optimization work is also quite nice

- Resolution of dependencies
  - may sometimes involve code restructuring and splitting of loops

- Further features available: combination of device control directives with SIMD
  - platform dependence
  - not discussed in this talk

Now: Fourth exercise session
More on Synchronization and Correctness

Memory model
Identifying correctness problems
Named critical regions
Atomic operations
Loop dependencies
Mutual exclusion with locks
Concurrent updates on shared variables

**Scenario:**

```fortran
real :: a

a = 0
!$omp parallel shared(a) num_threads(2)
    a = a + 1
    write(*,'("a on thread ",i0," is ",i0)') &
        omp_get_thread_num(), a
!$omp end parallel
write(*,'("a after construct is ",i0)') a
```

- the above is **non-conforming**
- data race causes **unpredictable** results to be produced

**Reason:**

- different threads can have different views on same variable: temporary view (in-register value) vs. memory value
- these two views become inconsistent when a thread modifies the variable

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

possible results in first `write`

possible results in second `write`: 1 or 2
Memory consistency rules

- **Flush Operation**
  - is performed on a set of (shared) variables or on the whole thread-visible data state of a program
  - **discards** temporary view:
    → modified values are forced to cache/memory (requires exclusive ownership)
    → next read access must be from cache/memory
  - **further** memory operations only allowed after all involved threads complete flush:
    → restrictions on memory instruction reordering (by compiler)

- **Ensure consistent view of memory:**
  - Assumption: want to write a data item with one thread, read it with another one
  - Order of execution **required**:
    1. thread 0 writes to shared variable
    2. thread 0 flushes variable
    3. thread 1 flushes same variable
    4. thread 1 reads variable
  - The challenge is to assure step 3 happens **after** step 2
  - OpenMP construct synchronization semantics assure this as well as the necessary implicit flush operations (if correctly used)

```c
!$omp flush [list]
```

recommend to **avoid** use of explicit flushes
But it is possible to make mistakes ...

- **Example: update via critical region**
  - mutual exclusion is only assured for the statements *inside* the block
    i.e., subsequent threads executing the block are synchronized against each other

- **If other statements access the shared variable, you may be in trouble:**

```c
!$omp parallel shared(x) ...
:
!$omp critical
    x = x + y
!$omp end critical
...
    a = f(x, ...
!$omp end parallel
```

Race on read to `x`. Most likely, a barrier is required **before** this statement to assure that all threads have executed their mutexed updates
Using Intel Inspector on x86-based systems

- OpenMP correctness analysis:
  - no special compiler option needed (except perhaps –g)
  - GUI also for Linux-based system
- Identify memory issues in addition to threading issues
  - leaks, dangling pointers etc.
- Start up GUI
  - prerequisites: set up environment and possibly stack limit
  - then, invoke the GUI with `inspxe-gui`
  - command line `inspxe-cl` is also available, but will not be discussed in this talk
Starting up the GUI → start a new project

Enter project name then press „create project“
Configure the project

- **Needed information:**
  - executable name (must have been built with OpenMP)
  - executable path (autocompleted)
  - arguments if needed by executable

- **Further advanced settings are possible**
Select analysis mode, then start
- here: Threading Error Analysis → locate deadlocks and data races
- note potentially high performance impact
Error indication by severity

Note:
- Requires debug option for compiled code

A race condition was identified
Source window: conflicting reads/places

Introduction to OpenMP
Critical regions: consider multiple updates

a) **same** shared variable

```
subroutine foo()
  !$omp critical
  x = x + y
  !$omp end critical

subroutine bar()
  !$omp critical
  x = x + z
  !$omp end critical
```

critical region is **global** $\rightarrow$ OK

b) **different** shared variables

```
subroutine foo()
  !$omp critical
  x = x + y
  !$omp end critical

subroutine bar()
  !$omp critical
  w = w + z
  !$omp end critical
```

mutual exclusion not required $\rightarrow$ unnecessary loss of performance
Named critical regions

- Solution:
  - use a **named** critical

```fortran
subroutine foo()
!$omp critical (foo_x)
  x = x + y
!$omp end critical (foo_x)
subroutine bar()
!$omp critical (foo_w)
  w = w + z
!$omp end critical (foo_w)
```

mutual exclusion only if same name is used for critical regions acting on different code blocks

- **Note:** The atomic directive is bound to the updated variable
  → problem does not occur when such a directive is used.
More variants of atomic operations

**Assumption:**
- v, w private or shared scalar variables
- x a shared scalar variable

**Atomic read:**

```c
#pragma omp atomic read
v = x;
```

**Atomic write:**

```c
#pragma omp atomic write
x = v;
```

**Atomic capture**

```c
!$omp atomic capture
v = x
x = x <op> w
!$omp end atomic
```

**Not atomic:**
- evaluation of expressions or updates on v

**Atomic update:**
- !$omp atomic update
- same as „traditional“ atomic directive
Atomic operations require care

**Atomic directives**
- permit the programmer to explicitly program with race conditions

**Rationale for use:**
- performance
- tailored synchronizations → will usually require explicit flush operations (not discussed)

**Programmer's responsibility**
- to assure that no inconsistencies result → must evaluate results from all possible interleavings of execution by different threads
- tools might not be able to observe problems

**Synchronization effect**
- apart from the value change on the variable itself being visible, no synchronization is done
- **sequentially consistent atomic operations**:
  ```c
  #pragma omp atomic \ seq_cst update 
  x = x + v;
  ```
  perform a flush on all thread-visible variables (but no synchronization otherwise). Semantics are the same as for such operations in the C++11 standard
Dealing with dependencies in a loop

- **Variant of theoretical exercise**
  - race condition \( i-1 \rightarrow i \) on one statement

\[
\begin{align*}
\text{real :: a(n), b(n), x(n)} \\
a = \ldots; b = \ldots; x(1) = \ldots \\
!\text{omp parallel shared(a,b,x)} \\
!\text{omp do} \\
do \ i=2, n \\
\quad b(i) = f1( a(i), b(i) ) \\
\triangle x(i) = a(i) \times x(i-1) + b(i) \\
a(i) = f2( b(i), x(i) ) \\
\text{end do} \\
!\text{omp end do} \\
!\text{omp end parallel}
\end{align*}
\]

- race occurs on chunk boundaries executed by different threads:

- **Ordered clause and directive**
  - Syntax: uses a clause for the loop and an ordered region inside the loop body

\[
\begin{align*}
!\text{omp do ordered do} \ldots \\
\quad \ldots \\
!\text{omp ordered} \ldots \\
!\text{omp end ordered} \ldots \\
\text{end do} \\
!\text{omp end do}
\end{align*}
\]

- statements in **ordered** region are executed in same order as the loop's iteration variable is increased in a serial execution
- statements outside the **ordered** region can execute in parallel
- only **one** ordered region permitted

© 2010-19 LRZ/RRZE
Introduction to OpenMP
93
Corrected example

- Assures that code block with flow dependency is effectively serialized

```fortran
!$omp parallel shared(a,b,x)
!$omp do ordered
do i=2, n
    b(i) = f1( a(i), b(i) )
!$omp ordered
    x(i) = a(i) * x(i-1) + b(i)
!$omp end ordered
    a(i) = f2( b(i), x(i) )
end do
!$omp end do
!$omp end parallel
```

- $T_a$ signals completion of its chunk to $T_b \rightarrow$ synchronization avoids the race
Explicit cross-iteration dependency specification

- **depend clause**

```c
#pragma omp for ordered (1)
for (i=10; i<N; i++) {
    ... // independent execution

#pragma omp ordered \
    depend(sink:i-10)
    x[i] = a[i] * x[i-10] + b[i];

#pragma omp ordered \
    depend(source)
    ... // independent execution
}
```

- **Additional semantics:**
  - may be needed due to possibly different (schedule dependent) chunk assignment
  - might improve concurrency

© 2010-19 LRZ/RRZE
Introduction to OpenMP
Doacross loop nests

- **Ordered clause with a nesting depth specification**

```c
#pragma omp for ordered (2)
for (i=1; i<N; i++) {
  for (j=1; j<N; j++) {
    #pragma omp ordered depend(sink:i-1,j) depend(sink:i,j-1)
    a[i][j] = \ldots \ast a[i-1][j] + \ldots \ast a[i][j-1];
    #pragma omp ordered depend(source)
  }
}
```

- parallel execution within diagonal $i+j=d$ is possible, in order of $d$
- loop schedule tuning will be required

⚠️ beware potential deadlocks (incorrectly specified dependencies)
Mutual exclusion with locks

A shared lock variable can be used to implement specifically designed synchronization mechanisms

- mutual exclusion bound to objects → more flexible than critical regions
OpenMP lock variables

- Two variants of locks exist:
  - simple locks
  - nestable locks (will not be dealt with in detail in this course)

- Declaration of a lock variable

```fortran
use omp_lib

integer(omp_lock_kind) :: a_lock
integer(omp_nest_lock_kind) :: a_nestable_lock
```

```c
#include <omp.h>

... 

omp_lock_t a_lock;
omp_nest_lock_t a_nestable_lock;
```
Preparing locks for use

- The initial state of a lock variable is "uninitialized"
  - i.e. it is not actually associated with a lock variable
- Need to invoke an initialization function on it before it is used
  - subroutines / void functions provided in OpenMP run time

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_init_lock(omp_lock_t *lock)</td>
<td>initializes an uninitialized lock; the lock variable has the state &quot;unlocked&quot; on return</td>
</tr>
<tr>
<td>omp_destroy_lock(omp_lock_t *lock)</td>
<td>destroys a lock that has the state &quot;unlocked&quot;.</td>
</tr>
<tr>
<td>omp_init_nest_lock (omp_nest_lock_t *lock)</td>
<td>initializes an uninitialized nestable lock; the lock variable has the state &quot;unlocked&quot; on return, and its nesting count is zero.</td>
</tr>
<tr>
<td>omp_destroy_nest_lock (omp_nest_lock_t *lock)</td>
<td>destroys a nested lock that has the state &quot;unlocked&quot;.</td>
</tr>
</tbody>
</table>

- Fortran: replace *lock argument by integer of appropriate kind
Lock ownership

- An initialized OpenMP lock can be in one of the states unlocked, or locked.
- The (unique) task region that has successfully acquired the lock is said to own the lock.
- Only the task region that owns the lock can release it, returning it to the unlocked stage.

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>omp_set_lock(omp_lock_t *lock)</code></td>
<td>If the lock is already locked by another task region, block until the state of the lock changes. If the lock is in the state unlocked, acquire it, setting it to the locked state, and continue execution.</td>
</tr>
<tr>
<td><code>omp_unset_lock(omp_lock_t *lock)</code></td>
<td>Release the lock that is owned by the executing task region.</td>
</tr>
</tbody>
</table>

**Notes:**
- state combinations not described in the table are not permitted (e.g., a task region trying to unset a lock it does not own)
- the lock variable must be shared in the calling scope
Simplest possible example

- **Usage pattern analogous to named critical region**
  - programmer is responsible for relationship between lock and objects protected by it

```fortran
use omp_lib
integer(omp_lock_kind) :: lock
call omp_init_lock(lock)
...
!$omp parallel
call omp_set_lock(lock)
...  
call omp_unset_lock(lock)
...
!$omp end parallel
call omp_destroy_lock(lock)
```

```c
#include <omp.h>
omp_lock_t lock;
omp_init_lock(&lock);
#pragma omp parallel
{
    omp_set_lock(&lock);
    ...
    omp_unset_lock(&lock);
    ...
    }
omp_destroy_lock(&lock);
```

starts in unlocked state

only one thread at a time gets to play with the red balls

release resources
Non-blocking attempt at ownership

- **Function call signature**

  ```fortran
  logical function omp_test_lock(lock)
  int omp_test_lock(omp_lock_t *lock)
  ```

  - if the lock is already locked by another task region, return "false"
  - if the lock has the state unlocked, acquire it (setting the state to locked) and return the value "true".

- **Permits implementing additional concurrency**

  ```fortran
  !$omp parallel
  do while (.not. omp_test_lock(lock))
    ...
  end do
  ...
  call omp_unset_lock(lock)
  !$omp end parallel
  ```

  ```c
  #pragma omp parallel
  {
  while (!omp_test_lock(&lock)) {
    ...
  }
  ...
  omp_unset_lock(&lock);
  ```
Final notes on locking

- **Potential performance issues**
  - locks are a relatively expensive synchronization mechanism
  - lock contention (algorithm dependent)

- **Programming issues**
  - easy to produce deadlock (non-composable against other constructs)

- **Nestable locks**
  - extended semantics for repeated locking (additional nesting count)

- **Locks with hints (OpenMP 4.5)**
  - programmer can specify expected usage pattern, but the actual effect is implementation dependent
  - this is an advanced topic, and using this feature may require special hardware features (transactional processing)
  - OpenMP 5.0 will likely have some changes in this area
Synchronization overhead

- Syncbench from the EPCC OpenMP microbenchmarks is used
  - evaluates the overheads for all synchronizing constructs systematically
  - overhead is what remains even if no workload is processed
- Showing results as a function of thread count
  - alternatively, depending on node architecture and used compiler
- Note order of magnitude
  - a microsecond typically corresponds to a couple of thousand CPU cycles
Thread count dependence

Westmere 4-socket node overhead with ICC 15

overhead in µs

1 Thread 2 Threads 4 Threads 8 Threads 16 Threads 32 Threads

Parallel Barrier Critical / Lock Atomic Reduction

2 sockets used
Architecture dependence

2-socket results with ICC 15

overhead in µs

- Westmere-EX (20)
- AMD Magny Cours (16)
- Sandy Bridge EP (16)
- Haswell EP (28)

Parallel, Barrier, Critical / Lock, Atomic, Reduction
Compiler dependence

Westmere 20 thread results

overhead in µs

Parallel Barrier Lock Atomic Reduction

ICC 15 GCC 6.1 PGCC 15

© 2010-19 LRZ/RRZE
Introduction to OpenMP
How to deal with synchronization overhead

- **Therapy 1:**
  - use the right compiler
  - **note:** x86 does not (yet) support hardware synchronization

- **Therapy 2:**
  - execute serially for small problem sizes
  - conclude parallel execution if not needed any more

- **Therapy 3 (may be most effective):**
  - reduce the synchronization requirements of your algorithm
  - **Examples:** `nowait` clause, or extend parallel regions to reduce number of forks/joins

Now: Fifth exercise session
Tasking

Work sharing for irregular problems, recursive problems and information structures

Acknowledgement due to L. Meadows/T. Mattson (Intel) for their SC08 slides
**Processing information structures**

### Example: linked list

#### Fortran

```fortran
type :: list
    type(list), pointer :: next => null()
    real, allocatable :: data(:)
end type
```

#### C

```c
typedef struct {
    List *next;
    real *data; int n;
} List;
```

### Data layout

- each list item may carry a different payload
- parallel processing on a per-list-item basis $\rightarrow$ load imbalance is likely to occur
- the list as a whole is intended to be *shared* (i.e. no copies of payload should be created during processing)
Serial processing of a linked list

- **Not a regular loop in the sense of OpenMP**
  - cannot use work sharing constructs even though potential concurrency is obvious.

- **In general:**
  - API calls for processing information structures often are recursively invoked → OpenMP 2.5 offers no means of parallelization for this situation, although concurrency can be formally exposed.
What is a task?

- **Aim:** make OpenMP worksharing more flexible
- **Semantics:**
  - When a thread encounters a *task construct*, a task is generated from the code of the associated structured block.
  - *Data environment of the task* is created (according to the data-sharing attributes, defaults, …)
  - The encountering thread may immediately execute the task, or defer its execution.
    In the latter case, *any thread in the team* may be assigned the task.
- **Introduced with OpenMP 3.0**
  - additional features and improvements added in later versions of the standard
Concept of tasking

- If free resources are available,
  - expect task to start execution immediately
- Task binds to innermost enclosing parallel region

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

Illustration of deferred tasks:
- A thread (any one) encounters a `task` directive.
- OpenMP scheduler assigns execution of block to a `free` resource.
- Block and data put into queue.
- Task queue (a limited resource).

Simplest example: code sections

Fortran

```fortran
program code_sections
  use mod_functions
  implicit none
  real :: a, b
  integer :: n = ...

!$omp parallel
!$omp master

!$omp task
  a = function_1(n)
!$omp end task

!$omp task
  b = function_2(n)
!$omp end task

!$omp end master
!$omp end parallel
  write(*,*) a + b
end program
```

C

```c
int main() {
  float a, b;
  int n = ...;

  #pragma omp parallel
  #pragma omp master
  {
    #pragma omp task
    { a = function_1(n); }

    #pragma omp task
    { b = function_2(n); }

    // end parallel and master
  }
  printf("%f\n", a + b);
}
```

- Concurrency: `a` and `b` are concurrently executed if sufficiently many threads are available.
- Synchronization: There is no synchronization (different than `single`).
- Task Creation: Only thread 0 creates tasks.
- Thread Execution: Threads waiting here will be put to work.

© 2010-19 LRZ/RRZE
Data scoping in task regions

Recommendation:
• use a `default(none)` clause on all task directives
• explicitly specify the scoping for each data object

```c
int main() {
    float a, b;
    int n, i;
    a = ...; n = ...;
    #pragma omp parallel private(b) b = ...;
    #pragma omp master
    #pragma omp task
    {
        for (i=0; i<n; i++) {
            b = b + ...;
            ... = a + foo(i);
        }
    }
}
```

- `a` is shared (because it is shared in all lexically enclosing constructs)
- `i` is private (loop index)
- `b` is firstprivate
**Tasked linked list**

- Need to have local pointer `p` firstprivate:
  - avoid race condition on shared original (vs. subsequent update)
  - assure that association status is copied to thread executing the task region

```fortran
subroutine process_list(head)
  type(list), target :: head
  type(list), pointer :: p

  !$omp parallel
  !$omp single shared(p)
    p => head
  do while (associated(p))
    !$omp task firstprivate(p)
      call do_work(p%data)
    !$omp end task
    p => p%next
  end do
  !$omp end single nowait
  !$omp end parallel
end subroutine
```

```c
void process_list(list *head) {
  list *p = head;
  #pragma omp parallel
  {
    #pragma omp single \nowait shared(p)
    {
      while (p) {
        #pragma omp task firstprivate(p)
          { do_work(p->data, p->n); }
        p = p->next;
      }
    } // end single
  } // end parallel
}
```
The „if“ clause on a task directive

- When „if“ argument evaluates to „false“, 
  - the parent task must suspend execution until the encountered task region has been completed (an „undeferred task“). However, it is not fully clear from the standard whether the child task must be executed by the same thread.
  - but otherwise semantics are the same (with respect to data environment and synchronization) as for a „deferred“ task

```c
#pragma omp task firstprivate(p) if ( sizeof(p->data) > threshold )
{ do_work(p->data); }
```

```fortran
!$omp task firstprivate(p) if ( size(p%data) > threshold )
call do_work(p%data)
!$omp end task
```

- User-directed optimization („task pruning“)
  - avoid overhead for deferring small task
  - avoid creating too many tasks (resource limits!)
  - cache locality / memory affinity are likely to change
Recursive tasking

- **Divide and conquer**
  - initial function invocation in a parallel region, usually from a single thread

```c
float daq(float *data, int n) {
    float xl, yl;
    int n1 = ..., n2 = ...;
    float *data2 = ...;
    if (n1 < THRESHOLD) {
        ...
    }
    #pragma omp task shared(xl)
    {
        xl = daq(data, n1);
    }
    #pragma omp task shared(yl)
    {
        yl = daq(data2, n2);
    }
    #pragma omp taskwait
    return xl - yl;
}
```

- **Previous example:**
  - only sibling tasks are created

- **This example:**
  - each task creates two child tasks → „deep hierarchy“ of tasks

- **Scoping for x₁, y₁:**
  - start out as private variables
  - only newly created tasks share scope with these variables
  - shared scope is needed to communicate data outside the task regions

- **Terminate recursion**

© 2010-19 LRZ/RRZE

Introduction to OpenMP
The taskwait directive

- **suspends execution** until immediate child tasks of current task complete
  (the directive does **not** apply for descendants of child tasks)

**Syntax:**

```
 !$omp taskwait
```

```
 #pragma omp taskwait
```

**Needed in example from previous slide**

- avoid race condition of assignments vs. evaluation
- avoid local variables vanishing into thin air while tasks are still executing
Task switching

Possible issues with task scheduling:

- large number of tasks are created → implementation-defined limit on unassigned tasks may be reached
- all currently active tasks reach a synchronization statement → threat of deadlock?

Task switching

- permits a thread to suspend a task and start or resume another task at a task scheduling point
- for tied tasks, the same thread is obliged to resume execution of the suspended task later

Task scheduling points

- immediately after generation of a task
- at the end of a task region
- in implicit or explicit barrier regions (wait until all tasks executed by the team are done)
- in a taskwait region
- in a taskyield region
- at the end of a taskgroup region

e.g., a thread that creates lots of tasks may stop doing so and start working on one of them

tasks are tied by default ...
Thread switching

- Default behaviour:
  - a task assigned to a thread must be (eventually) completed by that thread → task is tied to the thread

- Change this via the untied clause
  - execution of task block may change to another thread of the team at any task scheduling point

- Deployment of untied tasks
  - Starvation scenario:
    Task switching has caused the task-generating thread to run a long calculation, with the result that all generated tasks were consumed and most threads idle.
    If the task that generates the work is untied, a different thread can take over the task-generating workload.
Interactions of Untied Tasks with other OpenMP features

⚠️ Thread-related semantics used in the untied task region are likely to trip you up, for example ...

- relying on results delivered by `omp_get_thread_num()`
  → may become inconsistent after thread switch

- referencing and defining values stored in threadprivate global variables
  → may access a different copy after thread switch

### Workaround

- revert from untied to tied for the duration of problematic operations, if possible

```c
#pragma omp task untied
{
    ...
}
```

```c
#pragma omp task final (1)
{
    ...
} // end included tied task
```

- or use an "if (0)" clause (undeferred task might be executed by a different thread, though)
Further potentially dangerous interactions

- **Use of threadprivate data by tied tasks**
  - value of threadprivate variables cannot be assumed to be unchanged across a task scheduling point. Might have been modified by another task executed by the same thread.

- **Tasks and locks:**
  - if a lock is held across a task scheduling point, interleaved code trying to acquire it (maybe using the same thread) may cause **deadlock**

  **Comment:** implementation-defined task scheduling points in **untied** tasks have been removed from the standard

- **Tasks and critical regions:**
  - similar issue if suspension of a task happens inside a critical region and the same thread tries to access the same critical region in another scheduled task

- **Tools?**
  - correctness tools will currently only find some of the issues that can arise
Programmer-defined task scheduling points

- **Syntax and Semantics**
  - Syntax:
    ```fortran
    !$omp taskyield
    #pragma omp taskyield
    ```
    - permits (but does not force) task suspension for the current task at the point where the directive is placed

- **Example**
  - Example:
    ```fortran
    subroutine foo ( lock, n )
    use omp_lib
    integer(kind=omp_lock_kind) :: lock
    integer :: n
    integer :: i
    do i = 1, n
      !$omp task
        call something_useful()
        do while &
          ( .not. omp_test_lock(lock) )
        !$omp taskyield
      end do
      call something_critical()
      call omp_unset_lock(lock)
    !$omp end task
    end do
    end subroutine
    ```
  - Example:
    ```c
    #pragma omp taskyield
    ```
    - avoid deadlock in a mutual exclusion region (taken from the OpenMP examples)
Task group synchronization

**Purpose:**

- synchronize all tasks created inside a structured block
- includes all descendants, not only immediate child tasks
- synchronization (i.e. waiting for task completion) happens at the end of the taskgroup region (task scheduling point)

**Syntax:**

```
$omp taskgroup
structured block
$omp end taskgroup
```

**Note:**

- tasks that were created before the taskgroup region started execution are not synchronized

**Example:**

recursive tasking with atomic updates in each task → guarantee completeness of updates
Final and mergeable tasks

### Final tasks
- use a **final** clause with a condition on a task directive
- if the condition evaluates to „true“, the resulting task is always **undeferred**, and is immediately executed by the parent task’s thread
- reduces the overhead of placing tasks in the “task pool”
- all tasks created inside task region are also final (different from an **if** clause)
- inside a task block, `omp_in_final()` can be used to check whether the task is final

### Merged tasks
- using a **mergeable** clause **may** create a merged task if it is undeferred or final
- a merged task has the **same data environment** as its creating task region

### Final and/or mergeable
- can be used for optimization purposes
- e.g. to optimize wind-down phase of a recursive algorithm

Current implementations seem not to actively support merging.
Task priority

- **Syntax**

  ```
  !$omp task priority(priority_value)
  #pragma omp task priority(priority_value)
  ```

- **Semantics**

  - Provides a hint to the run time on prioritizing (ordering) task execution
  - The priority value must be a non-negative integer; higher values correspond to higher priorities; maximum value is `omp_get_max_task_priority()`
  - Do not rely on a particular ordering of tasks imposed by specifying a priority

  ```
  #pragma omp task priority(9999)
  participants.get_coffee(100);
  ```
Example program

```fortran
s1 = ' '; s2 = 'and'; s3 = 'chaos'

!$omp parallel
!$omp master
!$omp task
  s1 = 'order'
  write(*, fmt='(a)', advance='NO') trim(s1) // ' '
!$omp end task
!$omp task
  write(*, fmt='(a)', advance='NO') trim(s2) // ' '
!$omp end task
!$omp task
  write(*, fmt='(a)', advance='NO') trim(s3) // ' '
!$omp end task
!$omp end master
!$omp end parallel
write(*, fmt='(a)', advance='NO') new_line('a')
```

Observed output with 3 threads can be any of ...
and order chaos
order chaos and
chaos order and
order and chaos
chaos and order
and chaos order
Introducing data-driven dependencies: the „depend“ clause

- Consider
  - a set of sibling tasks
  - a shared variable \( x \) that is referenced or defined by more than one of them

![Diagram showing dependencies between tasks A, B, and C]

- **depend(out:x)**: synchronizes memory operations against previously started tasks with an **inout** or **out** dependence on **same** memory location
- **depend(in:x)**: synchronizes memory operations against **any** defined dependence on **same** memory location for previously started task

- **depend(inout:x)**:...

**List of variables defines critical memory locations**

© 2010-19 LRZ/RRZE

Introduction to OpenMP
Via addition of depend clauses

```c
s1 = ''; s2 = 'and'; s3 = 'chaos'

!$omp parallel
!$omp master
!$omp task depend(out:s1)
    s1 = 'order'
    write(*, fmt='(a)', advance='NO') trim(s1) // ''
!$omp end task
!$omp task depend(in:s1) depend(out:s2)
    write(*, fmt='(a)', advance='NO') trim(s2) // ''
!$omp end task
!$omp task depend(in:s2)
    write(*, fmt='(a)', advance='NO') trim(s3) // ''
!$omp end task
!$omp end master
!$omp end parallel
write(*, fmt='(a)', advance='NO') new_line('a')
```

Observed output with 3 threads can only be order and chaos

The type of memory operation that is actually performed is irrelevant for the ordering properties (although it usually determines what type of dependency must be declared to avoid race conditions)
Real-world example: Cholesky decomposition

- Drawing the square root of a matrix

\[ A = L \cdot L^T \]

symmetric positive definite

lower triangular

- Recursive blocked algorithm

\[
\begin{array}{c|c}
A_{11} & A_{21}^T \\
A_{21} & A_{22} \\
\hline
L_{11} & O \\
L_{21} & L_{22} \\
\end{array}
\]

\[
\begin{array}{c|c}
L_{11}^T & L_{21}^T \\
O & L_{22}^T \\
\end{array}
\]

- LAPACK algorithm \( ? \)POTRF
Loop Parallel Cholesky

- **Phase 1**
  - one thread only
  - „hot“ block column is J

- **Phase 2**
  - parallel updates of columns $K = J + NB, J + 2*NB, ...$

---

**Step 1:** \(?\text{POTRF2}\) (non-blocked)

**Step 2:** \(?\text{TRSM}\) (linear equation)

**Step 3:** \(?\text{SYRK}\)

**Step 4:** \(?\text{GEMM}\)

Load imbalance → use suitable schedule
Reminiscence:

- Parallelization of Linear Algebra Algorithms on the KSR1, R. Bader (1994)
- same basic structure of algorithm, but OpenMP is more elegant
Extracting more parallelism

- **Phase 1**:  
  - multithread the TRSM update by subdividing the block column

- **Phase 2**:  
  - multithread the GEMM update by subdividing the block column
  - pipelined startup of SYRK/GEMM updates possible as phase 1 blocks complete

**Tasking makes this easy to do**

**Requirement:**  
need to specify the data dependencies  
→ Fortran array sections in depend clauses

**Note:**  
- nested parallelism has more overhead and is more difficult to manage
 Tasked implementation sketch

```c
!$OMP PARALLEL PRIVATE(JB, JJB, KB)
   DO J = 1, N, NB
      JB = MIN( NB, N-J+1 )
   !$OMP SINGLE
   !$OMP TASK &
   !$OMP& DEPEND(inout: &)
      CALL POTRF2( ... )
   !$OMP END TASK
   DO JJ = J+JB, N, NB
      JJB = MIN( NB, N-JJ+1 )
   !$OMP TASK &
   !$OMP& DEPEND(in: &)
   !$OMP& A(J:J+JB-1,J:J+JB-1) &
   !$OMP& DEPEND(inout: &)
   !$OMP& A(JJ:JJ+JJB-1,J:J+JB-1))
      CALL DSYRK( ... )
   !$OMP END TASK
   END DO
   !$OMP END SINGLE
   END DO
   !$OMP END PARALLEL

DO K = J+NB, N, NB
   KB = MIN( NB, N-K+1 )
   !$OMP TASK DEPEND(in: &)
   !$OMP& A(K:K+KB-1,J:J+JB-1))
   !$OMP& DEPEND(inout: &)
   !$OMP& A(K:K+KB-1,K:K+KB-1))
      CALL DSYRK( ... )
   !$OMP END TASK
   DO JJ = K+KB, N, NB
      JJB = MIN( NB, N-JJ+1 )
   !$OMP TASK DEPEND(in: &)
   !$OMP& A(JJ:JJ+JJB-1,J:J+JB-1),&
   !$OMP& A(K:K+KB-1,J:J+JB-1))
   !$OMP& DEPEND(inout: &)
   !$OMP& A(JJ:JJ+JJB-1,K:K+KB-1))
      CALL DSYRK( ... )
   !$OMP END TASK
   END DO
   END DO
   !$OMP END PARALLEL
```
Performance numbers

problem size: n = 10,000, block size: nb = 256

Cholesky Decomposition on 2.6 GHz Haswell

- MKL loop parallel
- task parallel

Solution time (s) vs. Threads
Performance numbers
problem size: n = 10,000, block size: nb = 256

Cholesky Decomposition on 1.4 GHz KNL

Solution time (s) vs. Threads

MKL, loop parallel, task parallel
### Technology advances ...

#### Comparing the N = 6000 solution time

<table>
<thead>
<tr>
<th></th>
<th>KSR1 (24 cells)</th>
<th>Haswell (28 cores)</th>
<th>KNL (64 cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>year of release</strong></td>
<td>1992</td>
<td>2014</td>
<td>2015</td>
</tr>
<tr>
<td><strong>solution time (s)</strong></td>
<td>270</td>
<td>0.13</td>
<td>0.16</td>
</tr>
<tr>
<td><strong>GFlop/s</strong></td>
<td>0.267</td>
<td>566</td>
<td>440</td>
</tr>
</tbody>
</table>

- Memory limit of machine
- Strong scaling limit
Applying tasking to loops

- Tasking and worksharing loops:
  - coexistence is difficult, because tasks are often issued in a context that does not permit application of "omp do/for"
  - creating a task for each loop iteration may be too fine-grained

- New construct: taskloop

```fortran
!$omp taskloop [clauses]
do var = ni, ne
  ...
end do
$omp end taskloop
```

```c
#pragma omp taskloop [clauses]
for (var = ni; var <= ne; var++) {
  ...
}
```

- creates task regions for iterations of associated loop(s)
Taskloop clauses

- **Scoping:**
  - private, firstprivate, shared, default

- **Inherited from work sharing:**
  - collapse, lastprivate

- **Inherited from tasking:**
  - if, final, mergeable, priority, untied

- **New clauses:**
  - grainsize(size)
  - num_tasks(num)
  - nogroup

  - **grainsize(size):**
    - Constrains number of iterations assigned to each task (upper limit < 2*grainsize)

  - **num_tasks(num):**
    - Maximum number of tasks created

  - **nogroup:**
    - By default, a taskloop construct implies a taskgroup region. This is similar to the sync at the end of a worksharing construct. The nogroup clause removes this additional synchronization.

Current compiler support is limited
Reductions across tasks

- **Known reduction properties:**
  - operation and involved variables
  - scope of clause: well defined starting point for creating private copies, and end points (usually with synchronization) for putting together partial results

  - the second property is not trivially assured in the context of tasking
  - it is not obvious which created tasks participate in a reduction

- **Two step procedure for reductions across tasks:**
  1. define the scope of the reduction (2 variants; note the synchronization point)

     ```
     !$omp parallel reduction(task, +:x)  
     ... ! create tasks (see below) 
     !$omp end parallel
     ```

     - parallel or worksharing region

     ```
     !$omp taskgroup task_reduction(+:x) 
     ... ! create tasks (see below) 
     !$omp end taskgroup
     ```

     - taskgroup or taskloop region

  2. inside the region, specify which explicit tasks participate:

     ```
     !$omp task in_reduction(+:x)  
     ... ! tasked code & data 
     !$omp end task
     ```

     requires consistency with specification in enclosing scope definition

Now: 6th exercise session
Performance: Architectural aspects
We need ideas on ...

- **What can be expected from the processor architecture?**
  - want at least an estimate for performance limits → avoid „stumbling in the dark“

- **How to exploit the architecture as best as possible**
  - use optimal data access patterns
  - minimize synchronization overhead
  - Account for interactions of OpenMP features with „serial“ optimization techniques (might be compiler optimization or lack thereof!)
Processor Architecture

- **Performance Characteristics**
  - determined by memory hierarchy

- **Impact on Application performance**: depends on where data are located
  - **temporal locality**: reuse of data stored in cache allows higher performance
  - **no temporal locality**: reloading data from memory (or high level cache) reduces performance

- **For multi-core CPUs**, available bandwidth may need to be shared between multiple cores
  - → shared caches and memory
Concept of cache

- A small but fast memory area
  - used for storing a (small) memory working set for efficient access

- Reasons:
  - physical and economic limitations

- Loads (stores) to (from) core registers
  - may trigger cache miss → transfer of memory block ("cache line", CL) from memory

- Cache fills up …
  - usually least recently used CL is evicted

- Example: $c(:) = a(:) + \ldots$
Control of Affinity NUMA effects False Sharing
Current Node architecture ...

- multi-core multi-threaded processors with a deep cache hierarchy
- typically, two **sockets** per node

Illustration shows 4 cores per socket. Current sockets have 8 – 14 cores

**ccNUMA** architecture: „cache-coherent non-uniform memory access“
An implementation might support this:

```c
#include <stdio.h>
int main() {
    #pragma omp parallel {
        ...
        #pragma omp parallel
        {
            ...
        }
    }
    return 0;
}
```

- nesting of parallel regions

mentioned here for illustrative purposes
Resource assignment

- **Suitable environment settings**
  ```
  export OMP_NUM_THREADS=4,2
  export OMP_NESTED=true
  export OMP_DYNAMIC=false
  ...
  ./my_nested_openmp_program.exe
  ```
  - one integer for each nesting level
  - else, „inner“ regions might/will execute with 1 thread only.
  - forbid implementation to interfere with number of threads assigned

- **Operating system:**
  - responsible for assigning hardware resources to threads
  - in general not trivial – note that (active) thread count can change during execution

- **Possible issues (performance impact):**
  - threads might move around between cores
  - multiple threads might share a core (or other resources)

→ a mechanism for controlling thread affinity / binding is desirable
Thread affinity – Processor binding

- **Two aspects:**
  1. What entity should a thread be bound to? → concept of place
  2. How should the binding be performed (if at all ...)?

- **Optimal binding strategy** depends on machine and application
- **Putting threads far apart** („spread“, „scatter“) might
  - improve aggregate memory bandwidth
  - improve combined cache size
  - decrease performance of synchronization constructs
- **Putting threads close together** (i.e. on two adjacent cores) might
  - improve performance of synchronization constructs
  - decrease available memory bandwidth and cache size per thread

→ available since **OpenMP 4.0**

before that: implementation-specific mechanisms
OpenMP place: a container unit for pinning of threads

- **Places are defined via either**
  - an abstract name (threads, cores, or sockets), optionally followed by a bracketed positive integer (number of places):
    
    ```
    export OMP_PLACES="cores(8)"
    ```

  - or an explicit list of places, specified as list of integer intervals (in the following example, all three specs are equivalent)
    ```
    export OMP_PLACES="{0,1,2,3},{4,5,6,7}"
    export OMP_PLACES="{0:4},{4:4}"
    export OMP_PLACES="{0:4}:2:4"
    ```

  meaning of the index is implementation defined, but you can expect the smallest unit of execution (a hardware thread on x86) to be used.
OpenMP binding

- **Determine whether threads should be pinned**
  - environment variable `OMP_PROC_BIND`
  - with values `true` or `false`, or
  - a comma-separated list of entries:

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>master</td>
<td>bind created threads to same place as master thread</td>
</tr>
<tr>
<td>close</td>
<td>bind created threads to a place close to the one assigned to the master</td>
</tr>
<tr>
<td>spread</td>
<td>use a sparse distribution pattern to bind created threads to places</td>
</tr>
</tbody>
</table>

- **Example:**

```
export OMP_PROC_BIND=spread,close
```

- binding is determined for at most two levels of parallel nesting
Example for OpenMP binding

**Nested parallelism example from earlier**

- Overcommitment causes places to be reused (i.e. multiple threads per place)
Identifying binding strategy within the program

- The function

```fortran
integer(...) function omp_get_proc_bind()
```

```c
omp_proc_bind_t omp_get_proc_bind(void)
```

returns one of the following **constants**:

<table>
<thead>
<tr>
<th>omp_proc_bind_false</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_proc_bind_true</td>
<td>1</td>
</tr>
<tr>
<td>omp_proc_bind_master</td>
<td>2</td>
</tr>
<tr>
<td>omp_proc_bind_close</td>
<td>3</td>
</tr>
<tr>
<td>omp_proc_bind_spread</td>
<td>4</td>
</tr>
</tbody>
</table>

- The value may depend on the nesting level from which the function is called
Identifying placement

- A number of functions exist to handle various inquiries:

<table>
<thead>
<tr>
<th>Name</th>
<th>Result type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>omp_get_num_places()</td>
<td>int</td>
<td>number of places available</td>
</tr>
<tr>
<td>omp_get_place_num_procs (int place_num)</td>
<td>int</td>
<td>number of processors available in place_num (0 .. number of places - 1)</td>
</tr>
<tr>
<td>omp_get_place_proc_ids (int place_num, int *ids)</td>
<td>void</td>
<td>ids contains numerical identifiers of processors in place place_num</td>
</tr>
<tr>
<td>omp_get_place_num()</td>
<td>int</td>
<td>place number of place to which calling thread is bound</td>
</tr>
<tr>
<td>omp_get_partition_num_places()</td>
<td>int</td>
<td>number of places in place partition of innermost implicit task</td>
</tr>
<tr>
<td>omp_get_partition_place_nums (int *place_nums)</td>
<td>void</td>
<td>list of place numbers for innermost implicit task</td>
</tr>
</tbody>
</table>
A proc_bind clause can be specified

Example:

```c
#pragma omp parallel num_threads(4) proc_bind(spread)
{ ...
#pragma omp parallel num_threads(2) proc_bind(close)
{ ...
}
```

executed with OMP_PLACES=cores(8)

- Node
  - outer region
  - inner region

- Socket 0
  - $S_0$, $S'_0$, $S_1$, $S''_0$, $S''_1$

- Socket 1
  - $S_2$, $S'''_0$, $S'''_1$, $S^v_0$, $S^v_1$
Identifying node topology

- **Topology =**
  - Where in the machine does core #n reside?
  - awkward numbering anyway?
  - which cores share which cache levels
  - which hardware threads (“logical cores”) share a physical core?

- **Use LIKWID tool to identify**
  - developed by J. Treibig
  - [https://github.com/RRZE-HPC/likwid](https://github.com/RRZE-HPC/likwid) has source code and documentation

- **Available commands**
  - `likwid-topology`: Print thread and cache topology
  - `likwid-pin`: Pin threaded application without touching code
    - `likwid-perfctr`: Measure performance counters
    - `likwid-mpirun`: mpirun wrapper script for easy LIKWID integration
    - `likwid-bench`: Low-level bandwidth benchmark generator tool
      - … some more
Output of likwid-topology --g (ASCII art section):

Socket 0:

```
Socket 0:
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
| 0  16 | 1  17 | 2  18 | 3  19 | 4  20 | 5  21 | 6  22 | 7  23 |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
| 32kB | 32kB | 32kB | 32kB | 32kB | 32kB | 32kB | 32kB |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
| 256kB | 256kB | 256kB | 256kB | 256kB | 256kB | 256kB | 256kB |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
|                                      20MB                                      |
```

Socket 1:

```
Socket 1:
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
| 8  24 | 9  25 | 10  26 | 11  27 | 12  28 | 13  29 | 14  30 | 15  31 |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
| 32kB | 32kB | 32kB | 32kB | 32kB | 32kB | 32kB | 32kB |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
| 256kB | 256kB | 256kB | 256kB | 256kB | 256kB | 256kB | 256kB |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
|                                      20MB                                      |
```

- Hyperthreaded cores
- L1D
- L2
- Shared L3
- Each socket forms a NUMA domain
Output of **likwid-topology –g** (ASCII art section):

---

**Socket 0:**

```
+-------------------------------------------------------------+
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |                         |                     |
| +-------------------------------------------------------------+
| | 64kB | 64kB | 64kB | 64kB | 64kB | 64kB | 64kB | 64kB |                     |                     |
| +-------------------------------------------------------------+
| | 512kB | 512kB | 512kB | 512kB | 512kB | 512kB | 512kB | 512kB |                     |                     |
| +-------------------------------------------------------------+
| | 5MB | 5MB |                     |                     |                     |                     |                     |                     |
+-------------------------------------------------------------+
```

**Socket 1:**

```
+-------------------------------------------------------------+
| | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |                         |                     |
| +-------------------------------------------------------------+
| | 64kB | 64kB | 64kB | 64kB | 64kB | 64kB | 64kB | 64kB |                     |                     |
| +-------------------------------------------------------------+
| | 512kB | 512kB | 512kB | 512kB | 512kB | 512kB | 512kB | 512kB |                     |                     |
| +-------------------------------------------------------------+
| | 5MB | 5MB |                     |                     |                     |                     |                     |                     |
+-------------------------------------------------------------+
```

- **single threaded cores**
- **L1D**
- **L2**
- **shared L3**
- **each socket forms two NUMA domains**
likwid-pin – Overview

- Pins processes/threads to specific cores **without touching code**
  - Directly supports pthreads, gcc OpenMP, Intel OpenMP
  - Based on combination of wrapper tool together with overloaded pthread library → binary must be **dynamically linked**!

- Can also be used as a superior replacement for Linux command **taskset**

- Supports logical core numbering within a node and within an existing CPU set
  - Useful for running inside CPU sets defined by someone else, e.g., the MPI start mechanism or a batch system

- Usage examples:
  - Physical numbering (as given by likwid-topology):
    ```bash
    likwid-pin -c 0,2,4-6 ./myApp parameters
    ```
  - Logical numbering by topological entities:
    ```bash
    likwid-pin -c S0:0-3 ./myApp parameters
    ```
Memory affinity

- **Allocation of memory** (with C malloc() / Fortran ALLOCATE)
  - only provides a virtual memory address
- **Physical memory**
  - is assigned when a memory location is initialized ("first touch")
  - units of pages (note overhead due to page faults!)
- **Consequence for OpenMP**
  - possible memory accesses across socket boundaries

```fortran
a(:) = 0.0
!$omp parallel do
DO i=1, size(a)
    ... = ... a(i) ...
END DO
!$omp end parallel do
```

- only **half** the available memory BW might be exploited on a 2-socket system
Balancing memory affinity

- **Desirable and** **scalable** memory access pattern:
  - requires initialization with an OpenMP parallelized loop

- **Distributed first touch**
  - ideally, uses same loop schedule as later processing

```fortran
$omp parallel do
DO i=1, size(a)
  a(i) = ...
END DO
$omp end parallel do
...
$omp parallel do
DO i=1, size(a)
  ... = ... a(i) ...
END DO
$omp end parallel do
```

- now, the **full** available memory BW can be exploited on a **multi**-socket system
MVM performance for N=8000

- Measured on two AMD Magny Cours sockets
  - thread pinning uses "close" strategy
### Tasking and NUMA effects

- **Remember:**
  - tasking decouples data items and associated functions from the threading model

  ```
  #pragma omp task
  execute_my_function(a, b, c);
  ```

- **Consequence:**
  - repeated execution of tasking on data items might use different threads → memory affinity will get lost!

  ```
  #pragma omp task shared(a, b, c)
  establish_my_data(a, b, c);
  #pragma omp taskwait
  #pragma omp task shared(a, b, c)
  execute_my_function(a, b, c);
  ```

  This function might execute on a **different** thread than this one.
Partial therapy: register locality

- **At initialization**
  - store which thread performed it – threads are color coded below

```
integer :: work_item(idm, nthr)
```

- **Working on data items**
  - first work on items that are local to the executing thread
  - next work on items that are located elsewhere (nearby first)
    - task stealing due to unpredictable thread assignment
  - additional bookkeeping (mutual exclusion) is needed to assure complete and unique execution
Simple example: NUMA-aware tasked triads

work shared vector triad with 16 threads on Sandy Bridge

- omp do
- omp task
- omp task with locality

biggest possible chunk size was used → 1 chunk per thread possible

tasking overhead

mutex effect

effective memory BW

© 2010-19 LRZ/RRZE
Performance problems with small shared variables

- Example program: count even and odd array values

```fortran
integer is(2), ict(2, ntdm), ia(n)
...
!$omp parallel private(myid) shared(ict, ia)
  myid = omp_get_thread_num()+1
!$omp do private(index)
  do i=1,n
    index = mod(ia(i),2)+1
    ict(index,myid) = ict(index,myid) + 1
  end do
!$omp end do
!$omp critical
  is = is + ict(1:2,myid)
!$omp end critical
!$omp end parallel
```

Initialization is omitted formally correct, no race condition
Example program parallel efficiency

- Baseline 1 thread execution time: AMD 0.75 s, Intel SandyBridge 0.37 s
Updating neighbouring data from different cores

- **Store operation**
  - write back always done on complete cache lines
  - "merging of partial cache lines" is not possible

- **Cache coherence protocol**
  - keeps track of cache line status
  - assures data consistency by enforcing hardware synchronization between writes
Typical sequence of write operations

- **Hardware execution sequence for write on Core 0:**
  1. Request exclusive access to CL (Core 0 issues it first)
  2. **Invalidate** CL in Cache 1
  3. Modify CL in Cache 0 (exclusively owned)
  4. mark CL **shared**

- **Hardware execution sequence on Core 1:**
  5. Request CL from memory for reading (granted after CL is marked shared)
  6. Request exclusive access to CL
  7. **Invalidate** CL in Cache 0
  8. Modify CL in Cache 1 (exclusively owned)
  9. mark CL **shared**

Diagram shows state after step 3
Consequences

- Repeated access to data in same cache line:
  - causes thrashing of cache lines
  - for each access, more than twice the memory latency may be accumulated, resulting in significant performance reduction

- This effect is called "false sharing"
Avoidance of false sharing

- Privatization – here through use of a reduction variable

```fortran
integer is(2), ia(n)
...
!
omp parallel shared(ict, ia)
!
omp do private(index) reduction(+:is)
  do i=1,n
    index = mod(ia(i),2)+1
    is(index) = is(index) + 1
  end do
!
omp end do
!
omp end parallel
```

- Alternative for retaining shared variables: Add padding
  - tradeoff: may lose spatial locality
Parallel efficiency for improved example

- Baseline 1 thread execution time: AMD 0.81 s, Intel SandyBridge 0.36 s

Remaining performance degradation is due to saturation of memory bus.
Outlook: Towards quantifying performance
Using synthetic loop kernels for performance evaluation

- **Characteristics**
  - known operation count, load/store count
  - some variants of interest:

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Name</th>
<th>Flops</th>
<th>Loads</th>
<th>Stores</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s = s + a_i \times b_i )</td>
<td>Scalar Product</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>( n^2 = n^2 + a_i \times a_i )</td>
<td>Norm</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( a_i = b_i \times s + c_i )</td>
<td>Linked Triad (Stream)</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>( a_i = b_i \times c_i + d_i )</td>
<td><strong>Vector Triad</strong></td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

- run repeated iterations for varying vector lengths (working set sizes)
Vector Triad \( D(+) = A(+) + B(+) \times C(+) \)

- **Synthetic benchmark**: bandwidths of „raw“ architecture for a **single core** Sandy Bridge 2.7 GHz / ifort 13.1

- **Comparative Graph**:
  - **L1D** – 32kB: < 112 GB/s
  - **L2** – 256 kB: < 62 GB/s
  - **L3** – 20 MB: ~ 33 GB/s
  - **Memory** ~ 14.7 GB/s

- **Vectorization (256 Bit registers)** provides performance boost mostly in L1/L2 cache

- Measured „effective“ BW:
  - 3 LD+1ST
  - 16 Bytes / Flop, repeated execution (actually issued: 4 LD+1ST in L2 and higher)
### Theoretical performance limit

#### Sandy Bridge vector unit:
- 256 Bit SIMD (single instruction multiple data)
- Example: addition of 8 Byte words

#### Instruction capability
- 1 vector add and 1 vector mult per cycle → theoretical Peak 8 Flops/cycle

#### LD/ST issue capability
- 4 Words LD/cycle
- 4 Words ST/(2 cycles)

#### Vector triad:
- required loads limit performance to 8 Flops / 3 cycles
  i.e. 7.2 GFlop/s at 2.7 GHz

#### Consult processor-specific architecture manual
Vector Triad $D(\cdot) = A(\cdot) + B(\cdot) \times C(\cdot)$

- **Throughput mode:** run with independent threads **up to number of cores** on a socket

![Graph showing performance with different thread counts](image)

- L1/L2/L3 bandwidths scale well
- Effective per-core share of L3 shrinks
- Memory interface of socket: saturated with 4 threads
Looking at Memory Performance

N=12506888 Vector Triad

- Satisfaction of 1st socket with 4 threads
- Second socket memory interface

Per-socket bandwidth:
- Sandy Bridge: 40 GB/s
- Magny Cours: 24 GB/s
More on cache-based memory systems

- **Loads and Stores**
  - usually apply to cache lines
  - size: 64, 128 or more Bytes

- **Pre-fetch**
  - avoid latencies when streaming data

- pre-fetches usually done in hardware
- decision according to memory access pattern

**Pre-Requisite:**
- **spatial** locality
- violation of spatial locality:
  if only part of a cache line is used → effective reduction in bandwidth
Performance of strided triad on Sandy Bridge
- loss of spatial locality

\[ D(\cdot:\text{stride}) = A(\cdot:\text{stride}) + B(\cdot:\text{stride}) \times C(\cdot:\text{stride}) \]

Notes:
- stride known at compile time
- serial compiler optimizations may compensate performance losses in real-life code

- ca. 40 MFlop/s
  (remains constant for strides > ~25)
Returning to the matrix-vector product

\[ r = M \cdot x \quad \text{i.e.} \quad r_i = \sum_{j=1}^{n} M_{ij} x_j \]

- First parallelization attempt:

```plaintext
!$omp parallel
!$omp do
DO j = 1, n
  DO k = 1, n
    r(j) = r(j) + a(j, k) * x(k)
  END DO
END DO
!$omp end do
... = r(…)
!$omp end parallel
```

- Parallel patterns used:
  - data decomposition (load balanced)
  - loop parallelism (no dependencies)

- Directive placement:
  - coarse grained parallelism to avoid synchronization overhead
Measured performance (size 8000)

- **Speed-Up:** \[ S(n_t) = \frac{T(1)}{T(n_t)} \]
  
  as a function of number of threads on 8-core processors

- **Absolute performance:**
  - MFlop/s = 2 \cdot n^2 / time

- Scaling **bad** beyond 4 threads

- **Speed-Up useless** if baseline performance is bad

- used \texttt{dgemv} for serial run

- A measure for execution time if problem size is constant
Improved Matrix-Vector Multiply

- **Switch loop order**
  - map *column* blocks to threads:

- **Variant 2 of code**:
  - *contiguous* access to M
  - array reduction on result vector

```c
!$omp parallel do reduction(+:r)
DO k = 1, n
  DO j = 1, n
    r(j) = r(j) + M(j, k) * x(k)
  END DO
END DO
!$omp end parallel do
```

- **Performance estimate for single thread**:
  - double that of triad → 1.86 GFlop/s

Cannot be the whole truth – remember serial performance: 3.7 GFlop/s!
For variant 2 of the MVM: Performance in MFlop/s

Comments:

- „no OpenMP“ → variant 2 compiled without OpenMP
- Conclusion: compiler stops making certain serial optimizations if OpenMP switch is toggled
Variant 3: Reduce memory traffic

- **Outer loop unrolling**

  ```c
  !$omp parallel do reduction(+:r)
  DO k = 1, n-3, 4
    DO j = 1, n
      r(j) = r(j) + M(j, k) * x(k) &
            + M(j, k+1) * x(k+1) &
            + M(j, k+2) * x(k+2) &
            + M(j, k+3) * x(k+3)
    END DO
  END DO
  !$omp end parallel do
  ```

  - conditioning omitted
  - asymptotically increases intensity to 2 Flops per word (1 load on matrix per original loop iteration)

Unrolling is **limited** by number of available registers and prefetch streams (architecture-dependent!)

- **Expected performance**

  - for M from memory (i.e. outside any cache)
  - contiguous streaming of data
  - assuming 40 GB/s bandwidth for a socket

  \[
  \text{Perf} = \frac{2 \text{ Flop}}{8 \text{ Bytes}} \times \frac{40 \text{ GB}}{\text{s}} = 10 \text{ GFlop/s}
  \]

- estimation method is known as „Roofline Model“
Variant 3 MVM performance (N=8000)

- **In MFlop/s.** Unroll factors: Sandy Bridge 4, Magny Cours 8

- **Comment:**
  - roofline model only predicts „saturated“ performance
  - single-thread performance is limited by non-overlapping memory/core operations (see ref. (2))
Why use variant 3 …

- … if variant 2 gives us the full performance anyway?
  - even if this only is attained with 8 threads

- Possible reasons:
  - „switch off“ cores 6-8 to save energy (relevant for you if this is budgeted – may happen not too far in the future!)
  - use cores 6-8 for other tasks that are cache bound
  - use cores 6-8 for MPI communication (I/O via PCI) if you do hybrid programming (i.e., combine MPI with OpenMP)
References
Recommended reading

(1) OpenMP 5.0 standard and OpenMP 4.5 examples at
http://www.openmp.org/specifications/

(2) Parallel programming in OpenMP
Rohit Chandra et al; Morgan Kaufmann 2000

(3) Using OpenMP - portable shared memory parallel programming
B. Chapman, G. Jost, R. van der Pas; MIT Press 2008

(4) Using OpenMP - the next step
R. van der Pas, E. Stotzer. Ch. Terboven; MIT Press 2017

(5) J. Treibig, G. Hager, G. Wellein: LIKWID
A lightweight performance-oriented tool suite for x86 multicore environments.
PSTI2010, Sep 13-16, 2010, San Diego, CA DOI: 0.1109/ICPPW.2010.38; Preprint:
http://arxiv.org/abs/1004.4431

(6) G. Hager, J. Treibig, J. Habich, and G. Wellein:

Appendix:
Setting up Vtune Amplifier
Tuning of serial and threaded programs
  • performance counter access requires group rights

Start up GUI
  • prerequisites: set up environment and possibly stack limit
  • then, invoke the GUI with `amplxe-gui &`

• command line `amplxe-c1` is also available, but will not be discussed

Project generation analogous to Intel Inspector
Example run: A badly performing solution of the histogram calculation

```c
#pragma omp parallel private(seed,i,k,me)
{
    me = omp_get_thread_num();
    seed = 123 + 159*me;
    for (k=0; k<100000; ++k) {
        #pragma omp for
        for (i=0; i<10000; ++i) {
            ir[i] = rand_r(&seed) & 0xf;
        }
        #pragma omp master
        for (i=0;i<10000; ++i) {
            hist[ir[i]]++;
        }
        #pragma omp barrier
        // prevents ir from being modified
        // before hist update is done
    }
}
```
Various types are provided
- select "Concurrency"
- in the project properties, set OMP_NUM_THREADS to number of physical cores

Note:
performance quality evaluation assumes complete system is used

Note:
- analysis may take quite a long time to run, even for programs of small size
Result tabs: Summary

- **Result:**
  - thread concurrency very low although CPU usage is high
Observation:
- much time spent in OpenMP run time library
- lots of transitions indicated → have false sharing
Drill down to source

- Click on routine with significant resource usage

```
19 // cannot exactly reproduce serial result
20 // but with fixed thread number result
21 // between runs
22 for (k=0; k<100000; ++k) {
23     #pragma omp for
24     for (i=0; i<10000; ++i) {
25         ir[i] = rand_r(&seed) & 0x7f;
26     }
27 #pragma omp master
28     for (i=0; i<10000; ++i) {
29         hist[ir[i]]++;
30     }
31 #pragma omp barrier
32 // barrier prevents ir from being modi
33 }
```

- many updates to small shared variable
- resulting load imbalance is collected at barriers