Shared-Memory Parallel Programming with OpenMP

An Introduction

July 5.-6. 2016 | Alexander Schnurpfeil, presented by Jochen Kreutz |

Literature

- OpenMP Architecture Review Board, "OpenMP Application Program Interface", Version 4.0 July 2013.
- R. Chandra et al., "Parallel Programming in OpenMP", Morgan Kaufmann Publishers, San Francisco (2001).
- B. Chapman et al., "Using OpenMP - PORTABLE SHARED MEMORY PARALLEL PROGRAMMING", MIT Press, Cambridge (2008).
- S. Hoffmann, R. Lienhart, "OpenMP - Eine Einführung in die parallele Programmierung mit C/C++", Springer, Berlin (2008).
- `http://openmp.org/wp`

Acknowledgements

- Rolf Rabenseifner for his comprehensive course on MPI and OpenMP
- Marc-André Hermanns for his course material on MPI and OpenMP

Outline

Open Multi-Processing – OpenMP

- Part I: Introduction
  - OpenMP - some general hints
  - Basic OpenMP concepts
  - Typical OpenMP example
  - Data environment
  - Paralellization of loops
- Part II: Further frequently used features
- Part III: Miscellaneous
- Part IV: Debugging and analysis
  - Race conditions and data dependency

Shared-Memory Parallel Programming with OpenMP

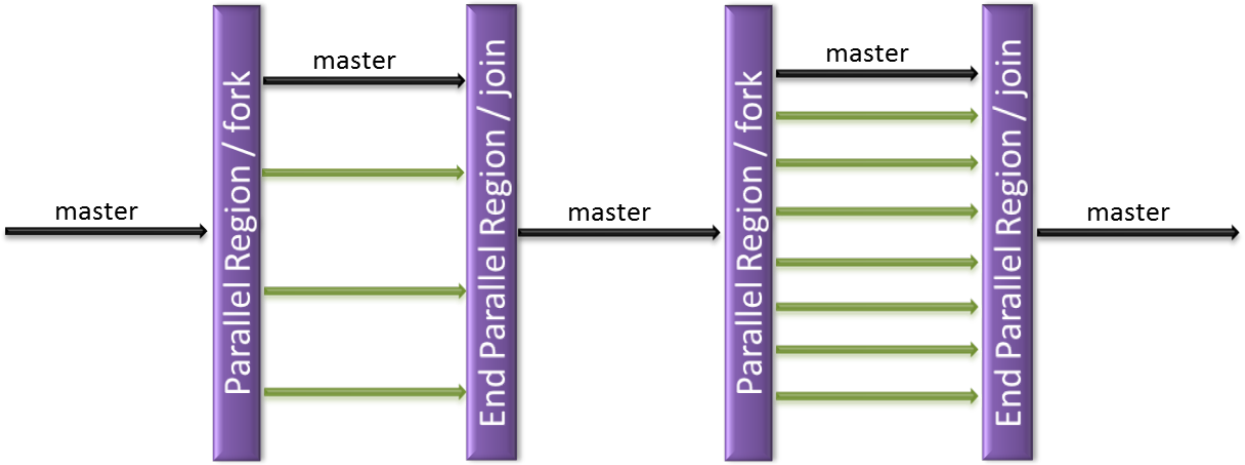
Part I: Introduction

July 5.-6. 2016 | Alexander Schnurpfeil, presented by Jochen Kreutz

OpenMP - Some General Hints

- Portable shared memory programming
- Easy to learn
  - OpenMP specific commands in source codes are processed by the compiler
  - OpenMP functionality is switched on by a compiler specific option
- Parallelization is fully controlled by programmer
  - Directives for Fortran 77/90 and pragmas for C/C++
  - Run-time library routines
  - Environment variables

OpenMP – Fork-Join Programming Model



- Master Thread (MT) executes sequentially the program
- A team of threads is being generated when MT encounters a Parallel Region (PR)
- All but the MT are being destroyed at the end of a PR

Threads

- Threads are numbered from 0 to $n - 1$, n is the number of threads
- `omp_get_num_threads` gives the number of available threads
- `omp_get_thread_num` tells the thread its number
- A single program with several threads is able to handle several tasks concurrently
- Program code, global data, heap, file descriptors etc., can be shared among the threads
- Each thread has its own stack and registers

Very Simple OpenMP Example

C/C++

```
#include <stdio.h>

int main() {
#pragma omp parallel
  {
    printf("Hello World\n");
  }
 }
```

Compile

```
icc -openmp helloworld.c -o helloworld
gcc -fopenmp helloworld.c -o helloworld
```

Execute

```
export OMP_NUM_THREADS=2
./helloworld
```

Very Simple OpenMP Example

Fortran

```
PROGRAM hello_world
!$OMP PARALLEL
WRITE(*,*) "Hello World"
!$OMP END PARALLEL
END PROGRAM hello_world
```

Compile

```
ifort -openmp helloworld.f90 -o helloworld
gfortran -fopenmp helloworld.f90 -o helloworld
```

Execute

```
export OMP_NUM_THREADS=2
./helloworld
```

Comparing C/C++ and Fortran usage

C/C++

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

int main() {
#pragma omp parallel
  {
    printf("Hello World\n");
  }
}
```

Fortran

```
PROGRAM hello_world
USE omp_lib

!$OMP PARALLEL
WRITE(*,*) "Hello World"
!$OMP END PARALLEL
END PROGRAM hello_world
```

- C and C++ use exactly the same constructs
- Slight differences between C/C++ and Fortran

Exercise 1

1. Write a program where the master thread forks a parallel region. Each thread should print its thread number. Let the program run with a different number of threads. The master thread should write out the number of used threads.

☞ Don't declare any variables but only use the functions mentioned on slide 7 where needed
☞ For Fortran: include "use omp_lib"
☞ For C/C++: include omp.h, i.e. "#include <omp.h>"
☞ Set number of threads in the shell, i.e.

```
export OMP_NUM_THREADS=n (n=1,2,...)
```

Syntax of Directives I

C/C++

(V4.0, pp. 25-27)

- Directives are special compiler pragmas
- Directive and API function names are case-sensitive and are written lower-case
- There are no END directives, but rather the directives apply to the following structured block (statement with one entry and one exit)

C/C++

```
#pragma omp parallel [ParameterList]
{
#pragma omp DirectiveName [ParameterList]
  {
    C/C++ source code
  }
} // end parallel region
```

Syntax of Directives II

C/C++

- Directive continuation lines:

C/C++

```
#pragma omp DirectiveName here-is-something \
        and-here-is-some-more
```

Syntax of Directives III

Fortran

- Directives are special-formatted comments
  - Directives ignored by non-OpenMP compiler
  - Case-insensitive
- END directive indicates end of block

Fortran

```
!$OMP PARALLEL [ParameterList]
!$OMP DirectiveName [ParameterList]
 Fortran source code
!$OMP END DirectiveName
!$OMP END PARALLEL
```

Syntax of Directives IV

Fortran

- Directive continuation lines

Fortran

```
!$OMP DirectiveName here-is-something &
!$OMP and-here-is-some-more
```

Typical OpenMP Usage Example I

C/C++

The following probably doesn't behave the way you want it to

C/C++

```
void simple(int n, float *a, float *b) {
int i;
#pragma omp parallel
 {
   for(i=1; i<n ; i++)
   b[i] =(a[i] + a[i-1])/2.0;
 }
}
```

- Here, all threads process concurrently the loop

Typical OpenMP Usage Example II

C/C++

Doing it the right way

(V4.0, pp. 53-60)

C/C++

```
void simple(int n, float *a, float *b) {
int i;
#pragma omp parallel
  {
#pragma omp for
  for(int i=1; i<n ; i++)
  b[i] =(a[i] + a[i-1])/2.0;
  }
}
```

#pragma omp for *[clause[[,] clause]...]*

for-loop

- Curly brackets are implied with the construct.

- Use the parallel loop construct.
- Each thread works on a certain range of *i*.

Typical OpenMP Usage Example III

C/C++

The same example with the *combined construct*

C/C++

```
void simple(int n, float *a, float *b) {
int i;

#pragma omp parallel for
for(int i=1; i<n ; i++)
  b[i] =(a[i] + a[i-1])/2.0;
```

#pragma omp parallel for
[clause[[,] clause]...]
for-loop

- Curly brackets are implied with the construct.

Typical OpenMP Usage Example IV

Fortran

Doing it the right way

(V4.0, pp. 53-60)

Fortran

```
SUBROUTINE SIMPLE(N, A, B)
INTEGER I, N
REAL B(N), A(N)
!$OMP PARALLEL
!$OMP DO
DO I=2,N
B(I) = (A(I) + A(I-1)) / 2.0
ENDDO
!$OMP END DO
!$OMP END PARALLEL
END SUBROUTINE SIMPLE
```

!$OMP DO *[clause[[,] clause]...]*
do-loop
[**!$OMP END DO**]

- The terminating **!$OMP END DO** construct is optional

- Use the parallel loop construct.
- Each thread works on a certain range of *i*.

Typical OpenMP Usage Example V

Fortran

The same example with the combined construct

Fortran

```
SUBROUTINE SIMPLE(N, A, B)
INTEGER I, N
REAL B(N), A(N)
!$OMP PARALLEL DO
DO I=2,N
 B(I) = (A(I) + A(I-1)) / 2.0
ENDDO
!$OMP END PARALLEL DO
END SUBROUTINE SIMPLE
```

!$OMP PARALLEL DO
[clause[[,] clause]...]
do-loop
[**!$OMP END PARALLEL DO**]

- The terminating **!$OMP END PARALLEL DO** construct is optional

Data Environment

(V4.0, pp. 155-166)

- Data objects (variables) can be shared or private
- By default almost all variables are shared
  - Accessible to all threads
  - Single instance in shared memory
- Variables can be declared private
  - Then each thread allocates its own private copy of the data
  - Only exists during the execution of a parallel region!
  - Value undefined upon entry of parallel region
- Exceptions to default shared:
  - Loop index variable in parallel loop
  - Local variables of functions called inside parallel regions
  - Variables declared in the lexical context of a parallel region

Data-Sharing Attribute Clauses I

shared (V4.0, 157)

- Only a single instance of variables in shared memory
- All threads have read and write access to these variables

```
int value = 99;

#pragma omp parallel shared(value)
{
```

Thread 0	Thread 1	Thread 2
value = omp_get_thread_num();	value = omp_get_thread_num();	value = omp_get_thread_num();

```
}
Here, value is either 0 or 1 or 2
```

Data-Sharing Attribute Clauses II

private (V4.0, 159)

- Each thread allocates its own private copy of the data element.
- These local copies only exist in parallel region
  - Undefined when entering the parallel region
  - Undefined when exiting the parallel region

```
int value = 99;

#pragma omp parallel private(value)
{
```

Thread 0	Thread 1	Thread 2
value is undefined	value is undefined	value is undefined

```
}
Here, value is 99 again
```

Data-Sharing Attribute Clauses III

firstprivate (V4.0, 162)

- Variables are also declared to be private like in the *private* clause.
- Additionally, get initialized with value of original variable.

```
int value = 99;

#pragma omp parallel firstprivate(value)
{
```

Thread 0	Thread 1	Thread 2
value is 99	value is 99	value is 99

```
}
Here, value is 99 again
```

Data-Sharing Attribute Clauses IV

lastprivate (V4.0, 164)

- Declares variables as private.
- Corresponding shared variable after parallel region gets value from that thread that finished the parallel region.

```
int value = 99;

#pragma omp parallel for lastprivate(value)
for (i=0 ; i<size ; ++i)
{
  value = i; /* Thread 0 */
}
```

```
value = i;  /* Thread 1 */
```

```
value = i;  /*  Thread 2 */
```

Here, value is size-1

Data-Sharing Attribute Clauses V

C/C++: default

- default(shared | none) (V4.0, 156)

Fortran: default

- default(private | firstprivate | shared | none) (V4.0, 156)

- It is recommended to use default(none).

Data-Sharing Attribute Clauses V

C/C++: default

- default(shared | none) (V4.0, 156)

Fortran: default

- default(private | firstprivate | shared | none) (V4.0, 156)

- It is recommended to use default(none).

☞ Play around with the data-sharing attributes to understand their meaning (15 min.)

Runtime Library Functions and Environment Variables

- OpenMP provides routines to gather information from the thread environment.
- Programmer needs possibility to adapt the environment.

C/C++ (V4.0, p. 188, p. 287)

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

Fortran (V4.0, p. 188, p. 287)

```
include "omp_lib.h"
! alternatively: use omp_lib
```

Some Useful Functions and Environment Variables I

Affect the parallelism: number of threads

- omp_set_num_threads(num-threads) (V4.0, 189)
- OMP_NUM_THREADS (V4.0, 239, 301)

How many threads are available?

- omp_get_num_threads() (V4.0, 191)

Thread's "name"

- omp_get_thread_num() (V4.0, 193)

Some Useful Functions and Environment Variables II

Time measurement

- omp_get_wtime() (V4.0, 233)
  - Returns elapsed wall clock time in seconds

```
double start_time, elapsed_time;
start_time = omp_get_wtime();
 // code block
 elapsed_time = omp_get_wtime() - start_time
```

Enable nested parallel regions

- omp_set_nested(nested) (V4.0, 200)
- OMP_NESTED (V4.0, 243)

Conditional Compilation

C/C++ Fortran

Preprocessor macro _OPENMP for C/C++ and Fortran

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

Special comment for Fortran preprocessor

```
!$ iam = OMP_GET_THREAD_NUM()
```

- Helpful check of serial and parallel version of the code

Parallel Loops Revisited

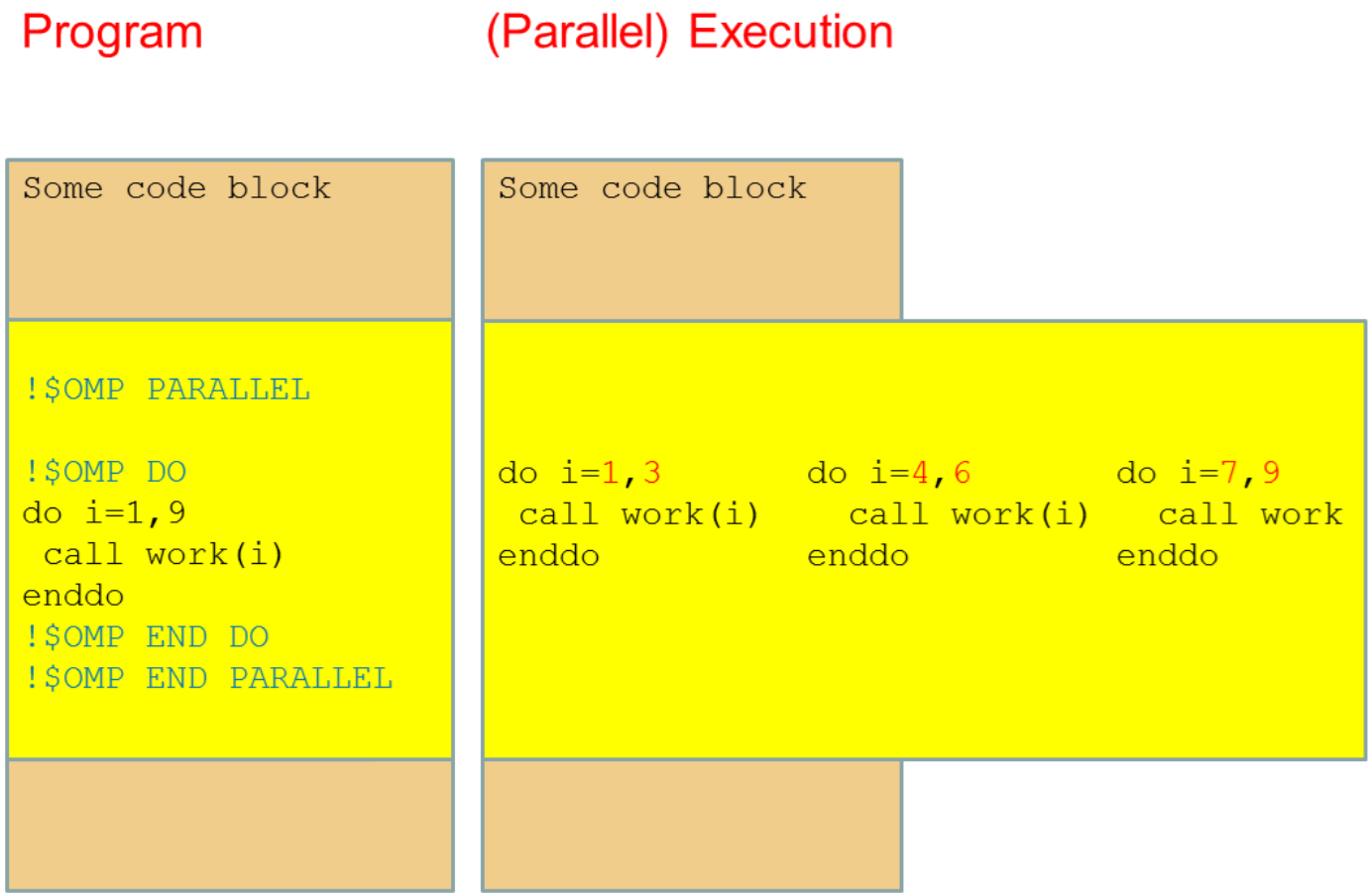
Parallel loop directive in C/C++ (V4.0, pp. 53-60)

```
#pragma omp for [ParameterList]
  for (stmt, cond, stmt) {
    block }
```

Parallel loop structure in Fortran (V4.0, pp. 53-60)

```
!$OMP DO [ParameterList]
  do ..
    block
  enddo
!$OMP END DO
```

Parallel Execution of a Loop



Canonical Loop Structure I

- Same restrictions as with Fortran DO-loop

The for loop can only have a very restricted form

```
#pragma omp for
for ( var = first ; var cmp-op end ; incr-expr) {
loop-body }
```

Comparison operators (cmp-op)

```
<, <=, >, >=
```

Canonical Loop Structure II

Increment expression (incr-expr)

```
++var, var++, --var, var--,
var += incr, var -= incr,
var = var + incr, var = incr + var
```

- first, end, incr: constant expressions

Canonical Loop Structure III

Restrictions on loops to simplify compiler-based parallelization

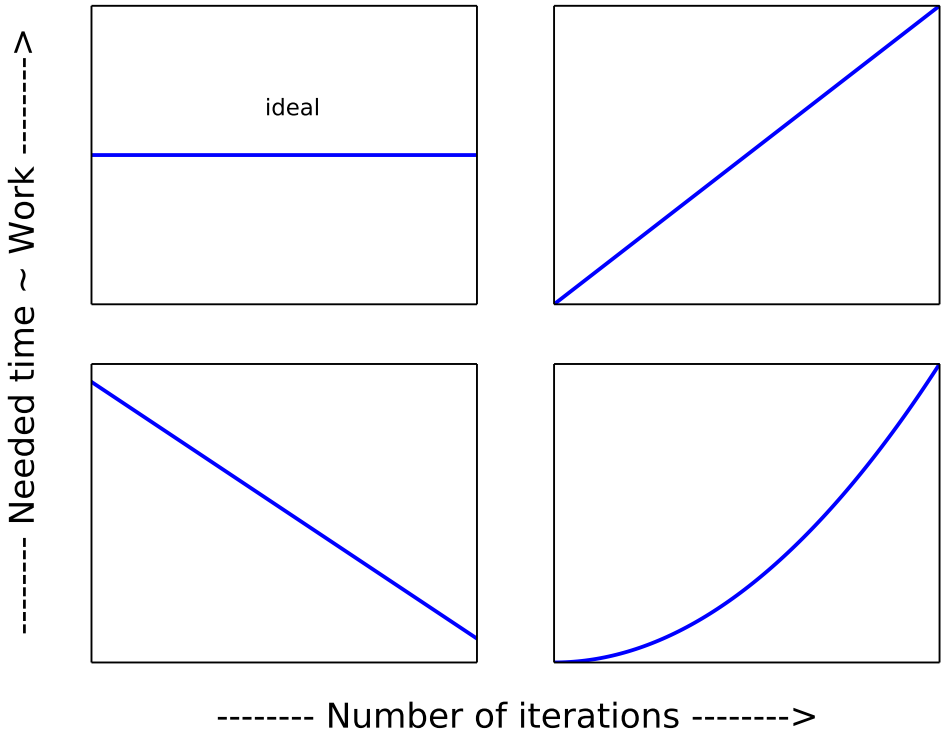
- Allows number of iterations to be computed at loop entry
- Program must complete all iterations of the loop
  - no break or goto leaving the loop — C/C++
  - no exit or goto leaving the loop — Fortran
- Exiting current iteration and starting next one possible
  - continue allowed — C/C++
  - cycle allowed — Fortran
- Termination of the entire program inside the loop possible
  - exit allowed — C/C++
  - stop allowed — Fortran

Parallel Loop: Further Prerequisites

- Iterations of a parallel loop are executed in parallel by all threads of the current team
- The calculations inside an iteration must not depend on other iterations ➔ responsibility of the programmer
- A schedule determines how iterations are divided among the threads
  - Specified by “schedule” parameter
  - Default: undefined!
- The form of the loop has to allow computing the number of iterations prior to entry into the loop ➔ e.g., no WHILE loops
- Implicit barrier synchronization at the end of the loop

Loops & Performance Optimization

- The loop should be processed as fast as possible, this means the work should be well balanced among the threads.



Scheduling Strategies

(V4.0, pp. 55-60)

- Distribution of iterations occurs in chunks
- Chunks may have different sizes
- There are different assignment algorithms (types)

SCHEDULE parameter C/C++/Fortran

```
#pragma omp parallel for schedule(type [,chunk])
!$omp parallel do schedule(type [,chunk])
```

- Schedule types
  - static
  - dynamic
  - guided
  - runtime

Static Scheduling I

(V4.0, pp. 55-60)

Static scheduling

- Distribution is done at loop-entry based on
  - Number of threads
  - Total number of iterations
- Less flexible
- Almost no scheduling overhead

Static Scheduling II

static with chunk size

- Chunks with specified size are assigned in round-robin fashion

schedule(static, 2)

t0 t1 t2 t3 .....

Represents an iteration

Static Scheduling III

static without chunk size

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

t0 t1 t2 t3

t0 t1 t2 t3

Represents an iteration

Dynamic Scheduling I

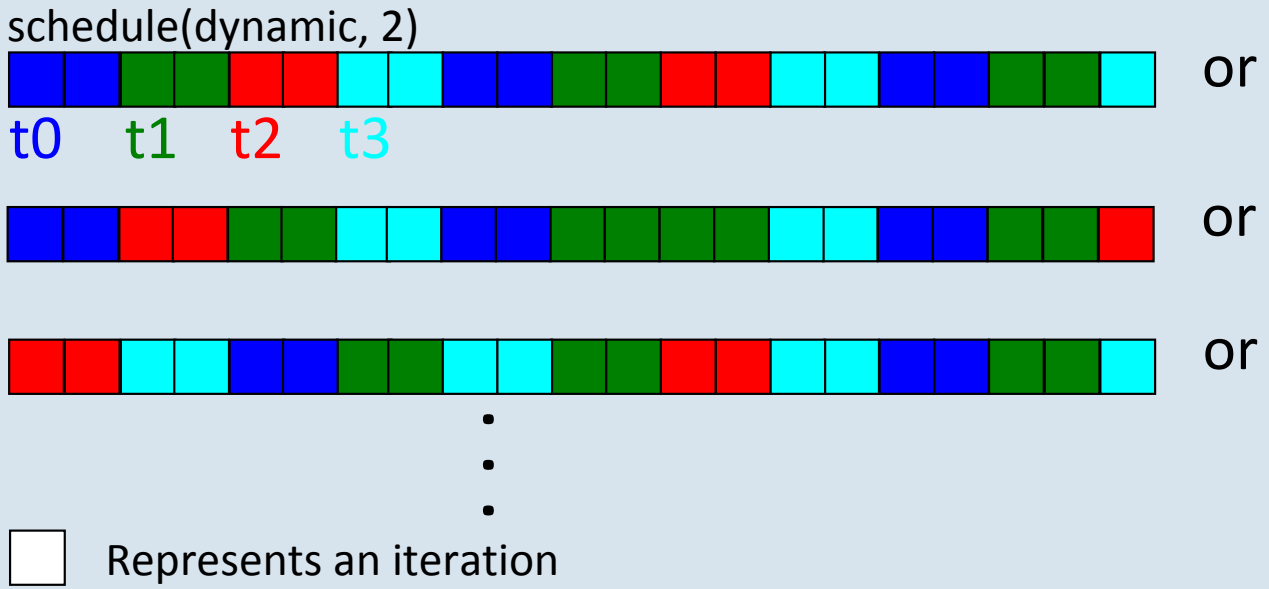
Dynamic scheduling

- Distribution is done during execution of the loop
  - Each thread is assigned a subset of the iterations at loop entry
  - After completion each thread asks for more iterations
- More flexible
  - Can easily adjust to load imbalances
- More scheduling overhead (synchronization)

Dynamic Scheduling II

dynamic

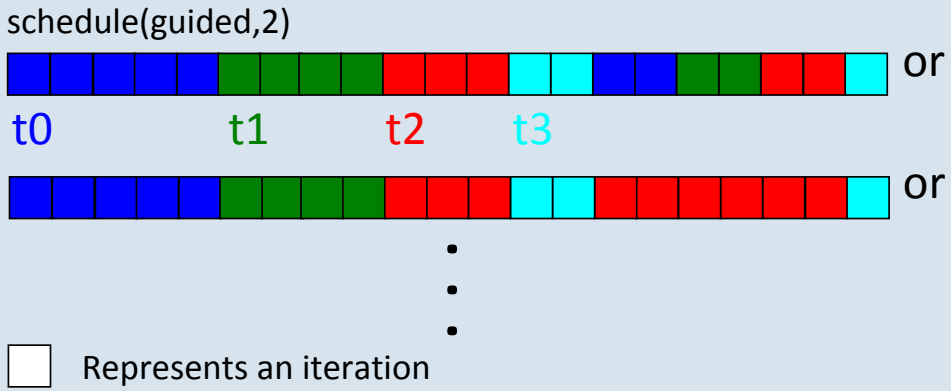
- Threads request new chunks dynamically during runtime
- Default chunk size is 1



Guided Scheduling

guided

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



Set Scheduling on Demand

schedule(runtime)

- Scheduling strategy can be chosen by environment variable
- If variable is not set, scheduling implementation dependent

```
export OMP_SCHEDULE="type [, chunk]"
```

- If no schedule parameter is given then scheduling is implementation dependent
- Correctness of program must not depend on scheduling strategy
- *omp_set_schedule* / *omp_get_schedule* allow to set scheduling via runtime environment

Exercise 2

1. Write a program that implements DAXPY (Double precision real Alpha X Plus Y): $\vec{y} = \alpha\vec{x} + \vec{y}$
   Vector sizes and values can be chosen as you like.
2. Measure the time it takes in dependence of the number of threads.

If Clause: Conditional Parallelization I

C/C++

- Avoiding parallel overhead because of low number of loop iterations

Explicit version

```
if (size > 10000) {
#pragma omp parallel for
  for(i=0 ; i<size ; ++i) {
    a[i] = scalar * b[i];
  }
} else {
  for(i=0 ; i<size ; ++i) {
    a[i] = scalar * b[i];
  }
}
```

If Clause: Conditional Parallelization II

Using the „if clause“

C/C++

```
#pragma omp parallel for if (size > 10000)
for (i=0 ; i<size ; ++i) {
  a[i] = scalar * b[i];
}
```

Fortran

```
!$OMP PARALLEL DO IF (size .gt. 10000)
do i = 1, size
  a(i) = scalar * b(i)
enddo
!$OMP END PARALLEL DO
```

Switch Off Synchronization: nowait I

Loops & nowait clause

- Use "nowait" parameter for parallel loops which do not need to be synchronized upon exit of the loop
  - Keeps synchronization overhead low
- Hint: barrier at the end of parallel region cannot be suppressed.
- In Fortran, "nowait" needs to be given in the end part
- Check for data dependencies before "nowait" is being used

Switch Off Synchronization: nowait II

Example

C/C++

```
#pragma omp parallel
{
#pragma omp for nowait
for (i=0 ; i<size ; ++i) {
  a[i] = b[i] + c[i]; }

#pragma omp for nowait
for (i=0 ; i<size ; ++i) {
  z[i] = SQRT(b[i]); }
}
```

Fortran

```
!$OMP PARALLEL
!$OMP DO
DO i = 1, size
  a(i) = b(i) + c(i)
ENDDO
!$OMP END DO NOWAIT
!$OMP DO
DO i = 1, size
  z(i) = SQRT(b(i))
ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

- "nowait" can be used

Switch Off Synchronization: nowait III

No barrier in spite of data dependency

```
#pragma omp parallel
{
#pragma omp for schedule(static) nowait
  for (i=0 ; i<N ; ++i) {
a[i] =  ...; }

#pragma omp for schedule(static)
for (i=0 ; i<N ; ++i) {
c[i] = a[i] + ...; }
}
```

- OpenMP 3.0 guarantees for "static" schedule same number of iterations
- Second loop is divided in the same chunks
- Not guaranteed in OpenMP 2.5 or previous versions

JÜLICH
FORSCHUNGSZENTRUM

Shared-Memory Parallel Programming with OpenMP

Part II: Further Frequently used Features

July 5.-6. 2016 | Alexander Schnurpfeil, presented by Jochen Kreutz

Parallel Sections I

C/C++ (V4.0, 60)

#pragma omp sections *[clause[[,] clause] ...]*

{

[**#pragma omp section**]

structured block

...

}

Fortran (V4.0, 60)

!$omp sections *[clause[[,] clause] ...]*

[**!$omp section**]

structured block

...

!$omp end sections

- A parallel section contains blocks of statements which can be executed in parallel
- Each block is executed once by one thread of the current team

Parallel Sections II

- Scheduling of the block executions is implementation defined and cannot be controlled by the programmer
- Sections must not depend on each other
- Most frequent use case: parallel function calls

Supported clauses

- private, firstprivate, lastprivate
- reduction
- nowait

Parallel Sections III

Example

C/C++

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

#define N 1000000

int main() {
int i, a[N], b[N];
  #pragma omp parallel sections private(i)
  {
  #pragma omp section
  {
    for (i=0 ; i<N ; ++i) a[i] = 100;
  }
  #pragma omp section
  {
    for (i=0 ; i<N ; ++i) b[i] = 200;
  }
  }
}
```

Parallel Sections IV

Example

Fortran

```
PROGRAM sections

PARAMETER(N=1000)
INTEGER i, a(N), b(N)

!$OMP PARALLEL SECTIONS PRIVATE(i)
!$OMP SECTION
  DO i= 1, N
    a(i) = 100
  ENDDO
!$OMP SECTION
  DO i= 1, N
    b(i) = 200
  ENDDO
!$OMP END PARALLEL SECTIONS

END PROGRAM sections
```

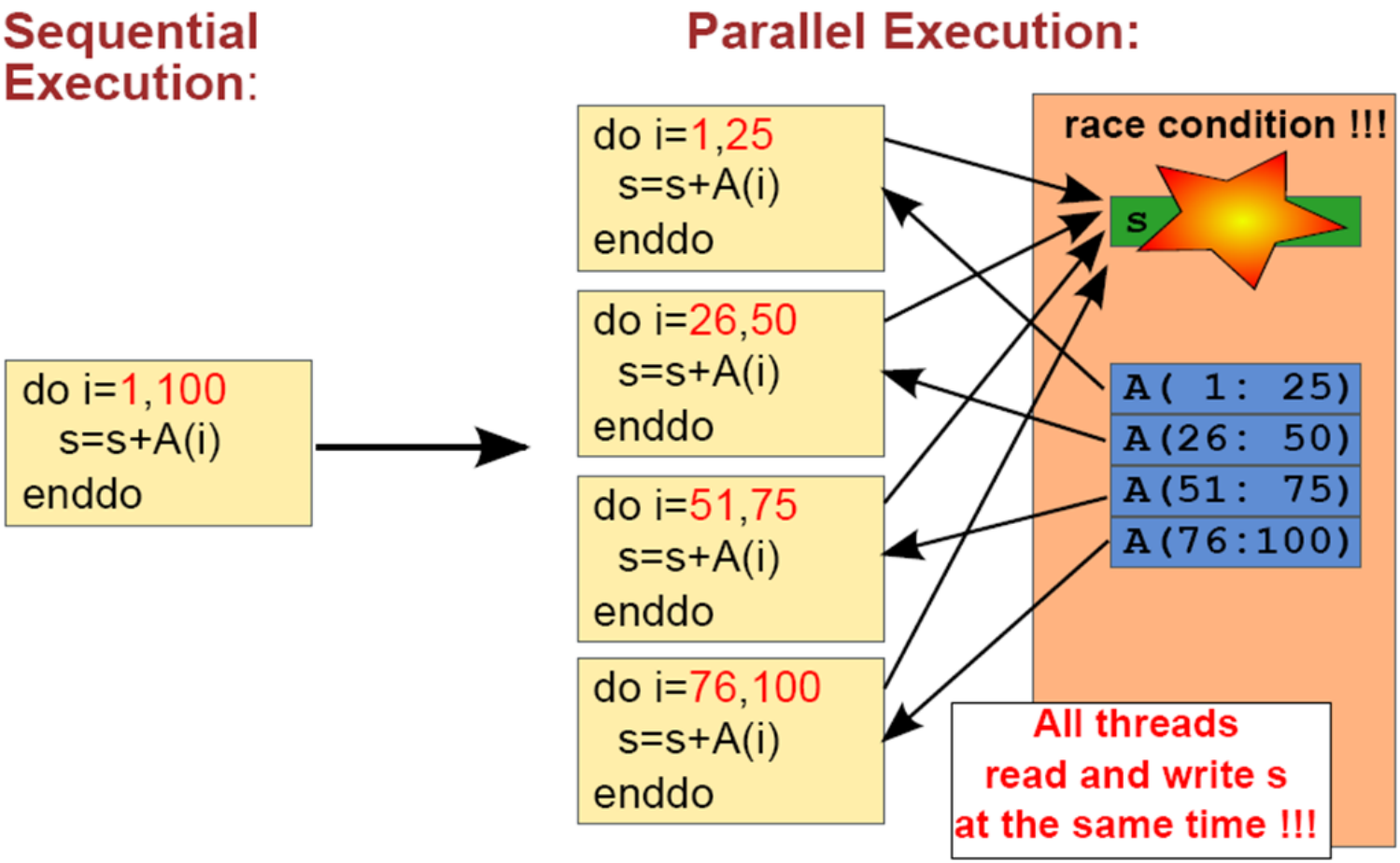
What now?

Intermezzo: race condition

```
do i=1,100
  s=s+A(i)
enddo
```

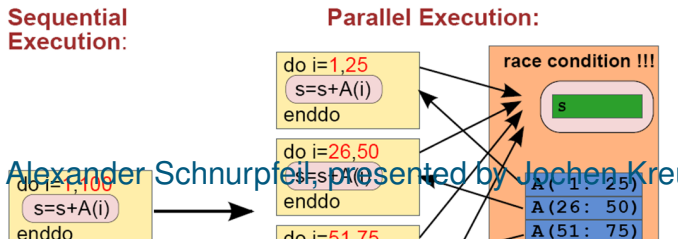
What now?

Intermezzo: Race Condition



Critical Region I

Possible solution



Critical Region I

Possible solution

C/C++ (V4.0, 122)

#pragma omp critical *[(name)]*

structured block

Fortran (V4.0, 122)

!$omp critical *[(name)]*

structured block

!$omp end critical *[(name)]*

- A critical region restricts execution of the associated block of statements to a single thread at a time
- An optional name may be used to identify the critical region
- A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name

Critical Region II

Possible Solution

C/C++

```
#include <omp.h>

int main() {

double s = 0.0;
double a[100];
int i;
s = 0.0;
#pragma omp parallel for
  for (i=0 ; i<100 ; ++i) {
  #pragma omp critical
  {
    s += a[i];
  }
  }
}
```

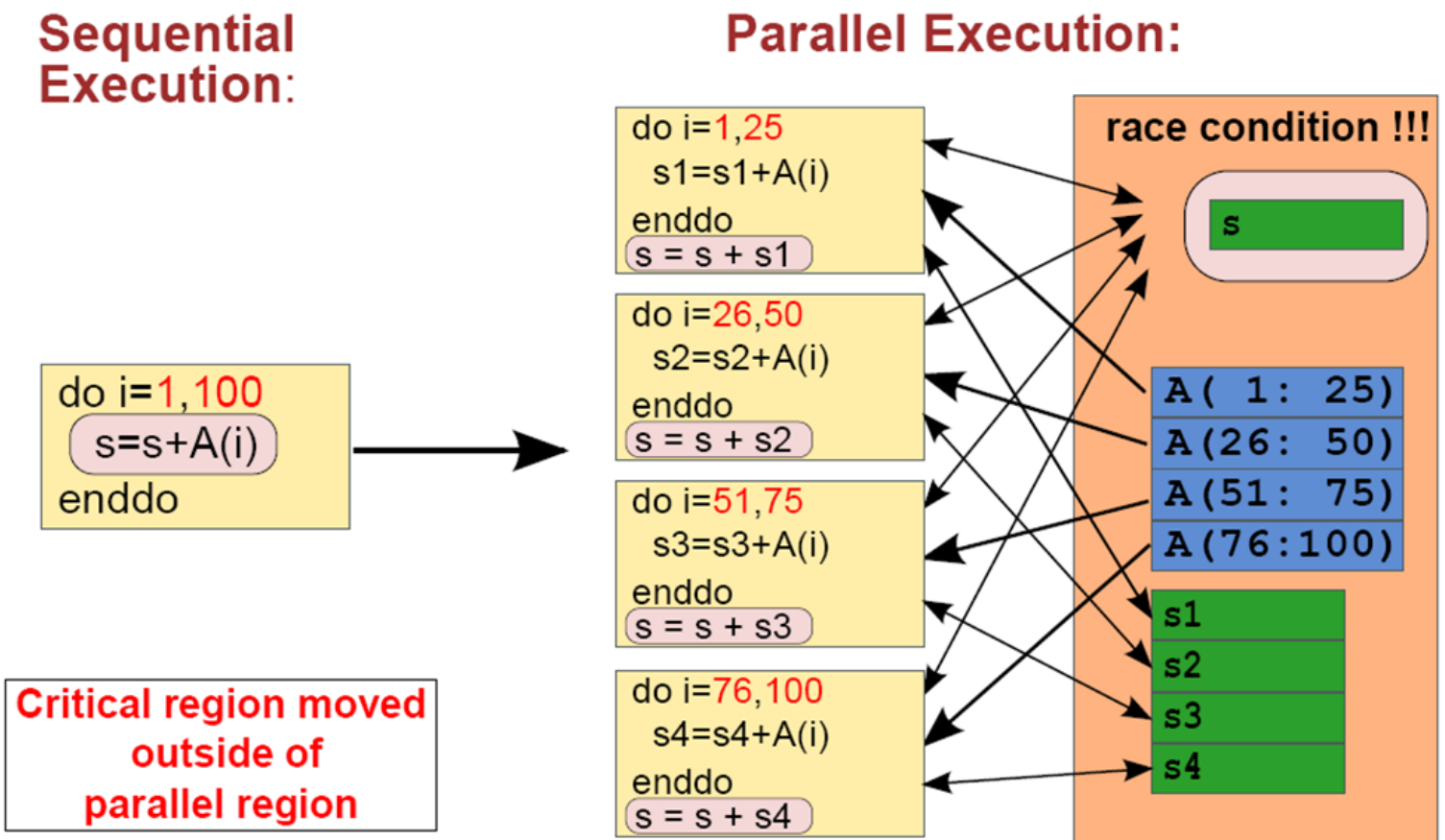
Fortran

```
PROGRAM critical

REAL :: s = 0.0
REAL, DIMENSION(0:100) :: a

!$OMP PARALLEL DO private(i)
          DO i = 1, 100
          !$OMP CRITICAL
            s = s + a(i)
          !$OMP END CRITICAL
          ENDDO
!$OMP END PARALLEL DO

END PROGRAM critical
```

- Note: the loop body only consists of a critical region
- Program gets extremely slow ➔ no speedup

Critical Region III

Better solution



Critical Region IV

Better Solution

C/C++

```
#pragma omp parallel private(i, s_local)
{
  s_local = 0.0;
  #pragma omp for
  for (i=0 ; i<100 ; ++i)
  {
    s_local += a[i];
  }
  #pragma omp critical
  {
    s += s_local;
  }
}
```

Fortran

```
!$OMP PARALLEL PRIVATE(i, s_local)
s_local = 0.0;
  !$OMP DO
    DO i = 1, 100
      s_local = s_local + a(i)
    ENDDO
  !$OMP END DO
  !$OMP CRITICAL
    s = s + s_local
  !$OMP END CRITICAL
!$OMP END PARALLEL
```

Atomic Statement I

Maybe an even better solution

C/C++ (V4.0, 127)
#pragma omp atomic *statement*

Fortran (V4.0, 127)
!$omp atomic *statement*

- The ATOMIC directives ensures that a specific memory location is updated atomically.
  - No thread interference
- OpenMP implementation could replace it with a CRITICAL construct. ATOMIC construct permits better optimization (based on hardware instructions)

Atomic Statement II

Maybe an even better solution

Selection of allowed statements in C/C++

- x binop= expr
  - binop = +,*,-,/,&,^,|,«,»
  - *expr* is a scalar type
  - x++, ++x, x--, --x

Selection of allowed statements in Fortran

- x = x op expr
- x = expr op x
  - op = +,*,-,/,.AND.,.OR.,.EQV.,.NEQV.
  - *expr* is a scalar expression
- x = intrinsic(x, expr)
- x = intrinsic(expr, x)

- OpenMP V4.0 extends "atomic" statement; not covered in this course

Atomic Statement III

Maybe an even better solution

C/C++

```
int main() {

double s = 0.0, s_local = 0.0;
double a[100];
int i;

#pragma omp parallel private(i, s_local)
{
  s_local = 0.0;
  #pragma omp for
  for (i=0 ; i<100 ; ++i)
  {
    s_local += a[i];
  }
  #pragma omp atomic
    s += s_local;
}
}
```

Fortran

```
PROGRAM atomic

REAL :: s = 0.0
REAL :: s_local = 0.0
REAL, DIMENSION(0:100) :: a

!$OMP PARALLEL PRIVATE(i, s_local)
  s_local = 0.0;
  !$OMP DO
    DO i = 1, 100
      s_local = s_local + a(i)
    ENDDO
  !$OMP END DO
  !$OMP ATOMIC
    s = s + s_local
!$OMP END PARALLEL

END PROGRAM atomic
```

Reduction Statement I

Maybe the best solution to the problem

- For solving cases like this, OpenMP provides the "reduction" parameter

C/C++ (V4.0, 167)

```
int main() {

double s = 0.0;
double a[100];
int i;

#pragma omp parallel for private(i) \
  reduction(+:s)
for (i=0 ; i<100 ; ++i)
{
  s += a[i];
}
}
```

Fortran (V4.0, 167)

```
PROGRAM reduction

REAL :: s = 0.0
REAL, DIMENSION(0:100) :: a

!$OMP PARALLEL DO PRIVATE(i) REDUCTION(+:s)
  DO i = 1, 100
    s = s + a(i)
  ENDDO
!$OMP END PARALLEL DO

END PROGRAM reduction
```

Reduction Statement II

Syntax (V4.0, 167)

reduction(*operator* | *intrinsic* **:** *varlist* **)**

- Reductions often occur within parallel regions or loops
- Reduction variables have to be shared in enclosing parallel context
- Thread-local results get combined with outside variables using reduction operation
- Note: order of operations unspecified ➔ can produce slightly different results than sequential version (rounding error)
- Typical applications: Compute sum of array or find the largest element

Reduction Statement III

Operators for C/C++

Operator	Data Type	Initial Value
+	Floating point, Integer	0
*	Floating point, Integer	1
-	Floating point, Integer	0
&	Integer	all bits on
|	Integer	0
^	Integer	0
&&	Integer	1
||	Integer	0

- The table shows:
  - The operators and intrinsic allowed for reductions
  - The initial values used to initialize
- Since OpenMP 3.1: *min, max*

Reduction Statement IV

Operators for Fortran

Operator	Data Types	Initial Value
+	floating point, integer (complex or real)	0
*	floating point, integer (complex or real)	1
-	floating point, integer (complex or real)	0
.AND.	logical	.TRUE.
.OR.	logical	.FALSE.
.EQV.	logical	.TRUE.
.NEQV.	logical	.FALSE.
MAX	floating point, integer (real only)	smallest possible value
MIN	floating point, integer (real only)	largest possible value
IAND	integer	all bits on
IOR	integer	0
IEOR	integer	0

User Defined Reductions I

C/C++ Fortran

Declare reduction in C/C++ (V4.0, pp. 180-185)

```
#pragma omp declare reduction \
 (reduction-identifier:type-list:combiner) \
 [initializer-clause] newline
```

Declare Reduction in Fortran (V4.0, pp. 180-185)

```
!$OMP DECLARE REDUCTION &
!$OMP (reduction-identifier:type-list:combiner) &
!$OMP [initializer-clause] newline
```

User Defined Reductions II

- **reduction-identifier**: name to be later on in the reduction clause
- **type-list**: can be int, double, use-defined, ...
- **combiner**: specifies how partial results can be combined into single value (uses special identifiers *omp_in* and *omp_out*).
- **initializer**: specifies how to initialize private copies of the reduction variable (uses special identifier *omp_priv*).

- *omp_in*: refers to the storage to be combined
- *omp_out*: refers to the storage that holds the combined value
- *omp_priv*: refers to the storage to be initialized

User Defined Reductions II

- **reduction-identifier**: name to be later on in the reduction clause
- **type-list**: can be int, double, use-defined, ...
- **combiner**: specifies how partial results can be combined into single value (uses special identifiers *omp_in* and *omp_out*).
- **initializer**: specifies how to initialize private copies of the reduction variable (uses special identifier *omp_priv*).

- *omp_in*: refers to the storage to be combined
- *omp_out*: refers to the storage that holds the combined value
- *omp_priv*: refers to the storage to be initialized

☞ Example: example_declare_reduction.c

Exercise 3

1. Have a look at declare-reduction-c.c (declare-reduction-f90.f90), compile and let it run. What's going wrong?
2. Use the reduction clause to repair the program.
3. Declare your own reduction to repair the program.

Exercise 4

1. π can be calculated in the following way:

$$\pi = 4.0\int_0^1 \frac{1.0}{1.0 + x\text{^}2} dx \approx \sum_{i=1}^{n} \frac{4.0}{(1.0 + x\text{^}2)} \Delta x$$

$$\Delta x = \frac{1}{n}$$

$$x = (i - 0.5)\Delta x$$

➔ Write a program that does the calculation and parallelize it by using OpenMP. Consider reduction clauses.

Master / Single I

More on Synchronization

- Sometimes it is useful that within a parallel region just one thread is executing code, e.g., to read/write data. OpenMP provides two ways to accomplish this:
  - The MASTER construct: The master thread (thread 0) executes the enclosed code, all other threads ignore this block of statements, i.e. there is no implicit barrier

C/C++ (V4.0, 120)

```
#pragma omp master
        structured block
```

Fortran (V4.0, 120)

```
!$OMP MASTER
        structured block
!$OMP END MASTER
```

Master / Single II

- Single construct: The first thread reaching the directive executes the code. All threads execute an implicit barrier synchronization unless "NOWAIT" is specified

C/C++ (V4.0, 63)

#pragma omp single *[nowait, private(list),...]*
structured block

Fortran (V4.0, 63)

!$OMP SINGLE *[private(list),...]*
structured block
!$OMP END SINGLE *[NOWAIT]*

Barrier

C/C++ (V4.0, 123)	Fortran (V4.0, 123)
#pragma omp barrier	**!$OMP BARRIER**

- The barrier directive explicitly synchronizes all the threads in a team.
- When encountered, each thread in the team waits until all the others have reached this point
- There are also implicit barriers at the end
  - of parallel region, cannot be changed
  - of work share constructs (DO/for, SECTIONS, SINGLE) ➔ these can be disabled by specifying NOWAIT

Locks

C/C++

- More flexible way of implementing “critical regions”
- Lock variable type: `omp_lock_t`, passed by address

Initialize lock

```
omp_init_lock(&lockvar)
```

Remove (deallocate) lock

```
omp_destroy_lock(&lockvar)
```

Blocks calling thread until lock is available

```
omp_set_lock(&lockvar)
```

Release lock

```
omp_unset_lock(&lockvar)
```

Test and try to set lock (returns 1 if success else 0

```
intvar = omp_test_lock(&lockvar)
```

Locks

Fortran

- More flexible way of implementing “critical regions”
- Lock variable has to be of type integer

Initialize lock

```
CALL OMP_INIT_LOCK(lockvar)
```

Remove (deallocate) lock

```
CALL OMP_DESTROY_LOCK(lockvar)
```

Blocks calling thread until lock is available

```
CALL OMP_SET_LOCK(lockvar)
```

Release lock

```
CALL OMP_OMP_UNSET_LOCK(lockvar)
```

Test and try to set lock (returns 1 if success else 0

```
logicalvar = OMP_TEST_LOCK(lockvar)
```

Locks II

C/C++ Example (API Examples V4.0.0, p. 153 for Fortran example)

Lock mechanism

```
int main() {
  omp_lock_t lck;
  int id;
  omp_init_lock(&lck);
#pragma omp parallel shared(lck) private(id)
{
  id = omp_get_thread_num();

  while (!omp_test_lock(&lck)) {
    skip(id);
        /* we do not yet have the lock,
           so we must do something else */
  }
  work(id);
    /* we now have the lock
       and can do the work */
  printf("Key given back by %i\n",id);
  omp_unset_lock(&lck);
}
  omp_destroy_lock(&lck);
  return 0;
}
```

Locks II

C/C++ Example (API Examples V4.0.0, p. 153 for Fortran example)

Lock mechanism

```
int main() {
  omp_lock_t lck;
  int id;
  omp_init_lock(&lck);
#pragma omp parallel shared(lck) private(id)
{
  id = omp_get_thread_num();

  while (!omp_test_lock(&lck)) {
    skip(id);
        /* we do not yet have the lock,
           so we must do something else */
  }
  work(id);
    /* we now have the lock
       and can do the work */
  printf("Key given back by %i\n",id);
  omp_unset_lock(&lck);
}
  omp_destroy_lock(&lck);
  return 0;
}
```

☞ example_lock.c

Shared-Memory Parallel Programming with OpenMP

Part III: Miscellaneous

July 5.-6. 2016 | Alexander Schnurpfeil, presented by Jochen Kreutz

Nested Loops I

Doesn't work

```
#pragma omp parallel
{
  #pragma omp for
    for (i=0 ; i<n ; ++i)
    {
      #pragma omp for /* illegal  */
        for (j=0 ; i<m ; ++j){
          /* whatever */
        }
    }
}
```

Working version

```
#pragma omp parallel for
  for (x=0; x<n*m ; ++x) {
    i = x/m;
    j = x%m;

    /* what ever */
  }
```

- Parallel loop construct only applies to loop directly following it
- Nesting of work-sharing constructs is illegal in OpenMP!

Nested Loops II

- Parallelization of nested loops
  - Normally try to parallelize outer loop ➔ less overhead
  - Sometimes necessary to parallelize inner loops (e.g., small n) ➔ re-arrange loops?
  - Manually ➔ re-write into one loop
  - Nested parallel regions not yet supported by all compilers (current version of GNU compiler actually does)

Nested Loops III

"collapse" clause is available since OpenMP V3.0

(V4.0, 55)

Loop collapsing example in C/C++

```
void foo(int a, int b, int c) { /* do something */ }

int main() {

int N = 100;
int i,j,k;
#pragma omp parallel for collapse(2)
  for (i=0 ; i<N ; ++i) {
    for (j=0 ; j<N ; ++j) {
      for (k=0 ; k<N ; ++k) {
        foo(i,j,k);
          }
    }
  }
}
```

- Rectangular iteration space from the outer two loops
- Outer loops are collapsed into one larger loop with more iterations

Orphaned Work-sharing Construct

Orphaned construct example in C/C++

```
void do_something(int v[], int n) {
int i;
#pragma omp for
  for (i=0;i<n;++i) {
    v[i] = 0; }
}

int main() {
int size = 10;
int v[size];
#pragma omp parallel
  {
    do_something(v,size); /*Case 1*/
  }
    do_something(v,size); /*Case 2*/
    return 0;
}
```

- Work-sharing construct can occur anywhere outside the lexical extent of a parallel region ➔ orphaned construct
- Case 1: called in a parallel context ➔ works as expected
- Case 2: called in a sequential context ➔ “ignores” directive

Task Parallelism

What is a task?

A Task is

- a unit of independent work (block of code, like sections)
- a direct or deferred execution of work performed by one thread of the team
- is composed of
  - code to be executed, and
  - the data environment (constructed at creation time)
- tied to a thread: only this thread can execute the task
- useful for unbounded loops, recursive algorithms, work on linked lists (pointer), consumer/producer processes

Task Directive

(V4.0, pp. 113-116)

C/C++

#pragma omp task *[clause[[,] clause] ...]*
structured block

Fortran

!$OMP TASK *[clause[[,] clause] ...]*
structured block
!$OMP END TASK

- Allowed data scope:
  `default, private, firstprivate, shared`
- Each encountering thread creates a new task
- Tasks can be nested, into another task or a worksharing construct

Task Directive

(V4.0, pp. 113-116)

C/C++

#pragma omp task *[clause[[,] clause] ...]*
structured block

Fortran

!$OMP TASK *[clause[[,] clause] ...]*
structured block
!$OMP END TASK

- Allowed data scope:
  `default, private, firstprivate, shared`
- Each encountering thread creates a new task
- Tasks can be nested, into another task or a worksharing construct

☞ example_task_fib_serial.c, example_task_fib_parallel.c

Thoughts on Parallelizing Codes with OpenMP I I

- Is the serial version of the code well optimized?
- Which compiler settings might increase the performance of the code?
- Estimate scalability with the help of Amdahl‘s law
- Which parts of the code consume the most computation time?
- Is the amount of parallel regions as small as possible?
- Was the most outer part of nested loops parallelized?
- Use the „nowait“ clause whenever possible
- Is the workload well balanced over all threads?
- Avoid false sharing effects
- Name all critical sections
- Consider the environment in which the program runs

JÜLICH
FORSCHUNGSZENTRUM

Shared-Memory Parallel Programming with OpenMP

Part IV: Debugging / Analysis

July 5.-6. 2016 | Alexander Schnurpfeil, presented by Jochen Kreutz

Race Conditions and Data Dependencies I

Most important rule

Parallelization of code must not affect the correctness of a program!

- In loops: the results of each single iteration must not depend on each other
- Race conditions must be avoided
- Result must not depend on the order of threads
- Correctness of the program must not depend on number of threads
- Correctness must not depend on the work schedule

Race Conditions and Data Dependencies II

- Threads read and write to the same object at the same time
  - Unpredictable results (sometimes it works, sometimes not)
  - Wrong answers without a warning signal!
- Correctness depends on order of read/write accesses
- Hard to debug because the debugger often runs the program in a serialized, deterministic ordering.
- To insure that readers do not get ahead of writers, thread synchronization is needed.
  - Distributed memory systems: messages are often used to synchronize, with readers blocking until the message arrives.
  - Shared memory systems need: barriers, critical regions, locks, ...
- Note: be careful with synchronization
  - Degrades performance
  - Deadlocks: threads waiting for a locked resource that never will become available

Intel Inspector XE 2013 Memory & Thread Analyzer

- Memory error and thread checker tool
- Supported languages on linux systems: C/C++, Fortran
- Maps errors to the source code line and call stack
- Detects problems that are not recognized by the compiler (e.g. race conditions, data dependencies)

☞ Other useful tools: gprof, threadspotter, valgrind, insure, vtune

Usage

On the command line:

```
module load inspector (on JUDGE)
module load Inspector (on JUROPA3 & JURECA)
inspxe-gui &
```

Exercise 5

- Figure out which loops have data dependencies

I

```
for (int i=1 ; i<size ; ++i) {
v[i] = v[i] + v2[i];
}
```

II

```
for (int i=1 ; i<size ; ++i) {
v[i] = v[i] + v[i-1] * 2;
}
```

III

```
for (int i=1 ; i<size ; ++i) {
v[i] = v[i] + v2[i-1];
}
```

IV

```
for (int i=1 ; i<size ; i+=2) {
v[i] = v[i] + v[i-1] * 2;
}
```

V

```
for (int i=1 ; i<size/2 ; ++i) {
v[i] = v[i] + v[i+size/2] * 2;
}
```

VI

```
for (int i=1 ; i<(size/2+1) ; ++i) {
v[i] = v[i] + v[i+size/2-1] * 2;
}
```

Exercise 6

1. Resolve data dependency in *data-dependency-01.c* and parallelize with OpenMP
2. Resolve data dependency in *data-dependency-02.c* and parallelize with OpenMP
3. Resolve data dependency in *data-dependency-03.c* and parallelize with OpenMP