Concurrent Programming
Lecture 9

ROYAL INSTITUTE
OF TECHNOLOGY

Tutorial: An Introduction to OpenMP*

Vladimir Vlassov
KTH/ICT/EECS

(Slides adapted from Intel sources)

* The name “OpenMP” is the property of the OpenMP Architecture Review Board

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks

OpenMP* Overview:

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

C$OM

C$OM

C$C

C$

#pr

C$

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

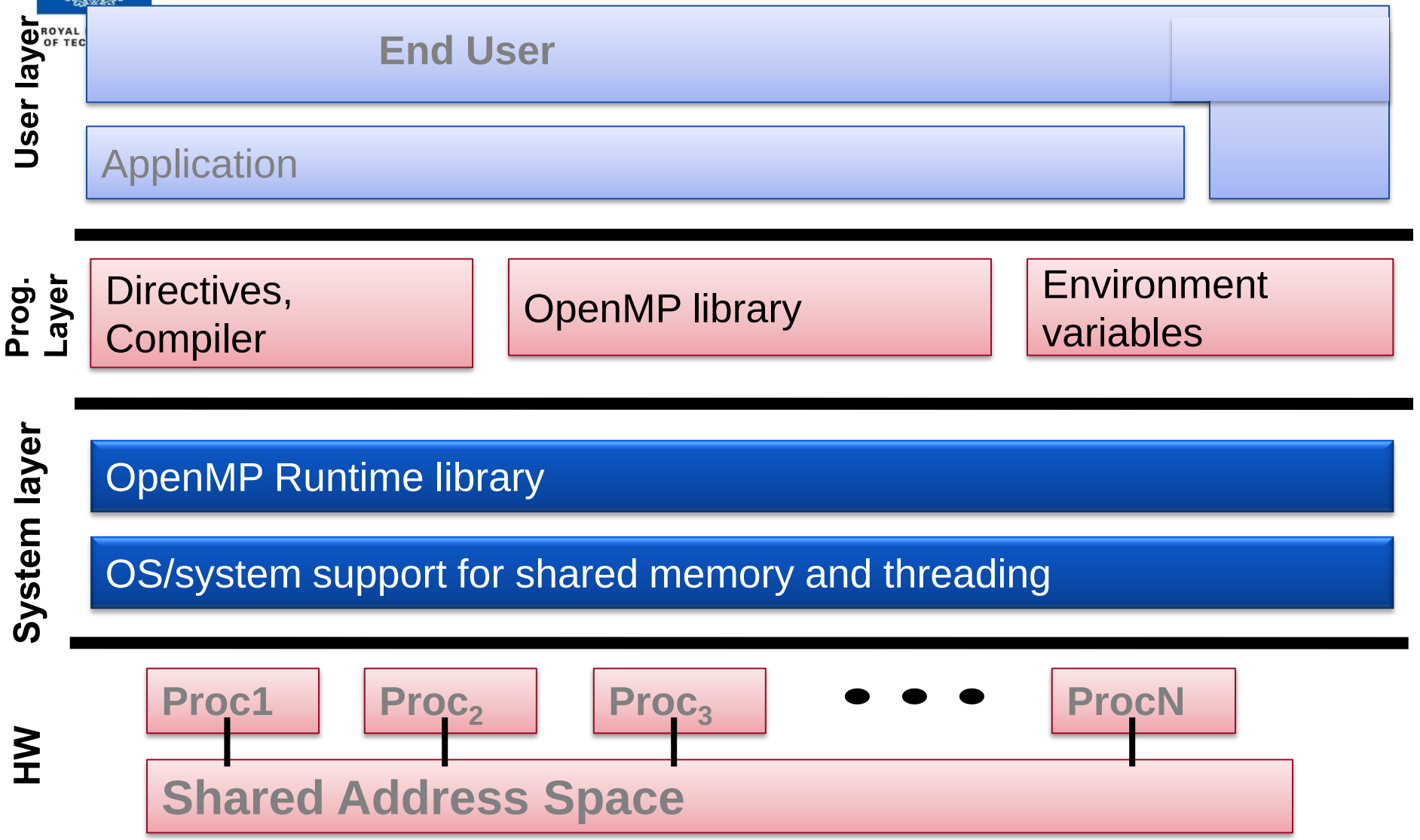
omp_set_lock(lck)

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

* The name "OpenMP" is the property of the OpenMP Architecture Review Board.

OpenMP Basic Defs: Solution Stack



OpenMP core syntax

- Most of the constructs in OpenMP are compiler directives.

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

  - Example

    ```
    #pragma omp parallel num_threads(4)
    ```

- Function prototypes and types in the file:

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

- Most OpenMP constructs apply to a structured block.
- ***Structured block***: a block of one or more statements with one point of entry at the top and one point of exit at the bottom.
  - It's OK to have an `exit()` within the structured block.

Exercise 1: Hello World (1/3)

- A program that prints "Hello World" (**hello.c**).

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  printf("Hello World \n");
}
```

- A zip file **omp-hands-on.zip** from the course website to your computer for hands-on…

Hello world (2/3)

- Write a multithreaded program where each thread prints "Hello World" (see a sample solution in `hello_par.c`).

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
     int nthreads = 4;
     omp_set_num_threads(nthreads);
#pragma omp parallel
     {
        int id = omp_get_thread_num();
        printf("Hello World from thread = %d", id);
        printf(" with %d threads\n", omp_get_num_threads());
     }
     printf("all done, with hopefully %d threads\n", nthreads);
}
```

Switches for compiling and linking

`gcc -fopenmp`	gcc
`icc –openmp`	Intel (linux)
`cc –xopenmp`	Oracle cc

Hello World (3/3)

- A multithreaded program where each thread prints "Hello World" (see a sample solution in `hello_par.c`).

OpenMP include file

```
#include <omp.h>
int main ()
{
  int nthreads = 4;
  omp_set_num_threads(nthreads);
#pragma omp parallel
  {
   int id = omp_get_thread_num();
   printf("Hello World from thread = %d", id);
   printf(" with %d´threads\n", omp_get_num_threads());
   }
   printf("all done, with hopefully %d threads\n",nthreads);
}
```

Parallel region with default number of threads

Sample Output:

```
Hello World from thread = 0 with 4 threads
Hello World from thread = 2 with 4 threads
Hello World from thread = 1 with 4 threads
Hello World from thread = 3 with 4 threads
all done, with hopefully 4 threads
```

Runtime library function to get the number of threads.

Runtime library function to get a thread ID.

End of the Parallel region

ROYAL INSTITUTE OF TECHNOLOGY

How do threads interact?

- OpenMP is a multi-threading, shared address model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to prevent data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

ROYAL INSTITUTE OF TECHNOLOGY

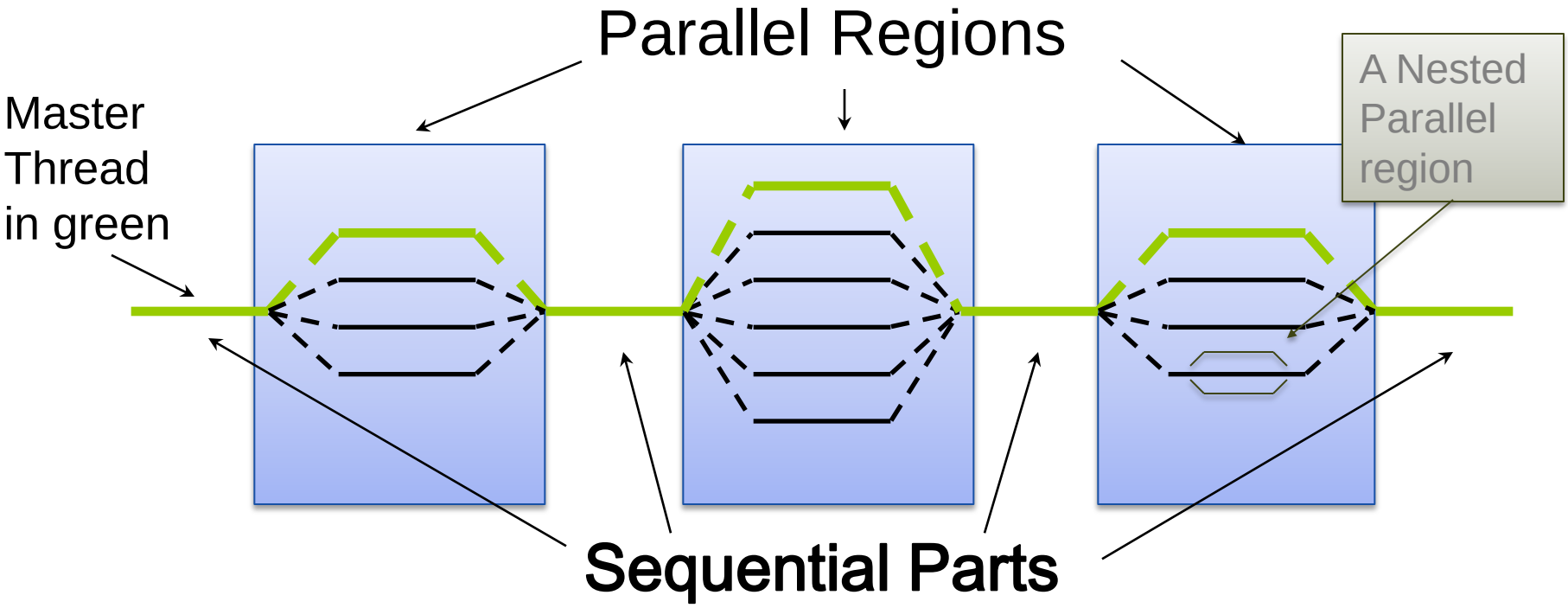
Outline

- Introduction to OpenMP
- **Creating Threads**
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks

OpenMP Programming Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program.



Thread Creation: Parallel Regions

- You create threads in OpenMP* with the **parallel** construct.
- N.B. Be careful to whether variables are shared or private
  - ***Variables declared inside parallel region are private,***
  - ***Outside parallel region: shared***
- For example, to create a 4 thread parallel region:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block

Runtime function to request a certain number of threads

Runtime function returning a thread ID

- Each thread calls `pooh(ID,A)` for `ID` = 0 to 3

Thread Creation: Parallel Regions (cont)

- You create threads in OpenMP* with the **parallel** construct.
- N.B. Be careful to whether variables are shared or private
  - ***Variables declared inside parallel region are private,***
  - ***Outside parallel region: shared***
- For example, to create a 4 thread parallel region:

```
double A[1000];
//omp_set_num_threads(4);
#pragma omp parallel num_threads(4)
{
   int ID = omp_get_thread_num();
   pooh(ID,A);
}
```

Clause to request a certain number of threads

Each thread executes a copy of the code within the structured block

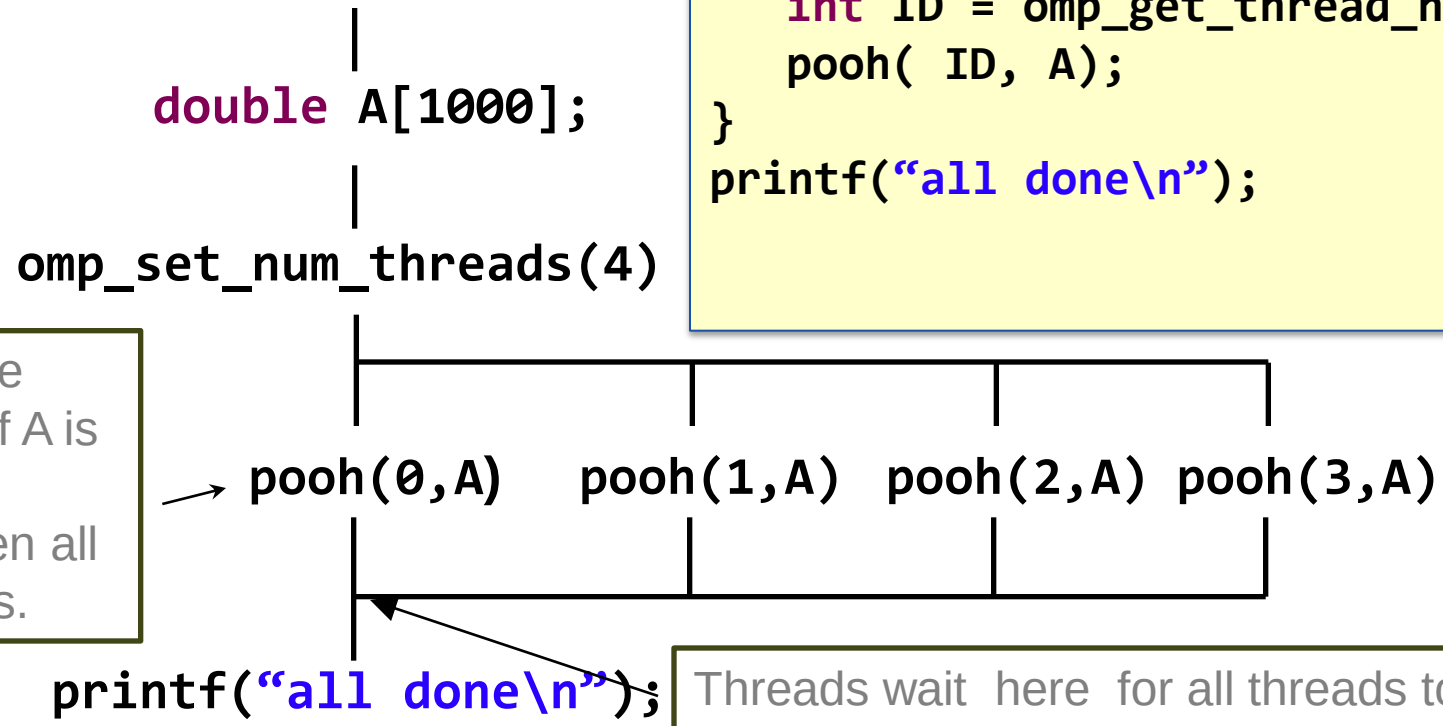
Runtime function returning a thread ID

- Each thread calls `pooh(ID,A)` for `ID` = 0 to 3

Thread Creation: Parallel Regions example

- Each thread executes the same code redundantly.

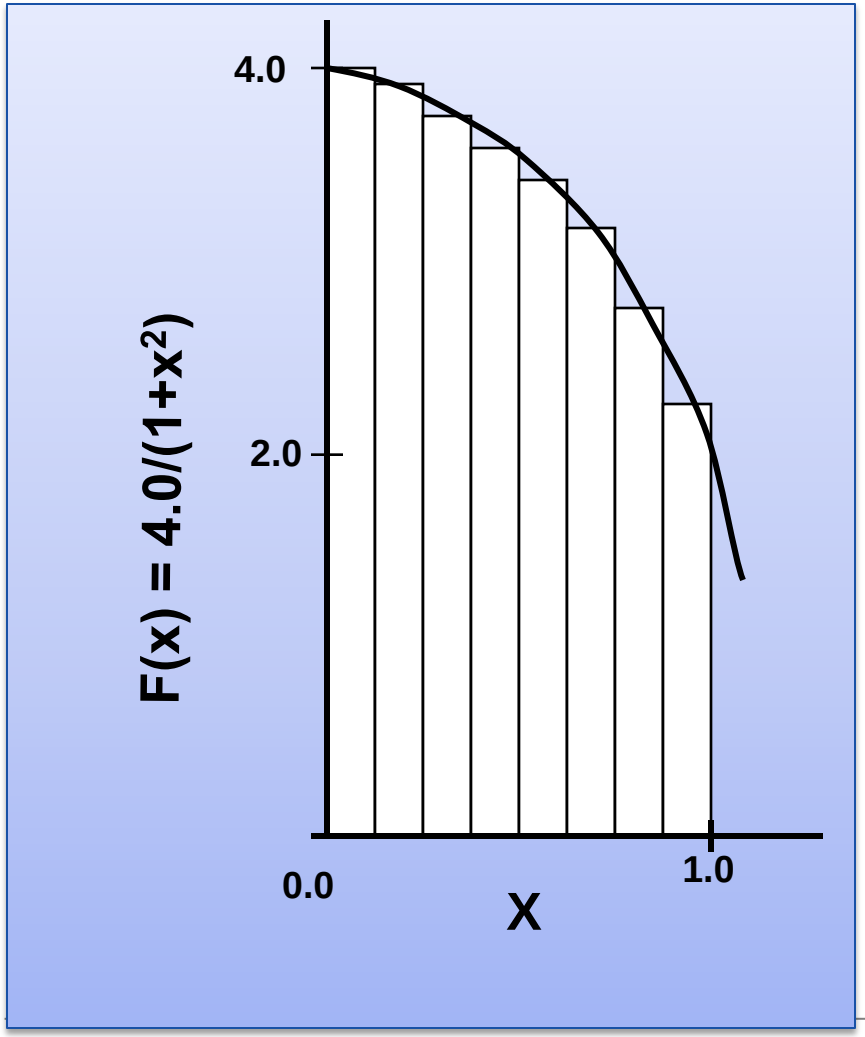
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh( ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

Exercises 2 to 4: Numerical Integration



Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

ROYAL INSTITUTE OF TECHNOLOGY

Exercises 2 to 4: Compute PI. Serial program, **pi-seq.c**

```
/*... Written by Tim Mattson, 11/99. */
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;
double step;
int main ()
{
  int i; double x, pi, sum = 0.0;
  double start_time, run_time;
  step = 1.0/(double) num_steps;
  start_time = omp_get_wtime();
  for (i = 1; i <= num_steps; i++){
     x = (i-0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
  run_time = omp_get_wtime() - start_time;
  printf("\n pi with %d steps is %f in %f seconds \n",
         num_steps,pi,run_time);
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 2: Parallel PI program

- Create a parallel version of the pi program using a parallel construct.
- Pay close attention to shared versus private variables.
  - Variables declared inside parallel region are private,
  - Outside parallel region: shared
- In addition to a parallel construct, you will need the runtime library routines
  - `int omp_get_num_threads(); // Number of threads in the team`
  - `int omp_get_thread_num(); // Thread ID or rank`
  - `double omp_get_wtime(); // Time in sec since a fixed point in the past`

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 2: Parallel PI program (SPMD simple, **pi_spmd_simple.c**)

```
/*... Written by Tim Mattson, 11/99. */
#include <stdio.h>
#include <omp.h>
#define MAX_THREADS 32
static long num_steps = 100000000;
double step;
int main() {
  int i, j;
  double pi, full_sum = 0.0;
  double start_time, run_time;
  volatile double sum[MAX_THREADS];
  step = 1.0 / (double) num_steps;
  for (j = 1; j <= MAX_THREADS; j = j * 2) {
    omp_set_num_threads(j);
    full_sum = 0.0;
    start_time = omp_get_wtime();
```

```
#pragma omp parallel
    {
      int i;
      int id = omp_get_thread_num();
      int numthreads = omp_get_num_threads();
      double x;
      sum[id] = 0.0;
      if (id == 0)
        printf(" num_threads = %d", numthreads);
      for (i = id; i < num_steps; i += numthreads) {
        x = (i + 0.5) * step;
        sum[id] = sum[id] + 4.0 / (1.0 + x * x);
      }
    }
    for (full_sum = 0.0, i = 0; i < j; i++)
      full_sum += sum[i];
    pi = step * full_sum;
    run_time = omp_get_wtime() - start_time;
    printf("\n pi is %f in %f seconds %d thrds \n",
      pi, run_time, j);
  }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Introduction to OpenMP
- Creating Threads
- **Synchronization**
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks

Synchronization

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

Synchronization is used to impose order constraints and to protect access to shared data

Discussed later

Synchronization: critical

- ***Mutual exclusion***: Only one thread at a time can enter a **critical** region.

Threads wait their turn – only one at a time calls consume()

```
float res;
#pragma omp parallel
{      float B;   int i, id, nthrds;
       id = omp_get_thread_num();
       nthrds = omp_get_num_threads();
       for (i=id; i<niters; i+=nthrds) {
           B =  big_job(i);
#pragma omp critical
           consume (B, res);
       }
}
```

Synchronization: Atomic

- **atomic** provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

```
#pragma omp parallel
{
        double tmp, B;
        B =  DOIT();
        tmp = big_ugly(B);
 #pragma omp atomic
       X +=  tmp;
}
```

Atomic only protects the read/update of X

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 3: Parallel PI program

- In exercise 2, you probably used a shared array to create space for each thread to store its partial sum.
- If array elements happen to share a cache line, this leads to ***false sharing***.
  - Non-shared data in the same cache line so each update invalidates the cache line … in essence "sloshing independent data" back and forth between threads.
- Modify your "pi program" from exercise 2 to avoid false sharing due to the sum array.

Exercise 3: Parallel PI program (SPMD final, **pi_spmd_final.c**)

```
#include <stdio.h>
#include <omp.h>
#define MAX_THREADS 32
static long num_steps = 100000000;
double step;
int main() {
  int i, j;
  double pi, full_sum = 0.0;
  double start_time, run_time;
  double sum[MAX_THREADS];
  step = 1.0 / (double) num_steps;
  for (j = 1; j <= MAX_THREADS; j = j * 2) {
    omp_set_num_threads(j);
    full_sum = 0.0;
    start_time = omp_get_wtime();
```

```
#pragma omp parallel private(i)
    {
      int id = omp_get_thread_num();
      int numthreads = omp_get_num_threads();
      double x;
      volatile double partial_sum = 0;
#pragma omp single
      printf(" num_threads = %d", numthreads);
      for (i = id; i < num_steps; i += numthreads) {
        x = (i + 0.5) * step;
        partial_sum += +4.0 / (1.0 + x * x);
      }
#pragma omp critical
      full_sum += partial_sum;
    }
    pi = step * full_sum;
    run_time = omp_get_wtime() - start_time;
    printf("\n pi is %f in %f seconds %d threds \n ",
            pi, run_time, j);
  }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- **Parallel Loops**
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks

ROYAL INSTITUTE OF TECHNOLOGY

SPMD vs. worksharing

- A parallel construct by itself creates an ***SPMD*** or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - This is called ***worksharing***
    - Loop construct
    - Sections/section constructs
    - Single construct
    - Task construct …. Available in OpenMP 3.0

Discussed later

The loop worksharing Constructs

- The loop worksharing construct splits up loop iterations among the threads in a team

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

Loop construct name:

- C/C++: **for**
- Fortran: **do**

The variable I is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

Loop worksharing Constructs
A motivating example

Sequential code

```
for (i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region (SPMD)

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

OpenMP parallel region and a worksharing for construct

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

ROYAL INSTITUTE OF TECHNOLOGY

Loop worksharing Constructs: The schedule clause

- The **schedule** clause affects how loop iterations are mapped onto threads
  - **schedule(static [,chunk])**
    - Deal-out blocks of iterations of size "chunk" to each thread.
    - By default, the chunk size is loop_count/number_of_threads.
  - **schedule(dynamic[,chunk])**
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled. By default, the chunk size is 1.
    - The extra overhead involved.
  - **schedule(guided[,chunk])**
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
    - By default the chunk size is approximately loop_count/number_of_threads.
  - **schedule(runtime)**
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0).

Loop worksharing Constructs: The schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Least work at runtime : scheduling done at compile-time

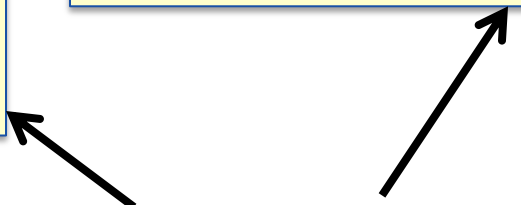
Most work at runtime : complex scheduling logic used at run-time

Combined parallel/worksharing construct

- OpenMP shortcut: Put the "parallel" and the worksharing directive on the same line

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

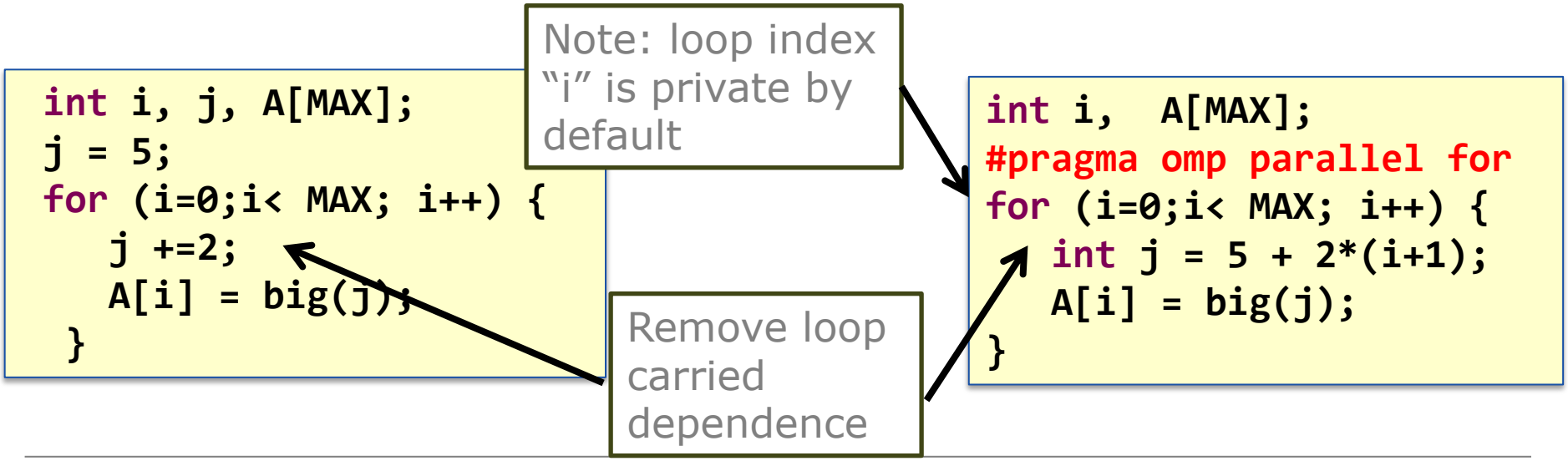
```
 double  res[MAX];  int i;
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
         res[i] = huge();
    }
```



These are equivalent

Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test



Reduction

- How do we handle this case?

```
double  ave=0.0, A[MAX];    int i;
 for (i=0;i< MAX; i++) {
     ave + = A[i];
 }
ave = ave/MAX;
```

  - We are combining values into a single accumulation variable (**ave**) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a "***reduction***".
- Support for reduction operations is included in most parallel programming environments.

Reduction

- OpenMP reduction clause: `reduction (op : list)`
- Inside a parallel or a work-sharing construct:
  - A local copy of each `list` variable is made and initialized depending on the `op` (e.g. 0 for "+").
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in `list` must be shared in the enclosing parallel region.

```
double  ave=0.0, A[MAX];    int i;
 #pragma omp parallel for reduction (+:ave)
 for (i=0;i< MAX; i++) {
     ave + = A[i];
 }
ave = ave/MAX;
```

OpenMP: Reduction operands/initial-values

- Many different associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.
MIN*	Largest pos. number
MAX*	Most neg. number

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 4: Pi with loops

- Go back to the serial pi program and parallelize it with a loop construct
- Your goal is to minimize the number of changes made to the serial program.

Exercise 4: Parallel PI with loops (pi_loop.c)

```
#include <stdio.h>
#include <omp.h>
static long num_steps = 100000000;
double step;
int main() {
  int i;
  double x, pi, sum = 0.0;
  double start_time, run_time;
  step = 1.0 / (double) num_steps;
  for (i = 1; i <= 4; i++) {
    sum = 0.0;
    omp_set_num_threads(i);
    start_time = omp_get_wtime();
```

```
#pragma omp parallel
{
#pragma omp single
    printf(" num_threads = %d", omp_get_num_threads());
#pragma omp for reduction(+:sum) private(x)
    for (i = 1; i <= num_steps; i++) {
      x = (i - 0.5) * step;
      sum = sum + 4.0 / (1.0 + x * x);
    }
}
    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("\n pi is %f in %f seconds and %d threads\n",
        pi, run_time, i);
  }
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 5: Optimizing loops

- Parallelize the matrix multiplication program in the file matmul.c
- Can you optimize the program by playing with how the loops are scheduled?

- N.B. Be careful to whether variables are shared or private
  - Variables declared inside parallel region are private,
  - Outside parallel region: shared

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- **Synchronize single masters and stuff**
- Data environment
- OpenMP Tasks

Synchronization: Barrier

- ***Barrier***: Each thread waits until all threads arrive.

```
#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 at the end of a for worksharing construct

implicit barrier at the end of a parallel region

no implicit barrier due to nowait

ROYAL INSTITUTE OF TECHNOLOGY

Master Construct

- The **master** construct denotes a structured block that is only executed by the master thread.
- The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel
{
        do_many_things();
#pragma omp master
        {     exchange_boundaries();   }
#pragma omp  barrier
        do_many_other_things();
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Single worksharing Construct

- The **single** construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
- A barrier is implied at the end of the single block (can remove the barrier with a **nowait** clause).

```
#pragma omp parallel
{
        do_many_things();
#pragma omp single
        {     exchange_boundaries();   }
        do_many_other_things();
}
```

Sections worksharing Construct

- The **sections** worksharing construct gives a different structured block to each thread.

```
#pragma omp parallel
{
   #pragma omp sections
   {
   #pragma omp section
       x_calculation();
   #pragma omp section
       y_calculation();
   #pragma omp section
       z_calculation();
   }
}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

ROYAL INSTITUTE OF TECHNOLOGY

Synchronization: ordered

- The **ordered** region executes in the sequential order.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)
      for (I=0;I<N;I++){
            tmp = NEAT_STUFF(I);
#pragma ordered
            res += consum(tmp);
      }
```

Synchronization: Lock routines

A lock implies a memory fence (a "flush") of all thread visible variables

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - `omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()`
- Nested Locks
  - A nested lock is available if it is unset or if it is set but owned by the thread executing the nested lock function
    - `omp_init_nest_lock(), omp_set_nest_lock(), omp_unset_nest_lock(), omp_test_nest_lock(), omp_destroy_nest_lock()`

Note: a thread always accesses the most recent copy of the lock, so you don't need to use a flush on the lock variable.

Synchronization: Simple Locks

- Protect resources with locks.

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
      id = omp_get_thread_num();
      tmp = do_lots_of_work(id);
      omp_set_lock(&lck);
         printf("%d %d", id, tmp);
      omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

Wait here for your turn.

Release the lock so the next thread gets a turn.

Free-up storage when done.

ROYAL INSTITUTE OF TECHNOLOGY

Runtime Library routines

- Runtime environment routines:
  - Modify/Check the number of threads
    - `omp_set_num_threads(), omp_get_num_threads(), omp_get_thread_num(), omp_get_max_threads()`
  - Are we in an active parallel region?
    - `omp_in_parallel()`
  - Do you want the system to dynamically vary the number of threads from one parallel construct to another?
    - `omp_set_dynamic, omp_get_dynamic();`
  - How many processors in the system?
    - `omp_num_procs()`

…plus a few less commonly used routines.

Runtime Library routines

- To use a known, fixed number of threads in a program, (1) tell the system that you don't want dynamic adjustment of the number of threads, (2) set the number of threads, then (3) save the number you got.

```
#include <omp.h>
void main()
{   int num_threads;
      omp_set_dynamic( 0 );
      omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
    {     int id= omp_get_thread_num();
#pragma omp single
            num_threads = omp_get_num_threads();
          do_lots_of_stuff(id);
    }
}
```

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.

Protect this op since Memory stores are not atomic

Even in this case, the system may give you fewer threads than requested. If the precise # of threads matters, test for it and respond accordingly.

ROYAL INSTITUTE OF TECHNOLOGY

Environment Variables

- Set the default number of threads to use.
  - `OMP_NUM_THREADS int_literal`
- Control how "omp for schedule(RUNTIME)" loop iterations are scheduled.
  - `OMP_SCHEDULE "schedule[, chunk_size]"`

… Plus several less commonly used environment variables.

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- **Data environment**
- OpenMP Tasks

ROYAL INSTITUTE OF TECHNOLOGY

Data environment: Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms (Fortran) or functions(C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

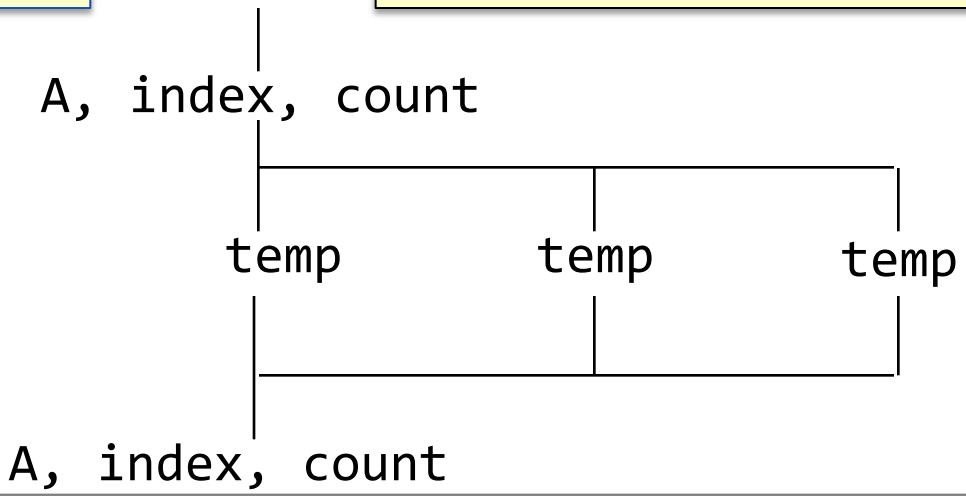
Data sharing: Examples

```
double A[10];
int main() {
   int index[10];
   #pragma omp parallel
     work(index);
   printf("%d\n", index[0]);
}
```

```
extern double A[10];
void work(int *index)
   {
   double temp[10];
   static int count;
   ...
}
```

A, index and count are shared by all threads.

temp is local to each thread



Data sharing: Changing storage attributes

- One can selectively change storage attributes for constructs using the following clauses*
  - `SHARED`
  - `PRIVATE`
  - `FIRSTPRIVATE`
- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with:
  - `LASTPRIVATE`
- The default attributes can be overridden with:
  - `DEFAULT (PRIVATE | SHARED | NONE)`
  - `DEFAULT(PRIVATE)` is Fortran only

All the clauses on this page apply to the OpenMP construct

NOT to the entire region.

All data clauses apply to parallel constructs and worksharing constructs except “shared” which only applies to parallel constructs.

Data Sharing: Private Clause

- **private(var)** creates a new local copy of **var** for each thread.
  - The value is uninitialized
  - In OpenMP 2.5 the value of the shared variable is undefined after the region

```
void wrong() {
      int tmp = 0;
#pragma omp for private(tmp)
      for (int j = 0; j < 1000; ++j)
            tmp += j;
      printf("%d\n", tmp);
}
```

tmp was not initialized

tmp: 0 in 3.0, unspecified in 2.5

Data Sharing: Firstprivate Clause

- **firstprivate** is a special case of `private`.
  - Initializes each private copy with the corresponding value from the master thread.

```
void useless() {
      int tmp = 0;
#pragma omp for firstprivate(tmp)
      for (int j = 0; j < 1000; ++j)
        tmp += j;
     printf("%d\n", tmp);
 }
```

Each thread gets its own **tmp** with an initial value of 0

tmp: 0 in 3.0, unspecified in 2.5

Data sharing: Lastprivate Clause

- Lastprivate passes the value of a private from the last iteration to a global variable.

```
void closer() {
    int tmp = 0;
#pragma omp parallel for firstprivate(tmp) \
   lastprivate(tmp)
   for (int j = 0; j < 1000; ++j)
      tmp += j;
   printf("%d\n", tmp);
}
```

Each thread gets its own **tmp** with an initial value of 0

tmp is defined as its value at the "last sequential" iteration (i.e., for j=999)

Data Sharing: A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

```
        variables A,B, and C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- "A" is shared by all threads; equals 1
- "B" and "C" are local to each thread.
  - B's initial value is undefined
  - C's initial value equals 1

Outside this parallel region ...

- The values of "B" and "C" are unspecified in OpenMP 2.5, and in OpenMP 3.0 if referenced in the region but outside the construct.

ROYAL INSTITUTE OF TECHNOLOGY

Data Sharing: Default Clause

- Note that the default storage attribute is **`DEFAULT(SHARED)`** (so no need to use it)
  - Exception: `#pragma omp task`
- To change default: **`DEFAULT(PRIVATE)`**
  - each variable in the construct is made private as if specified in a private clause
  - mostly saves typing
- **`DEFAULT(NONE)`**: no default for variables in static extent. Must list storage attribute for each variable in static extent. Good programming practice!

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

Data Sharing: tasks (OpenMP 4.0)

- The default for tasks is usually `firstprivate`, because the task may not be executed until later (and variables may have gone out of scope).
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared, because the barrier guarantees task completion.

```
#pragma omp parallel shared(A) private(B)
{
   ...
#pragma omp task
   {
      int C;
      compute(A, B, C);
   }
}
```

A is shared
B is firstprivate
C is private

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 6: Molecular dynamics (MolDyn folder)

- The code supplied is a simple molecular dynamics simulation of the melting of solid argon.
- Computation is dominated by the calculation of force pairs in subroutine forces (in forces.c)
- Parallelise this routine using a parallel for construct and atomics. Think carefully about which variables should be SHARED, PRIVATE or REDUCTION variables.
- Experiment with different schedules kinds.

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 6 (cont.)

- Once you have a working version, move the parallel region out to encompass the iteration loop in main.c
  - code other than the forces loop must be executed by a single thread (or workshared).
  - how does the data sharing change?
- The atomics are a bottleneck on most systems.
  - This can be avoided by introducing a temporary array for the force accumulation, with an extra dimension indexed by thread number.
  - Which thread(s) should do the final accumulation into f?

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- **OpenMP Tasks**

ROYAL INSTITUTE OF TECHNOLOGY

General task characteristics

- A task has
  - Code to execute
  - A data environment (it *owns* its data)
  - An assigned thread that executes the code and uses the data
- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at some later time

ROYAL INSTITUTE OF TECHNOLOGY

Definitions

- *Task construct* – `task` directive plus structured block
- *Task* – the package of code and instructions for allocating data created when a thread encounters a task construct
- *Task region* – the dynamic sequence of instructions produced by the execution of a task by a thread

ROYAL INSTITUTE OF TECHNOLOGY

Tasks and OpenMP

- Tasks have been fully integrated into OpenMP
- Key concept: OpenMP has always had tasks, we just never called them that.
  - Thread encountering parallel construct packages up a set of implicit tasks, one per thread.
  - Team of threads is created.
  - Each thread in team is assigned to one of the tasks (and tied to it).
  - Barrier holds original master thread until all implicit tasks are finished.
- We have simply added a way to create a task explicitly for the team to execute.
- Every part of an OpenMP program is part of one task or another!

ROYAL INSTITUTE OF TECHNOLOGY

task Construct

```
#pragma omp task [clause[[,]clause] ...]
                 structured-block
```

where ***clause*** can be one of:

```
if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none  )
```

ROYAL INSTITUTE OF TECHNOLOGY

The **if** clause

- When the **if** clause argument is false
  - The task is executed immediately by the encountering thread.
  - The data environment is still local to the new task...
  - ...and it's still a different task with respect to synchronization.
- It's a user directed optimization
  - when the cost of deferring the task is too great compared to the cost of executing the task code
  - to control cache and memory affinity

ROYAL INSTITUTE OF TECHNOLOGY

When/where are tasks complete?

- At thread barriers, explicit or implicit
  - applies to all tasks generated in the current parallel region up to the barrier
  - matches user expectation

- At task barriers
  - i.e. Wait until all tasks defined in the current task have completed.

```
#pragma omp taskwait
```

  - Note: applies only to tasks generated in the current task, not to “descendants” .

Example – parallel pointer chasing using tasks

```
#pragma omp parallel
{
  #pragma omp single private(p)
   {
    p = listhead ;
    while (p) {
       #pragma omp task
                process (p);
       p = next (p) ;
     }
   }
}
```

p is firstprivate inside this task

Example – parallel pointer chasing on multiple lists using tasks

```
#pragma omp parallel
{
   #pragma omp for private(p)
   for ( int i = 0; i < numlists ; i++) {
       p = listheads [ i ] ;
       while ( p ) {
       #pragma omp task
           process (p);
       p = next (p ) ;
       }
   }
}
```

Example: postorder tree traversal

```
void postorder(node *p) {
    if (p->left)
      #pragma omp task
        postorder(p->left);
    if (p->right)
      #pragma omp task
        postorder(p->right);
#pragma omp taskwait // wait for descendants
  process(p->data);
}
```

Task scheduling point

- Parent task suspended until children tasks complete

ROYAL INSTITUTE OF TECHNOLOGY

Task switching

- Certain constructs have task scheduling points at defined locations within them
- When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (called task switching)
- It can then return to the original task and resume

ROYAL INSTITUTE OF TECHNOLOGY

Task switching example

```
#pragma omp single
 {
   for (i=0; i<ONEZILLION; i++)
     #pragma omp task
       process(item[i]);
 }
```

- Too many tasks generated in an eye-blink
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - execute an already generated task (draining the "task pool")
  - dive into the encountered task (could be very cache-friendly)

ROYAL INSTITUTE OF TECHNOLOGY

Thread switching

```
#pragma omp single
 {
   #pragma omp task untied
     for (i=0; i<ONEZILLION; i++)
       #pragma omp task
         process(item[i]);
 }
```

- Eventually, too many tasks are generated
- Generating task is suspended and executing thread switches to a long and boring task
- Other threads get rid of all already generated tasks, and start starving…

- With thread switching, the generating task can be resumed by a different thread, and starvation is over
- Too strange to be the default: the programmer is responsible!

ROYAL INSTITUTE OF TECHNOLOGY

Conclusions on tasks

- Enormous amount of work by many people
- Tightly integrated since 3.0 spec
- Flexible model for irregular parallelism
- Provides balanced solution despite often conflicting goals
- Appears that performance can be reasonable

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 7: tasks in OpenMP

- Consider the program **linked.c**
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using tasks.
- Compare your solution's complexity to an approach without tasks.

ROYAL INSTITUTE OF TECHNOLOGY

Exercise 8: linked lists the hard way

- Consider the program **linked.c**
  - Traverses a linked list computing a sequence of Fibonacci numbers at each node.
- Parallelize this program using constructs defined in OpenMP 2.5 (loop worksharing constructs … i.e. don't use OpenMP 3.0 tasks).
- Once you have a correct program, optimize it.

ROYAL INSTITUTE OF TECHNOLOGY

Parallel loops

- Guarantee that this works … i.e. that the same schedule is used in the two loops:

```
#pragma omp for schedule(static) nowait
for (i=0; i<n; i++){
   a[i] = ....
}
#pragma omp for schedule(static)
for (i=0; i<n; i++) {
.... = a[i]
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Loops (cont.)

- Allow collapsing of perfectly nested loops

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

- Will form a single loop and then parallelize that

Loops (cont.)

- Made **schedule(runtime)** more useful
  - can get/set it with library routines
    omp_set_schedule()
    omp_get_schedule()
  - allow implementations to implement their own schedule kinds
- Added a new schedule kind AUTO which gives full freedom to the runtime to determine the scheduling of iterations to threads.
- Allowed C++ Random access iterators as loop control variables in parallel loops

ROYAL INSTITUTE OF TECHNOLOGY

Portable control of threads

- Added environment variable to control the size of child threads' stack

  OMP_STACKSIZE

- Added environment variable to hint to runtime how to treat idle threads

  OMP_WAIT_POLICY

  - **ACTIVE** keep threads alive at barriers/locks
  - **PASSIVE** try to release processor at barriers/locks

ROYAL INSTITUTE OF TECHNOLOGY

New features in OpenMP 4.0

- **Support for accelerators**
  - To describe regions of code where data and/or computation should be moved to another computing device.
  - Several prototypes for the accelerator proposal have already been implemented.
- **SIMD constructs**
  - To vectorize both serial as well as parallelized loops.
- **Thread affinity**
  - To define where to execute OpenMP threads in order to achieve better locality, less false sharing and more memory bandwidth.
- **Tasking extensions**
  - Task groups, task dependency.
- **Support for Fortran 2003**
- **User-defined reductions**
- **Sequentially consistent atomics**
  - To enforce sequential consistency when a specific storage location is accessed atomically.

ROYAL INSTITUTE OF TECHNOLOGY

Conclusion

- We have now covered major part of the OpenMP specification.
  - We've left off some minor details, but we've covered all the major topics … remaining content you can pick up on your own.
- Download the spec to learn more … the spec is filled with examples to support your continuing education.
  - www.openmp.org
- Get involved:
  - get your organization to join the OpenMP ARB.
  - Work with us through Community.

ROYAL INSTITUTE OF TECHNOLOGY

Examples and Exercises

- A zip file **omp-hands-on.zip** at the course website