Introduction to OpenMP

ICHEC
Irish Centre for High-End Computing

Dr. Prithwish Nandi
Dr. Adam Ralph
UCD, June 2018

Outline

- Introduction to OpenMP
- OpenMP Directives
  - Directive format
  - Parallel Construct
  - Work-Sharing Constructs
  - Synchronisation Constructs
- Performance Considerations
- OpenMP versions and features

What is OpenMP?

- Open Multi-Processing.
- API to explicitly specify multi-threaded, shared-memory parallelism.
- Three primary API components:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- Designed for C, C++ and Fortran.
- Open standard for portable and scalable parallel programming, multi-platform.

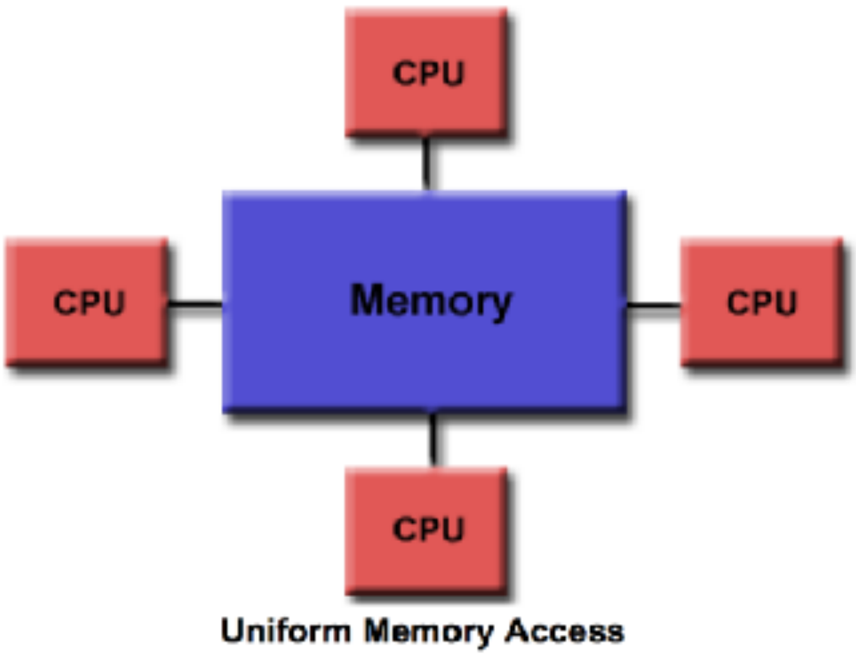
ICHEC

Terminology

- **OpenMP thread:** a lightweight process - an instance of the programme and data.
- **thread team**: a set of threads co-operating on a task.
- **master thread:** the co-ordinating thread.
- **thread-safety**: threads are not interfering with data from other threads.
- **OpenMP directive**: pre-processed OpenMP code.
- **construct**: an OpenMP executable directive.
- **clause**: controls the operation of the directive or data.

OpenMP Programming Model … (1)

- Multi-processor/core systems.
- Shared-memory UMA or NUMA systems.



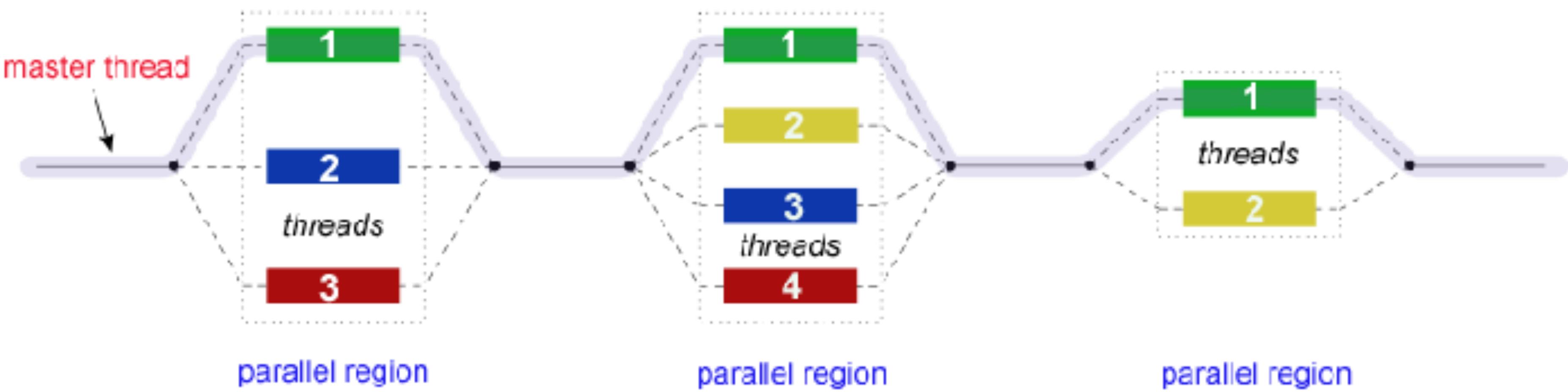
OpenMP Programming Model … (2)

- Thread Based Parallelism
  - Execute a block of code with multiple threads.
  - Thread is the smallest unit of processing scheduled by an OS.
  - Typically, number of threads = number of cores.
    - But, can be decided by the application and/or programmer.

- Explicit Parallelism
  - OpenMP directives, library functions and environment variables explicitly specify what and how to parallelise a block of code.

OpenMP Programming Model … (3)

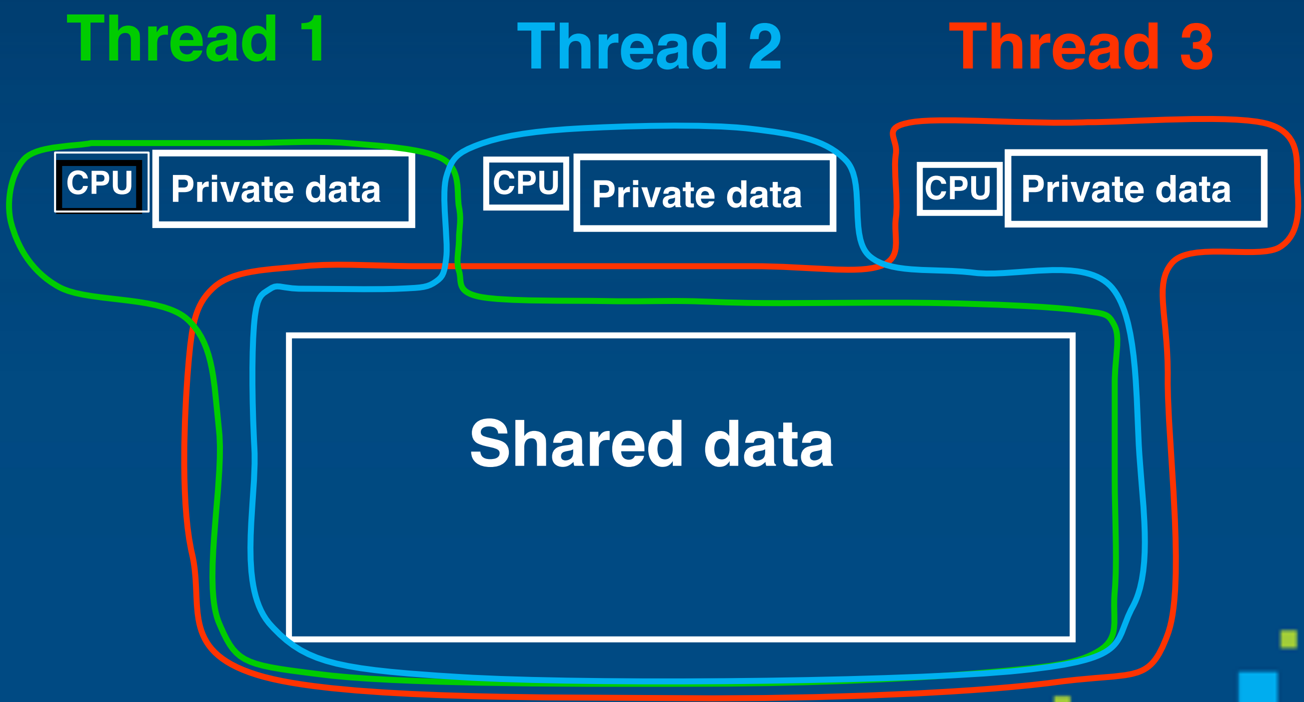
- Fork-Join Model



- Nested parallelism
  - Placement of parallel regions inside parallel regions
- Dynamic threads
  - Runtime environment can dynamically alter number of threads used to execute a parallel region.

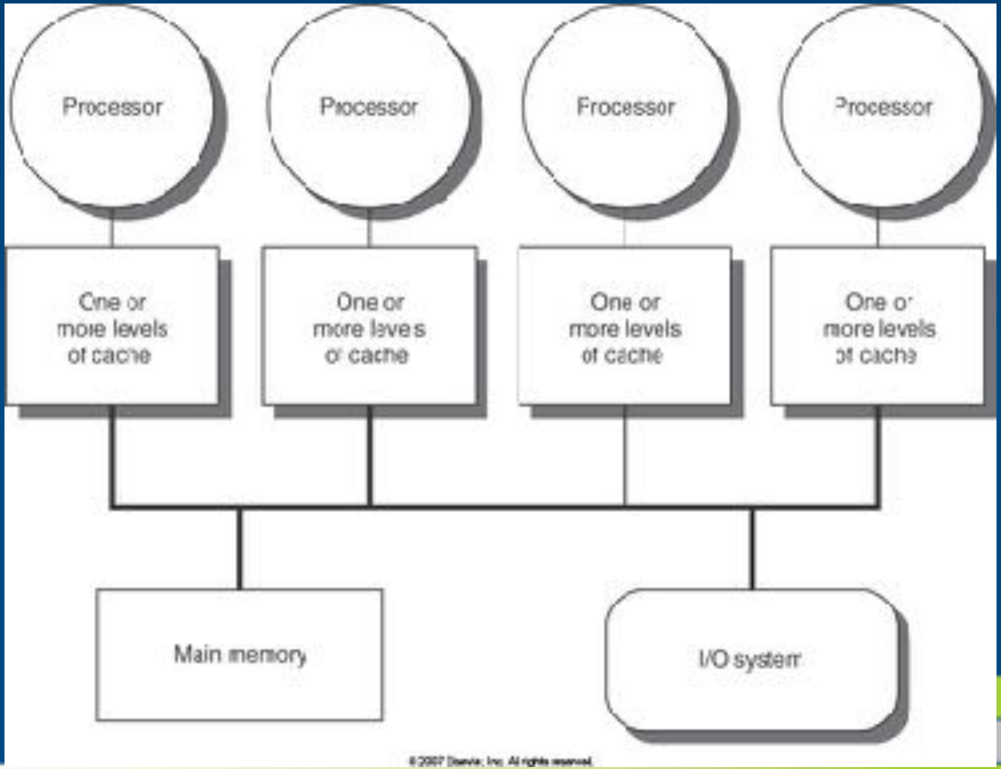
Memory model … (1)

- Threads have access to shared data.
- Threads communicate by sharing data (variables) with other threads.
- Threads have private data.



Memory model … (2)

- Threads allowed to have its own temporary view of memory (cache, registers) to avoid going to memory for every reference to a variable.
- If multiple threads write the same memory unit then a data race occurs.
  - Use synchronization to protect data conflicts.
  - Careless use of synchronization can lead to dead-locks



OpenMP Components

- Compiler Directives
  - Specify a parallel region
  - Divide work among threads
  - Synchronize threads
- Runtime Library Functions
  - Set and query thread-related information
    - number of threads, identifier, team size, …
  - Query if in parallel region
  - Implement and coordinate locks
  - Query timing-related information
    - wall-clock time and resolution
- Environment Variables
  - Specify number of threads
  - Divide work among threads
  - Bind threads to processors

Writing an OpenMP Program

- Decompose the problem into tasks
  - Ideally, these tasks can be worked on independently of the others.
- Map tasks onto "threads of execution" (processors/cores)
- Decide data type. Threads have *shared* and *private* data
  - Shared: used by more than one thread
  - Private: local to each thread
- Write source code using compiler directives and library functions.
- Decide the necessary environment variables.
- Choices may depend on (among many things)
  - The nature of the problem
  - The level of performance needed

Compiling:

	Compiler	Flag
Intel	icc (C) icpc (C++) ifort (Fortran)	-qopenmp -openmp
GNU	gcc (C) g++ (C++) g77/gfortran (Fortran)	-fopenmp

See: http://openmp.org/wp/openmp-compilers/ for the full list.

Outline

- Introduction to OpenMP
- OpenMP Directives
  - Directive format
  - Parallel Construct
  - Work-Sharing Constructs
  - Synchronization Constructs
- Performance Considerations
- OpenMP versions and features

Directive Format

C/C++:
```
#pragma omp directive-name [clause[clause]...]
{
    block of code
}
```

Fortran free form:
```
!$omp directive-name [clause[clause]…]
   block of code
!$omp end directive-name
```

Fortran fixed form:
```
!$omp | c$omp | *$omp directive-name [clause[clause]…]
   block of code
!$omp | c$omp | *$omp end directive-name
```

Outline

- Introduction to OpenMP
- OpenMP Directives
  - Directive format
  - Parallel Construct
  - Work-Sharing Constructs
  - Synchronization Constructs
- Performance Considerations
- OpenMP versions and features

Parallel Construct

- The fundamental construct in OpenMP.
- Creates team of threads
- Every thread executes the same statements inside the parallel region at the end of the parallel region there is an implicit barrier

C/C++:

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

Fortran:

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

Parallel Construct

- Clauses:

```
num_threads (integer-expression)
if (scalar_expression)
```

  - Data Clauses:

```
private (list)
shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
```

How many threads?

The number of threads in a parallel region is determined by:

- Use of `num_threads(n)` clause.
- Setting of the `OMP_NUM_THREADS` environment variable.
- Use of the `omp_set_num_threads(n)` library function.
- The implementation default - usually the number of CPUs/cores on a node

Threads are numbered from 0 (master thread) to n-1 where n=the total number of threads.

Example

```
void main()
{
    double Res[1000];
    #pragma omp parallel num_threads(4)
    {
        block of code
    }
}
```

- Environment Variables
  - To control the execution of parallel program at run-time.
  - csh/tcsh: `setenv OMP_NUM_THREADS n`
  - ksh/sh/bash: `export OMP_NUM_THREADS=n`
    `echo $OMP_NUM_THREADS`

- Runtime Functions
  - To manage the parallel program dynamically.
  - `omp_set_num_threads(n)` - set the desired number of threads
  - `omp_get_num_threads()` - returns the current number of threads
  - `omp_get_thread_num()` - returns the id of this thread
  - `omp_in_parallel()` – returns `.true.` if inside parallel region

C/C++: Add #include<omp.h>

Fortran: Add use omp_lib

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    foo(tid,A);
}
printf("All Done\n");
```

```
double A[1000];

omp_set_num_threads(4);

foo(0,A); foo(1,A); foo(2,A); foo(3,A);

printf("All Done\n");
```

Hello World

C - Serial:

```c
#include<stdio.h>

int main(int argc, char**argv)
{
    printf("Hello world!\n");
}
```

C - Parallel:

```c
#include<stdio.h>
#include<omp.h>
int main(int argc, char**argv)
{
    #pragma omp parallel
    printf("Hello from thread %d out of %d\n", omp_get_thread_num(),
                                                omp_get_num_threads());
}
```

Hello World

Fortran - Serial:

```
program hello
implicit none

print *, 'Hello world!’

end program hello
```

Fortran - Parallel:

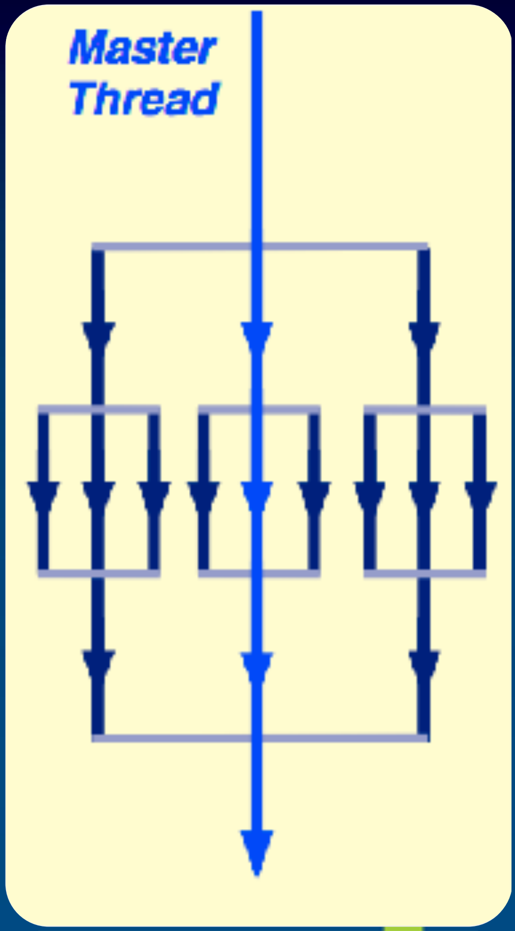
```
program hello
use omp_lib
implicit none

!$omp parallel
print *, 'Hello from thread', omp_get_thread_num(),'out of’, &
       omp_get_num_threads()
!$omp end parallel

end program hello
```

Nested parallel regions:

- If a parallel directive is encountered within another parallel directive, a new team of threads will be created.
- `omp_set_nested()`, `OMP_NESTED`, `omp_get_nested()`
- Num threads affects the new regions
- New threads with one thread unless nested parallelism is enabled
- `num_threads(n)` clause or dynamic threading for different num threads



Dynamic threads:

- Used to create a parallel region with a variable number of threads
- `omp_set_dynamic()`, `OMP_DYNAMIC`, `omp_get_dynamic()`
- OpenMP runtime will decide the number of threads

```
omp_set_dynamic(0);
omp_set_num_threads(10);
#pragma omp parallel
printf("Num threads in non-dynamic region = %d\n",  omp_get_num_threads());

omp_set_dynamic(1);
omp_set_num_threads(10);
#pragma omp parallel
printf("Num threads in dynamic region is = %d\n", omp_get_num_threads());
```

Parallel Construct

- Clauses:

```
num_threads (integer-expression)
if (scalar_expression)
nowait
```

  - Data Clauses:

```
private (list)
shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
```

If Clause:

- Used to make the parallel region directive itself conditional.
- Only execute in parallel if expression is true.

C/C++:

```
#pragma omp parallel if(n>100)
{
    …
}
```

Fortran:

```
!$omp parallel if(n>100)
    ...
!$omp end parallel
```

nowait Clause:

- allows threads that finish earlier to proceed without waiting

```
#pragma omp parallel nowait
{
    …
}
```

```
!$omp parallel
    ...
!$omp end parallel nowait
```

Parallel Construct

- Clauses:

```
num_threads (integer-expression)
if (scalar_expression)
nowait
```

  - Data Clauses:

```
private (list)
shared (list)
default (shared | none)
firstprivate (list)
reduction (operator: list)
copyin (list)
```

Data Clauses

- Used in conjunction with several directives to control the scoping of enclosed variables.
  - `default(shared|none)`: The default scope for all of the variables; private for Fortran
  - `shared(list)`: Variable is shared by all threads in the team. All threads can read or write to that variable.

    C/C++: `#pragma omp parallel default(none) shared(n)`

    Fortran: `!$omp parallel default(none) shared(n)`
  - `private(list)`: Each thread has a private copy of variable. It can only be read or written by its own thread.

    C/C++: `#pragma omp parallel default(shared) private(tid)`

    Fortran: `!$omp parallel default(shared) private(tid)`

Example

C:

```c
#include<stdio.h>
#include<omp.h>
int main()
{
    int tid, nthreads;
    #pragma omp parallel private(tid), shared(nthreads)
    {
        tid = omp_get_thread_num();
        nthreads = omp_get_num_threads();
        printf("Hello from thread %d out of %d\n", tid, nthreads);
    }
}
```

Example

Fortran:

```
program hello
use omp_lib
implicit none
integer tid, nthreads

!$omp parallel private(tid), shared(nthreads)
tid = omp_get_thread_num()
nthreads = omp_get_num_threads()
print*, 'Hello from thread',tid,'out of',nthreads
!$omp end parallel

end program hello
```

- How do we decide which variables should be shared and which private?
  - Loop indices - private
  - Loop temporaries - private
  - Read-only variables - shared
  - Main arrays - shared
- Most variables are shared by default
  - **C/C++:** File scope, static variables
  - **Fortran:** COMMON blocks, SAVE, MODULE variables
  - **Both:** dynamically allocated variables
- Variables declared in parallel region are always private

ICHEC

Additional Data Clauses

- **firstprivate(*list*)**: pre-initialize private vars with value of variable with same name before parallel construct.

- **lastprivate(*list*)**: On exiting the parallel region, this gives private data the value of last iteration (if sequential)

- **threadprivate(*list*)**: Used to make global file scope variables (C/C++) or common blocks (Fortran) private to thread.

- **copyin(*list*)**: Copies a value from master thread to all threadprivate variables of a thread team.

```
j = jstart;
#pragma omp parallel for firstprivate(j)
{
    for(i=1; i<=n; i++)
    {
        j = j + 1;
        a[i] = a[i] + j;
    }
}
```

```
#pragma omp parallel for lastprivate(x)
{
    for(i=1; i<=n; i++)
    {
        x = sin( pi * dx * (float)i );
        a[i] = exp(x);
    }
}
lastx = x;
```

Outline

- Introduction to OpenMP
- OpenMP Directives
  - Directive format
  - Parallel Construct
  - Work-Sharing Constructs
  - Synchronization Constructs
- Performance Considerations
- OpenMP versions and features

Work-Sharing Constructs

- Work-sharing construct divides work among the thread team; what kind of work to be parallelised.
- Specify inside the parallel region.
- No new threads created. Construct must be encountered by all threads in the team.
- No implied barrier on entry to a work-sharing construct; Yes at end of construct.
- Work-sharing Constructs:
  - Loop
  - Sections
  - Single
  - Workshare

Loop Construct

- All programs have loops; For computationally intensive loops; proper for OpenMP parallelisation
- Splits up loop iterations among the threads in a team
- Data parallelism

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

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

Sequential program

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

Parallel program

```
void main()
{
    double Res[1000];
    #pragma omp parallel num_threads(4)
    #pragma omp for
    for(int i=0;i<1000;i++)
    {
        do_huge_comp(Res[i]);
    }
}
```

Loop Construct

- Clauses:

```
shared (list)
private (list)
reduction (operator: list)
schedule (type [,chunk])
ordered
firstprivate (list)
lastprivate (list)
nowait
```

Example: (Parallel matrix vector product y=Ax)

C - Serial:

```
for(i=0; i<n*n; ++i)
    A[i]=i;

for(i=0; i<n; ++i)
    x[i]=1;

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("y[%d]=%f\n",i, y[i]);
}
```

C - Parallel:

```
#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]);
    }
}
```

Example: (Parallel matrix vector product y=Ax)

Fortran - Serial:

```
do j=1,n
  do i=1,n
    A(i,j)=(i-1)*n+j-1
  end do
end do

 do i=1,n
    x(i)=1
  end do

do i=1,n
  sum=0.0
  do j=1,n
    sum=sum+A(i,j)*x(j)
  end do
  y[i]=sum
    print *,"y[",i,"]=",y[i]
end do
```

Fortran - Parallel:

```
!$omp parallel default(none),
               shared(n, A, x, y),
               private(i, j, sum, tid)
    tid=omp_get_thread_num()
    !omp omp do
    do i=1,n
       sum=0.0
       do j=1,n
          sum=sum+A(i,j)*x(j)
       end do
       y[i]=sum
       print *,tid,": y[",i,"]=",y[i]
    end do
    !$omp end do
!$omp end paralllel
```

Size n = 16.

Compile: (Intel)

```
$icc hello.c -openmp
```

Execute:

```
$export OMP_NUM_THREADS=4
$qsub script.pbs
```

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

ICHEC

Data dependency

- Two rows may not be updated simultaneously; but columns may be
- For the inner loop: it requires n-1 fork and join
- Invert the loops; better in Fortran

```
for(i=0; i<n; i++)
   for(j=0; j<n; j++)
      a[i][j]=2*a[i-1][j];
```

Loop dependency

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

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

Reduction Clause

C - Serial:

```
for(i=0; i<n*n; ++i)
    A[i]=i;

for(i=0; i<n; ++i)
    x[i]=1;

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("y[%d]=%f\n",i, y[i]);
}
```

C - Parallel:

```
#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]);
    }
}
```

Reduction Clause

- Performs a reduction operation on the variables in the list

  C/C++: `reduction(operator: list)`

  Fortran: `reduction(operator|intrinsic: list)`

- How do we handle this case?

```
sum=0;
for(i=1; i<n; ++i)
   sum+=a[i];
```

```
sum=0;
#pragma omp parallel
#pragma omp for reduction(+:sum)
for(i=1; i<n; ++i)
   sum+=a[i];
```

Reduction Clause

- Performs a reduction operation on the variables in the list

  C/C++: `reduction(operator: list)`

  Fortran: `reduction(operator|intrinsic: list)`

- How do we handle this case?

```
sum=0
do i=1,n
   sum=sum+a(i)
enddo
```

```
sum=0
!$omp parallel
!$omp do reduction(+:sum)
do i=1,n
  sum=sum+a(i)
enddo
!$omp end do
!$omp end parallel
```

ICHEC

Reduction Clause

- A private copy of each list variable is created for each thread.
- Each thread does the partial reduction.
- At the end, reduction variable is combined with private copies.
- Reduction operations: +, -, *, … (**C/C++/Fortran**)
  &&, ||, … (**C/C++**)
  .AND., .OR., … (**Fortran**)
  max, min, … (**Fortran**)

Example: (Parallel matrix vector product y=Ax)

C - Serial:

```
for(i=0; i<n*n; ++i)
    A[i]=i;

for(i=0; i<n; ++i)
    x[i]=1;

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("y[%d]=%f\n",i,
y[i]);
}
```

C - Parallel:

```
#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]);
    }
}
```

ICHEC

Example: (Parallel matrix vector product y=Ax)

C - Serial:

```c
for(i=0; i<n*n; ++i)
    A[i]=i;

for(i=0; i<n; ++i)
    x[i]=1;

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("y[%d]=%f\n",i, y[i]);
}
```

C:

```c
#pragma omp parallel default(shared)
{
for (i = 0; i < n; ++i)
{
  sum=0;
  #pragma omp for 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]);
}
}
```

ICHEC

Example: (Calculation of y[0], i=0)

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

y[0]=120.

Example: (Parallel matrix vector product y=Ax)

Fortran:

```
!$omp parallel default(shared)
do i=1,n
   sum=0;
   !$omp do private(j, tid) reduction(+:sum)
   do j=1,n
      sum=sum+A(i,j)*x[j];
      if(i==0) then
        tid=omp_get_thread_num();
        print *,"Tid=",tid, "A[",i,",",",j]=",a(i,j), "sum=", sum
      end if
   end do
   !$omp end do
   y[i]=sum
   print *,"y[",i,"]=",y[i]
enddo
!$omp end parallel
```

ICHEC

Loop Construct

- Clauses:

```
shared (list)
private (list)
reduction (operator: list)
schedule (type [,chunk])
ordered
firstprivate (list)
lastprivate (list)
nowait
```

Schedule Clause

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

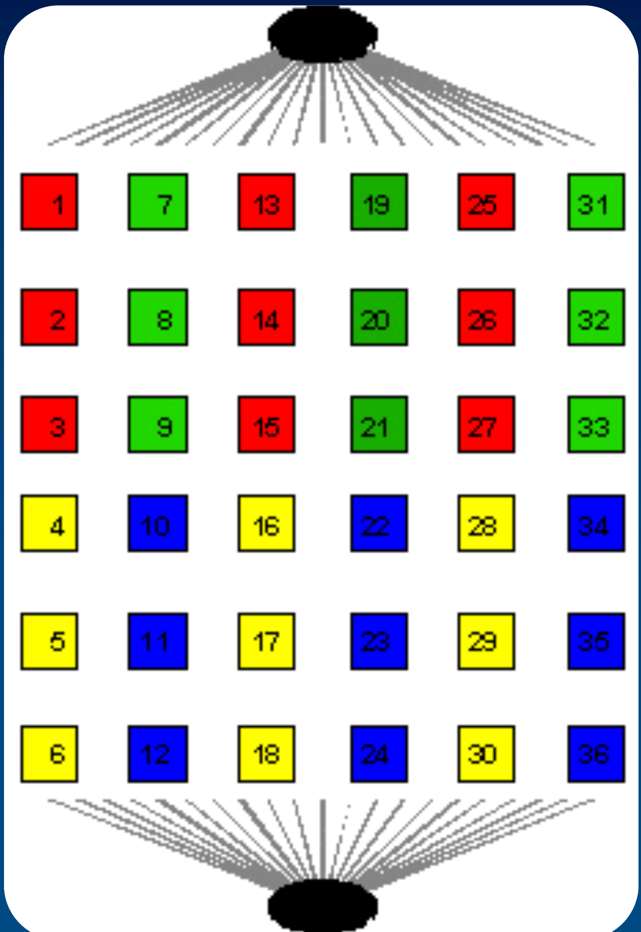
  C/C++: `schedule (type [,chunk])`

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

- Types:
  - `schedule(static [,chunk])`: divided into pieces of size chunk, and statically assigned to threads.
  - `schedule(dynamic [,chunk])`: divided into pieces of size chunk, and dynamically scheduled as requested.
  - `schedule(guided [,chunk])`: size of chunk decreases over time.

ICHEC

Schedule (Static)

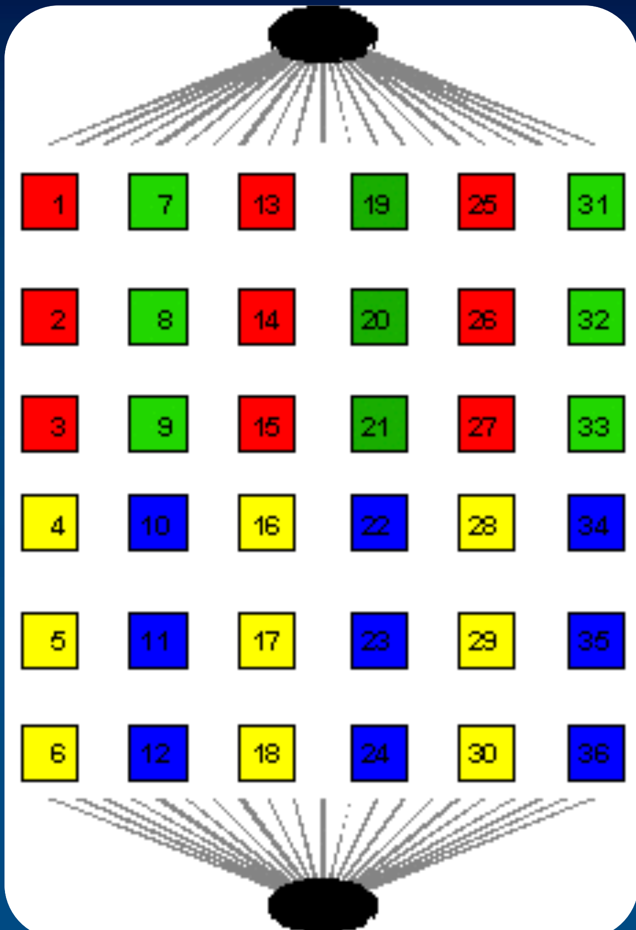


- Iterations are divided evenly among threads
- Divides the work into chunk sized parcels
- If there are n threads, each does every n^{th} chunk of work

C/C++:

```
#pragma omp for schedule(static,3)
for(i=0;i<36;++i)
   Work(i);
```

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:

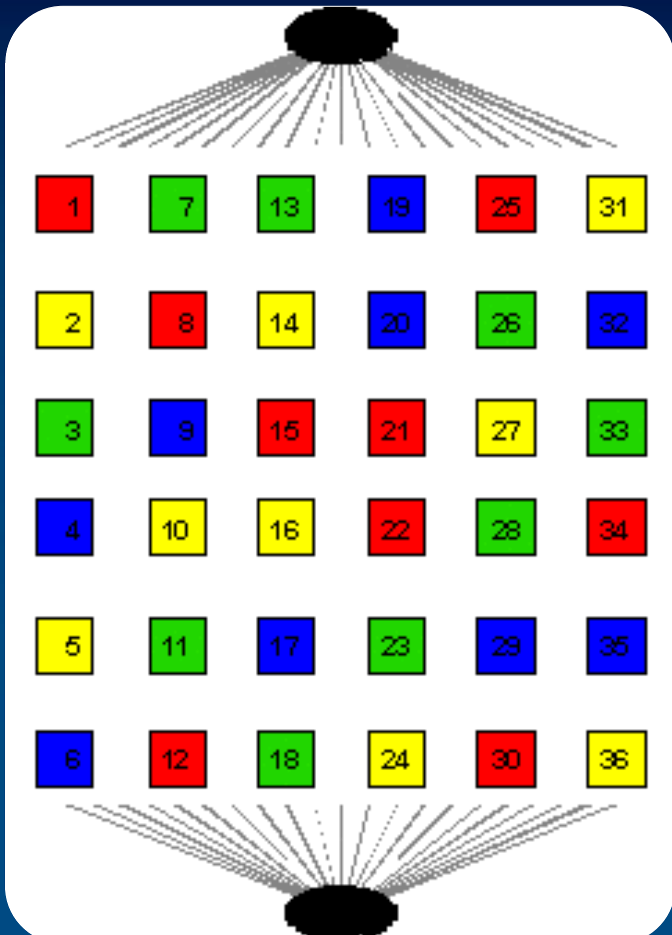
```
!$omp do schedule(static,3)

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

!$omp end do
```

ICHEC

Schedule(dynamic)



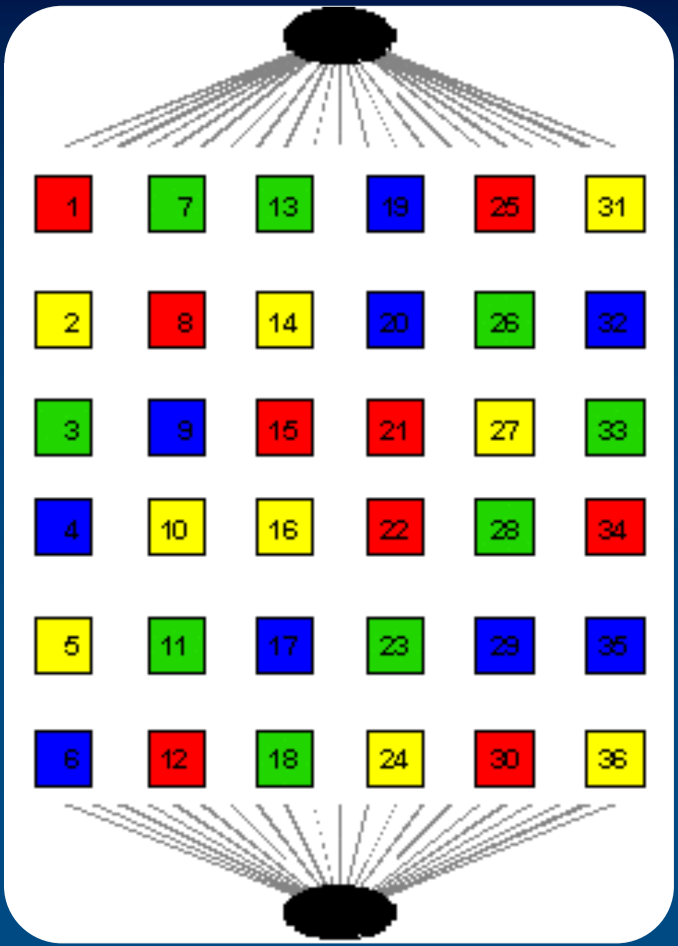
- Threads grab chunks of iterations dynamically
- As a thread finishes one chunk, it grabs the next available chunk
- More overhead than static, but better load balancing

C/C++:

```
#pragma omp for schedule(dynamic,1)
for(i=0;i<36;++i)
   Work(i);
```

ICHEC

Schedule(dynamic)



- Threads grab chunks of iterations dynamically
- As a thread finishes one chunk, it grabs the next available chunk
- More overhead than static, but better load balancing

Fortran:

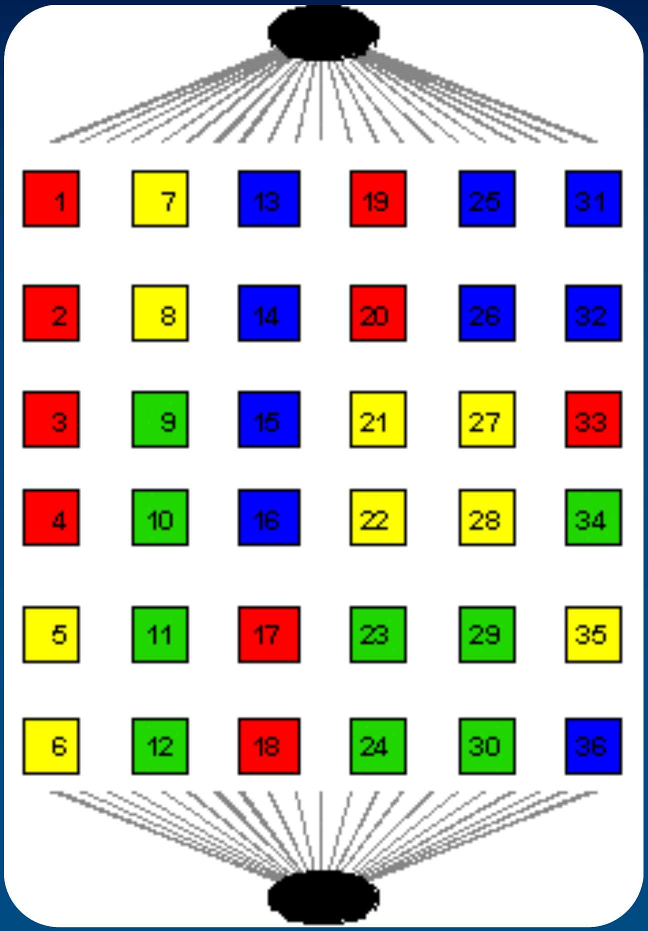
```
!$omp do schedule(dynamic,1)

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

!$omp end do
```

ICHEC

Schedule(guided)



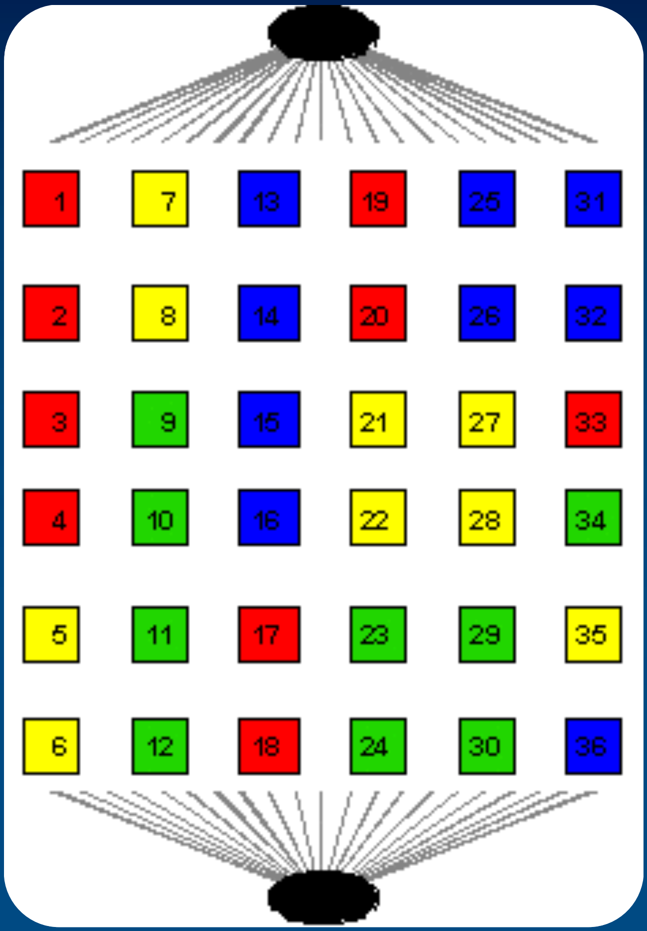
- Iterations are divided into chunks such that the size of each successive chunk decreases.
- chunk: the size of the smallest chunk size
- Less overhead than dynm., good l. b.

C/C++:

```
#pragma omp for schedule(guided,1)
for(i=0;i<36;++i)
   Work(i);
```

ICHEC

Schedule(guided)



- Iterations are divided into chunks such that the size of each successive chunk decreases.
- chunk: the size of the smallest chunk size
- Less overhead than dynm., good l. b.

Fortran:

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

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

!$omp end do
```

Work-Sharing Constructs

- Work-sharing construct divides work among the thread team; what kind of work to be parallelised.
- Inside the parallel region.
- No new threads created. Construct must be encountered by all threads in the team.
- No implied barrier on entry to a work-sharing construct; Yes at end of construct.
- Work-sharing Constructs:
  - Loop
  - Sections
  - Single
  - Workshare

ICHEC

Sections Construct

- Specifies enclosed sections that are distributed over the threads in the team; implicit barrier at the end
- Each section is executed by one thread.
- Functional parallelism

C/C++:

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

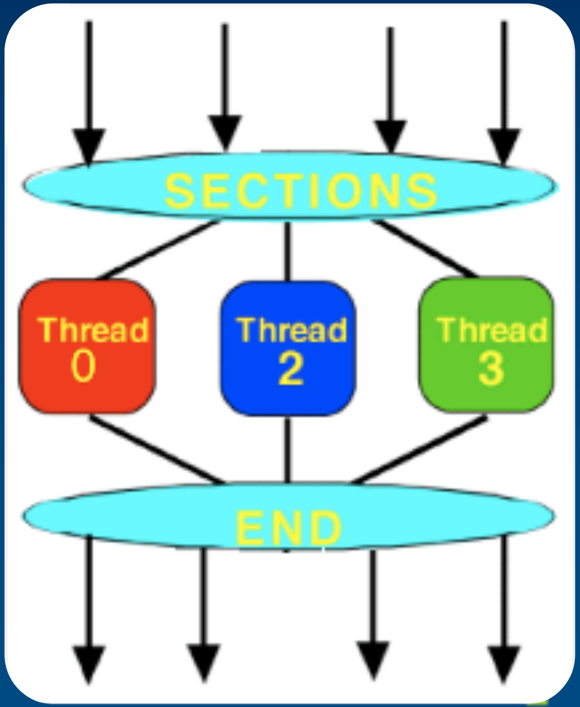
Fortran:

```
!$omp parallel [clauses]
!$omp sections [clauses]
   !$omp section
   ...
   !$omp section
   ...
!$omp end sections
[nowait]
!$omp end parallel
```

```
#include <stdio.h>
#include <omp.h>
int main()
{
    int tid;
    #pragma omp parallel private(tid)
    {
       tid = omp_get_thread_num();
       #pragma omp sections
       {
           #pragma omp section
           printf("Hello from thread %d \n", tid);
           #pragma omp section
           printf("Hello from thread %d \n", tid);
           #pragma omp section
           printf("Hello from thread %d \n", tid);
       }
    }
}
```

```
$export OMP_NUM_THREADS=4
```

```
Hello from thread 0
Hello from thread 2
Hello from thread 3
```



Sections Construct

- Clauses:

```
private (list)
firstprivate (list)
lastprivate (list)
reduction (operator: list)
nowait
```

Work-Sharing Constructs

- Work-sharing construct divides work among the thread team; what kind of work to be parallelised.
- Inside the parallel region.
- No new threads created. Construct must be encountered by all threads in the team.
- No implied barrier on entry to a work-sharing construct; Yes at end of construct.
- Work-sharing Constructs:
  - Loop
  - Sections
  - Single
  - Workshare

Single Construct

- Specifies a block of code that is executed by only one of the threads in the team.
- Rest of threads wait at the end of enclosed code block
- May be useful when dealing with sections of code that are not thread-safe.

C/C++:

```
#pragma omp parallel [clauses]
{
   #pragma omp single [clauses]
   …
}
```

Fortran:

```
!$omp parallel [clauses]
   !$omp single [clauses]
   ...
   !$omp end single
!$omp end parallel
```

ICHEC

Single Construct

- Clauses:

```
private (list)
firstprivate (list)
copyprivate (list)
nowait
```

```
#pragma omp parallel num_threads(4)
{
    #pragma omp single copyprivate(a)
    read_from_file(a);

    compute(a);

    #pragma omp single
    write_to_file(result);
}
```

- `Copyprivate(list)`: used to broadcast private variable values from a single thread to all instances of the private variables in the other threads.

Work-Sharing Constructs

- Work-sharing construct divides work among the thread team; what kind of work to be parallelised.
- Inside the parallel region.
- No new threads created. Construct must be encountered by all threads in the team.
- No implied barrier on entry to a work-sharing construct; Yes at end of construct.
- Work-sharing Constructs:
  - Loop
  - Sections
  - Single
  - Workshare

Workshare construct

- Fortran only
- Divides the execution of the enclosed structured block into separate units of work
- Threads of the team share the work
- Each unit is executed only once by one thread
- Allows parallelisation of
  - array and scalar assignments
  - WHERE statements and constructs
  - FORALL statements and constructs
  - parallel, atomic, critical constructs

```
!$omp workshare
   ...
!$omp end workshare
[nowait]
```

ICHEC

```
program wshare
use omp_lib
implicit none

integer :: i
real :: a(10), b(10), c(10)
do i=1,10
   a(i)=i
   b(i)=i+1
enddo

!$omp parallel shared(a, b, c)
!$omp workshare
   c=a+b
!$omp end workshare nowait
!$omp end parallel

end program wshare
```

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 except nowait.

```
#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
```

- Readibility
- Performance advantage

Outline

- Introduction to OpenMP
- OpenMP Directives
  - Directive Format
  - Parallel Construct
  - Work-Sharing Constructs
  - Synchronization Constructs
- Performance Considerations
- OpenMP Versions and Features

Synchronization Constructs

- Threads communicate through shared variables.
- Uncoordinated access of these variables can lead to undesired effects.
  1. Two threads update (write) a shared variable in the same step of execution, the result is dependent on the way this variable is accessed. This is called a race condition.
  2. Suppose that one processor has an updated result in private cache. Second processor wants to access that memory location - but a read from memory will get the old value since original data not yet written back.

Synchronization Constructs

- Synchronization imposes order constraints.
- Used to protect access to shared data.
- High-Level Synchronization Constructs:
  - master
  - critical
  - barrier
  - atomic
  - ordered
- Low-Level Synchronization Constructs:
  - flush
  - locks

ICHEC

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

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

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

```
#pragma omp parallel

......

#pragma omp critical
   result=result+temp;
```

Threads wait here: only one thread at a time executes the statement. So this is a piece of sequential code.

Barrier directive:

- Threads wait each other until all have reached that barrier

C/C++:

```
#pragma omp barrier
```

Fortran:

```
!$omp barrier
```

```
for(i= 0; i<N; i++)
   a[i] = b[i] + c[i];
#pragma omp barrier
for(i= 0; i<N; i++)
   d[i] = a[i] + b[i];
```

Threads wait here and only continue when all threads have reached the barrier point.

ICHEC

Atomic directive:

- Single thread can access a memory location at a time.
- x binop=expr, x++, --x, … (**C/C++**)
- x=x op expr, x=expr op x, … (**Fortran**)

```
#pragma omp parallel
……
#pragma omp atomic
   x+=temp;
```

C/C++:

```
#pragma omp atomic
   statement
```

Fortran:

```
!$omp atomic
   statement
```

ICHEC

Ordered directive:

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

C/C++:

```
#pragma omp for ordered
    …
#pragma ordered …
```

Fortran:

```
!$omp do ordered
. . .
!$omp ordered
   …
!$omp end ordered
!$omp end do
```

```
#pragma omp parallel for ordered
for(i=0;i<N;i++)
{
    #pragma ordered
    printf(" %d\n", i);
}
```

The output always will be
0, 1, 2, …., N

Synchronization Constructs

- Synchronization imposes order constraints and is used to protect access to shared data.
- High-Level Synchronization Constructs:
  - master
  - critical
  - barrier
  - atomic
  - ordered
- Low-Level Synchronization Constructs:
  - flush
  - locks

ICHEC

Flush directive:

- A synchronization point at which thread visible variables are written back to the memory

C/C++:

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

Fortran:

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

```
#pragma omp section
a=compute();
flush(a);
```

Locks

- Occasionally we may require more flexibility than is provided by critical and atomic directives.
- A lock is a special variable that may be set by a thread. No other thread may set the lock until the thread which set the lock has unset it.
- A lock must be initialised before it is used, and may be destroyed when it is not longer required.
- Lock variables should not be used for any other purpose.

C/C++:

```
omp_lock_t lock;

void omp_init_lock(omp_lock_t *lock)
void omp_destroy_lock(omp_lock_t *lock)
void omp_set_lock(omp_lock_t *lock)
void omp_unset_lock(omp_lock_t *lock)
int omp_test_lock(omp_lock_t *lock)
```

Fortran:

```
integer(omp_lock_kind) :: lock;

subroutine omp_init_lock(lock);
subroutine omp_destroy_lock(lock);
subroutine omp_set_lock(lock);
subroutine omp_unset_lock(lock);
subroutine omp_test_lock(lock);
```

- There are also nestable lock routines which allow the same thread to set a lock multiple times before unsetting it the same number of times.

Lock Example

```
omp_lock_t writelock;
omp_init_lock(&writelock);
#pragma omp parallel for
for ( i = 0; i < x; i++ )
{
    // some stuff
    omp_set_lock(&writelock);
    // one thread at a time stuff
    omp_unset_lock(&writelock);
    // some stuff
}

omp_destroy_lock(&writelock);
```

ICHEC

Timing with OpenMP

- `omp_get_wtime`

  C/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 t;
t = omp_get_wtime();
 … work to be timed …
t = omp_get_wtime()- t;
```

```
real(8) :: t
t= omp_get_wtime()
 … work to be timed …
t= omp_get_wtime()-t
```

- `omp_get_wtick`: Returns the number of seconds between processor clock ticks.

Outline

- Introduction to OpenMP
- OpenMP Directives
  - Directive format
  - Parallel Construct
  - Work-Sharing Constructs
  - Synchronization Constructs
- Performance Considerations
- OpenMP versions and features

Correctness vs Performance

- It may be easy to write a correctly functioning OpenMP program.
- But, it is not so easy to create a program that provides the desired level of performance!

ICHEC

Basic Strategies

- If a parallelized loop does not perform well, consider
  - Thread start up costs, The amount of time spent handling OpenMP constructs, Thread termination time
  - Avoid parallel overhead at low iteration counts

```
#pragma omp parallel for if(M > 800)
for(j=0; j< M; j++)
    aa[j] = alpha*bb[j] + cc[j];
```

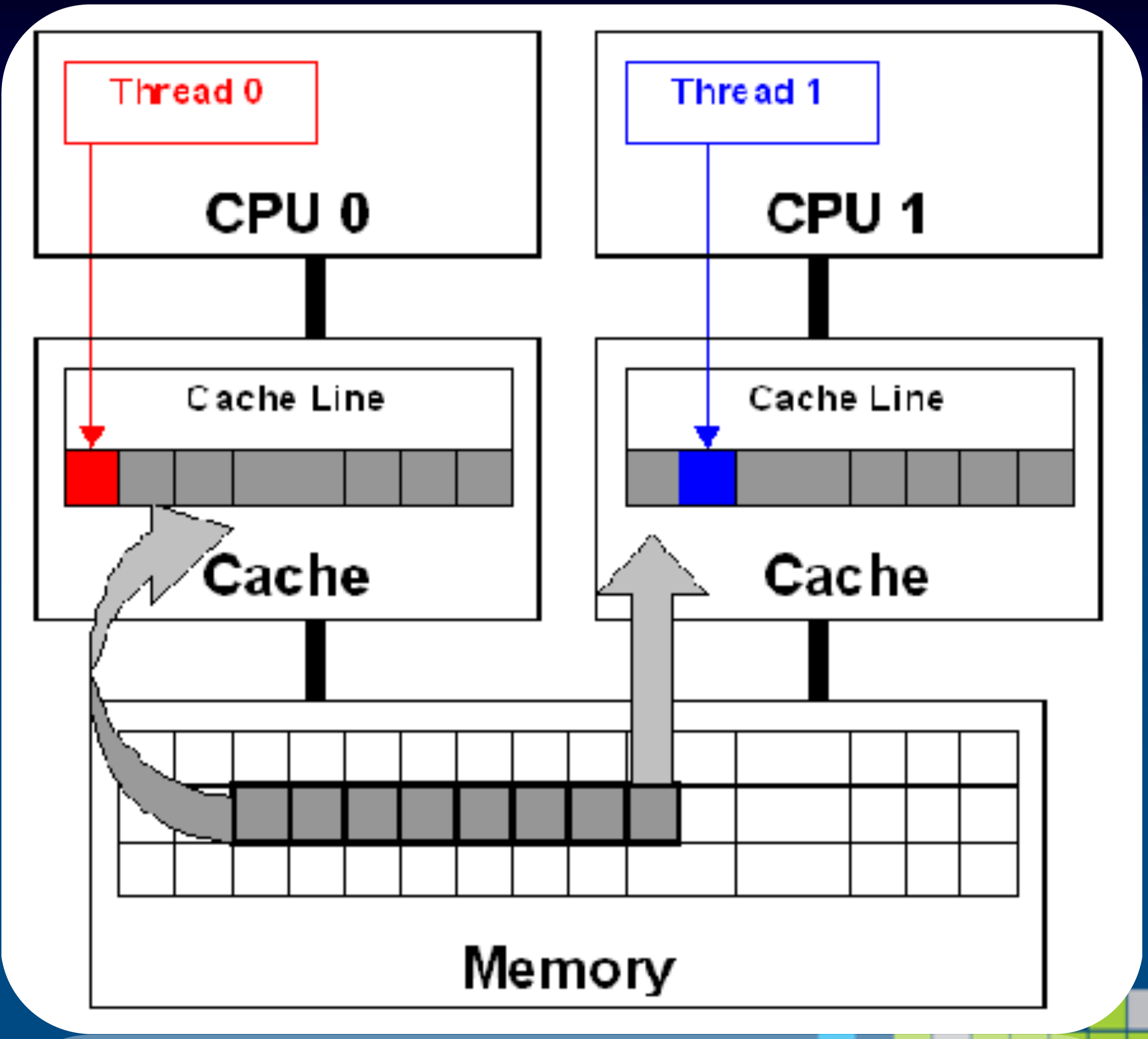
  - Load imbalances: Unequal work loads lead to idle threads and wasted time. Use proper scheduling type.
  - Unnecessary synchronization; Large Critical Regions; prefer atomic update if possible

Basic Strategies

- Many references to shared variables
  - Use private data - allocated on stack
- Low cache reuse
  - If each thread accesses a distinct portion of data consistently through the program, then they will load this data to their local cache and they don't need to go to memory for each reference.
  - In C, a 2-D array is stored in rows (and in columns in Fortran). Organize data accesses so that values are used as often as possible while they are still in the cache.
- False sharing
  - when two threads update different data elements in the same cache line.

This invalidates the cache line and forces a memory update to maintain cache coherency.

It can be reduced by making use of private data as much as possible



Debugging OpenMP Code

- Shared memory parallel programming opens up a range of new programming errors arising from unanticipated conflicts between shared resources.

- Race conditions:
  - Multiple threads are updating the same shared variable simultaneously.
  - Hard to find, not reproducible, answer varies with number of threads.

- Deadlock:
  - When threads hang while waiting on a locked resource that will never become available.
  - A simple approach is to put print statement in front of all lock calls.

Deadlock

```
omp_lock_t  lock;
omp_init_lock(&lock);
#pragma omp parallel sections
{
#pragma omp section
{
   omp_set_lock(&lock)
   ierr=work1();
   if (ierr==0) {
      omp_unset_lock(&lock);
   }else{
      printf("Error\n");
   }
#pragma omp section
{
   omp_set_lock(&lock);
   ierr=work2();
   omp_unset_lock(&lock);
}
}
omp_destroy_lock(&lock);
```

Deadlock

```
integer(omp_kind_lock) :: ilock
call omp_init_lock(ilock)
!$omp parallel sections
!$omp section
   call omp_set_lock(ilock)
   call work1(ierr)
   if (ierr.eq.0) then
      call omp_unset_lock(ilock)
   else
      print *, "Error"
   endif
!$omp section
   call omp_set_lock(ilock)
   call work2(ierr)
   call omp_unset_lock(ilock)
!$omp end parallel sections

call omp_destroy_lock(ilock)
```

Using Print Statements

- Advantages:
  - simple
  - useful for deterministic bugs
  - monitoring the iterations on threads
- Disadvantages:
  - slow
  - specification of what to display
  - human intensive bug hunting

Debugging and Profiling Tools

- Debugging Tools:
  - Intel debuggers: gdb-ia
  - Intel Inspector
  - DDT
  - Totalview
- Profiling Tools:
  - Valgrind
  - GNU profiler: gprof
  - Intel VTune
  - ompP
  - Tau
  - VampirTrace

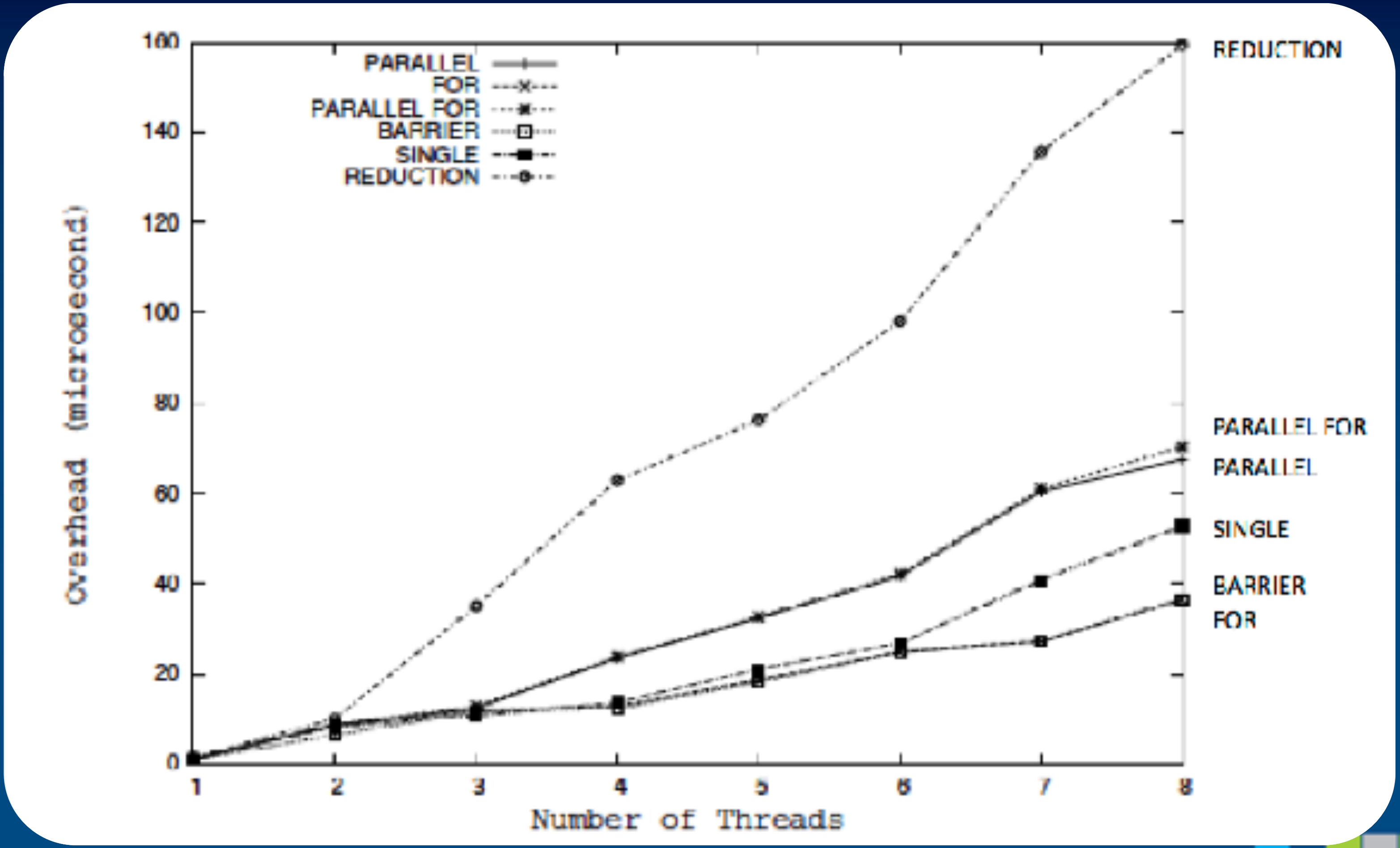
General Performance Considerations

- Be aware of directives cost
  - Parallelize outer loops
  - Minimize the number of directives
  - Minimize synchronization – minimize the use of barrier, critical, ordered
  - Consider using nowait clause of omp for/do when enclosing several loops inside one parallel region.
  - Merge loops to reduce synchronization cost
  - Maximize parallel regions

- Be aware of the Amdahl's law
  - Minimize serial code
  - Remove dependencies among iterations
- Balance the load
  - Experiment with using schedule clause
- Reduce false sharing
  - Use private variables
- Try task level parallelism

ICHEC

OpenMP Directives Overhead

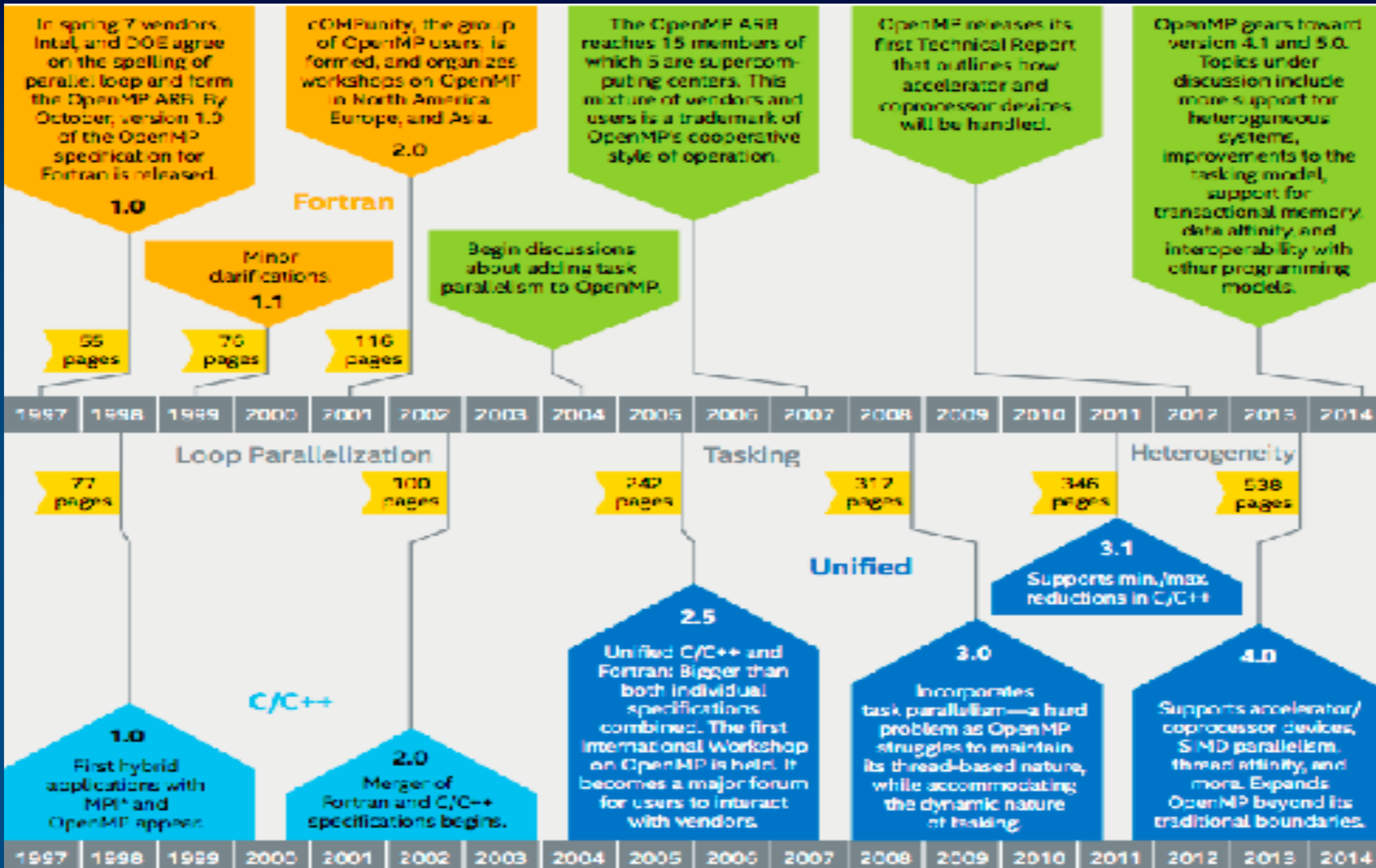


Outline

- Introduction to OpenMP
- OpenMP Directives
  - Directive format
  - Parallel Construct
  - Work-Sharing Constructs
  - Synchronization Constructs
- Performance Considerations
- OpenMP versions and features

ICHEC

OpenMP Timeline



OpenMP 3.0

Version 3.0 released in May 2008

- New task level parallelism
- Improvements to loop and nested parallelism
- Additional Clauses, runtime functions and environment variables

Version 3.0 Features

- Adding tasking is the biggest addition for 3.0
- OpenMP has always had tasks, never called them explicitly that.
- For parallelizing irregular problems, unbounded loops, recursive algorithms, multi-block grids and many others

C/C++:

```
#pragma omp task [clauses]
{
  …
}
```

Fortran:

```
!$omp task [clauses]
  …
!$omp end task
```

- Clauses: if, untied, shared, private, firstprivate, default
  - When if is false, executed immediately by the encountering thread.
  - Tied by default: The same thread from beginning to end will execute the code

- untied: to specify that different threads execute different parts of the code

```
#pragma omp parallel num_threads(n)
{
#pragma omp task
  function_A();
#pragma omp barrier
#pragma omp single
  {
  #pragma omp task
  function_B();
  }
}
```

- Tasks can create descendants to form a task tree. They may run in parallel to the parent or suspend and run synchronously.
- Task Synchronisation:
  - taskwait: Waits for all child tasks to be completed.

    C/C++: #pragma omp taskwait

    Fortran: !$omp taskwait
  - Can be controlled by omp barrier/ implicit barrier

Ex: (Fibonacci Sequence)

fib(0)=0

fib(1)=1

fib(n)=fib(n-1)+fib(n-2), n>1

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

```
int main(){
  int n=50;

   omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf("fib(%d)=%d\n", n, fib(n));
  }
}
```

```
int fib(int n)
{
int i, j;
if(n<2) return n;
else{
   #pragma omp task shared(i) firstprivate(n)
   i=fib(n-1);
   #pragma omp task shared(j) firstprivate(n)
   j=fib(n-2);
   #pragma omp taskwait
   return i+j;
}}
```

- Better support for nested parallelism
  - controls the maximum number of nested active parallel regions

  omp_set_max_active_levels(), OMP_MAX_ACTIVE_LEVELS
  - sets the number of OpenMP threads to use for the whole OpenMP program

  omp_get_thread_limit(), OMP_THREAD_LIMIT

- Improvements to loop parallelism
  - schedule(runtime): to set iterations at runtime through the environment variable OMP_SCHEDULE
  - schedule(auto): The compiler or runtime system decides what is best to use depending on the implementation

ICHEC

collapse(n) clause: parallelize perfectly nested multi-dimensional loops; takes a constant positive integer as a parameter, which determines how many loops are collapsed. Compiler then forms a single loop and parallelizes it.

```
#pragma omp parallel for collapse(2)
for(i=0;i<n;i++)
   for(j=0;j<m;j++)
      …
```

OMP_STACK_SIZE size [B|K|M|G]:

control of therad's stack size

OpenMP 3.1

Version 3.1 released in July 2011

- Additional Clauses
- Improvements to task parallelism
- Initial support for thread binding

Version 3.1 Features

- Extensions to OpenMP tasking model
  - Taskyield construct: define task-switching points; suspend the current task and execute another

    C/C++: #pragma omp taskyield

    Fortran: !$omp taskyield
  - mergeable clauses: a task can have the same data region as the generating task region; avoid potentially expensive initialization of the task environment.
  - final clause: a task that makes all its child tasks to be executed sequentially in the same region; a task may not be scheduled for deferred execution, but instead is immediately executed.

ICHEC

Version 3.1 Features

  - omp_in_final(): returns true if the calling task is final.
- Initial support for thread binding: Control whether OpenMP threads are allowed to move between processors or not
  - OMP_PROC_BIND to TRUE for not moving threads
- Control nested thread team sizes: export OMP_NUM_THREADS=n1, n2, n3
- For C/C++, the reduction clause now accepts min and max functions.
- The atomic construct now accepts the clauses: read, write, update and capture

ICHEC

OpenMP 4.0

OpenMP 4.0 released in July 2013

- SIMD directives
- Extended support for thread affinity
- New user-defined reductions
- Error Handling
- Accelerators support
- Support for Fortran 2003

ICHEC

Version 4.0 Features

- Simd Construct: Transform the loop into a simd loop
- multiple iterations of the loop can be executed concurrently
- Loop to be executed using SIMD lanes:

C/C++:

```
#pragma omp simd [clauses]
   for – loops
```

Fortran:

```
!$omp simd [clauses]
   do – loops
!$omp end simd
```

- Function that can be called from a SIMD loop.

```
#pragma omp declare simd [clauses]
   function
```

```
!$omp declare simd [clauses]
   function
!$omp end simd
```

ICHEC

Version 4.0 Features

Clauses for simd: safelen, linear, aligned, private, lastprivate, reduction, collapse

- safelen (length): limits the number of iterations in a SIMD chunk
- linear (list): declares a number of list items to be private to a SIMD lane
- aligned (list): declares a number of items to be aligned to some number of bytes

Clauses for declare simd: simdlen, linear, aligned, uniform, reduction, inbranch, notinbranch

- simdlen (length): number of concurrent arguments for the function
- uniform (argument): arguments to have an invariant value for all function invocations
- inbranch/notinbranch: the function will always/never be called from inside an if statement of a SIMD loop

- New Reduction Clause: use your own reduction operation on your own type

C/C++:
#pragma omp declare reduction*(reduction-identifier : typename-list : combiner) [initializer-clause]*

Fortran:
!$omp declare reduction*(reduction-identifier : type-list : combiner) [initializer-clause]*

- Reduction-identifier: gives a name to the operator
- Typename-list: A list of types to which it applies
- Combiner: specifies how to combine values
- Initializer-clause: specifies how to initialise the private elements of each thread

```
#pragma omp declare reduction (merge : std::vector<int> :
omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

```
void schedule (std::vector<int> &v, std::vector<int> &filtered) {
#pragma omp parallel for reduction (merge : filtered)
  for (std:vector<int>::iterator it = v.begin(); it < v.end(); it++)
    if ( filter(*it)) filtered.push_back(*it);
}
```

- omp_out refers to private copy that holds combined value
- omp_in refers to the other private copy

- Affinity Support: better locality, less false sharing, more memory bandwidth.
  - New Clause to the parallel construct: specifes the places to use for the threads in the team

    proc_bind(master|close|spread)

    master: threads to the same place as the master thread

    close: threads close to the place of the master thread

    spread: spread threads across the machine
  - OMP_PROC_BIND can now specify master, close, spread
  - New RTLs: omp_proc_bind_t omp_get_proc_bind(void)
  - OMP_PLACES: bind the OpenMP threads to the places

  in the list in terms of threads, core, sockets

- Error Model:
  - improve the stability of OpenMP applications against system-level, runtime-level, and user-defined errors.
  - based on conditional cancellation and user-defined cancellation points
  - Cancel construct: cancellation of all tasks in the same construct

C/C++:

```
#pragma omp cancel [clauses]
```

Fortran:

```
!$omp cancel [clauses]
```

Clauses: parallel, sections, for/do, taskgroup, if

taskgroup: marks a region such that all tasks started in it belong to a group

- OMP_CANCELLATION set to TRUE to enable

- Accelerator model:
  - Enables the usage of accelerators and coprocessors to offload computation
  - target construct: marks a region to execute on device
    - Creates device tasks that are executed by device-only threads.
  - A device has a data environment. Variables are copied in and out of a data environment, or

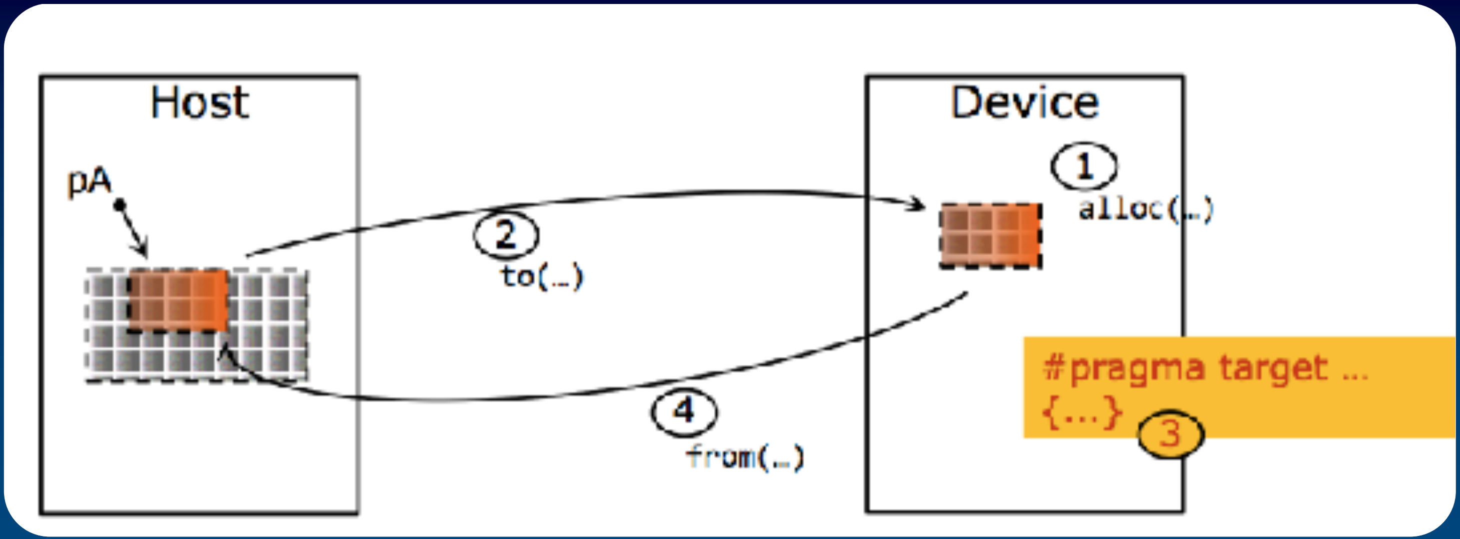
C/C++:

```
#pragma omp target [clauses]
  …
```

Fortran:

```
!$omp target [clauses]
  …
!$omp end target
```

- Clauses: if, device(expression), map (*[map-type:] list*)



- to: existing host variables copied to a corresponding variable in the target before
- from: target variables copied back to a corresponding variable in the host after
- tofrom: Both from and to
- alloc: Neither from nor to, but ensure the variable exists on the target but no relation to host variable.

OpenMP 5.0 Plans

- Support for memory affinity
- Refinements to accelerator support
- Additional task/thread synchronization mechanisms
- Completing extension of OpenMP to Fortran 2003

ICHEC

Conclusions

- OpenMP is a parallel programming model for SMP machines.
- All threads have access to a shared main memory; each thread can have local memory.
- The parallelism has to be decided explicitly by the programmer: Data parallelism, Task parallelism
- To control the parallelization, thread exclusion and synchronization constructs can be used.

- OpenMP is successful in small-to-medium SMP systems.
- Multiple cores/CPUs dominate the future computer architectures; OpenMP would be the major parallel programming language in these architectures.
- Simple: everybody can learn it in 2 weeks.
- Not so simple: Don' t stop learning! Keep learning it for better performance.

References

- http://openmp.org
- https://computing.llnl.gov/tutorials/openMP
- http://openmp.org/wp/openmp-specifications
- Michael J. Quinn, Parallel Programming in C with MPI and OpenMP, Mc Graw Hill, 2003.
- Barbara Chapman, Gabriele Jost and Ruud Van Der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, Volume 10, MIT Press, 2008.