Lecture 4

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables

OpenMP: Work-Sharing Constructs

- The “for” Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
```

By default, there is a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.

Work Sharing Constructs

A motivating example

Sequential code

```
for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
            int id, i, Nthrds, istart, iend;
            id = omp_get_thread_num();
            Nthrds = omp_get_num_threads();
            istart = id * N / Nthrds;
            iend = (id+1) * N / Nthrds;
            for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a work-sharing for-construct

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

OpenMP Work-sharing constructs:

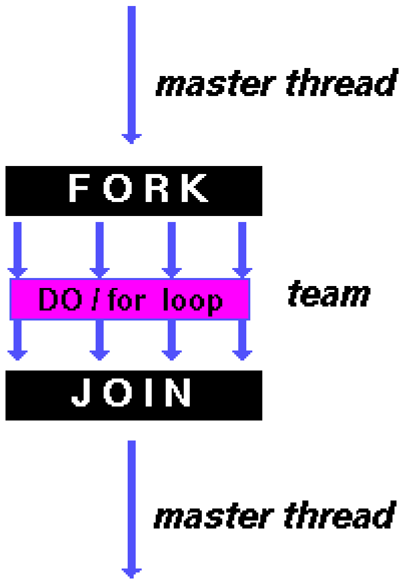
- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.

- Work-sharing constructs do not launch new threads

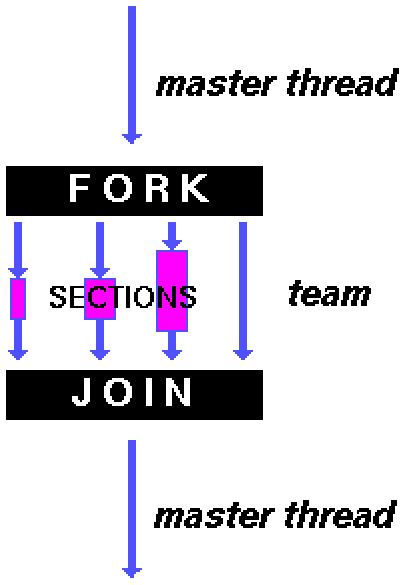
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

Types of Work-Sharing Constructs:

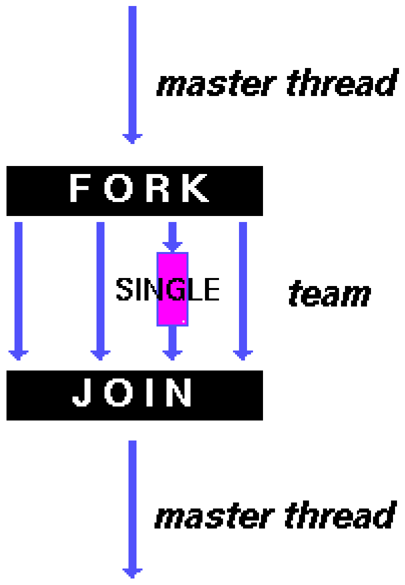
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



SINGLE - serializes a section of code



OpenMP Work-sharing constructs: Notes

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

Work-sharing constructs: Loop construct

- The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

```
#pragma omp parallel
#pragma omp for
      for (I=0;I<N;I++){
            NEAT_STUFF(I);
      }
```

```
!$omp parallel
!$omp do
    do I=1,N
      call NEAT_STUFF(I)
    end do
!$omp end do
!$omp end parallel
```

Simple examples: serial do-loop code

```
const int N = 60000000;
float x[N];

main () {
   int i;

   for (i=0; i<N; i++)
      x[i] = 1/(float)i;

return;
}
```

/home/syam/ces745/openmp/Fortran/do-loop/simple-do/loop-seq.f90

parallel do-loop

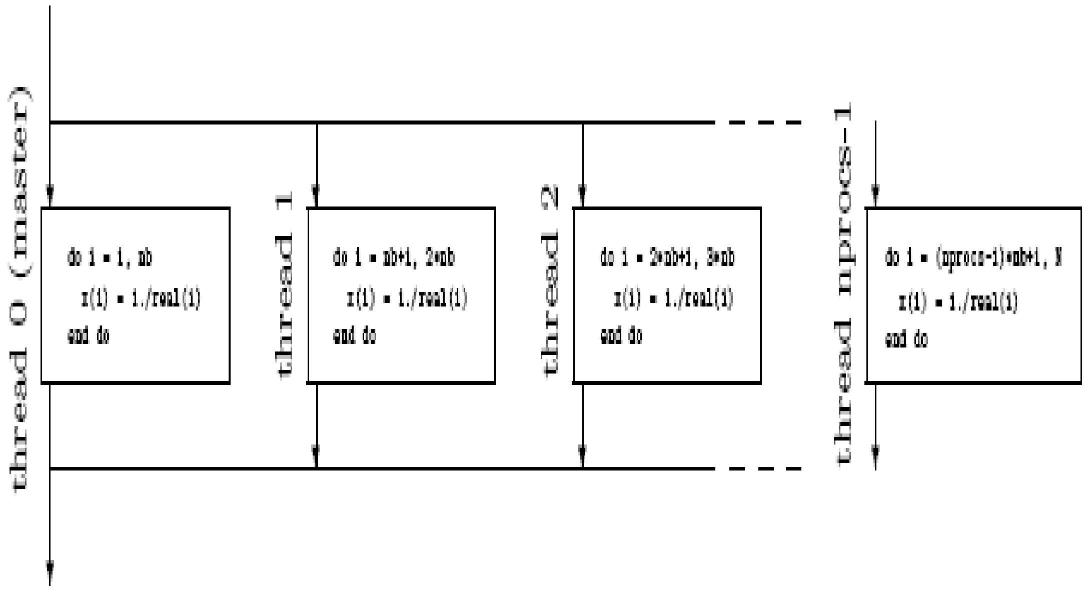
One possible parallel version of the preceding code (distribute the loop to different threads by hard coding).

```
#include <omp.h>
const int N = 60000000;
float x[N];

main () {
int i;
int nprocs, myid, nb, istart, iend;

#pragma omp parallel private(i,myid,istart,iend)
  {
   nprocs = omp_get_num_threads();
   myid = omp_get_thread_num();
   nb = N/nprocs;
   istart = myid*nb;
   if (myid != nprocs-1)
       iend = (myid + 1)*nb;
   else
       iend = N;

   for (i=istart; i<iend; i++)
      x[i] = 1/(float)i;
  }
}
```



/home/syam/ces745/openmp/Fortran/do-loop/simple-do/loop-par-reg.f90

For directive

Instead of hard-coding, we can simply use OpenMP loop directive to achieve the same goal.

```
#include <omp.h>
const int N = 60000000;
float x[N];

main () {
int i;
#pragma omp parallel
  #pragma omp for
     for (i=0; i<N; i++)
        x[i] = 1/(float)i;

return;
}
```

/home/syam/ces745/openmp/Fortran/do-loop/simple-do/loop-par-do.f90

Parallel do: Combined Directives

```
#include <omp.h>
const int N = 60000000;
float x[N];

main () {
int i;
#pragma omp parallel for
    for (i=0; i<N; i++)
       x[i] = 1/(float)i;

return;
}
```

/home/syam/ces745/openmp/Fortran/do-loop/simple-do/loop-par-do-comb.f90

DO/for Format

Fortran

```
!$OMP DO [clause ...]
       SCHEDULE (type [,chunk])
       ORDERED
       PRIVATE (list)
       FIRSTPRIVATE (list)
       LASTPRIVATE (list)
       SHARED (list)
       REDUCTION (operator | intrinsic : list)
  do_loop
!$OMP END DO [ NOWAIT ]
```

C/C++

```
#pragma omp for [clause ...] newline
       schedule (type [,chunk])
       ordered
       private (list)
       firstprivate (list)
       lastprivate (list)
       shared (list)
       reduction (operator: list)
       nowait
for_loop
```

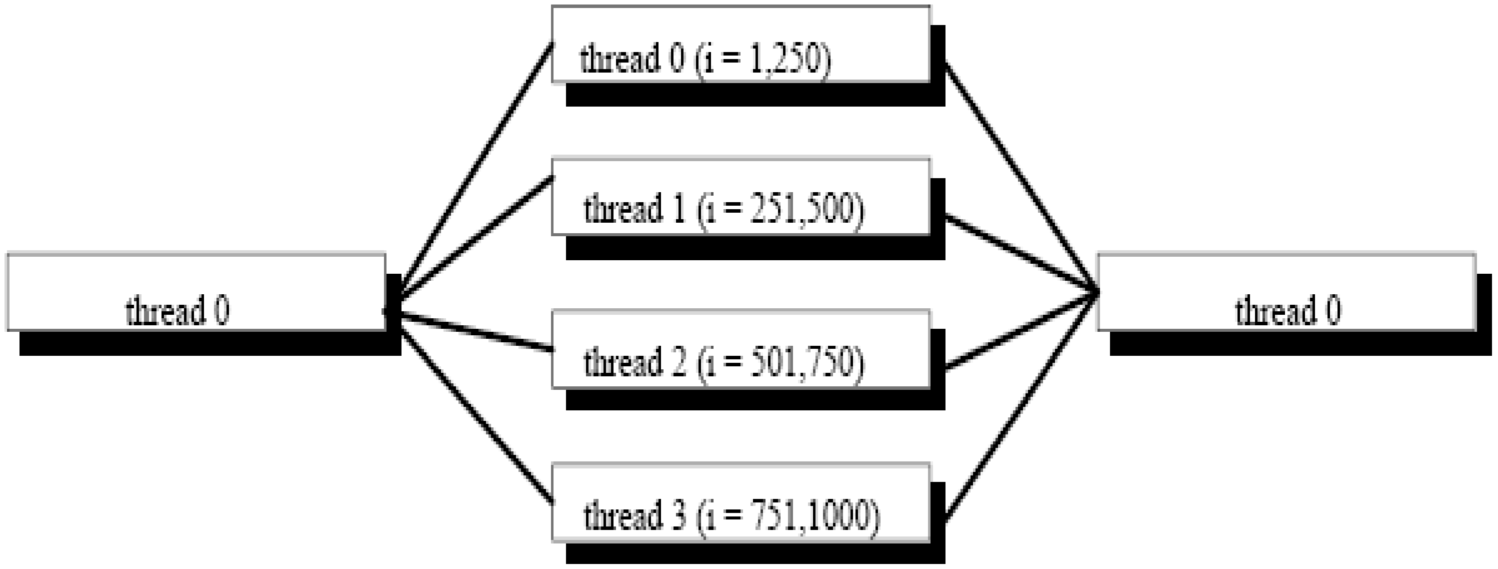
OpenMP For constuct:
The schedule clause

- The schedule clause effects how loop iterations are mapped onto threads
  - ◆schedule(static [,chunk])
    - – Deal-out blocks of iterations of size “chunk” to each thread.
  - ◆schedule(dynamic[,chunk])
    - – Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - ◆schedule(guided[,chunk])
    - – Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - ◆schedule(runtime)
    - – Schedule and chunk size taken from the OMP_SCHEDULE environment variable.

schedule(static)

- **Iterations are divided evenly among threads**

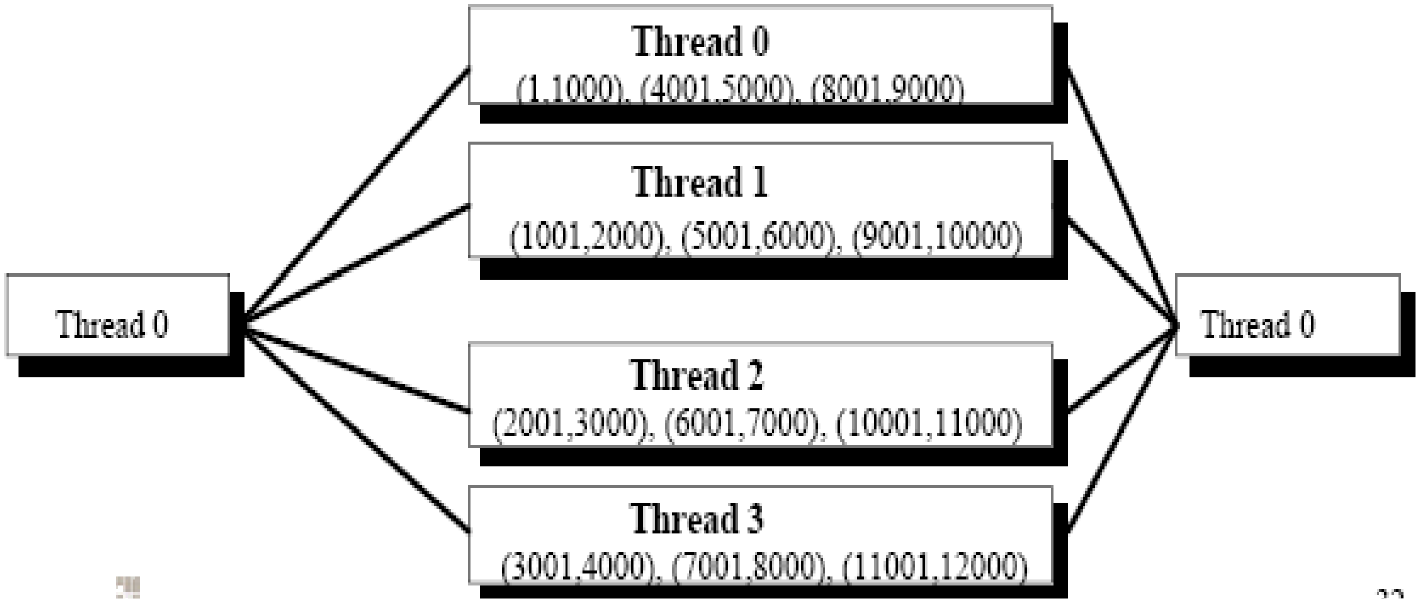
```
#pragma omp for shared(x) private(i) schedule(static)
        for (i=0; i<1000; i++)
          x[i] = a;
```



schedule(static,chunk)

- Divides the work load in to chunk sized parcels
- If there are N threads, each thread does every Nth chunk of work

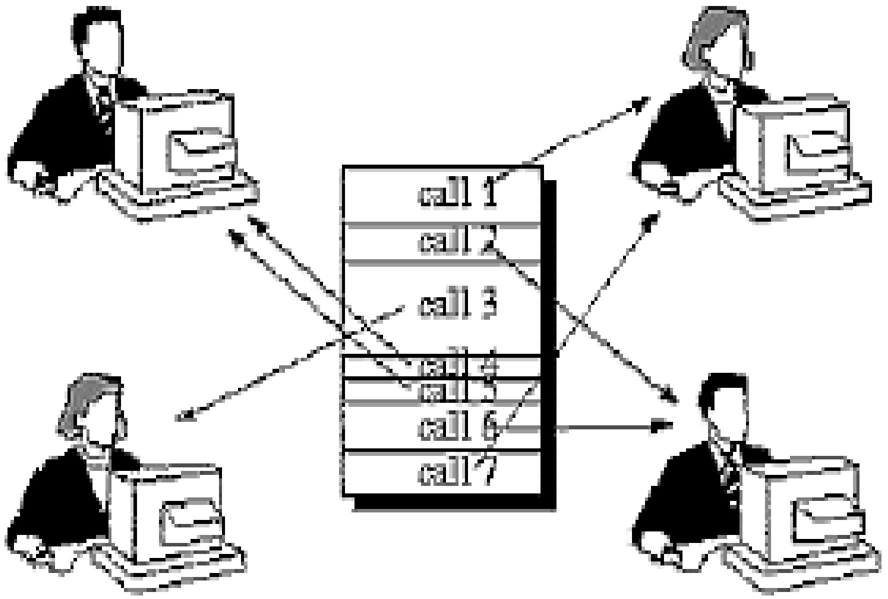
```
#pragma omp for shared(x) private(i) schedule(static,1000)
        for (i=0; i<12000; i++)
           … work …
```



schedule(dynamic,chunk)

- Divides the workload into chunk sized parcels.
- As a thread finishes one chunk, it grabs the next available chunk.
- Default value for chunk is 1.
- More overhead, but potentially better load balancing.

```
# pragma omp for shared(x) private(i) \
   schedule(dynamic,1000)
      for (i=0; i<10000; i++)
         … work …
```



schedule(guided,chunk)

- Like dynamic scheduling, but the chunk size varies dynamically.
- Chunk sizes depend on the number of unassigned iterations.
- The chunk size decreases toward the specified value of chunk.
- Achieves good load balancing with relatively low overhead.
- Insures that no single thread will be stuck with a large number of leftovers while the others take a coffee break.

```
#pragma omp for shared(x) private(i) \
schedule(guided,55)
    for (i=0; i<12000; i++)
        … work …
```

Dynamic schedule

Guided schedule

schedule(runtime)

- Scheduling method is determined at runtime.
- Depends on the value of environment variable **OMP_SCHEDULE**
- This environment variable is checked at runtime, and the method is set accordingly.
- Scheduling method is static by default.
- Chunk size set as (optional) second argument of string expression.
- Useful for experimenting with different scheduling methods without recompiling.

```
$ export OMP_SCHEDULE=static,1000
$ export OMP_SCHEDULE=dynamic
```

DO/for construct: Notes

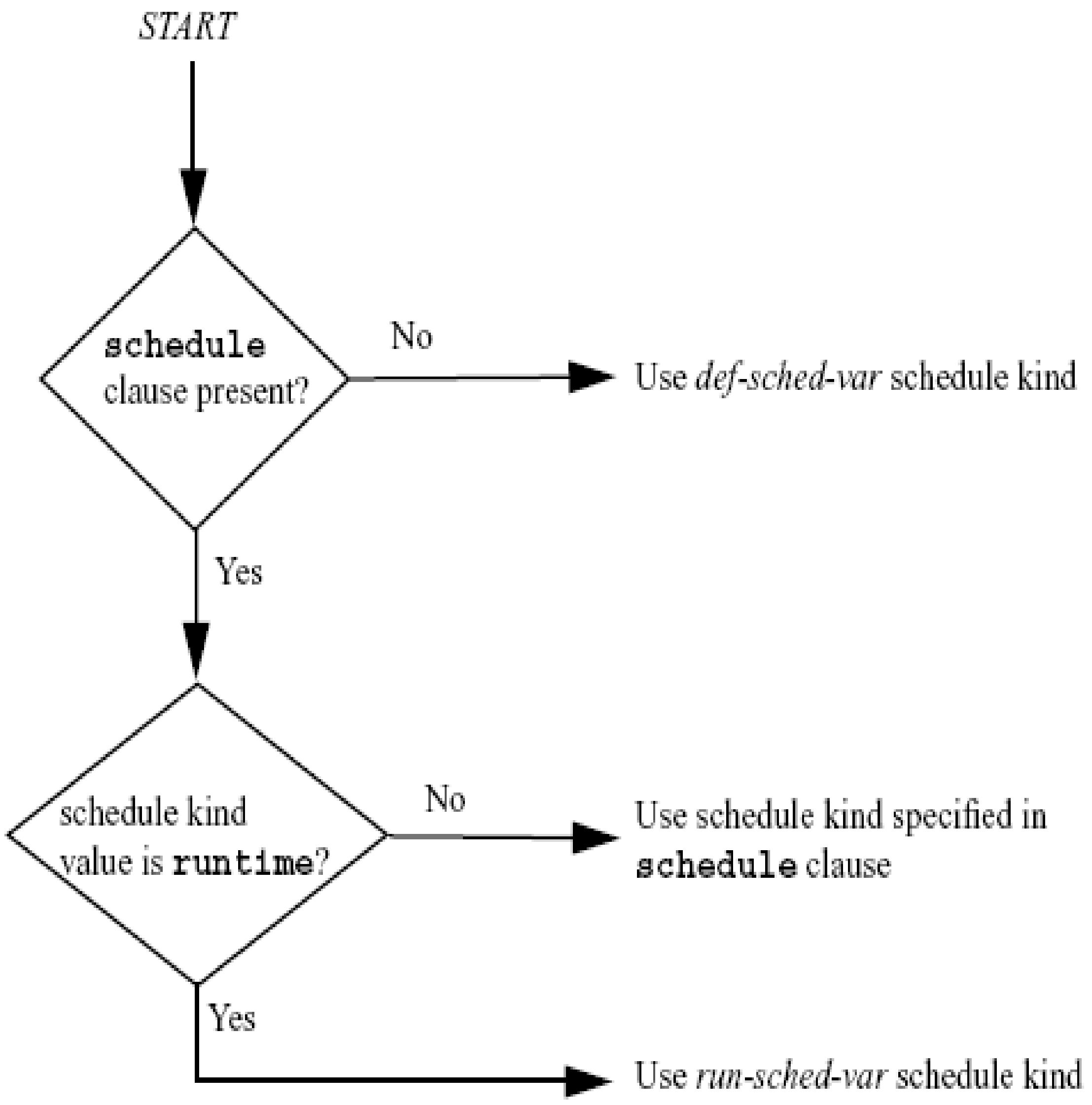
- The DO loop can not be a DO WHILE loop, or a loop without loop control. Also, the loop iteration variable must be an integer and the loop control parameters must be the same for all threads.

- Program correctness must not depend upon which thread executes a particular iteration.

- It is illegal to branch out of a loop associated with a DO/for directive.

- The chunk size must be specified as a loop invariant integer expression, as there is no synchronization during its evaluation by different threads.

- ORDERED and SCHEDULE clauses may appear once each.


START
schedule clause present?
No
Use def-sched-var schedule kind
Yes
schedule kind value is runtime?
No
Use schedule kind specified in schedule clause
Yes
Use run-sched-var schedule kind

Determining the schedule for a work-sharing loop.

Example: Simple vector-add program

- Three Arrays: A, B, C
- Arrays A, B, C, and variable N will be shared by all threads.
- Variable I will be private to each thread; each thread will have its own unique copy.
- The iterations of the loop will be distributed dynamically in CHUNK sized pieces.
- Threads will not synchronize upon completing their individual pieces of work (NOWAIT).

Fortran - DO Directive Example

```
          PROGRAM VEC_ADD_DO

          INTEGER N, CHUNKSIZE, CHUNK, I
          PARAMETER (N=1000)
          PARAMETER (CHUNKSIZE=100)
          REAL A(N), B(N), C(N)

!         Some initializations
          DO I = 1, N
            A(I) = I * 1.0
            B(I) = A(I)
          ENDDO
          CHUNK = CHUNKSIZE

!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)
!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
          DO I = 1, N
            C(I) = A(I) + B(I)
          ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL

          END
```

C / C++ - for Directive Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000

main () {
int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
    {
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
    } /* end of parallel section */

}
```

More on Schedule clause, demo

/home/syam/ces745/openmp/Fortran/do-loop/schedule

do-schedule-static.f
do-schedule-dynamic.f
do-schedule-guide.f
do-schedule-runtime.f

Work-Sharing Constructs: SECTIONS Directive

Purpose:

The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.

Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team. Different sections may be executed by different threads. It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such.

OpenMP: Work-Sharing Constructs

- The Sections work-sharing construct gives a different structured block to each thread.

```
#pragma omp parallel
#pragma omp sections
{
        X_calculation();
#pragma omp section
        y_calculation();
#pragma omp section
        z_calculation();
}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

Format:

Fortran

```
!$OMP SECTIONS [clause ...]
        PRIVATE (list)
        FIRSTPRIVATE (list)
        LASTPRIVATE (list)
        REDUCTION (operator | intrinsic : list)
!$OMP SECTION
        structured_block
!$OMP SECTION
        structured_block
!$OMP END SECTIONS [ NOWAIT ]
```

C/C++

```
#pragma omp sections [clause ...] newline
        private (list)
        firstprivate (list)
        lastprivate (list)
        reduction (operator: list)
        nowait
{
#pragma omp section newline
     structured_block
#pragma omp section newline
    structured_block
}
```

Clauses:

1. There is an implied barrier at the end of a SECTIONS directive, unless the NOWAIT/nowait clause is used.

2. Clauses are described in detail later, in the Data Scope Attribute section.

Restrictions:

1. It is illegal to branch into or out of section blocks.

2. SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive

Examples: 3 loops

Serial code with three independent tasks, namely, three for loops, each operating on a different array using different loop counters and temporary scalar variables.

```
 const int NX = 10000000;
 const int NY = 20000000;
 const int NZ = 30000000;
 float x[NX];
 float y[NY];
 float z[NZ];

main () {
   int i, j, k;
   float ri, rj, rk;
   printf("Start\n");
   for (i=0; i<NX; i++) {
        ri = (float)i;
       x[i] = atan(ri)/ri;
   }

   for (j=0; j<NY; j++) {
        rj = (float)j;
        y[j] = cos(rj)/rj;
   }

   for (k=0; k<NZ; k++) {
        rk = (float)k;
       z[k] = log10(rk)/rk;
   }
   printf ("End\n");
}
```

/home/syam/ces745/openmp/Fortran/do-loop/section

Examples: 3-loops

One possible parallel version of the preceding code (distribute the loop to different threads by hard coding).

```
#include <omp.h>
const int NX = 10000000;
const int NY = 20000000;
const int NZ = 30000000;
float x[NX];
float y[NY];
float z[NZ];

main () {
   int i, j, k;
   float ri, rj, rk;
   printf("Start\n");
   #pragma omp parallel {
   switch (omp_get_thread_num()) {
     case 0:
     for (i=0; i<NX; i++) {
          ri = (float)i;
         x[i] = atan(ri)/ri;}
     break;

     case 1:
     for (j=0; j<NY; j++) {
          rj = (float)j;
          y[j] = cos(rj)/rj;}
     break;

     case 2:
     for (k=0; k<NZ; k++) {
          rk = (float)k;
         z[k] = log10(rk)/rk;}
     }
   }
   printf ("End\n");
}
```

Examples: 3-loops

Instead of hard-coding, we can use OpenMP task sharing directive (section) to achieve the same goal.

```
#include <omp.h>
const int NX = 10000000;
const int NY = 20000000;
const int NZ = 30000000;
float x[NX];
float y[NY];
float z[NZ];

main () {
   int i, j, k;
   float ri, rj, rk;
   printf("Start\n");
   #pragma omp parallel
   {
     #pragma omp sections
     {
        #pragma omp section
           for (i=0; i<NX; i++) {
                ri = (float)i;
               x[i] = atan(ri)/ri;}

        #pragma omp section
           for (j=0; j<NY; j++) {
                rj = (float)j;
                y[j] = cos(rj)/rj;}

        #pragma omp section
           for (k=0; k<NZ; k++) {
                rk = (float)k;
               z[k] = log10(rk)/rk;}
     }
   }
   printf ("End\n");
}
```

More examples:

Combined Directives

```
#include <omp.h>
const int NX = 10000000;
const int NY = 20000000;
const int NZ = 30000000;
float x[NX];
float y[NY];
float z[NZ];

main () {
   int i, j, k;
   float ri, rj, rk;
   printf("Start\n");
   #pragma omp parallel sections
   {
        #pragma omp section
          for (i=0; i<NX; i++) {
               ri = (float)i;
              x[i] = atan(ri)/ri;}

        #pragma omp section
          for (j=0; j<NY; j++) {
               rj = (float)j;
               y[j] = cos(rj)/rj;}

        #pragma omp section
          for (k=0; k<NZ; k++) {
               rk = (float)k;
              z[k] = log10(rk)/rk;}
   }
   printf ("End\n");
}
```

Example: Vector-add

Fortran: vector-add

```
      PROGRAM VEC_ADD_SECTIONS
      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N)
!     Some initializations
      DO I = 1, N
         A(I) = I * 1.0
         B(I) = A(I)
      ENDDO
```

```
!$OMP PARALLEL SHARED(A,B,C)
          PRIVATE(I)
!$OMP SECTIONS
!$OMP SECTION
          DO I = 1, N/2
             C(I) = A(I) + B(I)
          ENDDO
!$OMP SECTION
          DO I = 1+N/2, N
             C(I) = A(I) + B(I)
          ENDDO
!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL
          END
```

- The first n/2 iterations of the DO loop will be distributed to the first thread, and the rest will be distributed to the second thread
- When each thread finishes its block of iterations, it proceeds with whatever code comes next (NOWAIT)

C/C++: vector-add

```
#include <omp.h>
#define N 1000

main () {
int i; float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
     a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)
{
     #pragma omp sections nowait
     {
          #pragma omp section
          for (i=0; i < N/2; i++)
                c[i] = a[i] + b[i];
          #pragma omp section
           for (i=N/2; i < N; i++)
                c[i] = a[i] + b[i];
       } /* end of sections */
} /* end of parallel section */
}
```

Questions:

- What happens if the number of threads and the number of SECTIONs are different? More threads than SECTIONs? Fewer threads than SECTIONs?

Answer: If there are more threads than sections, some threads will not execute a section and some will. If there are more sections than threads, the implementation defines how the extra sections are executed.

- Which thread executes which SECTION?

Answer: It is up to the implementation to decide which threads will execute a section and which threads will not, and it can vary from execution to execution

Work-Sharing Constructs: SINGLE Directive

Purpose:

The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.

May be useful when dealing with sections of code that are not thread safe (such as I/O)

OpenMP Work Sharing Constructs - single

- Ensures that a code block is executed by only one thread in a parallel region.
- The thread that reaches the single directive first is the one that executes the single block.
- Equivalent to a sections directive with a single section - but a more descriptive syntax.
- All threads in the parallel region must encounter the single directive.
- Unless nowait is specified, all uninvolved threads wait at the end of the single block

```
#pragma omp parallel private(i) shared(a)
{
#pragma omp for
        for (i=0; i<n; i++) {
        …work on a(i) …
        }

#pragma omp single
        … process the results of the for loop …

#pragma omp for
        for (i=0; i<n; i++) {
        … more work …
        }
}
```

OpenMP Work Sharing Constructs - single

- Fortran syntax:

c$omp single [clause [clause…]]
structured block
c$omp end single [nowait]

where clause is one of
- private(*list*)
- firstprivate(*list*)

OpenMP Work Sharing Constructs - single

- C syntax:

  #pragma omp single [clause [clause…]]

  structured block

where clause is one of

- private(*list*)
- firstprivate(*list*)
- nowait

OpenMP Work Sharing Constructs - single

Clauses:

- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a NOWAIT/nowait clause is specified.

Restrictions:

- It is illegal to branch into or out of a SINGLE block.

Examples

```
#include <omp.h>

main() {
  printf (“start\n”);
  #pragma omp parallel default(none) private(i)
  {
     #pragma omp for
     for (int i=1; i<=5; i++)
        printf (“%d\n”, i);

     #pragma omp single
     printf (“begin single directive\n”);
     for (int i=1; i<=5; i++)
        printf (“hello %d\n”,i);
  }

printf (“end\n”);
}
```

```
$ ./single-1
 start
 1
 4
 5
 2
 3
 begin single directive
 hello 1
 hello 2
 hello 3
 hello 4
 hello 5
 end
$
```

/home/syam/ces745/openmp/Fortran/do-loop/single

```
#include <omp.h>
main() {
int NTHREADS, TID, TID2;

printf ("Start\n");
#pragma omp parallel private(TID,i)
{

   #pragma omp for
   for (i=1; i<=8; i++) {
      TID = omp_get_thread_num();
      printf ("thread: %d i= %d\n", TID, i);
   }

   #pragma omp single
   printf ("SINGLE - begin\n");
   for (i=1; i<=8; i++) {
      TID2 = omp_get_thread_num();
      printf (“This is from thread = %d\n”, TID2);
      printf (“hello %d”,i);
   }
}

printf ("End\n");
}
```

```
$ ./single-3
 Start
 thread:  0 i =  1
 thread:  1 i =  5
 thread:  1 i =  6
 thread:  1 i =  7
 thread:  1 i =  8
 thread:  0 i =  2
 thread:  0 i =  3
 thread:  0 i =  4
 SINGLE - begin
 This is from thread =  0
 hello 1
 This is from thread =  0
 hello 2
 This is from thread =  0
 hello 3
 This is from thread =  0
 hello 4
 This is from thread =  0
 hello 5
 This is from thread =  0
 hello 6
 This is from thread =  0
 hello 7
 This is from thread =  0
 hello 8
 End
```

OpenMP: Combined Parallel Work-Sharing Constructs

- A short hand notation that combines the Parallel and work-sharing construct.

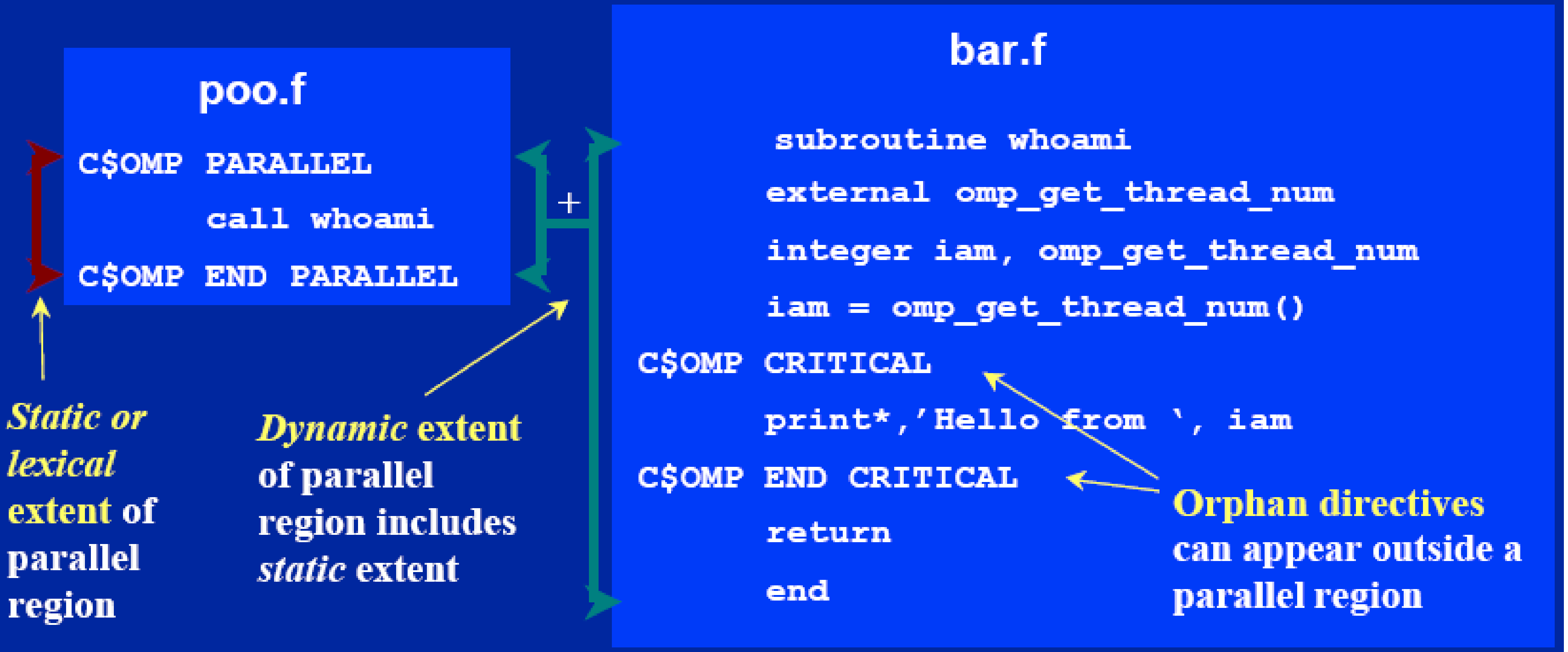
```
#pragma omp parallel for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
```

- There’s also a “parallel sections” construct.

OpenMP:

More details: Scope of OpenMP constructs

OpenMP constructs can span multiple source files.



poo.f

```
C$OMP PARALLEL
      call whoami
C$OMP END PARALLEL
```

bar.f

```
       subroutine whoami
      external omp_get_thread_num
      integer iam, omp_get_thread_num
      iam = omp_get_thread_num()
C$OMP CRITICAL
      print*,'Hello from ', iam
C$OMP END CRITICAL
      return
      end
```

Static or lexical **extent** of parallel region

Dynamic **extent** of parallel region includes *static* extent

Orphan directives can appear outside a parallel region