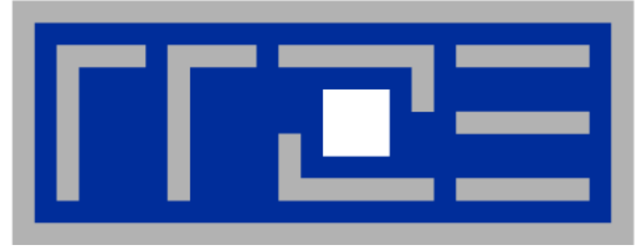


Programming Techniques for Supercomputers:

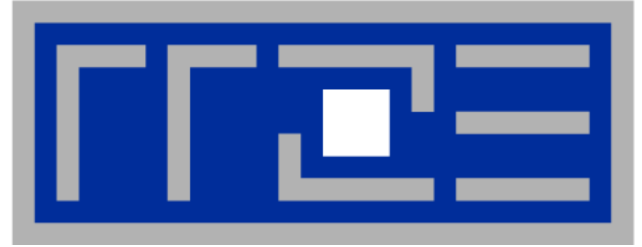
Shared-memory parallel programming with OpenMP

Getting started
Data Scoping
Workload distribution / workshare constructs
Reduction operations
Synchronization
Binding

Prof. Dr. G. Wellein[a,b] , Dr. G. Hager[a], J. Hammer[b], C.L. Alappat[b]

[a]HPC Services – Regionales Rechenzentrum Erlangen
[b]Department für Informatik

University Erlangen-Nürnberg, Sommersemester 2020



Shared-memory parallel processing with OpenMP

Getting started
Data Scoping
Workload distribution / workshare constructs
Reduction operations
Synchronization
Binding

Introduction to OpenMP: Basics

- "Easy", incremental and portable parallel programming of shared-memory computers: OpenMP

- Original design goal: Data level shared memory parallelism – many extensions: Task parallelism, Accelerator offloading, SIMD support,…

- Standardized set of compiler directives & library functions: `http://www.openmp.org/`
  - FORTRAN, C and C++ interfaces are defined
  - Supported by most/all commercial compilers, GNU starting with 4.2
  - Serial code version can be maintained
  - Free tools are available

- R.Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon: Parallel programming in OpenMP. Academic Press, San Diego, USA, 2000, ISBN 1-55860-671-8
- B. Chapman, G. Jost, R. v. d. Pas: Using OpenMP.MIT Press, 2007, ISBN 978-0262533027

Introduction to OpenMP: Software Architecture

Application	User
Compiler Directives	Environment Variables
Runtime Library	
Threads in OS Cores in hardware	

- **Programmer's view:**
  - **directives/pragmas** in application code
  - (a few) library routines
- **User's view:**
  - **environment variables** determine:

    resource allocation

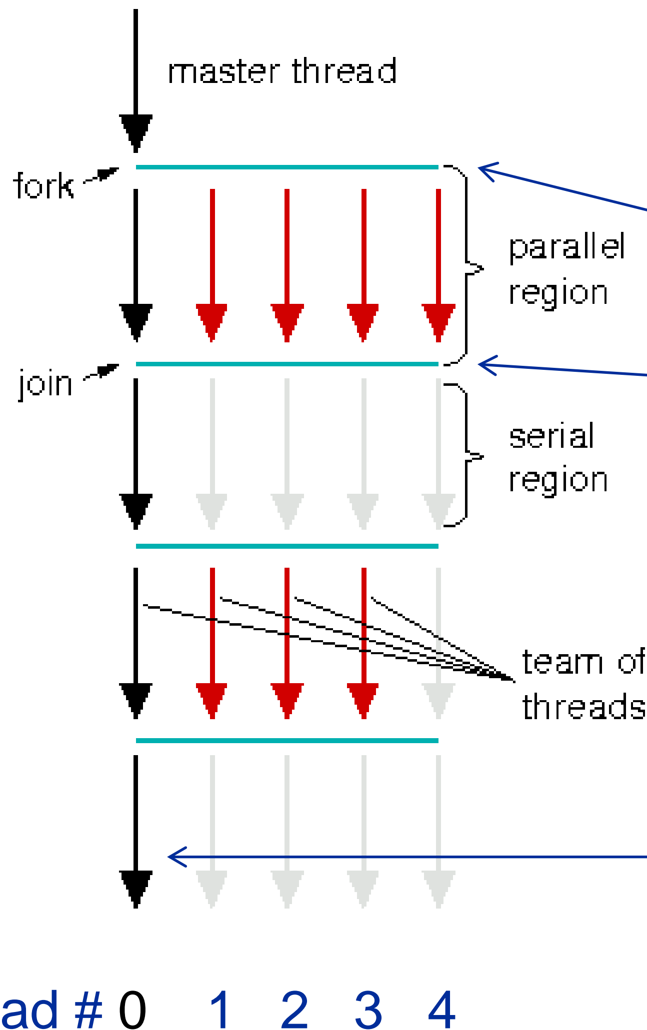
    scheduling strategies and other (implementation-dependent) behavior
- **Operating system view:**
  - parallel work done by **threads**

Central concept of OpenMP programming: Threads

- **Threads:**
  - Spawned by a process
  - Own register set / instruction pointers / stack
  - Share global address space
- **Data: shared or private**
  - shared data available to all threads (in principle)
  - private data only to thread that owns it
- **Data transfer transparent to programmer**

Other threading libs. Available, e.g. pthreads

Introduction to OpenMP: Fork and join execution model



Thread # 0 1 2 3 4

Program start:
one process (master thread) runs

Parallel region: team of threads is generated ("fork")

Synchronize when leaving parallel region ("join")

Serial region:
only master executes
  worker threads usually sleep

Task and data distribution possible via directives

Often best choice 1 thread/core

Introduction to OpenMP: General syntax in C/C++

- Include file: `#include <omp.h>`
- Compiler directive:

```
#pragma omp [directive [clause ...]]
   structured block
```

- Conditional compilation: Compiler's OpenMP switch sets preprocessor macro (acts like `-D_OPENMP`)

```
#ifdef _OPENMP

 ... do something

#endif
```

If OpenMP is not enabled by compiler → redundant comment/not compiled

Introduction to OpenMP: General syntax in Fortran

- Each directive starts with sentinel in column 1:
  - fixed source: !$OMP or C$OMP or *$OMP
  - free source: !$OMP

  followed by a `directive` and, optionally, `clauses`.

- Access to OpenMP library calls:
  - Use include file (omp_lib.h) for API call prototypes (or Fortran 90 module omp_lib if available)
  - Perform conditional compilation of lines starting with !$ or C$ or *$ to ensure compatibility with sequential execution

- Example:

```
   myid = 0
!$ myid = omp_get_thread_num()
   numthreads = 1
!$ numthreads = omp_get_num_threads()
```

- If OpenMP is not enabled by compiler → redundant comment

Introduction to OpenMP: Define parallel region

- `#pragma omp parallel`
  `structured block`

  - Makes structured block a **parallel region**: **All code executed** between start and end of this region is executed **by all threads**.
  - This includes subroutine calls within the region (unless explicitly sequentialized)
  - Local variables inside the block (stack variables) are automatically private to each thread

- `END PARALLEL` required in Fortran

```
    use omp_lib
    …
!$OMP PARALLEL
  call work(omp_get_thread_num(), omp_get_num_threads())
!$OMP END PARALLEL
```

Introduction to OpenMP: Compile and run

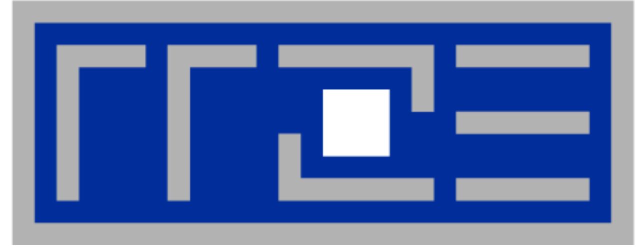
- Compiler must be instructed to recognize OpenMP directives (Intel compiler: `-qopenmp`)
- Number of threads: Determined by shell variable `OMP_NUM_THREADS`

```
$ icc -qopenmp myprog.c
$ export OMP_NUM_THREADS=4
$ ./a.out
```

- Executable should be able to run with any number of threads!
- Thread pinning & core/thread affinity via LIKWID

```
$ export OMP_NUM_THREADS=4
$ likwid-pin -c 0-3 ./a.out
```

→ Map 4 threads to the “first” 4 cores and keep them there



Shared-memory parallel processing with OpenMP

Getting started

Data Scoping

Workload distribution / workshare constructs

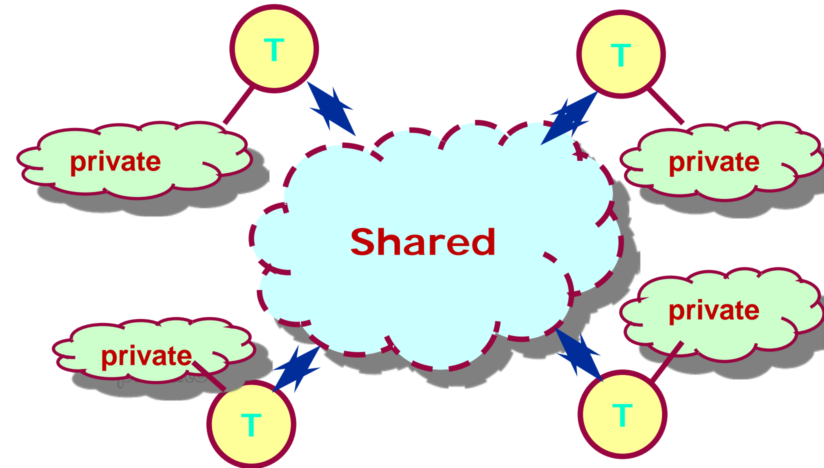
Reduction operations

Synchronization

Binding

Data scoping: Shared vs. private data

The OpenMP memory model



Data in a parallel region can be:

- private to each executing thread
  → each thread has its own local copy of data
- shared between threads
  → there is only one instance of data available to all threads
  → this does not mean that the instance is always visible to all threads!
- OMP clause specifies scope of variables:
  - Specify data scope of private variables in a parallel region:

```
#pragma omp parallel private(var1, tmp) shared(eps)
```

Data scoping: Shared vs. private data

- **Default: All data in a parallel region is shared**
  This includes global data (global/static variables, C++ class variables)

- **Exceptions:**
  1. Loop variables of parallel (“sliced”) loops are private (cf. workshare constructs)
  2. Local (stack) variables within parallel region
  3. Local data within enclosed function calls are private unless declared **`static`**

- **Stack size limits → may be necessary to make large arrays static**
  - This presupposes it is safe to do so!
  - If not: make data dynamically allocated
  - As of OpenMP 3.0: **`OMP_STACKSIZE`** may be set at run time (increase thread-specific stack size):

```
$ setenv OMP_STACKSIZE 100M
```

Data scoping: Private data – example

```
   use omp_lib
   integer myid, numthreads
   …
   myid=0; numthreads=1
!$OMP PARALLEL PRIVATE(myid,numthreads)
!$ myid = omp_get_thread_num()
!$ numthreads= omp_get_num_threads()
  call work(myid, numthreads)
!$OMP END PARALLEL
```

```
   include <omp.h>
 …
#pragma omp parallel{
   int myid=0, numthreads=1;
#ifdef _OPENMP
   myid = omp_get_thread_num();
   numthreads= omp_get_num_threads();
#endif
   work(myid, numthreads);}
```

Local variables are private to each thread!

Data scoping: Simple program example

```
program hello
  use omp_lib
  implicit none
  integer :: nthr, myth

!$omp parallel private(myth, nthr)

  nthr = omp_get_num_threads()

  myth = omp_get_thread_num()

  write(*,*) `Hello from `,myth, &
      & `of `, nthr

!$omp end parallel
end program hello
```

- Parallel region directive:
  - enclosed code executed by all threads („lexical construct“)
  - may include subprogram calls („dynamic region“)
- API function calls:
  - module `omp_lib` provides interface
  - Here: get number of threads (`nthr`) and index of executing thread (`myth=0,…,nthr-1`)
- Data scoping:
  - uses a clause on the directive
  - `myth, nthr` thread-local: private

(will be discussed in more detail later)

Data scoping: Simple program example

```
$ export OMP_NUM_THREADS=8
$ likwid-pin -c 0-7 ./a.out
 Hello from  0 of  8
 Hello from  3 of  8
 Hello from  1 of  8
 Hello from  4 of  8
 Hello from  7 of  8
 Hello from  6 of  8
 Hello from  2 of  8
 Hello from  5 of  8
```

- Ordering of stdout is not defined
- Avoid extensive output to stdout in parallel regions!

Data scoping: Important side effects

- **Incorrect `shared` attribute may lead to**
  - **correctness** issues ("race conditions")
  - **performance** issues ("false sharing")

Incorrect result

(Very) slow

- **Scoping of local function data and global data**
  - is not changed
  - compiler cannot be assumed to have knowledge
- **Recommendation: Use**
  `#pragma omp parallel default(none)`
  - to not overlook anything
  - compiler complains about every variable that has no explicit scoping attribute

```
real :: s

  s = …
!$omp parallel private(s)

  s = …
  … = … + s


!$omp end parallel
  … = … + s
```


- **Masking relevant for**
  - privatized variables defined in scope outside the parallel region

- **Masking also applies to**
  - association status of pointers
  - allocation status of allocatable variables

Data scoping: Private data – Initial / final values

- **What if initialization of privatized variables is required?**
  - `FIRSTPRIVATE( var )` clause for setting each private copy to the previous global value

- **What if value of last iteration is needed after the loop?**
  - `LASTPRIVATE( var )`

  `var` is updated by the thread that computes
    - **the sequentially last iteration (on `do` or `for` loops)**
    - **the last section**

(What if a global (or COMMON) variable needs to be privatized?
  - `THREADPRIVATE` / `COPYIN`
  - cf. standards documents)

The firstprivate clause

```
real :: s

  s = …
!$omp parallel &
!$omp firstprivate(s)
  …
  s = … + s
  call foo(…)
  …= … + s
!$omp end parallel
  … = … + s
```

shared

private

S

fork:

T0 T1 T2 T3

S

S_0 S_1 S_2 S_3

persists (inaccessible)

S

S_0 S_1 S_2 S_3

S

join

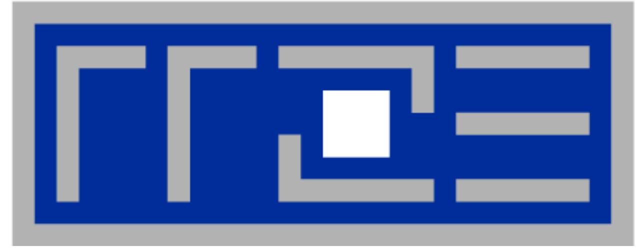
time

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

The lastprivate clause

```
real :: s

  s = …
!$omp parallel &
!$omp lastprivate(s)
!$omp do
  do i = …
      s = …
  end do
!$omp end do
!$omp end parallel
  … = … + s
```

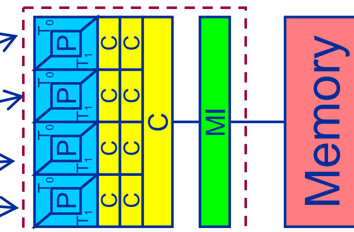

- **Extension of private:**
  - value from thread which executes last iteration of loop is transferred back to master copy
  - restrictions similar to **`firstprivate`**



Shared-memory parallel processing with OpenMP

Getting started
Data Scoping
Workload distribution / workshare constructs
Reduction operations
Synchronization
Binding

```
integer i, N
dp, dimension(N):: a,b,c,d
    …
do i=1,N
   a(i)=b(i)+c(i)*d(i)
enddo
```



Workload distribution: Manual loop scheduling

```
    use omp_lib
    integer tid, numth, i, bstart, bend, blen, N
    double precision, dimension(N):: a,b,c,d
    …
!$OMP PARALLEL PRIVATE(tid, numth, bstart, bend, blen, i)
    tid=0; numth=1
!$  tid   = omp_get_thread_num()
!$  numth = omp_get_num_threads()
    blen = N/ numth
    if(tid.lt.mod(N,numth)) then
       blen  =blen+1
       bstart=blen*tid+1
    else
       bstart=blen*tid+mod(N,numth)+1
    endif
    bend=bstart+blen-1
    do i=bstart,bend
       a(i)=b(i)+c(i)*d(i)
    enddo
!$OMP END PARALLEL
```

Equally distribute workload: Each thread gets a consecutive chunk of the loop iterations

Not a low overhead solution……

Workload distribution: Workshare construct

`!$OMP DO[clause]` declares the **loop** following to be divided between threads if within a parallel region ("sliced")

```
   integer i, N
   double precision, dimension(N):: a,b,c,d
   …
!$OMP PARALLEL
!$OMP DO             ! Parallelize loop
  do i=1,N
      a(i)=b(i)+c(i)*d(i)
   enddo
!$OMP END DO
!$OMP END PARALLEL
```

- Loop counter of parallel loop is declared private implicitly
- No impact in serial region ("orphaned directive")
- Implicit thread synchronization at END DO and END PARALLEL
- Suppress barrier at END DO: clause= `NOWAIT`

Workload distribution: Combined workshare construct

`!$OMP PARALLEL DO[clause]` Combined workshare construct

```
   integer i, N
   double precision, dimension(N):: a,b,c,d
   …
!$OMP PARALLEL DO ! Fork team of threads & parallelize
  do i=1,N
      a(i)=b(i)+c(i)*d(i)
   enddo
!$OMP END PARALLEL DO
```

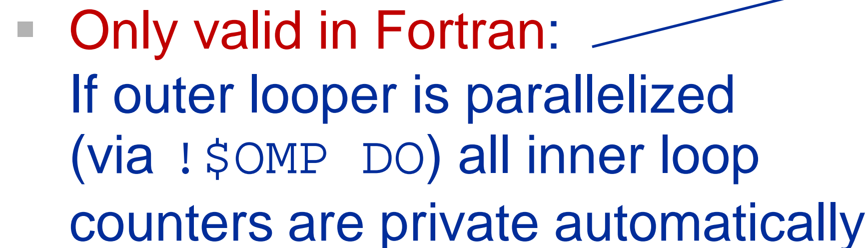
Workload distribution: Worksharing constructs

#pragma omp for [clause] & !$OMP DO [clause]

- Only the loop immediately following the directive is workshared
- Restrictions on parallel loops (especially in C/C++)
  - trip count must be computable (no `do while`)
  - loop body with single entry and single exit point

- Standard random access iterator loops are supported by OpenMP 3.0:

```
#pragma omp for
for(vector::iterator i=v.begin(); i!=v.end(); ++i) {
   ... do stuff using *i etc ... }
```



- Only valid in Fortran:
  If outer looper is parallelized
  (via `!$OMP DO`) all inner loop
  counters are private automatically

```
!$OMP PARALLEL DO
  do i=1,N
     do j=1,N
        a(i,j)=b(j,i)
     enddo
   enddo
!$OMP END PARALLEL DO
```

Workload distribution: Worksharing constructs (general)

- **Distribute the execution of the enclosed code region among the members of the team**
  - Must be enclosed dynamically within a parallel region
  - Threads do not (usually) launch new threads
  - No implied barrier on entry
  - Implicit barrier at end of worksharing (unless NOWAIT is specified)

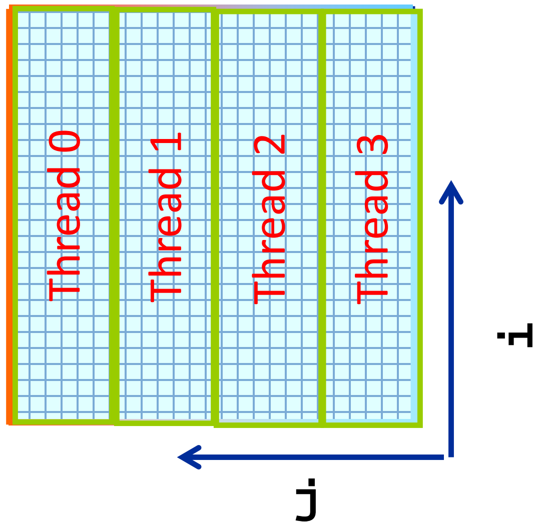
- **Directives**
  - `do` directive (Fortran), `for` directive (C/C++)

  - `section(s)` directives (we ignore this)
  - `workshare` directive (Fortran 90 only)

  - Tasking constructs (advanced – available since OpenMP 3.0)

Workload distribution: Worksharing constructs (example)

- **Making parallel regions useful …**
  - divide up work between threads
- **Example:**
  - working on an array processed by a nested loop structure




```
real :: a(ndim, ndim)
…
!$omp parallel
!$omp do
do j=1, ndim  ! sliced
  do i=1, ndim
    …
    a(i, j) = …
  end do
end do
!$omp end do    synchronization
  …  ! further parallel stuff
!$omp end parallel
```

- iteration space of **directly nested loop** is sliced

Introduction to OpenMP: Workshare constructs (clauses)

- *clause* can be one of the following:
  - `private, firstprivate, lastprivate`
  - `nowait` [see below]
  - `collapse(`*n*`)` [see below]
  - `reduction` (*operator* : *list*) [see later]
  - `schedule` (*type* [, *chunk*]) [see next slide]
  - ... and a few others

- **Implicit barrier at the end of loop unless `nowait` is specified**
- **If `nowait` is specified, threads do not synchronize at the end of the parallel loop**

- **`collapse`: Fuse nested loops to a single (larger one) and parallelize it**

- **`schedule` clause specifies how iterations of the loop are distributed among the threads of the team.**

Introduction to OpenMP: `schedule` clause

Within `schedule(` *type* [, *chunk*] `)` *type* can be one of the following:

- **`static`**: Iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number.
  Default chunk size: one contiguous piece for each thread.
- **`dynamic`**: Iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. *Default chunk size: 1.*
- **`guided`**: The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.
  chunk specifies the smallest piece (except possibly the last).
  Default chunk size: 1. Initial chunk size is implementation dependent.
- ***`runtime`***: The decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable.
- ***`auto`***: Compiler decides

Default `schedule`: implementation dependent.

Introduction to OpenMP: `schedule` clause

Iteration	`STATIC`	`STATIC,3`	`DYNAMIC[,1]`	`DYNAMIC,3`	`GUIDED[,1]`
1	T0	T0	T0	T0	T0
2	T0	T0	T1	T0	T0
3	T0	T0	T2	T0	T0
4	T0	T1	T0	T1	T0
5	T0	T1	T1	T1	T0
6	T0	T1	T0	T1	T0
7	T0	T2	T2	T2	T1
8	T1	T2	T2	T2	T1
9	T1	T2	T0	T2	T1
10	T1	T0	T1	T1	T2
11	T1	T0	T2	T1	T2
12	T1	T0	T1	T1	T2
13	T1	T1	T1	T2	T0
14	T1	T1	T0	T2	T0
15	T2	T1	T2	T2	T1
16	T2	T2	T2	T0	T1
17	T2	T2	T1	T0	T0
18	T2	T2	T2	T0	T1
19	T2	T0	T1	T2	T2
20	T2	T0	T2	T2	T0

T0 T1 T2

Introduction to OpenMP: `schedule` clause

- **Dense matrix-vector multiplication**

```
#pragma omp parallel
{
  for(int j=0; j<niter; j++){

#pragma omp for schedule(...)
    for(int m=0; m<size; m++){
      for(int n=0; n<size; n++){
          y[m]+=a[m*size+n]*x[n];}
    }
    if(y[size>>1]<0){
      dummy(a,x,y,0);
    }

  }
}
```

Parallel execution starts here!

Workload Distribution: Every Thread receives “some” m-iterations

Full inner loop is executed by every thread for it’s m-iterations

Where is the private data????

- Dense triangular MVM

```
#pragma omp parallel
{
  for(int j=0; j<niter; j++){

#pragma omp for schedule(...)
    for(int m=0; m<size; m++){
      for(int n=m; n<size; n++){
        y[m]+=a[m*size+n]*x[n];
      }
    }
    if(y[size>>1]<0){
      dummy(a,x,y,0);
    }

  }
}
```

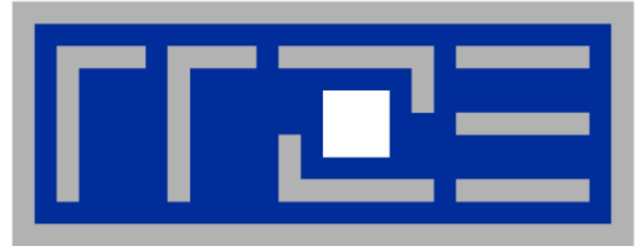
Excercise

- Dense MVM + scalar product

```
#pragma omp parallel
{
  for(int j=0; j<niter; j++){

#pragma omp for schedule(...)
    for(int m=0; m<size; m++){
      for(int n=m; n<size; n++){
        y[m]+=a[m*size+n]*x[n];
      }
    }
    s=0;
    for(int m=0; m<size;m++)
      s+=x[m]*y[m];
  ...
  }
}
```

?

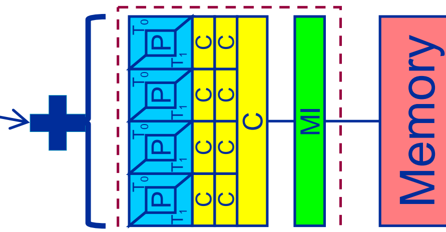
Reduction



Shared-memory parallel processing with OpenMP

Getting started
Data Scoping
Workload distribution / workshare constructs
Reduction operations
Synchronization
Binding

```
integer i, N
dp, dimension(N):: a,b
s=0.d0
!$omp parallel do private(s)
do i=1,N
   s=s+a(i)*b(i)
enddo
! How to sum up the s?
!$omp end parallel
```



Reduction operations: reduction(:) clause

Reduction clause (do/parallel directive): **`reduction( operation, variable )`**

```
real :: s

!$omp parallel
!$omp do reduction(+:s)
  do i = …
       :
       :
     s = s + …
  end do
!$omp end do
  … = … * s
!$omp end parallel
```


Reduction variable must be shared in enclosing context!

- **At synchronization point:**
  - reduction operation is performed
  - result is transferred to master copy
  - restrictions similar to `firstprivate`

Reduction operations: Basics

Initial values of reduction variable depend on operation

C / C++ has an analogous set

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	0
IOR	0

- **Consistency required**
  - operation specified in clause vs. update statement
  - rely on algebraic rules!
  - `X = expr - X` is **not** allowed

- **Multiple reductions:**
  - multiple scalars, operations or an array:

```
real :: x, y, z
!$OMP do reduction(+:x, y, z)
```

```
!$OMP do reduction(+:x, y) &
!$OMP    reduction(*:z)
```

```
real :: a(n)
!$OMP do reduction(*:a)
```

Reduction operations: Example

- Dense MVM + scalar product

```
double s;
…
#pragma omp parallel
{
  for(int j=0; j<niter; j++){

#pragma omp for schedule(...)
    for(int m=0; m<size; m++){
      for(int n=m; n<size; n++){
        y[m]+=a[m*size+n]*x[n];
      }
    }
 …
#pragma omp for reduction(+:s)
    for(int m=0; m<size;m++) s+=x[m]*y[m];
…
  }
}
```

Reduction operations: Max. value

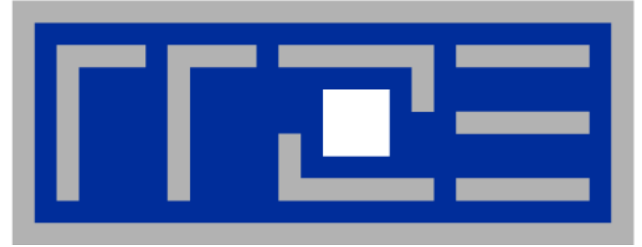
```
double precision, dimension(0:N+1,0:N+1,0:1) :: phi
Double precision :: maxdelta, eps
Integer :: t0,t1,i,k,it
eps=1.d-14;maxdedlta=2.d0*eps; t0=0;t1=1

do while(maxdelta.gt.eps)
   maxdelta=0.d0

!$OMP parallel do reduction(max:maxdelta)
   do k=1,N
      do i=1,N
         phi(i,k,t1)=0.25d0*(phi(i+1,k,t0)+phi(i-1,k,t0)+
                             phi(i,k+1,t0)+phi(i,k-1,t0) )
         maxdelta=max( maxdelta, abs(phi(i,k,t1)-phi(i,k,t0)) )
      enddo
   enddo
!$OMP end parallel do
    t0 ←→ t1

enddo
```

Reduction operations: Arrays

- **Reduction operations performed on arrays: elementwise**

```
!$omp parallel do reduction(+:y)
do c = 1 , C
  do r = 1 , R
     y(r) = y(r) + A(r,c) * x(c)
  enddo
enddo
!$omp end parallel do
```

```
1 double *v = (double*)malloc(N*sizeof(double));
2 ...
3 #pragma omp parallel reduction(+:v[0:N])
4 {
5   for(int i=0; i<W; ++i) {
6     int idx = calc_idx(i,N);
7     v[idx] += workfunc(idx);
8 }
```



Shared-memory parallel processing with OpenMP

Getting started
Data Scoping
Workload distribution / workshare constructs
Reduction operations
Synchronization
Binding

Synchronization/Serialization: Why do we need it?

Multiple threads update/write a shared variable, e.g. a = a + 1

race condition!

T

T

private

private

Shared Memory

a = a + 1

private

private

T

T

Synchronization: Barrier synchronization

- **Explicit via directive:**

  `!$omp barrier`

  - synchronization requirement:

    the execution flow of each thread blocks upon reaching the barrier until **all** threads have reached the barrier
  - flush synchronization of all accessible shared variables happens before all threads continue → after the barrier, all shared variables have consistent value visible to all threads (similar to `volatile` argument)
  - barrier may **not** appear within work-sharing code block (e.g. `!$omp do` block) →potential of deadlock – different loop iterations per thread

- **Implicit (barrier) for some directives:**
  - at the **beginning and end** of parallel regions
  - at the **end** of `do, single, sections, workshare` blocks unless a `nowait` clause is allowed and specified
  - all threads in the executing team are synchronized
  - this is what makes these directives “easy-to-use”

Always remember: Synchronization/barrier costs time → performance overhead

Relaxing synchronization requirements

- **Use a `nowait` clause**
  - on end do / end sections / end single / end workshare (Fortran)
  - on for / sections / single (C/C++)
  - **removes** the synchronization at end of block
  - potential performance **improvement** (especially if load imbalance occurs within construct)

```
              !$omp parallel
              !$omp do shared(a)
                ... (loop)
Thread 0 →      a(i) = ...
              !$omp end do nowait
                ... ! some other parallel work (don’t reference a)
              !$omp barrier
Thread 1 →      ... = a(i)   ! after deferred barrier
              !$omp end parallel
```

threads continue without waiting

- programmer’s responsibility to prevent races

Cost of synchronization: Example dMVM (1)

- **To keep synchronization/barrier costs low → parallelization of outermost loops**

```
void dmvm(int n, int m, double *lhs, double *rhs, double *mat){
…
#pragma omp parallel for private(offset,r) schedule(static){

 for(c=0; c<n; ++c)
    offset=m*c;

    for(r=0; r<m; ++r)
      lhs[r] += mat[r + offset]*rhs[c];
}
}
```

One barrier only!

BUT:
Incorrect result:
Race condition on `lhs`

Cost of synchronization: Example dMVM (2)

- **Use inner loop parallelization → correct result!**

```
void dmvm(int n, int m, double *lhs, double *rhs, double *mat){
…

#pragma omp parallel private(offset,c){
 for(c=0; c<n; ++c)
    offset=m*c;
#pragma omp for schedule(static)
    for(r=0; r<m; ++r)
      lhs[r] += mat[r + offset]*rhs[c];
}
}
```

Parallel region started once

But: `n` barriers to be executed!

Correct result: Threads work on separate parts of `lhs`

Cost of synchronization: Example dMVM (3)

- Use inner loop parallelization → correct result!
- Use `nowait` to avoid (useless) synchronization

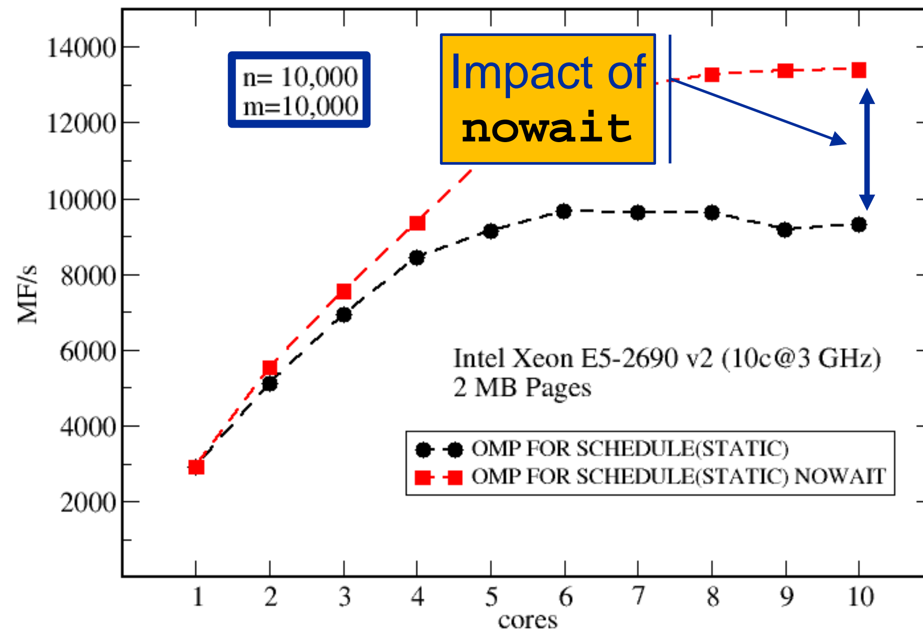
```
void dmvm(int n, int m, double *lhs, double *rhs, double *mat){
...

#pragma omp parallel private(offset,c){
 for(c=0; c<n; ++c)
    offset=m*c;
#pragma omp for nowait schedule(static)
    for(r=0; r<m; ++r)
      lhs[r] += mat[r + offset]*rhs[c];
}
}
```

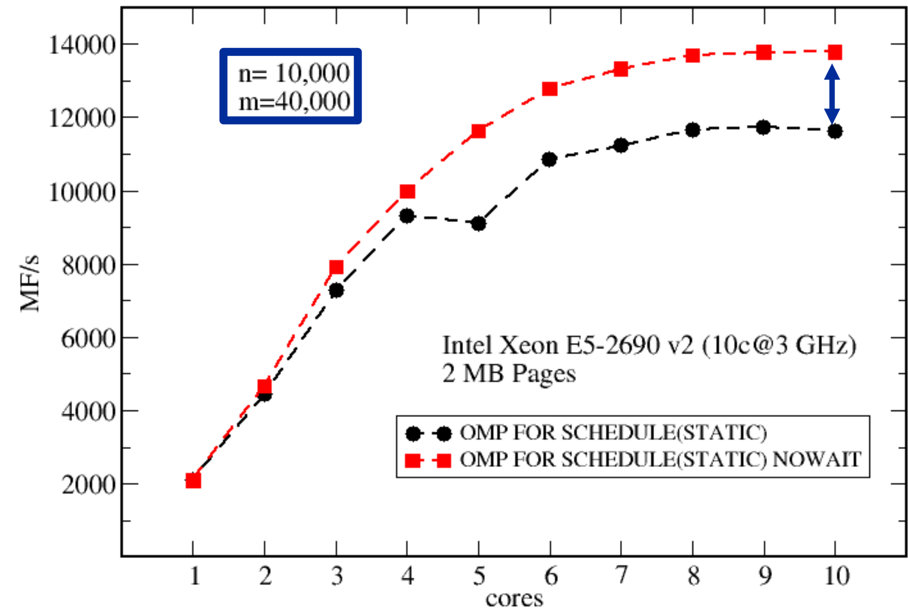
Parallel region started once

NO barrier

Correct result: Threads work on separate parts of `lhs`

Cost of synchronization: Example dMVM (4)

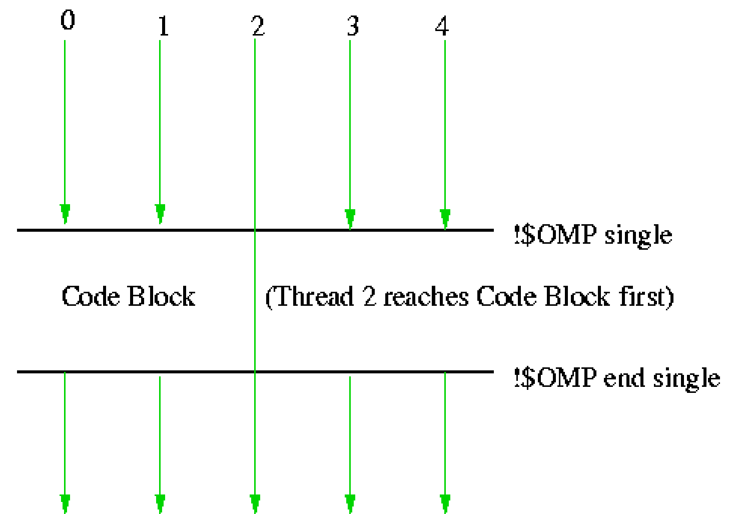




- Barrier overhead may substantially decrease performance
- Performance impact decreases as inner loop length (`m`, i.e. work performed per barrier) increases (see `m=40,000` vs. `m=10,000`)
- Use `nowait` with due care for correctness reasons!
- Can we explain performance numbers and impact of barrier? → See roofline lecture

Synchronization Constructs: `single` directive

- The enclosed code is executed by exactly one thread, which one is unspecified
- C/C++:
  `#pragma omp single` *[clause[[,]clause]…] [nowait] new-line*
  structured-block
- The other threads in the team skip the enclosed section of code and continue execution.
- **Implied barrier** at the exit of the `single` section!
- `single` may not appear within a `parallel do` (deadlock!)
- `nowait` clause at start of parallel region suppresses synchronization



Synchronization: `master` directive

Fortran:

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

C/C++:

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

- Only thread zero (from the current team) executes the enclosed code block
- Other threads continue without synchronization
- Not all threads must reach the construct

Synchronization: `critical` regions

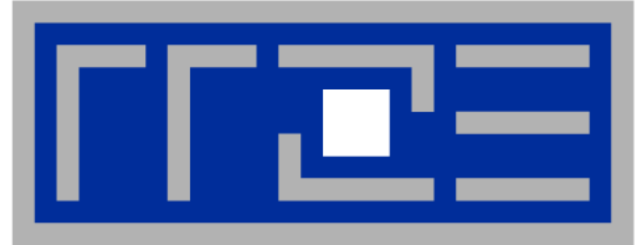
- **The `critical` and `atomic` directives:**
  - **each** thread reaching the critical statement, executes code (in contrast to `single`)
  - but only **one at a time** within code
  - `atomic`: code block must be a **single line** update of a scalar entity with an intrinsic operation

Fortran:

```
!$omp critical                 !$omp atomic
  block                          x = x <op> ...
!$omp end critical
```

C/C++:

```
# pragma omp critical          # pragma omp atomic
  { block }                      x = x <op> ... ;
```



Shared-memory parallel processing with OpenMP

Getting started
Data Scoping
Workload distribution / workshare constructs
Reduction operations
Synchronization
Binding

Binding of Directives

Which parallel region does a directive refer to?

- `do/for, sections, single, master, barrier, task:`

  to (dynamically) closest enclosing parallel region, if one exists

  if not → directive is "orphaned": only one thread used if not bound to a parallel region

  - close nesting of `do, sections` **not** allowed
  - close nesting of barriers inside explicit `task`s (see later) **not** allowed

- `atomic, critical`**: mutual exclusion applies for all threads, not just current team**

Orphaning

```
!$OMP parallel
   …
   call foo(…)
  …
!$OMP end parallel
   call foo(…)
```

```
subroutine foo(…)
   …
!$OMP do
   do I=1,N
    …
   end do
!$OMP end do
```

Inside parallel region:

`foo` called by all threads

Outside parallel region:

`foo` called by one thread

- **OpenMP directives in `foo` are orphaned**
  - since they may or may not bind to a parallel region
  - decided at runtime
  - in both cases executed correctly

Example for incorrect nesting

```
!$OMP parallel
!$OMP do
   do i=1,n
      call foo(…)
   end do
!$OMP end do
!$OMP end parallel
```

```
subroutine foo(…)
   …
!$OMP do
   do I=1,N
    …
   end do
!$OMP end do
```

Not allowed:

`do` **nested within a** `do`