Parallel Programming in OpenMP

Introduction

Dieter an Mey

Center for Computing and Communication
Aachen University of Technology

anmey@rz.rwth-aachen.de
OpenMP is a parallel programming model for shared memory multiprocessors.
Multithreading versus Multi-Processing

- Multiple Processes (Heavyweight Process model)
  - traditional UNIX process model
  - interprocess communication techniques supported by OS: shared memory, sockets, file IO, memory map
  - Higher overhead associated with process creation and destruction
- Multiple Threads (Lighweight Process model, LWP)
  - thread concept: independent flow of control within one process with its own context: stack, register set
  - process data and opened files are shared
  - lower overhead of thread creation and destruction
  - shared address space
  - Auto-Parallelization, OpenMP, Explicit Multithreading using P-Threads
- Hybrid Models (e.g. MPI + OpenMP)
OpenMP - History

1997: OpenMP Version 1.0 for Fortran
- de facto standard for shared memory programming
- now available for all SMP systems
- replaces proprietary parallelization directives and in many cases the explicit programming of threads

1998: OpenMP V1.0 for C and C++

1999: OpenMP V1.1 for Fortran (error corrections, explanations)

2000: OpenMP V2.0 for Fortran (support of Fortran90 modules)

2001: OpenMP V2.0 for C and C++ draft

http://www.OpenMP.org
http://www.cOMPunity.org
OpenMP - Information

• The OpenMP Architecture Review Board (ARB)
  Fortran and C Application Program Interfaces (APIs)
  www.openmp.org

• The Community of OpenMP Users, Researchers, Tool Developers and Providers
  www.compunity.org

• OpenMP-Courses in the Web
  • Tutorial by the OPENMP ARB at SC1998
    http://www.openmp.org/presentations/index.cgi?sc98_tutorial
  • University of Minnesota
    http://www.msi.umn.edu/tutorials/shared_tutorials/openMP/
  • Boston University
    http://scv.bu.edu/SCV/Tutorials/OpenMP/

• Book: Rohit Chandra, et.al. „Parallel Programming in OpenMP“
  Morgan Kaufmann, ISBN 1-55860-671-8
OpenMP-Compilers on Sun Machines

- SUN Forte Developer
  - f77 / f90 / f95 –openmp ... (since version 6)
  - cc –xopenmp ... (since version 6U2)
  - CC –xopenmp ... (since version 7U0)

- KAP Pro/Toolset, compiler and tools (KAI/Intel)
  - guidef77 / guidef90 / guidec / guidec++
    (preprocessors, evoking native compilers)

- Includes the unique verification tools
  assuref77 / assuref90 / assurec / assurec++
Sun versus KAP Pro/Toolset Compiler (1)

- f90 / f95 and guidef90: OpenMP V2.0
- cc / CC / f90 / f95: automatically turn on –xO3 => debugging is impossible
- f90 / f95 / cc: combination auf OpenMP and auto parallelization is supported
- CC: no support for C++ - specific features

- guide*: any optimization level of the underlying native compiler => debugging is possible
- guide*: support by the TotalView parallel debugger
- guidef90: no internal subroutines in parallel regions
- guidedc++ includes the famous KCC C++ compiler and evokes the native C compiler

- different performance characteristics, different defaults
The following list details the known limitations of the OpenMP functionality in the C++ compiler:

• No support for C++ - specific features using class objects within OpenMP regions or using OpenMP pragmas within member functions can result in errors or incorrect results. Throwing exceptions within OpenMP regions may result in undefined behavior.
• No support for nested parallelism
• No checks for loop index modification
• The compiler does not confirm that OpenMP for loop indices are not modified within the body of the loop.
• No checks for overloaded operators used in reduction clause
• Error message text still in review
OpenMP Concepts

- Parallel Regions (fork-join)
- Worksharing
- Variable Scoping (private versus shared data)
- Critical Regions
- Synchronization

- Not covered in this tutorial
  - Nested parallelism
  - Lock functions
The Components of OpenMP (Fortran)
Environment Variables, Directives, Runtime Library

### Environment Variables

```
#!/bin/ksh
# Shell-Script
f90 -openmp test.f90
export OMP_NUM_THREADS=4
a.out
```

### Directives (special comment lines)

```
! Source file test.f90
program main
integer omp_get_thread_num

 !$omp parallel
   print *, 'me: ', omp_get_thread_num()
 !$omp end parallel

end program
```

### Runtime Library

```
me:  0
me:  3
me:  2
me:  1
```
The Components of OpenMP (C)
Environment Variables, Directives, Runtime Library

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

int main(void)
{
    #pragma omp parallel
    {
        printf("me: %d\n", omp_get_thread_num());
    }
}
```

```
#!/bin/csh
# Shell-Script
cc –xopenmp test.f90
setenv OMP_NUM_THREADS 4
a.out
```

```
/* Source file test.c */
#include <stdio.h>
#include <omp.h>

int main(void)
{
    #pragma omp parallel
    {
        printf("me: %d\n",omp_get_thread_num());
    }
}
```
OpenMP Components Diagram

- user
  - shell
  - environment variables
- OpenMP program
  - directives
  - runtime functions
- OpenMP runtime system
  - operating system - threads
# Directive Formats

**Fortran77:**

```plaintext
*** OpenMP directive
C$OMP directive [clause[,, clause ] ...]
*** OpenMP directive
*$OMP directive [clause[,, clause ] ...]
*** OpenMP directive with continuation line
C$OMP directive clause clause
C$OMP+clause ...
```

**Fortran90:**

```plaintext
*** OpenMP directive
!$OMP directive [clause[,]...]
*** OpenMP directive with continuation line
!$OMP directive clause clause &
!$OMP& clause ...
```

**C/C++:**

```plaintext
/// OpenMP directive */
#pragma omp directive [clause ..]
*** OpenMP directive with continuation line
#pragma omp directive clause \   
    clause ...
```
Conditional Compilation

equivalent:

C$ 10 IAM = OMP_GET_THREAD_NUM()

#ifdef _OPENMP
  10 IAM = OMP_GET_THREAD_NUM();
#endif

!$ IAM = OMP_GET_THREAD_NUM()
Parallel Regions (1)
The fork-join Concept

The OpenMP program starts like a serial program: single threaded.

In the beginning of the first parallel region the slave threads are started. Together with the master, they form a team.

Between the parallel regions the slave threads are put to sleep.
Parallel Regions (2)
Runtime Functions

program simple
    implicit integer (a-z)
    logical omp_in_parallel
    write (*,*) "inside parallel region? ", omp_in_parallel()
    write (*,*) "number of available processors ", omp_get_num_procs()
    write (*,*) "maximum number of threads ", omp_get_max_threads()
    call omp_set_num_threads ( max(1,omp_get_max_threads()-1) )
end program

!$omp parallel
    write (*,*) "inside parallel region? ", omp_in_parallel()
    write (*,*) "number of threads in the team ", omp_get_num_threads()
    write (*,*) "my thread id ", omp_get_thread_num()
!$omp end parallel

end program

export OMP_NUM_THREADS=3

inside parallel region?  F
number of available processors  16
maximum number of threads  3

inside parallel region?  T
number of threads in the team  2
my thread id  0

inside parallel region?  T
number of threads in the team  2
my thread id  1

redadant execution!
### Parallel Regions (3)
#### Runtime Functions

<table>
<thead>
<tr>
<th>Serial region</th>
<th>Parallel region</th>
</tr>
</thead>
</table>
| **call**omp_set_num_threads(integer)  
**void**omp_set_num_threads(int) | Set # threads to use in a team | don't |
| integer omp_get_num_threads()  
int omp_set_num_threads(void) | 1 | Return # threads |
| int omp_get_max_threads(void) | Return max # threads (OMP_NUM_THREADS) |
| int omp_get_thread_num(void) | 0 | Return thread id  
0 ... #threads-1 |
| int omp_get_num_procs(void) | Return # CPUs |
| **call**omp_set_dynamic(logical)  
**void**omp_set_dynamic(int) | Control dynamic adjustment of # threads | don't |
| logical omp_get_dynamic()  
int omp_get_dynamic(void) | .TRUE. if dynamic thread adjustment enabled,  
.FALSE. otherwise |
| logical omp_in_parallel()  
int omp_in_parallel(void) | .FALSE. | .TRUE. |
# Parallel Regions (4)

## Number of Threads in a Team

```fortran
program simple
  implicit integer (a-z)
  write (*,*) "region A: ", omp_get_thread_num()
!$omp parallel
  write (*,*) "region B: ", omp_get_thread_num()
!$omp end parallel
  write (*,*) "region C: ", omp_get_thread_num()
  call omp_set_num_threads(2)
!$omp parallel
  write (*,*) "region D: ", omp_get_thread_num()
!$omp end parallel
write (*,*) "region E: ", omp_get_thread_num()
!$omp parallel num_threads(3)
  write (*,*) "region F: ", omp_get_thread_num()
!$omp end parallel
write (*,*) "region G: ", omp_get_thread_num()
end program
```

---

**OpenMP V2.0**

- region A: 0
- region B: 0
- region B: 3
- region B: 1
- region B: 2
- region C: 0
- region D: 1
- region D: 0
- region E: 0
- region F: 2
- region F: 0
- region F: 1
- region G: 0
Parallel Regions (5) 
Adjustment of # Threads

• The default #threads is 1 when using the Sun OpenMP Compilers.
• The default #threads is equal #CPUs when using the Guide Compilers.
  => use OMP_NUM_THREADS

• With dynamic adjustment of the number of threads turned on, the runtime system is allowed to change the number of threads from one parallel region to another!

• Sun OpenMP Compilers have the dynamic adjustment turned on by default! But the #threads is only adjusted once in the beginning: The #threads is reduced, if the system is overloaded.

• Guide Compilers have the dynamic adjustment turned off by default.

• Attention: Changing the #threads from one PR to another, may produce wrong results, when using threadprivate.
  => use: call omp_set_dynamic(.false.)
Parallel Regions (6)
Sun specific

The environment variable SUMW_MP_THR_IDLE controls how deep the slave threads sleep.

SUMW_MP_THR_IDLE=spin (default) „busy waiting“ – the sleeping threads keep the CPU busy.
SUMW_MP_THR_IDLE=sleep „idle waiting“ – the sleeping threads release their CPU.
SUMW_MP_THR_IDLE=ns (seconds) SUMW_MP_THR_IDLE=nms (milliseconds) Compromise – the sleeping threads release their CPU after a while
Worksharing (1) - Principle

```
!! Worksharing (1) -- Principle !

!! A(1) !

.. 

!! B(1) !

.. 

!! C(1) !

.. 

!! A(100) !

.. 

!! B(100) !

.. 

!! C(100) !

.. 

!! Memory !

!! Processors !

!! Worksharing !

!! do i = 1, 25 !
   !
   a(i) = b(i) + c(i) !
   !
   end do !

!! do i = 26, 50 !
   !
   a(i) = b(i) + c(i) !
   !
   end do !

!! do i = 1, 100 !
   !
   a(i) = b(i) + c(i) !
   !
   end do !

!! do i = 51, 75 !
   !
   a(i) = b(i) + c(i) !
   !
   end do !

!! do i = 76, 100 !
   !
   a(i) = b(i) + c(i) !
   !
   end do !
```
Worksharing (2) – with omp_get_thread_num

### C Fortran77

```c
C$omp parallel
  if ( omp_get_thread_num() == 0 )
    do i = 1, 25
      a(i) = b(i) + c(i)
    end do
  else if ( omp_get_thread_num() == 1 )
    do i = 26, 50
      a(i) = b(i) + c(i)
    end do
  else if ( omp_get_thread_num() == 2 )
    do i = 51, 75
      a(i) = b(i) + c(i)
    end do
  else if ( omp_get_thread_num() == 3 )
    do i = 76, 100
      a(i) = b(i) + c(i)
    end do
end if
C$omp end parallel
```

### ! Fortran 90

```fortran
!$omp parallel
  select case ( omp_get_thread_num() )
    case ( 0 )
      a(1:25) = b(1:25) + c(1:25)
    case (1)
      a(26:50) = b(26:50) + c(26:50)
    case(2)
      a(51:75) = b(51:75) + c(51:75)
    case (3)
      a(76:100) = b(76:100) + c(76:100)
  end select
!$omp end parallel
```
Worksharing (3) – parallel sections

C Fortran77
C$omp parallel
C$omp sections
C$omp section
do i = 1, 25
  a(i) = b(i) + c(i)
end do
C$omp section
do i = 26, 50
  a(i) = b(i) + c(i)
end do
C$omp section
do i = 51, 75
  a(i) = b(i) + c(i)
end do
C$omp section
do i = 76, 100
  a(i) = b(i) + c(i)
end do
C$omp end sections
C$omp end parallel

/* C , abbreviated */
#pragma omp parallel sections
  for ( i=1; i<25; i++ ) { a[i] = b[i] + c[i] ; }
#pragma omp section
  for ( i=26; i<50; i++ ) { a[i] = b[i] + c[i] ; }
#pragma omp section
  for ( i=51; i<75; i++ ) { a[i] = b[i] + c[i] ; }
#pragma omp section
  for ( i=76; i<100; i++ ) { a[i] = b[i] + c[i] ; }
#pragma omp end parallel sections

! Fortran 90, abbreviated
!$omp parallel sections
  a(1:25) = b(1:25) + c(1:25)
!$omp section
  a(26:50) = b(26:50) + c(26:50)
!$omp section
  a(51:75) = b(51:75) + c(51:75)
!$omp section
  a(76:100) = b(76:100) + c(76:100)
!$omp end parallel sections

Use these abbreviations, if the parallel region only contains the parallel sections worksharing construct.
Worksharing (4) – parallel do

C Fortran77
C$omp parallel
C$omp do
  do i = 1, 100
    a(i) = b(i) + c(i)
  end do
C$omp end do
C$omp end parallel

! Fortran90, abbreviated
$omp parallel do
  do i = 1, 100
    a(i) = b(i) + c(i)
  end do

Use these abbreviations, if the parallel region only contains the parallel do worksharing construct.

/* C */
#pragma omp parallel
{  
#pragma omp for
    for ( i=1; i<100; i++ ) {
      a[i] = b[i] + c[i] ;
    }
}

/* C , abbreviated */
#pragma omp parallel for
    for ( i=1; i<100; i++ ) {
      a[i] = b[i] + c[i] ;
    }
NEW: OpenMP V2.0

```
! Fortran90 only
$omp parallel
$omp workshare
  a(1:100) = b(1:100) + c(1:100)
  d(2:99) = a(1:98) + a(3:100)
$omp end workshare
$omp end parallel
```

```
! abbreviation, Fortran90 only
$omp parallel workshare
  a(1:100) = b(1:100) + c(1:100)
  d(2:99) = a(1:98) + a(3:100)
$omp end parallel workshare
```

Attention: hidden barriers
Worksharing (6) – single

```c
#pragma omp parallel
{
    ...
    #pragma omp single
    {
        printf "one thread only\n";
    }
    ...
}
```
```cpp
#include <omp.h>
#define SIZE 10000

template <typename T, int size>
class Array {
private:
    T data[size];
public:
    Array() /* Default constructor */ { }
    ~Array() /* Array destructor */ { }
    Array(const T& r) /* Regular constructor */ {
        #pragma omp parallel for
        for (int i=0; i<size; i++) data[i] = r;
    }
    Array(const Array& rhs) /* Copy constructor */ {
        #pragma omp parallel for
        for (int i=0; i<size; i++) data[i] = rhs[i];
    }

    // Read only and read/write subscript operators
    const T& operator[](int i) const { return data[i]; }
    T& operator[](int i) { return data[i]; }

    Array& operator=(const Array& rhs) /* Assignment operator */ {
        #pragma omp parallel for
        for (int i=0; i<size; i++) data[i] = rhs[i];
        return *this;
    }

    template <typename T, int size>
    Array<T,size> operator+(const Array<T,size>& a, const Array<T,size>& b) {
        Array<T,size> ret;
        #pragma omp parallel for
        for (int i=0; i<size; i++) ret[i] = a[i] + b[i];
        return ret;
    }

    template <typename T, int size>
    void do_it(int repeat) {
        Array<double,SIZE> a(1.0), b(2.0), c(3.0), d(4.0), res(5.0);
        for (int i=0; i<repeat; i++) res = a * b - c + d;
    }
};
```
Orphaning

Directives belonging to a parallel region do not need to be placed in the same program unit.

In this example the worksharing construct is ignored, if the subroutine is called from a serial region. It is effective when the subroutine is called from a parallel region.
## Scope of Variables (1) – Intro

<table>
<thead>
<tr>
<th>Scope</th>
<th>shared</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>global</td>
<td>valid for all threads and in all program units</td>
<td>private for all threads, but accessible in all program units</td>
</tr>
<tr>
<td>local</td>
<td>valid for all threads, but only in the respective program unit</td>
<td>private for all threads, and valid only in the respective program unit</td>
</tr>
</tbody>
</table>
Scope of Variables (2) – data scope

By default all variables (in the static extend) are accessible by all threads, they are **shared**.

An exception are loop iteration variables, which automatically are **private**.

The default can be changed by: default (shared|private|none) resp. default (shared|none) (C/C+)

The default clause only effects variables in the static extend!
Scope of Variables (3) – defaults

- The shared memory programming model: By default all variables are shared.
- **Global** variables are shared:
  - Fortran: common blocks
  - Fortran: variables with the save attribute
  - Fortran: initialized variables
  - Fortran: module variables
  - C: Variables with a static or extern attribute
- Exception: Loop iteration variables are private.

- **ATTENTION:**
  Local variables of a subprogram called in a parallel region are put onto the stack. They are private (dynamic extend).
- Fortran: Variables of a subprogram called in a parallel region having the save attribute are shared.
- C/C++: static Variables of a subprogram called in a parallel region are shared.
Scope of Variables (4f) – defaults

program main
  integer n
  common / comblk / n
  double precision pi
  ...
  !$omp parallel
  do
    call calc_pi (pi)
  end do
  !$omp end parallel
  ...
end program Main

subroutine calc_pi (pi)
  integer :: n
  common / comblk / n
  double precision, save :: sum, h
  double precision :: a, x, f, pi
  ...
  pi = ...
  return
end subroutine calc_pi
Scope of Variables (4c) – defaults

```c
int n;
void calc_pi(double *);

main()
{
    double pi;
    ...
    #pragma omp parallel
    {
        for ( . . . ) {
            call calc_pi ( &pi )
        }
    } /* end of parallel region */
    ...
} /* end of program main */
```

```c
extern int n;
void calc_pi (double *pi)
{
    int i;
    static double sum, h;
    double a, x, f;
    ...
    *pi = ...
}
```
Scope of Variables (5) – private

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

int main(void)
{
    int i;
    i = 42;
    printf("before PR: i=%d\n", i);

    #pragma omp parallel private(i)
    {
        printf("(%d): i=%d\n",omp_get_thread_num(),i);
        i += omp_get_thread_num();
        printf("(%d):   i:%d\n",omp_get_thread_num(),i);
    }

    printf("\nundefined after PR: i=%d\n", i);

    return 1;
}
```

Output:
before PR: i=42
(1): i=0
(3): i=0
(1):   i: 1
(2): i=0
(3):   i: 3
(0): i=0
(0):   i: 0
(2):   i: 2
after PR: i=42

an uninitialized copy is allocated for each thread

according to the specifications i is undefined after the parallel region !!!
Scope of Variables (6) – firstprivate

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

int main(void)
{
    int i;
    i = 42;
    printf("before PR: i=%d\n", i);

    #pragma omp parallel firstprivate(i)
    {
        printf("(\d): i=%d\n", omp_get_thread_num(),i);
        i += omp_get_thread_num();
        printf("(\d): i=%d\n", omp_get_thread_num(),i);
    }
    printf("undefined after PR: i=%d\n", i);

    return 1;
}
```

Output:
before PR: i=42
(1): i=42
(3): i=42
(1): i=43
(2): i=42
(3): i=45
(0): i=42
(0): i=42
(2): i=44
after PR: i=42

The private copy is initialized with the original value before the parallel region.

according to the specifications \textit{i} is undefined after the parallel region !!!

OpenMP - Introduction, Dieter an Mey, 18 January 2003
Scope of Variables (7) – Lastprivate

```c
!$omp parallel default(none) shared(a,b,c)
!$omp do lastprivate(i)
    do i = 1, 100
        a(i) = b(i) + c(i)
    end do
!$omp end do
print *, i                                       ! 101
!$omp end parallel
print *, i                                       ! 101
```

i gets the value of the (sequentially) last iteration.
Scope of Variables(8) – threadprivate

module TP1
! OpenMP V2.0 only
  integer :: i5
  !$omp threadprivate(i5)
end module TP1

module TP2
  integer :: i6
  common / comblk3 / i6
  !$omp threadprivate(/comblk3/)
end module TP2

program test
  use TP1
  use TP2
  ! illegal
  integer :: i1
  !$omp threadprivate(i1)
  ! OpenMP V2.0 only
  integer, save :: i2
  !$omp threadprivate(i2)
  ! illegal
  integer :: i3
  common / comblk1 / i3
  !$omp threadprivate(i3)
  integer :: i4
  common / comblk2 / i4
  !$omp threadprivate(/comblk2/)
  
  Integer :: omp_get_thread_num
  Integer :: omp_get_num_threads
  
  call omp_set_num_threads(8)
  call omp_set_dynamic(.false.)

  !$omp parallel
    i1 = omp_get_thread_num()
    i2 = omp_get_thread_num()
    i3 = omp_get_thread_num()
    i4 = omp_get_thread_num()
    i5 = omp_get_thread_num()
    i6 = omp_get_thread_num()
  !$omp end parallel

  ...

  !$omp parallel
    write(*,"(6i8)")&
      i1,i2,i3,i4,i5,i6
  !$omp end parallel

end program test
Scope of Variables(9) – threadprivate

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

int t_private;
#pragma omp threadprivate(t_private)

void foo(void) {
    printf("(foo) thread(%d): t_private=%d\n", omp_get_thread_num(), t_private);
}

main()
{
    int priv;
    #pragma omp parallel
    {
        t_private = omp_get_thread_num();
        foo();
    }
}
## Scope of Variables (10) – Overview

<table>
<thead>
<tr>
<th>Scope</th>
<th>Default (F: Fortran)</th>
<th>Automatic (C: C/C++)</th>
<th>Stack (F: stackvar)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>local</strong></td>
<td>F: common</td>
<td>F: common + threadprivate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F: module + use</td>
<td>F: module + use + threadprivate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C: file scope</td>
<td>C: file scope + threadprivate</td>
<td></td>
</tr>
<tr>
<td><strong>global</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>F: common</td>
<td>F: common + threadprivate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F: module + use</td>
<td>F: module + use + threadprivate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C: file scope</td>
<td>C: file scope + threadprivate</td>
<td></td>
</tr>
<tr>
<td><strong>local</strong></td>
<td>default</td>
<td>private-clause</td>
<td></td>
</tr>
<tr>
<td>(static extend)</td>
<td>shared-Klausel</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>local</strong></td>
<td>F: save, initialized</td>
<td>default (f90 –stackvar )</td>
<td></td>
</tr>
<tr>
<td>(dynamic extend)</td>
<td>C: static, extern</td>
<td>C: automatic variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C: heap (malloc, new)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*A pointer may be private!*
### Scope of Variables (11) – Overview

<table>
<thead>
<tr>
<th>Scope</th>
<th>C: automatic variables</th>
<th>F: save, initialized</th>
<th>A pointer may be private!</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>default (f90 –stackvar )</td>
<td>default (f90 –stackvar )</td>
<td></td>
</tr>
<tr>
<td>(static extend)</td>
<td>shared</td>
<td>shared-Klausel</td>
<td></td>
</tr>
<tr>
<td>local</td>
<td>file scope</td>
<td>common</td>
<td></td>
</tr>
<tr>
<td>(dynamic extend)</td>
<td>array scope</td>
<td>module + use</td>
<td></td>
</tr>
<tr>
<td>global</td>
<td>heap (malloc, new)</td>
<td>heap (malloc, new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F: common</td>
<td>C: static, extern</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F: module + use</td>
<td>C: heap (malloc, new)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C: file scope</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heap allocated memory is shared. Therefore the administration has to be synchronised. Frequent allocation of dynamic memory with `malloc/new` may lead to performance degradations. By linking with `cc –xopenmp ... –lmtmalloc` this may be improved.
Critical Region (1)

**Processors**

- Do $i = 1, 25$
  - $s = s + a(i)$
  - End do
- Do $i = 26, 50$
  - $s = s + a(i)$
  - End do
- Do $i = 51, 75$
  - $s = s + a(i)$
  - End do
- Do $i = 76, 100$
  - $s = s + a(i)$
  - End do

**Memory**

All processors want to read and write $S$ simultaneously!
Critical Region (2)

only one processor is allowed to enter the critical region at a time

=> performance ? 😞
Critical Region (3) – critical / end critical

```fortran
!$omp parallel do private(i)
    do i = 1, 100
        s = s + a(i)
    end do
!$omp critical
    s = s + a(i)
!$omp end critical
    end do
!$omp end parallel do
```

only one processor is allowed to enter the critical region at a time.

As the loop body consists of a critical region only, the parallel program will run much slower.

```
#pragma omp parallel for private(i)
    for ( i=1; i<100; i++ ) {
        #pragma omp critical
            { s += a[i]; }
    }
```
The critical region is extracted out of the loop => performance!

```plaintext
DO i = 1, 25
   s1 = s1 + a(i)
END DO
S = S + S1

DO i = 26, 50
   s2 = s2 + a(i)
END DO
S = S + S2

DO i = 51, 75
   s3 = s3 + a(i)
END DO
S = S + S3

DO i = 76, 100
   s4 = s4 + a(i)
END DO
S = S + S4
```
Now the partial sums are calculated in parallel. The critical region is entered only once per thread.

Now the partial sums are calculated in parallel. The critical region is entered only once per thread.
Critical Region (6) – named critical region

Critical regions may be named. If multiple critical regions are used, this may be advantageous. The name of a critical region is a global name.

```c
!$omp parallel private(i,s_local)
    s_local = 0.0
!$omp do
    do i = 1, 100
        s_local = s_local + a(i)
    end do
!$omp end do
!$omp critical (sum)
    s = s + s_local
!$omp end critical (sum)
!$omp end parallel
```

```c
#pragma omp parallel private(i,s_local)
{
    s_local = 0.0;
#pragma omp for nowait
    for ( i=1; i<100; i++ ) { s_local += a[i]; }
#pragma omp critical (sum)
    { s += s_local; }
}
```
Critical Region (7) – atomic

If the critical region consists of one simple statement only

\[ var = var \text{ op expression} \]

or

\[ var = \text{ intrinsic} (\ var, \text{expression}) \]

resp.

\[ var \text{ binop} = \text{ expression}; \]

or

\[ var++; \ var--; ++\var;--\var \]

the \textbf{atomic} directive, which is mapped onto fast hardware mechanisms, may be used.
Reductions – reduction clause

The reduction clause is tailored for this frequently occurring case.

\texttt{reduction\{}\{op\textit{intrinsic}\}\texttt{:list}\texttt{}}

with

\begin{align*}
op & = \{ + | * | - | \text{.and.} | \text{.or.} | \text{.eqv.} | \text{.neqv.}\} \\
or & = \{ \max, \min, \text{iand}, \text{ior}, \text{iior} \}
\end{align*}

\texttt{list} is a comma separated list of variables.

\begin{verbatim}
do i = 1, 100
   s = s + a(i)
end do

!$omp parallel do private(i) reduction(+:s)
   do i = 1, 100
      s = s + a(i)
   end do
!$omp end parallel do

#pragma omp parallel for private(i) \ reduction(+:s)
   for ( i=1; i<100; i++ ) {
      s += a[i];
   }
\end{verbatim}
Reductions – Rounding Errors

• When parallelizing such recursions different rounding errors may occur.
  You may see different rounding errors:
  • serial – serial (different compiler options)
  • serial – parallel (OpenMP or autoparallel)
  • parallel – parallel (multiple program runs)
  • parallel – parallel (different processor number)
• First aid:
  • reduce the serial optimization by the compiler
    `-fsimple=0 -xnolibmopt`
  • use partial parallelization
  • use the `--noreduction` option when autoparallelizing

```c
!$omp parallel do reduction(+:s)
    do i = 1, 100
        s = s + a(i) * b(i) / c(i)
    end do
$omp end parallel do
```

```c
parallel do
    do i = 1, 100
        tmp(i) = a(i) * b(i) / c(i)
    end do
$omp end parallel do
```

```c
parallel do
    do i = 1, 100
        s = s + tmp(i)
    end do
```

Synchronization – barrier

Each thread has to wait at the barrier until all other threads reach this barrier as well.

```c
!$omp parallel
...
print *, “arrived \n“;
!$omp barrier
print *, “let’s continue together“;
...
!$omp end parallel
```

The following constructs have an implicit barrier, Unless it is turned off with an additional `nowait`-clause:

- `end do`
- `end sections`
- `end single`
- `end workshare`
Synchronization – master

This program segment is only executed by the master thread. All the other threads immediately continue their execution after the master section.

```c
!$omp parallel
...
!$omp master
print *, "the master only !";
!$omp end master
...
!$omp end parallel
```

```c
#pragma omp parallel
{
    ...
    #pragma omp master
    printf "the master only !\n";
    ...
}
```

In contrast to the single directive:

**No** implicit barrier at the end!
Synchronization – nowait

The **nowait** clause can be used to avoid unnecessary barriers. In many cases barriers are the main obstacles to speed-up.

```c
!$omp parallel
!$omp do schedule(static)
do i = 1, 100
    a(i) = ...
end do
!$omp end do nowait

!$omp do schedule(static)
do i = 1, 100
    b(i) = a(i) **2
end do
!$omp end do nowait

!$omp end parallel
```

**ATTENTION:**
with schedule(dynamic)
this may go wrong!
program main
  implicit integer (a-z)

  !$omp parallel
    !$omp do
      do i = 1, omp_get_num_threads()
        print *, 'me: ', omp_get_thread_num(), ' i: ', i
      end do
    !$omp end do

    !$omp do ordered
      do i = 1, omp_get_num_threads()
        !$omp ordered
          print *, 'me: ', omp_get_thread_num()
        !$omp end ordered
      end do
    !$omp end do

  !$omp end parallel
end
Synchronization - flush

!$omp flush [(list)]

The flush directive guarantees that all memory operations are finalized. [ according to the given list ]
The related variables will be fetched from memory later on. Candidates are shared variables.

The following constructs imply a flush:
- barrier
- critical and end critical
- end do
- end sections
- end single
- end workshare
- ordered and end ordered
- parallel and end parallel

BUT: No implicit flush at
- do
- master and end master
- sections
- single
- workshare
- end {do|sections|single} nowait
Ordered Output without an ordered clause - flush

```fortran
program ordered
    integer me, ticket, omp_get_thread_num
    !$OMP parallel private(me) shared(ticket)
    me = omp_get_thread_num()
    !$OMP single
    ticket = 0
    !$OMP end single
    do while ( ticket < me )
        !$OMP flush(ticket)
    end do
    call work(me) ! do ordered work here
    ticket = ticket + 1
    write (*,*) me
    !$OMP flush(ticket)
    !$OMP end parallel
end program flush
```

waiting loop