ICHEC
Irish Centre for High-End Computing

OpenMP
Loop Level Parallelism

Outline

- Work-sharing Constructs
  - loop
- Combined Constructs
- Reduction Clause
- Schedule Clause
- Synchronisation Constructs
  - master, critical, barrier, atomic, flush, ordered
- Monte Carlo Method
- Timing with OpenMP

Loop Construct

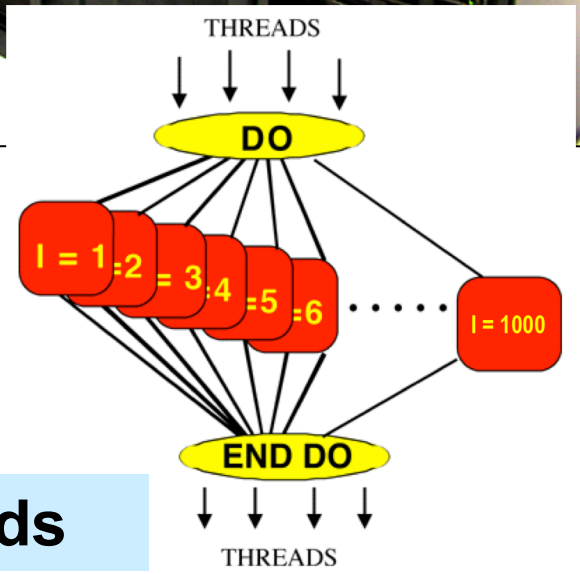
- splits up loop iterations among the threads in a team.
- don't create a team of threads;
- there is an implied barrier at the end (unless nowait is used).

C/C++:

```
#pragma omp parallel [clauses]
{
   #pragma omp for [clauses] [nowait]
   {
   …
   }
}
```

Fortran:

```
!$omp parallel [clauses]
   !$omp do [clauses]
   …
   !$omp end do [nowait]
!$omp end
parallel
```



How is OpenMP Typically Used?

- Find your most time consuming loops.
- Split them up between threads.

Split-up this loop between multiple threads

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

Sequential program

```
void main()
{
   double Res[1000];
   #pragma omp parallel
   #pragma omp for
   for(int i=0;i<1000;i++) {
      do_huge_comp(Res[i]);
   }
}
```

Parallel progra

Performance Considerations:

- Loop dependency
- Cache miss

```
for(i=1; i<n; i++)
  for(j=1; j<n; j++)
    a[i][j]=2*a[i-1][j];
```

```
#pragma omp parallel
#pragma omp for private(i)
for(int j=0; j<n; j++)
  for(i=1; i<n; i++)
    a[i][j]=2*a[i-1][j];
```

Example: (Parallel matrix vector product y=Ax)

C:

```
#pragma omp parallel
default(none), shared(n, A, x, y),
private(i, j, sum, tid)
{
tid=omp_get_thread_num();
#pragma omp for
for (i = 0; i < n; i++) {
   sum=0;
   for (j=0; j<n; j++)
      sum+=A[i*n+j]*x[j];
   y[i]=sum;
   printf("%d: y[%d]=%f\n", tid,
i, y[i]);
}
}
```

Fortran:

```
!$omp parallel default(none),
shared(A, x, y, n), private(i, j,
sum, tid)
tid=omp_get_thread_num()
!$omp do
do i=1,n
   sum=0
   do j=1,n
      sum=sum+A((i-1)*n+j)*x(j)
   enddo
   y(i)=sum
   print *, tid, y(i)
enddo
!$omp end do
!$omp end parallel
```

```
for(i=0; i<n*n; i++){
   A[i]=i;
}
for(i=0; i<n; i++){
   x[i]=1;
}
```

Compile: (Intel)

>icc -openmp hello.c -o a.out

>ifort -openmp hello.f90 -o a.out

Execute:

>export OMP_NUM_THREADS=4

>./a.out

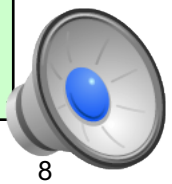
```
Thread 0: y[0]=120.000000
Thread 0: y[1]=376.000000
Thread 0: y[2]=632.000000
Thread 0: y[3]=888.000000
Thread 1: y[4]=1144.000000
Thread 1: y[5]=1400.000000
Thread 1: y[6]=1656.000000
Thread 1: y[7]=1912.000000
Thread 2: y[8]=2168.000000
Thread 2: y[9]=2424.000000
Thread 2: y[10]=2680.000000
Thread 2: y[11]=2936.000000
Thread 3: y[12]=3192.000000
Thread 3: y[13]=3448.000000
Thread 3: y[14]=3704.000000
Thread 3: y[15]=3960.000000
```

Combined Constructs

- The following shortcuts are supported:
  - parallel for / parallel do
  - parallel sections
  - parallel workshare
- Equivalent to a parallel construct followed by a work-sharing construct.

```
#pragma omp parallel for

Same as

#pragma omp parallel
#pragma omp for
```

```
!$omp parallel do

Same as

!$omp parallel
!$omp do
```



Parallel Loop Directives:

C/C++:

```
#pragma omp parallel for [clause[clause]…]

for ( index = first; index <= last ; index++ ){
        body of the loop
}
```

Fortran:

```
!$omp parallel do [clause[clause]…]

do index = first, last [, stride]
   body of the loop
enddo

!$omp end parallel do
```

Reduction Clause

- Performs a reduction operation on the variables in the list
  C: reduction(*operator: list*)
  Fortran: reduction(*operator|intrinsic: list*)

- A local copy of each list variable is created
- Compiler finds the reduction operation and updates local copy
- Combine local copies into a single variable
- for, parallel, sections
- Reduction operations: +, -, *, … (C/C++/Fortran)
  &&, ||, … (C/C++) .AND., .OR., … (Fortran)
  max, min, … (Fortran)

Example: (Parallel matrix vector product y=Ax)

C:

```
for (i = 0; i < n; i++) {
   sum=0;
#pragma omp parallel for
default(shared) private(j, tid)
reduction(+:sum)
   for (j=0; j<n; j++){
      sum=sum+(A[i*n+j]*x[j]);
      if(i==0){
         tid=omp_get_thread_num();
         printf("Tid=%d, A[%d]=%f,
sum=%f\n", tid, i*n+j, A[i*n+j], sum);
      }
   }
   y[i]=sum;
   printf("y[%d]=%f\n", i, y[i]);
}
```

```
>./a.out
Tid=0, A[0]=0.000000, sum=0.000000
Tid=0, A[1]=1.000000, sum=1.000000
Tid=0, A[2]=2.000000, sum=3.000000
Tid=0, A[3]=3.000000, sum=6.000000
Tid=1, A[4]=4.000000, sum=4.000000
Tid=1, A[5]=5.000000, sum=9.000000
Tid=1, A[6]=6.000000, sum=15.000000
Tid=1, A[7]=7.000000, sum=22.000000
Tid=2, A[8]=8.000000, sum=8.000000
Tid=2, A[9]=9.000000, sum=17.000000
Tid=2, A[10]=10.000000, sum=27.000000
Tid=2, A[11]=11.000000, sum=38.000000
Tid=3, A[12]=12.000000, sum=12.000000
Tid=3, A[13]=13.000000, sum=25.000000
Tid=3, A[14]=14.000000, sum=39.000000
Tid=3, A[15]=15.000000, sum=54.000000
```

Schedule Clause

- Describes how iterations of the loop are divided among the team threads

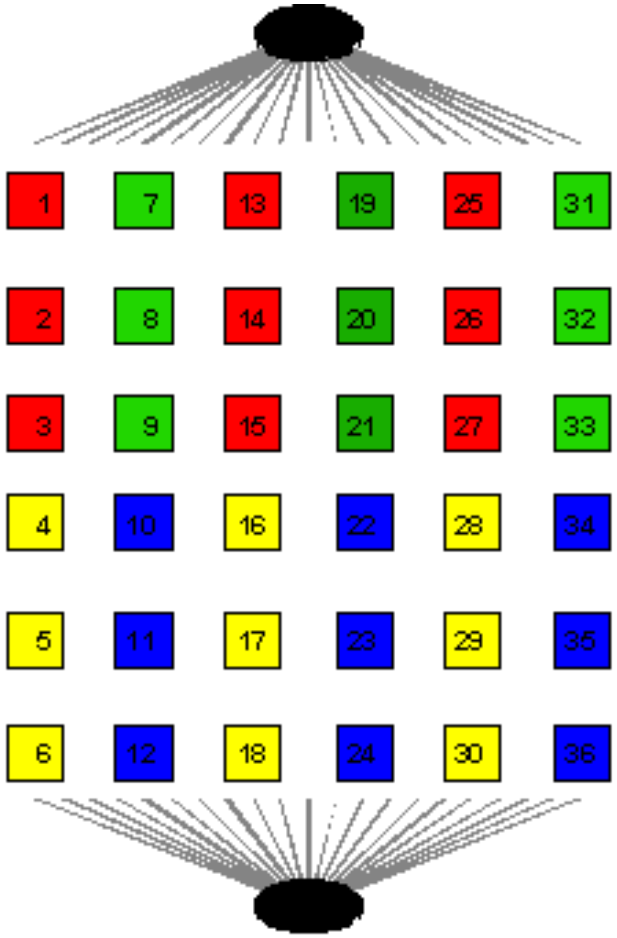
  C: schedule (*type [,chunk]*)

  Fortran: schedule (*type [,chunk]*)

- Types:
  - schedule(static *[,chunk]*)
  - schedule(dynamic *[,chunk]*)
  - schedule(guided *[,chunk]*)
  - schedule(runtime)

- More: ordered clause

SCHEDULE(STATIC):

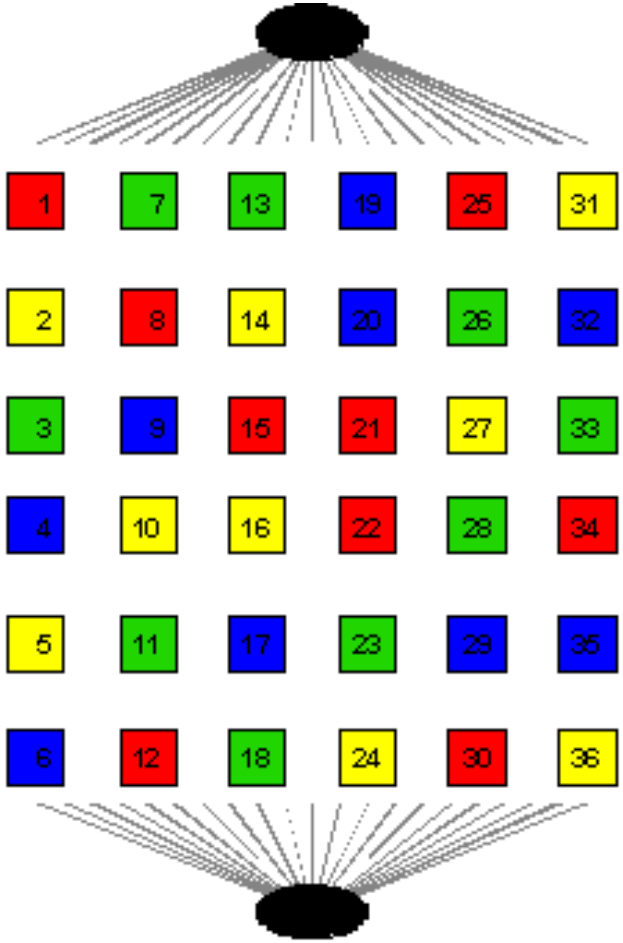


- Iterations are divided evenly among threads
- Divides the work into chunk sized parcels
- If there are N threads, each does every N^{th} chunk of work

Fortran:

```
!$omp parallel do schedule(static,3)

do i = 1, 36
   Work (i)
end do

!$omp end do
```

SCHEDULE(DYNAMIC):

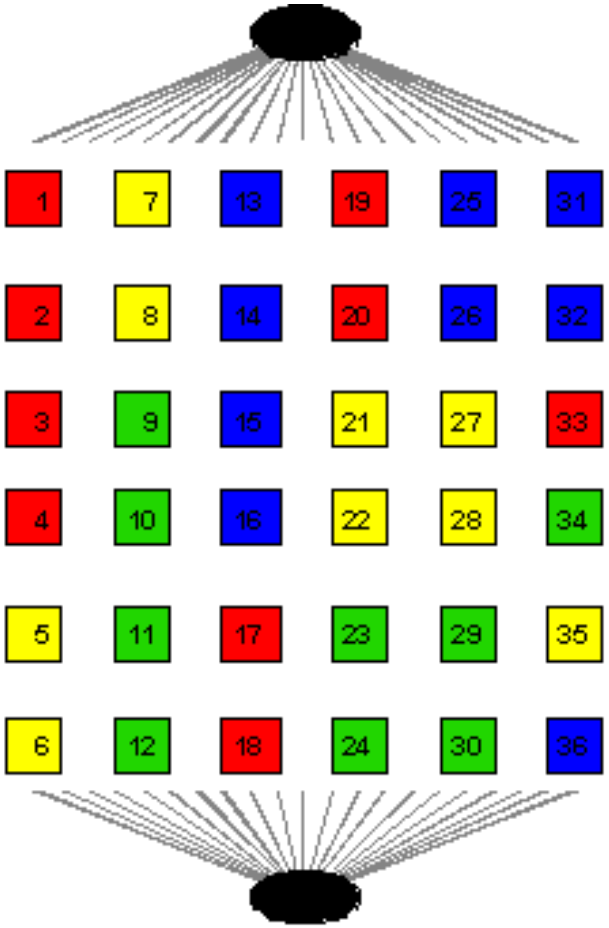


- Divides the workload into chunk sized parcels
- As a thread finishes one chunk, it grabs the next available chunk
- More overhead, but better load balancing

Fortran:

```
!$omp parallel do schedule(dynamic,1)

do i = 1, 36
   Work (i)
end do

!$omp end do
```

SCHEDULE(GUIDED):



- Iterations are divided into chunks such that the size of each successive chunk decreases.
- chunk: the size of the smallest chunk size

Fortran:

```
!$omp parallel do schedule(guided,1)

do i = 1, 36
   Work (i)
end do

!$omp end do
```

Synchronisation Constructs

- To control how the execution of each thread proceeds relative to other team threads
- To protect against data races
- Synchronisation Constructs:
  - master
  - critical
  - barrier
  - atomic
  - flush
  - ordered

Master directive:

- Only master thread can enter the structured block

C/C++:

```
#pragma omp master
   …
```

Fortran:

```
!$omp master
   …
!$omp end
master
```

Critical directive:

- Only one thread at a time can enter a critical section

C/C++:

```
#pragma omp critical [name]
   …
```

Fortran:

```
!$omp critical [name]
   …
!$omp end
critical
```

Barrier directive:

- Threads wait each other until all have reached that barrier

C/C++:

```
#pragma omp barrier
```

Fortran:

```
!$omp barrier
```

Atomic directive:

- Access to a specific memory location atomically
- x binop=expr, x++, --x, … (C/C++)
- x=x op expr, x=expr op x, … (Fortran)

C/C++:

```
#pragma omp atomic
   statement
```

Fortran:

```
!$omp atomic
   statement
```

Flush directive:

- A synchronization point at which thread visible variables are flushed

C/C++:

```
#pragma omp flush (list)
```

Fortran:

```
!$omp flush
(list)
```

Ordered directive:

- Iterations of the enclosed loop will be executed in the same order as if they were executed sequentially

C/C++:

```
#pragma omp ordered
   …
```

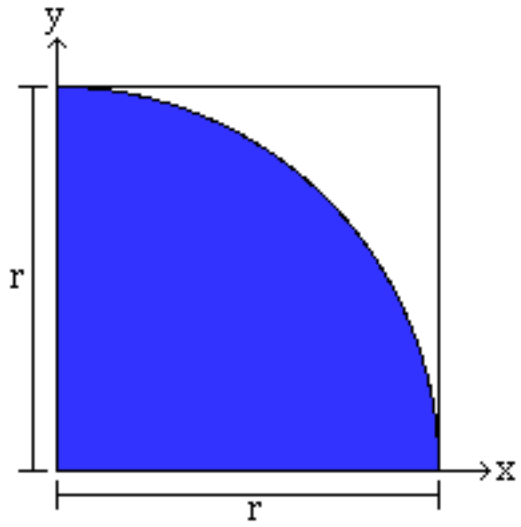
Fortran:

```
!$omp ordered
   …
!$omp end ordered
```

Monte Carlo Method

- Based on the use of random numbers and probability statistics
- Applied in many research areas
  - Financial Analysis
  - Nuclear Physics
  - Population Dynamics
  - Weather Prediction
- Random Number Generation
- Very easy to parallelize

Calculation of pi:



- Area of the shaded circle: $\pi r^2/4$
- Area of the square: r^2
- Probability=$\pi/4$

- r=1 at (0,0)
- distance from the origin: sqrt(x^2+y^2)
- If the distance ≤ 1, then it is a hit.

- Π = 4 x total number of hits/total number of random numbers

C:

```
#pragma omp parallel for default(shared), private(i),
reduction(+:hit), schedule(dynamic, 1)
for (i = 0; i < rnums; i++) {
   double x = (double) rand() / RAND_MAX;
   double y = (double) rand() / RAND_MAX;
   if (x * x + y * y <= 1)
      hit++;
}
printf("The number of hits=%d, Estimate of pi = %f\n", hit, 4.0
* hit / rnums);
```

True value of pi = 3.141593

rnums=10: The number of hits=8, Estimate of pi = 3.200000
rnums=100: The number of hits=78, Estimate of pi = 3.120000
rnums=1000: The number of hits=783, Estimate of pi = 3.132000
rnums=10000: The number of hits=7909, Estimate of pi = 3.163600
rnums=100000: The number of hits=78524, Estimate of pi = 3.140960

C:

```
#pragma omp parallel default(shared), private(thit, i, tid)
{
thit=0;
#pragma omp for schedule(dynamic, 1)
for (i = 0; i < rnums; i++) {
   double x = (double) rand() / RAND_MAX;
   double y = (double) rand() / RAND_MAX;
   if (x * x + y * y <= 1)
      thit++;
}
tid=omp_get_thread_num();
printf("Tid=%d thit=%d\n", tid, thit);
#pragma omp critical
   hit+=thit;
}
```

Tid=0 thit=22484, Tid=3 thit=20419, Tid=2 thit=18034, Tid=1 thit=17728. The number of hits=78665, Estimate of pi = 3.146600

Timing with OpenMP

- omp_get_wtime

  C: double omp_get_wtime(void)

  Fortran: double precision function omp_get_wtime()

- Returns a double-precision floating point value equal to the number of elapsed seconds since some point in the past.

```
double start, end, work;
start = omp_get_wtime();
 … work to be timed …
end = omp_get_wtime();
work=end-start;
```

- omp_get_wtick