Programming with OpenMP*

Objectives

At the completion of this module you will be able to

- Thread serial code with basic OpenMP pragmas
- Use OpenMP synchronization pragmas to coordinate thread execution and memory access

Agenda

What is OpenMP?

Parallel Regions

Worksharing Construct

Data Scoping to Protect Data

Explicit Synchronization

Scheduling Clauses

Other Helpful Constructs and Clauses

What Is OpenMP*?

Compiler directives for multithreaded programming

Easy to create threaded Fortran and C/C++ codes

Supports data parallelism model

Incremental parallelism

Combines serial and parallel code in single source

What Is OpenMP*?

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

C$OMP parallel do shared(a, b, c)

call omp_test_lock(jlok)

call OMP_INIT

C$OMP MASTER

C$OMP SINGLE PRIV

C

C$OMP PARALLEL DO

ynamic"

C$OMP ORDERED

C$OMP PARALLEL

IONS

#pragma omp parallel for private(A, B)

!$OMP BARRIER

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

http://www.openmp.org

Current spec is OpenMP 2.5

250 Pages

(combined C/C++ and Fortran)

OpenMP* Architecture

Fork-join model

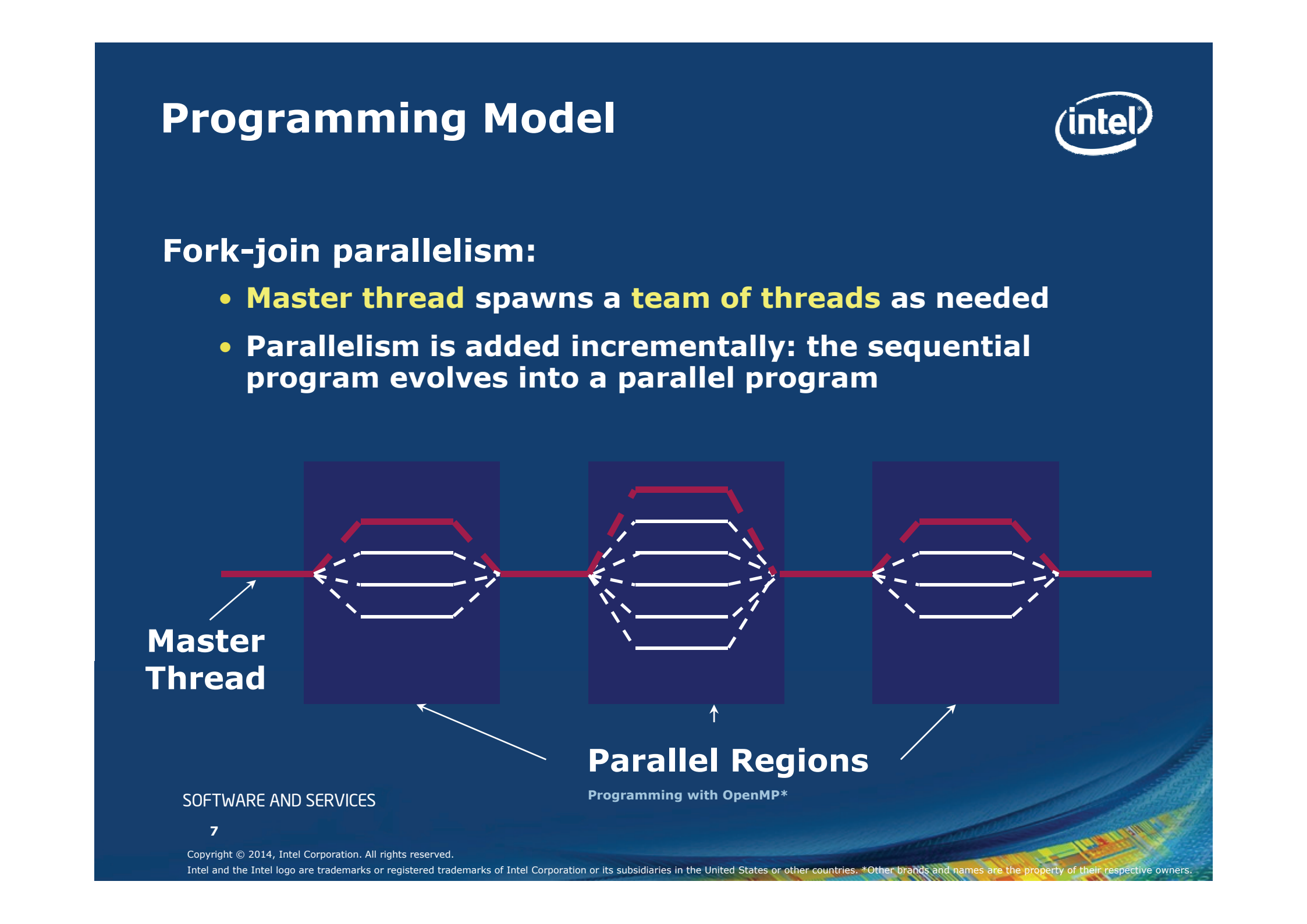
Work-sharing constructs

Data environment constructs

Synchronization constructs

Extensive Application Program Interface (API) for finer control

Programming Model



Fork-join parallelism:

- **Master thread** spawns a **team of threads** as needed
- Parallelism is added incrementally: the sequential program evolves into a parallel program

OpenMP* Pragma Syntax

Most constructs in OpenMP* are compiler directives or pragmas.

- For C and C++, the pragmas take the form:

```
#pragma omp construct [clause [clause]…]
```

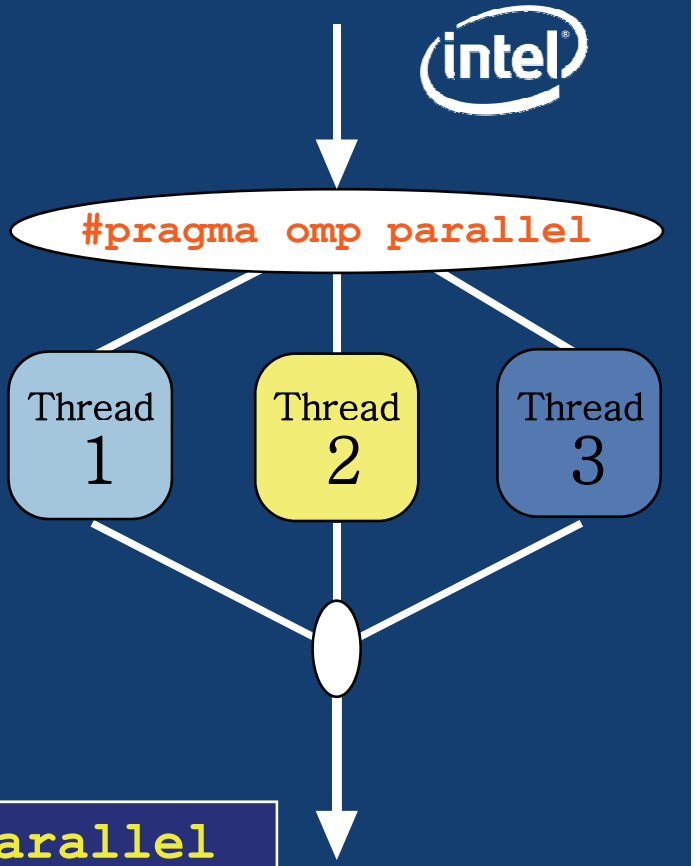
Parallel Regions

Defines parallel region over structured block of code

Threads are created as `'parallel'` pragma is crossed

Threads block at end of region

Data is shared among threads unless specified otherwise



C/C++ :

```
#pragma omp parallel
    {
        block
    }
```

How Many Threads?

Set environment variable for number of threads

```
set OMP_NUM_THREADS=4
```

There is no standard default for this variable

- Many systems:
  - # of threads = # of processors
  - Intel® compilers use this default

Activity 1: Hello Worlds

Modify the “Hello, Worlds” serial code to run multithreaded using OpenMP*

Work-sharing Construct

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

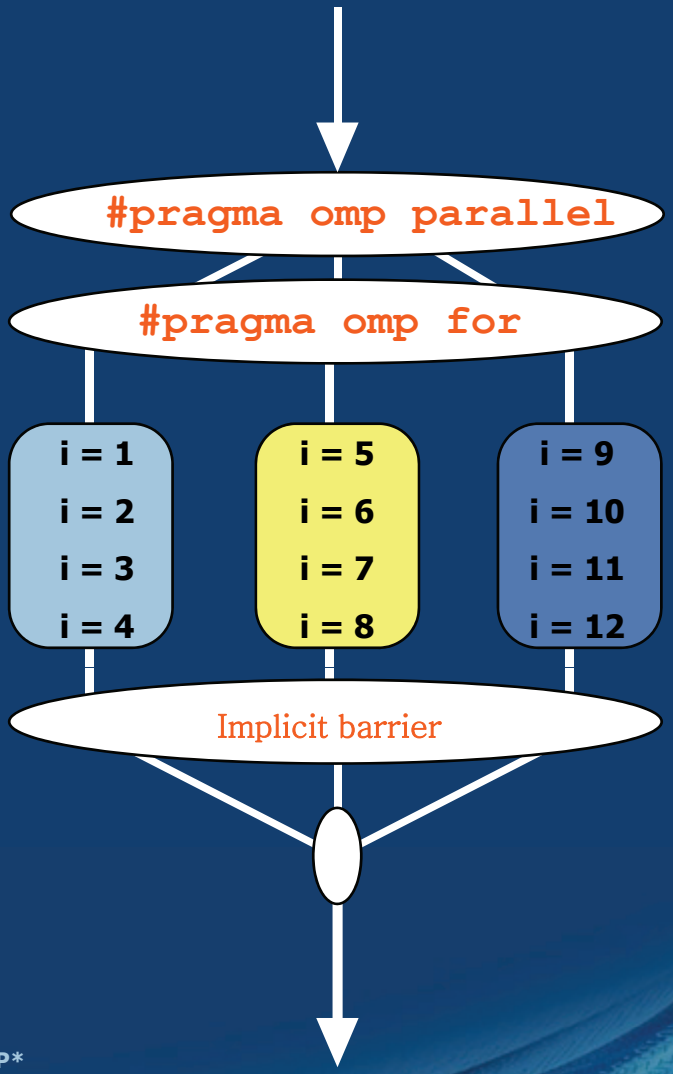
Splits loop iterations into threads

Must be in the parallel region

Must precede the loop

Work-sharing Construct

```
#pragma omp parallel
#pragma omp for
   for(i = 1, i < 13, i++)
      c[i] = a[i] + b[i]
```



Threads are assigned an independent set of iterations

Threads must wait at the end of work-sharing construct

Combining pragmas

These two code segments are equivalent

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
       res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
         res[i] = huge();
    }
```

Data Environment

OpenMP uses a shared-memory programming model

- Most variables are shared by default.
- Global variables are shared among threads
  - C/C++: File scope variables, static

Data Environment

But, not everything is shared...

- Stack variables in functions called from parallel regions are PRIVATE
- Automatic variables within a statement block are PRIVATE
- Loop index variables are private (with exceptions)
  - C/C+: The first loop index variable in nested loops following a **`#pragma omp for`**

Data Scope Attributes

The default status can be modified with

```
default (shared | none)
```

Scoping attribute clauses

```
shared(varname,…)
```

```
private(varname,…)
```

The Private Clause

Reproduces the variable for each thread

- Variables are un-initialized; C++ object is default constructed
- Any value external to the parallel region is undefined

```
void* work(float* c, int N) {
  float x, y; int i;
 #pragma omp parallel for private(x,y)
     for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
     }
}
```

Example: Dot Product

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

What is Wrong?

Protect Shared Data

Must protect access to shared, modifiable data

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
#pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
#pragma omp critical
     sum += a[i] * b[i];
   }
  return sum;
}
```

OpenMP* Critical Construct

```
#pragma omp critical [(lock_name)]
```

Defines a critical region on a structured block

Threads wait their turn –at a time, only one calls `consum()` thereby protecting RES from race conditions

Naming the critical construct `RES_lock` is optional

```
float RES;
#pragma omp parallel
{ float B;
#pragma omp for
  for(int i=0; i<niters; i++){
    B = big_job(i);
#pragma omp critical (RES_lock)
    consum (B, RES);
  }
}
```

OpenMP* Reduction Clause

```
reduction (op : list)
```

The variables in "*list*" must be shared in the enclosing parallel region

Inside parallel or work-sharing construct:

- A PRIVATE copy of each list variable is created and initialized depending on the "op"
- These copies are updated locally by threads
- At end of construct, local copies are combined through "op" into a single value and combined with the value in the original SHARED variable

Reduction Example

```
#pragma omp parallel for reduction(+:sum)
   for(i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
```

Local copy of *sum* for each thread

All local copies of *sum* added together and stored in "global" variable

C/C++ Reduction Operations

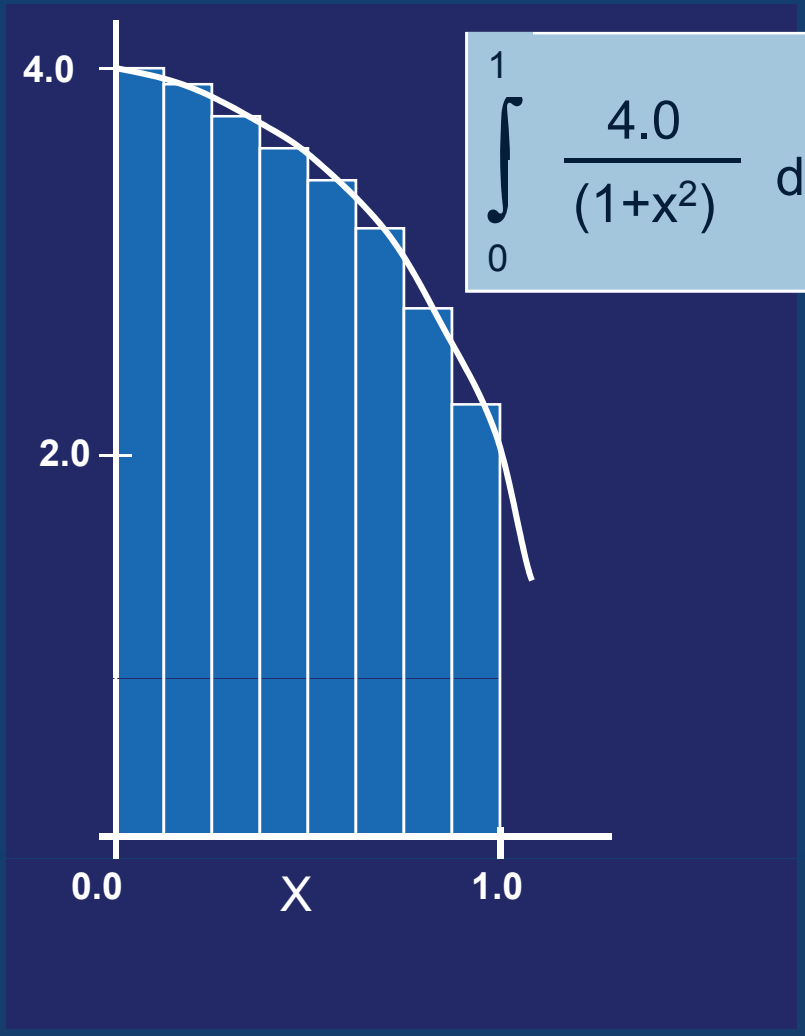
A range of associative operands can be used with reduction

Initial values are the ones that make sense mathematically

Operand	Initial Value
+	0
*	1
-	0
^	0

Operand	Initial Value
&	~0
|	0
&&	1
||	0

Numerical Integration Example



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx$$

```
static long num_steps=100000;
double step, pi;

void main()
{   int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

Activity 2 - Computing Pi

```
static long num_steps=100000;
double step, pi;

void main()
{   int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

Parallelize the numerical integration code using OpenMP

What variables can be shared?

What variables need to be private?

What variables should be set up for reductions?

Assigning Iterations

The schedule clause affects how loop iterations are mapped onto threads

`schedule(static [,chunk])`

- Blocks of iterations of size "chunk" to threads
- Round robin distribution

`schedule(dynamic[,chunk])`

- Threads grab "chunk" iterations
- When done with iterations, thread requests next set

`schedule(guided[,chunk])`

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than "chunk"

Which Schedule to Use

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Schedule Clause Example

```
#pragma omp parallel for schedule (static, 8)
    for( int i = start; i <= end; i += 2 )
    {
       if ( TestForPrime(i) )  gPrimesFound++;
    }
```

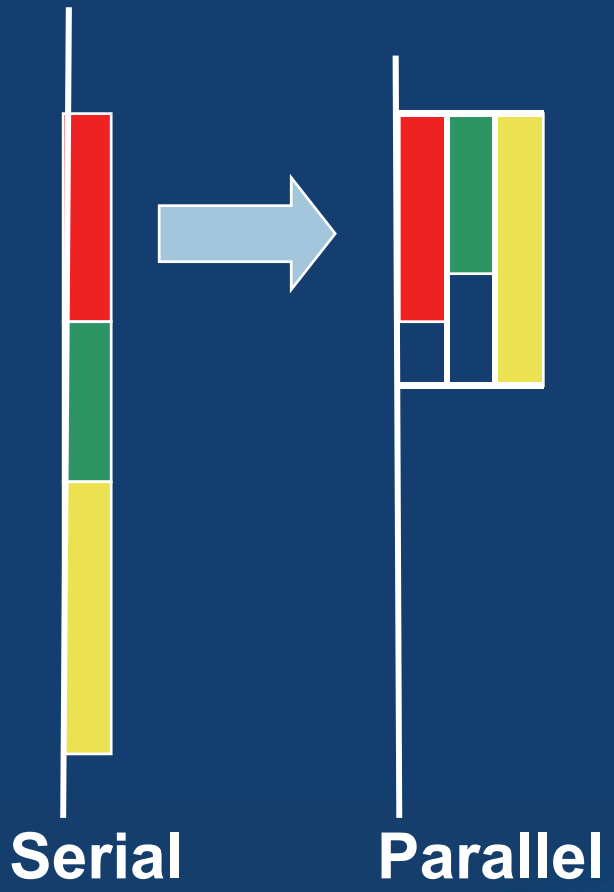
Iterations are divided into chunks of 8

- If start = 3, then first chunk is `i`={3,5,7,9,11,13,15,17}

Parallel Sections

Independent sections of code can execute concurrently

```
#pragma omp parallel sections
{
    #pragma omp section
    phase1();
    #pragma omp section
    phase2();
    #pragma omp section
    phase3();
}
```



Single Construct

Denotes block of code to be executed by only one thread

- First thread to arrive is chosen

Implicit barrier at end

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp single
   {
     ExchangeBoundaries();
   }  // threads wait here for single
   DoManyMoreThings();
}
```

Master Construct

Denotes block of code to be executed only by the master thread

No implicit barrier at end

```
#pragma omp parallel
{
   DoManyThings();
#pragma omp master
   {                    // if not master skip to next stmt
     ExchangeBoundaries();
   }
   DoManyMoreThings();
}
```

Implicit Barriers

Several OpenMP* constructs have implicit barriers

- `parallel`
- `for`
- `single`

Unnecessary barriers hurt performance

- Waiting threads accomplish no work!

Suppress implicit barriers, when safe, with the `nowait` clause

Nowait Clause

```
#pragma omp for nowait
   for(...)
     {...};
```

```
#pragma single nowait
{ [...] }
```

Use when threads would wait between independent computations

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
```

Barrier Construct

Explicit barrier synchronization

Each thread waits until all threads arrive

```
#pragma omp parallel shared (A, B, C)
{
        DoSomeWork(A,B);
        printf("Processed A into B\n");
#pragma omp barrier
        DoSomeWork(B,C);
        printf("Processed B into C\n");
}
```

Atomic Construct

Special case of a critical section

Applies only to simple update of memory location

```
#pragma omp parallel for shared(x, y, index, n)
   for (i = 0; i < n; i++) {
      #pragma omp atomic
        x[index[i]] += work1(i);
      y[i] += work2(i);
   }
```

OpenMP* API

Get the thread number within a team

```
int omp_get_thread_num(void);
```

Get the number of threads in a team

```
int omp_get_num_threads(void);
```

Usually not needed for OpenMP codes

- Can lead to code not being serially consistent
- Does have specific uses (debugging)
- Must include a header file

```
#include <omp.h>
```

Programming with OpenMP What's Been Covered

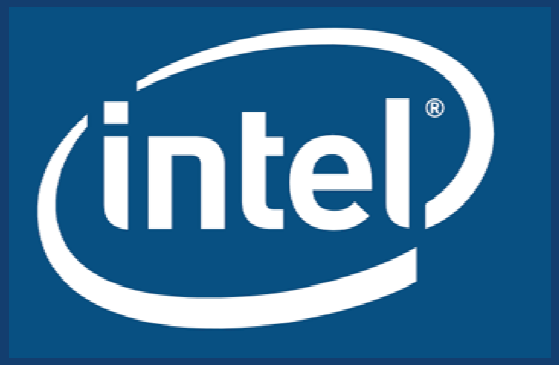
OpenMP* is:

- A simple approach to parallel programming for shared memory machines

We explored basic OpenMP coding on how to:

- Make code regions parallel (`omp parallel`)
- Split up work (`omp for`)
- Categorize variables (`omp private`….)
- Synchronize (`omp critical`…)

We reinforced fundamental OpenMP concepts through several labs


intel

Advanced Concepts

More OpenMP*

Data environment constructs

- **`FIRSTPRIVATE`**
- **`LASTPRIVATE`**
- **`THREADPRIVATE`**

Firstprivate Clause

Variables initialized from shared variable

C++ objects are copy-constructed

```
incr=0;
#pragma omp parallel for firstprivate(incr)
for (I=0;I<=MAX;I++) {
      if ((I%2)==0) incr++;
      A(I)=incr;
}
```

Lastprivate Clause

Variables update shared variable using value from last iteration

C++ objects are updated as if by assignment

```
void sq2(int n,
         double *lastterm)
{
  double x; int i;
  #pragma omp parallel
  #pragma omp for lastprivate(x)
  for (i = 0; i < n; i++){
     x = a[i]*a[i] + b[i]*b[i];
     b[i] = sqrt(x);
  }
  lastterm = x;
}
```

Threadprivate Clause

Preserves global scope for per-thread storage

Legal for name-space-scope and file-scope

Use copyin to initialize from master thread

```
struct Astruct A;
#pragma omp threadprivate(A)
…

#pragma omp parallel copyin(A)
  do_something_to(&A);
…

#pragma omp parallel
  do_something_else_to(&A);
```

Private copies of "A" persist between regions

Performance Issues

Idle threads do no useful work

Divide work among threads as evenly as possible

- Threads should finish parallel tasks at same time

Synchronization may be necessary

- Minimize time waiting for protected resources

Load Imbalance

Unequal work loads lead to idle threads and wasted time.

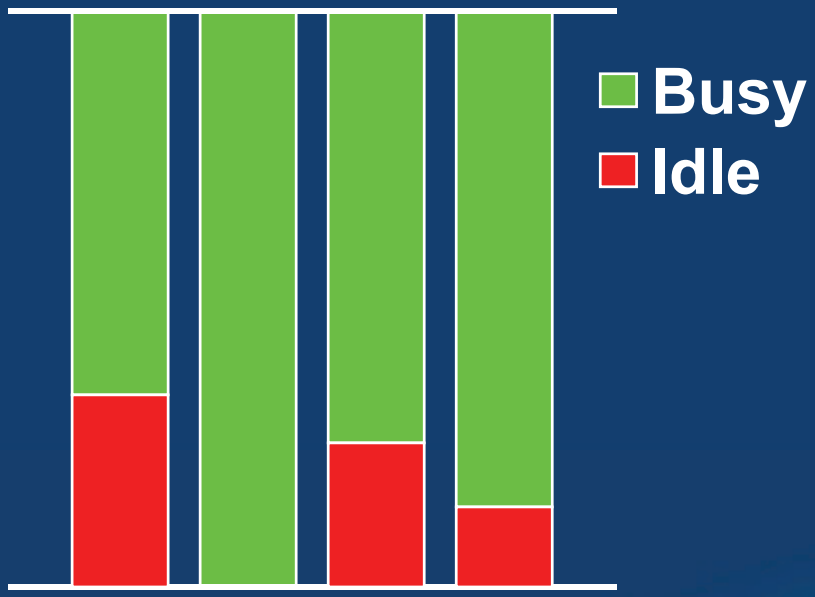
```
#pragma omp parallel
{

#pragma omp for
  for( ; ; ){

  }

}
```

time



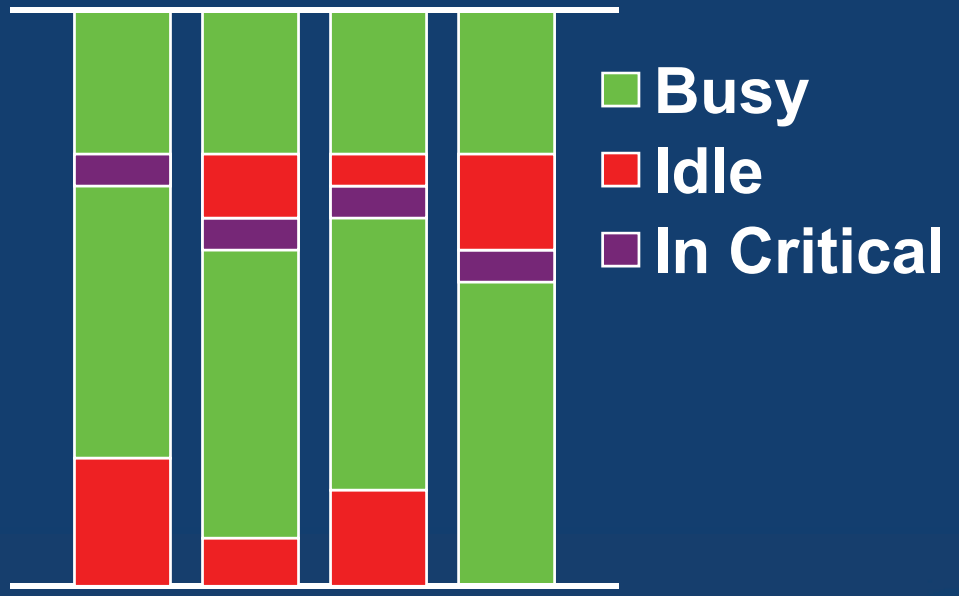
Synchronization

Lost time waiting for locks

```
#pragma omp parallel
{

#pragma omp critical
   {
     . . .
   }
   . . .
}
```

time



Performance Tuning

Profilers use sampling to provide performance data.

Traditional profilers are of limited use for tuning OpenMP*:

- Measure CPU time, not wall clock time
- Do not report contention for synchronization objects
- Cannot report load imbalance
- Are unaware of OpenMP constructs

Programmers need profilers specifically designed for OpenMP.

Static Scheduling: Doing It By Hand

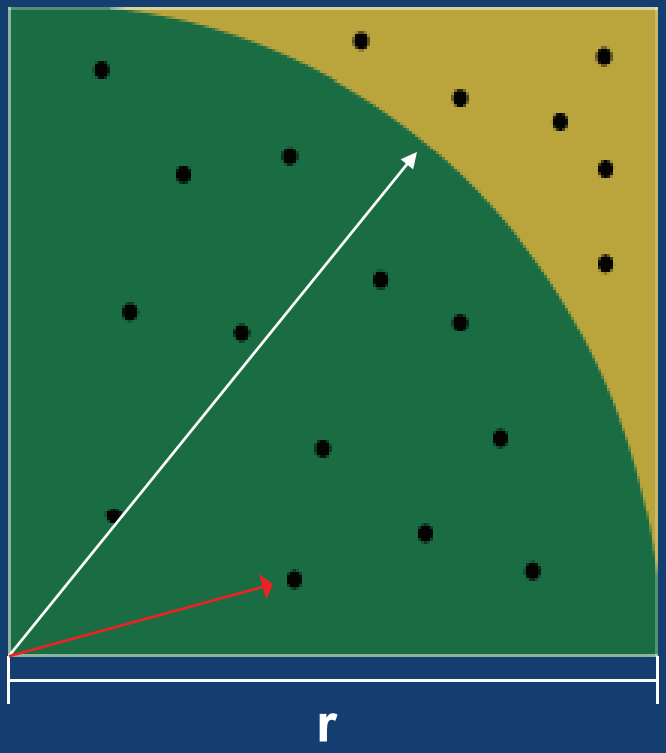
Must know:

- Number of threads (Nthrds)
- Each thread ID number (id)

Compute start and end iterations:

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

Monte Carlo Pi



$$\frac{\#\text{ of darts hitting circle}}{\#\text{ of darts in square}} = \frac{1/4\pi r^2}{r^2}$$

$$\pi = 4\frac{\#\text{ of darts hitting circle}}{\#\text{ of darts in square}}$$

```
loop 1 to MAX
    x.coor=(random#)
    y.coor=(random#)
    dist=sqrt(x^2 + y^2)
    if (dist <= 1)
      hits=hits+1
pi = 4 * hits/MAX
```

Making Monte Carlo's Parallel

```
hits = 0
call SEED48(1)
DO I = 1, max
    x = DRAND48()
    y = DRAND48()
    IF (SQRT(x*x + y*y) .LT. 1)
  THEN
        hits = hits+1
    ENDIF
END DO
pi = REAL(hits)/REAL(max) * 4.0
```

What is the challenge here?

Activity 3: Computing Pi

Use the Intel® Math Kernel Library (Intel® MKL) VSL:

- Intel MKL's VSL (Vector Statistics Libraries)
- VSL creates an array, rather than a single random number
- VSL can have multiple seeds (one for each thread)

Objective:

- Use basic OpenMP* syntax to make Pi parallel
- Choose the best code to divide the task up
- Categorize properly all variables