LUND UNIVERSITY

HPC2N, UmU

Data for parallel regions

Joachim Hein
LUNARC & Centre of Mathematical Sciences
Lund University

Pedro Ojeda May
HPC2N
Umeå University

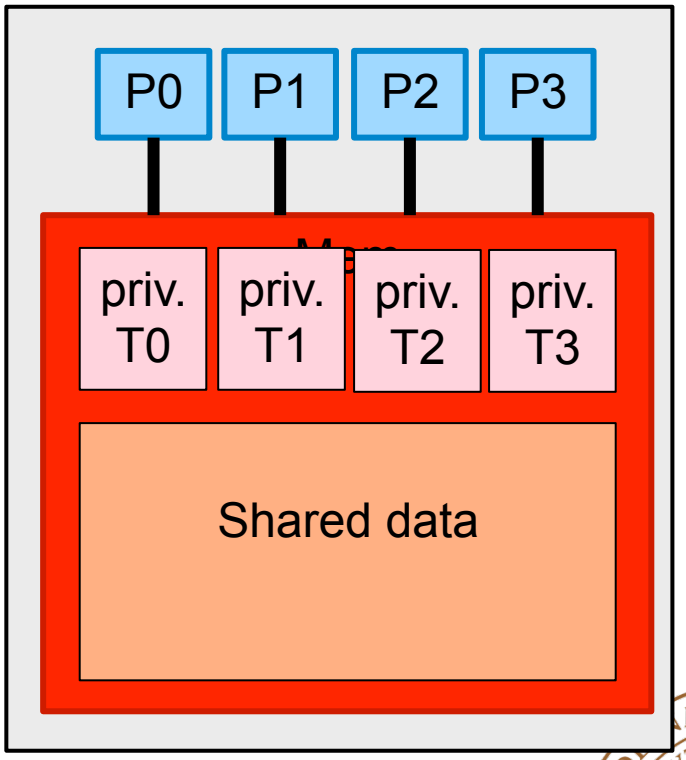
Overview

- Data
  - What is private data
  - What is shared data
  - How to control which is which
- Race conditions
  - Basic constructs to avoid data races

SHARED & PRIVATE DATA

Private and shared data

- In parallel region
  - Shared and private data
- Shared data
  - Every thread can access (potential for conflict)
  - Unchanged on entry to par. region
  - Survives after end of par. Region
- Private data
  - Each thread: **own private copy**
  - Normally **uninitialised** at begin of parallel region
  - Contents typically **lost** when parallel region finishes



Fortran: Clauses to control data sharing

- For data declared before start of parallel region:
  - Use clause `shared` to declare a data structure as shared
  - Use clause `private` to declare a data structure as private

```
 integer :: a=5, b

!$omp parallel &
!$omp shared(a) private(b)
 b = a &
    + omp_get_thread_num()
 print *, result=",b
!$omp end parallel
```

C: Clauses to control data sharing

- For data declared before start of parallel region:
  - Use clause `shared` to declare a data structure as shared
  - Use clause `private` to declare a data structure as private

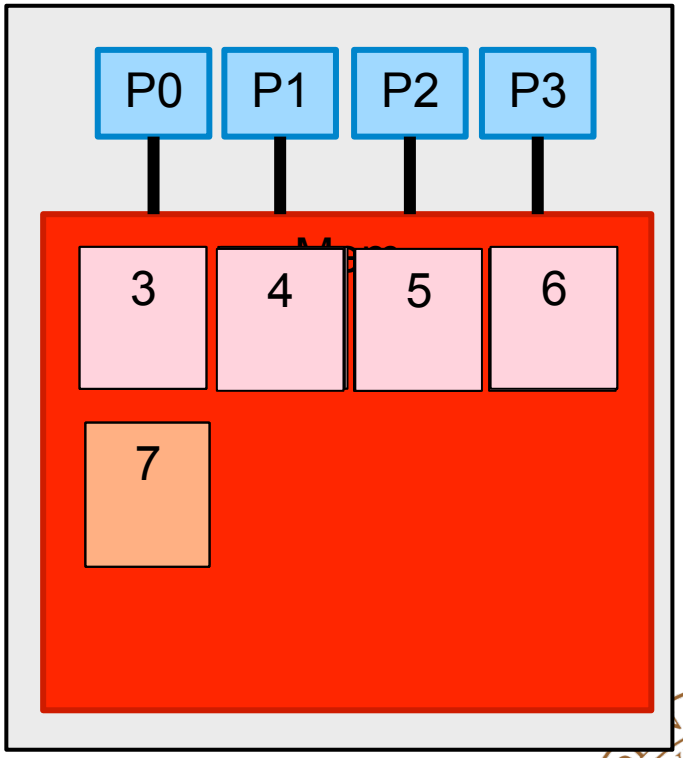
```
int a, b;
a = 5;
#pragma omp parallel \
  shared(a) private(b)
  {
    b = a +
    omp_get_thread_num();
    printf("%i\n", b);
  }
```

Private data

- Typically control variables, examples include
  - Thread identification
  - Loop indices
  - Variables internal to the algorithm
- Most variables declared inside a parallel region are private
  - Declared inside the block (C/C++)
  - In subroutine/function called from inside parallel region
  - **Exceptions include:**
    - static (C/C++) or save (Fortran) variables
    - file scope (C/C++) or COMMON blocks
    - Variables passed by reference inherit
- In Fortran**: Care needed** with `COMMON` and `EQUIVALENCE`

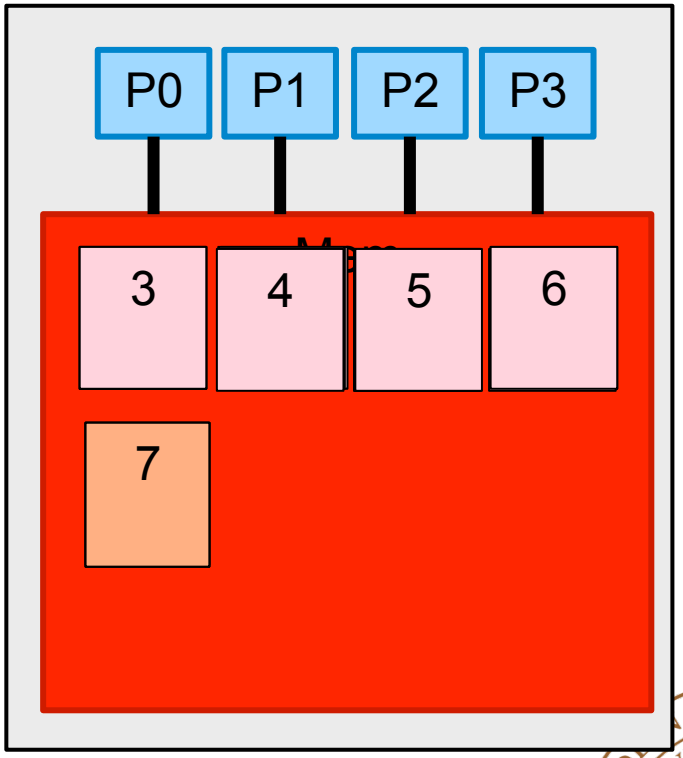
Example (Fortran): Memory movements for private data

```
integer b
b=5

!$OMP parallel &
!$OMP      private(b)
 b=get_omp_thread_num()
 b= b+3
!$OMP end parallel

b=7
```



Example (C-code): Memory movements for private data

```
int b;
b=5;
#pragma omp parallel \
  private(b)
{
 b=get_omp_thread_num();
 b+=3;
}
b=7;
```



Shared Data

- Majority of the data
  - Typically: large data structures (e.g. arrays)
- Keeps its value on
  - Entry to parallel region
  - Exit from parallel region
- Every thread can access (read and/or write) the data
  - Save: when read by multiple threads (only read)
  - **Danger looms when:**
    - Multiple threads access the same memory location
    - At least one of these is a write access
    - Easily results in a ***"Racecondition"***

Fortran-example: Initialisation of a Vector

$$v_i = 4i$$

```
 integer, parameter:: vleng=120
 integer:: vect(vleng), myNum, start, fin, i

!$omp parallel shared(vect) &
!$omp private(myNum, start, fin, i)
    myNum = vleng/omp_get_num_threads()
    start = 1 + omp_get_thread_num()     * myNum
    fin   =     (omp_get_thread_num()+1) * myNum
    do i = start, fin
      vect(i) = 4*i  ! threads write diff. elements
    enddo
!$omp end parallel
```

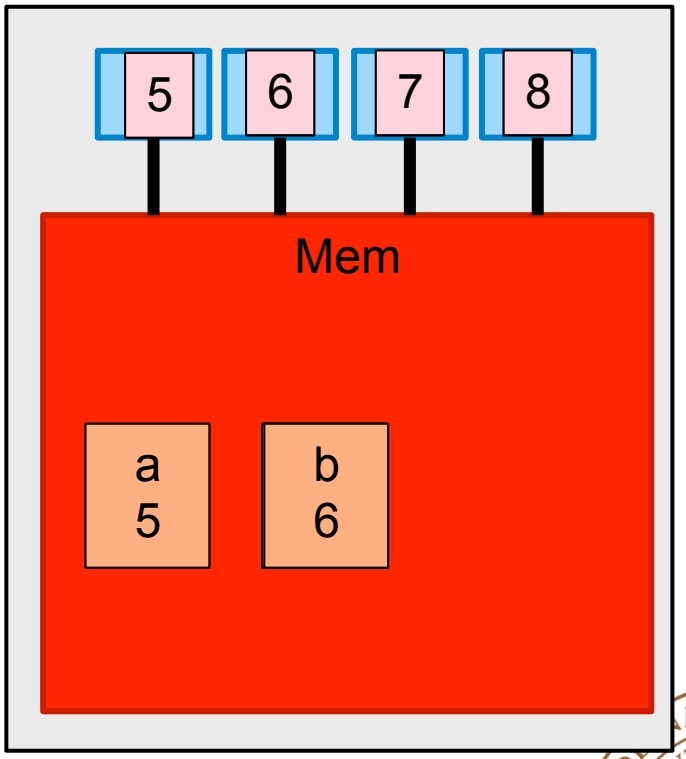
C-example: Initialisation of a Vector

$$v_i = 4i$$

```
const int vleng=120;
int vect[vleng], myNum, start, fin, i;

#pragma omp parallel shared(vect) \
  private(myNum, start, fin, i)
  {
    myNum = vleng/omp_get_num_threads();
    start = omp_get_thread_num() * myNum;
    fin = start + myNum;
    for (i = start; i < fin; i++)
      vect[i] = 4*i; // tasks write diff. elements
  }
```

Fortran example: Write conflict for shared data

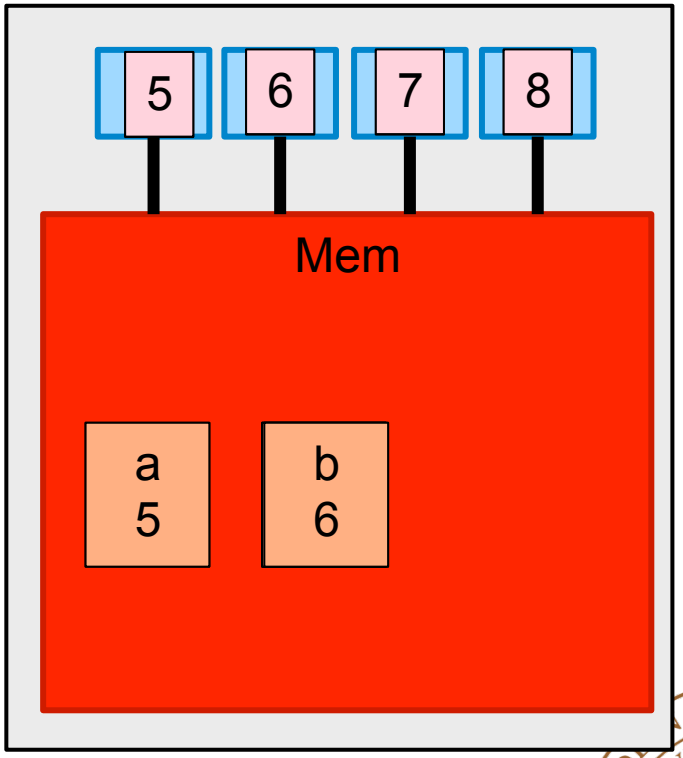
```
 integer :: a, b
 a=5
!$OMP parallel &
!$OMP    shared(a,b)
   b = a +
    omp_get_thread_num()
   print *,"updated"
   print *,"my b:", b
!$OMP end parallel
```



- final b-value: “random”
- Individual tasks might print b before b has final value!

C-example:
Write conflict for shared data

```
 int a, b;
 a=5;
#pragma omp parallel \
 shared(a,b)
 {
  b = a +
   omp_get_thread_num();
  printf("updated\n");
  printf("my b: %i", b);
 }
```



- final b-value: “random”
- Individual tasks might print b before b has final value!

Default clause

- Can be used on a `parallel` or `task` construct
- Determines data sharing of implicitly determined variables
  - In C

    `default(shared|none)`
  - In Fortran

    `default(shared|none|private|firstprivate)`
- For `parallel` construct, if no `default` clause supplied, `default(shared)` applies
- **Recommendation: `default(none)`** typically good idea!
  - All variables accessed in parallel region must be declared as `shared`, `private`, …

barrier, critical & atomic

FIXING DATA RACES

Data races

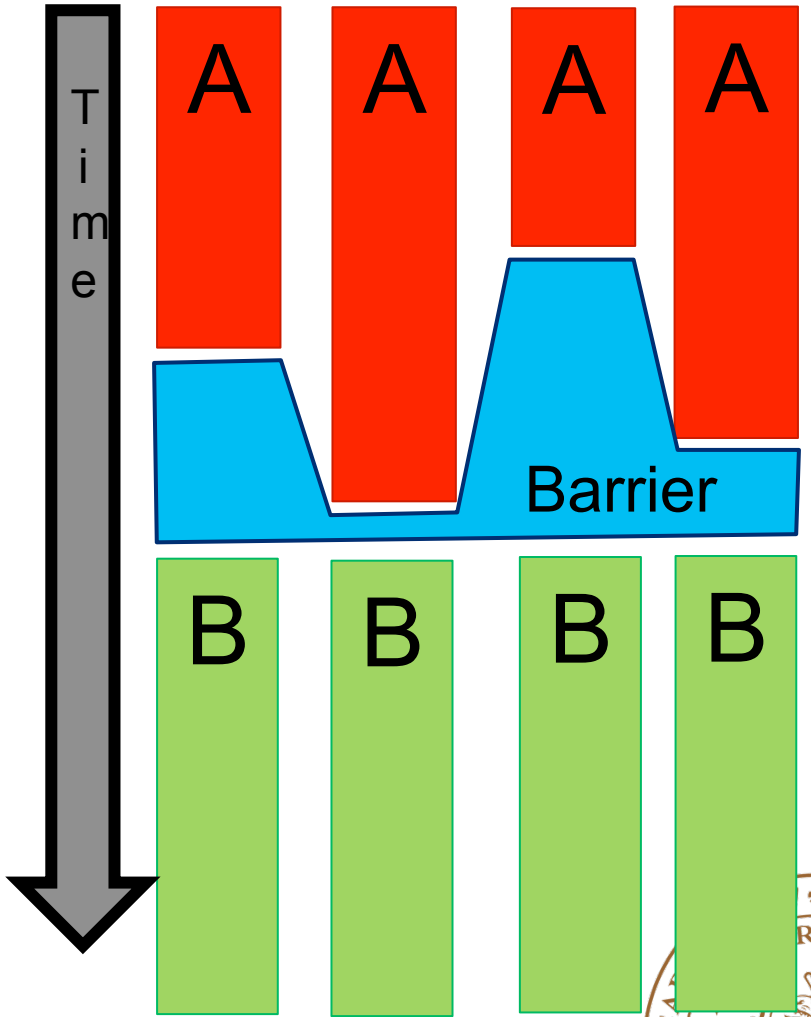
- Data races occur easily when:
  - Data is (to be) **written** to **shared memory location** on one thread
  - Accessed (read and/or write) by **another** thread
- Problems include:
  - Has writing thread reached the write statement?
  - If so, is the data in the memory system (cache, main memory) or still held in a processor register?
- Constructs to deal with data races impact code performance
  - Though we have no interest in “fast garbage”!!!

Barrier and synchronisation

- Barrier enforces synchronisation

```
#pragma omp barrier
!$omp barrier
```

- All threads wait for the last one to arrive
- Registers are flushed to memory system
- All threads must have the barrier in their line of execution
  - Otherwise: **dead-lock**!



Example: Data race in matrix transpose

```
!$omp parallel default(none) &
!$omp  private(mysize,tid,i,j) shared(matrix,mtrans)
    tid = omp_get_thread_num()
    mysize = nsize/omp_get_num_threads()
    do j = 1 + tid*mysize, (tid+1)*mysize
      do i = 1, nsize
        matrix(i,j) = 1000.0 * j + i
      enddo
    enddo
!$omp barrier
    do j = 1 + tid*mysize, (tid+1)*mysize
      do i = 1, nsize
        mtrans(i,j) = matrix(j,i)
      enddo
    enddo
!$omp end parallel
```


Example: Data race in matrix transpose

```
#pragma omp parallel default(none) \
  private(mysize, tid, i, j) shared(matrix, mtrans)
  { tid = omp_get_thread_num();
    mysize = nsize/omp_get_num_threads();

    for (i = tid*mysize; i < (tid+1)*mysize; i++)
      for (j = 0; j < nsize; j++)
        matrix[i][j] = 1000.0 * j + i;
    #pragma omp barrier
    for (i = tid*mysize; i < (tid+1)*mysize; i++)
      for (j = 0; j < nsize; j++)
        mtrans[i][j] = matrix[j][i];
  }
```

Critical regions

- Critical region: protect update of share memory location
- Only one thread executes critical region at the time

```
#pragma omp critical (name)
{
   code-block
}
```

- Name is optional
  - Single thread in all regions of same name
  - Single thread in all unnamed regions
- Implies a register flush at entrance/exit
- Also useful to execute non-thread-safe functions
- Performance penalty due to serialisation ☹

Critical regions

- Critical region: protect update of share memory location
- Only one thread executes critical region at the time

```
!$omp critical (name)
   code-block
!$omp end critical (name)
```

- Name is optional
  - Single thread in all regions of same name
  - Single thread in all unnamed regions
- Implies a register flush at entrance/exit
- Also useful to execute non-thread-safe functions
- Performance penalty due to serialisation ☹

Fortran syntax for critical region

- In Fortran a start and end mark is required

```
!$omp critical (name)
   code-block
!$omp end critical (name)
```

- Behaviour as in C, see above

Example for use of `critical`

$$\sum_{k=0}^{n-1} e^k$$

```
sum=0.0_dpr
!$omp parallel default(none) &
!$omp    shared(sum) private(tid, cont)

     tid = omp_get_thread_num()
     cont = func( tid )
!$omp critical (exp_up)
       sum = sum + cont
       print *, tid,": c=", cont, " s=", sum
!$omp end critical (exp_up)

!$omp end parallel
```

Example for use of `critical`

$$\sum_{k=0}^{n-1} e^k$$

```
sum=0.0;
#pragma omp parallel default(none) \
        shared(sum) private(tid, cont)
  {
    tid = omp_get_thread_num();
    cont = func( tid );
    #pragma omp critical (exp_up)
    {
      sum += cont;
      printf("%i: c=%f s=%f\n", tid, cont, sum);
    }
  }
```


Atomic

- Light-weight alternative to `critical` in simple cases
- Works with simple statements only
- Can use special hardware instructions if exist
- Flushes “protected” variable on entry and exit

- Four different versions, from OpenMP 3.1
  - `read`
  - `write`
  - `update`
  - `capture`
- OpenMP 4.0: adding `seq_cst` to `atomic` flushes all variables
  - Important for controlling instruction reordering
  - Example: implementing a lock

Atomic read of shared variable

- Protects only the reading of scalar intrinsic variable `x`
- Flushes `x` on entry and exit

- Fortran:

```
!$omp atomic read
  v = x
```

- C:

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

Atomic write of shared variable

- Protects only the writing of scalar intrinsic variable `x`
- No protection of evaluation of *`expr`* on the right hand side
- Flushes `x` on entry and exit

- Fortran:

```
!$omp atomic write
  x = expr
```

- C:

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

- Example expressions:

```
x = 5;
```

```
x = v;
```

```
x = func(a);
```

Atomic update of shared variable

- Only protects the update of `x`, not the right hand call to `func()`

```
x += func(a);
x = x + func(a)
```

- Works with simple statements only
- Can use special hardware instructions if exist
- Flushes update variable on entry and exit

- **Rem:** Only `atomic` operation prior to OpenMP 3.1

Fortran statements: `atomic update`

- Examples:

```
!$omp atomic update
  x = x + 1

!$omp atomic update
  x = x + f(a)
```

- The evaluation of `f(a)` is **not** protected – use `critical` if needed!
- The `update` is optional – consistency with old OpenMP standard

- Allowed operations:
  - `x = x` *`operator expr`*
  - `x =` *`expr operator`* `x`
  - `x =` *`intr_proc`*`(x,` *`expr_list`*`)`
  - `x =` *`intr_proc`*`(`*`expr_list`*`, x)`
- *`x`* scalar, intrinsic type
- *`operator`* is one of:
  `+, *, -, /,`
  `.AND.,.OR.,.EQV.,.NEQV.`
- *`intr_proc`* is one of
  `MAX,MIN,IAND,IOR,IEOR`

F90-example: Vector norm

$$\sqrt{\sum_i v(i) * v(i)}$$

```
  norm = 0.0D0
!$omp parallel default(none) &
!$omp shared(vect,norm) private( myNum, i, lNorm)
  lNorm = 0.0D0
  myNum = vleng/omp_get_num_threads() !local size
  do i = 1 + myNum * omp_get_thread_num(), &
             myNum *(1+omp_get_thread_num())
    lNorm = lNorm + vect(i)*vect(i)
  enddo
!$omp atomic update
  norm = norm + lNorm
!$omp end parallel
  norm = sqrt(norm)
```

C statements: `atomic update`

Examples:

```
#pragma omp atomic update
     x++;

#pragma omp atomic update
     x += f(a);
```

- Evaluation of f(a) is **not** protected – use `critical` if needed!
- The `update` is optional – consistency with old OpenMP standard

- Allowed operations:
  - `x` *`binop`*`= expr;`
  - `x++;`
  - `++x;`
  - `x--;`
  - `--x;`
  - `x = x` *`binop expr`*`;`
- `x` lvalue, scalar
- *`binop`* is one of:
  - `+, *, -, /, &, ^, |, <<, >>`

C-example: Vector norm

$$\sqrt{\sum_i v(i) * v(i)}$$

```
 norm = 0.0;
#pragma omp parallel default(none) \
  shared(vect,norm) private( myNum, i, lNorm)
  { lNorm = 0.0;
    myNum = vleng/omp_get_num_threads(); //local size
    for (i = myNum*omp_get_thread_num();
         i < myNum*(1+omp_get_thread_num()); i++)
        lNorm += vect[i]*vect[i];
#pragma omp atomic update
    norm += lNorm;
  }                          // synchonise at end parallel
  norm = sqrt(norm);
```

Atomic capture of a shared variable

- Atomic capture:
  - Update a shared variable
  - Keep a thread private copy of either (but not both)
    - Old value
    - New value

- Restrictions apply

C statements: `atomic capture`

```
#pragma omp atomic capture
  statement or strct. blk.
```

Allowed statements (OpenMP 4.0):

```
v = x++;
v = x--;
v = ++x;
v = --x;
v = x binop= expr;
v = x = x binop expr;
v = x = expr binop x;
```

Allowed structured blocks:

```
{v = x; x binop= expr;}
{x binop= expr; v = x;}
{v = x; x = x binop expr;}
{v = x; x = expr binop x;}
{x = x binop expr; v = x;}
{x = expr binop x; v = x;}
{v = x; x = expr;}
{v = x; x++;}
{v = x; ++x;}
{++x; v = x;}
{x++; v = x;}
{v = x; x--;}
{v = x; --x;}
{--x; v = x;}
{x--; v = x;}
```

Fortran statements: `atomic capture`

```
!$omp atomic capture
  update-statement
  capture statement
!$omp end atomic
```

or

```
!$omp atomic capture
  capture statement
  update-statement
!$omp end atomic
```

- Allowed update statements:
  - `x = x operator expr`
  - `x = expr operator x`
  - `x = intr_proc(x, expr_list)`
  - `x = intr_proc(expr_list, x)`
- Allowed capture statements:
  - `v = x`

Summary

- Private data
- Shared data

- Prevent race condition
  - `barrier`
  - `critical`
  - `atomic`

- Examples also showed parallelisation strategies