Link to the Slides and Other Material

https://tinyurl.com/y6v6ftwg

Outline

What is Interactive Parallelization Tool (IPT)?

Introduction to Our Approach for Teaching Parallel Programming

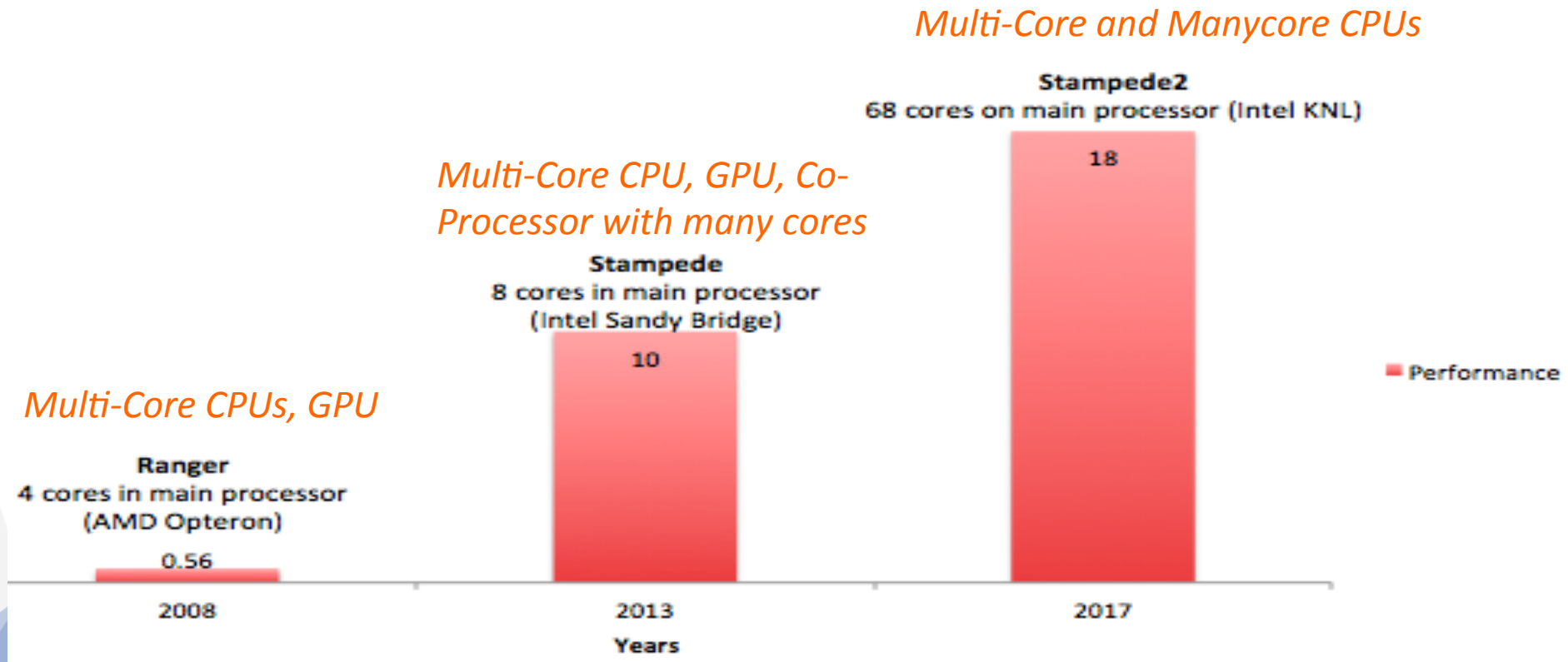
Introduction to OpenMP

Parallelizing applications using IPT (hands-on session)

Keeping-Up with the Advancement in HPC Platforms can be an Effort-Intensive Activity

- Code modernization can be required to take advantage of the continuous advancement made in the computer architecture discipline and the programming models
  - To efficiently use many-core processing elements
  - To efficiently use multiple-levels of memory hierarchies
  - To efficiently use the shared-resources
- The manual process of code modernization can be effort-intensive and time-consuming and can involve steps such as follows:
  1. Learning about the microarchitectural features of the latest platforms
  2. Analyzing the existing code to explore the possibilities of improvement
  3. Manually reengineering the existing code to parallelize or optimize it
  4. Explore compiler-based optimizations
  5. Test, and if needed, repeat from step 3

Evolution in the HPC Landscape – HPC Systems at TACC

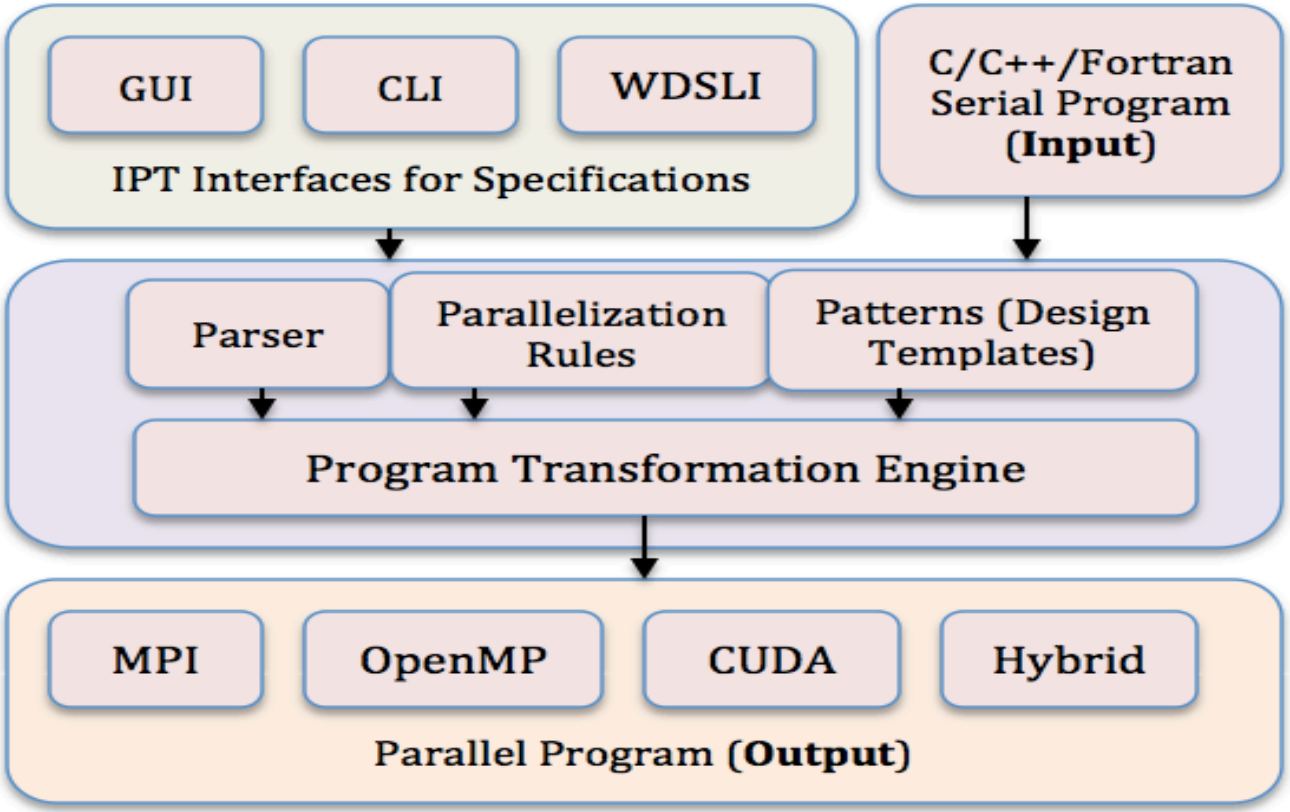


IPT – How can it help you?

If you know **what** to parallelize and **where**, IPT can help you with the syntax (of MPI/OpenMP/CUDA) and typical code reengineering for parallelization

- Main purpose of IPT: *a tool to aid in learning parallel programming*
- Helps in learning parallel programming concepts without feeling burdened with the information on the syntax of MPI/OpenMP/CUDA
- C and C++ languages supported as of now, Fortran will be supported in future

IPT: High-Level Overview



Before Using IPT

- It is important to know the logic of your serial application before you start using IPT
  - IPT is not a 100% automatic tool for parallelization
- Understand the high-level concepts related to parallelization
  - Data distribution/collection
    - For example: reduction
  - Synchronization
  - Loop/Data dependency
- Familiarize yourself with the user-guide

How are we teaching Parallel Programming with IPT?

- We have classes where we introduce the concept and many details, followed by some examples

<u>The IPT training class is different</u>

- Code modification with our tool IPT
- Short introduction
- Example: serial → parallel with IPT
- Inspection of the semi-automatically parallelized code
- Learning by doing
- Focus on concepts; less important syntax taken care of by IPT
- Next example focusing on other features
- …

First: Discuss High-Level Concepts

General Concepts Related to Parallel Programming:

- Data distribution/collection/reduction
- Synchronization
- Loop dependence analysis (exercise # 2)

Must know before using IPT

Specific to OpenMP:

IPT can help with most of these

- A **structured block** having a single entry and exit point
- Threads communicate with each other by reading/writing from/to a shared memory region
- Compiler directives for creating teams of threads, sharing the work among threads, and synchronizing the threads
- Library routines for setting and getting thread attributes

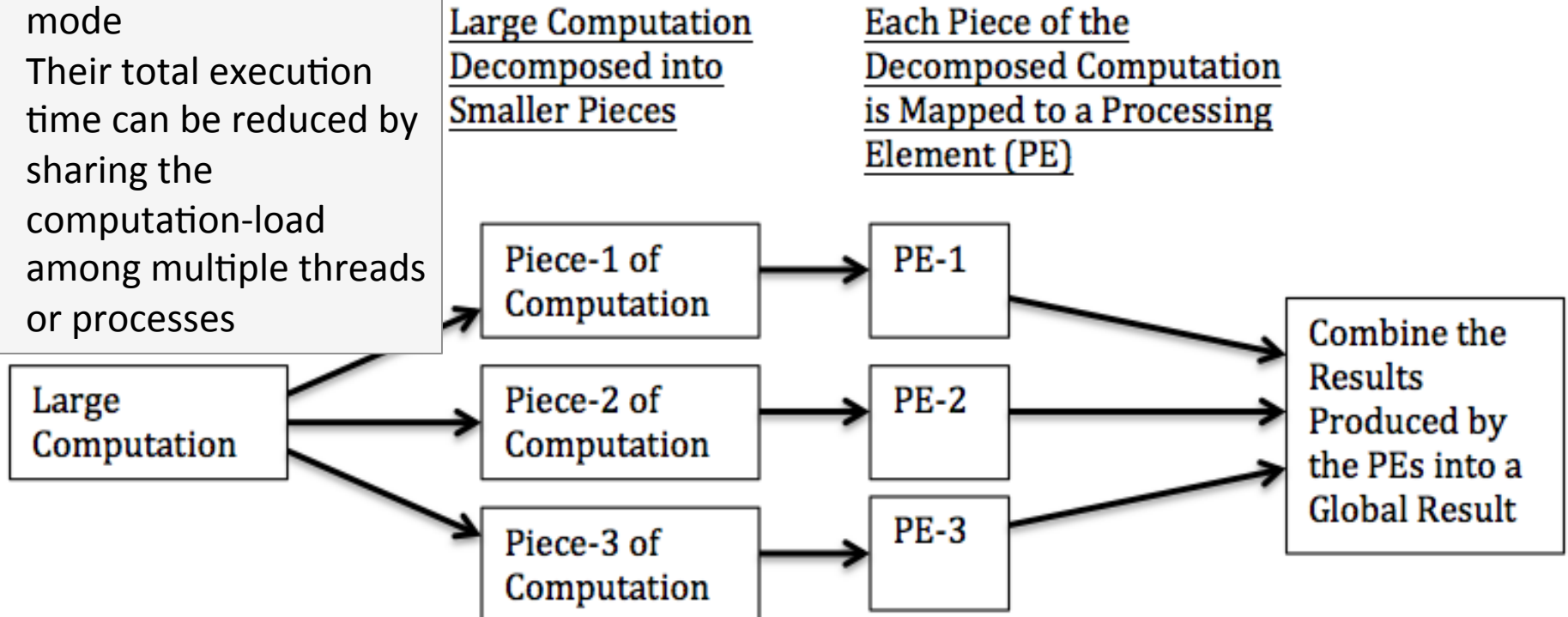
Additional Concepts Related to OpenMP:

Programmer needs to decide at run-time

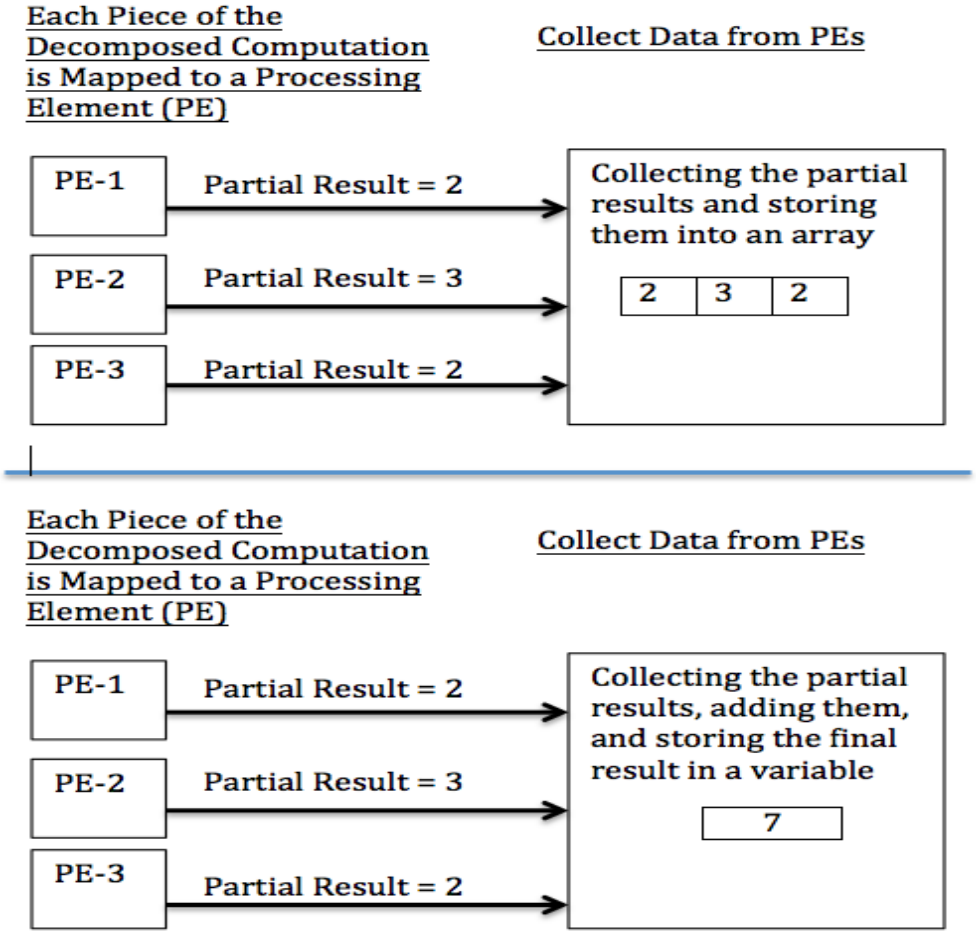
- Environment variables to control run-time behavior

Process of Parallelizing a Large Number of Computations in a Loop

- Loops can consume a lot of processing time when executed in serial mode
- Their total execution time can be reduced by sharing the computation-load among multiple threads or processes



Data Distribution/Collection/Reduction



Processing Element (PE) is a thread in OpenMP

Synchronization

- Synchronization helps in controlling the execution of threads relative to other threads in a team

- Synchronization constructs in OpenMP:

  master, single, atomic, critical, barrier, taskwait, flush, parallel {…}, ordered

Loop/Data Dependency

- Loop dependence implies that there are dependencies between the iterations of a loop that prevent its parallel processing
  - Analyze the code in the loop to determine the relationships between statements
- Analyze the order in which different statements access memory locations (data dependency)
- On the basis of the analysis, it may be possible to restructure the loop to allow multiple threads or processes to work on different portions of the loop in parallel
- For applications that have hotspots containing ante-dependency between the statements in a loop (leading to incorrect results upon parallelization), code refactoring should be done to remove the ante-dependency prior to parallelization. One such example is example2.c

As a Second Step: Gentle Introduction to OpenMP

Learning Objective

-- Introduction --

What is parallel computing?

What is OpenMP?

How to get started with OpenMP

Parallelizing code with IPT (2 examples)

What is scaling and parallel efficiency?

Some OpenMP basics

More examples

OpenMP?

OpenMP stands for **Open Multi-Processing**

Three primary components:

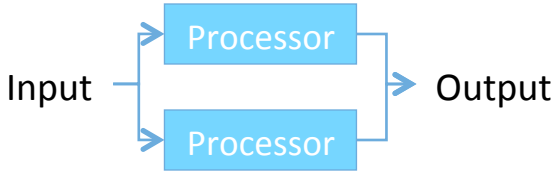
- Directives to the Compiler within code (pragmas in C/C++, comments in Fortran)
- Runtime Library Routines
- Environment Variables

http://www.openmp.org/

- OpenMP 4.5 Complete Specifications (Nov. 2015)
- OpenMP Examples 4.0.2 (March 2015)
- OpenMP Summary Card (C/C++/Fortran)

Why Parallel? (1) --- Power efficiency

Power = CV^2F



Cap = 1.0c
Volts = 1.0v
Freq = 1.0f
Power = $1.0cv^2f$

Do same work with 2 processors at 0.5 freq.

Voltage ~ Frequency
2x more wires ->
~2x Capacitance

Cap = 2.2c
Volts = 0.6v
Freq = 0.5f
Power = $0.4cv^2f$

https://www.youtube.com/watch?v=cMWGeJyrc9w&index=2&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG

Why Parallel? (2) --- and why OpenMP

- Power efficiency
- Faster execution --- less time to solution
- More available memory

<u>OpenMP is for shared-memory, i.e. for a single node</u>

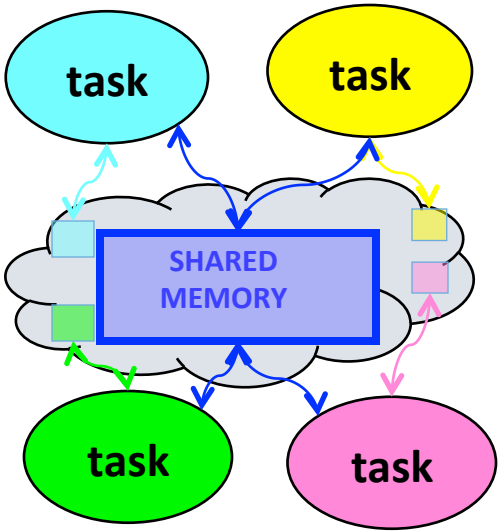
- Example: node with 16 cores

1. Execute serial code 16 times, one instance per core
2. Execute (OpenMP) parallel code on all 16 cores

- In setup #1 you'll have 1/16 of the memory available for each instance of the code

With MPI (distributed memory) you can use more than one node; same story, though …. but this is for a different time

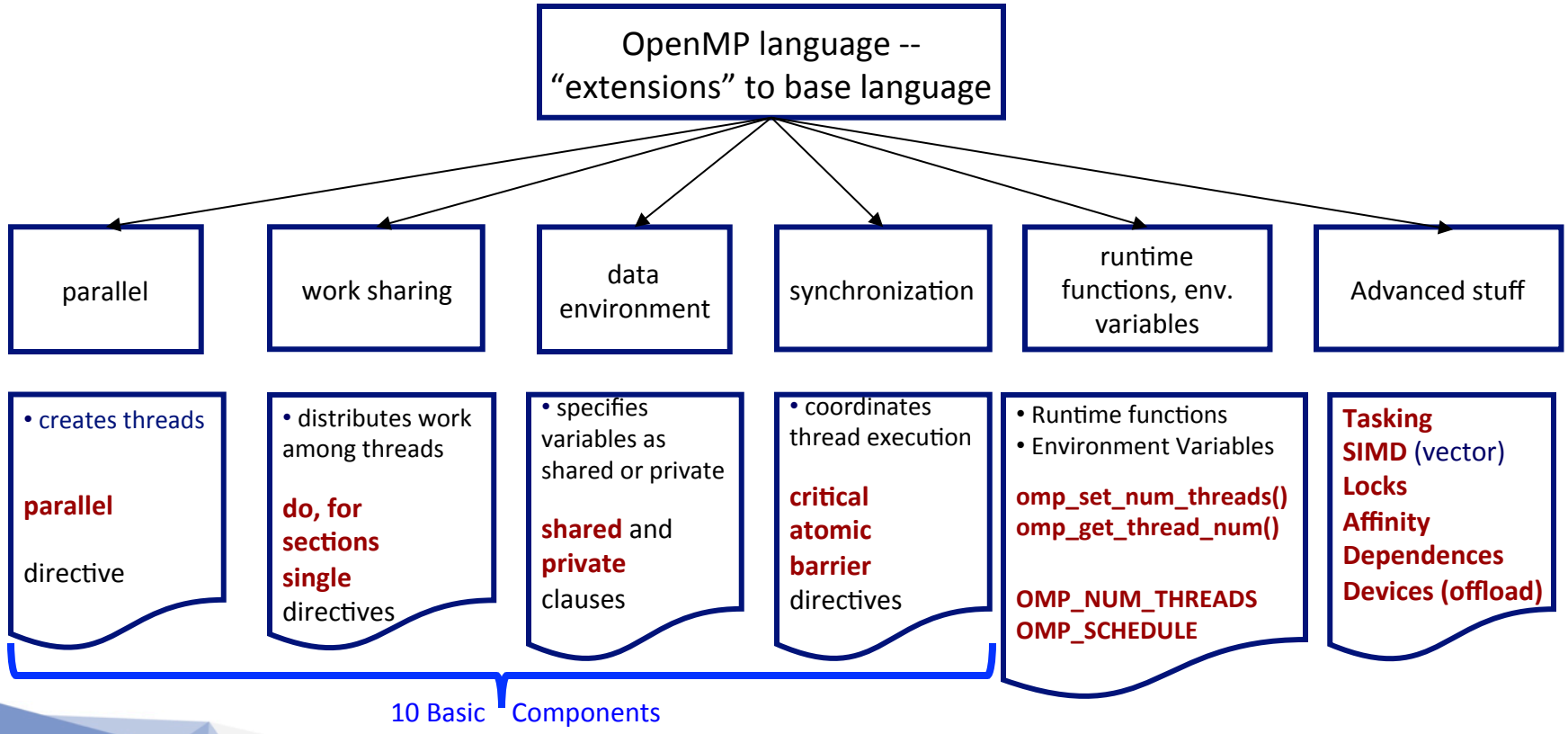
Shared-Data Model




Private
Memory

- Threads Execute on Cores/HW-threads
- In a parallel region, team threads are assigned (tied) to implicit tasks to do work. Think of tasks and threads as being synonymous.
- Tasks by “default” share memory declared in scope before a parallel region.
- Data: shared or private
  - Shared data: accessible by all tasks
  - Private data: only accessible by the owner task

OpenMP Constructs



How to teach OpenMP?

- We have classes where we introduce the concept and many details, followed by some examples

<u>This class is different</u>

- Code modification with our tool IPT
- Short introduction
- Example: serial → parallel with IPT
- Inspection of the semi-automatically parallelized code
- Learning by doing
- Focus on concepts; less important syntax taken care of by IPT
- Next example focusing on other features
- …

First Things First (1)

Logon to Stampede

How to logon to Stampede:

- You all should have a TACC portal account.
  - Use this account name and password to logon
- Open a terminal on your computer (MacOS or Linux)
- Use Putty on a Windows laptop

```
$ ssh <username>@stampede.tacc.utexas.edu
```

First Things First (2)

Start an interactive session on Stampede with `idev`

How to launch an **`idev`** job on Stampede:

- Idev: Interactive development
- When asked, accept to use the reservation
- Once the job is running and the prompt returns: check hostname

- `$ idev -m 120`
- `$ hostname`

- Now all of you have a single node where you can edit the code, run IPT, and run the serial and parallel code

Retrieve the Example Files

Copy the files from Ritu's account

- `$ cp -pr /work/01698/rauta/trainingIPT .`
- Now all of you have a single node where you can edit the code, run IPT, and run the serial and parallel code

Get IPT ready for use by executing a shell script

- `$ source ./runBeforeIPT.sh`
- `$ module load intel`

First example (1)

Let's start with example 1

- `$ cd exercises`
- `$ cd exercise1`
- `$ ls -al`

- There is a Fortran file and a C file
- For now IPT only works with C/C++ (but you can look at the Fortran source, if that is more convenient)

First example (2)

Let's start with example 1

- Before using IPT inspect the code (we will do this together here)
- Use one of these: 'more', 'vi', or 'emacs'

```
$ more example1.c
$ vi example1.c
$ emacs example1.c
```

IPT will ask you a lot of questions
These are the important ones (for now)

- Which loop should IPT parallelize?
- Is there a reduction?

First example (3)

Example1

2 arrays: x and y

Loop 1 initializes x

Loop 2: Stencil update

1. $y_{i,j}$ is calculated from $x_{i,j}$
2. A temporary variable is being used
3. The sum of all elements is calculated

The latter, i.e., calculation (3) is called a reduction

```
#include <stdio.h>
#include <sys/time.h>
#define N 30000
int main(){
  int i, j;
  double x[N+2][N+2], y[N+2][N+2], sum, tmp;

  //for timing the code section
  struct timeval start,end;
  float delta;
  for(i=0; i<=N+1; i++){
    for(j=0; j<=N+1; j++){
      x[i][j] = (double) ((i+j)%3) - 0.9999;
    }
  }
  printf("\nMemory allocation done successfully\n");
  //start timer and calculation
  gettimeofday(&start, NULL);

 for(j=1; j<N+1; j++){
   for(i=1; i<N+1; i++ ){
     tmp = 0.2 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1]);
     y[i][j] = tmp;
     sum = sum + tmp;
   }
  }

  //stop timer and calculation
  gettimeofday(&end, NULL);
  delta = ((end.tv_sec-start.tv_sec)*1000000u + end.tv_usec-start.tv_usec)/1.e6;
  printf("\nThe total sum is: %lf\n", sum);
  //print time to completion
  printf("run time    = %fs\n", delta);
  return 0;
}
```

Loop 1

Loop 2

First example (4)

Example1

Running IPT: essentials

- Parallelize loop #2
- Instruct IPT to add a reduction
  1. Reduction variable: sum
  2. Reduction operation: add

```
#include <stdio.h>
#include <sys/time.h>
#define N 30000
int main(){
  int i, j;
  double x[N+2][N+2], y[N+2][N+2], sum, tmp;

  //for timing the code section
  struct timeval start,end;
  float delta;
  for(i=0; i<=N+1; i++){
     for(j=0; j<=N+1; j++){
        x[i][j] = (double) ((i+j)%3) - 0.9999;
     }
  }
  printf("\nMemory allocation done successfully\n");
  //start timer and calculation
  gettimeofday(&start, NULL);

 for(j=1; j<N+1; j++){
    for(i=1; i<N+1; i++ ){
      tmp = 0.2 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1]);
      y[i][j] = tmp;
      sum = sum + tmp;
    }
  }

  //stop timer and calculation
  gettimeofday(&end, NULL);
  delta = ((end.tv_sec-start.tv_sec)*1000000u + end.tv_usec-start.tv_usec)/1.e6;
  printf("\nThe total sum is: %lf\n", sum);
  //print time to completion
  printf("run time    = %fs\n", delta);
  return 0;
}
```

Loop 1

Loop 2

First example (5)

Example1: All steps (I will demo this example in a minute)

Parallel Programming MPI, OpenMP, CUDA	OpenMP (2)
Choose function	main (1)
Parallel region, loop, or section	loop (2)
Select loop	select third loop
Reduction	yes
Reduction variable	sum (3)
Reduction operation	addition (1)
Dependency analysis	select (2)
Single thread	no
Another loop	no
Printing	no

First example (6)

Example1: On screen demo

First example (7)

Example1: Parallelized code

`rose_example1_OpenMP.c`

Let's compile and run it

Compile: `icc -qopenmp rose_example1_OpenMP.c`

Select number of threads: `export OMP_NUM_THREADS=4`
Execute: `./a.out`

First example (8)

Example1: Let's check the performance

Run the code with different numbers of threads and report the timing

```
export OMP_NUM_THREADS=1;   ./a.out
export OMP_NUM_THREADS=2;   ./a.out
export OMP_NUM_THREADS=4;   ./a.out
export OMP_NUM_THREADS=8;   ./a.out
export OMP_NUM_THREADS=16;  ./a.out
export OMP_NUM_THREADS=32;  ./a.out
```

First example (8)

Example1: Let's check the performance

Run the code with different numbers of threads and report the timing

```
export OMP_NUM_THREADS=1;   ./a.out        38.3
export OMP_NUM_THREADS=2;   ./a.out        19.5
export OMP_NUM_THREADS=4;   ./a.out        16.9
export OMP_NUM_THREADS=8;   ./a.out        16.4
export OMP_NUM_THREADS=16;  ./a.out        11.0
export OMP_NUM_THREADS=32;  ./a.out         9.3
```

First example (9)

Example1: Let's inspect the parallel version of the code

1. Header file in all routines with OpenMP content
2. Parallel region
   1. Threads are spawned
   2. All code within the curly brackets **{}** is executed by all threads
3. Worksharing for following loop (j loop)
   1. Worksharing = every thread is executing a different chunk of the loop

```
#include <omp.h>
#include <stdio.h>
#include <sys/time.h>
#define N 30000

int main()
{
  int i;
  int j;
  double x[30002UL][30002UL];
  double y[30002UL][30002UL];
  double sum;
  double tmp;
//for timing the code section
  struct timeval start;
  struct timeval end;
  float delta;
  for (i = 0; i <= 30000 + 1; i++) {
    for (j = 0; j <= 30000 + 1; j++) {
      x[i][j] = (((double )((i + j) % 3)) - 0.9999);
    }
  }
  printf("\nMemory allocation done successfully\n");
//start timer and calculation
  gettimeofday(&start,0);

#pragma omp parallel default(none) shared(sum,x,y) private(j,i,tmp)
{

#pragma omp for reduction ( + :sum)
  for (j = 1; j < 30000 + 1; j++) {
    for (i = 1; i < 30000 + 1; i++) {
      tmp = (0.2 * ((((x[i][j] + x[i - 1][j]) + x[i + 1][j]) + x[i][j - 1]) + x[i][j + 1]));
      y[i][j] = tmp;
      sum = (sum + tmp);
    }
  }
}
…
```

1: OpenMP header file

This code is now OpenMP parallel

2: Parallel region

3: Worksharing

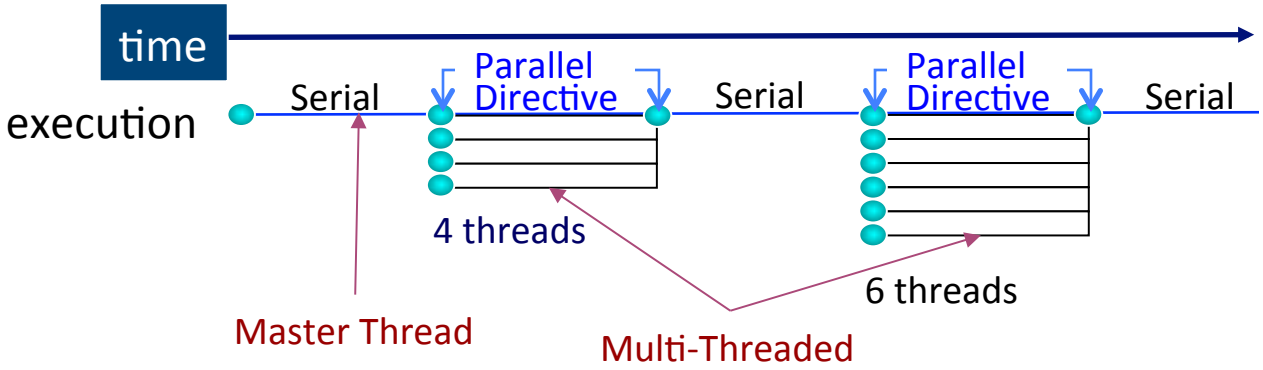
First example (10)

Example1: Look at the 2 OpenMP statements in the parallel code

```
#pragma omp parallel default(none) shared(sum,x,y) private(j,i,tmp)
{
#pragma omp for reduction ( + :sum)
for … {
…
}
```

Execution Model

- Programs begin as a single process: master thread
- Master thread executes in serial mode until the parallel construct is encountered
- After executing the statements in the parallel region, team threads synchronize and terminate (join) but master continues



OpenMP Syntax

Compiler directive syntax:

```
#pragma omp construct [clause [[,]clause]…]        C/C++
!$omp construct [clause [[,]clause]…]              F90
```

Example

Fortran

```
print*,"serial"

!$omp  parallel num_threads(4)
     …
!$omp end parallel

print*,"serial"
```

C/C++

```
printf("serial\n");

#pragma omp parallel num_threads(4)
{
      …
}

printf("serial\n");
```

Parallel Region & Worksharing

Use OpenMP directives to specify Parallel Region & Worksharing constructs

```
parallel


end parallel
```

Code block	Each Thread Executes
do / for	Worksharing
sections	Worksharing
single	Worksharing (one thread)

Sentinels "!$omp" and "#pragma omp") not shown here

Work-sharing Directives assign threads to units of work.
There is an implied barrier at the end of a worksharing construct!

C/C++

Worksharing: Loop

```
1  #pragma omp parallel
2  {
3    #pragma omp for
4    for (i=0; i<N; i++)
5    {
6      a[i] = b[i] + c[i];
7    }
8  }
```

Or

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

Line 1 Team of threads formed (parallel region).

Line 3-7 Loop iterations are split among threads.
implied barrier at }

Each loop iteration must be independent of other iterations.

First example (11)

Example1: How about the clauses in the 'omp parallel' statement?

```
#pragma omp parallel default(none) shared(sum,x,y) private(j,i,tmp)
{
…
}
```

Private variables are used to avoid race conditions

Every thread needs a private copy of
- loop indices i and j
- Scalar variable tmp

The input and output arrays are shared
- arrays: x and y

Every thread is writing to a different
- array element

```
#pragma omp parallel default(none) shared(x,y) private(j,i,tmp)
{

#pragma omp for reduction ( + :sum)
  for (j = 1; j < 30000 + 1; j++) {
    for (i = 1; i < 30000 + 1; i++) {
      tmp = (0.2 * ((((x[i][j] + x[i - 1][j]) + x[i + 1][j]) + x[i][j - 1]) + x[i][j + 1]));
      y[i][j] = tmp;
      sum = (sum + tmp);
    }
  }
}
…
```

Private Data Example

- In the following loop, each thread needs its own private copy of temp
- If temp were shared, the result would be unpredictable since each thread would be writing/reading to/from the same memory location

```
#pragma omp parallel for shared(a,b,c,n) private(temp,i)
for (i=0; i<n; i++){
    temp = a[i] / b[i];
    c[i] = temp + cos(temp);
}
```

- A **lastprivate(temp)** clause will copy the last loop(stack) value of temp to the (global) temp storage when the parallel for is complete.
- A **firstprivate(temp)** would copy the global temp value to each stack's temp.

First example (12)

Example1: How about the clauses in the 'omp for' statement?

```
#pragma omp for reduction (+:sum)
for …
{
…
}
```

Reduction variables store the local results of each thread. The local results are combined with a reduction operation to produce a global result.

A reduction is performed in the loop

Variable sum is updated by all threads

```
#pragma omp parallel default(none) shared(x,y) private(j,i,tmp)
{

#pragma omp for reduction ( + :sum)
  for (j = 1; j < 30000 + 1; j++) {
    for (i = 1; i < 30000 + 1; i++) {
      tmp = (0.2 * ((((x[i][j] + x[i - 1][j]) + x[i + 1][j]) + x[i][j - 1]) + x[i][j + 1]));
      y[i][j] = tmp;
      sum = (sum + tmp);
    }
  }
}
…
```

C/C++ Reduction

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
float asum=0.0, aprod=1.0;

#pragma omp parallel for reduction(+:asum) reduction(*:aprod)
  for (i=0; i<n; i++){
    asum  = asum  + a[i];
    aprod = aprod * a[i];
  }
```

Each thread has a private **asum** and **aprod**, initialized to the operator's identity

- **After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction**

First example (13)

Example1: Environment variable OMP_NUM_THREADS

```
export OMP_NUM_THREADS=4; ./a.out
```

OMP_NUM_THREADS sets the default number of threads

There are other ways to change the number of threads

Function call within code: `omp_set_num_threads(8)`

Clause at the omp parallel statement

```
#pragma omp parallel numthreads(8)
```

OpenMP Environment Variables

variable	description
OMP_NUM_THREADS=*integer*	Set to default no. of threads to use
OMP_SCHEDULE="*schedule-type[, chunk_size]*"	Sets "runtime" in loop schedule clause: "…omp for/do schedule(runtime)"
OMP_DISPLAY_ENV=*anyvalue*	Prints runtime environment at beginning of code execution.

[…] = optional

OpenMP Runtime Library Routines

function	description
omp_get_num_threads()	Number of threads in team, N
omp_get_thread_num()	Thread ID {0 -> N-1}
omp_get_num_procs()	Number of machine CPUs
omp_in_parallel()	True if in parallel region & multiple thread executing
omp_set_num_threads(#)	Set the number of threads in the team

C/C++

NUM_THREADS clause

- Use the NUM_THREADS clause to specify the number of threads to execute a parallel region

```
#pragma omp parallel num_threads(scalar int expression)
{
    <code block>
}
```

where scalar integer expression must evaluate to a positive integer

- num_threads() supersedes the number of threads specified by the OMP_NUM_THREADS environment variable or that set by the OMP_SET_NUM_THREADS function

Second example (1)

Let's start with example 2

- `$ cd ..`
- `$ cd exercise2`
- `$ ls -al`

- There is a Fortran file and a C file
- For now IPT only works with C/C++ (but you can look at the Fortran source, if that is more convenient)

If your idev session has expired:
Look at previous slides and start a new session
Do not forget to source the runBeforeIPT.sh script, etc.

Second example (2)

Continue with example 2

- Before using IPT inspect the code (we will do this together here)
- Use one of these: ‘more’, ‘vi’, or ‘emacs’

```
$ more example2.c
$ vi example2.c
$ emacs example2.c
```

IPT will ask you all the usual questions

- Which loop should IPT parallelize?
- Is there a reduction?

Second example (3)

Can you guess why this loop cannot be so easily parallelized?

```
for(j=1; j<N+1; j++){
   for(i=1; i<N+1; i++ ){
      tmp[i][j] = 0.167 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1] + y[i+1][j]);
      y[i][j] = tmp [i][j];
      sum = sum + tmp[i][j];
   }
 }
```

Second example (3)

But example 2 requires some code modifications

- Inside the loop, $y_{i,j}$ is updated (same as in example 1)
- However, the right-hand side refers also to $y_{i+1,j}$
- Loops can be parallelized easily when loop iterations are independent
  - e.g., when you can execute the loop iterations in any order
- Question: Would you get the same result when iterating through the loop backwards?

```
for(j=1; j<N+1; j++){
   for(i=1; i<N+1; i++ ){
      tmp[i][j] = 0.167 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1] + y[i+1][j]);
      y[i][j] = tmp [i][j];
      sum = sum + tmp[i][j];
   }
 }
```

Second example (4)

Code modifications

- In this loop the loop iterations are not independent
- Solution: Create 2 loop nests
  - The first one to calculate the temporary array (tmp)
  - The second one to copy tmp into y and to calculate the sum
- Then parallelize both loops separately

```
 for(j=1; j<N+1; j++){
    for(i=1; i<N+1; i++ ){
       tmp[i][j] = 0.167 * (x[i][j] + x[i-1][j] + x[i+1][j] + x[i][j-1] + x[i][j+1] + y[i+1][j]);
}
}
 for(j=1; j<N+1; j++){
    for(i=1; i<N+1; i++ ){
       y[i][j] = tmp [i][j];
       sum = sum + tmp[i][j];
}
}
```

Second example (5)

Example 2: Serial source code

Loop 2: Stencil update

1. Split the loop into 2
   - This is done by you, not by IPT
2. Run the code through IPT
3. Parallelize both loops

```
#include <stdio.h>
#include <sys/time.h>


#define N 30000

int main(){
  int i, j;
  double x[N+2][N+2], y[N+2][N+2], tmp[N+2][N+2];
  double sum=0;

  //for timing the code section
  struct timeval start,end;
  float delta;

  for(i=0; i <= N+1; i++){
     for(j=0; j <= N+1; j++){
         x[i][j] = (double) ((i+j)%3) - 0.9999;
         y[i][j]= x[i][j] + 0.0001;
     }
  }

  //start timer and calculation
  gettimeofday(&start, NULL);

  for(j=1; j<N+1; j++){
    for(i=1; i<N+1; i++ ){
       tmp[i][j] = 0.167 * (x[i][j] + x[i-1][j] + x[i+1][j] +
                            x[i][j-1] + x[i][j+1] + y[i+1][j]);
       y[i][j] = tmp [i][j];
       sum = sum + tmp[i][j];
    }
  }
  //stop timer and calculation
  gettimeofday(&end, NULL);
  delta = ((end.tv_sec-start.tv_sec)*1000000u +
                                    end.tv_usec-start.tv_usec)/1.e6;
  printf("\nThe total sum is: %lf\n", sum);
  //print time to completion
  printf("run time    = %fs\n", delta);
  return 0;
}
```

Loop 2

Second example (8)

Example 2: Let's check the performance

Run the code with different numbers of threads and report the timing

```
export OMP_NUM_THREADS=1;   ./a.out
export OMP_NUM_THREADS=2;   ./a.out
export OMP_NUM_THREADS=4;   ./a.out
export OMP_NUM_THREADS=8;   ./a.out
export OMP_NUM_THREADS=16;  ./a.out
export OMP_NUM_THREADS=32;  ./a.out
```

Other ways to avoid race conditions

OpenMP constructs for synchronization

- Critical and atomic
- Use this for code that is executed by all threads, but only by one thread at a time
- Example: when all threads update a variable

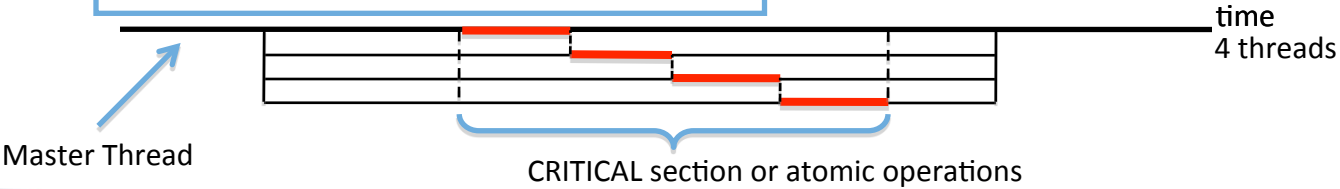
C/C++

Synchronization: Critical/Atomic Directives

- When each thread must execute a section of code serially the region must be marked with critical/end critical directives
- Use the #pragma omp atomic directive for simple cases: can use hardware support

```
#pragma omp parallel shared(sum,x,y)
{...
   #pragma omp critical

      update(x);
      update(y);
      sum=sum+1;
   }
....
}
```

```
#pragma omp parallel shared(sum)
{...

   #pragma omp atomic
   sum=sum+1
...
}
```



Atomic has read, write, update, capture clauses.

Other ways to avoid race conditions

OpenMP constructs for synchronization

- Critical and atomic
- Use this for code that is executed by all threads, but only by one thread at a time
- Example: when all threads update a variable

- Single and atomic
- Use this for code that is executed only once by one thread
- Example: When you initialize a variable inside a parallel region

Single Construct

Single: Any single thread executes the construct.

Since Single is worksharing there is an implied barrier.

Fortran

```
!$omp parallel private(id)
  !$omp single
    x = 1
  !$omp end single
  call foo(x)
!$omp end parallel
```

C/C++

```
#pragma omp parallel
private(id)
{
  #pragma omp single
  x = 1;

  foo(x);
}
```

Master Construct

Master: Only the master executes the construct.

There is no implied barrier for this construct.

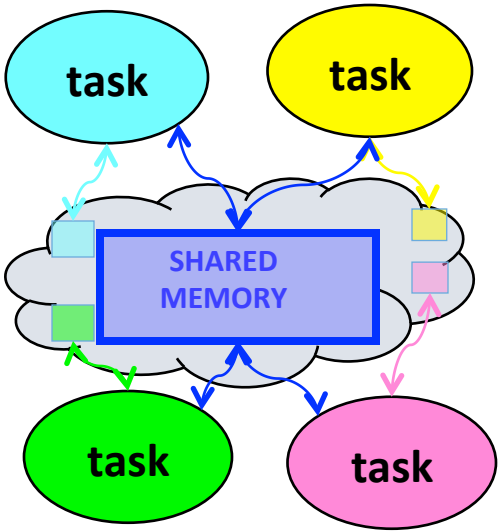
Fortran

```
!$omp parallel private(id)
  !$omp master
    x = 1
  !$omp end master
  call foo(x)
!$omp end parallel
```

C/C++

```
#pragma omp parallel
private(id)
{
  #pragma omp master
  x = 1;

  foo(x);
}
```

Recap: Why are there race conditions



Private
Memory

- Threads Execute on Cores/HW-threads
- In a parallel region, team threads are assigned (tied) to implicit tasks to do work. Think of tasks and threads as being synonymous.
- Tasks by “default” share memory declared in scope before a parallel region.
- Data: shared or private
  - Shared data: accessible by all tasks
  - Private data: only accessible by the owner task

Recap OpenMP

Parallel execution, worksharing, and avoiding race conditions

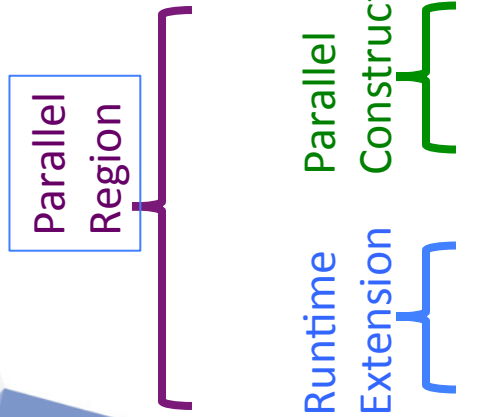
- Start a parallel region to launch threads
  - Code inside the parallel region is executed by all threads
  - This is just replicated work, unless you do some worksharing
- Use worksharing constructs (loop) to divide work among threads
  - This is where the speedup comes from
- Avoid race conditions
  - Make variables private (see example 1)
    - All threads will have a private copy to modify
  - Use OpenMP reduction for reduction operations (see example 1)
  - Modify code to create loops with independent loop iterations (see example 2)
  - Use critical/atomic to update a variable 'one thread at a time'
    - All threads are executing the code
  - Use single/master for serial code inside the parallel region
    - Only one thread is executing the code

LAB CONTENT WILL BE ADDED HERE

FOR FORTRAN USERS
and backups

OpenMP Directive Scope

- construct – the lexical extent of executable directive
- region – all code encountered in a construct (construct + routines)



Fortran

```
program ex1
use  omp_lib

!$omp parallel
   call foo(omp_get_thread_num())
!$omp end parallel

end program ex1

subroutine foo(id)
  write( *, '(" thrd_id ",i0.3)' ),id
end subroutine
```

C/C++

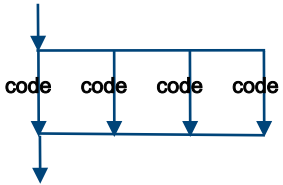
```
#include "omp.h"
void foo(int);
int main(){

  #pragma omp parallel
    { foo(omp_get_thread_num()); }


void foo(int id){
  printf("thrd_id %0.3d\n",id);
}
```

OpenMP Worksharing: Loops

Replicated : Work blocks are executed by all threads.
Work-Sharing : Work is divided among threads.

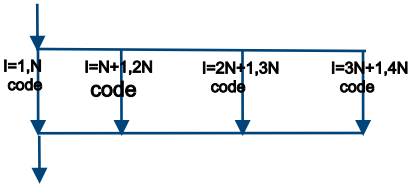
4 Threads

```
!OMP PARALLEL
  {code}
!OMP END PARALLEL
```



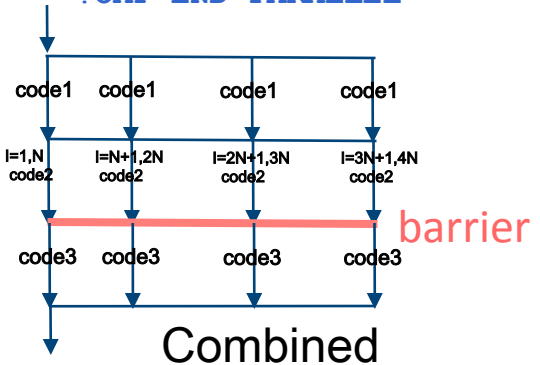
Replicated

```
!OMP PARALLEL
!OMP DO
  do I = 1,N*4
    {code}
  end do
!OMP END PARALLEL DO
```



Work-Sharing

```
!OMP PARALLEL
  {code1}
!OMP DO
  do I = 1,N*4
    {code2}
  end do
!OMP END DO
  {code3}
!OMP END PARALLEL
```

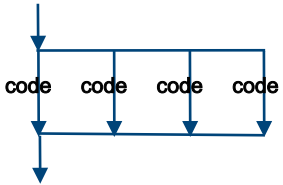


Combined

OpenMP Worksharing: Sections

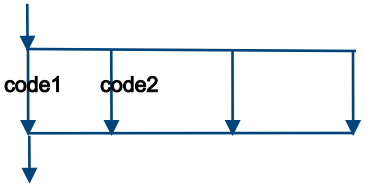
4 Threads

```
#pragma PARALLEL
  {code}
```



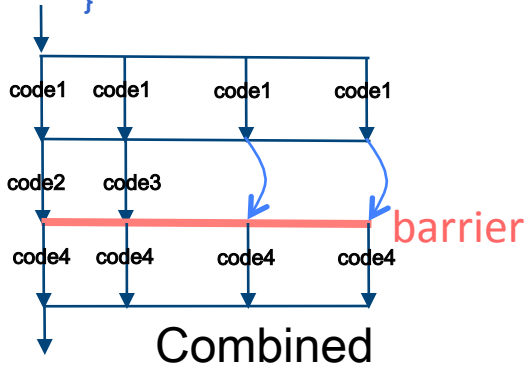
Replicated

```
#pragma PARALLEL sections
#pragma section
      {code1}
#pragma section
      {code2}
!OMP END PARALLEL sections
```



Work-Sharing

```
#pragma PARALLEL
{
    {code1}
    #pragma sections
    {
        #pragma section
          {code2}
        #pragma section
          {code3}
    }
    {code4}
}
```



Combined

Memory Model and Synchronization

- Every thread has access to shared memory.
  When multiple threads are created all threads share the same address space.

- Simultaneous updates to shared memory can create a *race condition*. Results change with different thread scheduling.

- Use mutual exclusion directives to avoid race conditions - but don't use too many because this will serialize performance.

- Barriers are used to synchronize threads.

- A flush exists at barriers, to make make local changes to data (writes) seen by all threads.

OpenMP Directive Scope

- OpenMP directives are comments/pragmas in source:

  F90 : **!$OMP** free-format*

  C/C++ : directives begin with the **# pragma omp** sentinel

Parallel regions are marked by enclosing parallel directives

  F90 : parallel … **end** parallel

  C/C++ : Enclosed in block **{ …}**, or single statement

- Work-sharing loops are marked by parallel do/for

* Fortran Fixed Format:
!$OMP, **C$OMP** or ***$OMP**

Fortran

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

!$omp parallel
  call foo(...)
!$omp end parallel
```

C/C++

```
# pragma omp parallel
  {...
  }

# pragma omp parallel
for
  foo(...);
```

C/C++

Parallel Regions

```
1   #pragma omp parallel
2   {
3       code block
4       work(…);
5   }
```

Line 1 Team of threads formed at parallel region

Lines 3-4 Each thread executes code block and subroutine calls.

No branching (in or out) in a parallel region

Line 5 All threads synchronize at end of parallel region (implied barrier)

Use the thread number to divide work among threads

F90

Parallel Regions

```
1   !$OMP PARALLEL
2       code block
3       call work(…)
4   !$OMP END PARALLEL
```

Line 1 Team of threads formed at parallel region.

Lines 2-3 Each thread executes code block and subroutine calls.

No branching (in or out) in a parallel region.

Line 4 All threads synchronize at end of parallel region (implied barrier).

Use the thread number to divide work among.

C/C++

Parallel Region & Number of Threads

- For example, to create a 4-thread Parallel region.
- export OMP_NUM_THREADS=4; a.out

```
double A[1000];
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID, A);
}
```

In C++ a local variables are private to the thread…

- Each thread executes the code within the structured block
- Thread numbers range from 0 to Nthreads-1
- Each thread calls foo(ID,A) with a different `ID {= 0 to 3}`

F90

Parallel Region & Number of Threads

- For example, to create a 4-thread Parallel region.
- export OMP_NUM_THREADS=4; a.out

```
real ::  A(1000; integer :: ID
!$omp parallel

   ID = omp_get_thread_num()
   call foo(ID, A)
!$omp end parallel
```

But we need to make id Private to the thread (more later)

- Each thread executes the code within the structured block
- Thread numbers range from 0 to Nthreads-1
- Each thread calls foo(ID,A) with a different `ID {= 0 to 3}`

C/C++

Work sharing: Loop

```
1  #pragma omp parallel
2  {
3    #pragma omp for
4    for (i=0; i<N; i++)
5    {
6      a[i] = b[i] + c[i];
7    }
8  }
```

Or

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

Line 1 Team of threads formed (parallel region).

Line 3-7 Loop iterations are split among threads.
implied barrier at }

Each loop iteration must be independent of other iterations.

F90

Work sharing: Loop

```
1   !$OMP PARALLEL
2      !$OMP DO
3      do i=1,N
4         a(i) = b(i) + c(i)
5      enddo
6   !$OMP END PARALLEL
```

Line 1 Team of threads formed (parallel region).

Line 3-4 Loop iterations are split among threads by DO construct.

Line 5 !$OMP END DO is optional after the enddo.

Line 5 Implied barrier at enddo.

Each loop iteration must be independent of other iterations.

OpenMP Combined Constructs

Combining '`parallel`' with '`do`' or '`for`'

Fortran

```
!$omp parallel
  !$omp do
    do ...
    end do
  !$omp end do
!$omp end parallel

!$omp parallel do
  do ...
  end do
```

! *

C/C++

```
# pragma omp parallel
{
  #pragma omp for
  for (...){...}
}


# pragma omp parallel for
  for(){...}
```

* Since Fortran DO has a block structure "!$omp end do" Is not required

C/C++

Work-Sharing: Sections

```
1  #pragma omp parallel sections
2  {
3    #pragma omp section
4      {
5        work_1();
6      }
7    #pragma omp section
8      { work_2(); }
9  }
```

Line 1 Team of threads formed (parallel region).

Line 3-8 One thread is working on each section.

Line 9 End of parallel sections with an implied barrier.

Scales only to the number of sections.

F90

Work-Sharing: Sections

```
 1  !$omp parallel sections
 2
 3    !$omp section
 4
 5      call work_1()
 6
 7    !$omp section
 8      call work_2()
 9
10  !$omp end parallel sections
```

Line 1 Team of threads formed (parallel region).

Line 3-9 One thread is working on each section.

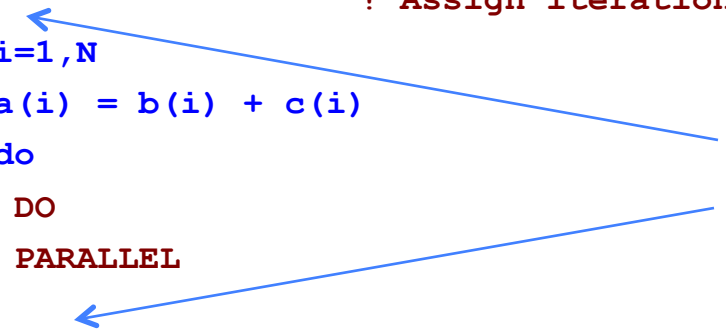
Line 10 End of parallel sections with an implied barrier.

Scales only to the number of sections.

Shorthand (again)

```
1  !$OMP PARALLEL DO
2        do i=1,N
3           a(i) = b(i) + c(i)
4        enddo
5  !$OMP END PARALLEL DO
```

```
1   !$OMP PARALLEL              ! Create team of threads
 2   !$OMP DO                    ! Assign iterations to threads
3         do i=1,N
4            a(i) = b(i) + c(i)
5         enddo
6   !$OMP END DO
7   !$OMP END PARALLEL
```



But, how does all this work
if we need other code/work here?

OpenMP Clauses

Clauses control the behavior of an OpenMP directive:

1. Schedule for for/do worksharing (Static, Dynamic, Guided, etc.)
2. Data scoping (Private, Shared, Default)
3. Initialization (e.g. COPYIN, FIRSTPRIVATE)
4. Parallelize a region or not (IF(logic_expression))
5. Number of threads used (NUM_THREADS)

Schedule Clause for loop worksharing

schedule(static)

Each CPU receives one set of contiguous iterations

```
!$omp do schedule(...)
   do i=1,128
     A(i)=B(i)+C(i)
   enddo
```

schedule(static, C)

Iterations are divided round-robin fashion in chunks of size C

schedule(dynamic, C)

Iterations handed out in chunks of size C as CPUs become available

schedule(guided, C)

Each of the iterations are handed out in pieces of exponentially decreasing size, with C minimum number of iterations to dispatch each time

schedule (runtime)

Schedule and chunk size taken from the OMP_SCHEDULE environment variable

Example - schedule(static,16), threads = 4

```
!$omp parallel do schedule(static,16)
      do i=1,128
       A(i)=B(i)+C(i)
      enddo
```

thread0:
`do i=1,16`
`  A(i)=B(i)+C(i)`
`enddo`
`do i=65,80`
`  A(i)=B(i)+C(i)`
`enddo`
	thread2:
	`do i=33,48`
	`  A(i)=B(i)+C(i)`
	`enddo`
	`do i = 97,112`
	`  A(i)=B(i)+C(i)`
	`enddo`

thread1:
`do i=17,32`
`  A(i)=B(i)+C(i)`
`enddo`
`do i = 81,96`
`  A(i)=B(i)+C(i)`
`enddo`
	thread3:
	`do i=49,64`
	`  A(i)=B(i)+C(i)`
	`enddo`
	`do i = 113,128`
	`  A(i)=B(i)+C(i)`
	`enddo`

Comparison of Scheduling Options

name	type	chunk	chunk size	chunk #	static or dynamic	compute overhead
partitioned	static	no	N/P	P	static	lowest
interleaved	static	yes	C	N/C	static	low
simple dynamic	dynamic	optional	C	N/C	dynamic	medium
guided	guided	optional	decreasing from N/P*	no fewer than C	dynamic	'lower than simple dynamic'
runtime	runtime	no	varies	varies	varies	varies

*Decreases as (N-unassigned)/P

OpenMP Data Environment

Clauses control the data-sharing attributes of variables within a parallel region:

shared, private, reduction (firstprivate, lastprivate)

Default variable scope (in parallel region):

1. Variables declared in main/program (C/F90) are shared by default
2. Global variables are shared by default
3. Automatic variables within subroutines called from within a parallel region are private (reside on a stack private to each thread)
4. Loop index of worksharing loops are private.
5. Default scoping rule can be changed with `default` clause

C/C++

Private & Shared Data

shared - Variable is shared (seen) by all threads

private - Each thread has a private instance (copy) of the variable

Defaults: The for-loop index is private, all other variables are shared

```
#pragma omp parallel for shared(a,b,c,n) private(i)
      for (i=0; i<n; i++){
            a[i] = b[i] + c[i];
         }
```

OK to be explicit;
but not necessary.

All threads have access to the same storage areas for a, b, c, and n, but each loop has its own private copy of the loop index, i

F90

Private & Shared Data

shared - Variable is shared (seen) by all threads

private - Each thread has a private instance (copy) of the variable

Defaults: The for-loop index is private, all other variables are shared

```
!$omp parallel do shared(a,b,c,n) private(i)
      do i = 1,n
            a(i) = b(i) + c(i)
      enddo
```

OK to be explicit;
but not necessary.

All threads have access to the same storage areas for a, b, c, and n, but each loop has its own private copy of the loop index, i

C/C++

Private Data Example

- In the following loop, each thread needs its own private copy of temp
- If temp were shared, the result would be unpredictable since each thread would be writing/reading to/from the same memory location

```
#pragma omp parallel for shared(a,b,c,n) private(temp,i)
for (i=0; i<n; i++){
    temp = a[i] / b[i];
    c[i] = temp + cos(temp);
}
```

- A **lastprivate(temp)** clause will copy the last loop(stack) value of temp to the (global) temp storage when the parallel for is complete.
- A **firstprivate(temp)** would copy the global temp value to each stack's temp.

F90

Private Data Example

- In the following loop, each thread needs its own private copy of temp
- If temp were shared, the result would be unpredictable since each thread would be writing/reading to/from the same memory location

```
!$omp parallel for shared(a,b,c,n) private(temp,i)
do i = 1,n
   temp = a(i) / b(i)
   c(i) = temp + cos(temp)
endo
```

- A **lastprivate(temp)** clause will copy the last loop(stack) value of temp to the (global) temp storage when the parallel DO is complete.
- A **firstprivate(temp)** would copy the global temp value to each stack's temp.

C/C++ Reduction

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
float asum=0.0, aprod=1.0;

#pragma omp parallel for reduction(+:asum) reduction(*:aprod)
  for (i=0; i<n; i++){
    asum  = asum  + a[i];
    aprod = aprod * a[i];
  }
```

Each thread has a private **asum** and **aprod**, initialized to the operator's identity

- **After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction**

F90

Reduction

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
real asum=0.0, aprod=1.0
!$omp parallel do reduction(+:asum) reduction(*:aprod)
   do i = 1,n
      asum  = asum  + a(i)
      aprod = aprod * a(i)
   enddo
 print*, asum, aprod
```

Each thread has a private **asum** and **aprod**, initialized to the operator's identity

- **After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction**

Synchronization

- Synchronization is used to impose order constraints and to protect access to shared data
- High-Level Synchronization
  - critical
  - atomic
  - barrier
  - ordered
- Low-Level Synchronization
  - locks

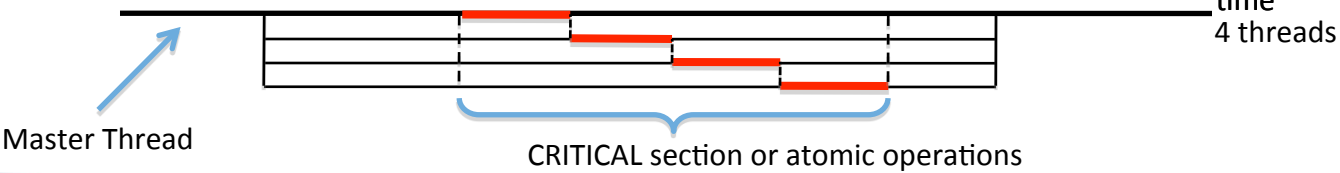
Synchronization: Critical/Atomic Directives

- When each thread must execute a section of code serially the region must be marked with critical/end critical directives
- Use the #pragma omp atomic directive for simple cases: can use hardware support

```
#pragma omp parallel shared(sum,x,y)
{...
   #pragma omp critical

      update(x);
      update(y);
      sum=sum+1;
   }
....
}
```

```
#pragma omp parallel shared(sum)
{...

   #pragma omp atomic
   sum=sum+1
...
}
```

Atomic has read, write, update, capture clauses.



F90

Synchronization: Critical/Atomic Directives

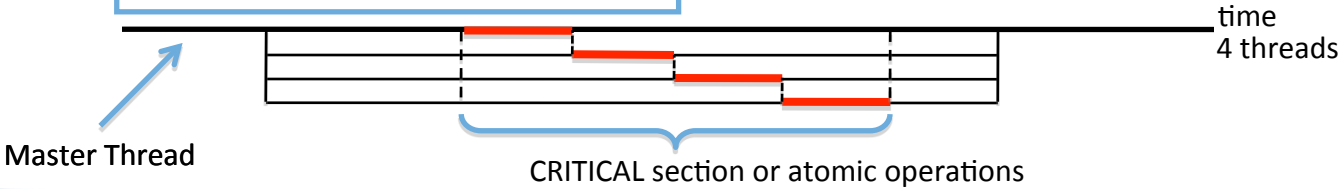
When each thread must execute a section of code serially the region must be marked with critical/end critical directives

Use the !$ omp atomic directive for simple cases: can use hardware support

```
!$omp parallel shared(sum,x,y)
 ...

   !$omp critical
     update(x);
     update(y);
     sum=sum+1;
   !$omp end critical
....
!$omp end parallel
```

```
!$omp parallel shared(sum)
...

      !$omp atomic
      sum=sum+1;
...
!$omp end parallel
```

Atomic has read, write, update, capture clauses.



C/C++

Synchronization: Barrier

Barrier: Each thread waits until all threads arrive and a flush occurs

```
#pragma omp parallel shared (A, B, C) private(id)
{
   id=omp_get_thread_num();
   A[id] = big_calc1(id);
   #pragma omp barrier
   #pragma omp for
   for(i=0;i<N;i++){
         C[i]=big_calc3(i,A);
   }
   #pragma omp for nowait
   for(i=0;i<N;i++){
         B[i]=big_calc2(C, i);
   }
   A[id] = big_calc4(id);
}
```

Implicit barrier

No implicit barrier due to nowait

Implicit barrier

F90

Synchronization: Barrier

Barrier: Each thread waits until all threads arrive and flush occurs

```
!$omp parallel shared (A, B, C) private(id)

   id=omp_get_thread_num()
   A(id) = big_calc1(id)
   !$omp barrier

   !$omp do
     do i = 1,N; C(i)=big_calc3(i,A); enddo
   !$omp end do
   !$omp do
     do i = 1,N; B(i)=big_calc2(C, i); enddo
   !$omp end do nowait
   A(id) = big_calc4(id);
!$omp end parallel
```

Implicit barrier

No implicit barrier due to nowait

Implicit barrier

Barriers

- The previous example could have been done with multiple parallel regions:
  1. There is an implicit barrier at the end of a parallel region.
  2. However, creating and recreating thread teams is expensive.

C/C++

Synchronization: Ordered

The ordered region executes in the sequential order

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:countVal)
for (i=0;i<N;i++){
  tmp = foo(i);
  #pragma omp ordered
  countVal+= consume(tmp);
}
```

F90

Synchronization: Ordered

The ordered region executes in the sequential order

```
!$omp parallel private (tmp)
!$omp do ordered reduction(+:countVal)
do i =1,N{
  tmp = foo(i)
  !$omp ordered
  countVal = countVal + consume(tmp)
end do
```

C/C++

NOWAIT

- When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.
- By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```
#pragma omp parallel
{
  #pragma omp for nowait
      {
         for (i=0; i<n; i++)
           {work(i);}
      }
  #pragma omp for schedule(guided,k)
      {
         for (i=0; i<m; i++)
           {x[i]=y[i]+z[i];}
      }
}
```

F90

NOWAIT

When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.

By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```
!$OMP PARALLEL
   !$OMP DO
      do i=1,n
         work(i)
      enddo
   !$OMP END DO NOWAIT
   !$OMP DO schedule(guided,k)
      do i=1,m
         x(i)=y(i)+z(i)
      enddo
   !$OMP END DO
!$OMP END PARALLEL
```

Runtime Library Routines

function	description
omp_get_num_threads()	Number of threads in team, N
omp_get_thread_num()	Thread ID {0 -> N-1}
omp_get_num_procs()	Number of machine CPUs
omp_in_parallel()	True if in parallel region & multiple thread executing
omp_set_num_threads(#)	Set the number of threads in the team

Environment Variables

variable	description
OMP_NUM_THREADS=*integer*	Set to default no. of threads to use
OMP_SCHEDULE="*schedule-type[, chunk_size]*"	Sets "runtime" in loop schedule clause: "…omp for/do schedule(runtime)"
OMP_DISPLAY_ENV=*anyvalue*	Prints runtime environment at beginning of code execution.

[…] = optional

OpenMP Wallclock Timers

```
real*8 :: omp_get_wtime,   omp_get_wtick()        (Fortran)
double    omp_get_wtime(), omp_get_wtick();       (C)
```

```
double t0, t1, dt, res;
...
t0 = omp_get_wtime();
<work>
t1 = omp_get_wtime();
dt = t1 - t0;

res = 1.0/omp_get_wtick();
printf("Elapsed time = %lf\n",dt);
printf("clock resolution = %lf\n",res);
```

C/C++

NUM_THREADS clause

- Use the NUM_THREADS clause to specify the number of threads to execute a parallel region

```
#pragma omp parallel num_threads(scalar int expression)
{
    <code block>
}
```

where scalar integer expression must evaluate to a positive integer

- num_threads() supersedes the number of threads specified by the OMP_NUM_THREADS environment variable or that set by the OMP_SET_NUM_THREADS function

F90

NUM_THREADS clause

- Use the NUM_THREADS clause to specify the number of threads to execute a parallel region

```
!$omp parallel num_threads(scalar integer expression)
    <code block>
!$omp end parallel
```

where scalar integer expression must evaluate to a positive integer

- num_threads() supersedes the number of threads specified by the OMP_NUM_THREADS environment variable or that set by the OMP_SET_NUM_THREADS function

F90 Mutual Exclusion: Lock Routines

When each thread must execute a section of code serially, locks provide a more flexible way of ensuring serial access than **CRITICAL** and **ATOMIC** directives

```
call OMP_INIT_LOCK(maxlock)
!$OMP PARALLEL SHARED(X,Y)
...
call OMP_set_lock(maxlock)
call update(x)
call OMP_unset_lock(maxlock)
...
!$OMP END PARALLEL
call OMP_DESTROY_LOCK(maxlock)
```

OpenMP 3.0

- First update to the spec since 2005
- Tasking: move beyond loops with generalized tasks and support complex and dynamic control flows
- Loop collapse: combine nested loops automatically to expose more concurrency
- Enhanced loop schedules: Support aggressive compiler optimizations of loop schedules and give programmers better runtime control over the kind of schedule used
- Nested parallelism support: better definition of and control over nested parallel regions, and new API routines to determine nesting structure

Tasks Parallelism

- Allows to parallelize irregular problems
- OpenMP had tasks internally: chunks of execution; with OpenMP
- Tasks depend on other tasks: dataflow
- You can declare many tasks, OpenMP executes them, satisfying dependencies

Task dependencies

```
1  ! In parallel single region here.  Single thread generates tasks
   ! Other threads of team execute tasks.
2  for (i=0; i<children; i++)
3  #pragma omp task
4    f(i)
5  #pragma omp taskwait
6  parent_function()
```

Children tasks are put in a queue can be executed in any order but before the parent

Tasks: Usage

Task Construct:

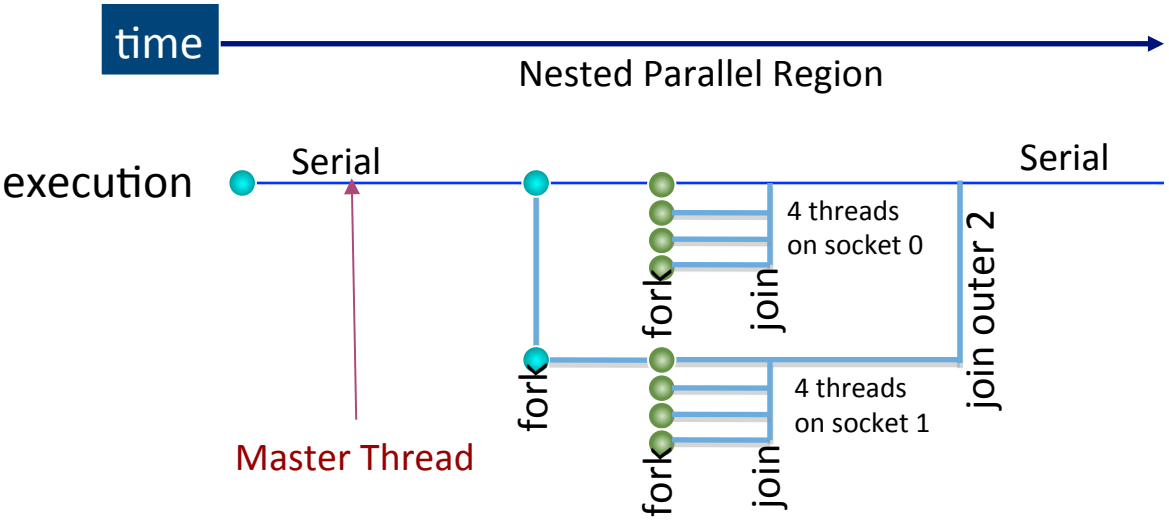
#pragma omp task [*clause*[[,]*clause*] ...]

structured-block

where clause can be

- Data scoping clauses
  shared (list), private (list), firstprivate (list), default(shared | none)
- Scheduling clauses
  untied
- Other clauses
  if (expression)

Loop Nesting



While OpenMP 3.0 supports nested parallelism, many implementations may ignore the nesting by serializing the inner parallel regions

Loop Collapse

- Allow collapsing of perfectly nested loops

- Will form a single loop and then parallelize it:

```
#pragma omp parallel do collapse(2)
for(i=0;i<n;i++){
   for(j=0;j<n;j++){
      .....
   }
}
```

F90

Loop Collapse

- Allow collapsing of perfectly nested loops

- Will form a single loop and then parallelize it:

```
!$omp parallel do collapse(2)
do i=1,n
   do j=1,n
      .....
   end do
end do
```

References

- http://www.openmp.org/
- *Parallel Programming in OpenMP*, by Chandra,Dagum, Kohr, Maydan, McDonald, Menon
- *Using OpenMP*, by Chapman, Jost, Van der Pas (OpenMP2.5)
- http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf
- http://webct.ncsa.uiuc.edu:8900/public/OPENMP/

FOR LABS

F90

Single Construct

Single: Any single thread executes the construct. Since Single is worksharing there is an implied barrier.

Fortran

```
!$omp parallel
private(id)

id=omp_get_thread_num()
   !$omp single
     x = 1
   !$omp end single
   call foo(x)
!$omp end parallel
```

C/C++

```
#pragma omp parallel
private(id)
{

id=omp_get_thread_num();
   #pragma omp single
   x = 1;
```

F90

Master Construct

Master: Only the master executes the construct.
There is no implied barrier for this construct.

race condition

race condition

Fortran

```
!$omp parallel
private(id)

id=omp_get_thread_num()
   !$omp master
     x = 1
   !$omp end master
   call foo(x)
!$omp end parallel
```

```
C/C++
#pragma omp parallel
private(id)
{

id=omp_get_thread_num();
   #pragma omp master
   x = 1;
```

Parallel Regions & Modes

There are two OpenMP “modes”

static mode

Fixed number of threads -- set in the **`OMP_NUM_THREADS`** env.

`Or the threads may be set by a function call (or clause) inside the code:`

```
omp_set_num_threads  runtime function
  num_threads(#)          clause
```

dynamic mode:

Number of threads can change under OS control from one parallel region to another using:

Note*: the user can only define the maximum number of threads, compiler can use a smaller number*

Default variable scoping (Fortran example)

```
Program Main
Integer, Parameter :: nmax=100
Integer :: n, j
Real*8  :: x(n,n)
Common /vars/ y(nmax)
...
n=nmax; y=0.0
!$OMP Parallel do
   do j=1,n
   call Adder(x,n,j)
   end do
...
End Program Main
```

```
Subroutine Adder(a,m,col)
Common /vars/ y(nmax)
SAVE array_sum
Integer :: i, m
Real*8  :: a(m,m)

do i=1,m
   y(col)=y(col)+a(i,col)
end do
array_sum=array_sum+y(col)

End Subroutine Adder
```

Default data scoping in Fortran (cont.)

Variable	Scope	Is use safe?	Reason for scope
n	shared	yes	declared outside parallel construct
j	private	yes	parallel loop index variable
x	shared	yes	declared outside parallel construct
y	shared	yes	common block
i	private	yes	parallel loop index variable
m	shared	yes	actual variable *n* is shared
a	shared	yes	actual variable *x* is shared
col	private	yes	actual variable *j* is private
array_sum	shared	no	declared with SAVE attribute