HPC2N, UmU

More on worksharing

Pedro Ojeda & Joachim Hein

High Performance Computing Center North &
Lund University

Outline

- `single` and `master`
- `if` clause
- Flushes and implicit barriers
- `nowait` clause
- orphan directives

WORK FOR SINGLE THREADS

Single

- Workshareing construct – place inside parallel region
- Does what is says: a single thread executes the region
- Not specified which thread is executing the region
  - Other threads wait in `barrier` at the end
- Useful for e.g.:
  - Guard when writing to shared variables
  - Guard when writing to stdout or file
    - Enforcing a single write
  - Guard when reading from stdin or file
    - Data read once
  - Starting `task` – later in course

Fortran-example use for `single`

```
!$omp parallel shared(a,b,n) private(i)
  !$omp single
    a = omp_get_num_threads()
  !$omp end single   //implied barrier, required!
  !$omp do
  do i=1, n
    b(i) = a
  enddo
!$omp end parallel
```

C-example use for `single`

```
#pragma omp parallel shared(a,b,n) private(i)
{
  #pragma omp single
  {
    a = omp_get_num_threads();
  }   // implied barrier, required!
  #pragma omp for
  for (i=0; i<n; i++)
    b[i] = a;
}
```

Master

- Similar to `single`

- But
  - work done on Master (thread 0)
  - **No** implied barrier/synchronisation

- More light weight than `single` if barrier is not needed

- Deterministic behaviour

Ordered

- Execute (part of) a loop body in **sequential** order
  - Performance penalty!!!
  - Requires enough other parallel work to pay overheads!
- Thread working first iteration enters, others wait
  - When done, thread for second iteration enters
  - And so on!
- `ordered` also required on the loop construct `(omp for)`
- No more than one `ordered` per thread and iteration
- Use cases include:
  - Ordered printing from parallel loops
  - Debugging, e.g.: data races

Example for ordered

```
#pragma omp parallel default(none) shared(b)
  {
#pragma omp for ordered schedule(dynamic,1)
    for (int i=0; i< PSIZE; i++)
      {
        b[i] = expensiveFunction(i);
#pragma omp ordered
        printf("b[%3i] = %4i\n", i, b[i]);
      }
  }
```

CLAUSES FOR PARALLEL

If clause

- Can be specified on `parallel` construct
- If evaluates to false, no parallel region is stated
  - Code executes serial
  - Useful for runtime evaluation, e.g.: loop count to small

Fortran-example for `if`

```
integer n=20
!$omp parallel if (n > 5) shared(n)

!$omp single
   print *,"The n is: “, n
!$omp end single

   print *,"Hello, I am thread",        &
       omp_get_thread_num(), “ of”      &
       omp_get_num_threads()
!$omp end parallel
```

C-example for `if`

```
int n=20;
#pragma omp parallel if (n > 5) shared(n)
{
#pragma omp single
   printf("The n is %i\n", n);

   printf("Hello, I am thread %i of %i\n",
          omp_get_thread_num(),
          omp_get_num_threads() );
}
```

Clause `num_threads`

- Clause `num_threads` can be added to `parallel`
- Specifies the number of threads started
- Example, starting parallel region with 3 threads:

  – C

```c
int nthread=3;
#pragma omp parallel num_threads(nthread)
```

  – Fortran

```fortran
integer nthread=3
!$omp parallel num_threads(nthread)
```

Flushing and barriers

KEEPING MEMORY CONSISTENT

OpenMP: relaxed memory model

- Thread allowed to have "*own temporary view*" of memory
  - not required to be consistent with memory
  - E.g. data in registers or cache, invisible to other thread
- This is a "may be" for the hardware
  - The programmer has to assume it is (portability)
- Scope for data races:
  - Memory modified by other thread not in temp. view
  - Own changes not visible to other threads

Ensuring memory consistency: `flush`

- Use `flush` to ensure memory consistency:

- Modifications in temporary view written to memory-system
  - Guarantied to be visible to other threads

- Temporary view get discarded
  - Next access needs to read from memory subsystem
  - Ensures modifications from other threads are “known”

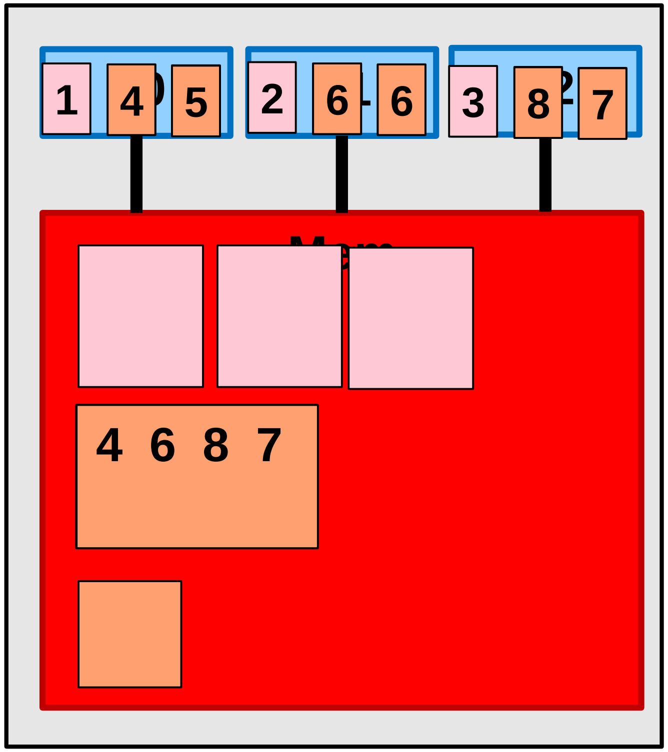
- No reordering of memory access and `flush`

What can happen without at a `flush`?

```
integer :: i
integer, dimension(4) :: b
b = (/ 3,4,5,6 /)

!$OMP parallel &
!$OMP  shared(b), private(i)
 i=get_omp_thread_num() + 1
 b(i) = b(i) + i

 b(i+1) = b(i+1) + 1
!$OMP end parallel
```

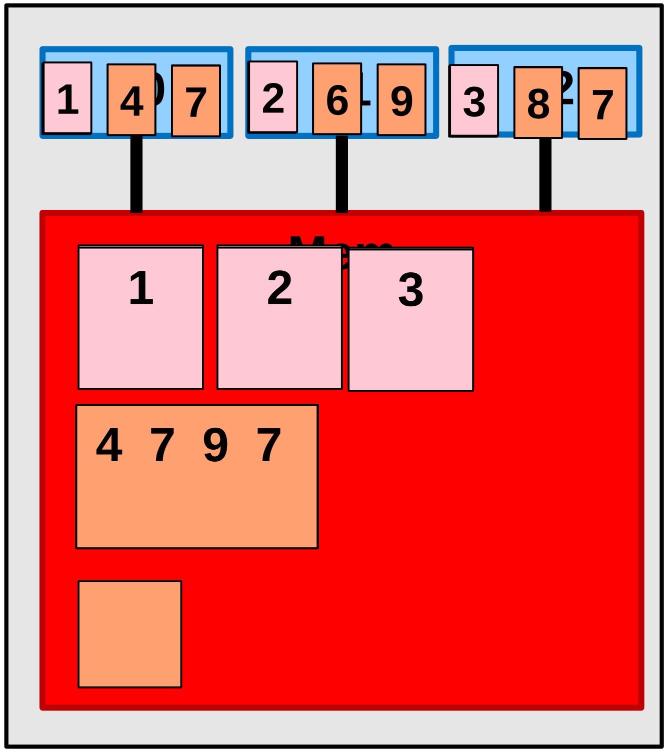


Not what we want!!!

How a `flush` implied in a barrier fixes matters

```
integer :: i
integer, dimension(4) :: b
b = (/ 3,4,5,6 /)

!$OMP parallel &
!$OMP  shared(b), private(i)
 i=get_omp_thread_num() + 1
 b(i) = b(i) + i
 !$OMP barrier
 b(i+1) = b(i+1) + 1
!$OMP end parallel
```



Sequence required for data to be visible on other thread

1. First thread writes
2. First thread `flush`
   Change into Memory-system
3. Second thread `flush`
   Discard local temp. view
4. Second thread reads

- A flush doesn't push
- You can issue a flush by

  `!$OMP flush`
- Fixing data race typically also requires **synchronisation**
- Implied flushes are often sufficient

Implicit barriers and data flushes

- At `barrier` (flush)
- Start & end: `parallel` (barrier & flush)
- Start & end: `critical` (flush) and `ordered` (flush)
- End: loop (`for/do`), `single`, `workshare` and `sections` construct (barrier & flush)
  - No barrier or flush at the start!
- Various locking operations
- Start & end of `atomic` flushes “protected” variable
  - use `seq_cst` on `atomic` to include “global” flush

- **No** barrier or flush associated with `master`

Memory reorder – out-of-order execution

Consider:

- Thread modifies a data-structure `A`
- Sets a shared variable to 1 to signal other threads

Problems:

- No guarantee that `A(5)` is in memory
- No guarantee that `A` is actually set:

  Optimising compiler might move
  `matrix_set = 1`

```
...

A(5) = 3.0

!$omp atomic write
matrix_set = 1

...
```

Fix issue by using a `flush`

The flush

- ensures the modified `A` is in memory
- prohibits reordering of memory accesses

```
 ...

 A(5) = 3.0

!omp flush
!omp atomic write
 matrix_set = 1

 ...
```

Clause `nowait`

- Barriers have performance implications
- Implied barrier of construct may not be required for correctness of code
- Specifying `nowait` on:
  - the construct in C
  - the end construct directive in Fortran

  can suppress the implied barrier incl. flush

Example: Tensor product

```
#pragma omp parallel shared(a,b,t,n,m)
{
  #pragma omp for nowait
  for (int i=0; i<n; i++)
    a[i] = funcA(i);          // no barrier needed!
  #pragma omp for
  for (int j=0; j<m; i++)
    b[j] = funcB(j);          // barrier needed!
  #pragma omp for
  for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
      t[i][j] = a[i]*b[j]; // bad access to b!
}
```

Fortran-example: Adding vectors

```
!$omp parallel shared(a,b,t,n)
!$omp do
   do i=1, n
      a(i) = sin(real(i))
!$omp end do nowait              !! no barrier here!
!$omp do
   do j=1, n
      b(j) = cos(real(j))   !! barrier here!
!$omp do
    do i=1, n
      t(i) = a(i) + b(i)
!$omp end parallel
```

- **Rem:** Demo code - Single loop would help performance

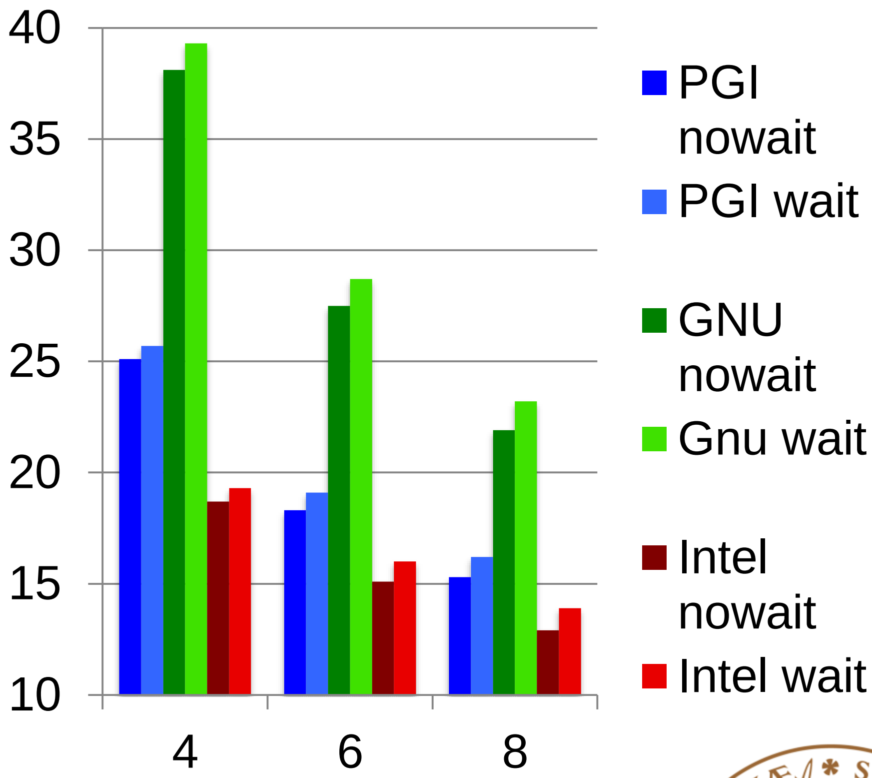
C-Example: Adding vectors

```
#pragma omp parallel shared(a,b,t,n)
  {
#pragma omp for nowait
    for (int i=0; i<n; i++)
      a[i] = sin((double)i);  // no barrier here!
#pragma omp for
    for (int j=0; j<n; j++)
      b[j] = cos((double)j);  // barrier needed!
#pragma omp for
    for (int i=0; i<n; i++)
      t[i] = a[i] + b[i];
  }
```

- **Rem:** Demo code - Single loop would help performance

Effect of `nowait` for "Adding vectors" example

- Dual socket, quadcore Intel Xeon E5520 (2.26 GHz)
- Compilers
  - PGI 10.9
  - GCC 4.4
  - Intel 12.0
- Problem size: n=1000
- Time for code block in μs
- Measured for 4, 6 and 8 threads
- Saving from `nowait` between: 0.6 and 1.3 μs



Speciality of static schedule

- Specifying a `static` schedule
  - Same iteration count
  - Same chunk size (or default)
  - Loops bind to same parallel region

- Save to assume same thread works same iteration in all loops

- Can have `nowait` even with dependency

Fortran: Adding vectors, static schedule

```
!$omp parallel shared(a,b,t,n)
!$omp do schedule(static)
   do i=1, n
      a(i) = sin(real(i))
!$omp end do nowait               !! no barrier here!
!$omp do schedule(static)
    do j=1, n
      b(j) = cos(real(j))
!$omp end do nowait               !! no barrier here!
!$omp do schedule(static)
    do i=1, n
      t(i) = a(i) + b(i)
!$omp end do nowait               !! no barrier here!
!$omp end parallel
```

C-example: Adding vectors, static vectors

```
#pragma omp parallel shared(a,b,t,n)
  {
#pragma omp for schedule(static) nowait
    for (int i=0; i<n; i++)
      a[i] = sin((double)i);  // no barrier here!
#pragma omp for schedule(static) nowait
    for (int j=0; j<n; j++)
      b[j] = cos((double)j);  // no barrier here!
#pragma omp for schedule(static)
    for (int i=0; i<n; i++)
      t[i] = a[i] + b[i];
  }
```

- **Rem:** The static schedule is crucial!

Orphan directives

Assuming thread safety:

- Calling subroutines and functions **inside** a parallel region is legal
- The called procedures may contain workshareing or synchronisation constructs
- Those directives are called “*orphan*” directives

C example: Orphan directive

```
#pragma omp parallel shared(v,vl) reduction(+:nm)
 {
  vectorinit( v, vl);
  nm = vectornorm(v, vl);
 }
```

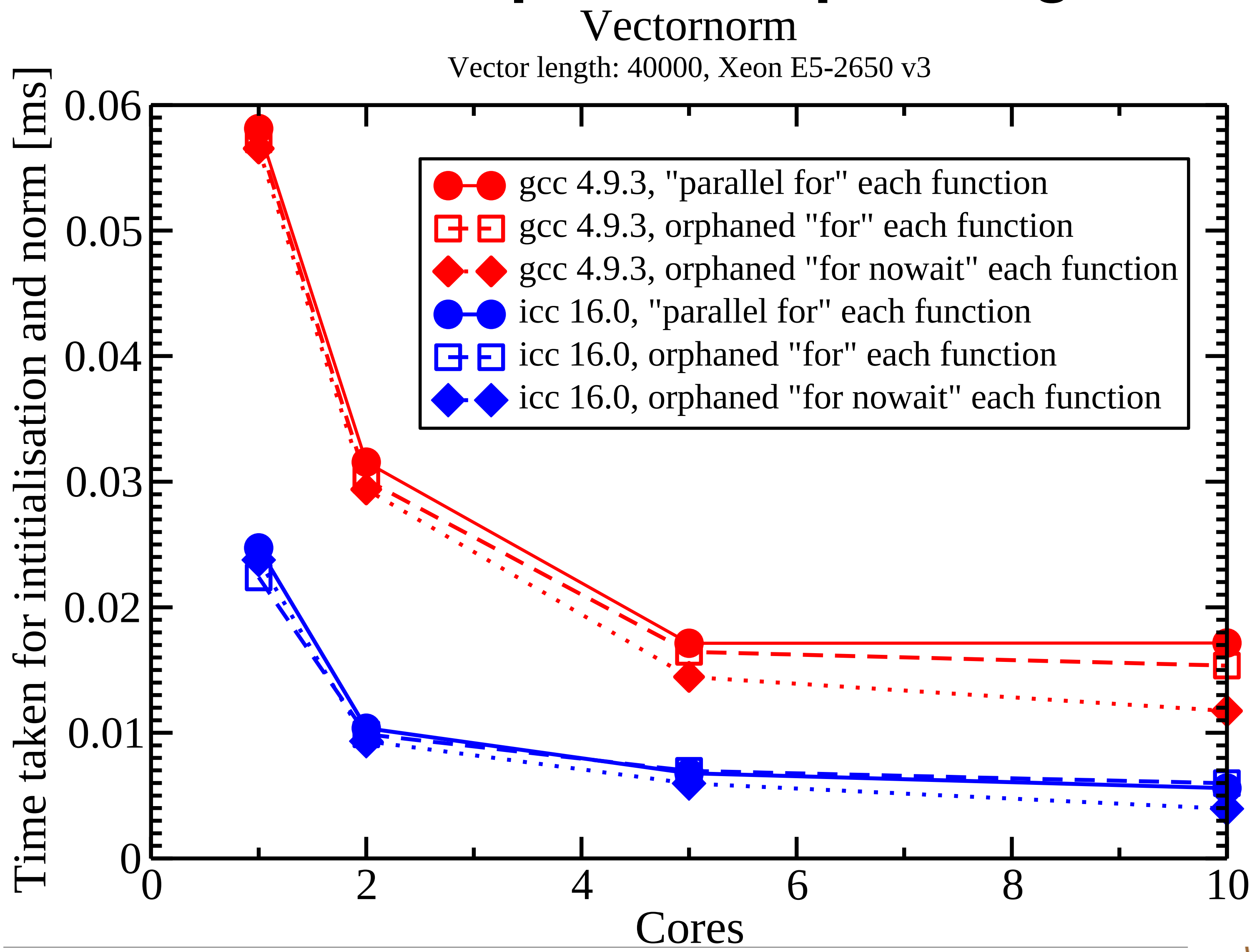
```
void vectorinit( double* vdata, int leng)
{
#pragma omp for
  for ( int i = 0; i < leng; i++)
    { vdata[i] = i;
    }
  return;
}
```

Fortran example: Orphan directive

```
!$omp parallel shared(v,vl) reduction(+:nm)
  call vectorinit(v, vl)
  nm = vectornorm(v, vl)
!$omp end parallel
```

```
subroutine vectorinit(vdata, leng)
  double precision, dimension(leng) :: vdata
  integer :: leng, i
!$omp do
  do i = 1, leng
     vdata(i) = i
  enddo
end subroutine vectorinit
```

Performance impact of orphaning



Discussion of orphan directives

- Reduces need for restructuring code
- Allows for longer parallel regions
  - starting/closing parallel regions is very expensive
- Can create issues:
  - Routine with orphan directive called outside parallel region

Summary

- `single` construct
- `if` clause
- `flush` to memory
- Implicit synchronisation and `nowait`
- Orphan directives