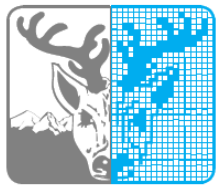


HPC2N, UmU

Worksharing and scheduling

Pedro Ojeda & Joachim Hein

High Performance Computing Center North &
Lund University

Overview

- **Worksharing constructs** allow easy distribution of the work onto the threads
  - Loop construct
    - Easy distributing of loops onto threads
    - Avoiding load imbalance in distributed loop using the `schedule` clause
  - `workshare` construct
    - Parallelsation of Fortran array syntax
  - `sections` construct
    - Distributing independent code blocks
    - Modern alternative: `task`

DISTRIBUTING LOOPS

Introduction for the loop construct

- Distributing large loops: typical target for OpenMP
- As seen in previous lecture: Can be accomplished with the features discussed
- Requires some management code:
  - Number of iterations per thread (may be unequal)
  - Starting index of current thread
  - Final index of current thread
- OpenMP offers "*loop construct*" to ease loop parallelisation
  - Convenience
  - Maintainability

The loop construct

- Distributes the following loop (C/C++/Fortran) onto threads
- The iteration variable automatically `private`
- Determines for you (without management code):
  - Number of iterations per thread
  - Start index of current thread
  - Final index of current thread
- Registers are flushed to memory at exit, unless `nowait`
  - **No** `flush` on entry!
- Offers mechanisms to balance the load for a number of situations

Loop construct in Fortran

- Works on Fortran standard compliant `do`-construct
  - not: `do while`
  - not: `do` without loop control
- `$omp end do`
  not required, optional

```
!$omp parallel &
!$  shared(…) &
!$  private(…)

   !$omp do
   do i=1, N
      loop-body
   end do

!$omp end parallel
```

F90-example: Vector norm Managing loop yourself

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

```
  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)
```

Example: Vector norm
Loop construct

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

```
  norm = 0.0d0;
!$omp parallel default(none) &
!$omp    shared(vect,norm) private( i, lNorm )
  lNorm = 0.0d0

!$omp do
  do i = 1, vleng              //same as serial case
       lNorm = lNorm + vect(i)*vect(i)
  enddo
!$omp atomic update
  norm = norm + lNorm
!$omp end parallel
  norm = sqrt(norm)
```

Loop construct in C

- Limited to “canonical” loops:
- First argument assignment to
  - `int`
  - pointer
  - random-access-iterator-type (C++)
- Second arg. comparison:
  - using `<=, <, >, >=`
- Third arg: increment
  - `i++, ++i, i--, --i`
  - `i+=inc, i-=inc`
  - `i=i+inc, i=inc+i, i=i-inc`
- All bounds, increments: **loop-invariant**

```
#pragma omp parallel \
    shared(…) \
    private(…)

    #pragma omp for
    for (i=0; i<N; i++)
    {
        loop-body
    }
```

Example: Vector norm
Managing loop yourself

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

```
 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;
  }
  norm = sqrt(norm);
```

Example: Vector norm
Loop construct

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

```
 norm = 0.0;
#pragma omp parallel default(none) \
  shared(vect,norm) private( i, lNorm )
  { lNorm = 0.0;
    #pragma omp for
    for (i = 0; i < vleng; i++) //same as serial case

        lNorm += vect[i]*vect[i];
    #pragma omp atomic update
    norm += lNorm;
  }
  norm = sqrt(norm);
```

Parallel loop construct in Fortran

- Shorthand if parallel region being a loop construct

```
!$omp parallel do
 do i, N
   loop-body
 enddo  // parallel region ends here!
```

- No `!$omp end parallel do` required (optional)
- Features of parallel region and normal loop construct apply similarly

Parallel loop construct in C

- Shorthand if parallel region being a loop construct

```
#pragma omp parallel for
 for (int i; i<N; i++)
 {
   loop-body
 } // parallel reg. & loop constr. end here!
```

- Features of parallel region and normal loop construct apply similarly

Loop reordering and data dependency

- In parallel loop: Iterations executed in *different order* from serial code
- Correct result only if current iteration *independent* of previous iteration (data dependency)
- If data dependency
  - modify/change algorithm
  - serialise relevant part of the loop using special OpenMP features (later in course)
  - execute loop serial

- Example for dependency:

```
a[0]=0;
for(i=1; i<N; i++)
  a[i] = a[i-1] + i;
```

- Possible fix (algor. change):

```
for (i=0; i<N; i++)
  a[i]=0.5*i*(i+1);
```

SCHEDULING LOOP ITERATIONS

Work per loop iteration

- So far assumed: Same work for each loop iterations
- Not always the case, e.g.:
  - Summing over triangular area

```
for (i=0; i<N; i++)
   for (j=0; j<i+1; j++)
```

  - Loop body iterates until required accuracy achieve
- Often cause for *load-imbalance*:
  - Some threads finished while others still work

    ➔ Poor performance!!!
- Dealing with such problems is typically easier in shared memory than in distributed memory programming

Schedule clause

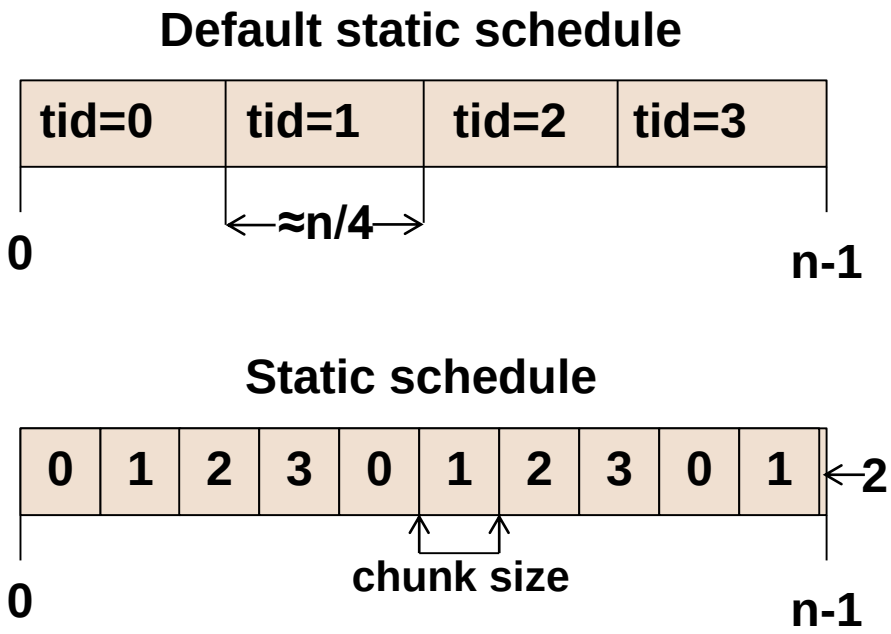
- To help load balance in a loop construct:

```
schedule(kind, [chunk_size])
```

- Default schedule implementation dependent (OpenMP 3.0)
- Choices for `kind`:
  - `static`
  - `dynamic`
  - `guided`
  - `auto`
  - `runtime`

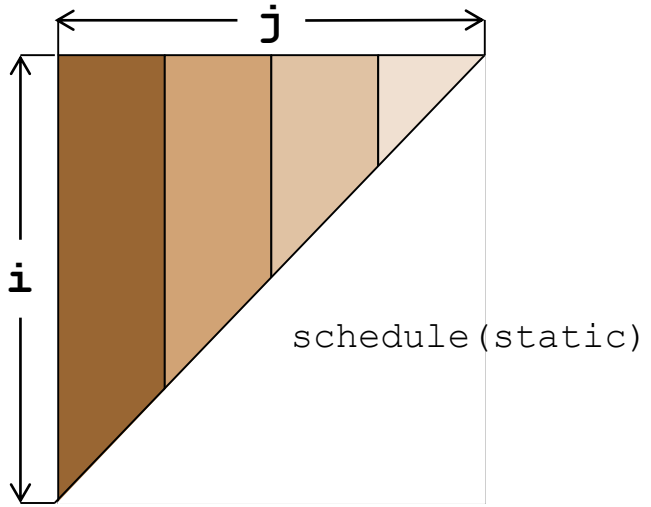
Static scheduling

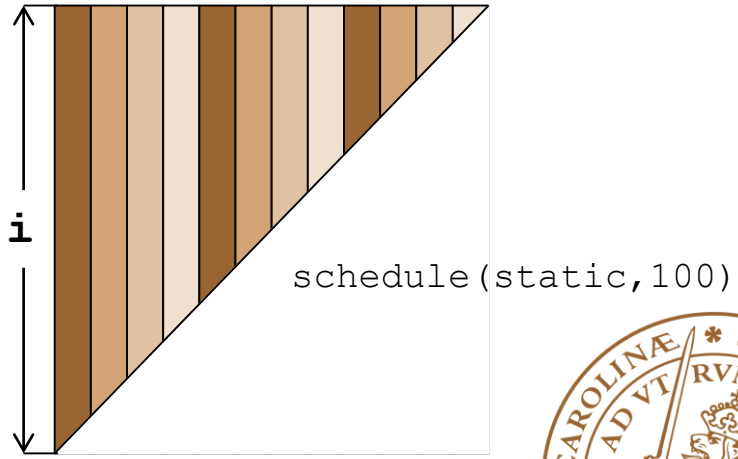
- Divide iteration count into chunks of equal size
  - Last chunk smaller if needed
- Thread assignment: “Round robin”
- Default chunk size: divide iteration count by number of threads
- Least overhead compared to other schedules



Example: Summation over triangular area

```
!$omp parallel do &
!$omp private(i,j)shared(a) &
!$omp schedule(static, 100)
  do j=1, 1200
   do  i=j+1, 1200
     a(i,j) = func(i,j)
     a(j,i) = -a(i,j)
   enddo
  enddo
```





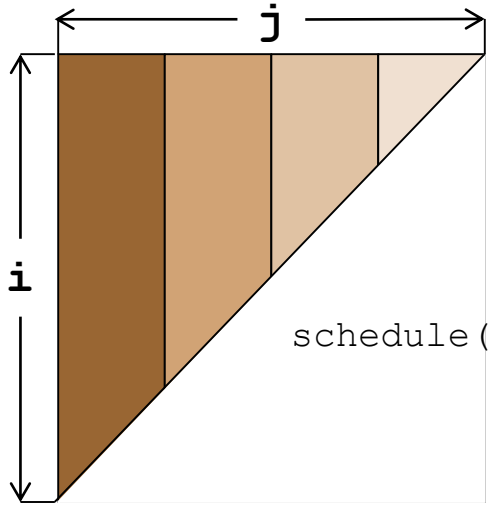
- `default:` max: 7/16 area
- `static,100:` max: 5/16 area
- smaller chunks: better balance
- more chunks: larger overhead

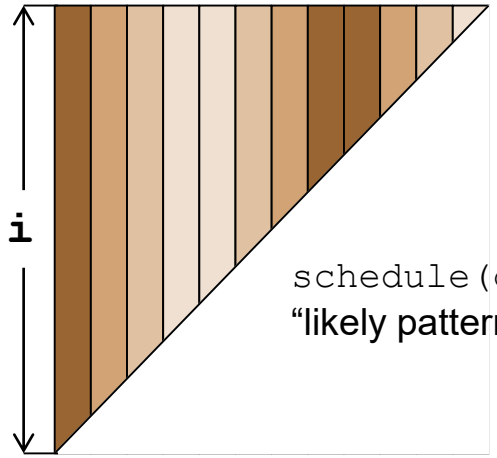
Dynamic scheduling

- Loop split into work packages of `chunk_size` iterations
- Each thread requests new work package once done with present
- Default `chunk_size`: 1 iteration

Example: Summation over triangular area

```
!$omp parallel do &
!$omp private(i,j)shared(a) &
!$omp schedule(dynamic, 100)
  do j=1, 1200
   do  i=j+1, 1200
     a(i,j) = func(i,j)
     a(j,i) = -a(i,j)
   enddo
  enddo
```





- `default:` max: 7/16 area
- `dynamic,100:` max: ≈0.27 area
- better balance than `static`
- larger overhead than `static`

Guided scheduling

- Similar to `dynamic:` Task request new work package once done
- Work package size proportional to
  number of unassigned iterations / number of threads
- But never smaller than `chunk_size` (unless last WP)
- Default `chunk_size = 1`
- Idea: Prevent expensive work packages at the end

Schedules auto and runtime

- For `auto` the implementation decides

- For `runtime` the schedule can be controlled:
  - set `schedule` using functions `omp_set_schedule()`
    - this is OpenMP 3.0
    - Example: omp_set_schedule(omp_sched_static, 10)
  - read environment variable `OMP_SCHEDULE`
    - `export OMP_SCHEDULE="guided,4"`
    - `setenv OMP_SCHEDULE "guided,4"`

- Do not specify `chunk_size` with `auto` or `runtime`

MULTIPLE LOOP PARALLELISATION

Simple example with two nested loops

- 3 basic options to parallelise
- Depends which one is best

```
do j=1, 3


  do i=1, 4

    a(i,j) =
     expensiveFunc(i,j)
  enddo
enddo
```

Simple example with two nested loops

- Distribute the `j`–loop
- Maximally 3 work-packages

```
!$omp parallel do
do j=1, 3


  do i=1, 4

    a(i,j) =
     expensiveFunc(i,j)
  enddo
enddo
```

Simple example with two nested loops

- Distribute `i`-loop
- Now four work packages
- Parallel before `j`-loop
  - better performance
  - needs private `i`
- Starts loop construct 3 times
- (more) cache line conflict when writing to `a`

```
!$omp parallel private(j)
do j=1, 3

!$omp do
 do i=1, 4

    a(i,j) =
     expensiveFunc(i,j)
  enddo
!$omp end do
enddo
!$omp end parallel
```

Simple example with two nested loops

- Use `collapse` clause
  - specify number of loops to collapse
  - **OpenMP 3.0**
- Distribute both loop
  - Creates single loop
  - Schedule as specified
    - default in this case
- Now: 12 work-packages
- (more) cache line conflict when writing to `a`

```
!$omp parallel
!$omp do collapse(2)
do j=1, 3
  do i=1, 4

    a(i,j) =
     expensiveFunc(i,j)

  enddo
enddo
```

Simple example with two nested loops

- 3 basic options to parallelise
- Depends which one is best

```
for (int i=0; i<3; i++)
 {

  for (int j=0; j<4; j++)
   {
    a[i][j] =
     expensiveFunc(i,j);
   }
  }
```

Simple example with two nested loops

- Distribute the `i`–loop
- Maximally 3 work-packages

```
#pragma omp parallel for
for (int i=0; i<3; i++)
 {

  for (int j=0; j<4; j++)
   {
    a[i][j] =
      expensiveFunc(i,j);
   }
  }
```

Simple example with two nested loops

- Distribute `j`-loop
- Now four work packages
- Parallel before `i`-loop
  - better performance
  - needs private `i`
- Starts loop construct 3 times
- (more) cache line conflict when writing to `a`

```
#pragma omp parallel
{
for (int i=0; i<3; i++)
 {
  #pragma omp for
  for (int j=0; j<4; j++)
   {
    a[i][j] =
     expensiveFunc(i,j);
   }
 }
}
```

Simple example with two nested loops

- Use `collapse` clause
  - specify number of loops to collapse
  - **OpenMP 3.0**
- Distribute both loop
  - Creates single loop
  - Schedule as specified
    - default in this case
- Now: 12 work-packages
- (more) cache line conflict when writing to `a`

```
#pragma omp parallel
#pragma omp for collapse(2)
for (int i=0; i<3; i++)
 {
  for (int j=0; j<4; j++)
   {
    a[i][j] =
      expensiveFunc(i,j);
   }
 }
```

WORKSHARE IN FORTRAN

Workshare construct for Fortran

- For Fortran OpenMP provides `workshare`
- This allows distribution of
  - Fortran array syntax

```
a(1:n, 1:m) = b(1:n, 1:m) + c(1:n, 1:m)
```

  - Fortran: `FORALL, WHERE`

Example for workshare

```
!$OMP PARALLEL SHARED(n, a, b, c)
!$OMP WORKSHARE
  b(1:n) = b(1:n) + 1
  c(1:n) = c(1:n) + 2
  a(1:n) = b(1:n) + c(1:n)
!OMP END WORKSHARE
!OMP END PARALLEL
```

- OpenMP ensures there is no data race
  - `b` and `c` ready before assignment to `a`
- Can have user defined functions, if declared `ELEMENTAL`

Scalar assignment in `workshare`

```
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
      INTEGER SHR
!$OMP PARALLEL SHARED(SHR)
!$OMP WORKSHARE
        AA=BB
        SHR=1
        CC=DD * SHR
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

- Legal OpenMP
- Single thread performs scalar assign to `SHR`

Scalar assignment to private in workshare ILLEGAL!!!

```
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
      INTEGER PRI
!$OMP PARALLEL PRIVATE(PRI)
!$OMP WORKSHARE
        AA=BB
        PRI=1
        CC=DD * PRI
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

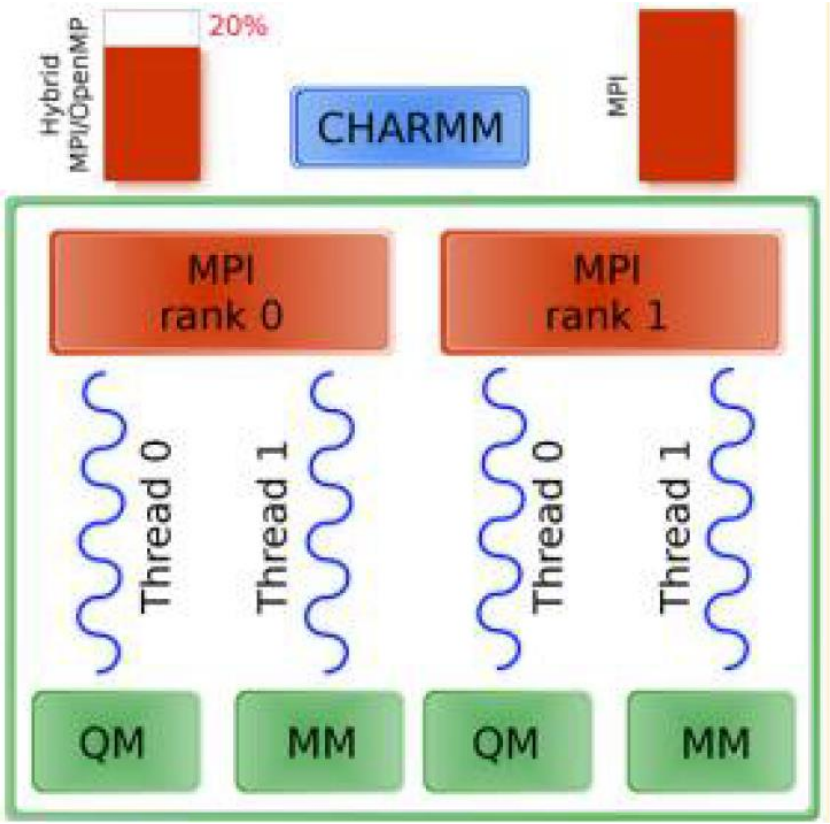
- Single thread performs scalar assign to `PRI`
- Undefined on other threads ☹

SECTIONS

Sections construct

- Allows parallelisation when code blocks can be executed independently
  - Example: Initialisation of multiple data structures
- Easy way to get different tasks to execute different code
- Mismatch: Number of code blocks and number of threads
  - Individual threads might execute multiple code blocks
  - Not every thread gets a code block
- Danger of load imbalance
  - Code blocks have different amount of work
  - Mismatch block and thread number

Sections construct



Acceleration of Semiempirical QM/MM methods, JCTC, 13, 3525-3536 (2017)

Example for use of `sections`

```
#pragma omp parallel shared(a, b, N, M)
 {
  #pragma omp sections
   {
   #pragma omp section
    { for(int i; i < N; i++)
       a[i] = i; }
   #pragma omp section
    { for(int i; i < M; i++)
       b[i] = initBmatrix(i,M); }
   }
 }
```

Alternatively: `parallel sections`

```
#pragma omp parallel sections shared(a, b, N, M)
    {
    #pragma omp section
     { for(int i; i < N; i++)
        a[i] = i; }
    #pragma omp section
     { for(int i; i < M; i++)
        b[i] = initBmatrix(i,M); }
    }
```

Summary

- OpenMP loop construct
  - easy distribution of standard `do/for` loops
  - the `schedule` clause deals with many cases of load imbalance
- OpenMP `workshare` construct
  - distributes Fortran array syntax statements

Summary

- OpenMP loop construct
  - easy distribution of standard `do/for` loops
  - the `schedule` clause deals with many cases of load imbalance
- OpenMP `workshare` construct
  - distributes Fortran array syntax statements
- OpenMP `sections` construct
  - distributes independent code block on different threads