

Work Replication with Parallel Region

```
#pragma omp parallel
{
  for ( j=0; j<10; j++)
    printf("Hello\n");
}
```

On 5 threads we get
50 print out of hello
since each thread
executes 10 iterations
concurrently with other
10 threads

```
#pragma omp parallel for
{
  for ( j=0; j<10; j++)
    printf("Hello\n");
}
```

Regardless of # of threads we
get 10 print out of hello
since do loop iterations
are executed in parallel
by team of threads

NOWAIT clause : C

```
#pragma omp parallel
{
    #pragma omp for nowait
    for ( j=1; j<n; j++)
        b[j]= (a[j]+a[j-1]) /2.0;

    #pragma omp for
    for ( j=1; j<n; j++)
        c[j] = d[j]/e[j] ;
}
```

Parallel Sections

- So far we have divided the work of one task among threads
- Parallel sections allow us to assign different tasks to different threads
 - Need to make sure that none of the later tasks depends on the results of the earlier ones
- This is helpful where it is difficult or impossible to speedup individual tasks by executing them in parallel
- The code for the entire sequence of tasks or sections begins with a **sections** directive and ends with an **end sections** directive
- The beginning of each section is marked by a **section** directive which is optional for the very first section

Fortran section clause

!\$omp parallel sections [clause..]

[!\$omp section]

code for 1st section

!\$omp section

code for 2nd section

!\$omp section

code for 3rd section

.

.

!\$omp end parallel sections

C/C++ section clause

```
#pragma omp parallel sections [clause...]
{
    [#pragma omp section]
        code for 1st section
    #pragma omp section
        code for 2nd section
    #pragma omp section
        code for 3rd section
    .
    .
}
```

- clause can be private, firstprivate, lastprivate, reduction
- In Fortran the NOWAIT clause goes at the end:
!\$omp end sections [nowait]
- In C/C++ NOWAIT is provided with the omp sections pragma: #pragma omp sections nowait
- Each section is executed once and each thread executes zero or more sections
- A thread may execute more than one section if there are more sections than threads
- It is not possible to determine if one section will be executed before another or if two sections will be executed by the same thread

Assigning work to **single** thread

- Within a parallel region a block of code may be executed just once by **any one** of the threads in the team
 - There is implicit barrier at the end of single (unless `nowait` clause supplied)
 - Clause can be `private` or `firstprivate`

Fortran :

```
!$omp single [clause...]
```

block of code to be executed by just one thread

```
!$omp end single [nowait]
```

C/C++ :

```
#pragma omp single [clause,..... nowait]
```

block of code to be executed by just one thread

single for I/O

- Common use of single is for reading in shared input variables or writing output within a parallel region
- I/O may not be easy to parallelize

omp_get_thread_num, omp_get_num_threads

- Remember OpenMP uses fork/join model of parallelization
- Thread teams are only created within a parallel construct (parallel do/for, parallel)
- **omp_get_thread_num** and **omp_get_num_threads** are only valid within a parallel construct where you have **forked** threads

Synchronization

- Critical - for any block of code
- Barrier – where all threads join
- Other synchronization directives :
 - master
 - ordered

Synchronization : **master** clause

- The **master** directive identifies a structured block of code that is executed by the master thread of the team
- No implicit barrier at the end of master directive
- Fortran :
 !\$omp master
 code block
 !\$omp end master
- C/C++ :
 #pragma omp master
 code block

master example

```
!$ (or #pragma) parallel
  !$ (or #pragma) omp do (or for)
    loop I = 1: n
      calculation
    end loop
  !$ (or #pragma) omp master
    print result (reduction) from above loop
  !$omp end master

  more computation
end parallel loop
```

Synchronization : **ordered** clause

- The structured block following an ordered directive is executed in the order in which iterations would be executed in a sequential loop
- Fortran :
 !\$omp ordered
 code block
 !\$omp end ordered
- C/C++:
 #pragma omp ordered
 code block

ordered example

parallel loop (with parallel do/for) ordered

```
loop I=1 : n
    a[I] = ..calculation...
!$ [OR #pragma] omp ordered
    print a[I]
!$omp end ordered
end parallel loop
```

OpenMP Performance

- Each processor has its own cache in shared memory machine
- Data locality in caches and loop scheduling
- False sharing

Data locality in caches and loop scheduling

- loop $j = 0 : n$
loop $k = 0 : n$
 $a[j][k] = k + 1 + a[j][k]$
- loop $j = 0 : n$
loop $k = 0 : n$
 $a[j][k] = 1./a[j][k]$
- Assume each processor's cache can hold local matrix
- After first loop each processor's cache will have some data (cache line dependent). For next iteration it may or may not get to operate on those data depending on scheduling
- Static scheduling may provide better cache performance than dynamic scheduling

False sharing

- If different processors update stride one elements of an array – this can cause poor cache performance
- Cache line has to be invalidated all the time among all the processors
- Parallel loop with schedule (static,1)
loop j = 1 : n
 $a[j] = a[j] + j$
- Proc1 updates $a[1]$, proc2 updates $a[2]$... etc.
- Cache line needs to be invalidated for each processor – this leads to bad performance

Look up from OpenMP standard

- Threadprivate
!\$omp threadprivate (/cb1/, /cb2/)
#pragma omp threadprivate(list)
- cb1, cb2 are common blocks in fortran, list is a list of named file scope or namespace scope variables in C
- Threadprivate makes named common blocks private to a thread but global within the thread
- Threadprivate makes the named file scope or namespace scope variables (list) private to a thread but file scope visible within the thread

Look up from OpenMP standard

- Atomic directive ensures that specific memory location is updated atomically – provides better optimization than critical due to hardware instructions
- C:
#pragma omp parallel for
for (I = 1; I < n; I++) {
#pragma omp atomic
a[index[I]] = a[index[I]] + 1 }
#pragma omp atomic
a[index[I]] = a[index[I]] + 1 }
- Fortran:
!\$omp parallel do
do I = 1, n
\$omp atomic
y(index(j)) = y(index(j)) + c