



# Introduction to OpenMP

R. Bader (LRZ)

G. Hager (RRZE)

V. Weinberg (LRZ)



## 1. Increase performance / throughput of CPU core

- Reduce cycle time, i.e. increase clock speed (Moore)
- Increase throughput, i.e. superscalar + SIMD

## 2. Improve data access time

- Increase cache size
- Improve main memory access (bandwidth & latency)

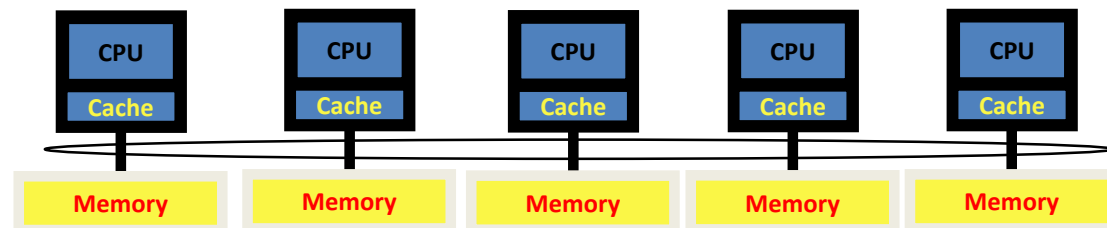
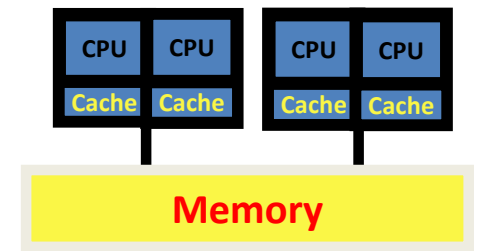
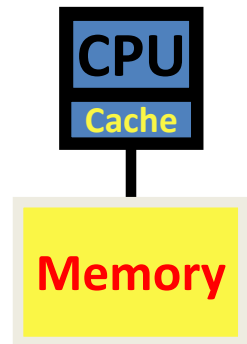
## 3. Use parallel computing (shared memory)

- Requires shared-memory parallel programming
- Shared/separate caches
- Possible memory access bottlenecks

## 4. Use parallel computing (distributed memory)

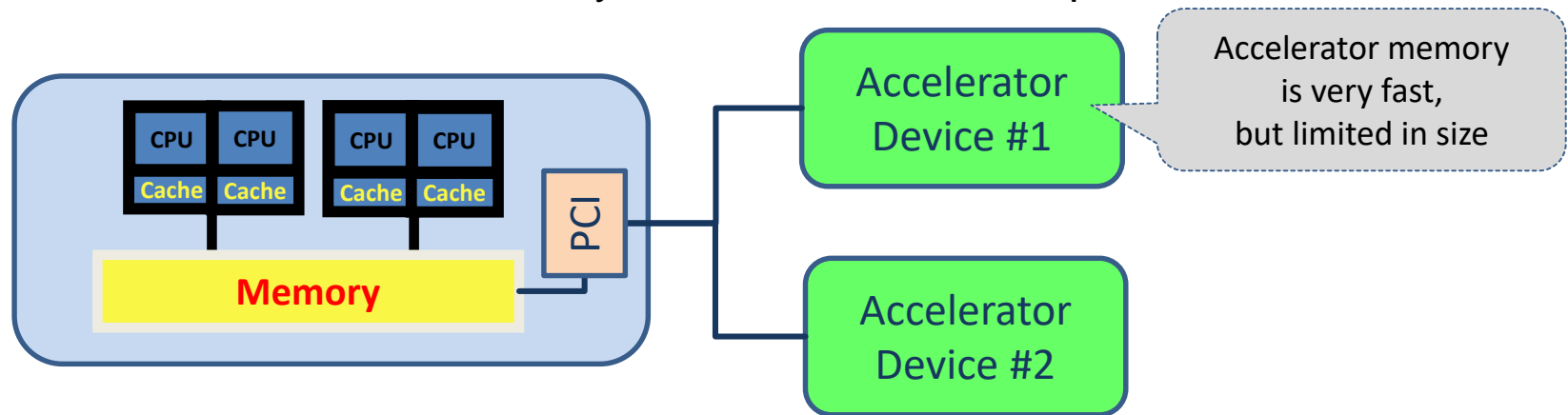
*“Cluster” of computers tightly connected*

- Almost unlimited scaling of memory and performance
- Distributed-memory parallel programming



## 5. Use an accelerator with your compute node

- Requires offload of program regions as well as data (semantics may be limited)
- Host and accelerator memory are connected, but separate



(Improvements are under way)

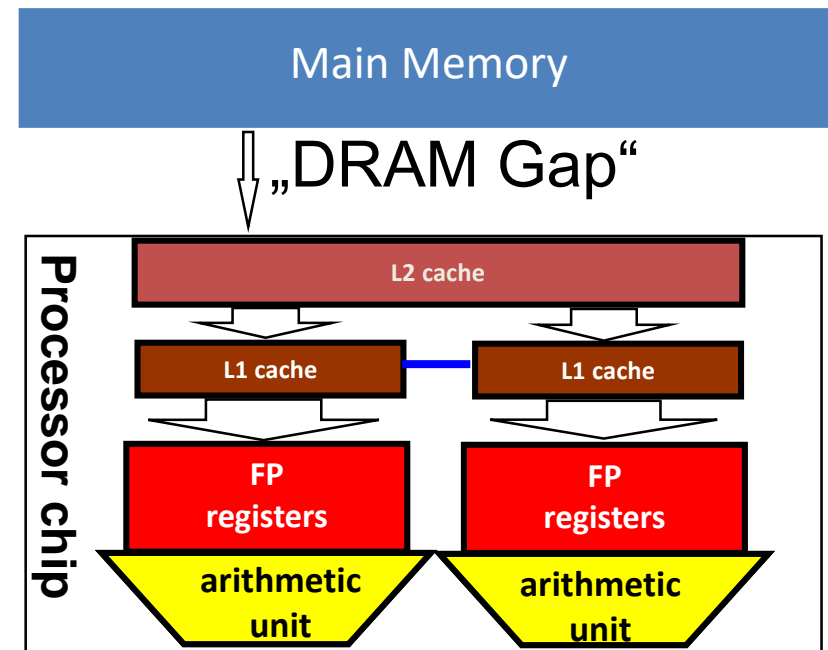
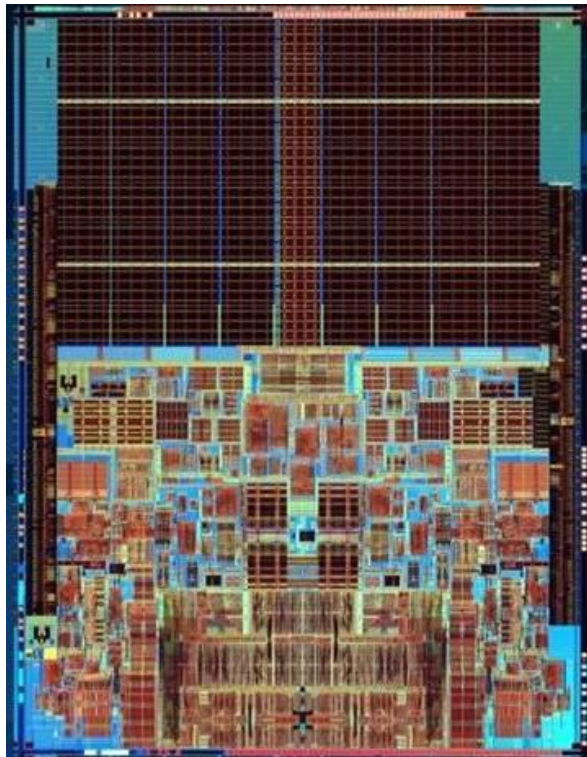
- Programming complexity is higher than for shared memory systems („heterogeneous parallel computing“)

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

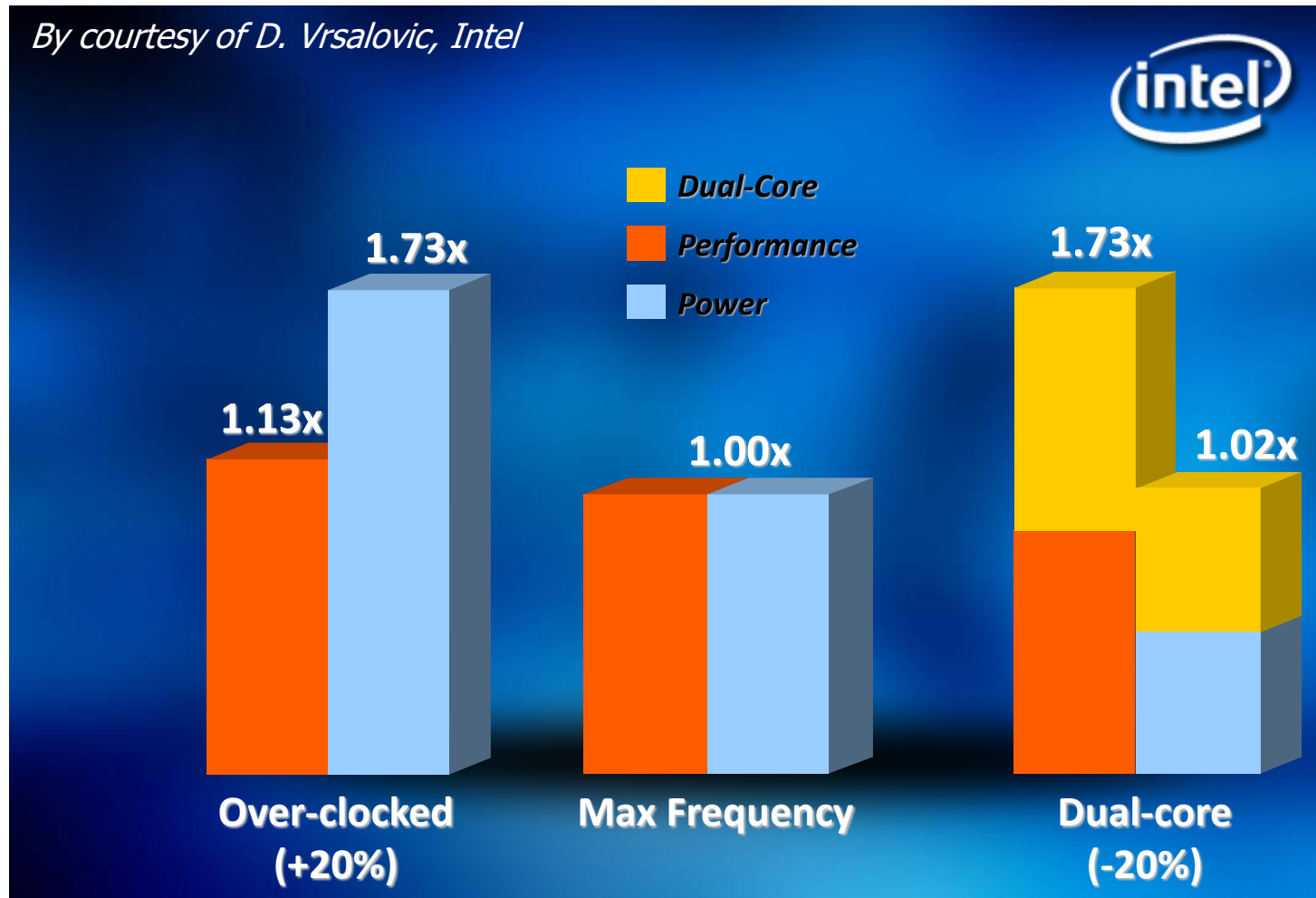
Put multiple processors (“cores”) on a chip which share resources (example shows a dual core that shares L2 cache and memory bandwidth)

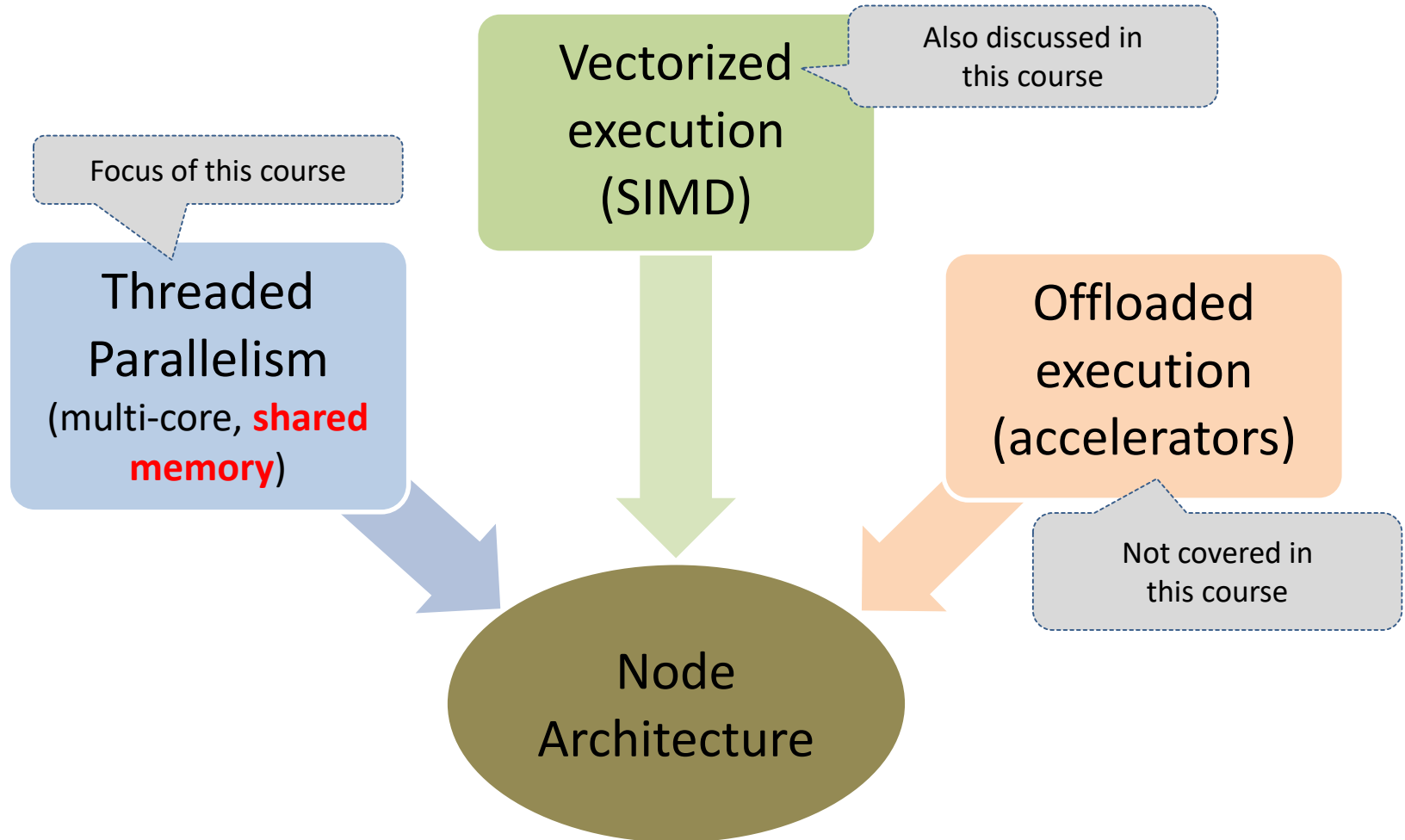
Efficient use of all cores for a single application → programmer

Intel Xeon (Woodcrest)



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





- **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

## 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?

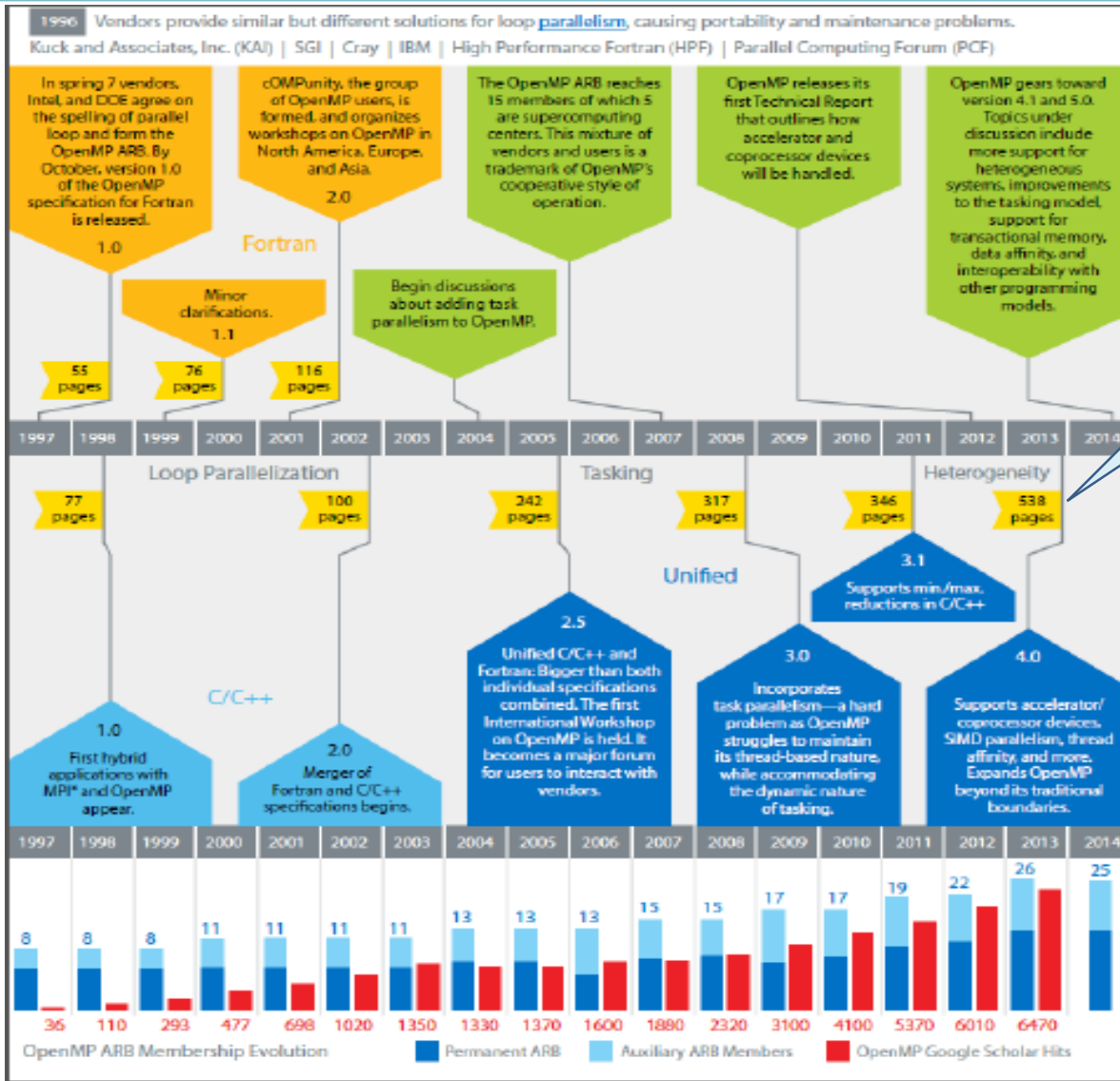
- **no associativity** for model number operations
- parallel execution might reorder operations  
(programmer may need to enforce ordering for reproducibility and/or numeric stability)



- **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> (including standard documents)
  - <http://www.compunity.org>
- **Note:**
  - LRZ has become a member of the OpenMP ARB in March, 2019

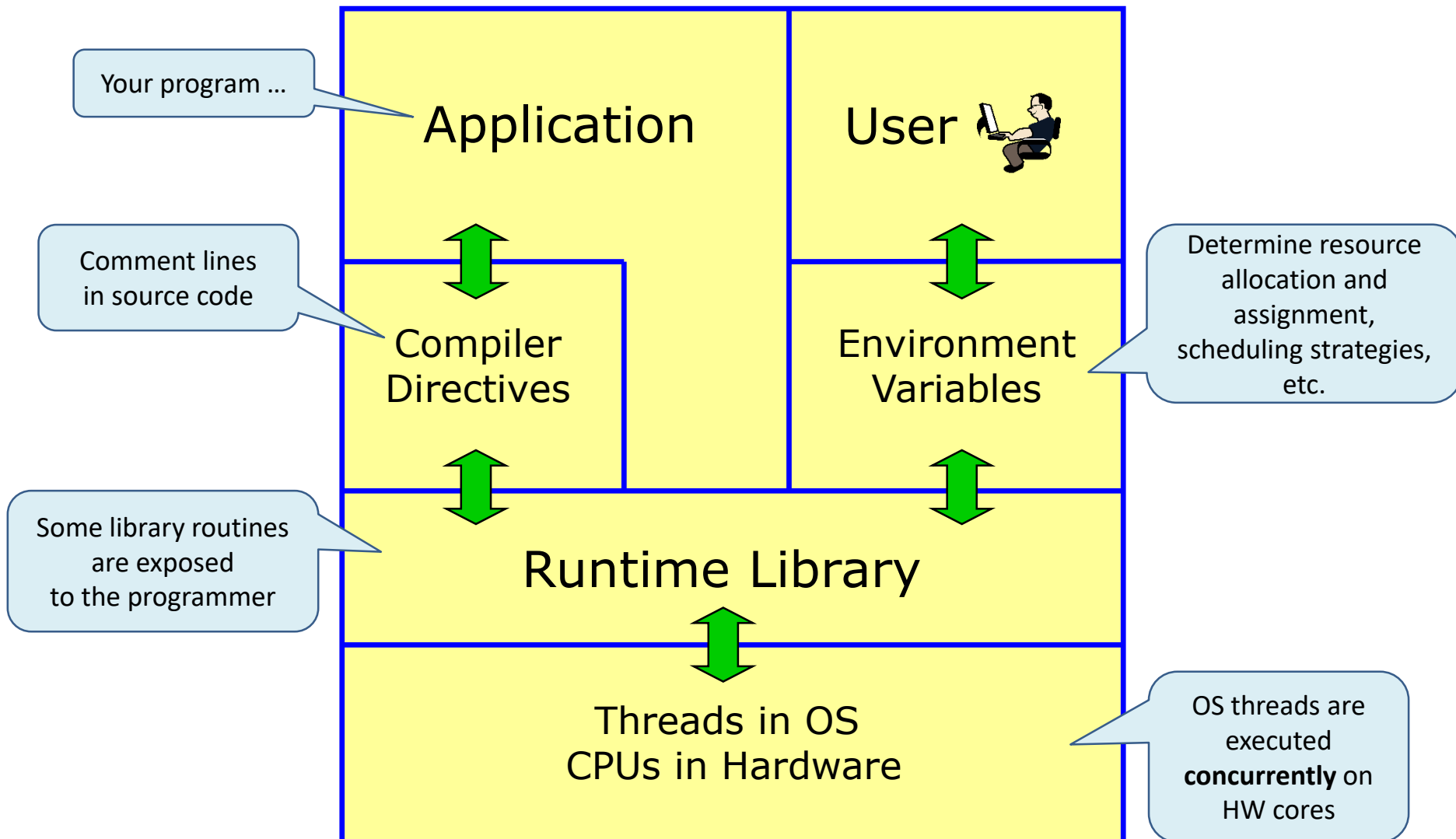
History of OpenMP  
starts in 1997

Fortran and C examples  
will be displayed



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



Fortran

```
program
  use m
  implicit none

  call f()

end program

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

Aim is to execute  
multiple instances of  
f() concurrently

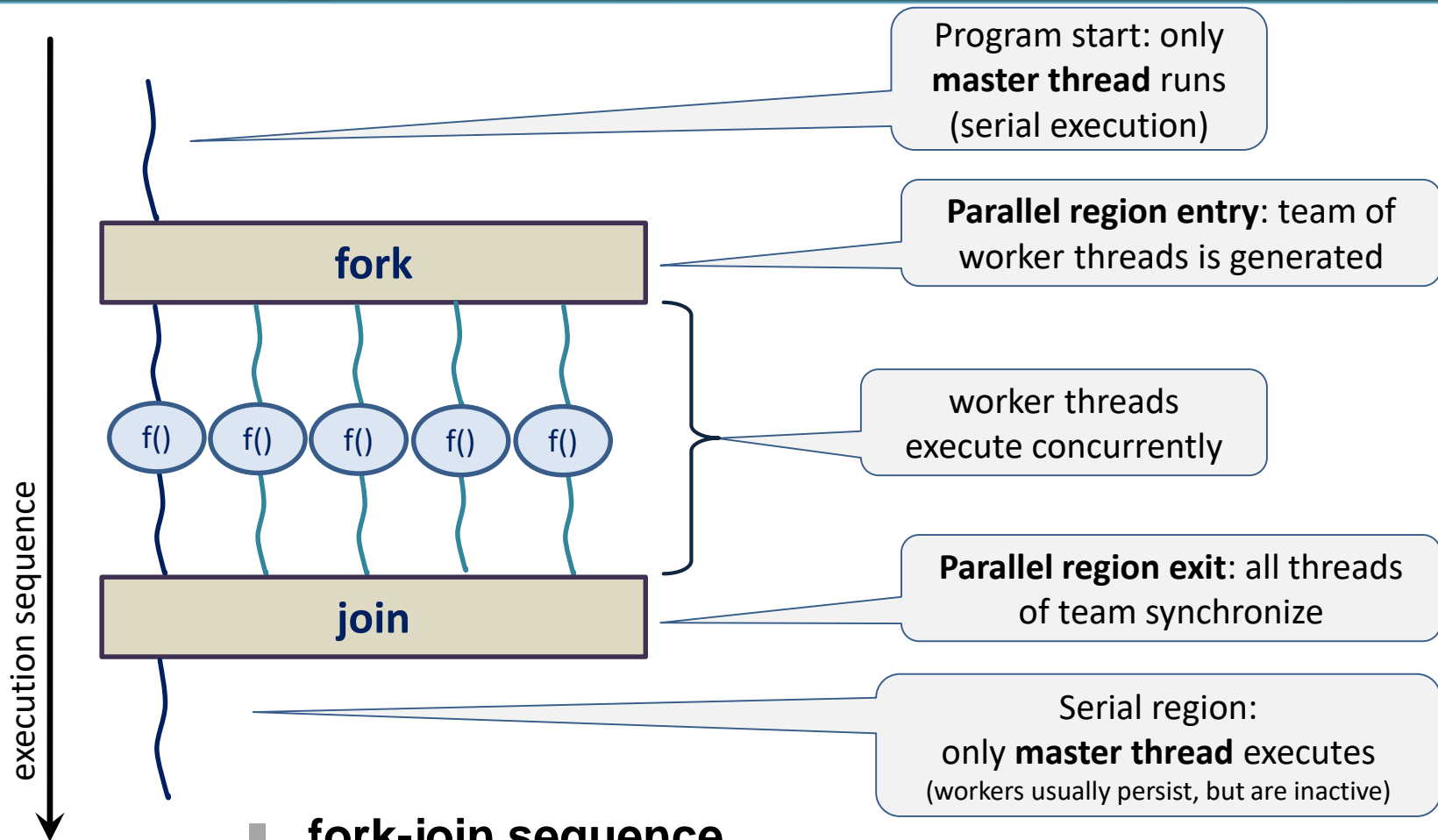
C

```
#include <stdio.h>
int main() {

  f();

  return 0;
}

void f() {
  printf("Hello\n");
}
```



- **fork-join sequence**
  - can repeat, with differing thread counts

Fortran

```

program
  use m
  implicit none
!$omp parallel
  call f()
!$omp end parallel
end program

```

enclosed  
lexical block

C

```

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

```

### General form of directives:

```
!$omp <directive> [<clause>]
```

sentinel

```
#pragma omp <directive> [<clause>]
```

sentinel

- clauses, if present, modify a directive's semantics
- multiple clauses per directive are possible
- continuation lines are supported for long directives:

Fortran

&amp;

C

\

## Fortran

- statements between a beginning and ending directive pair

## C / C++

- delineated by braces following a directive

**single point of entry**

- GOTO into block is prohibited
- setjmp() into block is prohibited

**single point of exit**

- GOTO, RETURN, EXIT outside block are prohibited
- longjmp() and throw() outside block are prohibited

**permitted: program termination**

- STOP, ERROR STOP
- exit()

Fortran

```

subroutine f()
!$ use omp_lib
  integer :: me
  me = 0
!$ me = omp_get_thread_num()
  print *, 'Hello from thread ', me
end subroutine

```

**OpenMP module:**  
explicit interfaces for API

returns an integer  
(avoid implicit typing!)

!\$ indicates statement should  
be compiled **conditionally**

C

```

#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);
}

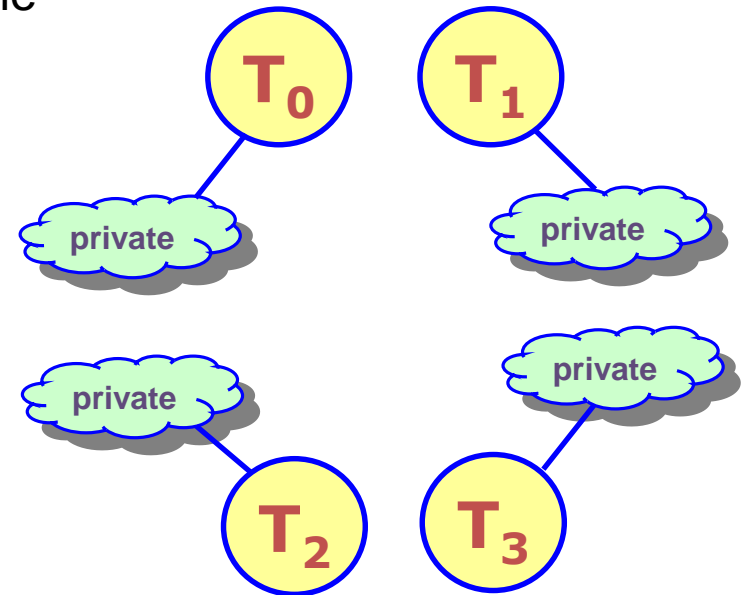
```

**OpenMP include file:**  
prototypes for API

**OpenMP-specific macro** for  
conditional compilation



- 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
```

- **Classes of routines:**

- Execution environment (36), Locking (12), Timing (2), Device Memory (7)

most commonly used subset

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

## Compilation:

Fortran

```
f90 -fopenmp -o hello.exe hello.f90
```

generic instructions ...

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

serial compilation may require stub library

## Execution:

```
export OMP_NUM_THREADS=4
./hello.exe
```

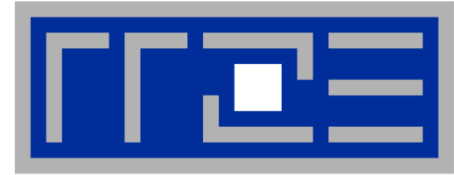
by default, parallel regions generate a team with 4 threads

## Output for example program:

```
Hello from 1
Hello from 3
Hello from 0
Hello from 2
```

ordering will vary between runs (asynchronous execution)

Now: First exercise session

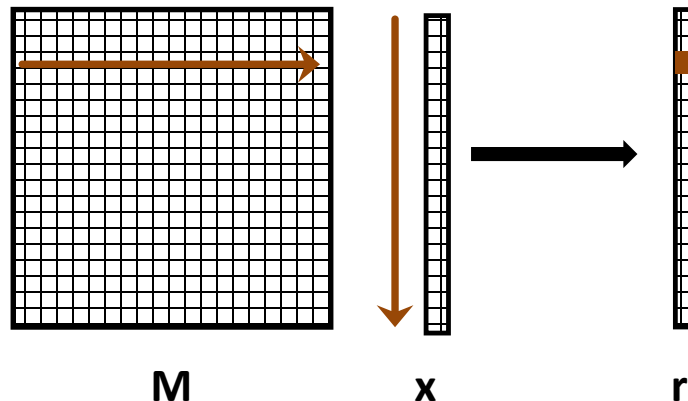


# **Simple work sharing, Scoping of Data, and Synchronization**

- **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

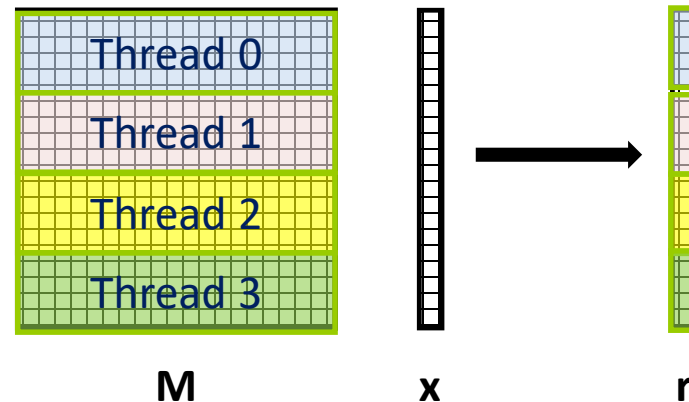
$$r = M \cdot x \text{ i.e.}$$

$$r_i = \sum_{j=1}^n M_{ij} x_j$$



A bunch of scalar products

- The idea is to split the work among threads



- **Note that**
  - all elements of  $x$  must be available to **all** threads
  - Matrix-Vector is often deployed iteratively  $\rightarrow r$  becomes  $x$  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"  $\rightarrow$  a natural concept for a shared memory programming model

```
real :: s, a(200) Fortran
```

```

s = ...
!$omp parallel shared(s,a)
  select case (me) thread ID
  case (0)
    a(1:100) = ... * s
  case (1)
    a(101:200) = ... * (-s)
  end select
!$omp end parallel

```

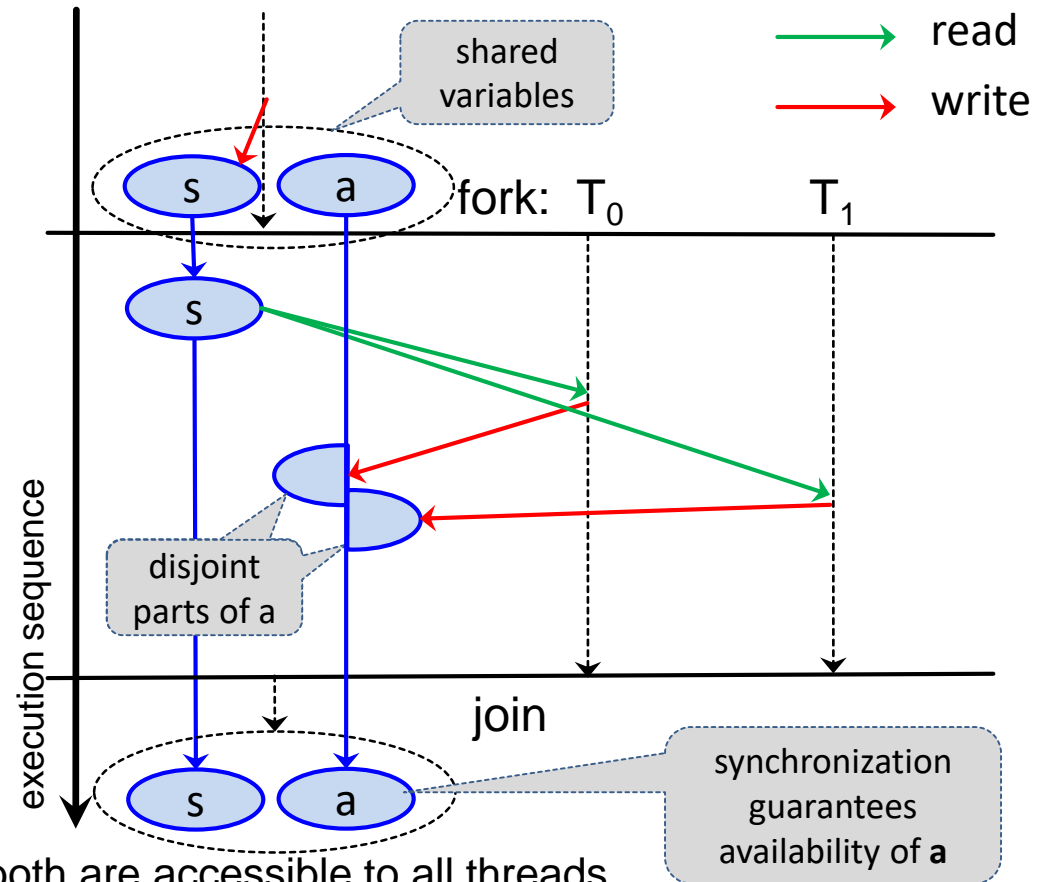
### ■ The „shared“ clause

- implies that scalar **s** and array **a** both are accessible to all threads

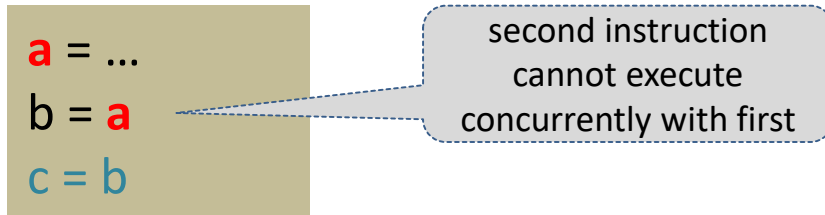
### Rules for **concurrent** accesses to a single object

- reads/writes or writes/writes by different threads are **not permitted** („data races“)

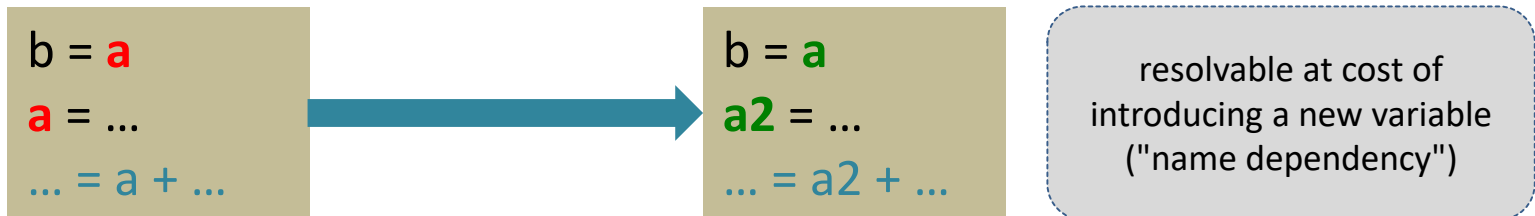
Note: updates to array **a** are OK because **disjoint parts** of object are updated



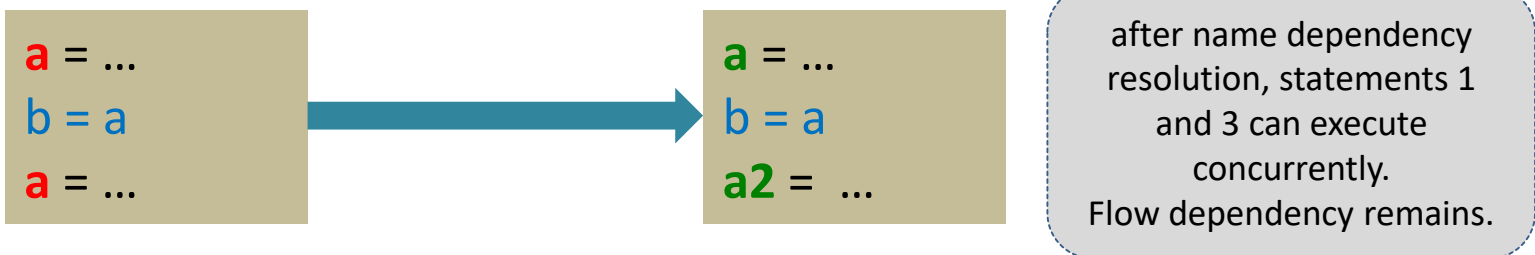
- Flow dependency ("read after write", RAW):



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



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





```

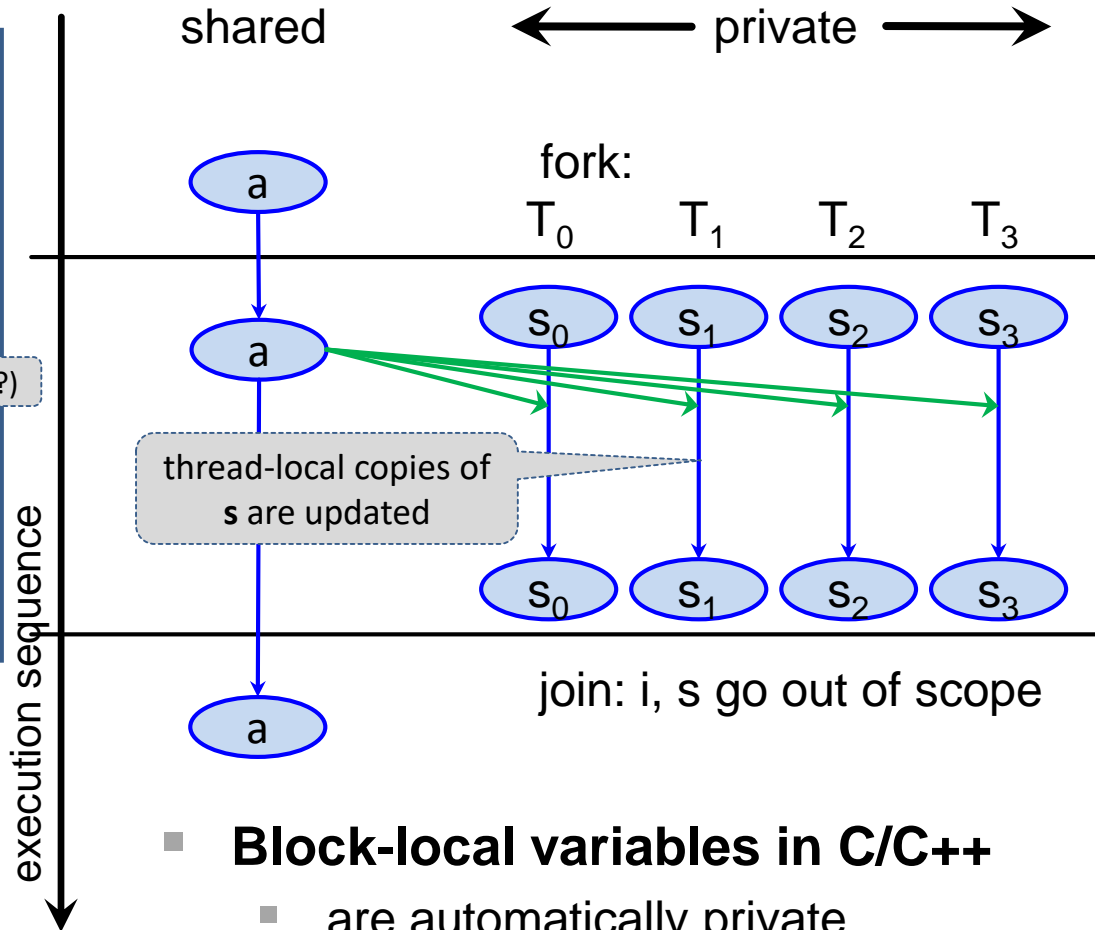
a[k] = ...;
#pragma omp parallel \
    shared(a)
{ int i; float s;
  s = 0.0;
  for (i=...; i<...; i++) {
    s += a[i];
  }
}
    
```

C

split iteration space (?)

example calculates **thread-individual** sums

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

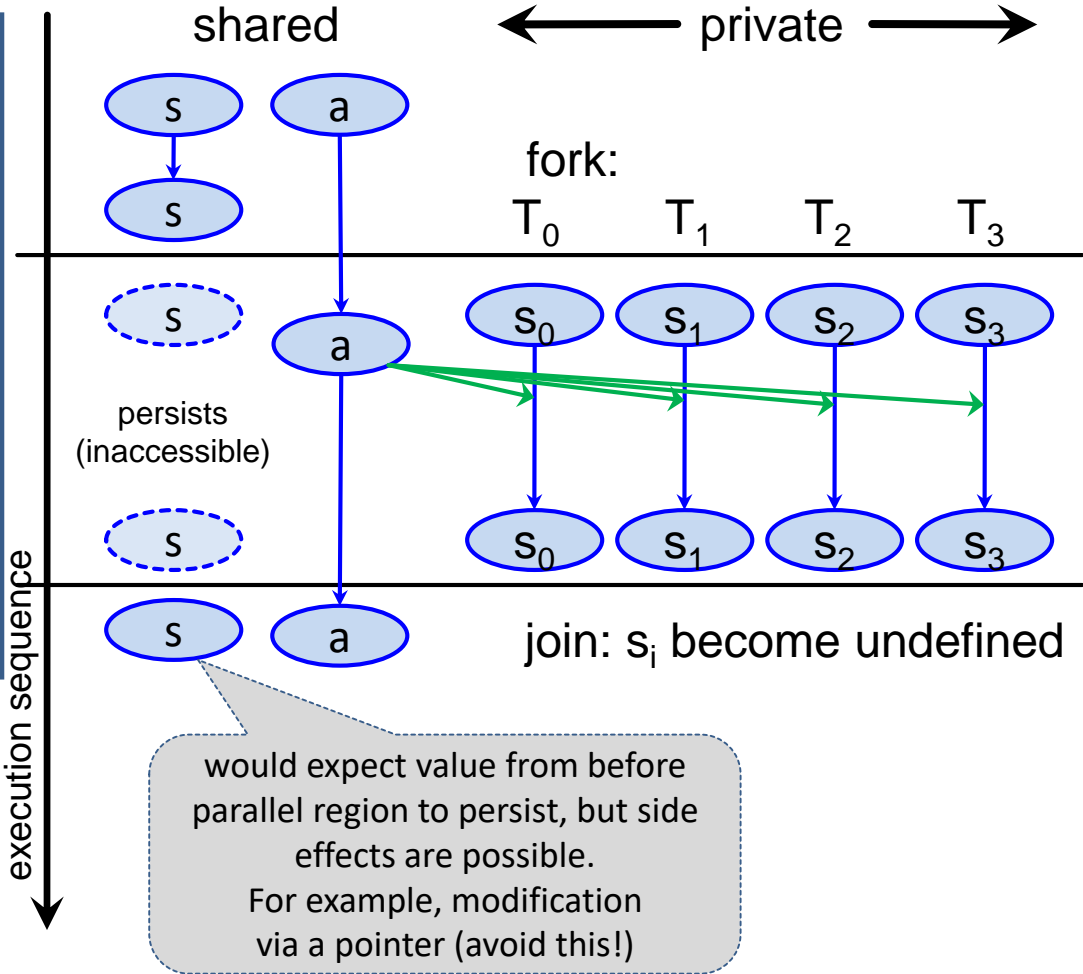


- **Block-local variables in C/C++**
  - are automatically private

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

```

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

! If *s* were shared, the program would have a race condition.

## Serial

Fortran

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

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

```
!$omp parallel
!$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
```

*r, a, x are shared by default*

*implicit barrier*

*all threads synchronize*

```
#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[...];
```

*applies to j-loop*

*j, k are private*

*no race condition against previous definitions*

- **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
  - **Disallowed:**
    - 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

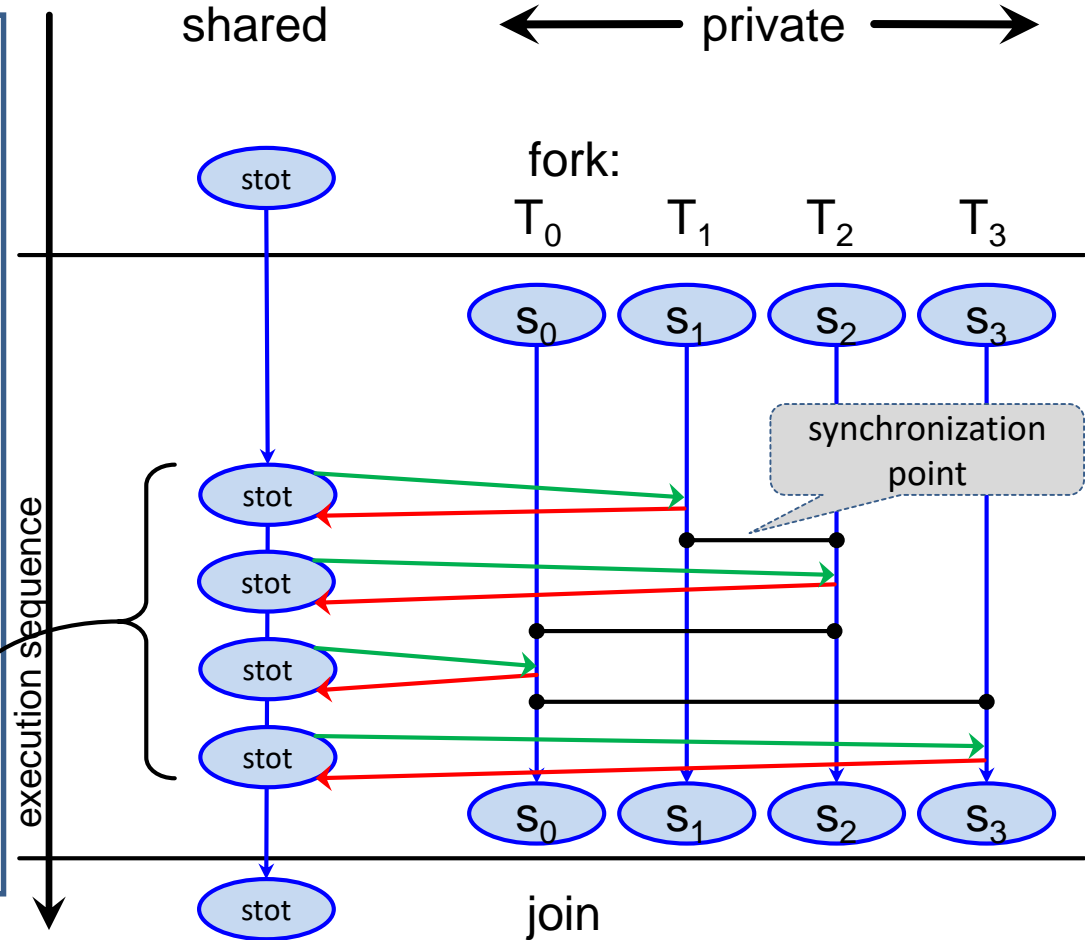
actually, we're caught between a rock and a hard place here ...

```

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
  
```

updates are now synchronized

parallel array summation



- Only one thread at a time can execute a **critical region**
  - others must wait → code in region is **effectively serialized**

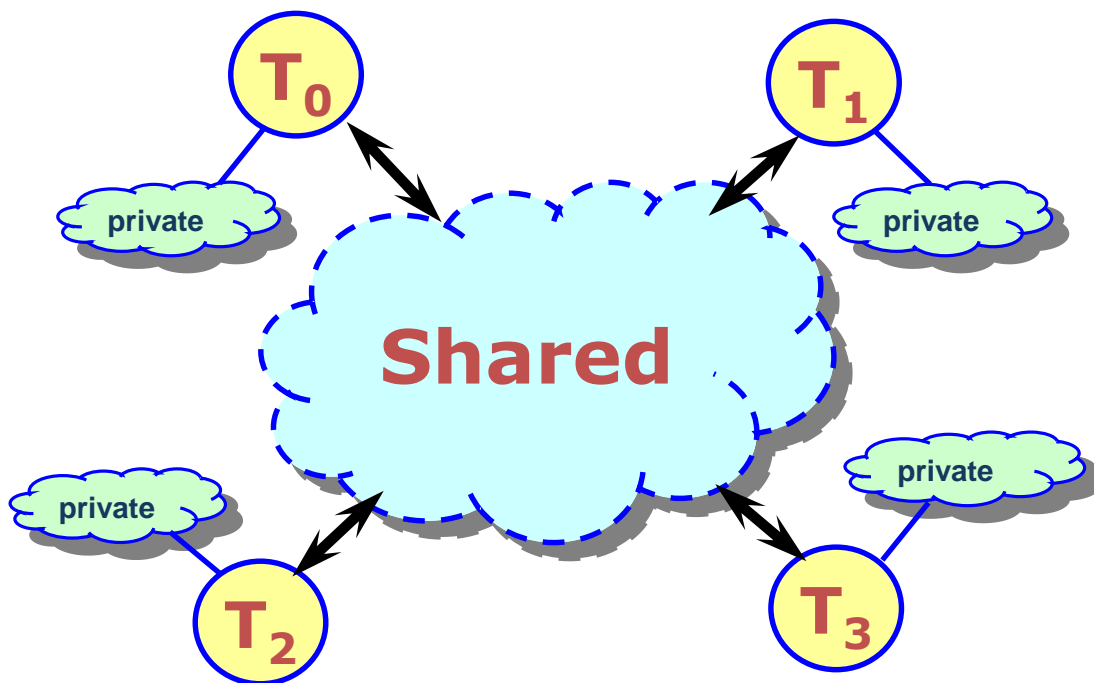
```
float stot;
stot = 0.0;
#pragma omp parallel \
    shared(a, stot)
{ int i; float s;
  s = 0.0;
#pragma omp for
  for (i=0; i<N; i++) {
    s += a[i];
  }
#pragma omp atomic update
  stot += s;
}
```

parallel array summation

legacy notation  
**omp atomic**  
is also permitted

- **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

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



- **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

```
real :: s
```

Fortran

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

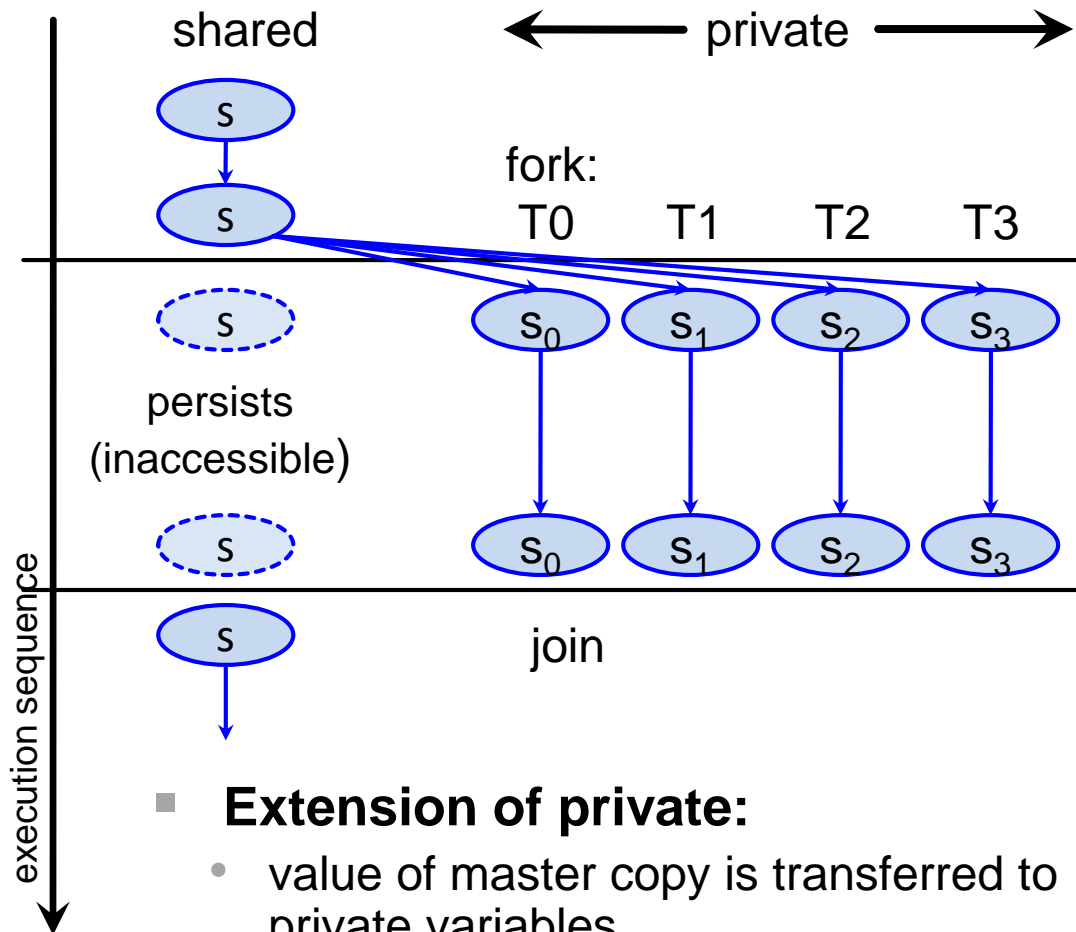
```
... = ... + s
```

uses value from  
master copy

```
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.



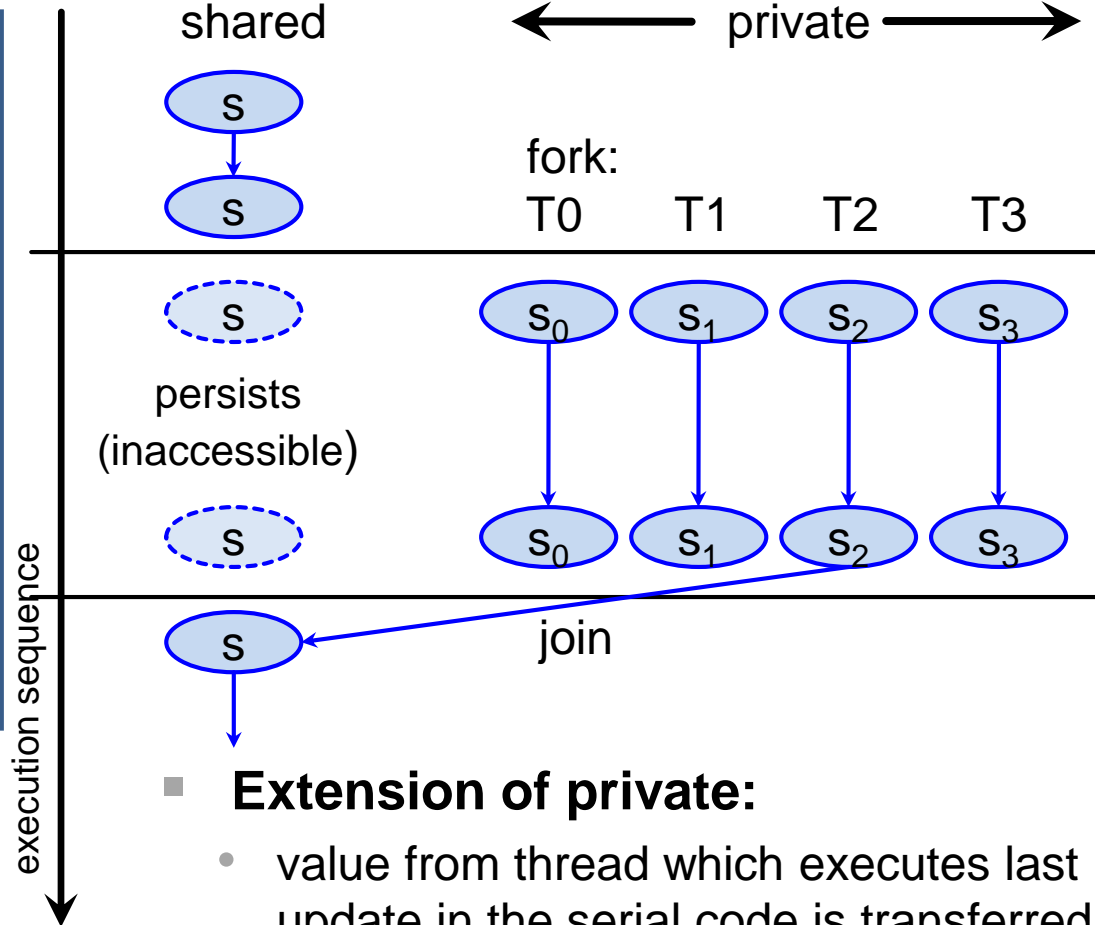
```

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

Fortran

on work sharing directive

s has value produced by i-loop iteration n



## 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**

- **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**.

this cannot be changed, of course

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

Fortran

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

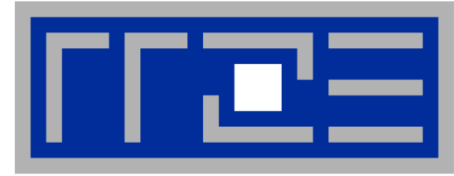
other values are possible

C

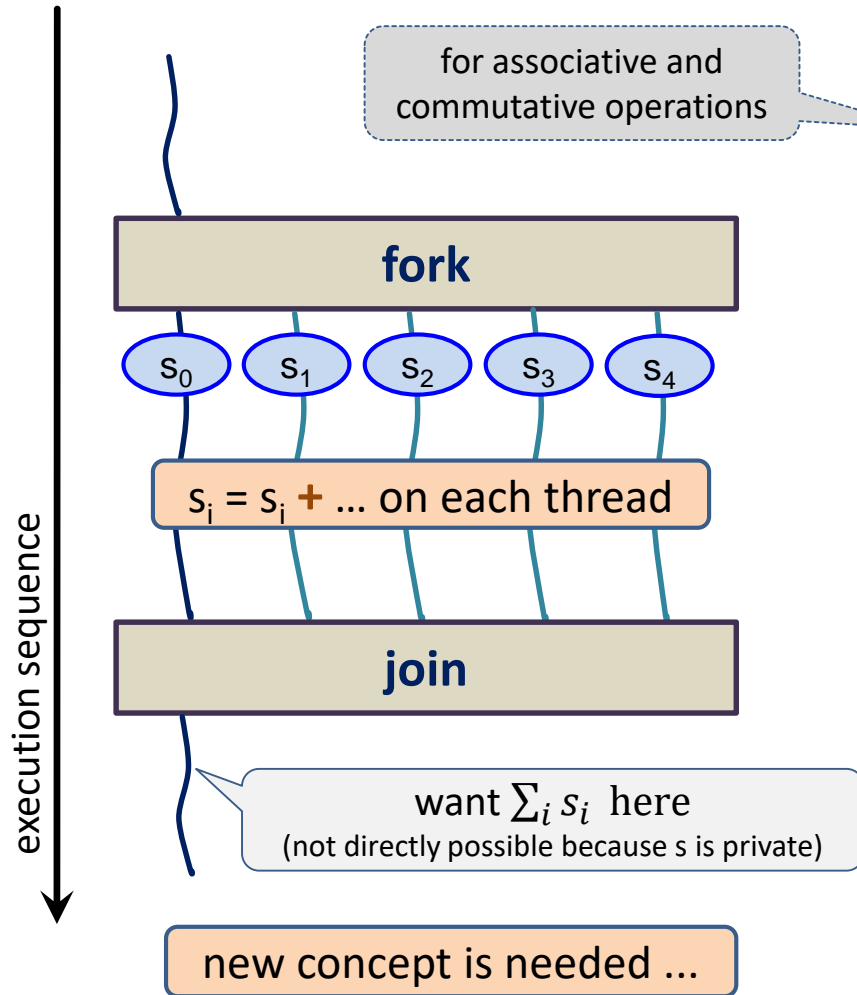
```
#pragma omp parallel default(none) \
shared(...) private(...) ...
...
```

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

Now: Second exercise session



# Reductions

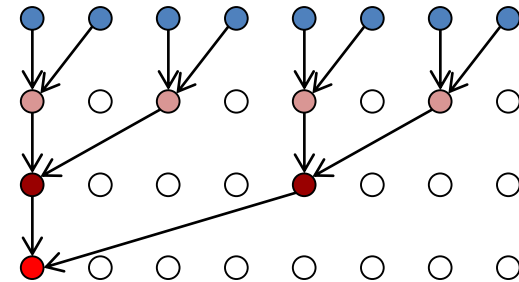


■ **Seen in previous exercise:**

- need for assembling partial results across threads
- up to now: with critical region

■ **OpenMP reductions:**

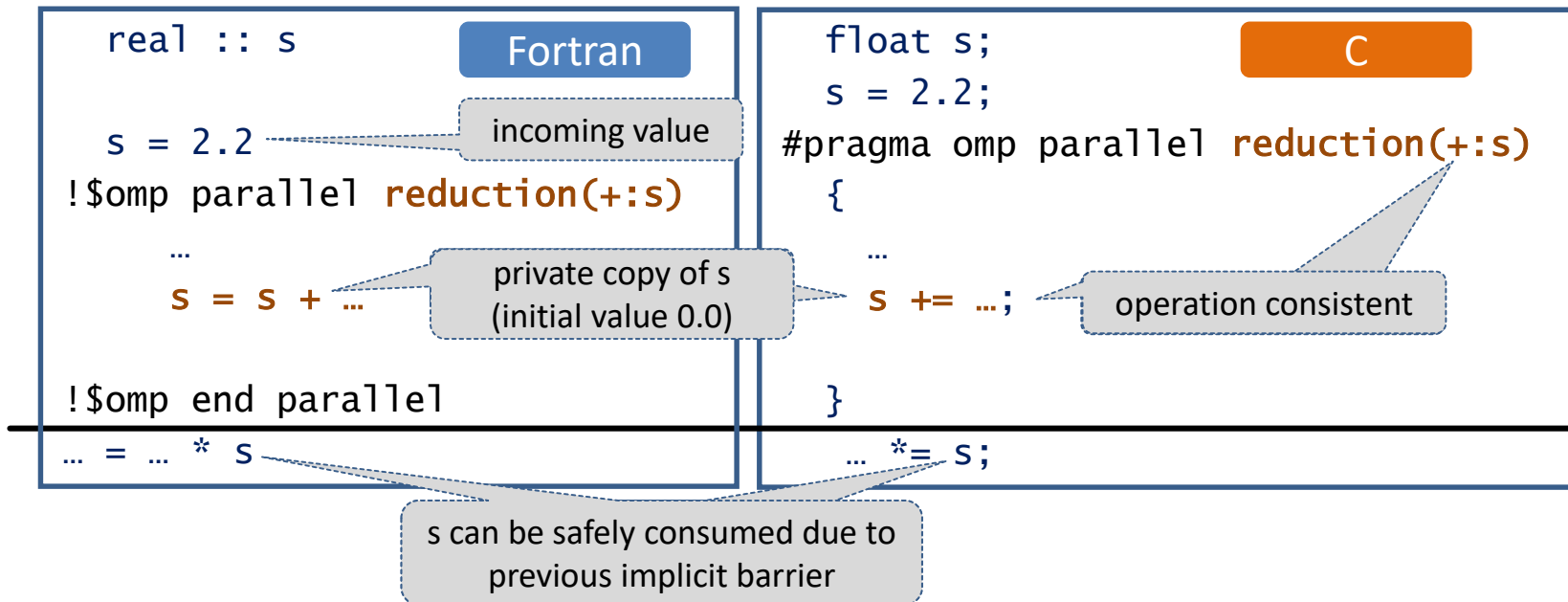
- sometimes more efficient at scale
- implementation tunings like



reduce complexity from  $O(n_{\text{threads}})$  to  $O(\log_2(n_{\text{threads}}))$

- always easier to understand and maintain

- Example 1: Sum reduction in a parallel region



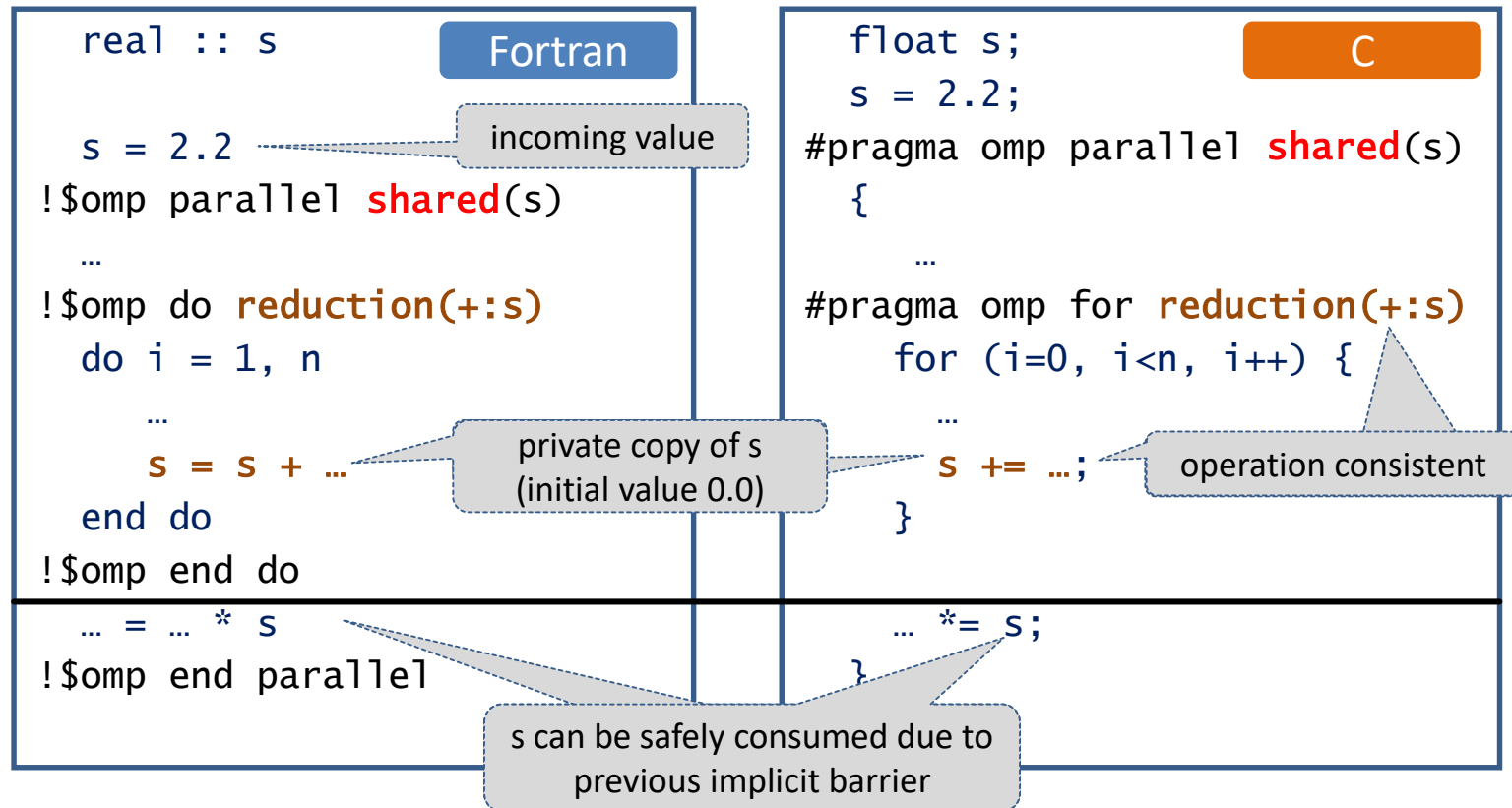
- value of s after end of parallel region:  $s_{\text{incoming}} + \sum_i s_i$

- Note: multiple reductions are permitted**

```
!$omp parallel reduction(+:x,y,z)
```

```
!$omp parallel reduction(+:x,y) &
!$omp reduction(*,z)
```

- Example 2: Sum reduction in a work shared region



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

- Depends on operation
- Supported intrinsic operations:

## Fortran

Operation	Initial value
+	0
-	0
*	1
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.
MAX	-HUGE(X)
MIN	HUGE(X)
IAND	all bits set
IEOR	all bits 0
IOR	all bits 0

## C / C++

Operation	Initial value
+	0
-	0
*	1
&	0
	0
^	0
&&	1
	0
MAX	smallest representable value
MIN	largest representable value



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

Fortran

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

must specify  
upper bound  
(assumed size)

```
float *a;
float b[N];
```

pointee created  
e.g. via malloc()

C/C++

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

same as  
b[0:N]

## Example

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

[lower bound : upper bound]

- C example does the same as the Fortran example
- OpenMP-defined sectioning syntax (differs from Fortran):

[lower bound : **length**]

## 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

no deallocate/free within reduction region



## Using derived types

Fortran

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

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

C

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

provide functions to add, etc.

## And now we want to write

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

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

- but the compiler will **refuse** to build it („+“ not known to OpenMP) unless further measures are taken ...

```
!$omp declare reduction(+:fraction:omp_out=omp_out+omp_in) &
!$omp initializer(omp_priv=fraction(0,1))
```

Fortran

```
#pragma omp declare reduction(+:Fraction: \
    Fraction_add(omp_out,omp_in)) \
    initializer(omp_priv=Fraction{0,1})
```

C

## ■ Combiner

```
declare reduction(<op>:<type>:<combiner>)
```

- connects to operator implementation

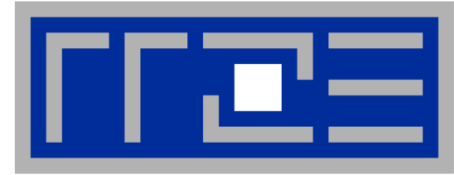
**Fortran:** example defers to overloaded „+“, **C:** references „**Fraction\_add**“  
 special OpenMP parameters **omp\_in**, **omp\_out** formally describe the two  
 operands for each operation needed

## ■ Initializer

```
initializer(omp_priv=...) or initializer(function(...))
```

- 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

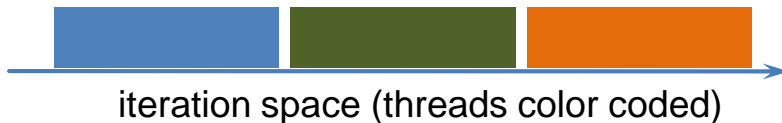
**Loops and loop scheduling**

**Collapsing loop nests**

**Parallel sections**

## Default scheduling:

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



## User-defined scheduling:

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

**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)
```



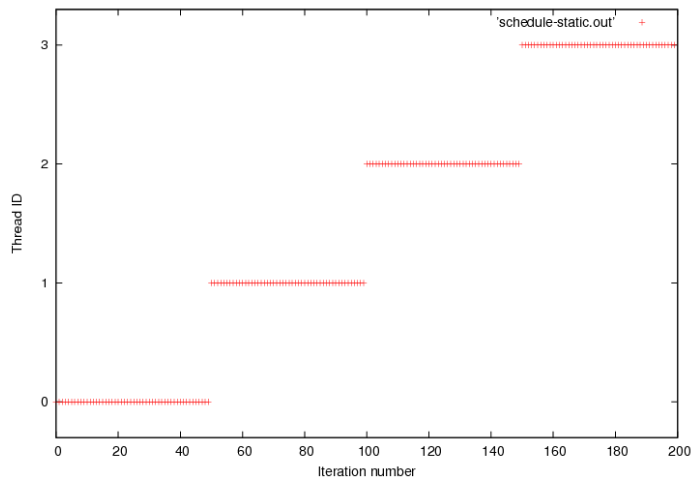
both threads take long to complete their chunk (workload imbalance)

- synchronization **overhead**
- default chunk value is **1**

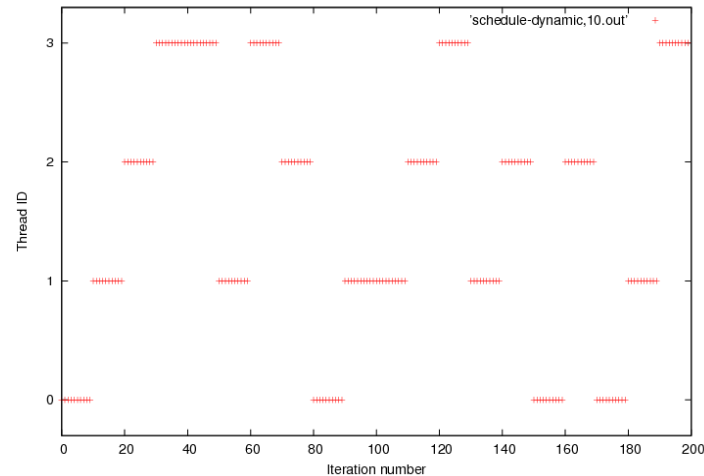
- **Size of chunks in dynamic schedule**
  - too small → large overhead
  - too large → load imbalance
- **Guided scheduling: dynamically vary chunk size.**
  - Size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to chunk-size. (default: → 1)
- **Chunk size:**
  - means minimum chunk size (except perhaps final chunk)
  - default value is **1**



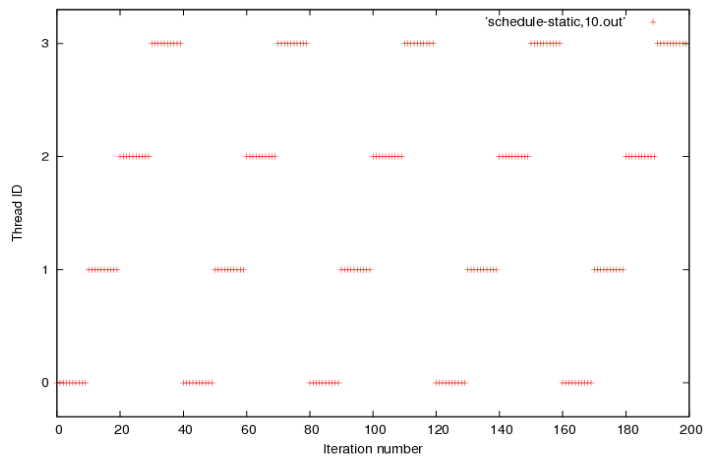
- both dynamic and guided scheduling are useful for handling **poorly balanced and unpredictable** workloads.



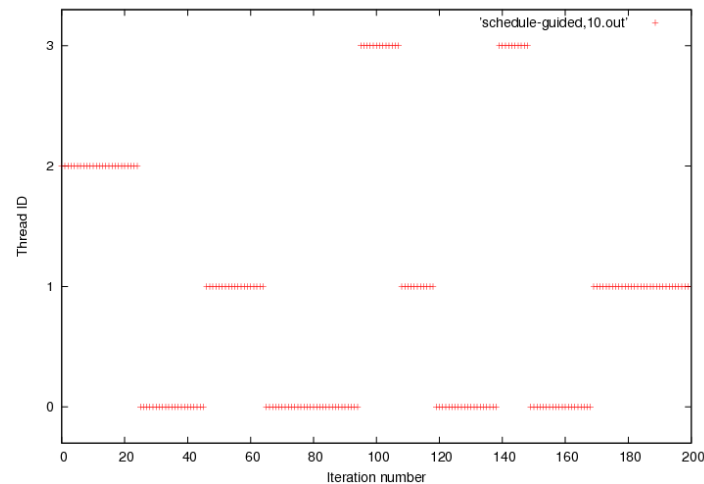
OMP\_SCHEDULE=static



OMP\_SCHEDULE=dynamic,10



OMP\_SCHEDULE=static,10



OMP\_SCHEDULE=guided,10

### Decided at run time:

```
Fortran auto  
!$OMP do schedule( runtime )
```

- **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 (...) {  
        ...  
    }  
}
```

C

- **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



### Example: Two nested loops

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

Fortran

- 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:

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

specify nesting level to collapse

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

$I_{coll}$	0	1	2	3	4	5
J	1	2	3	1	2	3
K	1	1	1	2	2	2

sequenced by serial execution order

### Restrictions:

- rectangular iteration space
- CYCLE/continue in innermost loop only

■ **Example:**

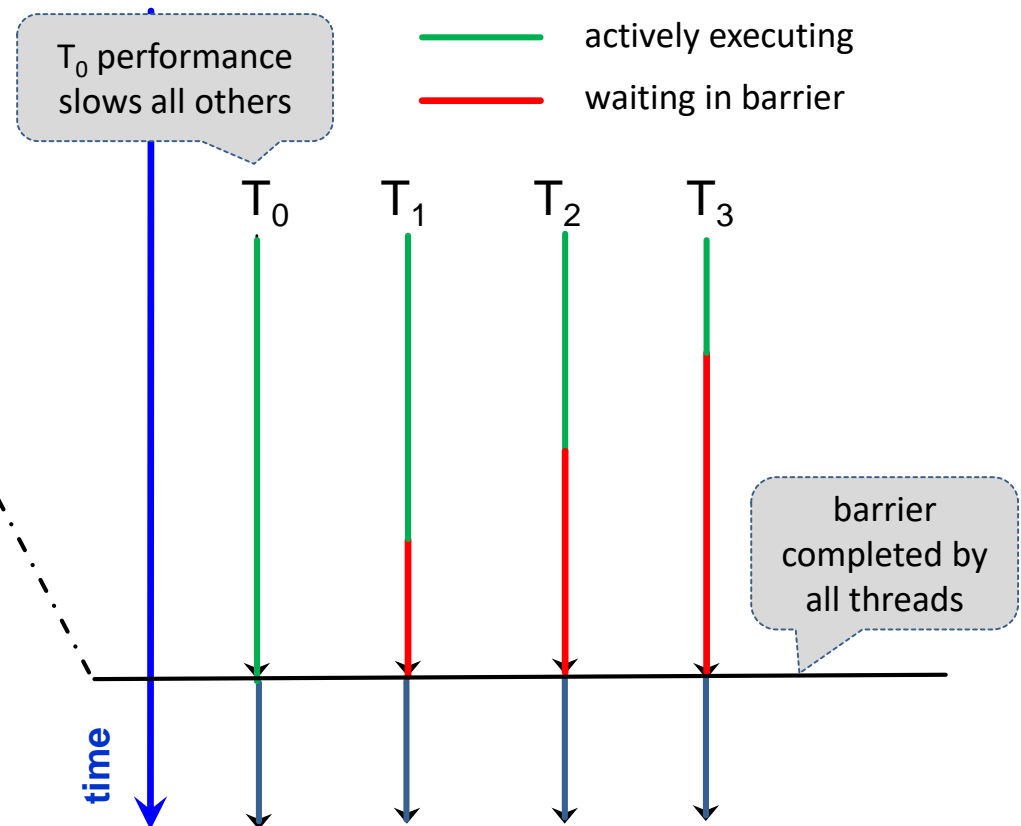
```
!$omp parallel
!$omp do reduction(+:tsum)
  do k=1, kmax
    tsum = tsum + foo(a, b, c)
  end do
!$omp end do
```

implicit  
barrier

...  
... = tsum ...

```
!$omp end parallel
```

Fortran



■ **Assumptions** on code following the synchronization point:

- does not involve **tsum**
- has a load imbalance that is inverse to that of preceding code block

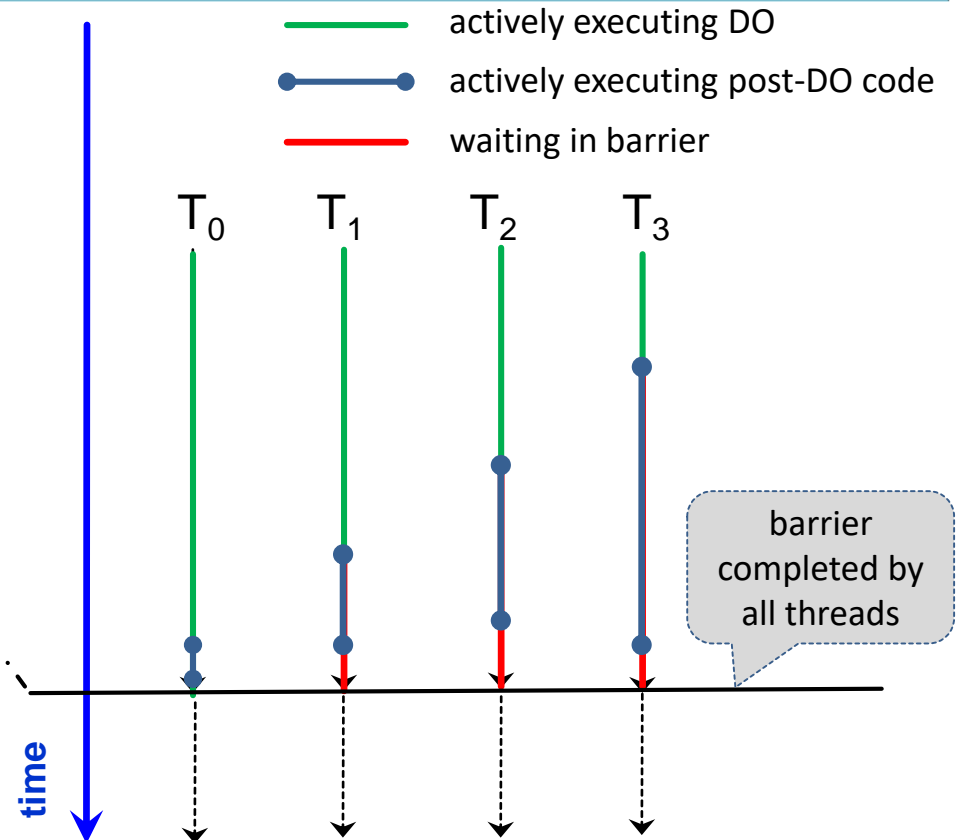
```

!$omp parallel
!$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

```

Fortran

- **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



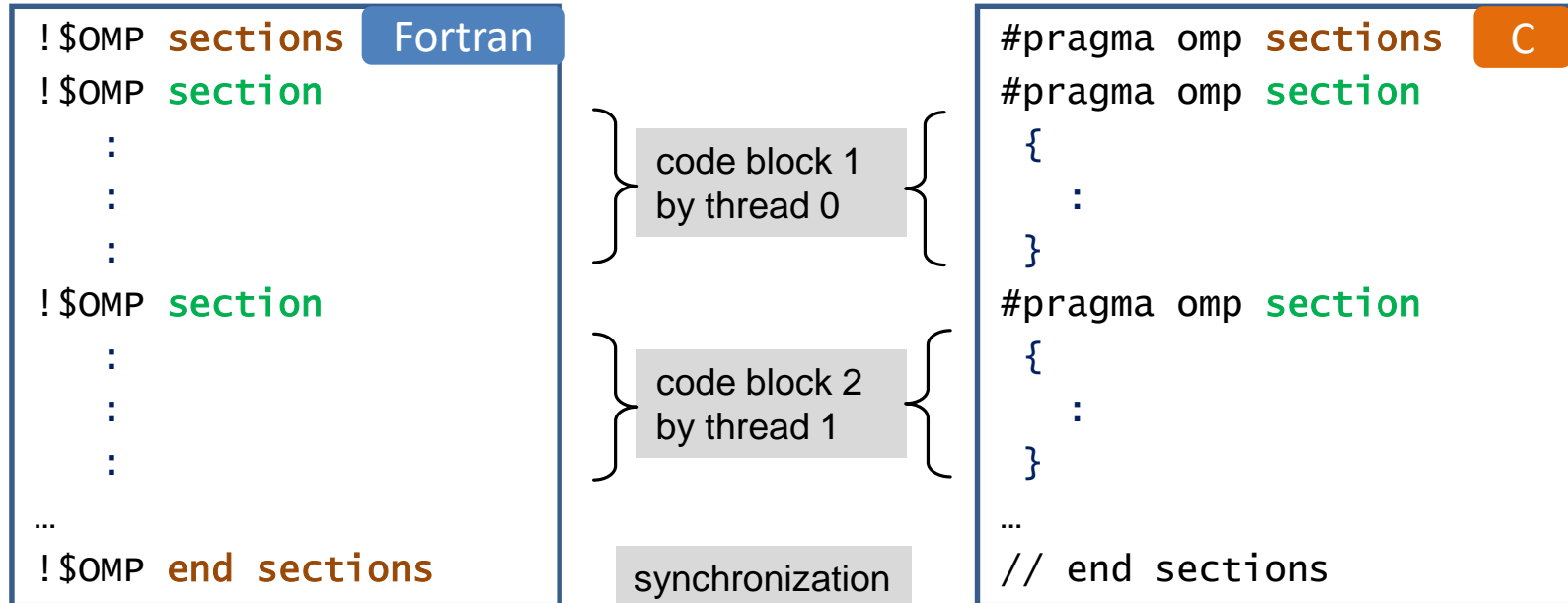
```

#pragma omp for reduction(+:tsum) \
nowait
{ ... }

```

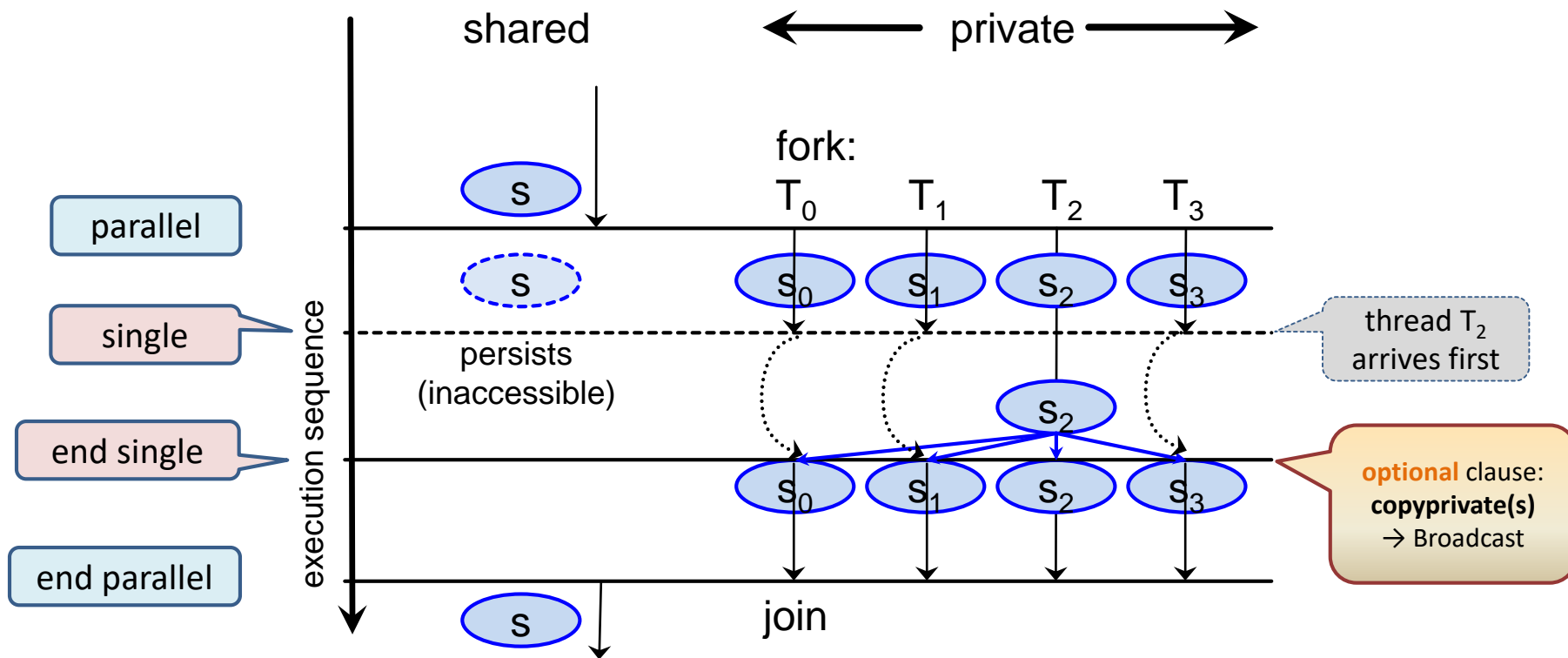
C

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



- each block is executed **exactly once** by one of the threads in the team
- **Allowed clauses on sections:**
  - private, first/lastprivate, reduction, nowait

- **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

```

real :: s
    s = ...
!$omp parallel private(s)
!$omp single
    ...
    s = ...
!$omp end single &
!$omp copyprivate(s)
    ... = ... + s
!$omp end parallel
  
```

Fortran

block executed by  
one thread only

```

float s;

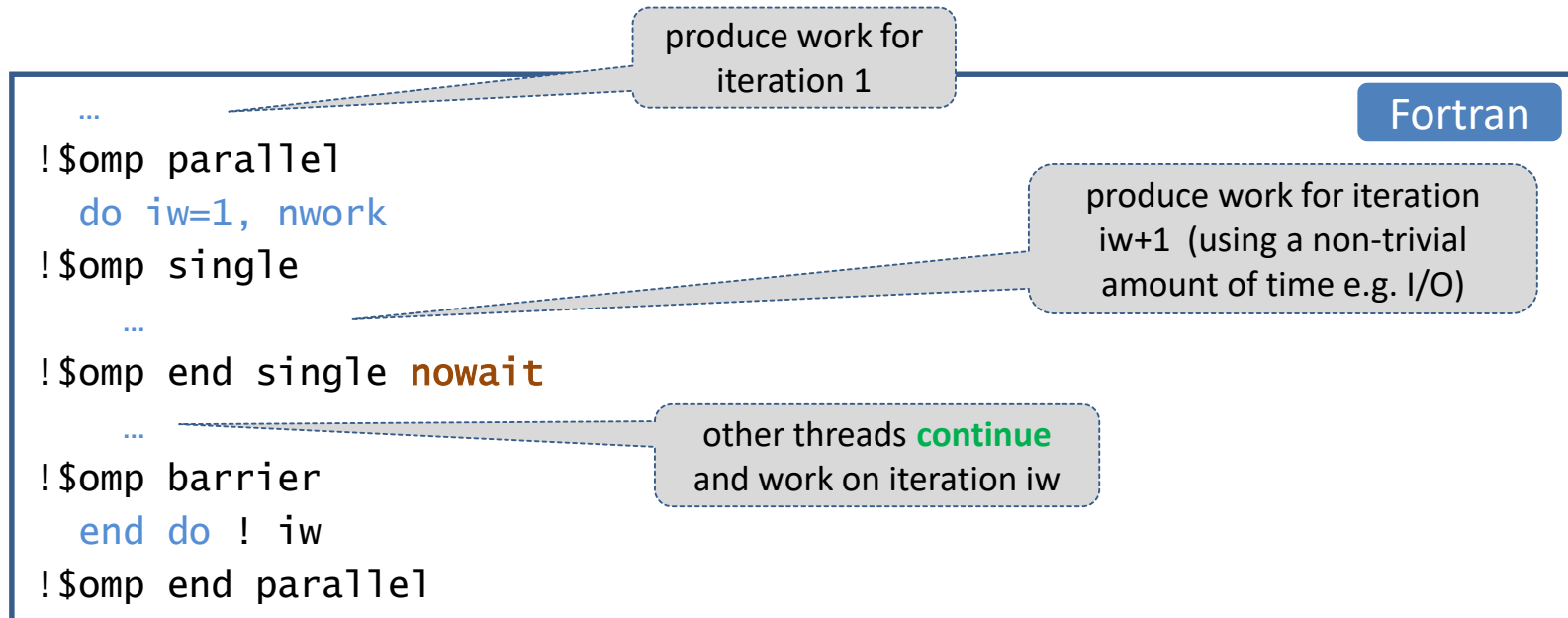
    s = ...;
#pragma omp parallel private(s)
{
#pragma omp single \
copyprivate(s)
{
    ...;
    s = ...;
} // end single
... = ... + s;
} // end parallel
  
```

C

## ■ Note:

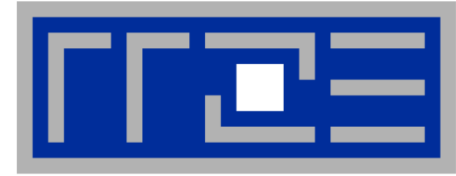
- update of shared variables inside a single block is safe against subsequent accesses, due to synchronization at the end of that block

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



- not the most efficient method  
→ preferably use tasking (covered tomorrow); the single construct will be relevant in that context





# Global variables and threading

- **Examples:**

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

Fortran

```
REAL :: A(1000)
INTEGER :: MY_COUNT
COMMON / MY_GLOBS / A, MY_COUNT
```

FORTRAN 77

```
#define NMAX 1000
float a[NMAX];
void my_func() {
  extern float a;
  ...
}
```

C

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

**Implication:**

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

directive placed in declaring  
program unit

## ▪ Copyin clause

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

```
allocate( a(ndim) )
```

Fortran

```
a(:) = ...
```

```
!$omp parallel copyin(a)
```

```
... = a(i) + ...
```

```
a(i) = ...
```

```
!$omp end parallel
```

uses value set on master

## ▪ 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**

Fortran

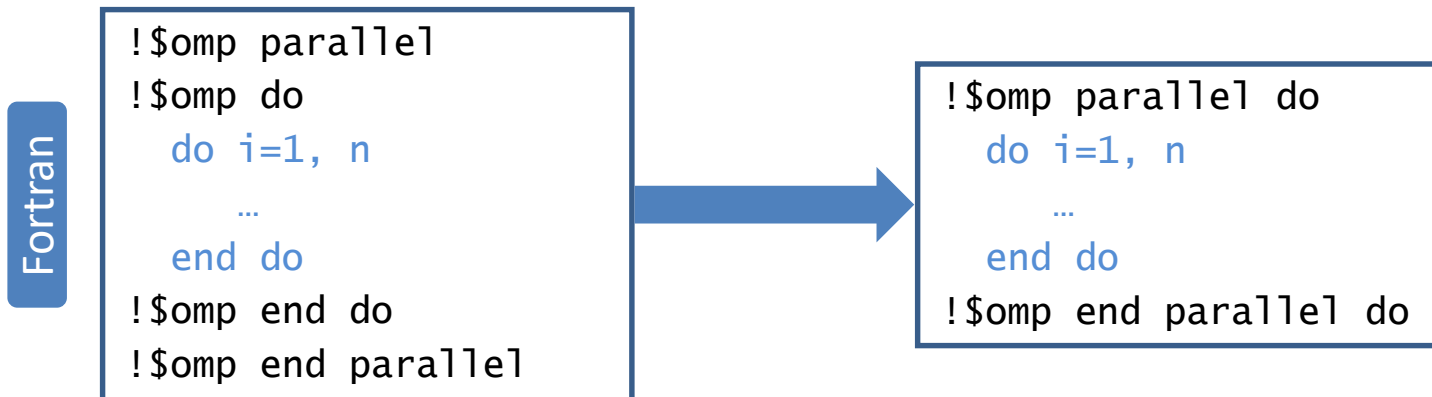
```
!$omp master  
    block  
!$omp end master
```

C

```
#pragma omp master  
    { block }
```

- **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

- **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 ...)**



- both variants have the same semantics

- Put an "if" clause on a parallel region

```

Fortran
!$omp parallel if (n > 8000)
...
!$omp end parallel
  
```

process work item of size  $O(n^p)$

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

- Specific uses:

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

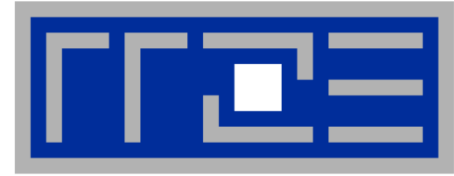
```

#pragma omp parallel if \
( ! omp_in_parallel() )
{
...
}
  
```

logical / int function from OpenMP run time: are we already parallel in executing scope?

Now: Third exercise session





# OpenMP 4.0

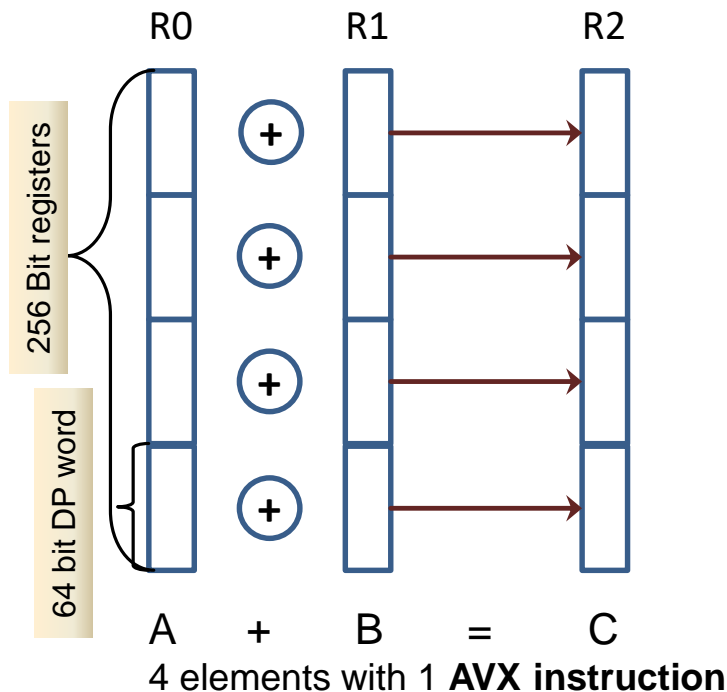
## SIMD (vectorization) directives

**Optimization of innermost  
loop structures**

Acknowledgment is due to M. Klemm (Intel)

### ■ 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

- ... 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

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

C

which may or may not get ignored by the compiler

- **Vectorize a loop nest**

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

- **Syntax**

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



```
!$omp simd [clause[[,] clause], ...]  
do loops  
[!$omp end simd]
```

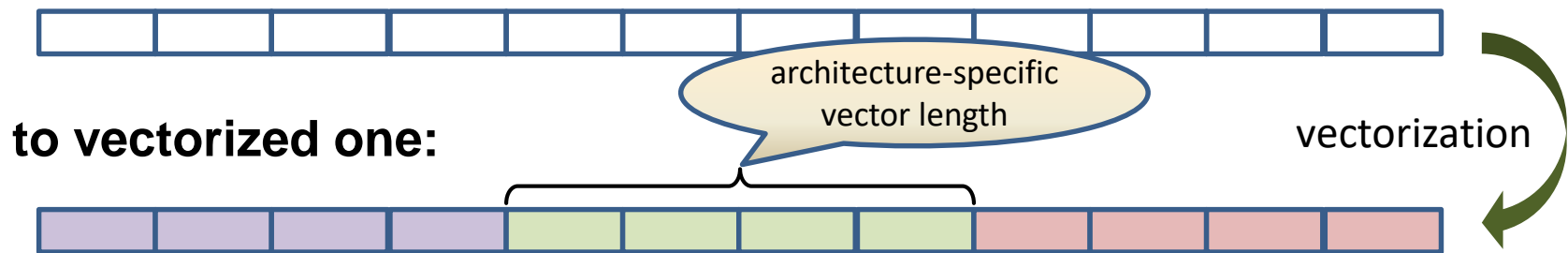


- **Scalar product**

```
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];  
    }  
}
```

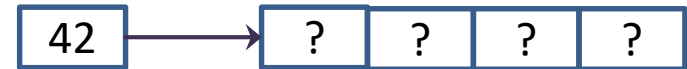
C

- **Converts serial element-wise execution**



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

- `private (var-list)`



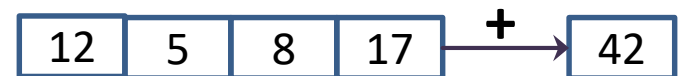
create uninitialized vectors for variables in `var-list`  
(loop iteration variables are private by default)

- `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



## ■ safelen (length)

- maximum distance between iterations that can run concurrently without breaking any dependencies

```
#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

## ■ linear (list[:linear-step])

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

- `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

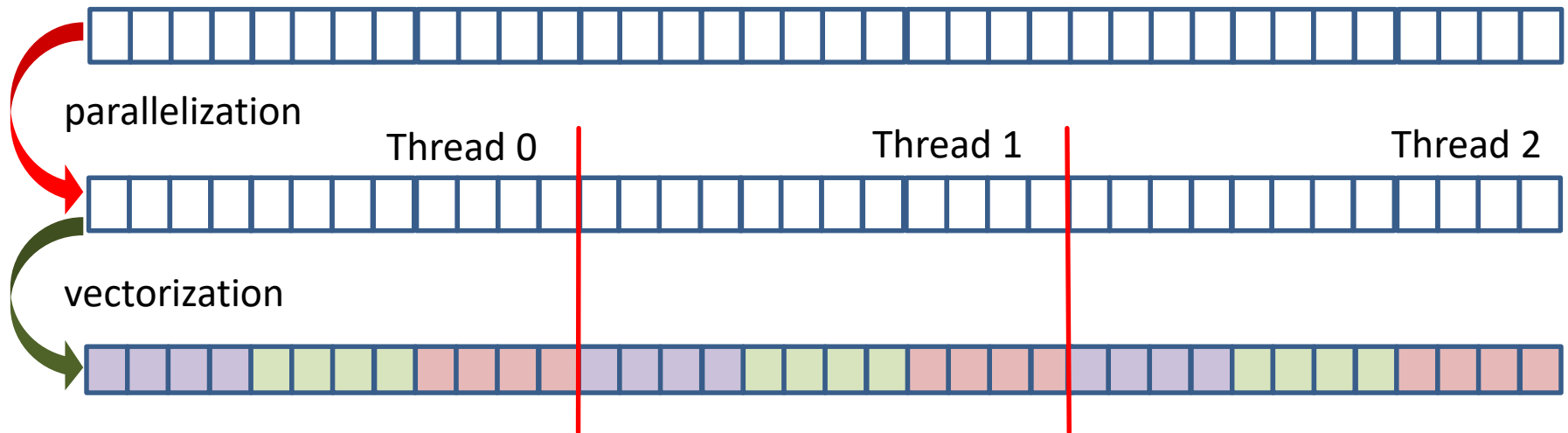
```
#pragma omp for simd [clause[[,] clause], ...]  
for loops
```

Fortran

```
!$omp do simd [clause[[,] clause], ...]  
do loops  
[!$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 call inside SIMD region

```
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] );  
    }  
}
```

may fail if functions  
outside file scope

- Therapy: explicitly declare for use in vectorized loops

- C/C++ syntax

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

- Fortran syntax

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

- clauses are also supported
- causes generation of multi-version code by the compiler

- vectorized versions of generated functions are shown

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

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

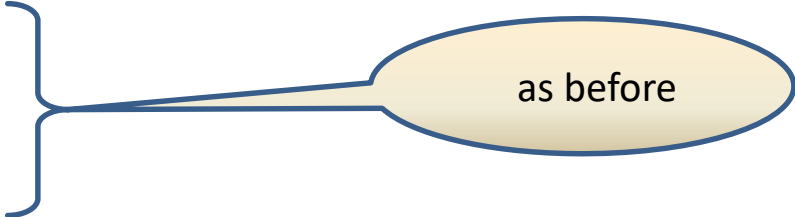
```
#pragma omp declare simd  
float distsq(float x, float y) {  
    return (x - y)*(x - y);  
}
```

```
vec8 distsq_v(vec8 x, vec8 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] );  
    }  
}
```

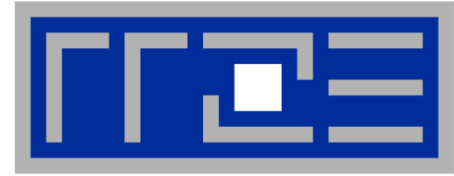
no SIMD directives permitted  
inside vectorized functions!

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

- `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)`
- 
- as before

- **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**

## Scenario:

```

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
  
```

fix number of threads  
for parallel execution

Fortran

- 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

possible results  
in first **write**

Thread 0	Thread 1
1	1
2	1
1	2

possible results in second  
**write**: 1 or 2



## ■ 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)

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

recommend to **avoid** use of explicit flushes

## ■ 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)

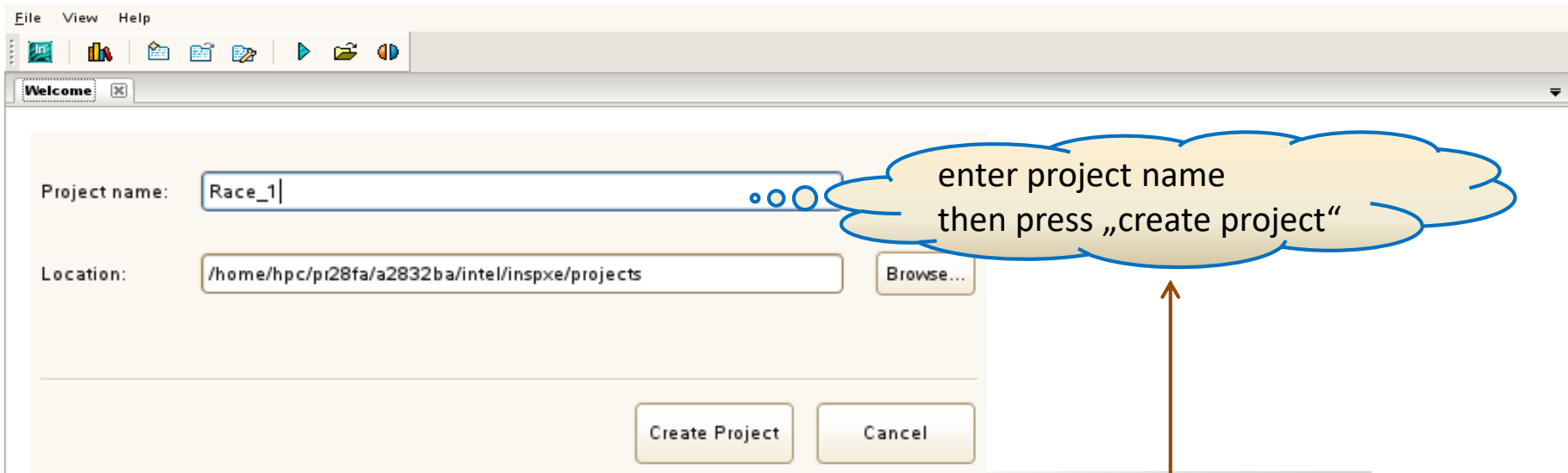
- **Example: update via critical region**
  - mutual exclusion is only assured for the statements **inside** the block i.e., subsequent threads executing the block are synchronized against each other
- **If other statements access the shared variable, you may be in trouble:**

```
!$omp parallel shared(x) ...  
:  
!$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



- **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

**Current project: heat\_tune1**

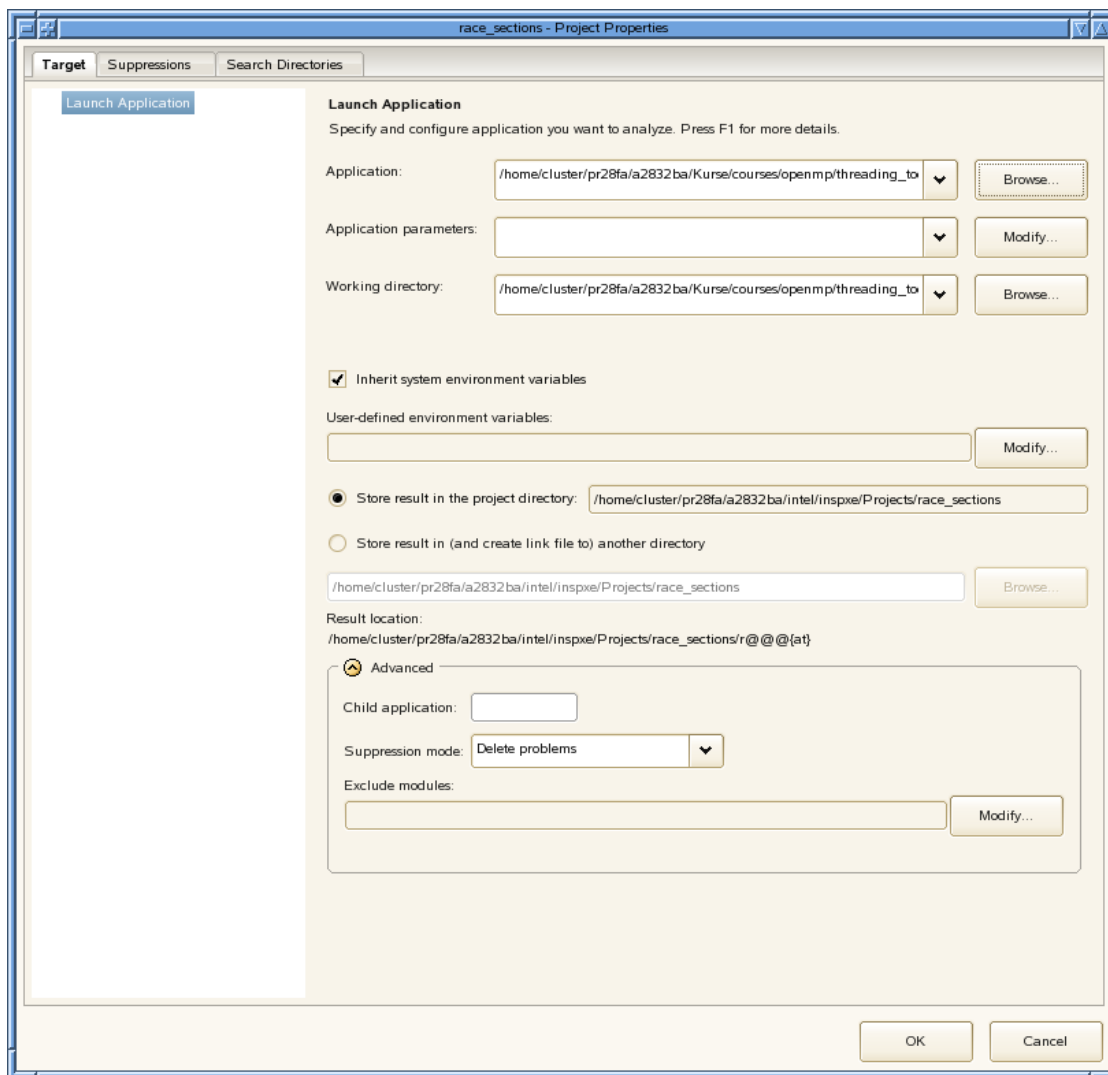
- ▶ [Threading Error Analysis / Detect Deadlocks and Data Races](#)
  - ▶ [Threading Error Analysis / Locate Deadlocks and Data Races](#)
  - ▶ [Memory Error Analysis / Detect Leaks](#)
  - ▶ [Memory Error Analysis / Locate Memory Problems](#)
  - ▶ [New Analysis...](#)
- [New Project...](#)
- [Open Project...](#)

**Recent Projects:**

- > [tasked\\_integral\\_c](#)
- > [tasked\\_integral](#)
- > [demo\\_kurs](#)
- > [fp](#)

**Recent Results:**

- > [r004ti3 \[heat\\_tune1\]](#)
- > [r003ti3 \[heat\\_tune1\]](#)
- > [r002ti3 \[heat\\_tune1\]](#)
- > [r001ti3 \[heat\\_tune1\]](#)
- > [r000ti3 \[heat\\_tune1\]](#)



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

File View Help

Welcome i000t1 New Inspector Result

Configure Analysis Type Intel Inspector XE 2013

Analysis Type

Threading Error Analysis

10x-40x Detect Deadlocks  
20x-80x Detect Deadlocks and Data Races  
40x-160x Locate Deadlocks and Data Races

Analysis Time Overhead Memory Overhead

Locate Deadlocks and Data Races

Copy

Widest scope threading error analysis type. Maximizes the load on the system and the time and resources required to perform analysis; however, detects the widest set of errors and provides context and maximum detail for those errors. Press F1 for more details.

Terminate on deadlock

Stack frame depth: 16

Scope: Normal

Remove duplicates

Analyze without debugger

Run an analysis and report all detected problems. Use to view correctness issues and examine them.

Enable debugger when problem detected

Run an analysis under the debugger and stop every time a problem is detected or a problem detected. Not recommended when running a threading analysis because it can significantly impact performance.

Select analysis start location with debugger

Run target application under the debugger with analysis disabled until you choose the application entry point. Set a code breakpoint to stop execution prior to when you reach this point, then use the 'monitor begin-analysis' command to turn on analysis for the duration of the analysis.

Details

Detect deadlocks:	Yes
Terminate on deadlock:	No
Detect lock hierarchy violations:	Yes
Save stack on lock creation:	Yes
Cross-thread stack access detection:	Hide problems/Show warnings

Start

Stop

Reset Leak/Growth Detection

Show Leaks/Growth Now

Close

Project Properties...

Command Line...



## Select analysis mode, then start

- here: Threading Error Analysis → locate deadlocks and data races
- note potentially high performance impact

File View Help

Project Navigator

/home/hpc/pi28fa/a2832ba/intel/inspxe/projects

- Race\_1
- Race\_2
- demo\_kurs
- heat\_tune1
- race\_on\_atomic
  - r000h3
- tasked\_integral
- tasked\_integral\_c

Welcome r000h3

### Locate Deadlocks and Data Races

Intel Inspector XE 2016

Target Analysis Type Collection Log Summary

Problems

ID	Type	Sources	Modules	State
P1	Data race	race.f90	race.exe	New

Filters

Severity	Count
Error	1 item(s)

Type	Count
Data race	1 item(s)

Source	Count
race.f90	1 item(s)

Module	Count
race.exe	1 item(s)

State	Count
New	1 item(s)

Suppressed	Count
Not suppressed	1 item(s)

Investigated	Count
Not investigated	1 item(s)

a race condition was identified

Code Locations: Data race

Description	Source	Function	Module
Read	race.f90:14	MAIN_\$omp\$parallel@9	race.exe
<pre> 12 x = x + y 13 !\$omp end critical 14 z = z + f(x, z) 15 !\$omp end parallel 16 write(*, '( "Result is ", E12.3)') z </pre>			
Write	race.f90:12	MAIN_\$omp\$parallel@9	race.exe
<pre> 10 y = omp_get_thread_num() 11 !\$omp critical 12 x = x + y 13 !\$omp end critical 14 z = z + f(x, z) </pre>			

Timeline

- MP Worker: Thread #2 (15628)
- MP Worker: Thread #3 (15629)



## Note:

- requires debug option for compiled code

Intel Inspector XE 2016

**Data race**

Read - Thread OMP Worker Thread #3 (15629) (race.exe!MAIN\_\_\$omp\$parallel@9 - race.f90:14)

```
5 integer :: i
6
7 z = 0.0
8 x = 0.0
9 !$omp parallel private(y) shared(x) reduction(+:z)
10 y = omp_get_thread_num()
11 !$omp critical
12 x = x + y
13 !$omp end critical
14 z = z + f(x, z)
15 !$omp end parallel
16 write(*,('Result is ',E12.3)) z
17 contains
18 pure real function f(a, b)
19 real, intent(in) :: a, b
20 f = a
21 end function
22 end program
23
```

Call Stack: race.exe!MAIN\_\_\$omp\$parallel@9 - race.f90:14

Write - Thread OMP Worker Thread #2 (15628) (race.exe!MAIN\_\_\$omp\$parallel@9 - race.f90:12)

```
3 implicit none
4 real :: x, y, z
5 integer :: i
6
7 z = 0.0
8 x = 0.0
9 !$omp parallel private(y) shared(x) reduction(+:z)
10 y = omp_get_thread_num()
11 !$omp critical
12 x = x + y
13 !$omp end critical
14 z = z + f(x, z)
15 !$omp end parallel
16 write(*,('Result is ',E12.3)) z
17 contains
18 pure real function f(a, b)
19 real, intent(in) :: a, b
20 f = a
21 end function
22
```

Call Stack: race.exe!MAIN\_\_\$omp\$parallel@9 - race.f90:12



a) **same** shared variable

thread 0

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

thread 1

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

Fortran

critical region is **global** → OKb) **different** shared variables

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

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

Fortran

mutual exclusion not required → unnecessary loss of performance

- **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.

- **Assumption:**

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

- **Atomic read:**

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

- **Atomic write:**

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

- **Atomic capture**

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

- different ordering of statements also allowed

- **Not atomic:**

- evaluation of expressions or updates on v

- **Atomic update:**

- !\$omp atomic update
- same as „traditional“ atomic directive

## ■ 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:

```
#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

## Variant of theoretical exercise

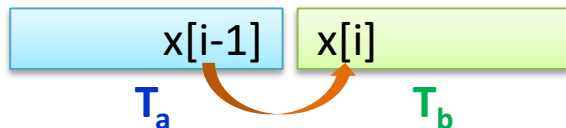
- race condition  $i-1 \rightarrow i$  on one statement

```

real :: a(n), b(n), x(n)
a = ...; b = ...; x(1) = ...
!$omp parallel shared(a,b,x)
!$omp do
do i=2, n
  b(i) = f1( a(i), b(i) )
  ⚠ x(i) = a(i) * x(i-1) + b(i)
  a(i) = f2( b(i), x(i) )
end do
!$omp end do
!$omp end parallel
  
```

Fortran

- 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

```

!$omp do ordered
do ...
  ...
!$omp ordered
  ...
!$omp end ordered
  ...
end do
!$omp end do
  
```

Fortran

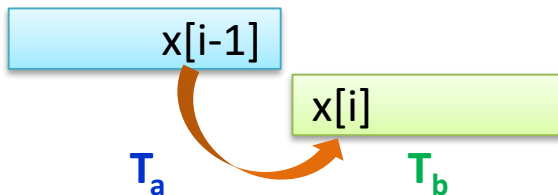
- statements in **ordered** region are executed in same order as the loop's iteration variable is increased in a serial execution
- statements outside the **ordered** region can execute in parallel
- only **one** ordered region permitted

- Assures that code block with flow dependency is effectively serialized

```
!$omp parallel shared(a,b,x)
!$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
```

Fortran

- $T_a$  signals completion of its chunk to  $T_b$  → synchronization avoids the race



- depend clause

```
#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
}
```

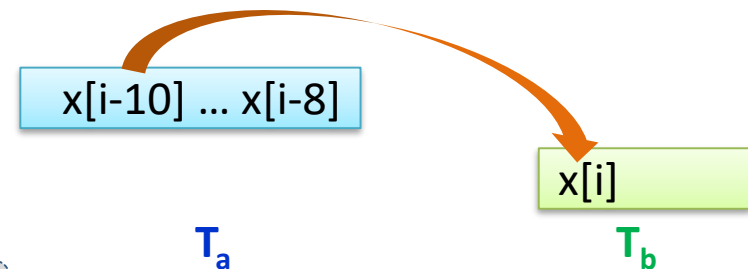
consumer  $T_b$   
must wait

supplier  $T_a$  signals  
completion

C

- Additional semantics:

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



## Ordered clause with a nesting depth specification

```
#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] = ... * a[i-1][j] + ... * a[i][j-1];
        #pragma omp ordered depend(source)
    }
}
```

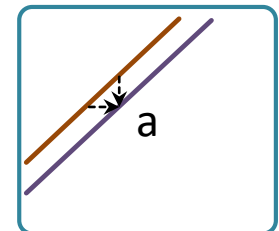
dependencies span across two nested loops

C

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

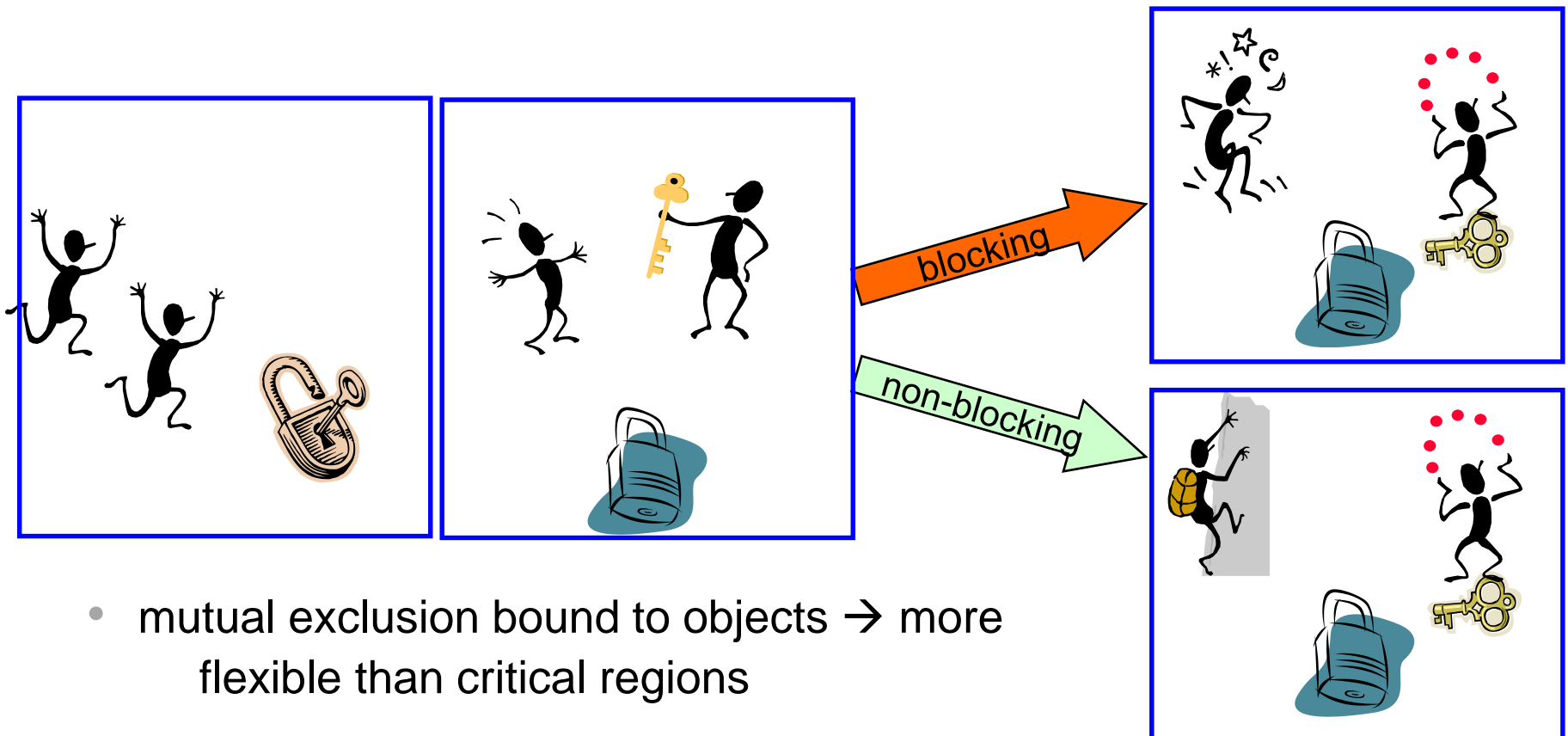


beware potential deadlocks (incorrectly specified dependencies)





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



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

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

```
use omp_lib
```

```
...
```

```
integer(omp_lock_kind) :: a_lock
```

```
integer(omp_nest_lock_kind) :: a_nestable_lock
```

typically an integer capable of  
representing an address

Fortran

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

```
...
```

```
omp_lock_t a_lock;
```

```
omp_nest_lock_t a_nestable_lock;
```

C

- **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

Name	Purpose
<code>omp_init_lock(omp_lock_t *lock)</code>	initializes an uninitialized lock; the lock variable has the state "unlocked" on return
<code>omp_destroy_lock(omp_lock_t *lock)</code>	destroys a lock that has the state "unlocked".
<code>omp_init_nest_lock (omp_nest_lock_t *lock)</code>	initializes an uninitialized nestable lock; the lock variable has the state "unlocked" on return, and its nesting count is zero.
<code>omp_destroy_nest_lock (omp_nest_lock_t *lock)</code>	destroys a nested lock that has the state "unlocked".

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

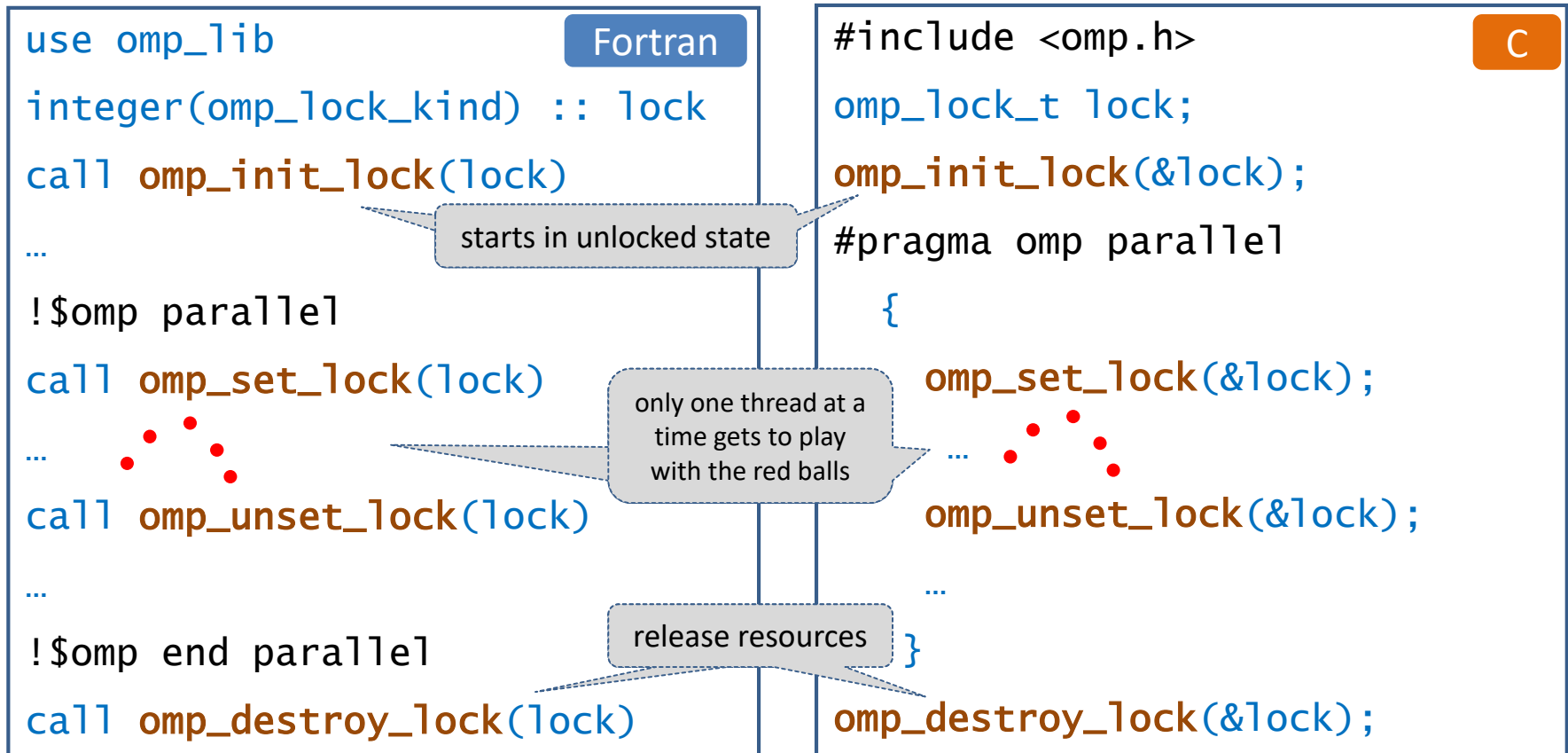
- An initialized OpenMP lock can be in one of the states **unlocked**, or **locked**
- The (unique) task region that has successfully acquired the lock is said to **own the lock**

thread + context it is executing in
- Only the task region that owns the lock can **release** it, returning it to the unlocked stage.

Name	Purpose
<code>omp_set_lock(omp_lock_t *lock)</code>	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.
<code>omp_unset_lock(omp_lock_t *lock)</code>	Release the lock that is owned by the executing task region.

- **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

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



## Function call signature

```
logical function omp_test_lock(lock)
```

Fortran

```
int omp_test_lock(omp_lock_t *lock)
```

C

- 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

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

Fortran

do stuff unrelated  
to the red balls

play with the red  
balls

```
#pragma omp parallel
{
while (! omp_test_lock(&lock)) {
...
}
...
omp_unset_lock(&lock);
}
```

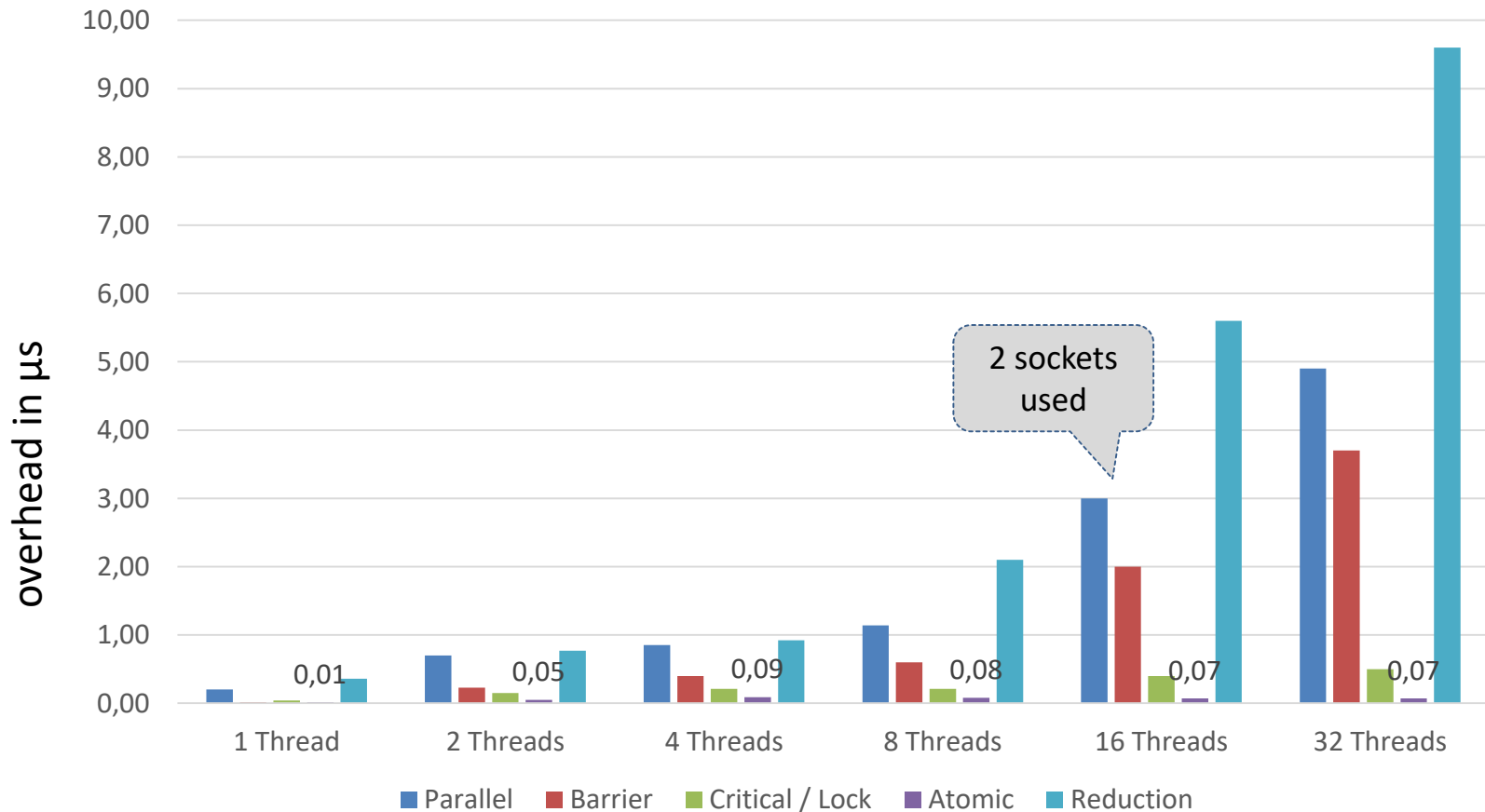
C

- **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

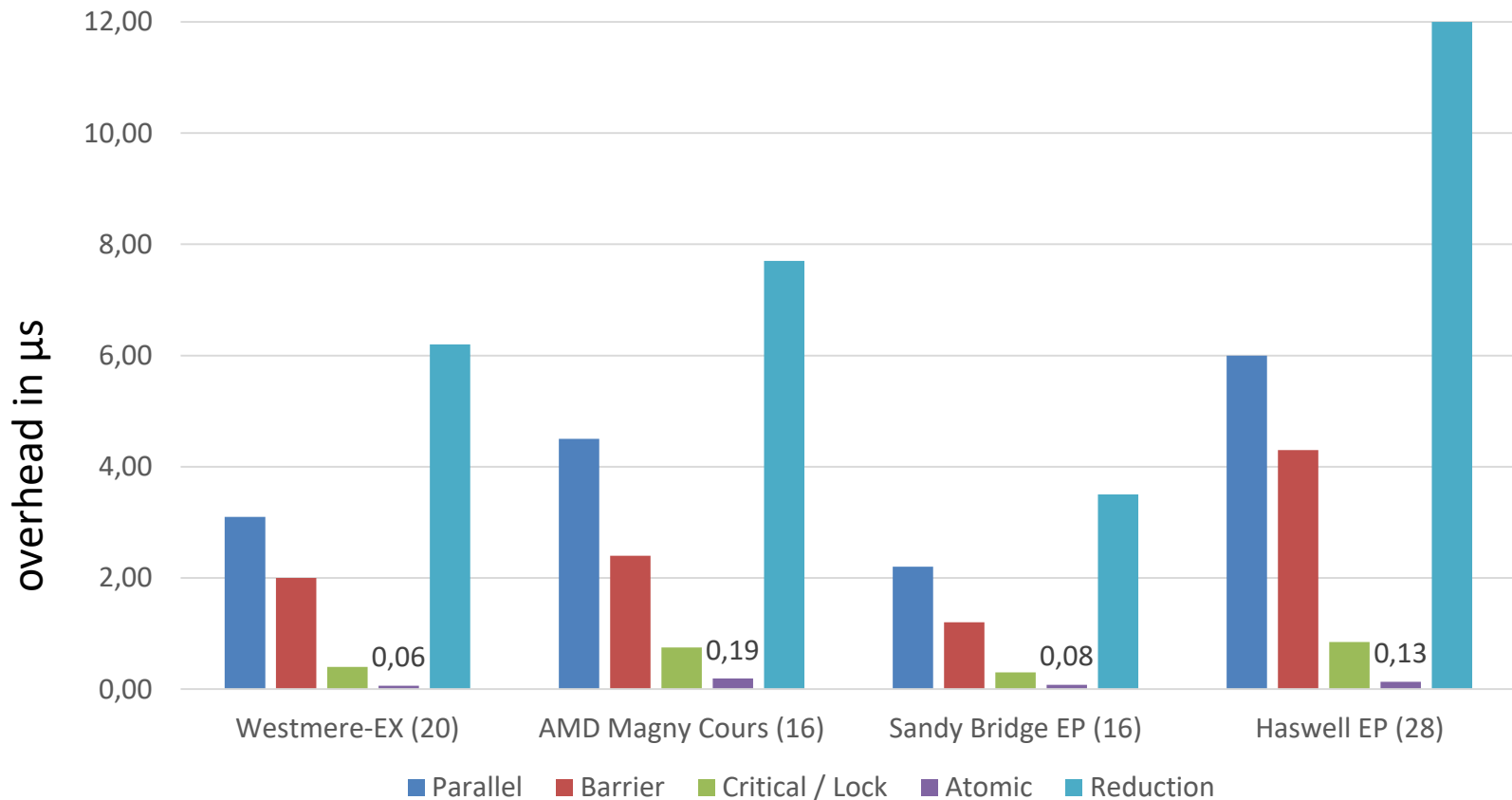
- **Syncbench from the EPCC OpenMP microbenchmarks is used**
  - evaluates the overheads for all synchronizing constructs systematically
  - overhead is what remains even if no workload is processed
- **Showing results as a function of thread count**
  - alternatively, depending on node architecture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



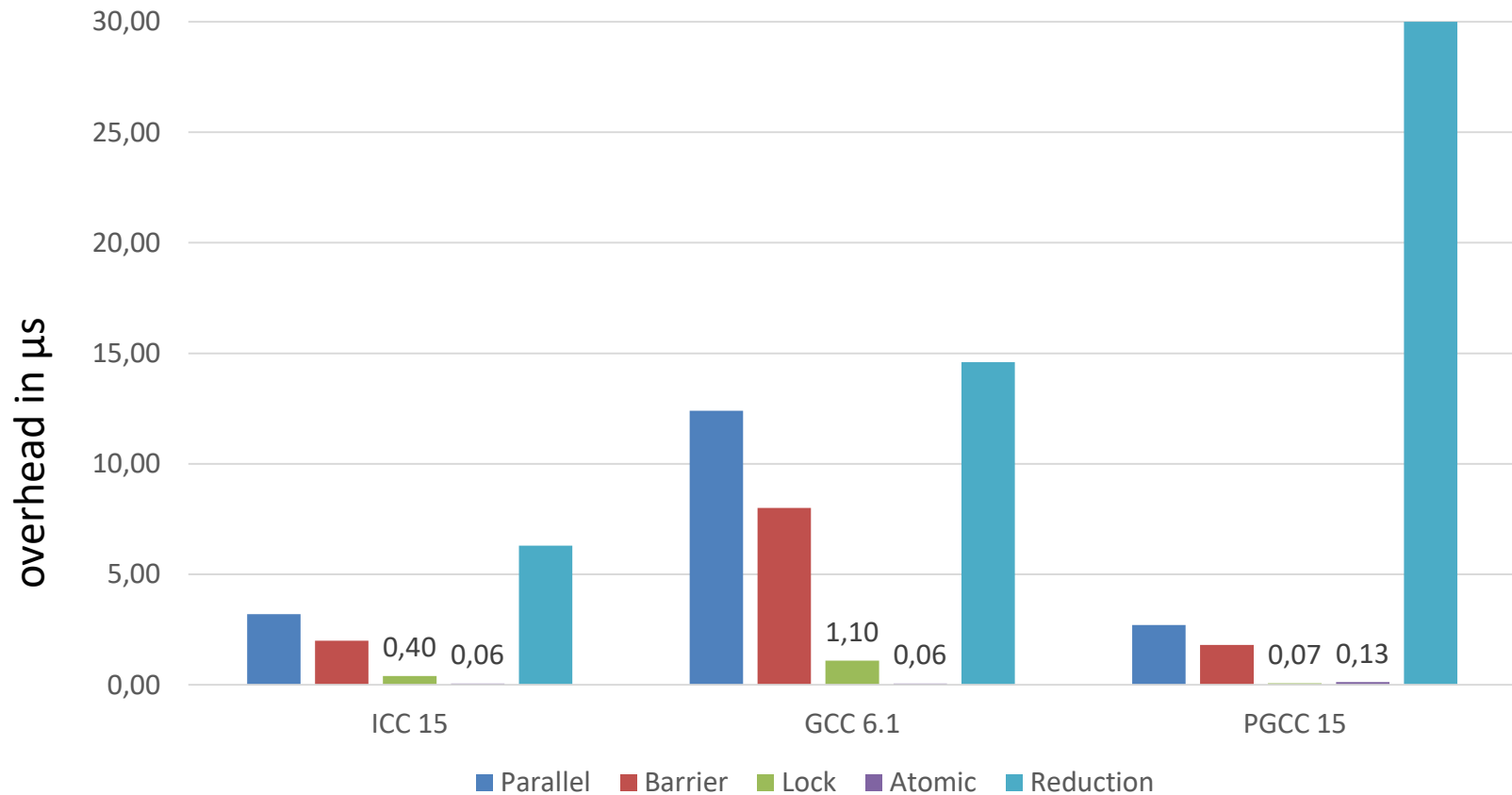
Westmere 4-socket node overhead with ICC 15



## 2-socket results with ICC 15

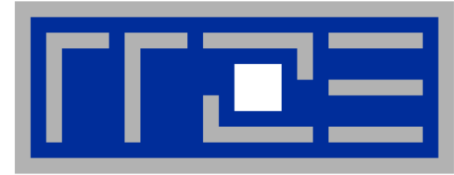


## Westmere 20 thread results



- **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**

## Example: linked list

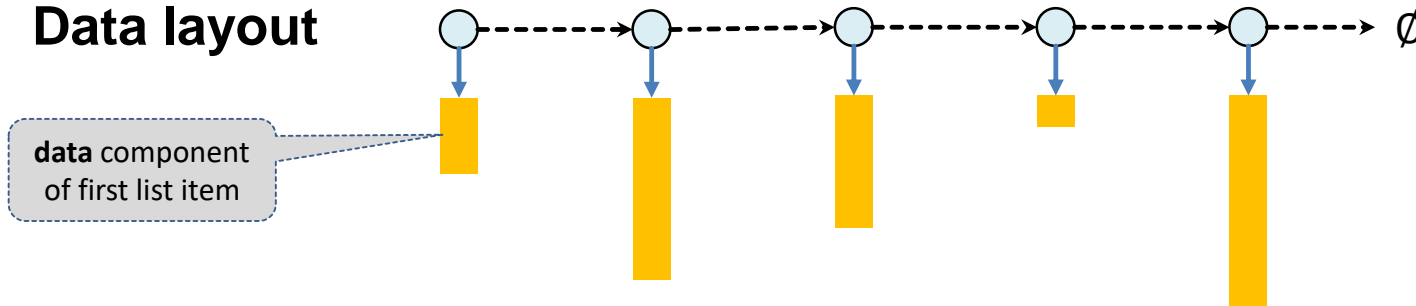
Fortran

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

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 → 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)

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

  p => head
  do while (associated(p))
    call do_work(p%data)
    p => p%next
  end do
end subroutine
```

Fortran

```
void process_list(list *head) {
  list *p = head;

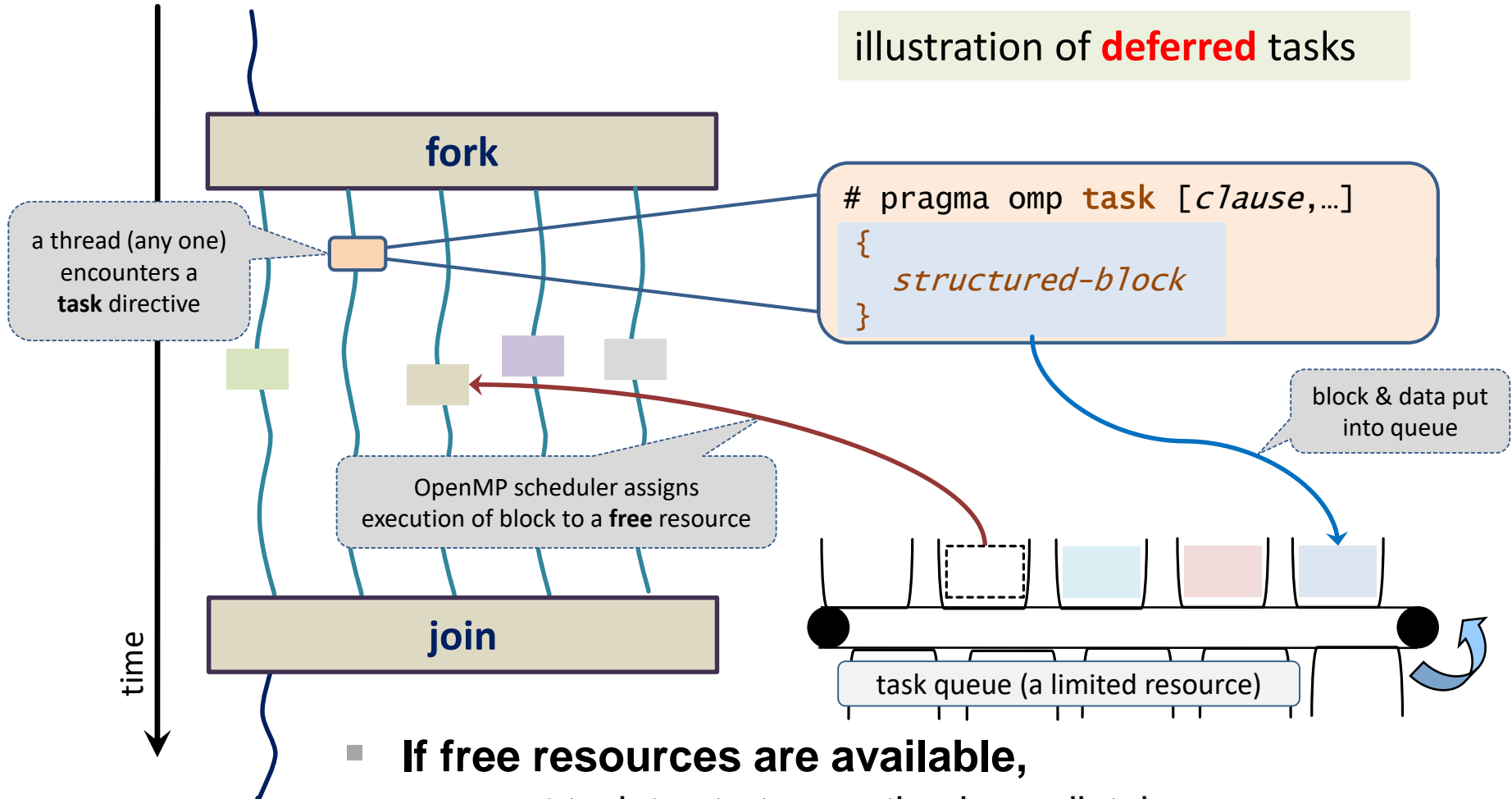
  while (p) {
    do_work(p->data, p->n);
    p = p->next;
  }
```

C

- **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.

- **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





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

```

program code_sections
  use mod_functions
  implicit none
  real :: a, b
  integer :: n = ...
!$omp parallel
!$omp master

```

Fortran

```

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

```

concurrently executed  
if sufficiently many  
threads available

```

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

```

```

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

```

no synchronization  
(different than **single**)

threads waiting here  
will be put to work

```

int main() {
  float a, b;
  int n = ...;
#pragma omp parallel
#pragma omp master
  {

```

only thread 0  
creates tasks

```

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

```

a and b have  
shared scope

```

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

```

```

} // end parallel and master

```

```

printf("%f\n", a + b);
}

```

```
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);
            }
        }
    }
}
```

**C**

i is **private** (loop index)

b is **firstprivate**

a is **shared** (because it is shared in all lexically enclosing constructs)

**default scopings**

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

```

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

```

Fortran

only one thread  
creates tasks

task region (includes  
procedure execution)

synchronization  
here → all tasks  
done

```

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
}

```

C

### ■ 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

- **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

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

C

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

Fortran

- **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

## Divide and conquer

```
float daq(float *data, int n) {
    float x1, y1;
    int n1 = ..., n2 = ...;
    float *data2 = ...;
    if (n1 < THRESHOLD)
        { ... }
    #pragma omp task shared(x1)
        { x1 = daq(data, n1); }
    #pragma omp task shared(y1)
        { y1 = daq(data2, n2); }
    #pragma omp taskwait
        return x1 - y1;
}
```

private at this point

terminate recursion

C

- initial function invocation in a parallel region, usually from a single thread

## Previous example:

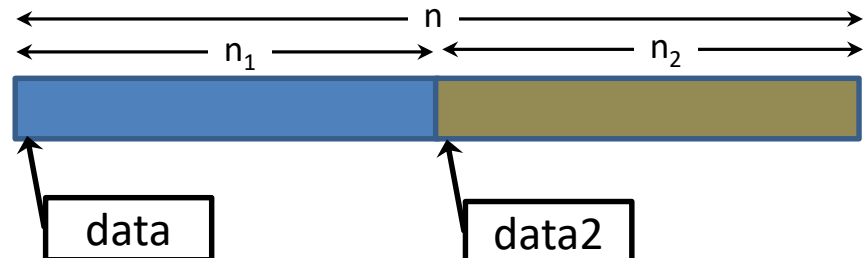
- only sibling tasks are created

## 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



- **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
```

Fortran

```
#pragma omp taskwait
```

C

- **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

- **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	discussed later
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 ...



- **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

```
# pragma omp task untied  
structured-block C
```

- **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.



## 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

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

all thread-centric programming is localized here

- or use an "if (0)" clause (undeferrred task might be executed by a different thread, though)

- **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

## ■ Syntax and Semantics

```
!$omp taskyield
```

Fortran

```
#pragma omp taskyield
```

C

- permits (but does not force) task suspension for the current task at the point where the directive is placed

## ■ Example

- avoid deadlock in a mutual exclusion region  
(taken from the OpenMP examples)

```
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
```

Fortran

## ■ 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)

↔ taskwait

## ■ Syntax:

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

new tasks are started during execution of this block („taskgroup set“)

Fortran

```
#pragma omp taskgroup
{
  structured block
}
```

C

### Example:

recursive tasking with atomic updates in each task → guarantee completeness of updates

## ■ Note:

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

↔ taskwait, barrier

## ■ 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.

## ■ Syntax

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

Fortran

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

C

## ■ 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

```
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
```

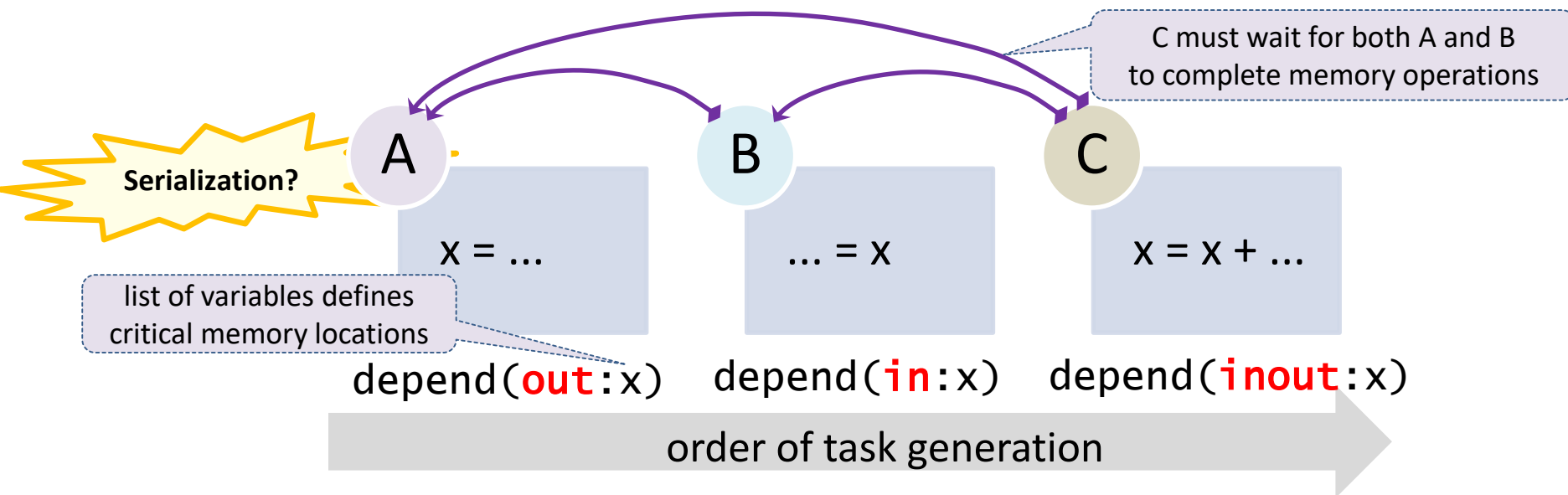


## Consider

- a set of sibling tasks

i.e., all created by the same  
(parent) task

-  a shared variable  $x$  that is referenced or defined by more than one of them



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

- Via addition of **depend** clauses

```
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)

- Drawing the square root of a matrix

symmetric positive definite

$$A = L \cdot L^T$$

lower triangular

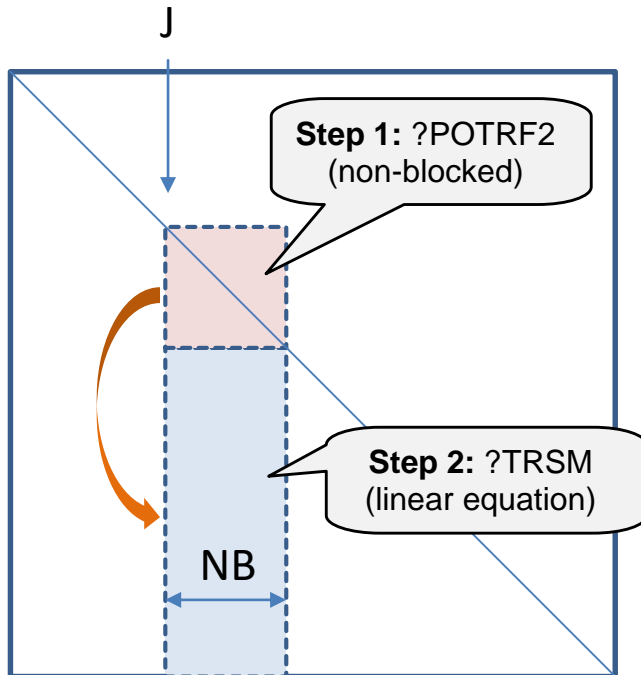
- Recursive blocked algorithm

$$\begin{array}{|c|c|} \hline A_{11} & A_{21}^T \\ \hline A_{21} & A_{22} \\ \hline \end{array}
 =
 \begin{array}{|c|c|} \hline L_{11} & 0 \\ \hline L_{21} & L_{22} \\ \hline \end{array}
 *
 \begin{array}{|c|c|} \hline L_{11}^T & L_{21}^T \\ \hline 0 & L_{22}^T \\ \hline \end{array}$$

- LAPACK algorithm ?POTRF

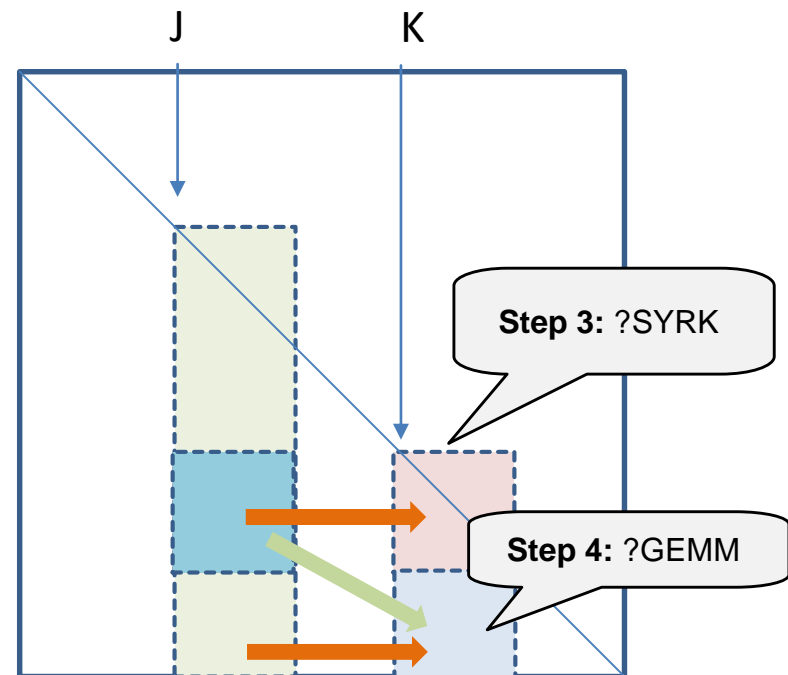
## Phase 1

- one thread only
- „hot“ block column is J



## Phase 2

- **parallel** updates of columns  $K = J+NB, J+2*NB, \dots$



load imbalance  
→ use suitable schedule

```

!$OMP PARALLEL PRIVATE(JB, KB)
  DO J = 1, N, NB
    JB = MIN( NB, N-J+1 )
!$OMP SINGLE
! Update the current diagonal block
! A(J,J), JB by JB and test for
! non-positive-definiteness
  CALL POTRF2( ... )
  IF ( J+JB.LE.N ) THEN
! using the above, solve for
! A(J+JB,J), N-J-JB+1 by JB
    CALL DTRSM( ... )
  END IF
!$OMP END SINGLE
  ...

```

```

!$OMP DO SCHEDULE(...)
  DO K = J+NB, N, NB
    KB = MIN( NB, N-K+1 )
! Update diagonal block A(K,K)
! from A(J,K)
    CALL DSYRK( ... )
    IF ( K+KB.LE.N ) THEN
! Update subdiagonal block A(K+KB,K)
! from A(K+KB,J) and A(K,J)
      CALL DGEMM( ... )
    END IF
  END DO
!$OMP END DO
END DO
!$OMP END PARALLEL

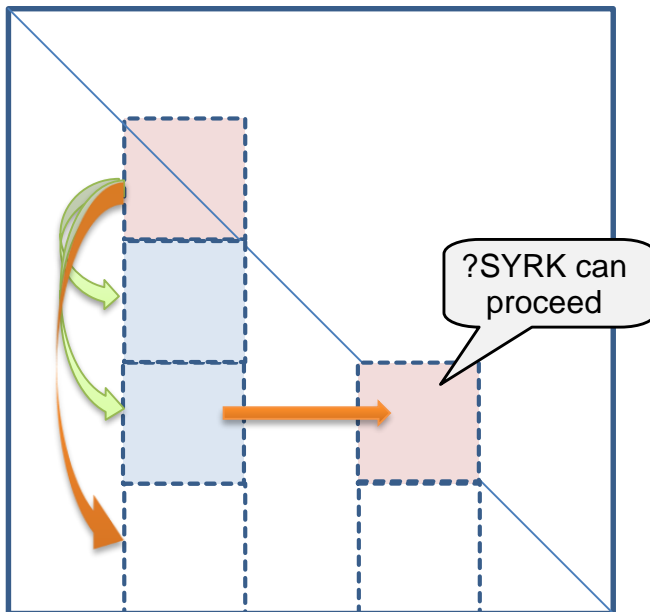
```

## ■ Reminiscence:

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

### 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

```

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

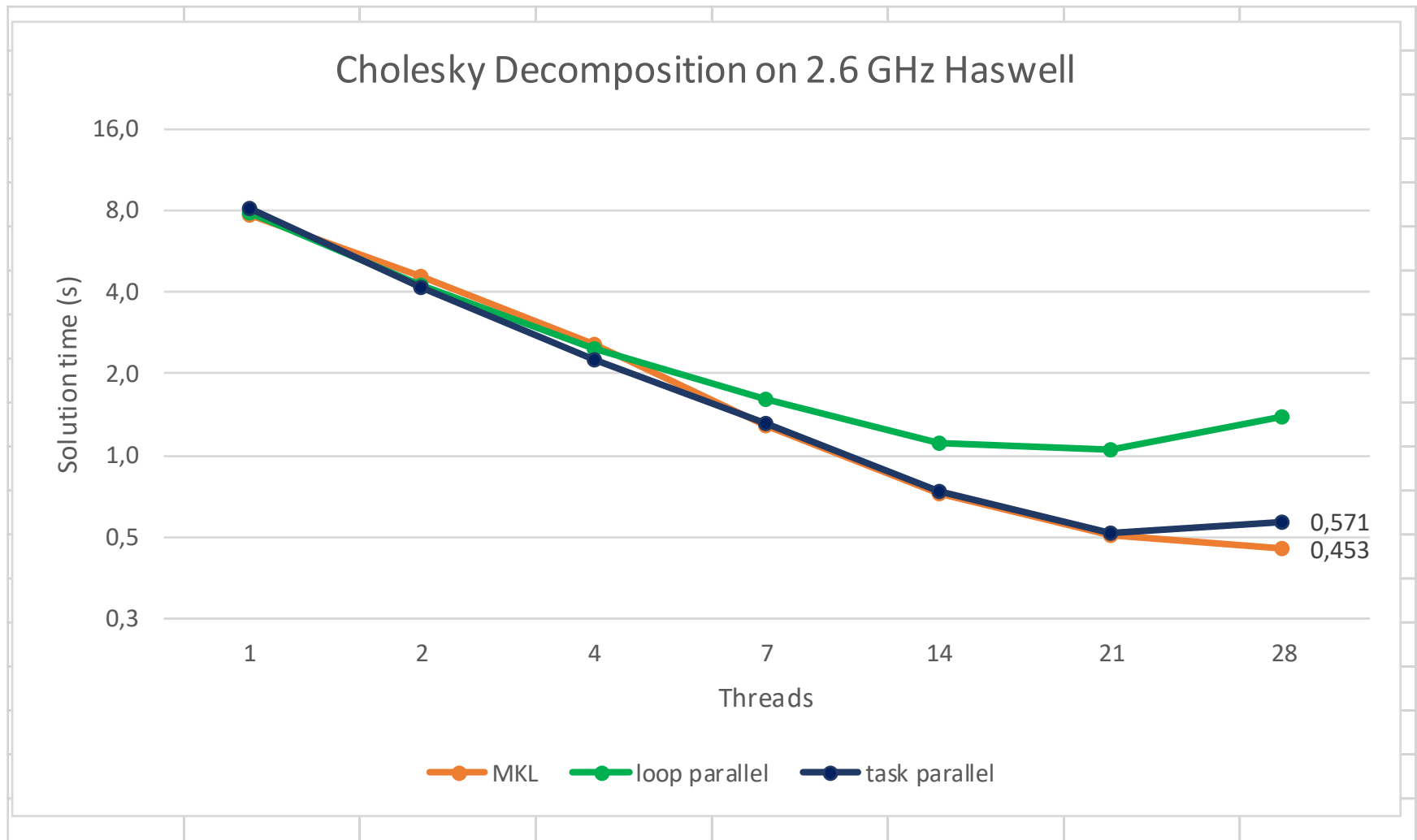
```

explicit  
synchronization  
point removed

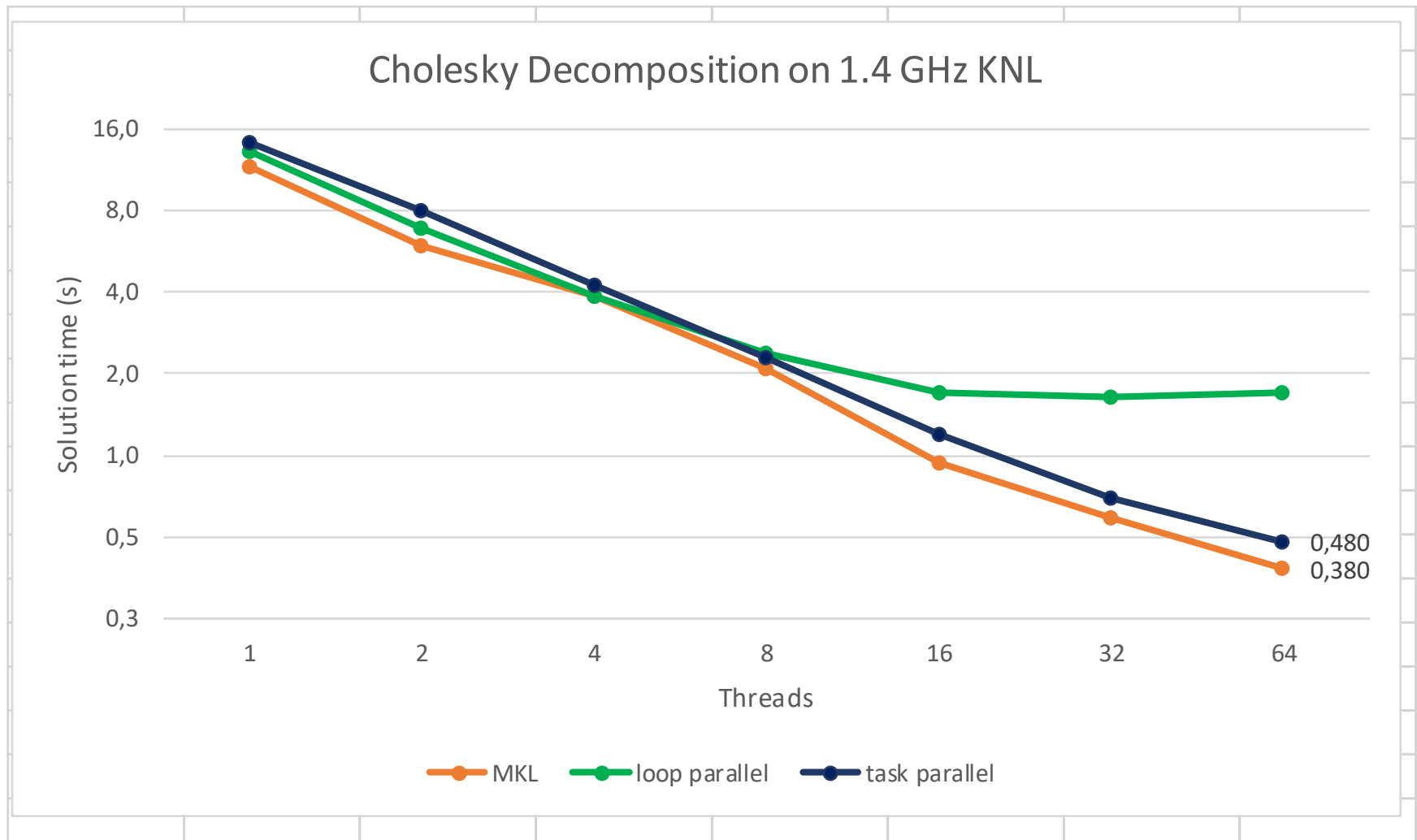
```

  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 DGEMM( ... )
!$OMP END TASK
  END DO
  END DO
!$OMP END SINGLE
END DO
!$OMP END PARALLEL

```







- Comparing the  $N = 6000$  solution time

	KSR1 (24 cells)	Haswell (28 cores)	KNL (64 cores)
year of release	1992	2014	2015
solution time (s)	270	0.13	0.16
GFlop/s	0.267	566	440

memory limit  
of machine

strong scaling limit



- **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**

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

Fortran

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

C

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

- **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

Current compiler support is limited

constrains number of iterations assigned to each task (upper limit  $< 2 * \text{grainsize}$ )

maximum number of tasks created

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.



## 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
```

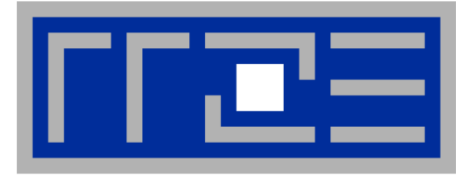
taskgroup or taskloop region

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



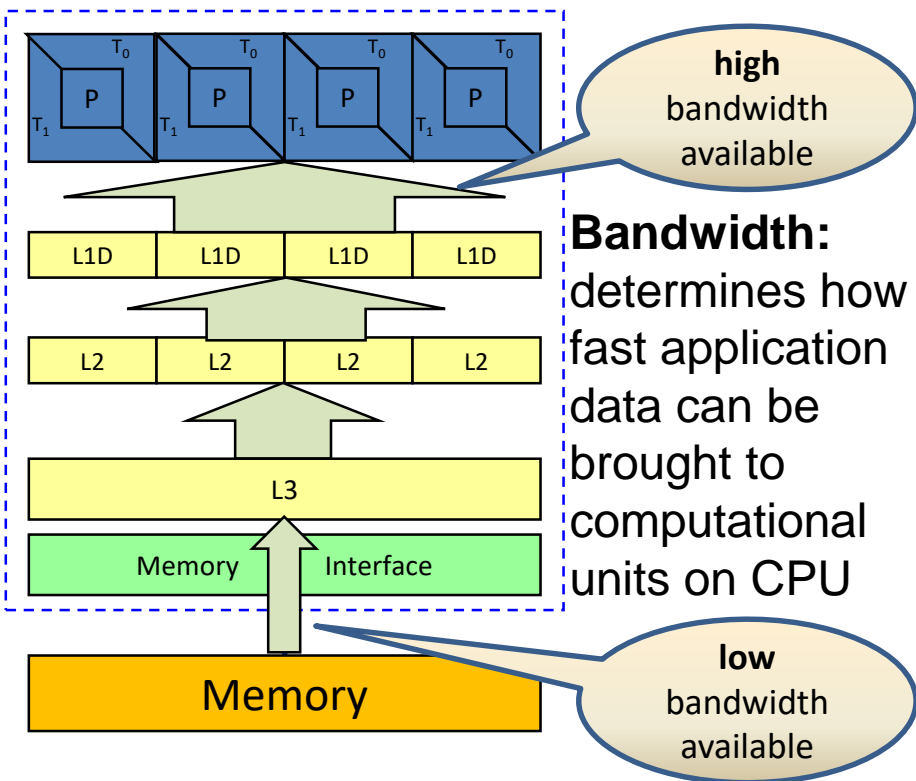
**Performance:**

**Architectural aspects**

- **What can be expected from the processor architecture?**
  - want at least an estimate for performance limits → avoid „stumbling in the dark“
  - much more detailed node performance engineering and modeling: course by G. Hager and G. Wellein – see <http://moodle.rrze.uni-erlangen.de/moodle/course/view.php?id=300&username=guest&password=guest&lang=en> and references cited within
  
- **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!)

## ■ 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



- **A small but fast memory area**

- used for storing a (small) memory working set for efficient access

- **Reasons:**

- physical and economic limitations

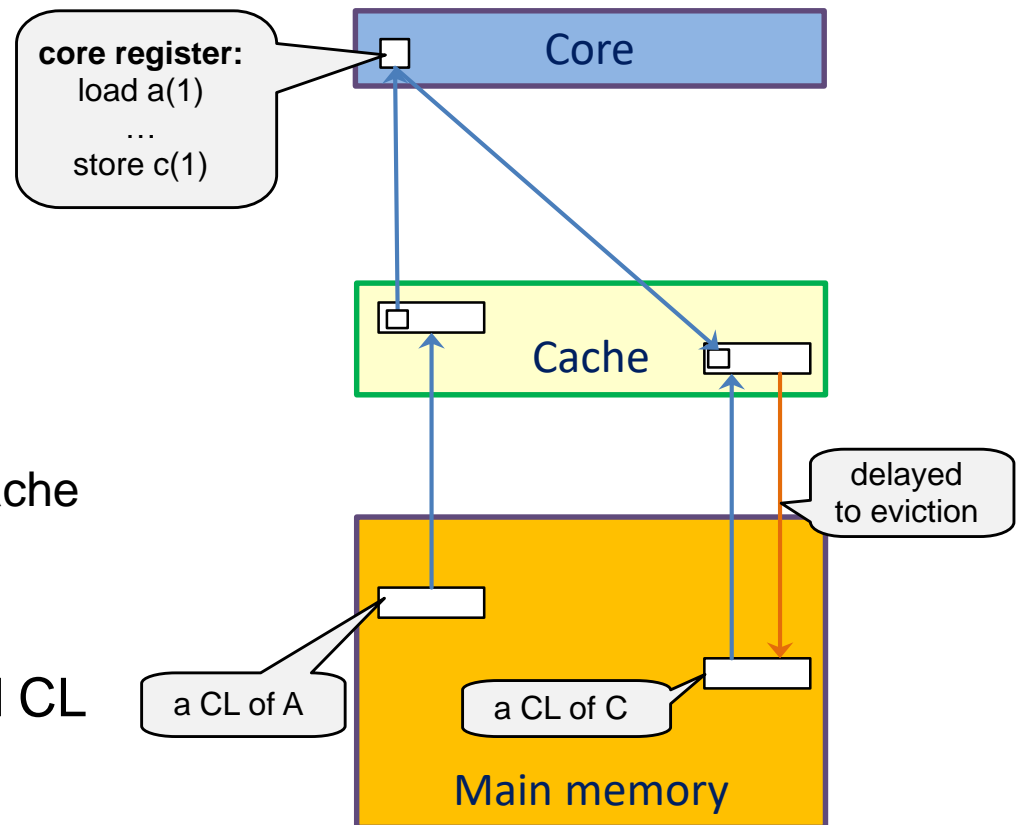
- **Loads (stores) to (from) core registers**

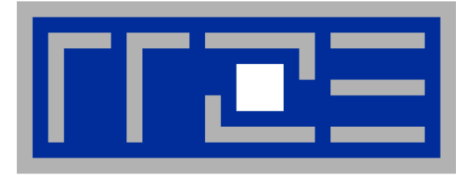
- may trigger cache miss → transfer of memory block („cache line“, CL) from memory

- **Cache fills up ...**

- usually least recently used CL is evicted

- **Example:**  $c(:) = a(:) + \dots$





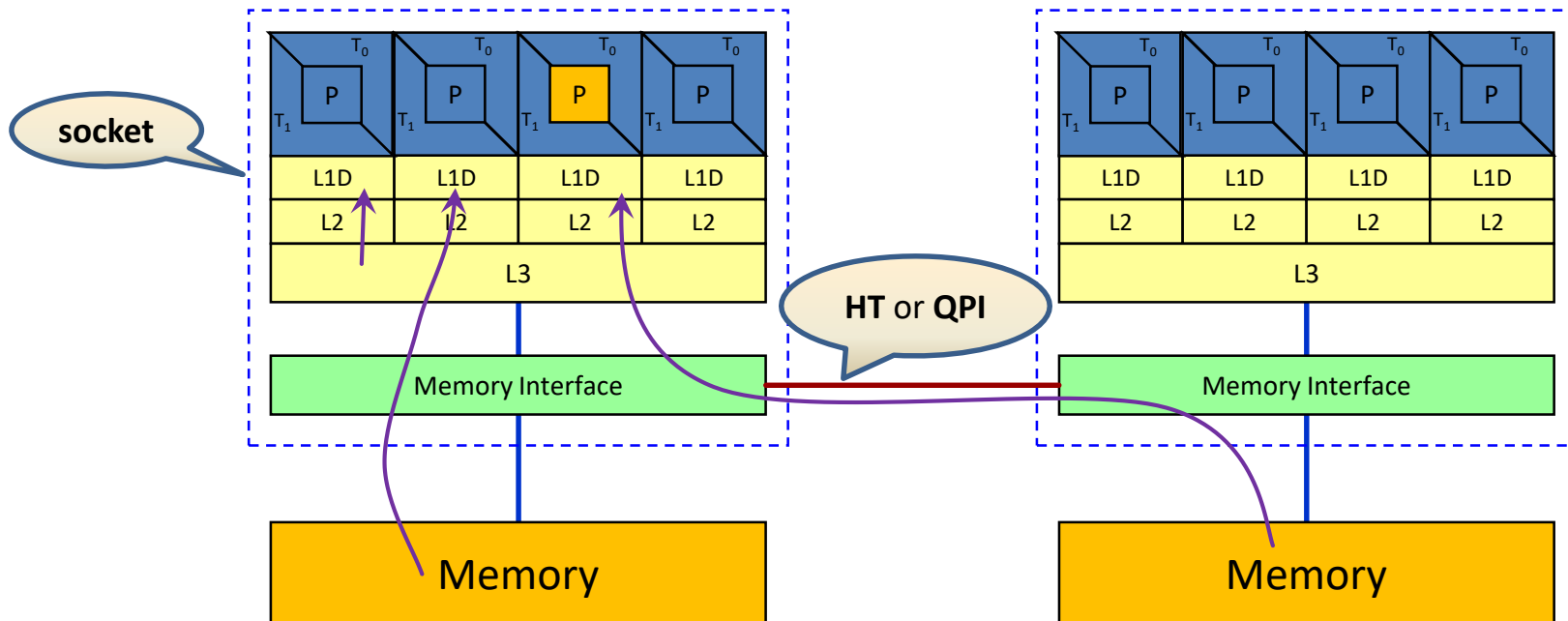
# **Control of Affinity**

## **NUMA effects**

## **False Sharing**

- **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:

```

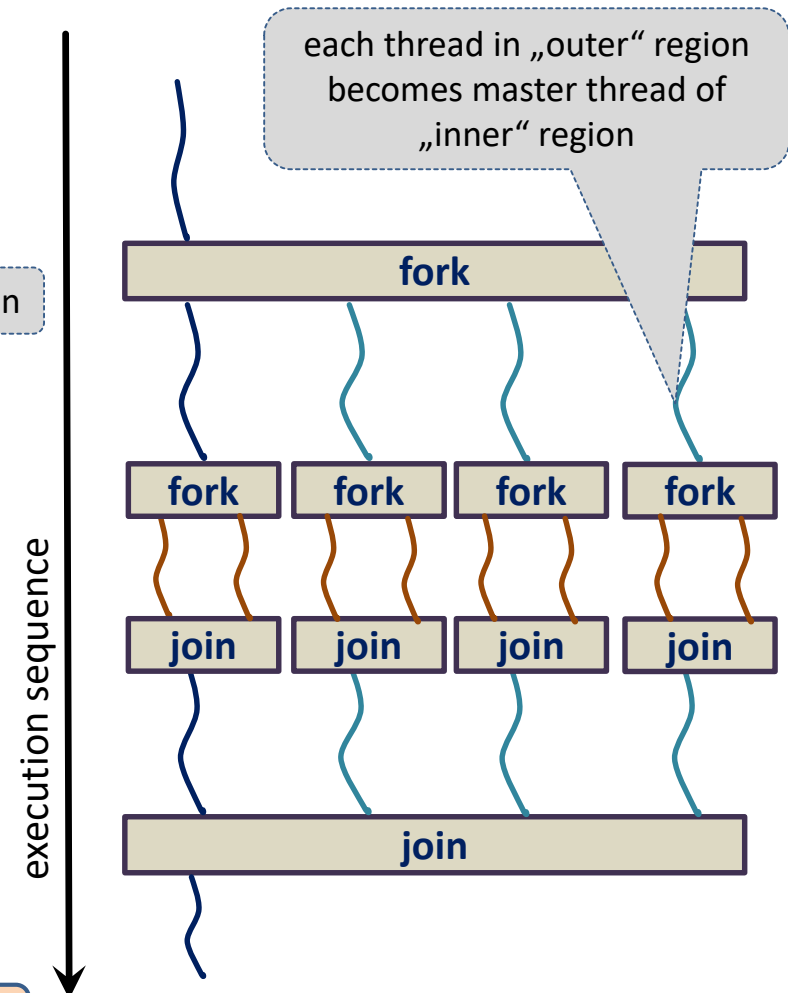
#include <stdio.h>
int main() {
#pragma omp parallel „outer“ region
{ ...
#pragma omp parallel „inner“ region
{
...
}
}
return 0;
}

```

**C**

- nesting of parallel regions

mentioned here for illustrative purposes



## ■ 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

- **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

## ■ 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)"
```

8 places with 1  
physical core each

- 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}"
```

2 places with 4 hw  
threads each

```
export OMP_PLACES="{0:4},{4:4}"
```

same, using  
<offset:length> notation

```
export OMP_PLACES="{0:4}:2:4"
```

same, using  
<firstplace:#\_of\_places:stride\_of\_offset>  
notation

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

- **Determine whether threads should be pinned**
  - environment variable OMP\_PROC\_BIND
  - with values **true** or **false**, or
  - a comma-separated list of entries:

master	bind created threads to same place as master thread
close	bind created threads to a place close to the one assigned to the master
spread	use a sparse distribution pattern to bind created threads to places

- **Example:**

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

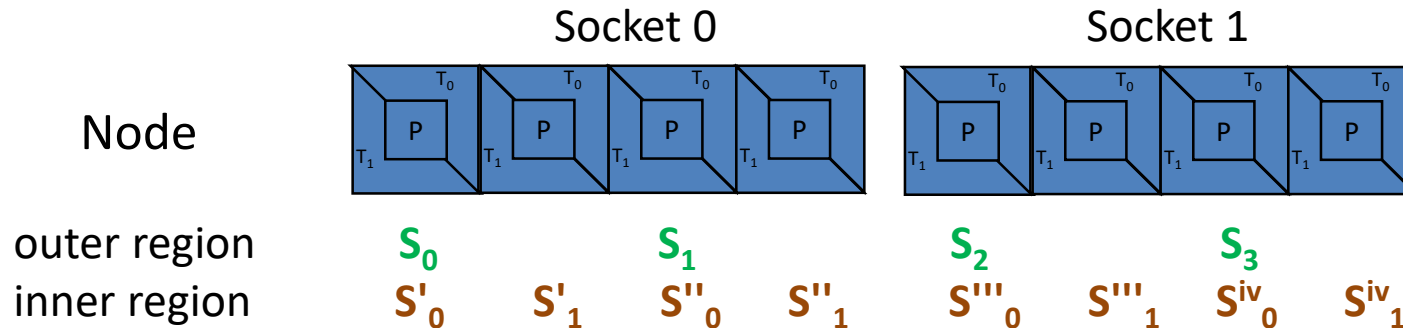
- binding is determined for at most two levels of parallel nesting



- Nested parallelism example from earlier

```
export OMP_NUM_THREADS=4,2
...
export OMP_PLACES="cores(8)"
export OMP_PROC_BIND=spread,close
./my_nested_openmp_program.exe
```

- Threads are named  $S_i$ , and  $S'_i$ ,  $S''_i$ , ..., for outer and inner region, respectively:



- Overcommitment causes places to be reused (i.e. multiple threads per place)

- **The function**

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

Fortran

```
omp_proc_bind_t omp_get_proc_bind(void)
```

C

returns one of the following **constants**:

omp_proc_bind_false	0
omp_proc_bind_true	1
omp_proc_bind_master	2
omp_proc_bind_close	3
omp_proc_bind_spread	4

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

- A number of functions exist to handle various inquiries:

Name	Result type	Purpose
<code>omp_get_num_places()</code>	int	number of places available
<code>omp_get_place_num_procs</code> <code>(int place_num)</code>	int	number of processors available in <code>place_num</code> (0 .. number of places - 1)
<code>omp_get_place_proc_ids</code> <code>(int place_num, int *ids)</code>	void	<code>ids</code> contains numerical identifiers of processors in place <code>place_num</code>
<code>omp_get_place_num()</code>	int	place number of place to which calling thread is bound
<code>omp_get_partition_num_places()</code>	int	number of places in place partition of innermost implicit task
<code>omp_get_partition_place_nums</code> <code>(int *place_nums)</code>	void	list of place numbers for innermost implicit task

- A `proc_bind` clause can be specified
- Example:

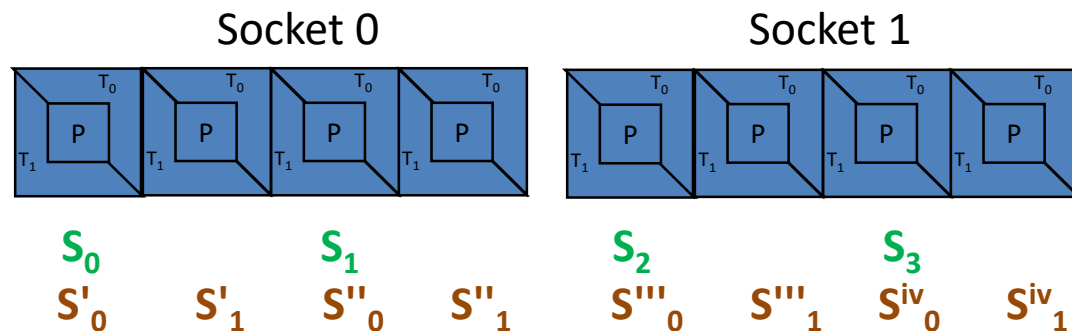
```
#pragma omp parallel num_threads(4) proc_bind(spread)
{ ...
#pragma omp parallel num_threads(2) proc_bind(close)
{ ...
}
}
```

C

executed with  
`OMP_PLACES=cores(8)`

Node

outer region  
inner region



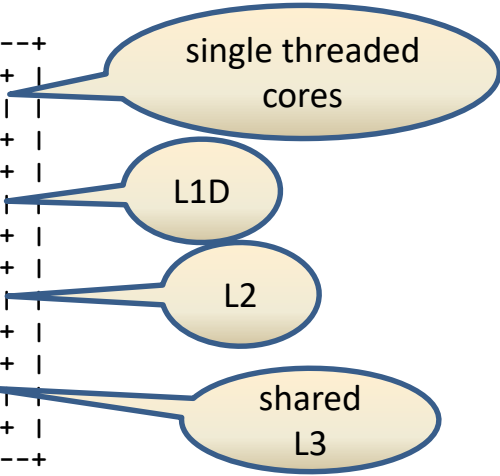
- **Topology =**
  - Where in the machine does core #n reside?
  - awkward numbering anyway?
  - which cores share which cache levels
  - which hardware threads (“logical cores”) share a physical core?
- **Use LIKWID tool to identify**
  - developed by J. Treibig
  - <https://github.com/RRZE-HPC/likwid> 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:

```
+-----+
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+
| |  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 |
| +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+ +-----+
+-----+
```

each socket forms **two** NUMA domains

- **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`):

```
likwid-pin -c 0,2,4-6 ./myApp parameters
```
  - Logical numbering by topological entities:

```
likwid-pin -c S0:0-3 ./myApp parameters
```



- **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

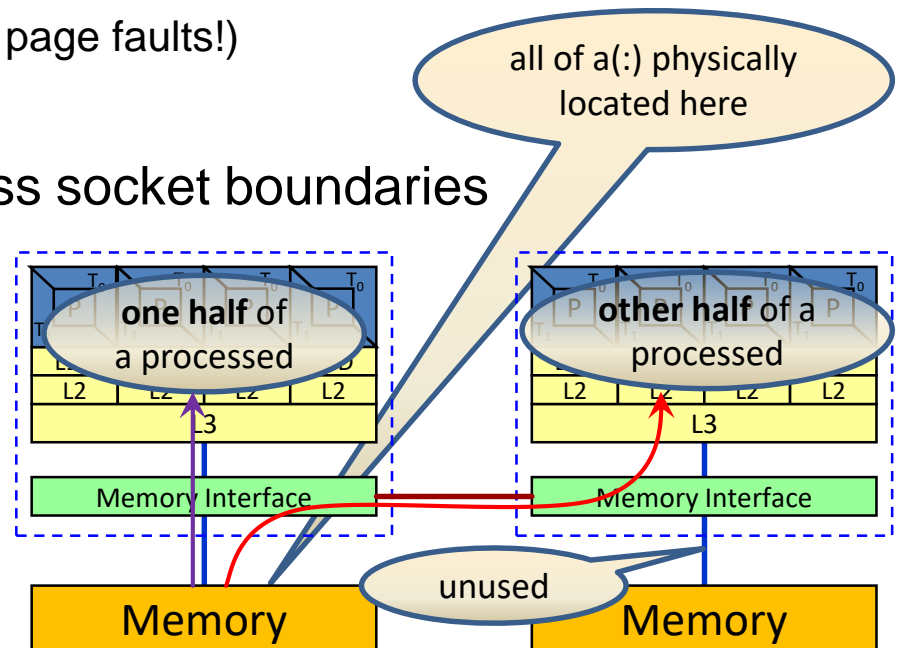
Fortran

```

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

```

first touch here



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

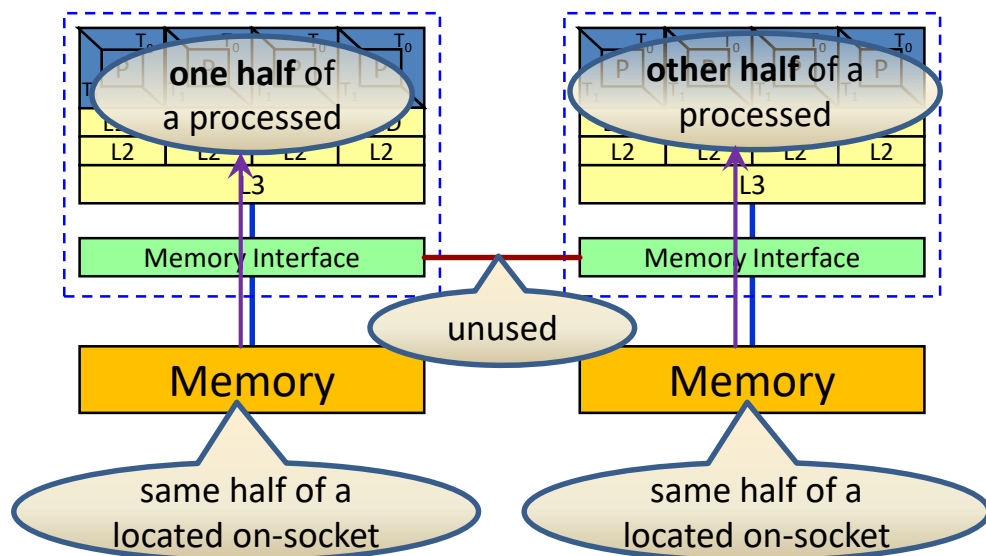
- **Desirable** and **scalable** memory access pattern:
  - requires initialization with an OpenMP parallelized loop
- **Distributed first touch**
  - ideally, uses same loop schedule as later processing

Fortran

```

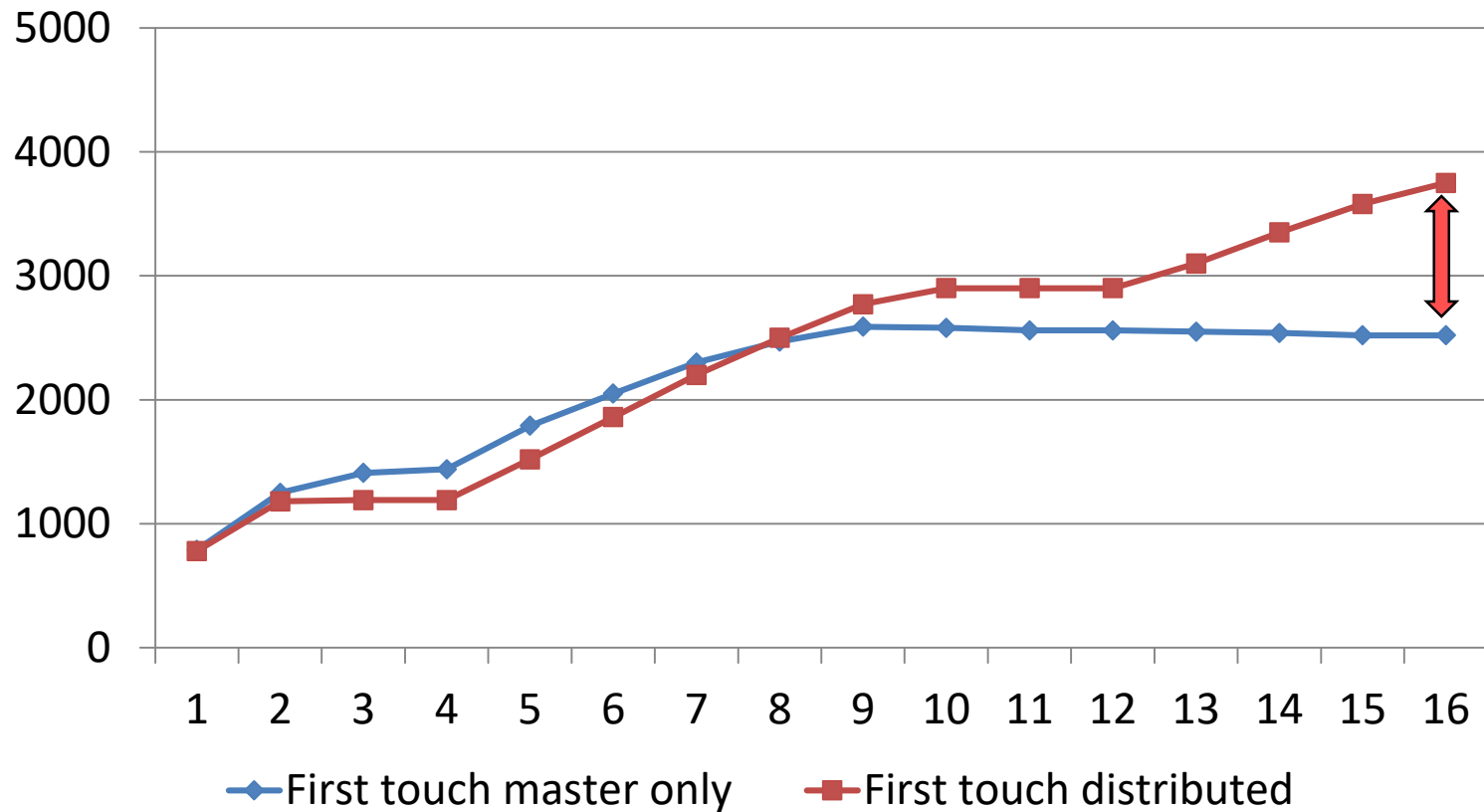
!$omp parallel do
DO i=1, size(a)
  a(i) = ...
END DO
!$omp end parallel do
...
!$omp parallel do
DO i=1, size(a)
  ... = ... a(i) ...
END DO
!$omp end parallel do

```



- now, the **full** available memory BW can be exploited on a **multi**-socket system

- **Measured on two AMD Magny Cours sockets**
  - thread pinning uses „close“ strategy



## Remember:

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

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

## Consequence:

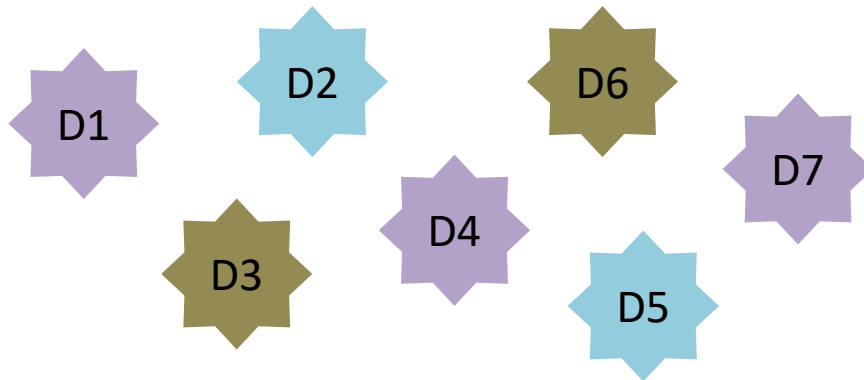
- repeated execution of tasking on data items might use different threads → memory affinity will get lost!

```
#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

## ■ At initialization

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



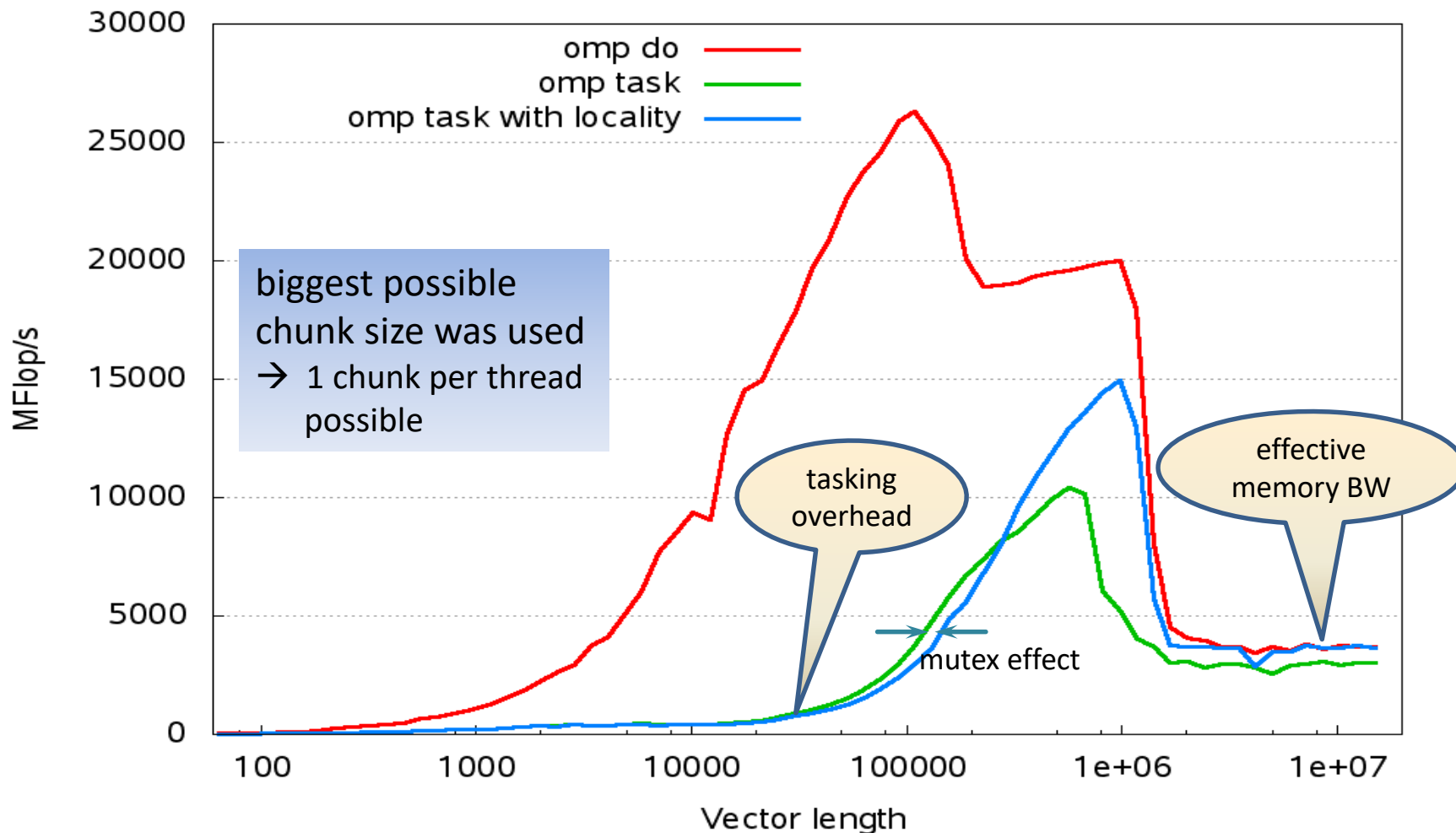
```
integer :: work_item(idm, nthr)
```

thread	0	1	2
item #	1	2	3
item #	4	5	6
item #	7	-	-

## ■ 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

work shared vector triad with 16 threads on Sandy Bridge



- Example program: count even and odd array values

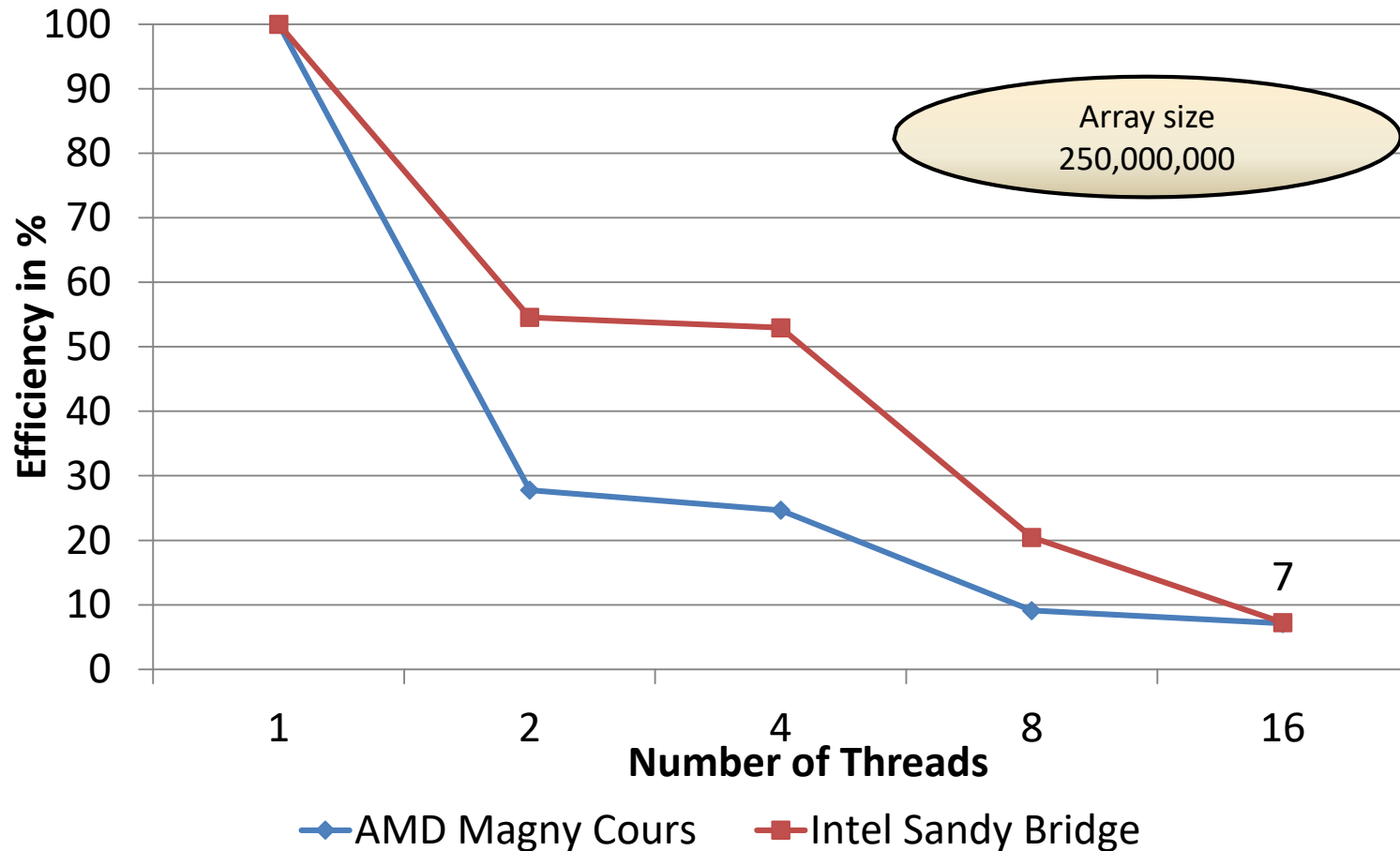
```
integer is(2), ict(2,ntdm), ia(n)
...
!$omp parallel private(myid) shared(ict, ia)
  myid = omp_get_thread_num()+1
!$omp do private(index)
  do i=1,n
    index = mod(ia(i),2)+1
    ict(index,myid) = ict(index,myid) + 1
  end do
!$omp end do
!$omp critical
  is = is + ict(1:2,myid)
!$omp end critical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

initialization omitted

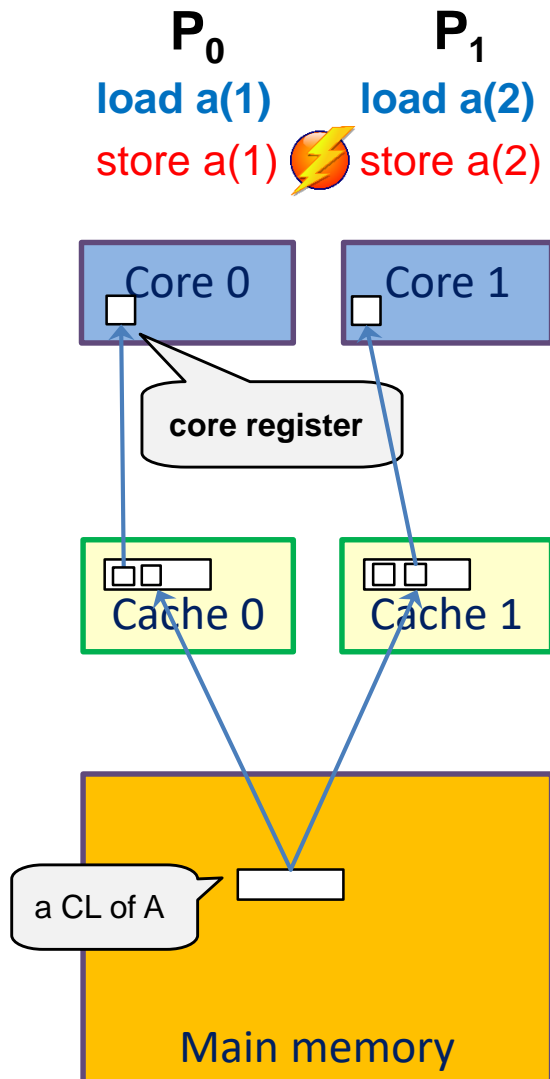
formally correct,  
no race condition

Fortran

- Baseline 1 thread execution time: AMD 0.75 s, Intel SandyBridge 0.37 s







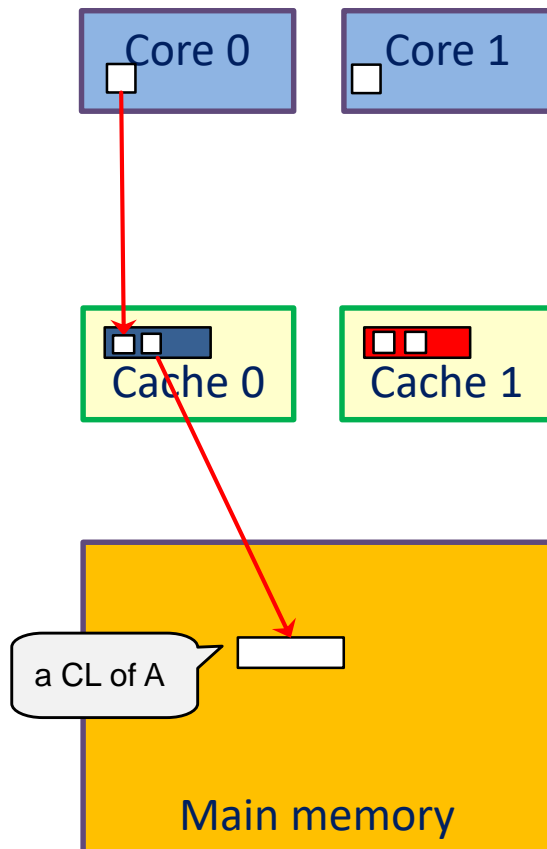
### Store operation

- 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

Diagram shows state after step 3



- **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**

- **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"**

- **Privatization – here through use of a reduction variable**

```
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
```

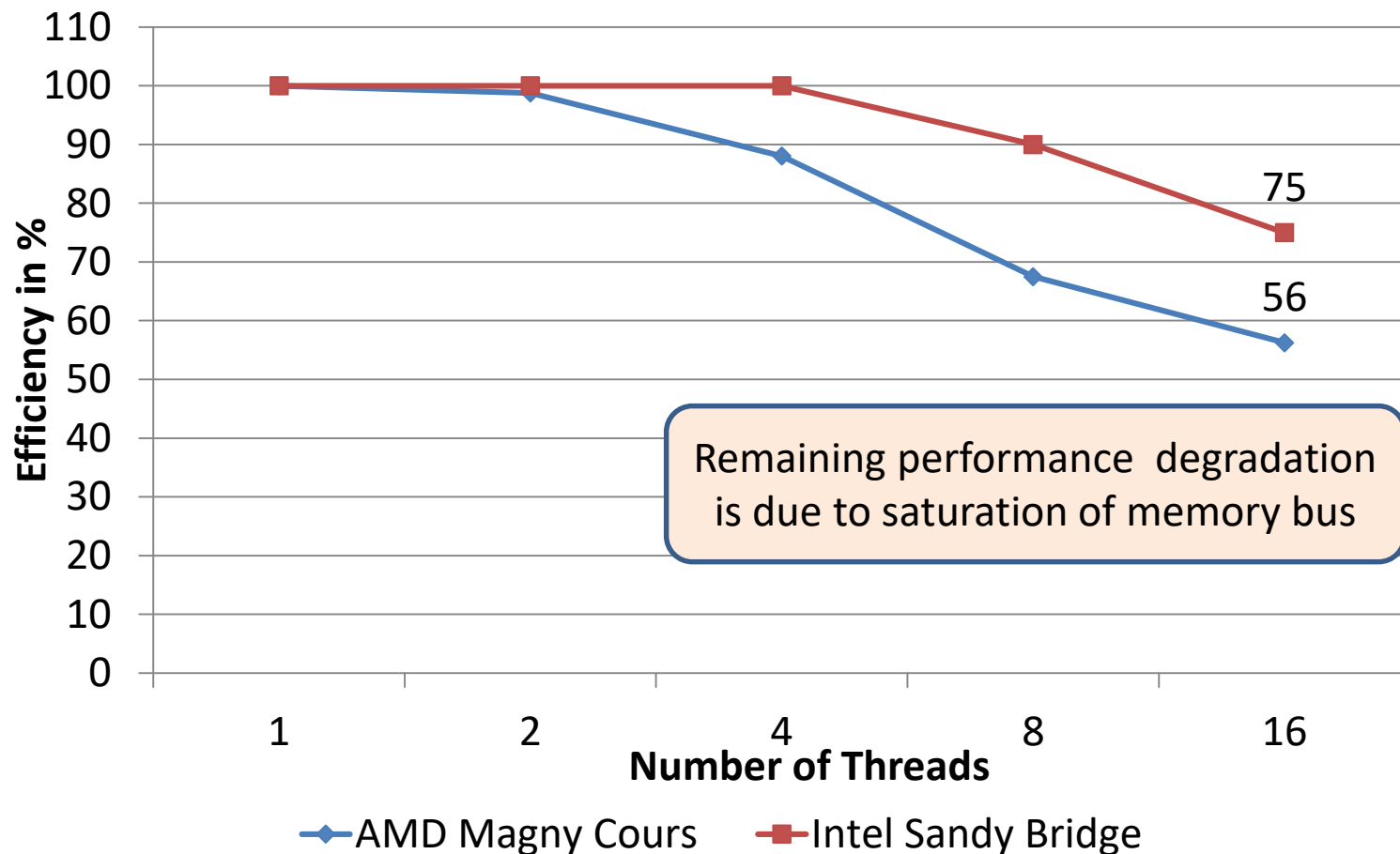
initialization omitted

private variables are assured of using well-separated parts of the physical memory (thread-individual stack or heap)

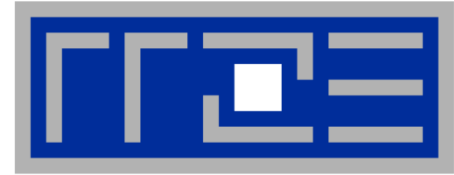
Fortran

- **Alternative for retaining shared variables: Add padding**
  - tradeoff: may lose spatial locality

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



Now: last exercise session



# Outlook: Towards quantifying performance

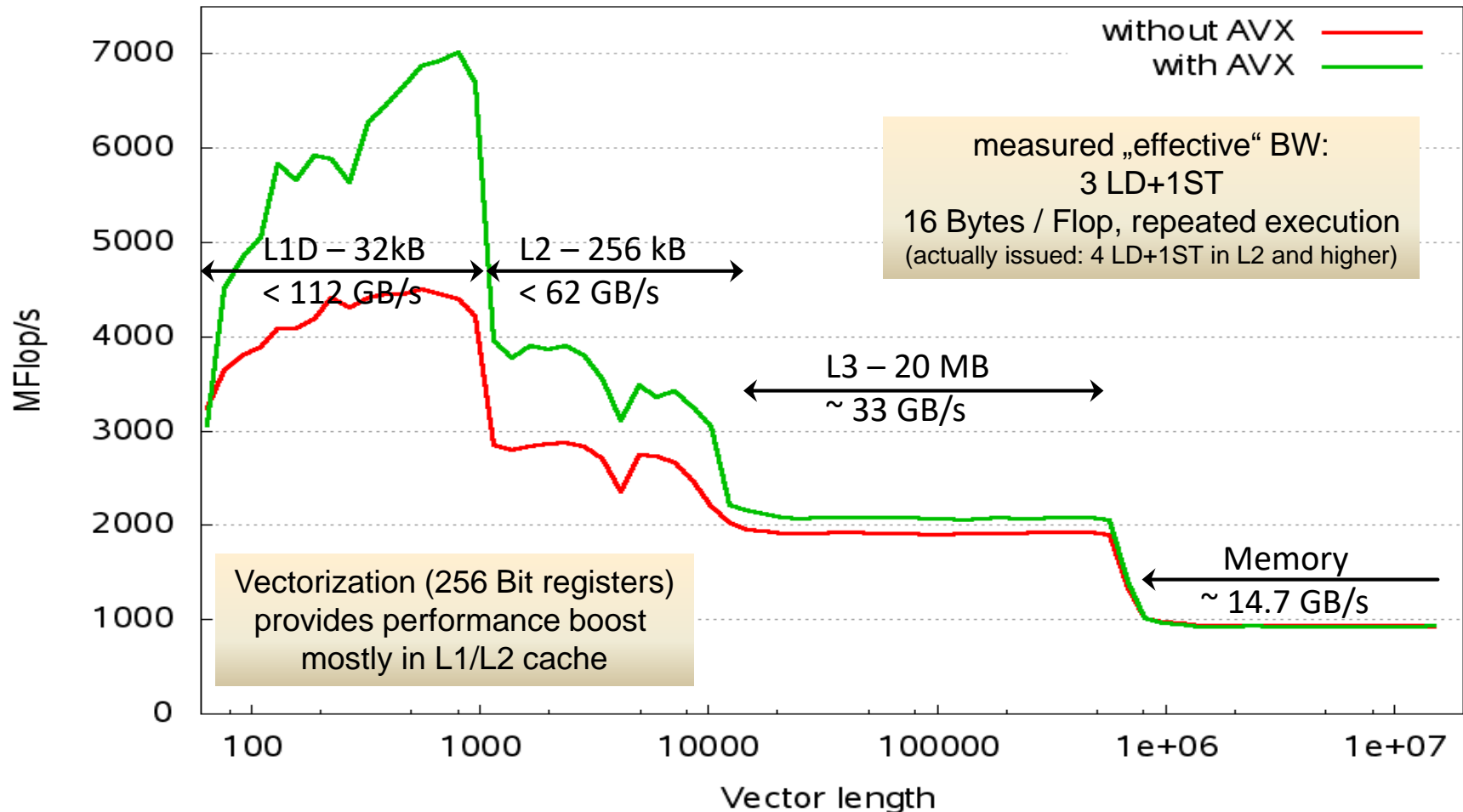
## ■ Characteristics

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

Kernel	Name	Flops	Loads	Stores
$s = s + a_i * b_i$	Scalar Product	2	2	0
$n^2 = n^2 + a_i * a_i$	Norm	2	1	0
$a_i = b_i * s + c_i$	Linked Triad (Stream)	2	2	1
$a_i = b_i * c_i + d_i$	<b>Vector Triad</b>	2	3	1

- run repeated iterations for varying vector lengths (working set sizes)

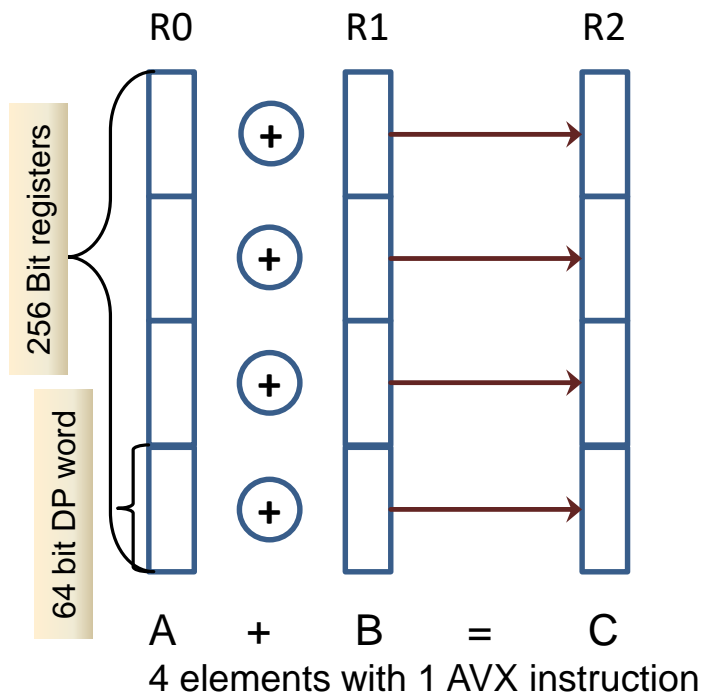
- Synthetic benchmark: bandwidths of „raw“ architecture**  
 for a **single core** Sandy Bridge 2.7 GHz / ifort 13.1





- **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)

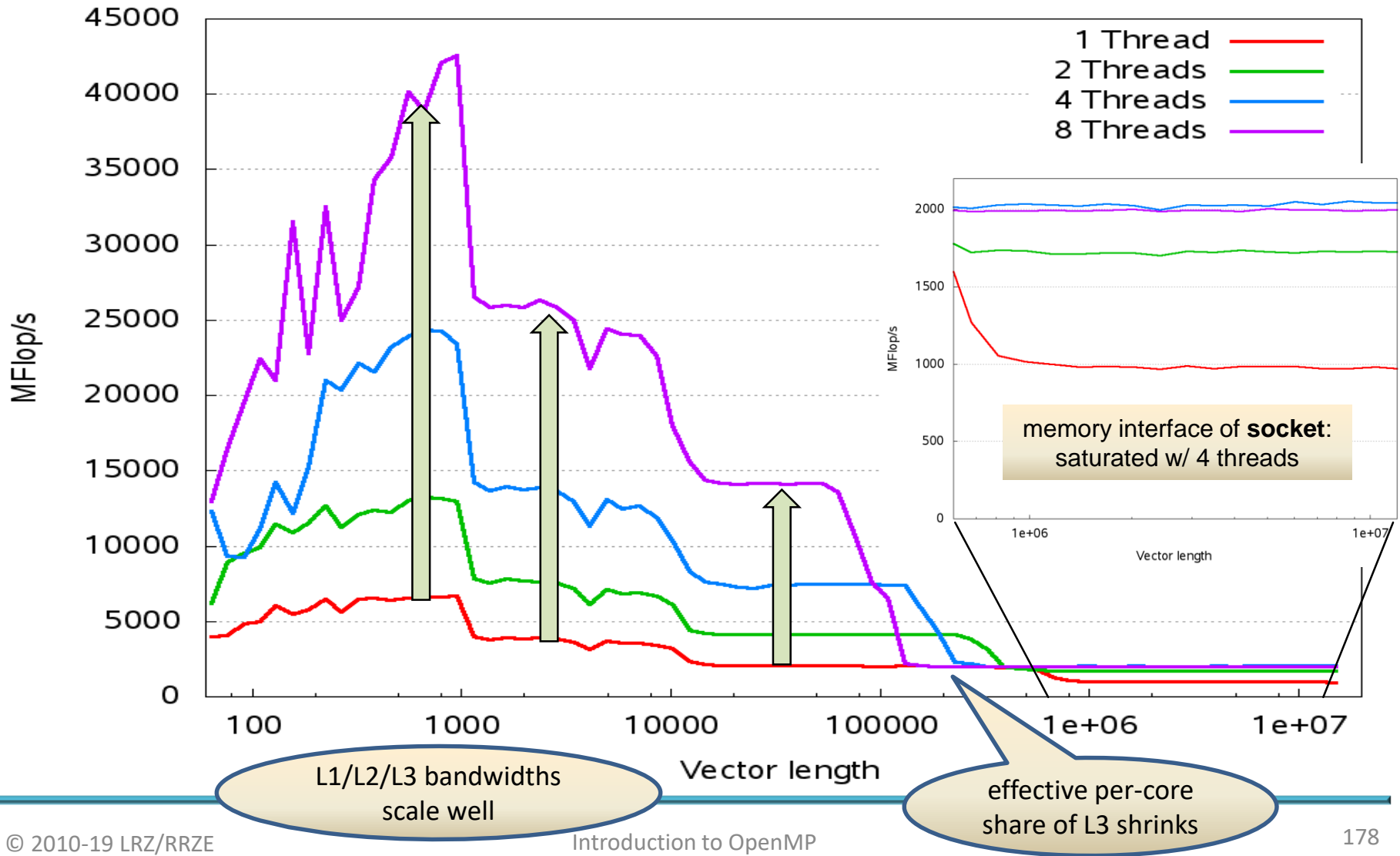
**Only L1** might maintain needed bandwidth

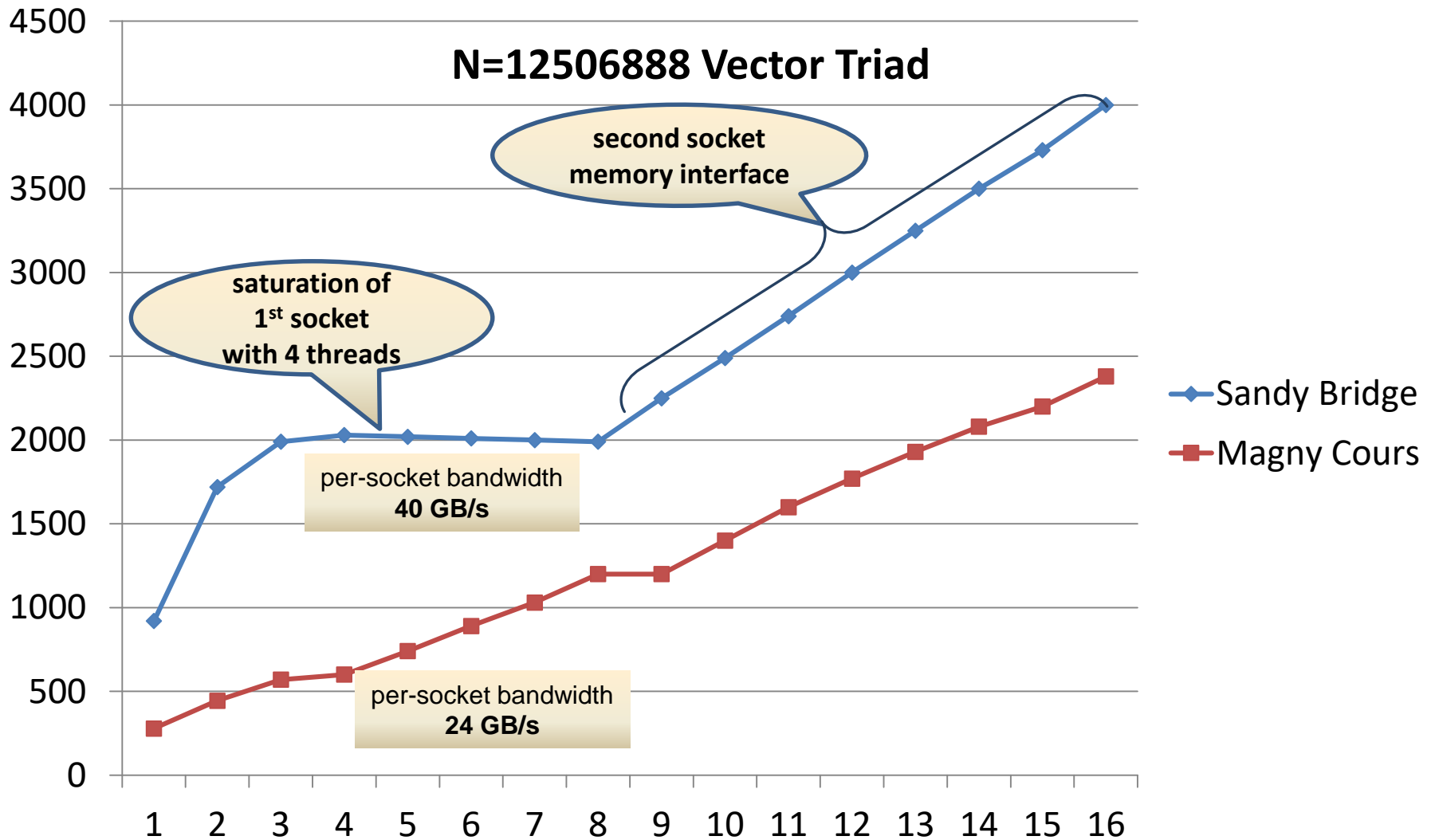
- **Vector triad:**

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

- **Consult processor-specific architecture manual**

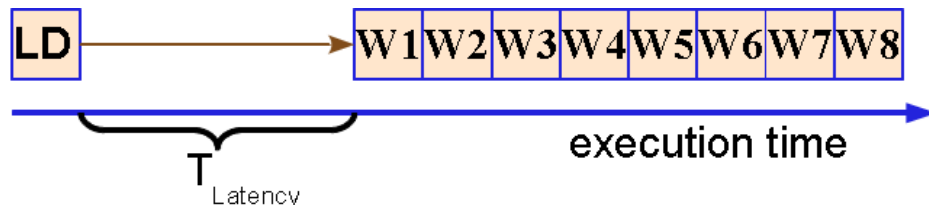
- **Throughput mode:** run with independent threads **up to number of cores** on a socket





## ■ Loads and Stores

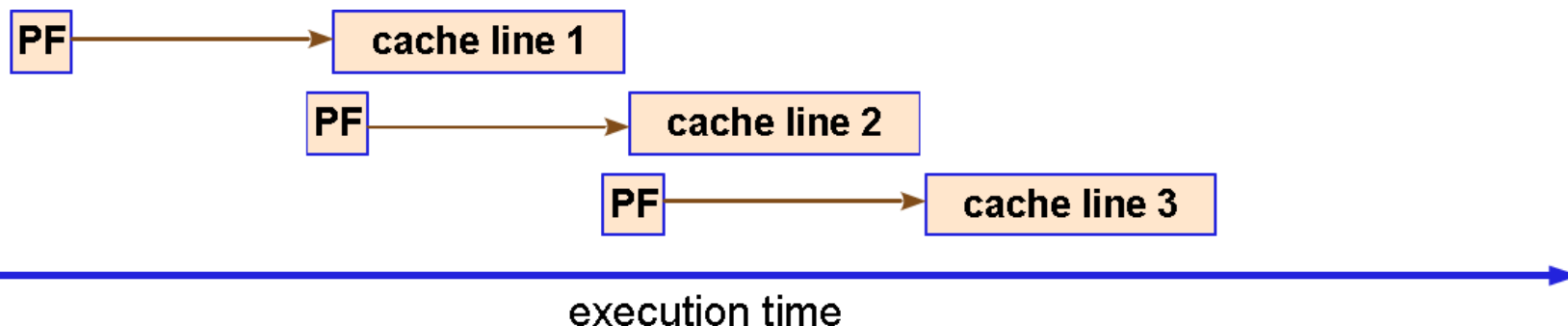
- usually apply to cache lines



- size: 64, 128 or more Bytes

## ■ Pre-fetch

- avoid latencies when streaming data



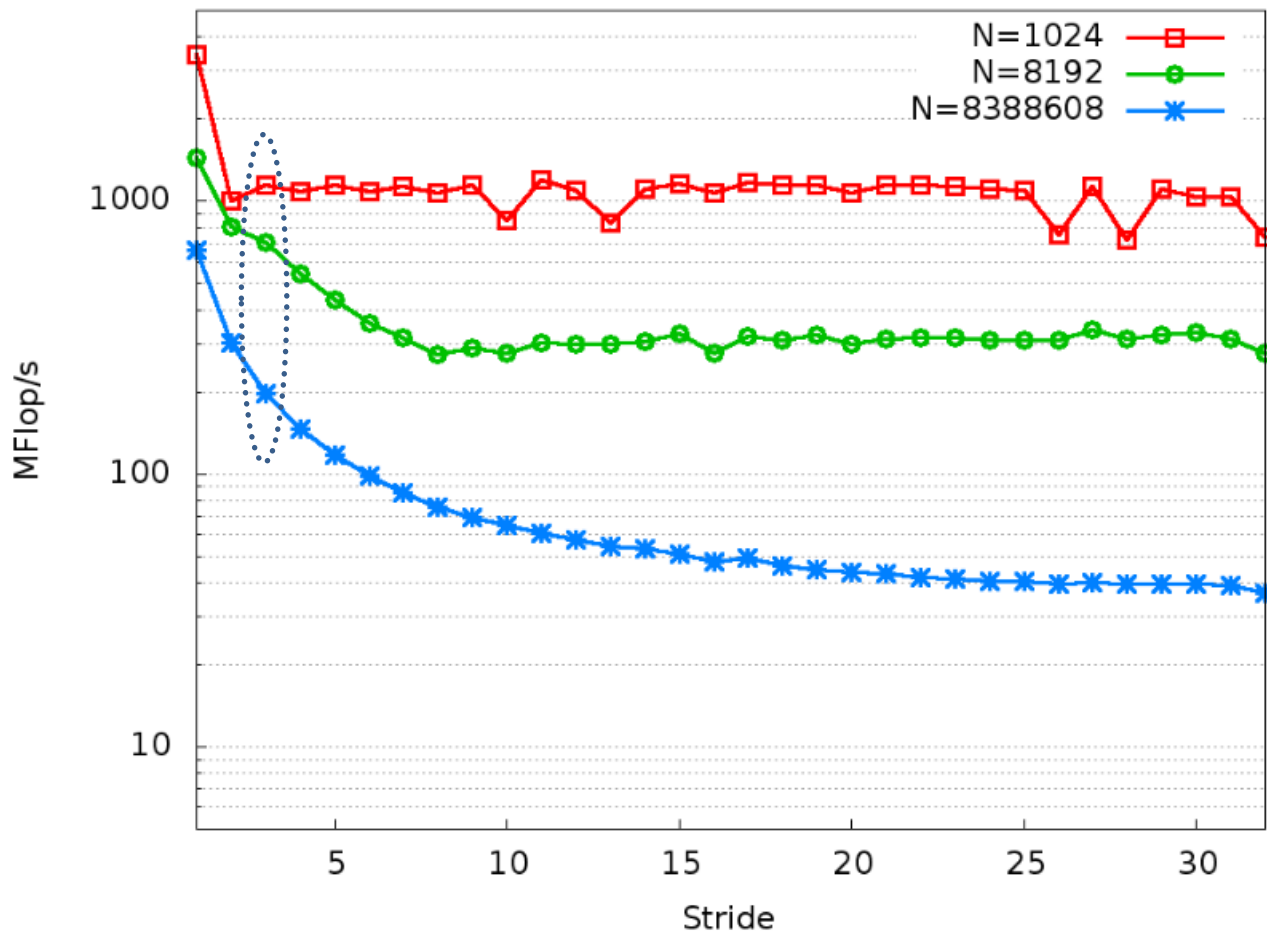
- pre-fetches usually done in hardware
- decision according to memory access pattern

## ■ Pre-Requisite:

- **spatial** locality
- violation of spatial locality:  
if only part of a cache line is used  
→ effective reduction in bandwidth

$$D(::\text{stride}) = A(::\text{stride}) + B(::\text{stride}) * C(::\text{stride})$$

Example: stride 3

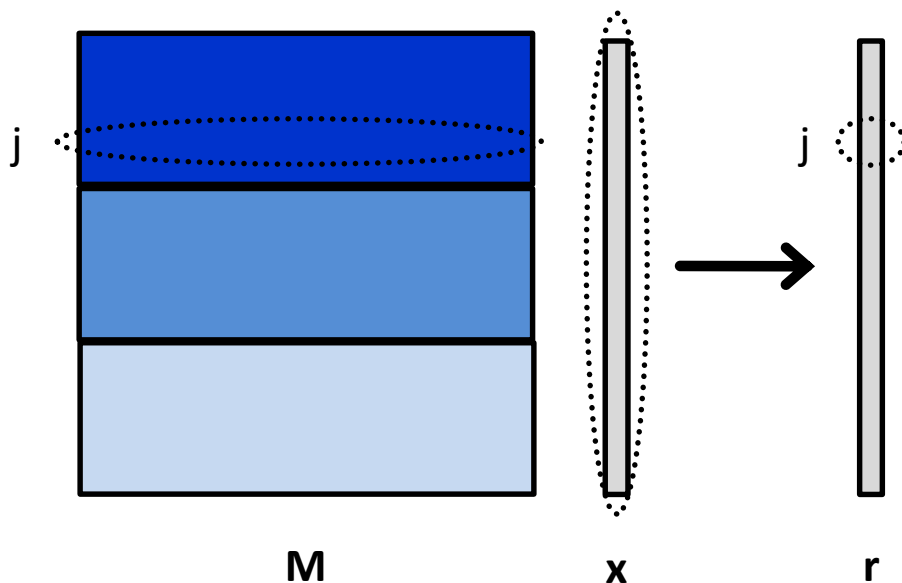


### Notes:

- 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)

- $\mathbf{r} = \mathbf{M} \cdot \mathbf{x}$  i.e.  $r_i = \sum_{j=1}^n M_{ij} x_j$



- **First parallelization attempt:**

```
!$omp parallel
!$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
```

index ordering  
causes non-contiguous  
accesses

- **Parallel patterns used:**

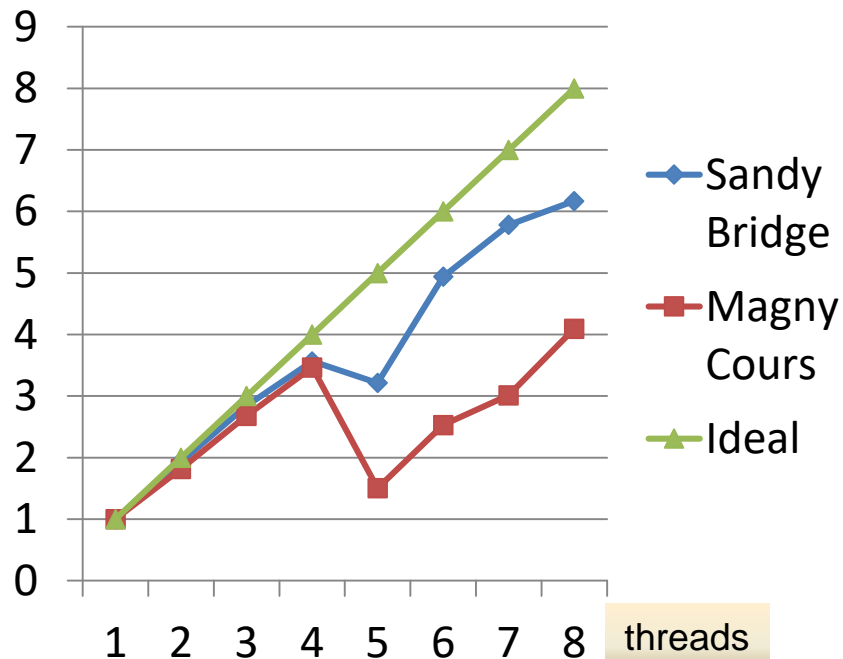
- data decomposition (load balanced)
- loop parallelism (no dependencies)

- **Directive placement:**

- coarse grained parallelism to avoid synchronization overhead

- Speed-Up:  $S(n_t) = \frac{T(1)}{T(n_t)}$

as a function of number of threads  
on 8-core processors

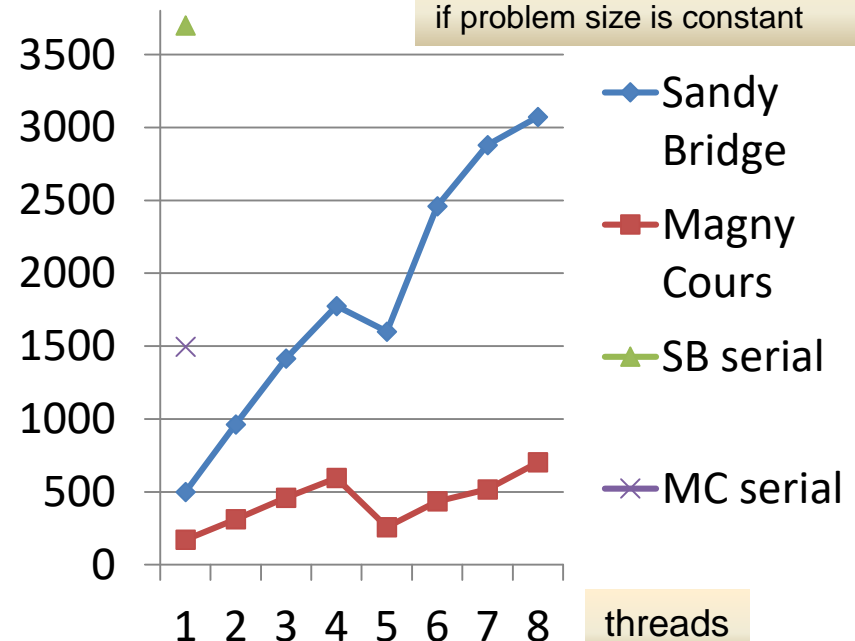


- Scaling **bad** beyond 4 threads

- Absolute performance:

- MFlop/s =  $2 \cdot n^2 / \text{time}$

a measure for execution time  
if problem size is constant

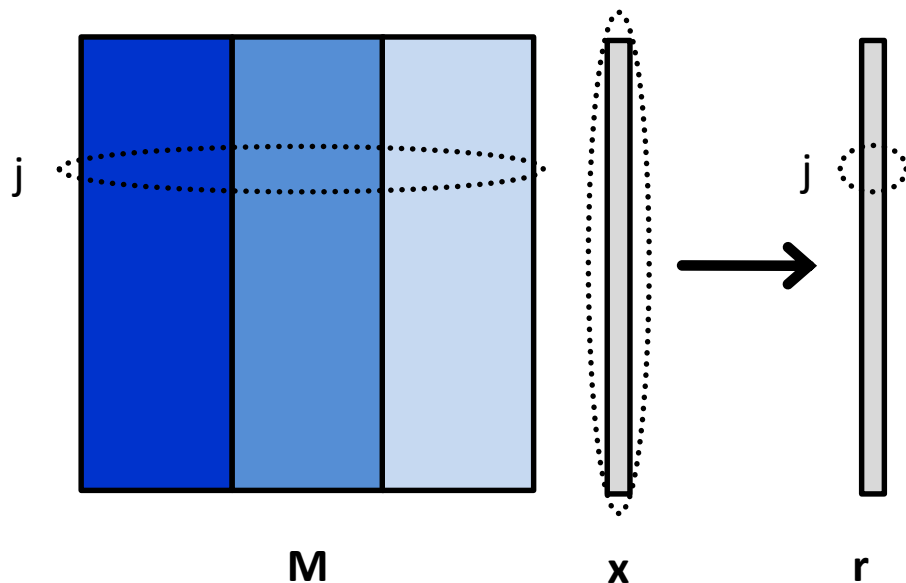


- used `dgemv` for serial run

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

- Switch loop order

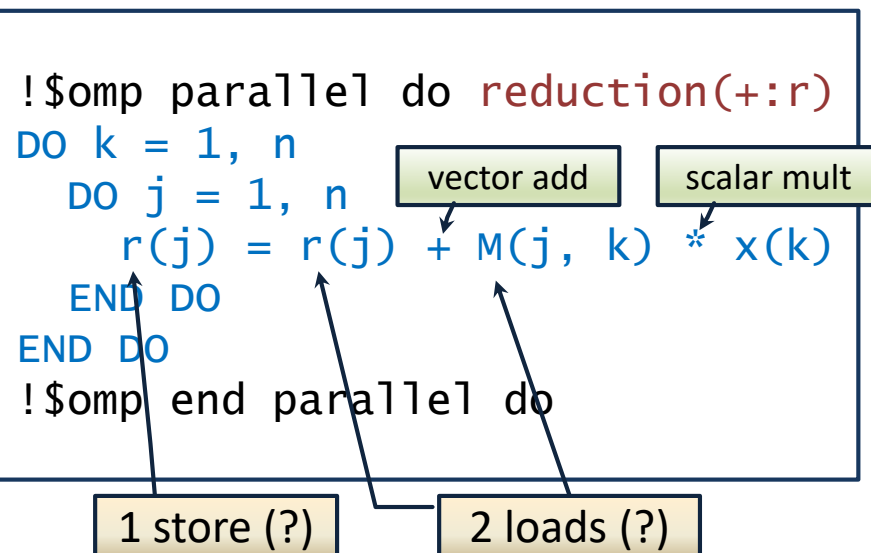
- map **column** blocks to threads:



- color code indicates thread assignment

- Variant 2 of code:

- contiguous** access to M
- array reduction on result vector



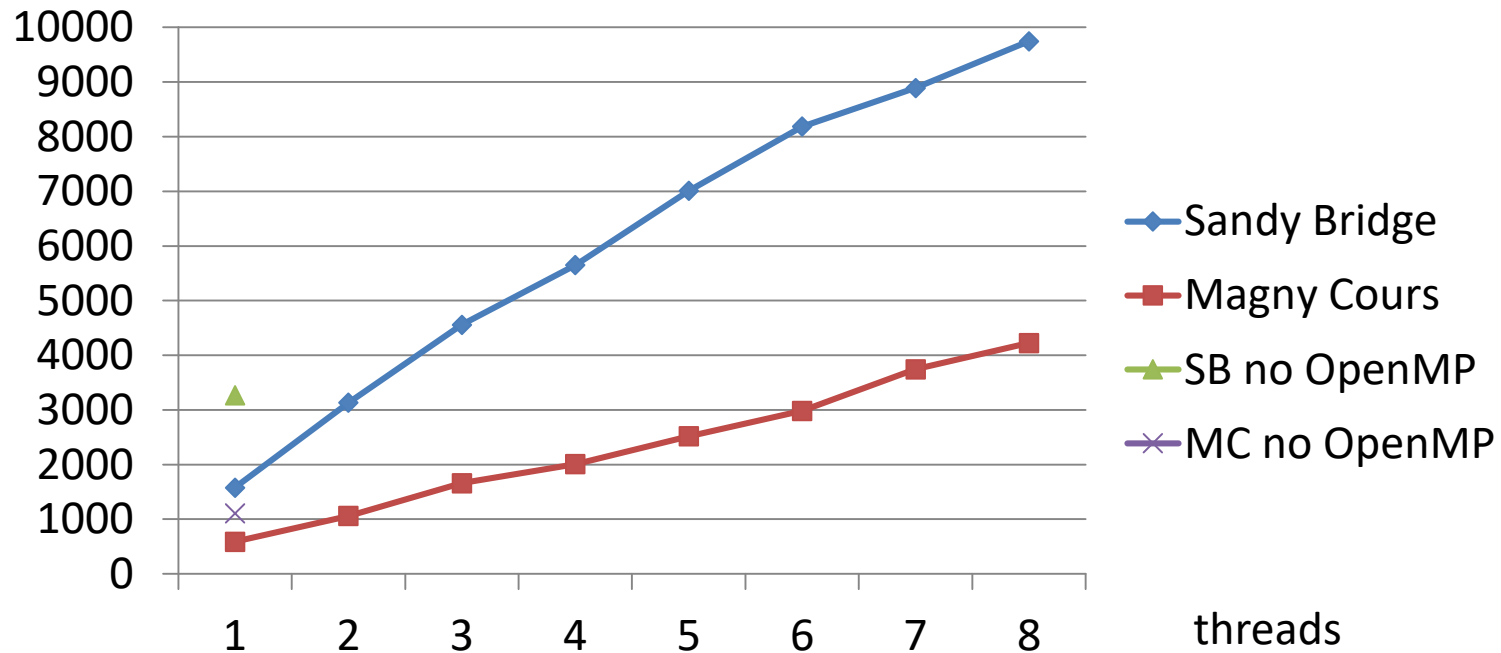
- Performance estimate for single thread:

- double that of triad  $\rightarrow$  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

## Outer loop unrolling

```

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

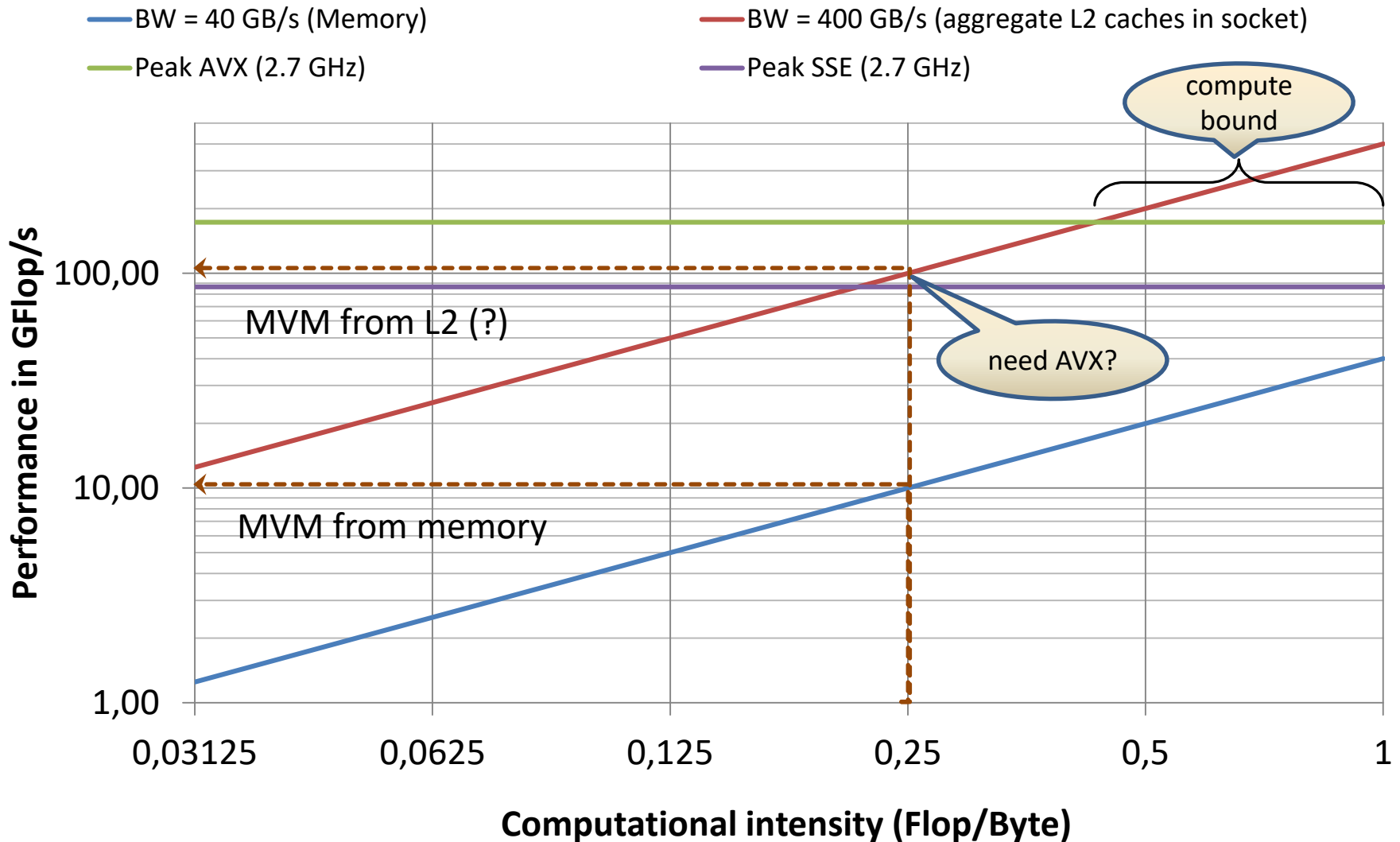
- for M from memory (i.e. **outside** any cache)
- contiguous streaming of data
- assuming 40 GB/s bandwidth for a socket

$$\text{Perf} = \frac{2 \text{ Flop}}{8 \text{ Bytes}} * \frac{40 \text{ GB}}{\text{s}} = 10 \text{ GFlop/s}$$

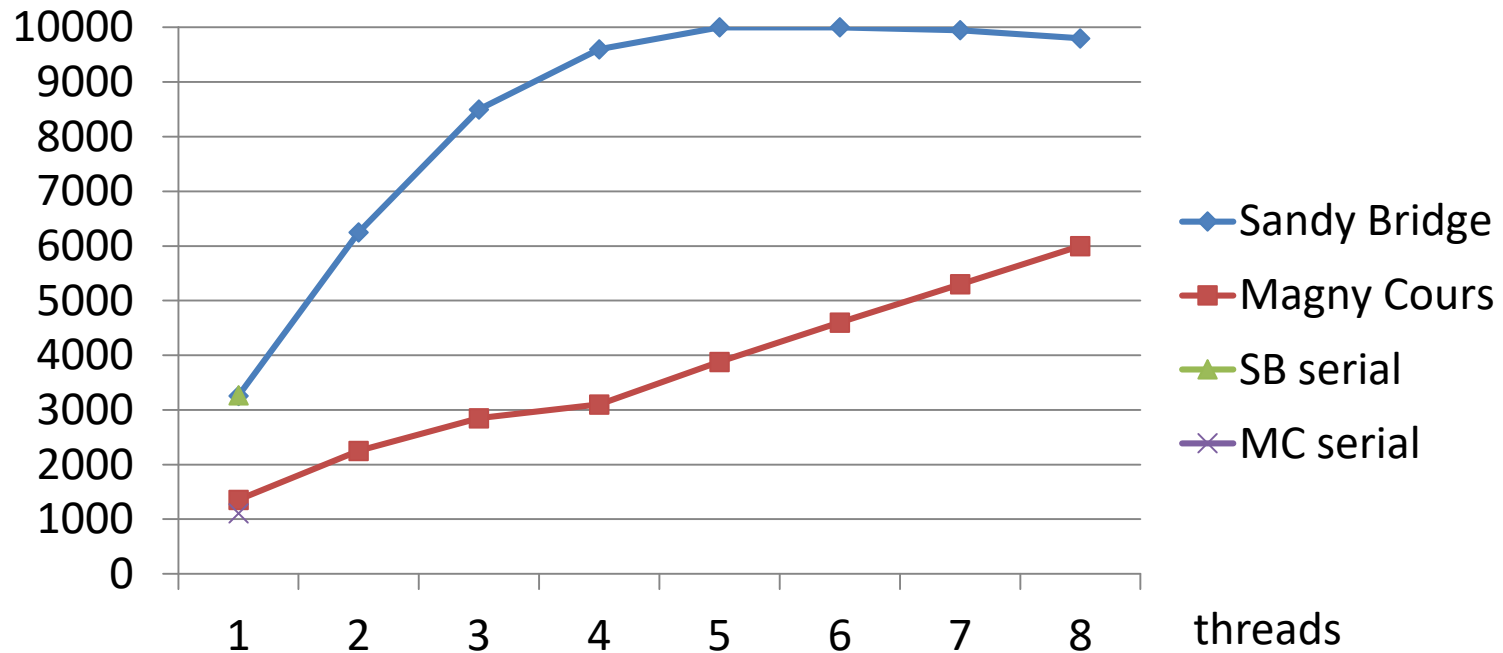
computational intensity

available bandwidth  
(slowest path)

- estimation method is known as „Roofline Model“

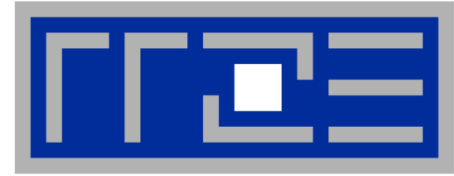


- In MFlop/s. Unroll factors: Sandy Bridge 4, Magny Cours 8



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

- ... 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

- (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:**  
Exploring performance and power properties of modern multicore chips via simple machine models. Preprint: arXiv:1208.2908
- (7) **G. Hager, G. Wellein:** Introduction to High Performance Computing for Scientists and Engineers. Chapman & Hall / CRC (2011)



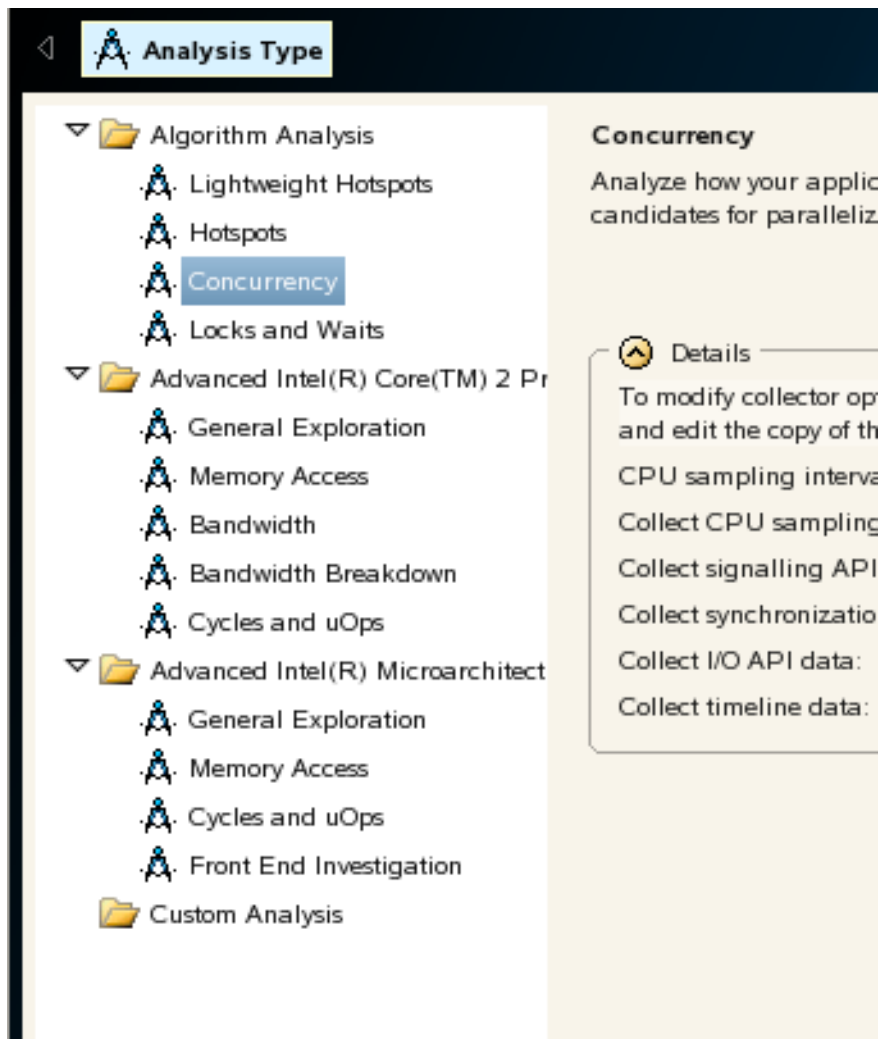
# Appendix: Setting up Vtune Amplifier



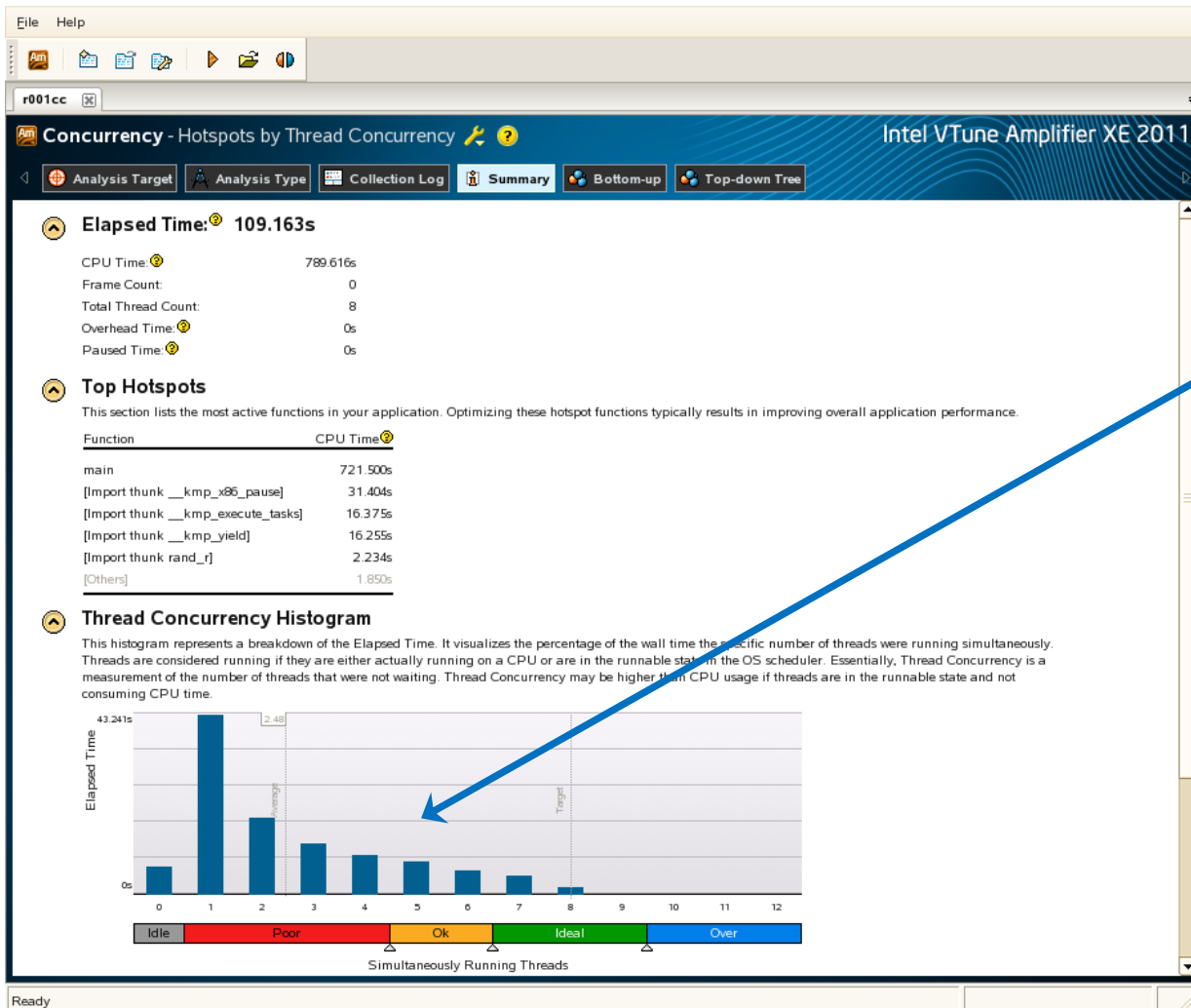


- **Tuning of serial and threaded programs**
  - performance counter access requires group rights
- **Start up GUI**
  - prerequisites: set up environment and possibly stack limit
  - then, invoke the GUI with `amp1xe-gui &`
  - command line `amp1xe-c1` is also available, but will not be discussed
- **Project generation analogous to Intel Inspector**

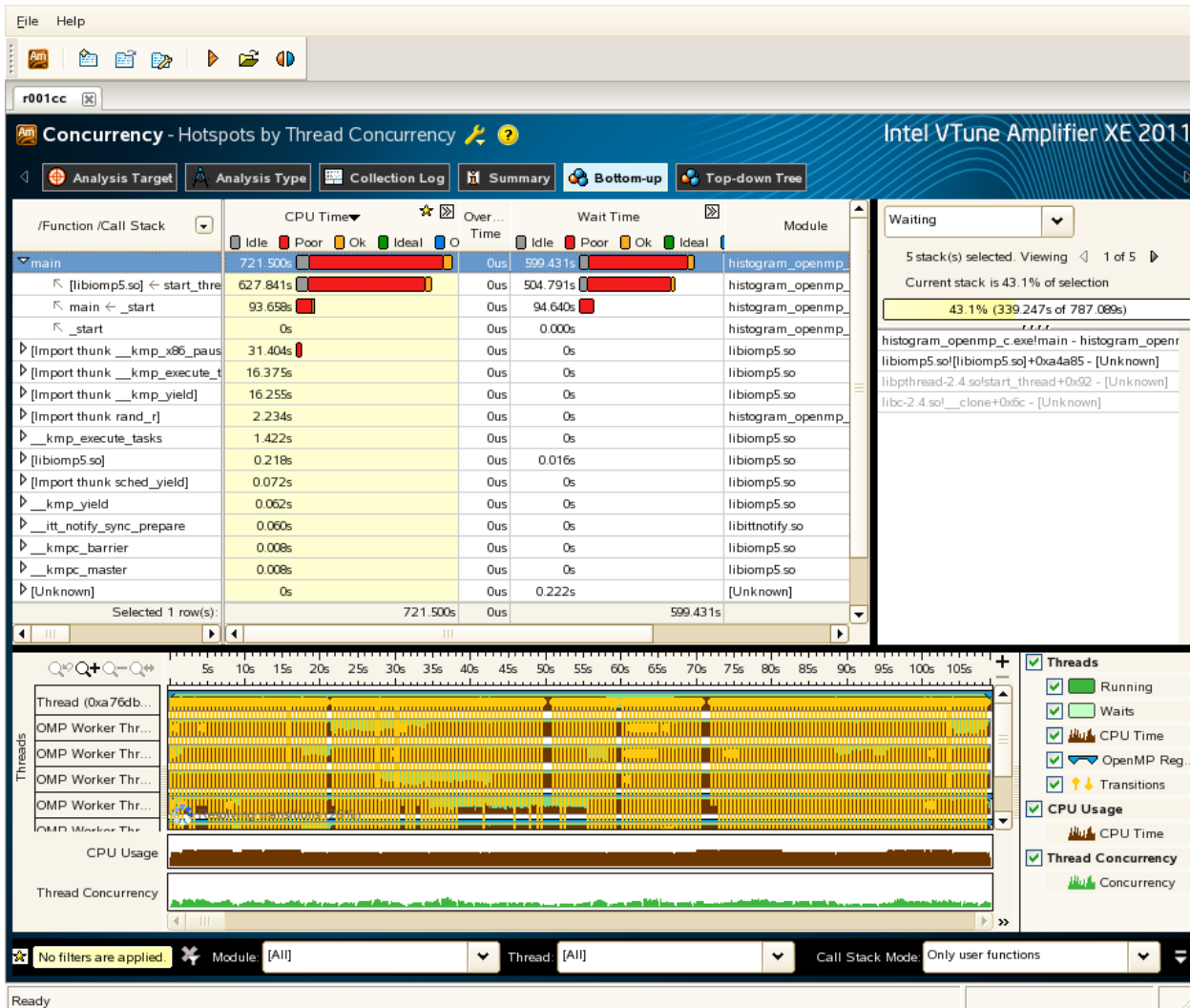
```
#pragma omp parallel private(seed,i,k,me)
{
    me = omp_get_thread_num();
    seed = 123 + 159*me;
    for (k=0; k<100000; ++k) {
#pragma omp for
        for (i=0; i<10000; ++i) {
            ir[i] = rand_r(&seed) & 0xf;
        }
#pragma omp master
        for (i=0;i<10000; ++i) {
            hist[ir[i]]++;
        }
#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:**
  - 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

- Click on routine with significant resource usage

