PDC Summer School *"Introduction to High-Performance Computing"*,
KTH Stockholm, Aug. 2018

ROYAL INSTITUTE OF TECHNOLOGY

PDC CENTER FOR HIGH PERFORMANCE COMPUTING

Shared memory programming with **OpenMP**

Christoph Kessler, prof., Linköping University

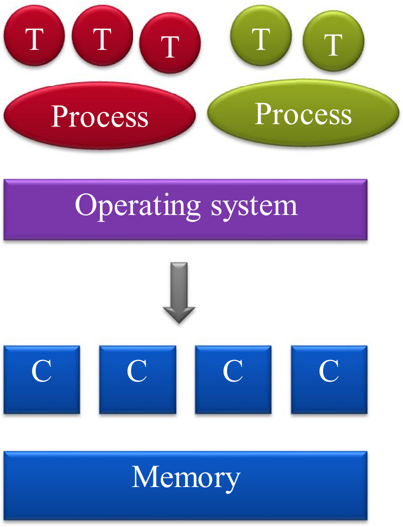
Most slides courtesy of **Mats Brorsson**, prof., KTH

LINKÖPING UNIVERSITY

SeRC Swedish e-Science Research Centre

SeSE

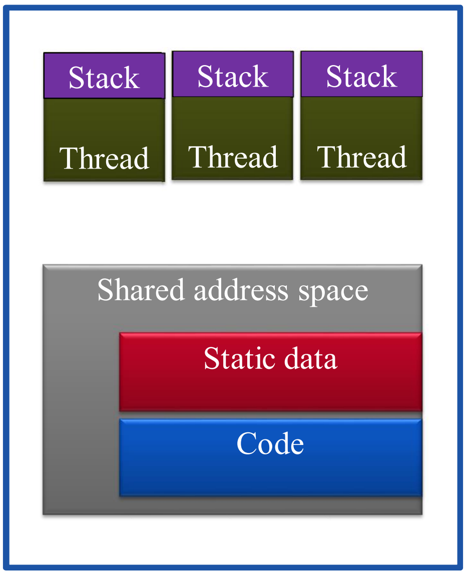
Shared-Memory Parallel Programming



Basic assumptions:

- Shared memory hardware support
  -There are multicores without shared memory also, but that's a different course
- An operating system that can provide
  - **Processes** with individual address spaces
  - **Threads** that share address space within a process
  - The OS schedules threads for execution on the **cores**

Processes vs. Threads


Stack
Thread
Stack
Thread
Stack
Thread
Shared address space
Static data
Code

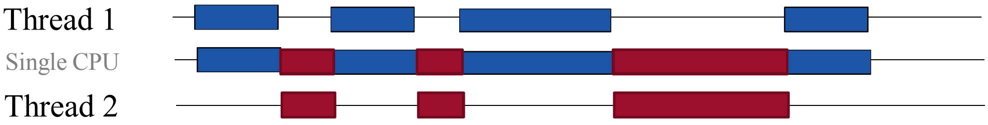
- A **process** is a container for a program in execution, with state, capabilities, and access rights to resources *shared* by all its threads:
  - Address space
  - Code
  - Data (static data, heap data)
  - Opened files
- A **thread** is a unit of control flow and CPU scheduling
  - Each thread has its own PC and stack
  - Any program starts its execution as a single **thread**
  - New threads can be created through OS calls

    → **Multithreaded process**

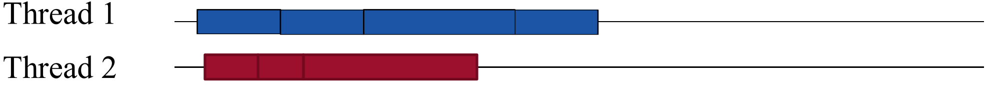
Concurrency vs. Parallelism

As defined by Sun/Oracle:

- **Concurrency**: A condition that exists when at least two tasks (threads, processes) are making progress. A more generalized form of parallelism that can include time-slicing as a form of *virtual parallelism*.
  - A property of the program/system

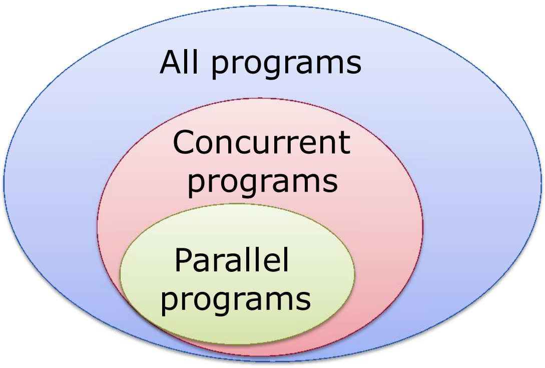

Thread 1
Single CPU
Thread 2

- **Parallelism**: A condition that arises when at least two tasks are executing *simultaneously*.
  - A run-time behaviour of executing a concurrent program
  - Requires more than one execution unit (CPU, core)


Thread 1
Thread 2

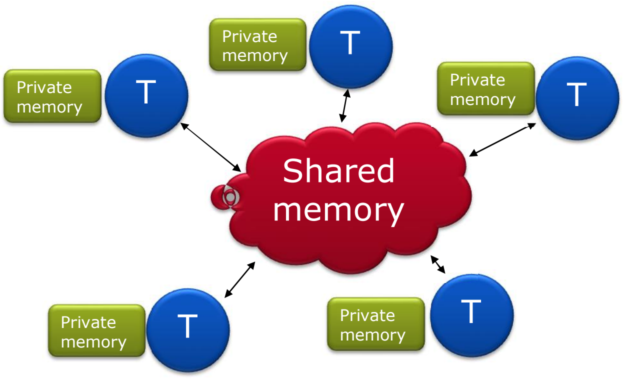
In other words…

- **Concurrency**: A condition of a system in which multiple tasks are logically active at one time.
- **Parallelism**: A condition of a system in which multiple tasks are actually active at one time.


All programs
Concurrent programs
Parallel programs

OpenMP

- A standardized (portable) way for writing concurrent programs for shared memory multiprocessors
  - For C/C++/Fortran

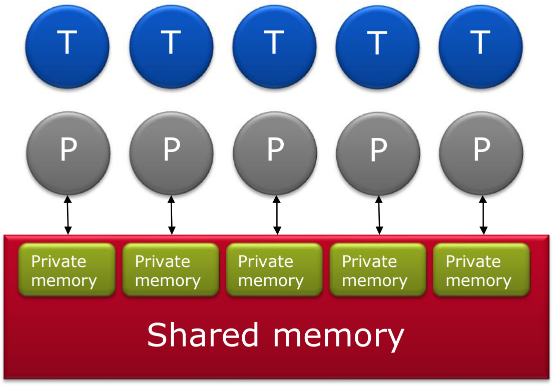

Private memory
T
Private memory
T
Private memory
T
Shared memory
Private memory
T
Private memory
T

Abstract machine model:

- Concurrent threads (~cores)
- A shared address space
- Private memory to each thread

OpenMP

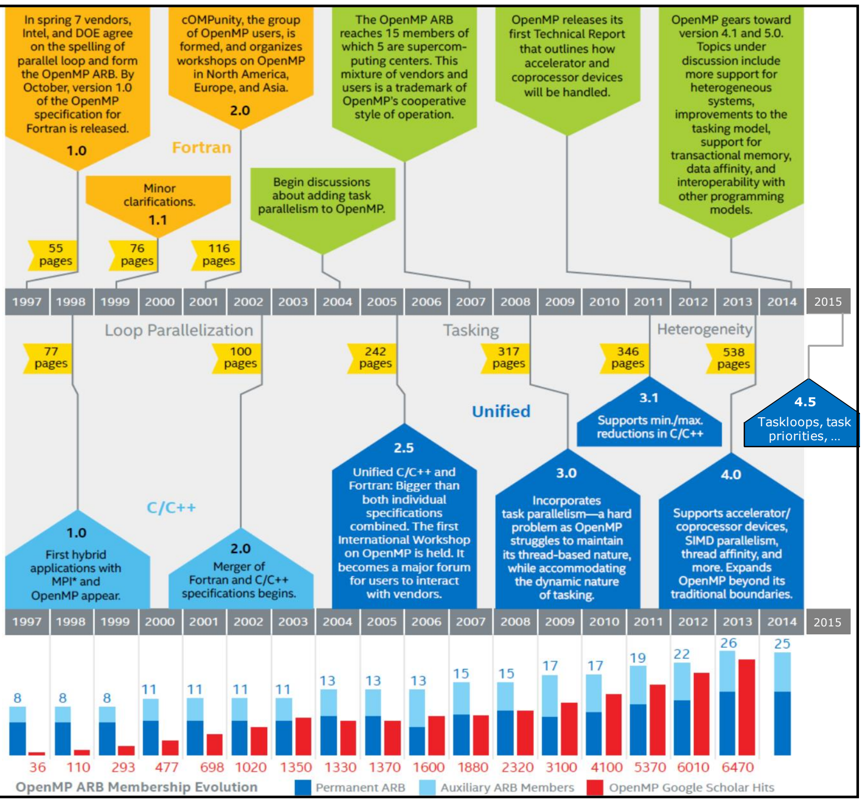
- A standardized (portable) way for writing concurrent programs for shared memory multiprocessors
  - For C/C++/Fortran



A more concrete model:

- Threads are scheduled to processors by the OS
- The private memory (for *thread*-private data) is located in the shared address space
- There may be local memory to each *processor*
  - Caches
  - NUMA

The evolution of OpenMP



In spring 7 vendors, Intel, and DOE agree on the spelling of parallel loop and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released. 1.0

Minor clarifications. 1.1

cOMPunity, the group of OpenMP users, is formed, and organizes workshops on OpenMP in North America, Europe, and Asia. 2.0

Begin discussions about adding task parallelism to OpenMP.

The OpenMP ARB reaches 15 members of which 5 are supercomputing centers. This mixture of vendors and users is a trademark of OpenMP's cooperative style of operation.

OpenMP releases its first Technical Report that outlines how accelerator and coprocessor devices will be handled.

OpenMP gears toward version 4.1 and 5.0. Topics under discussion include more support for heterogeneous systems, improvements to the tasking model, support for transactional memory, data affinity, and interoperability with other programming models.

1.0 First hybrid applications with MPI* and OpenMP appear.

2.0 Merger of Fortran and C/C++ specifications begins.

2.5 Unified C/C++ and Fortran: Bigger than both individual specifications combined. The first International Workshop on OpenMP is held. It becomes a major forum for users to interact with vendors.

3.0 Incorporates task parallelism—a hard problem as OpenMP struggles to maintain its thread-based nature, while accommodating the dynamic nature of tasking.

3.1 Supports min./max. reductions in C/C++

4.0 Supports accelerator/coprocessor devices, SIMD parallelism, thread affinity, and more. Expands OpenMP beyond its traditional boundaries.

4.5 Taskloops, task priorities, …

Agenda

Wednesday 16 Aug 2017

- 9-10 The basic concepts of OpenMP
- 10-12 Core features of OpenMP
  - » Parallel for (do) loops
  - » Tasks
- 13-14 Working with OpenMP

+ OpenMP programming exercises, by S. Markidis

Thursday 17 Aug

- 9-10 Task dependencies and accelerators
  - » OpenMP 4.0
- 10-12 Looking forward
  - » Alternatives to OpenMP
  - » Future OpenMP
  - » Recap

+ OpenMP advanced project, by S. Markidis

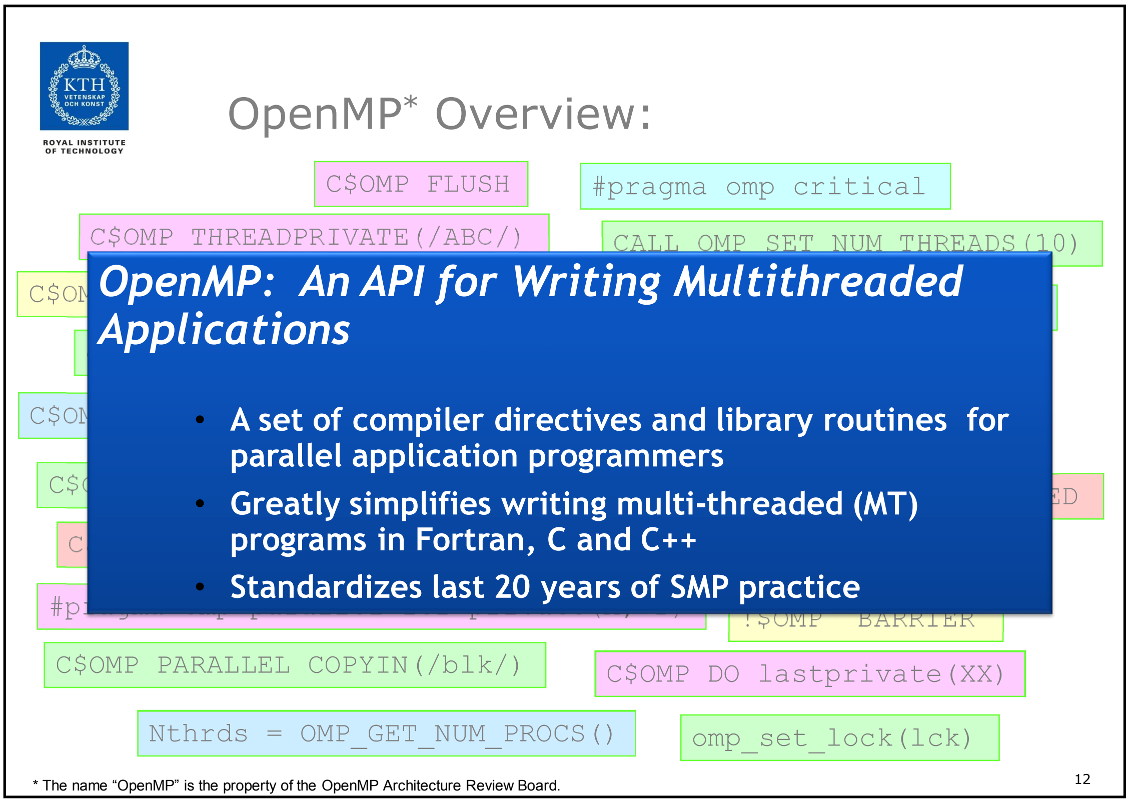
Caveat

- All programming examples are in C (C++)
- I can not provide equivalent examples in Fortran
- Ask if you are unsure about C

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.x: Task Dependences, Accelerators, SIMD …
- Alternatives to OpenMP
- Summary

OpenMP* Overview:

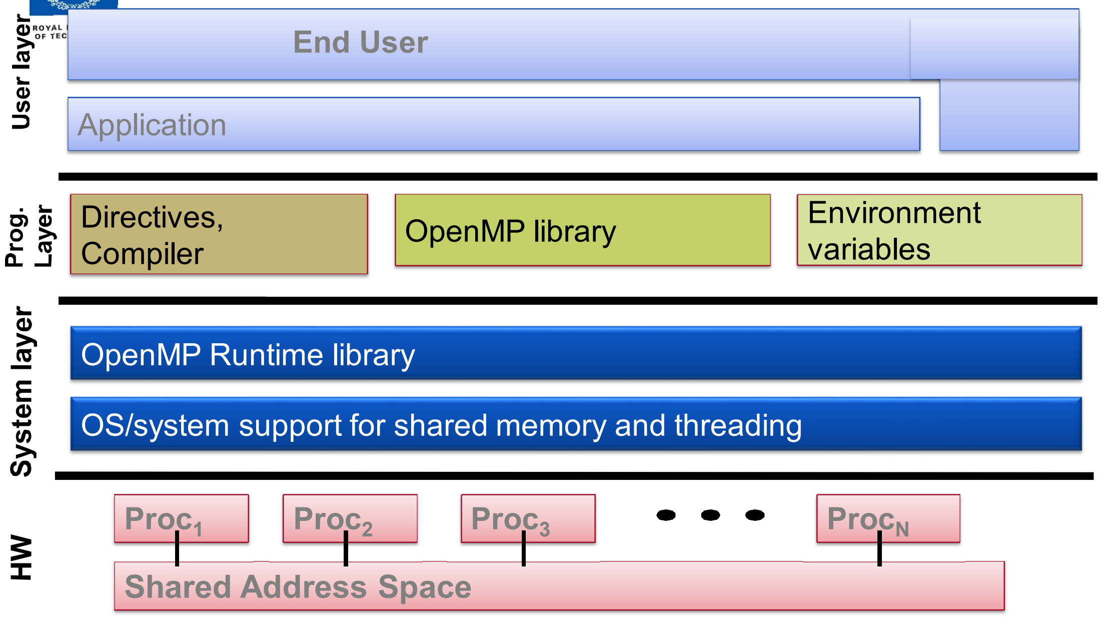

C$OMP FLUSH
#pragma omp critical
C$OMP THREADPRIVATE(/ABC/)
CALL OMP_SET_NUM_THREADS(10)
C$OM
C$OM
C$
C
ED
#p
!$OMP BARRIER
C$OMP PARALLEL COPYIN(/blk/)
C$OMP DO lastprivate(XX)
Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)

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



User layer

End User

Application

Prog. Layer

Directives, Compiler

OpenMP library

Environment variables

System layer

OpenMP Runtime library

OS/system support for shared memory and threading

HW

Proc$_1$ Proc$_2$ Proc$_3$ • • • Proc$_N$

Shared Address Space

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 is OK to have an exit() within the structured block.

Exercise 1, Part A: Hello world
Verify that your environment works

- Write a program that prints “hello world”.

```
int main()
{



     int ID = 0;

     printf(“ hello(%d) ”, ID);
     printf(“ world(%d) \n”, ID);

}
```

Exercise 1, Part B: Hello world
Verify that your OpenMP environment works

- Write a multithreaded program that prints “hello world”.

```
#include “omp.h”
void main()
{
 #pragma omp parallel
 {
     int ID = 0;

     printf(“ hello(%d) ”, ID);
     printf(“ world(%d) \n”, ID);
 }
}
```

Switches for compiling and linking

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

Exercise 1: Solution
A multi-threaded "Hello world" program

- Write a multithreaded program where each thread prints "hello world" and its thread ID.

```
#include "omp.h"

void main()
{

#pragma omp parallel
 {

    int ID = omp_get_thread_num();
    printf(" hello(%d) ", ID);
    printf(" world(%d) \n", ID);
  }
}
```

OpenMP include file

Parallel region with default number of threads

Runtime library function to return a thread ID.

End of the Parallel region

Sample Output:

```
hello(1) hello(0) world(1)

world(0)

hello (3) hello(2) world(3)

world(2)
```

OpenMP Overview:
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**: the program's outcome may change as the threads are scheduled differently.
    - No problem if all threads do read-only accesses
    - Write access → conflicts (data race) with reads and other writes (*order of accesses matters*)
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive, so:
  - Change how data is accessed to minimize the need for synchronization.

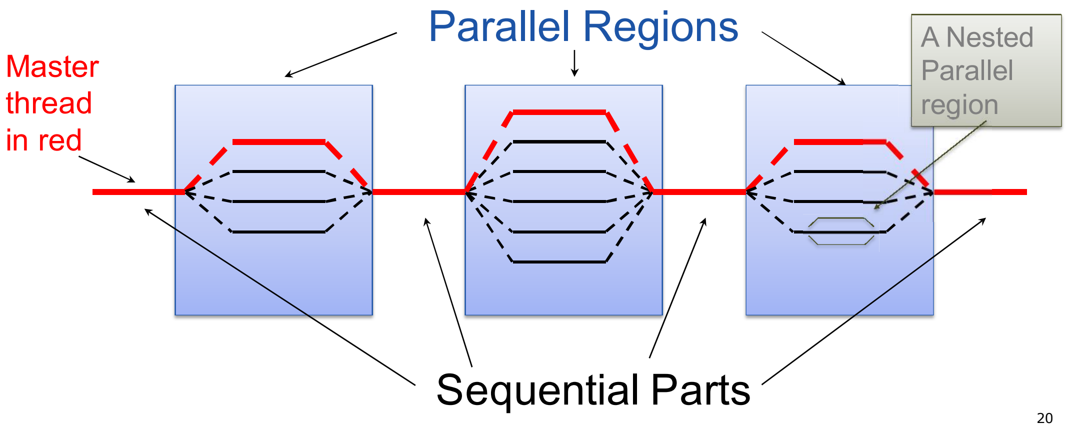
Outline

- Introduction to OpenMP
- ➡ Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.x: Task Dependences, Accelerators, SIMD …
- Alternatives to OpenMP
- Summary

OpenMP Programming Model:

◆ 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.


Parallel Regions
Master thread in red
A Nested Parallel region
Sequential Parts

Thread Creation: Parallel Regions

- You create threads in OpenMP* with the **parallel** construct.
- 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);
}
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

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

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

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

---

Thread Creation: Parallel Regions

- You create threads in OpenMP* with the **parallel** construct.
- For example, to create a 4-thread Parallel region:

```
double A[1000];

#pragma omp parallel num_threads(4)
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

clause to request a certain number of threads

Runtime function returning a thread ID

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

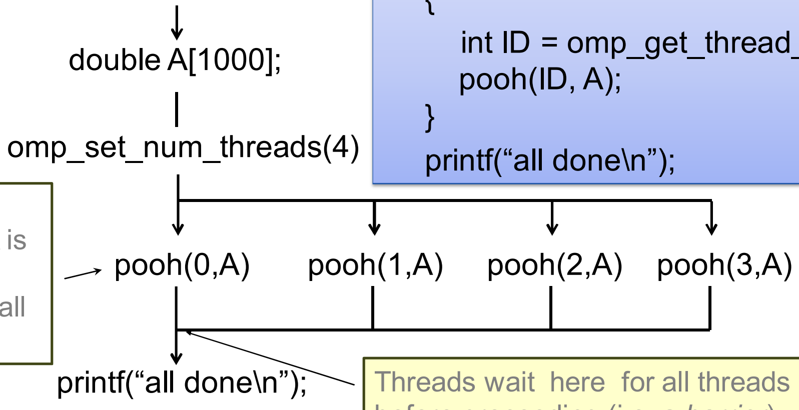
- Each thread calls pooh(ID,A) for ID = 0 to 3

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

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");
```



double A[1000];

omp_set_num_threads(4)

pooh(0,A) pooh(1,A) pooh(2,A) pooh(3,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*)

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

---

What an OpenMP compiler does

```
#pragma omp parallel num_threads(4)
{
   foobar ();
}
```

```
void body1()
{
   foobar ();
}

pthread_t tid[4];

for (int i = 1; i < 4; ++i)
   pthread_create( &tid[i],
                   0, body1,
                   0);

body1(); // by master thread

for (int i = 1; i < 4; ++i)
   pthread_join( tid[i] );
```

- The **OpenMP compiler** generates code logically analogous to that on the right of this slide, given an OpenMP pragma such as that on the top-left
- Here, only three threads are created because the last parallel section will be invoked from the parent thread.
- All known OpenMP implementations use a **thread pool**, so full cost of threads’ creation and destruction is not incurred anew for each executed parallel region.

OpenMP memory organization

Shared variables

- A change made to a shared variable by a thread is eventually visible for all threads in the program
- Memory consistency (time of updating visibility of changes) is implementation dependent, but programmable (→ `flush` directive)

`#pragma omp parallel shared ( varlist )`

Private variables

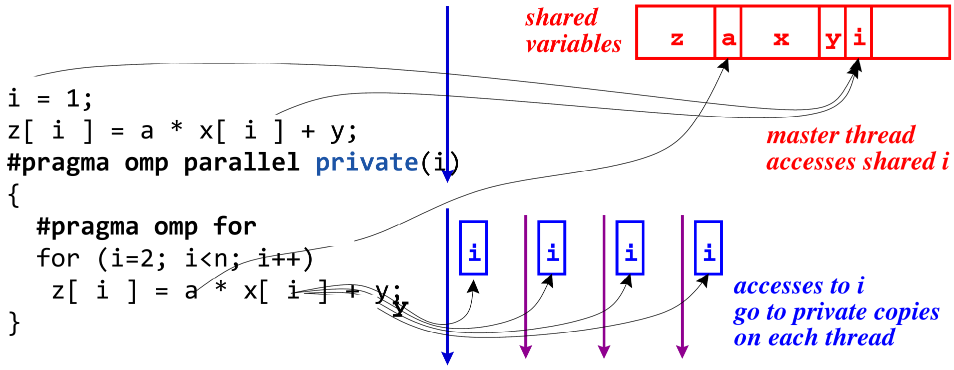
- One local instance per team thread
- Exclusive access by the thread

`#pragma omp parallel private ( varlist )`

allocates one copy of each variable in ***varlist*** on each thread's run-time stack

---

OpenMP memory organization (2)



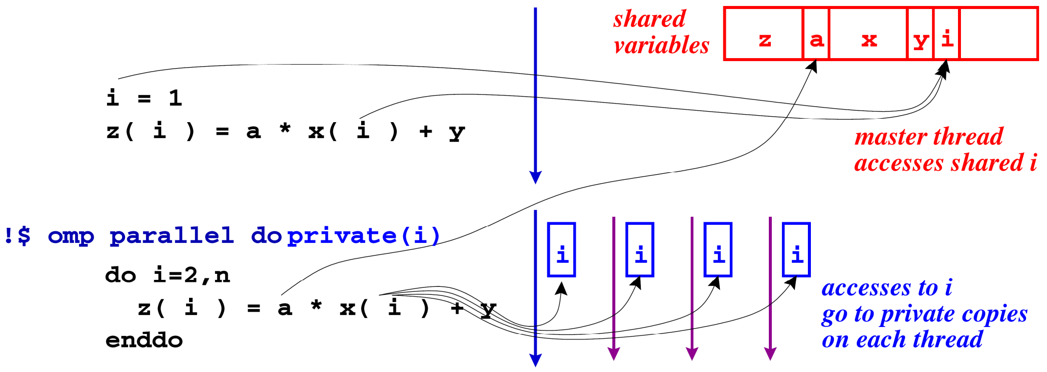
```
i = 1;
z[ i ] = a * x[ i ] + y;
#pragma omp parallel private(i)
{
  #pragma omp for
  for (i=2; i<n; i++)
   z[ i ] = a * x[ i ] + y;
}
```

shared variables
z
a
x
y
i
master thread accesses shared i
i
i
i
i
accesses to i go to private copies on each thread

Default sharity settings (for now, details come later):

- Global variables (in Fortran: common block) are shared by default.
- Variables allocated before entering the parallel region are shared
- Loop index variables of parallel do/for loops in parallel regions are private by default.

OpenMP memory organization (2) (Fortran)



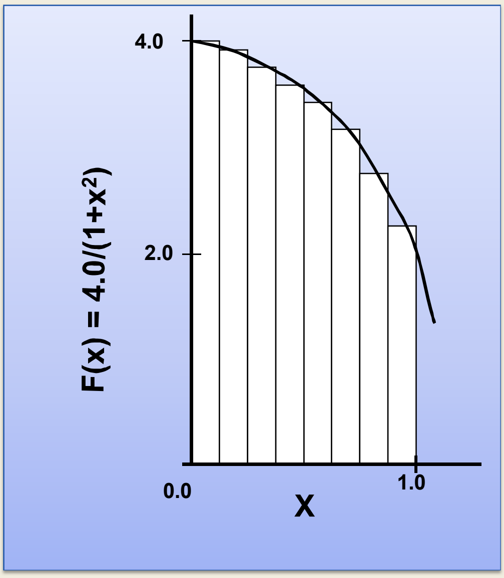
```
i = 1
z( i ) = a * x( i ) + y

!$ omp parallel do private(i)
      do i=2,n
        z( i ) = a * x( i ) + y
      enddo
```

shared variables
z
a
x
y
i
master thread accesses shared i
i
i
i
i
accesses to i go to private copies on each thread

Default sharity settings (for now, details come later):

- Global variables (in Fortran: common block) are shared by default.
- Variables allocated before entering the parallel region are shared
- Loop index variables of parallel do/for loops in parallel regions are private by default.

Exercises 2 to 4:
Numerical Integration



Mathematically, we know that:

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

We can approximate the integral of a function *F* as a sum of rectangles:

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

Where each rectangle has width $\Delta x = x / N$ and height $F(x_i)$ at the middle x_i of interval i.

28

Exercises 2 to 4: Serial PI Program

```
static long num_steps = 100000;
double step;
void main ()
{
            int  i;
            double  x,  pi,  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;
}
```

Exercise 2

- Create a parallel version of the pi program using the **parallel** construct.
- Pay close attention to shared versus private variables.
- In addition to a parallel construct, you will need these 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 seconds since a fixed point in the past

ROYAL INSTITUTE OF TECHNOLOGY

Outline

- Introduction to OpenMP
- Creating Threads
- ➡ Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.x: Task Dependences, Accelerators, SIMD
- Alternatives to OpenMP
- Summary

ROYAL INSTITUTE OF TECHNOLOGY

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
  - flush
  - locks (both simple and nested)

Discussed later

Synchronization: **critical**

- **Race condition, critical section/region**:

Example:

Threads wait their turn – only one at a time (atomically) adds its B to res

```
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
               res = res + B;
      }
}
```

Race Conditions lead to Nondeterminism

- Example: res = res + B   (res is shared, B is private)
- could be implemented in machine code as

```
39: register1 = res                  // load
40: register1 = register1 + B        // add
41: res = register1                  // store
```

Not atomic!

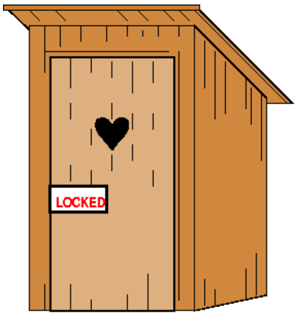
- Consider this execution interleaving, with “res = 5” initially:

```
39: thread1 executes register1 = res             { T1.register1 = 5 }
39: thread2 executes register1 = res             { T2.register1 = 5 }
40: thread1 executes register1 = register1 + 2   { T1.register1 = 7 }
40: thread2 executes register1 = register1 – 3   { T2.register1 = 2 }
41: thread1 executes res = register1             { res = 7 }
41: thread2 executes res = register1             { res = 2 }
```

- Compare to a different interleaving of memory accesses in time, *e.g.*, 39,40,41, 39,40,41…

  → Result depends on relative speed of the accessing threads (***race condition***) – can differ for different executions

Background: Critical Section



- **Critical Section**: A set of instructions, operating on (possibly modifying) shared data or resources, that should, in principle, be executed by a single thread at a time without interruption
  - **Atomicity** of execution
  - **Consistency**: inconsistent intermediate states of shared data shall not be visible to other threads outside

A *sufficient* method to guarantee atomic execution is:

Mutual exclusion: At most one thread should be allowed to operate inside at any time.

- General structure, with structured control flow:

... (non-critical section) …
Entry of critical section C
… critical section C: operation on shared data
Exit of critical section C
…

Synchronization: **critical**

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

Threads wait their turn – only one at a time (atomically) adds its B to res

```
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
            res = res + B;
      }
}
```

Synchronization: **critical**

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

Another Example:

Threads wait their turn – only one at a time calls enqueue(Q,B)

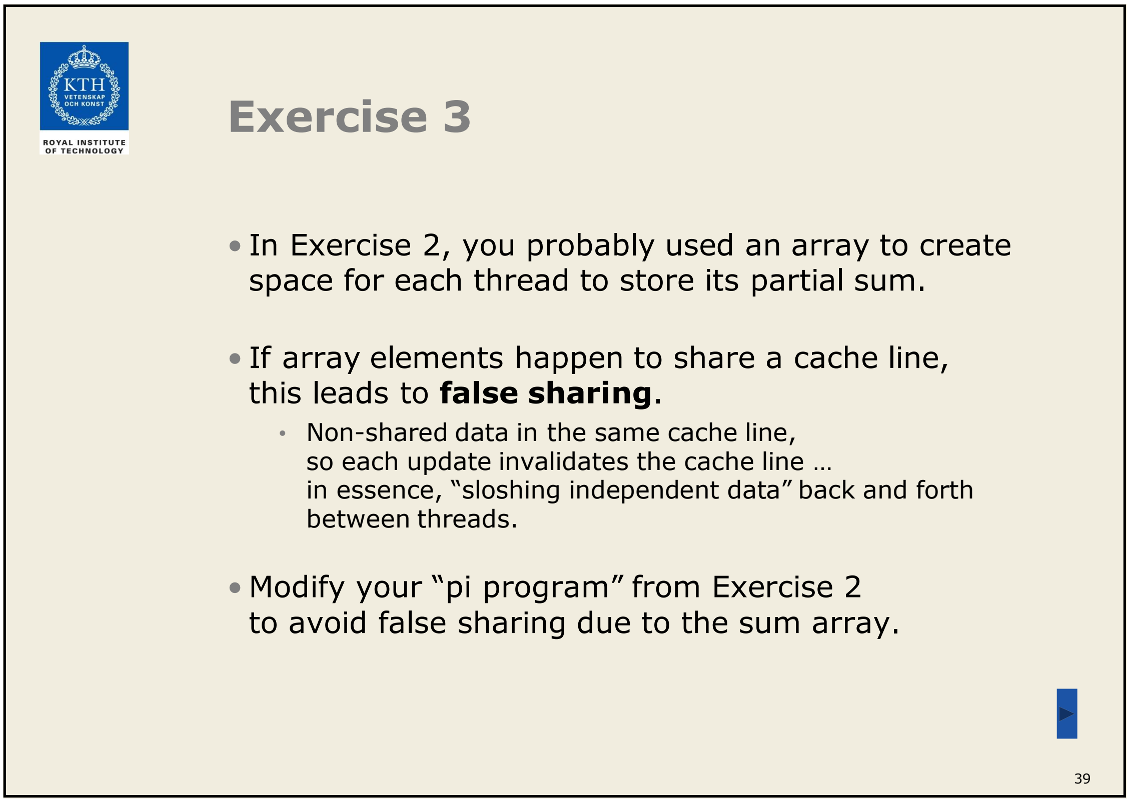
```
Queue Q;
#pragma omp parallel
{     float B;
      int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
      for (i=id; i<niters; i+nthrds)  {
              B =  big_job(i);
#pragma omp critical
              enqueue(Q,B);
      }
}
```

Synchronization: **Atomic**

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

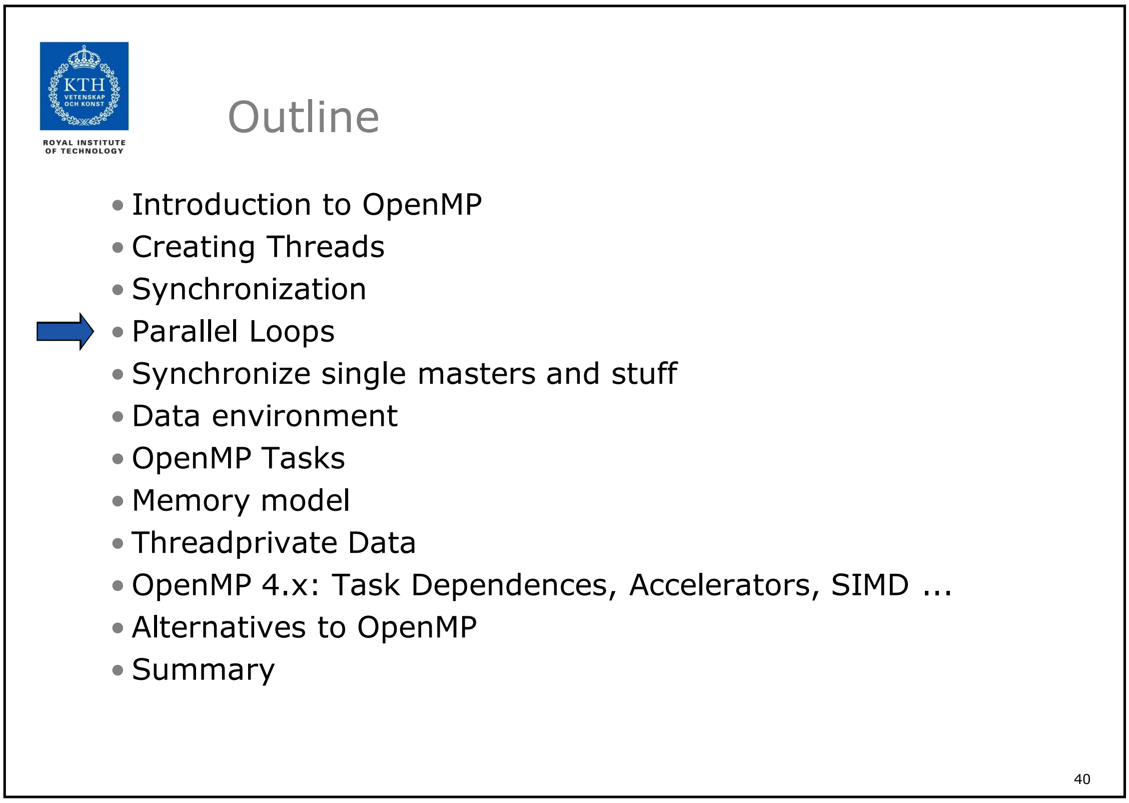
```
#pragma omp parallel
{
        double tmp, B;
        B =  DOIT();
        tmp = big_ugly( B );
 #pragma omp atomic
        X +=  tmp;
}
```

Atomic only protects the read/update of X



Exercise 3

- In Exercise 2, you probably used an 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.



Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.x: Task Dependences, Accelerators, SIMD …
- Alternatives to OpenMP
- Summary

SPMD vs. worksharing

- A **parallel** construct executes its body (the parallel region) in **SPMD** (“**Single Program Multiple Data**”) style … i.e., each thread in the team redundantly executes the same code,
  and no new threads are created or removed dynamically.

- How do you split up pathways through the code between threads within a team?
  - This is called **worksharing**
    - By hand (as in Exercise 2) **?**
      Possible, but cumbersome, low-level, error-prone …
    - By the **work-sharing constructs** in OpenMP
      - Parallel loop construct
      - Parallel sections/section constructs
      - Single construct
      - …
      - Task construct …. Available in OpenMP 3.0

Discussed later

The loop worksharing construct

- 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 loop index variable i is made “private” to each thread by default. You could do this explicitly with a “**private(i)**” clause

ROYAL INSTITUTE OF TECHNOLOGY

Loop worksharing construct
A motivating example

Sequential code

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

OpenMP parallel region with hand-programmed worksharing

```
#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

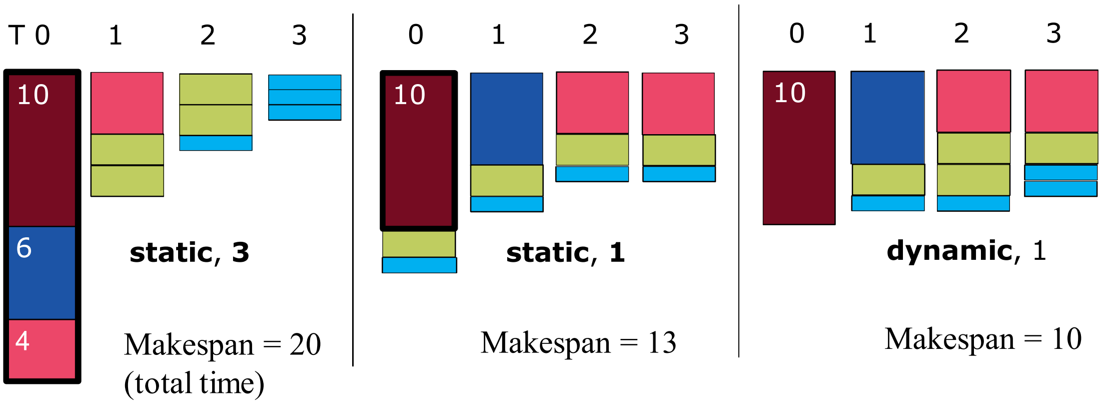
Loop worksharing construct:
The **schedule** clause

The **schedule** clause (after **for**/**do**)
affects how loop iterations are mapped onto threads

- **schedule**(**static** [,chunk])
  - Deal-out blocks of iterations of size “chunk” to each thread.
- **schedule**(**dynamic**[,chunk])
  - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
- **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.
- **schedule**(**runtime**)
  - Schedule and chunk size taken from the OMP_SCHEDULE environment variable (or the runtime library … for OpenMP 3.0).
- **schedule** (**auto**)
  - Schedule is up to the run-time to choose (does not have to be any of the above).

Why different schedules?

- Consider a loop with 12 iterations with the following execution times
  - 10, 6, 4, 4, 2, 2, 2, 2, 1, 1, 1, 1
- Assume four threads (cores)


T 0
1
2
3
10
6
4
static, **3**
Makespan = 20
(total time)
0
1
2
3
10
static, **1**
Makespan = 13
0
1
2
3
10
dynamic, 1
Makespan = 10

Loop work-sharing constructs:
The schedule clause

Schedule Clause	When To Use
STATIC	Iteration times known by the programmer to be (almost) equal
DYNAMIC	Unpredictable, highly variable work per iteration – need for dynamic load balancing
GUIDED	Special case of dynamic scheduling to reduce scheduling overhead
AUTO	The run-time system tries to “learn” from previous executions of the same loop

No overhead at runtime: scheduling done at compile-time

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

Combined parallel/worksharing constructs

- 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 (use a profiler)
- If the loop iterations are *independent* (without loop-carried dependencies)
  (or the loop can be rewritten to have independent iterations), they can safely execute in any order or in parallel
  - Place the appropriate OpenMP directive and test

```
int i, j, A[MAX];
 j = 5;
 for (i=0; i<MAX; i++) {
     j +=2;
     A[i] = big(j);
 }
```

Note: loop index “i” is private by default

```
int i,  A[MAX];
#pragma omp parallel for
 for (i=0; i<MAX; i++) {
     int j = 5 + 2*(i+1);
     A[i] = big(j);
 }
```

Remove loop carried dependence

Nested loops

- For perfectly nested rectangular loops we can parallelize multiple loops in the nest with the **collapse** clause:

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

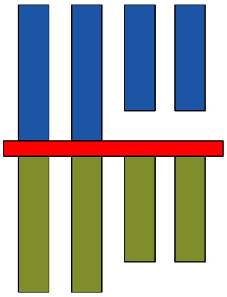
Number of loops to be parallelized, counting from the outside

- Will form a single flat loop of length NxM iterations and then parallelize that.
- Useful if N almost equals no. of threads so that parallelizing the outer loop only would make balancing the load difficult

Sequencing parallel loops (1)

Static: Guarantee that the same schedule is used in the two loops

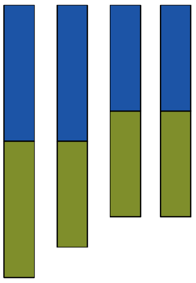
```
#pragma omp for schedule(static)
for (i=0; i<n; i++){
    a[i] = ....
}
#pragma omp for schedule(static)
for (i=0; i<n; i++) {
    .... = a[i]
}
```



Sequencing parallel loops (2)

Static: Guarantee 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]
}
```



- **nowait** clause suppresses the implicit barrier synchronization at the end of the annotated loop

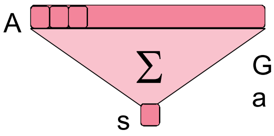
Remark: There may be a more cache-efficient solution for this case. Which one?

Reduction

- How do we handle this case?

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

- We combine values in a single *accumulation variable* (sum)
  - There is a true dependence (*data flow dependence*) between loop iterations (i.e., a *loop-carried dependence*) that can't be trivially removed
- This is a very common situation
  - it is called a *reduction*.



Global sum is 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.
  - Afterwards, 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, s=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:s)
 for (i=0; i< MAX; i++) {
     s + = A[i];
 }
 ave = s / MAX;
```

Reduction operands and initial-values

- Many different associative operators can be used with reduction
- Initialization values are the ones that make sense mathematically.

Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

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.

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.

Serial Pi program

```
static long num_steps = 100000;
double step;
void main ()
{
  int i;
  double x, pi, 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;
}
```

Parallel Pi program

```
static long num_steps = 100000;
double step;
void main ()
{
  int i;
  double x, pi, sum = 0.0;

  step = 1.0/(double) num_steps;
#pragma omp parallel for reduction(+:sum)
  for (i=0; i< num_steps; i++) {
     double x = (i+0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

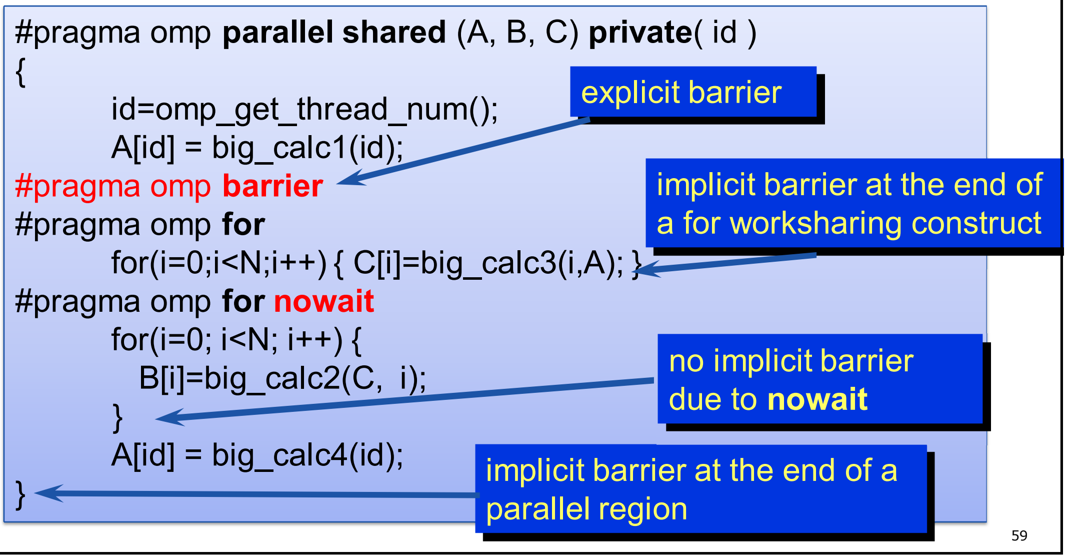
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- ➡ Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.x: Task Dependences, Accelerators, SIMD …
- Alternatives to OpenMP
- Summary

ROYAL INSTITUTE OF TECHNOLOGY

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);
}
```

explicit barrier

implicit barrier at the end of a for worksharing construct

no implicit barrier due to **nowait**

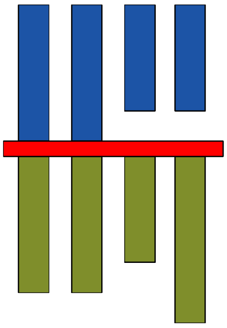
implicit barrier at the end of a parallel region

ROYAL INSTITUTE OF TECHNOLOGY

Recall: Sequencing parallel loops (1)

Static guarantees that the same schedule is used in the two loops

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

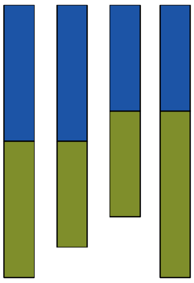


-

Recall: Sequencing parallel loops (2)

Static guarantees 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]
}
```



- **nowait** clause suppresses the implicit barrier synchronization at the end of the annotated loop

Remark: There may be a more cache-efficient solution for this case. Which one?

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).
- Is *not* considered a work-sharing construct, and could thus be nested inside other worksharing constructs (e.g. parallel loops).

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

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).

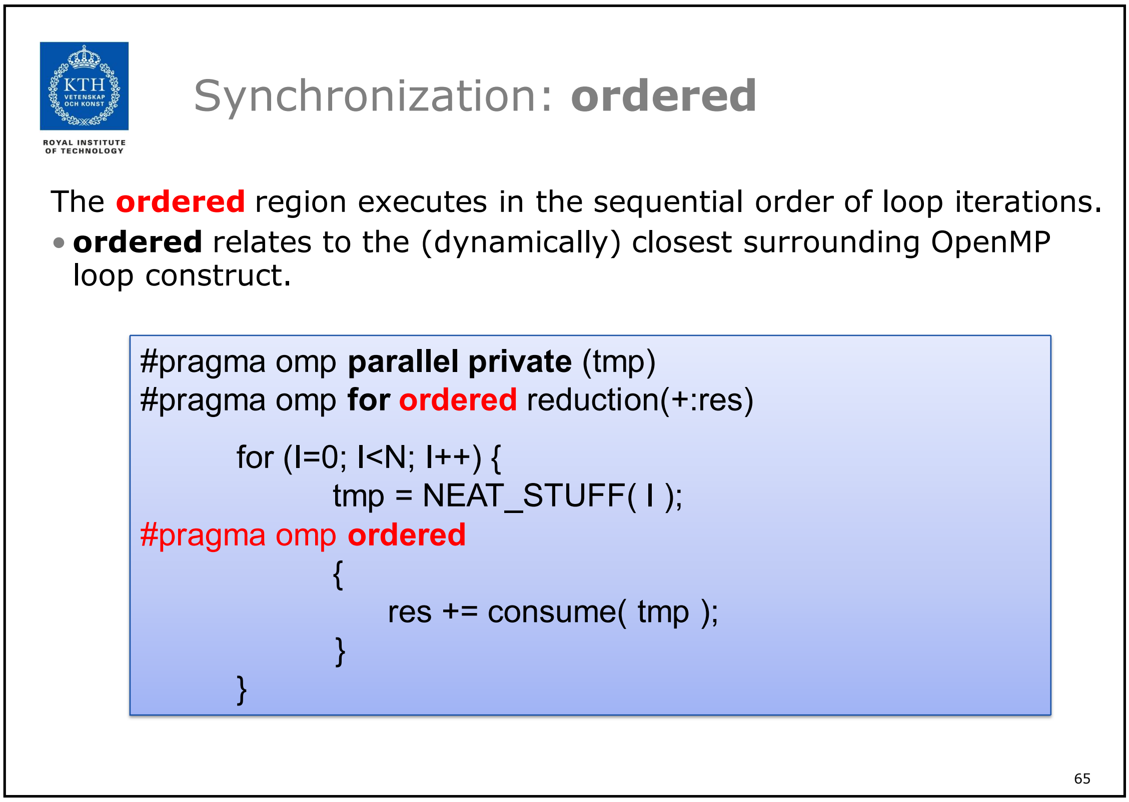
```
#pragma omp parallel
{
        do_many_things();
#pragma omp single
        {
            exchange_boundaries();
        }
        do_many_other_things();
}
```

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.

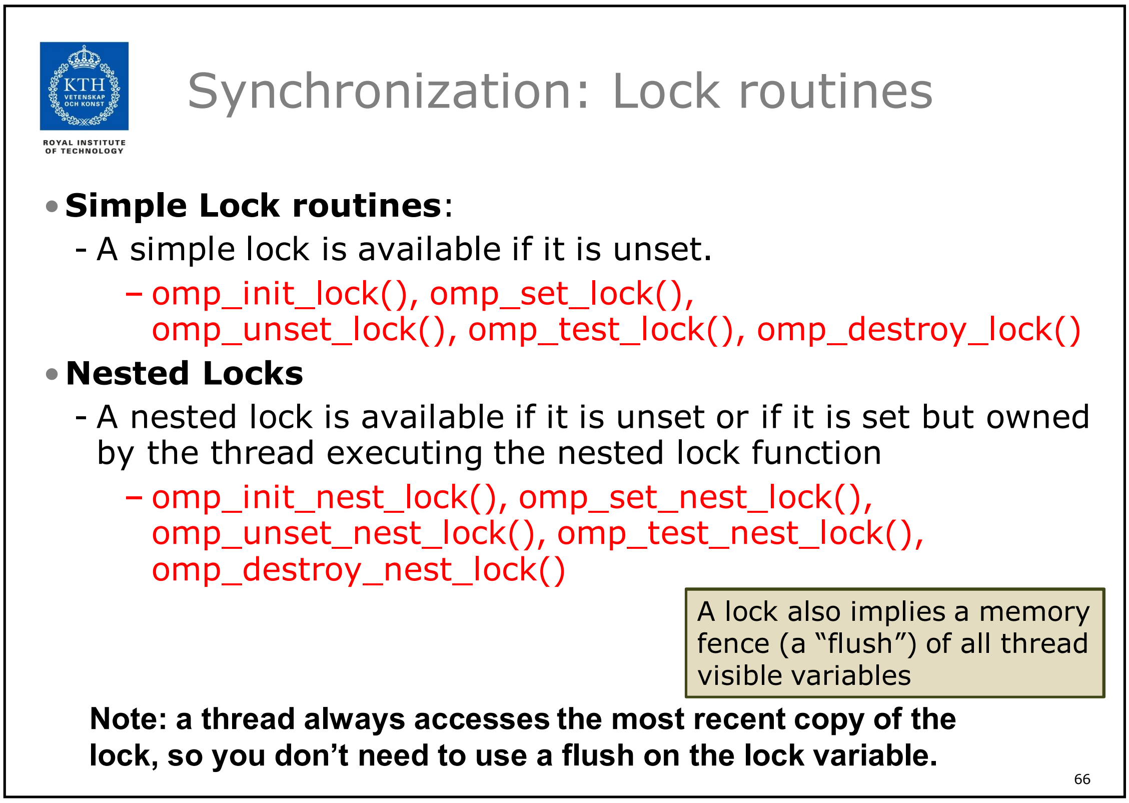


Synchronization: **ordered**

The **ordered** region executes in the sequential order of loop iterations.

- **ordered** relates to the (dynamically) closest surrounding OpenMP loop construct.

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

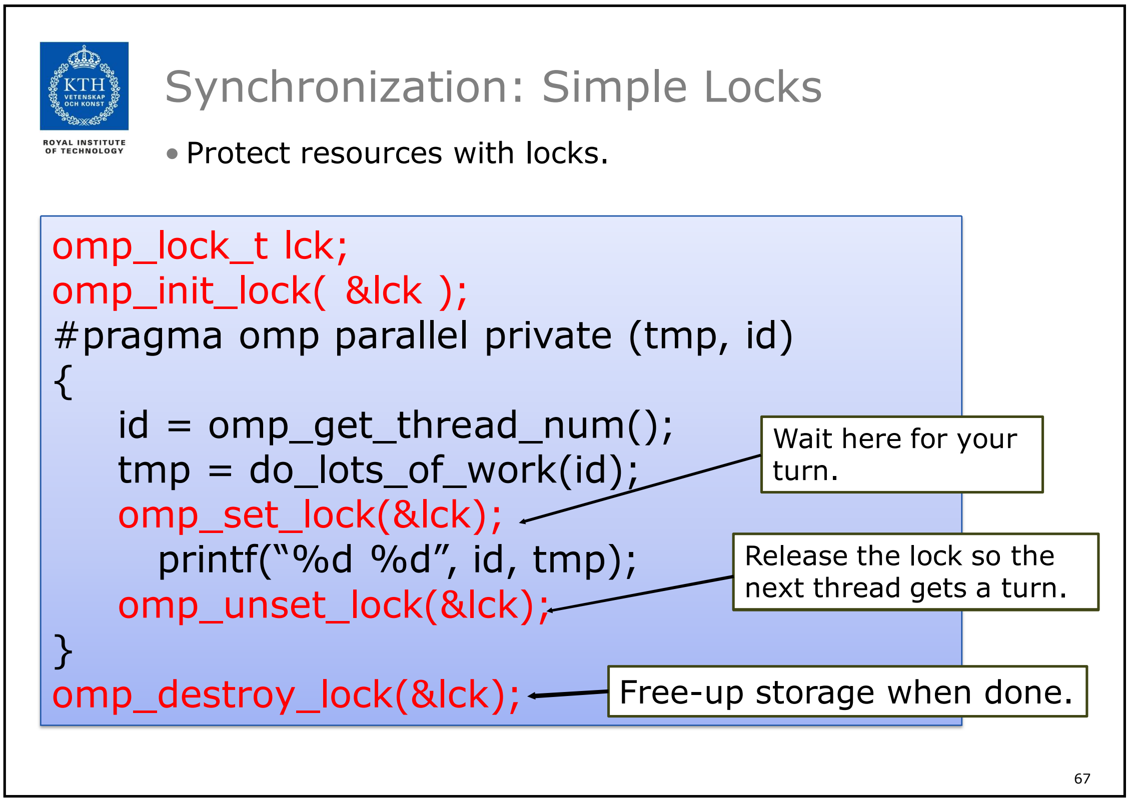


Synchronization: Lock routines

- **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()

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

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

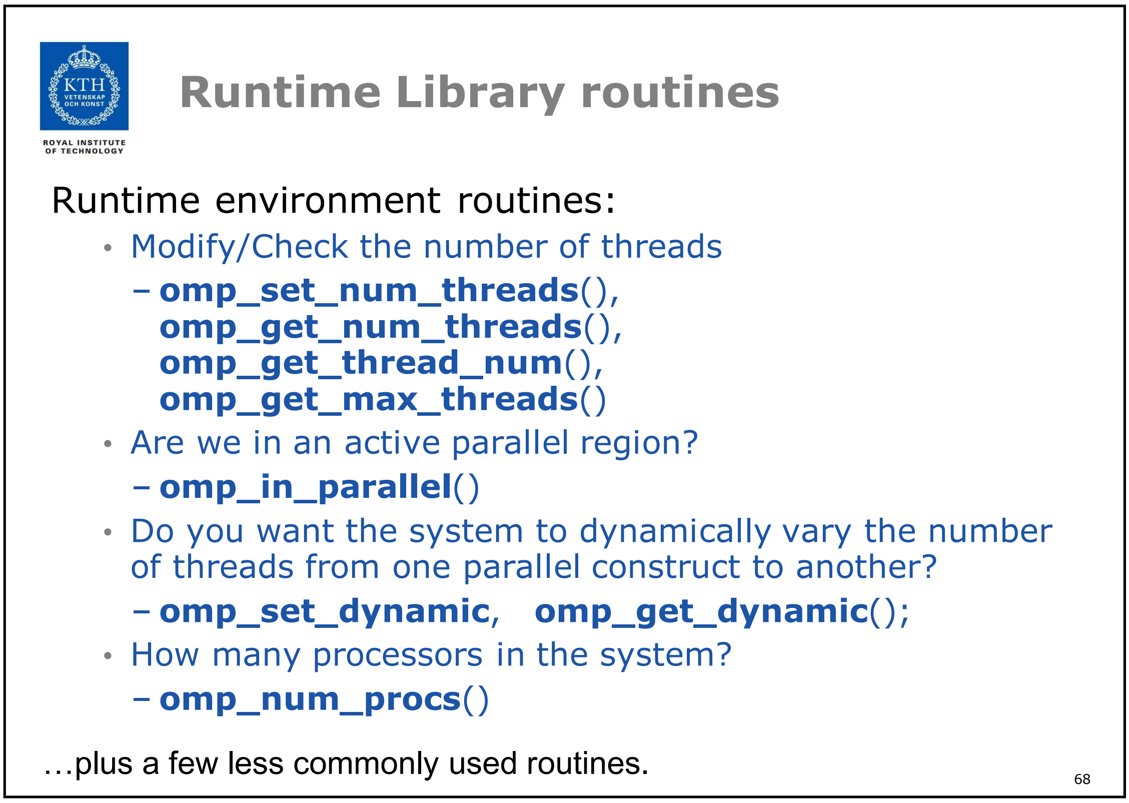
- Protect resources with locks.

```
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.



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 );
     }
}
```

(1) Disable dynamic adjustment of the number of threads.

(2) Request as many threads as you have processors.

Protect this op since memory stores are not atomic

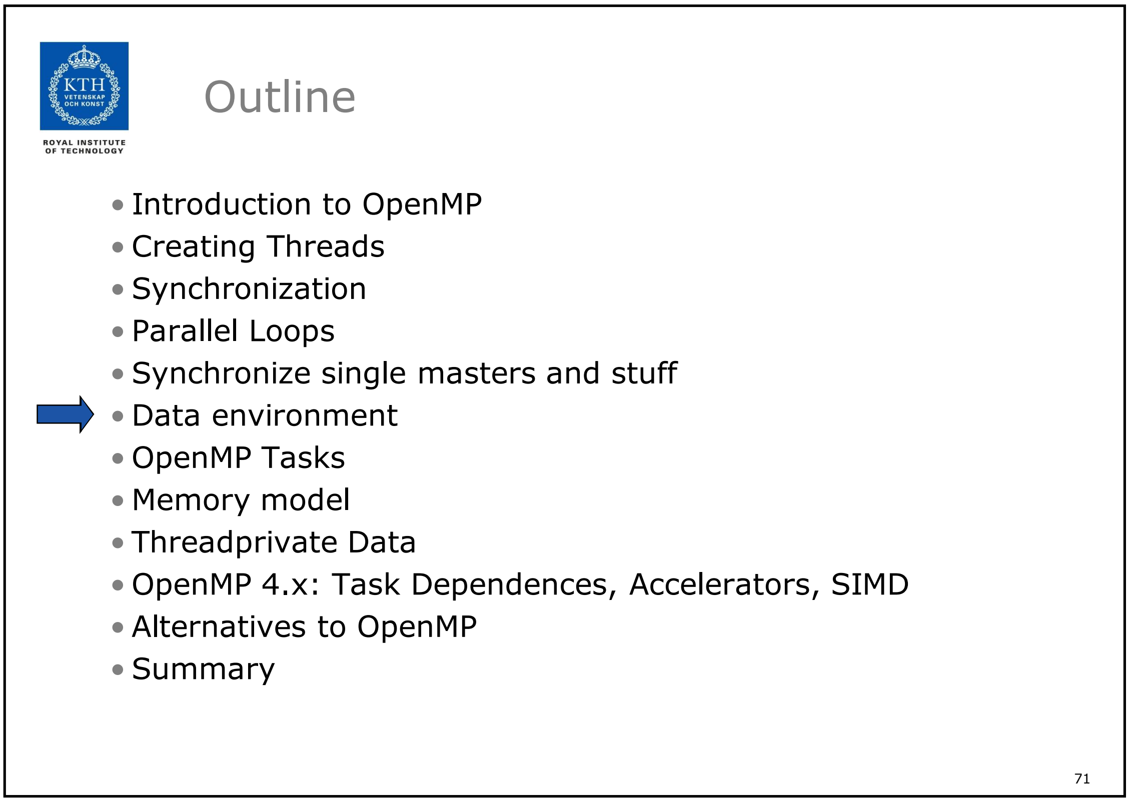
(3) Even in this case, the system may give you fewer threads than requested. If the precise number of threads matters, test for it and respond accordingly.

Environment Variables

- Set the default number of threads to use.
  - **OMP_NUM_THREADS** *int_literal*

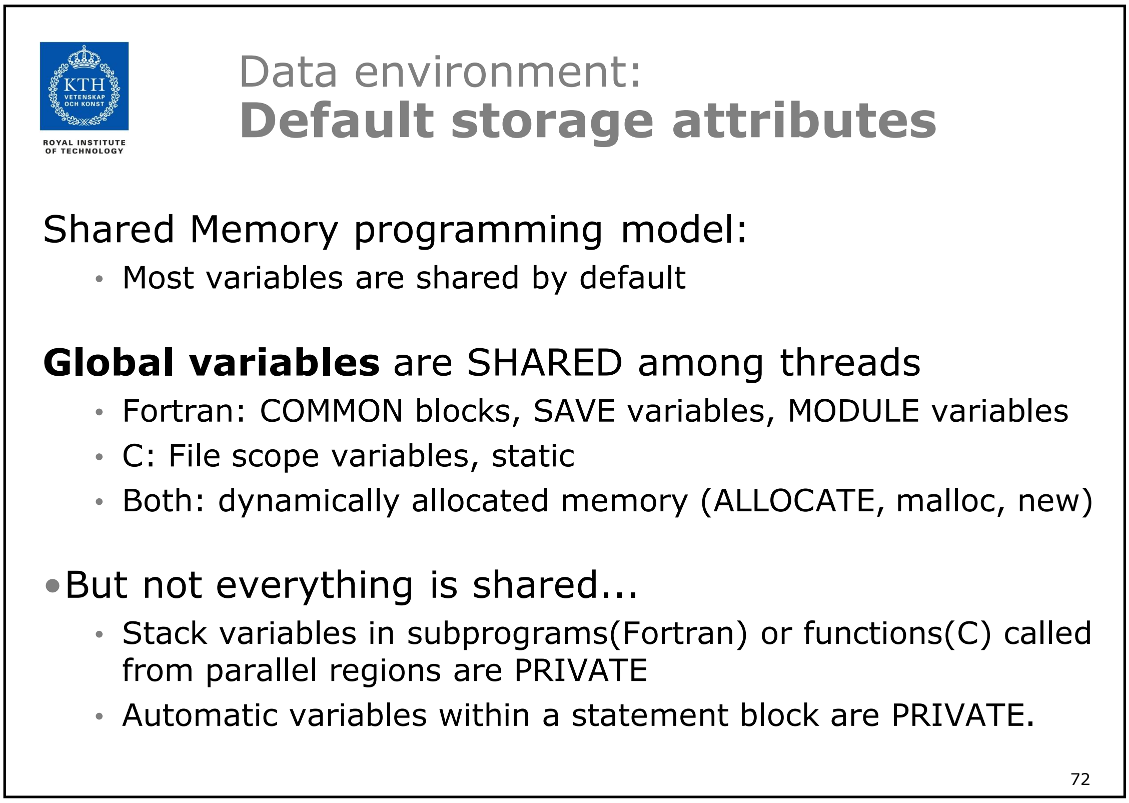
- Control how "omp for schedule(RUNTIME)" loop iterations are scheduled.
  - **OMP_SCHEDULE** "schedule[, chunk_size]"

… plus several less commonly used environment variables.



Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.x: Task Dependences, Accelerators, SIMD
- Alternatives to OpenMP
- Summary



Data environment:
Default storage attributes

Shared Memory programming model:

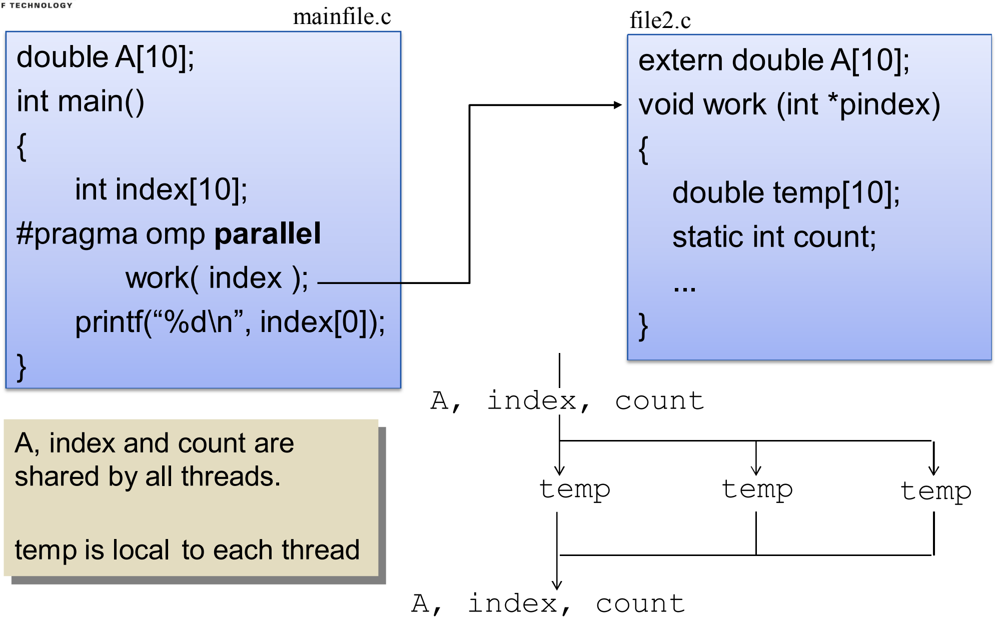
- Most variables are shared by default

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



mainfile.c

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

file2.c

```
extern double A[10];
void work (int *pindex)
{
    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**

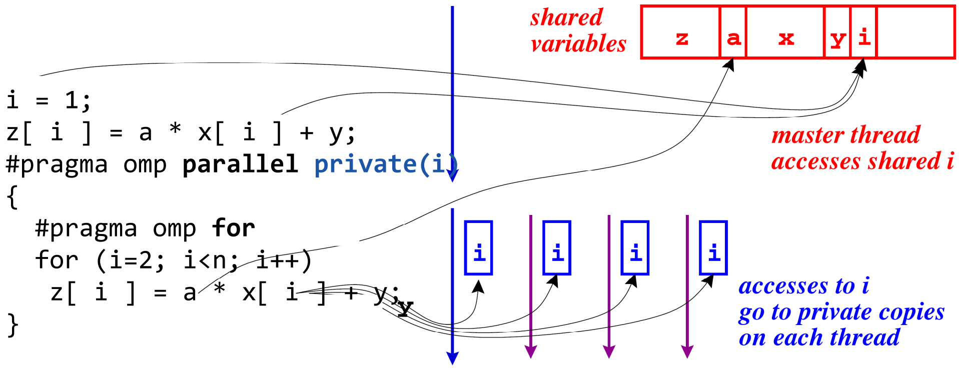
*All the clauses on this page apply to the OpenMP construct, NOT to the entire region.

- 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 data clauses apply to parallel constructs and worksharing constructs except "shared" which only applies to parallel constructs.

OpenMP memory organization (2)



```
i = 1;
z[ i ] = a * x[ i ] + y;
#pragma omp parallel private(i)
{
  #pragma omp for
  for (i=2; i<n; i++)
   z[ i ] = a * x[ i ] + y;
}
```

shared variables
z
a
x
y
i
master thread accesses shared i
i
i
i
i
accesses to i go to private copies on each thread

Default sharity settings (for now, details come later):

- Global variables (in Fortran: common block) are shared by default.
- Variables allocated before entering the parallel region are shared
- Loop index variables of do/for loops in parallel regions are private by default.

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 was undefined after the region

```
void wrong()
{
    int tmp = 0;
#pragma omp parallel 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: **Private** Clause - When is the original variable valid?

- The original variable's value is unspecified in OpenMP 2.5.
- In OpenMP 3.0, if it is referenced outside of the construct
  - Implementations may reference the original variable or a copy ….. A dangerous programming practice!

```
int tmp;   // shared
void danger()
{
  tmp = 0;
#pragma omp parallel private(tmp)
     work();
  printf("%d\n", tmp);
}
```

Refers to "original" tmp.

OpenMP2.5: unspecified value

```
extern int tmp;
void work()
{
      tmp = 5;
}
```

unspecified which copy of tmp is accessed.

(To make sure, pass a pointer to the privatized tmp if that was intended.)

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 parallel 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 privatized global variable from the thread executing the *sequentially* last iteration (here, 999) of a workshared for-loop to the corresponding global variable (here, tmp).

```
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.

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: **Default** Clause Example

```
    itotal = 1000
#pragma omp parallel private(np, each)
{
    np = omp_get_num_threads()
    each = itotal/np
    ………
}
```

```
    itotal = 1000
#pragma omp parallel default(private) shared(itotal)
{
    np = omp_get_num_threads()
    each = itotal/np
    ………
}
```

These two code fragments are equivalent

Exercise 6: **Molecular dynamics**

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.
- Use tools to find data races
- Experiment with different schedules kinds.

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?

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- ➡ OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.x: Task Dependenes, Accelerators, SIMD …
- Alternatives to OpenMP
- Summary

A motivational example: List traversal

How to parallelize this code
with known constructs of OpenMP?

```
p = head;
while (p) {
   process(p);
   p = p->next;
}
```

- Remember, the loop worksharing construct only works with loops for which the number of loop iterations can be represented by a closed-form expression at compile time.
  - Must be known at loop entry (and remain fixed)
- While-loops are not covered ☹

List traversal with for-loops

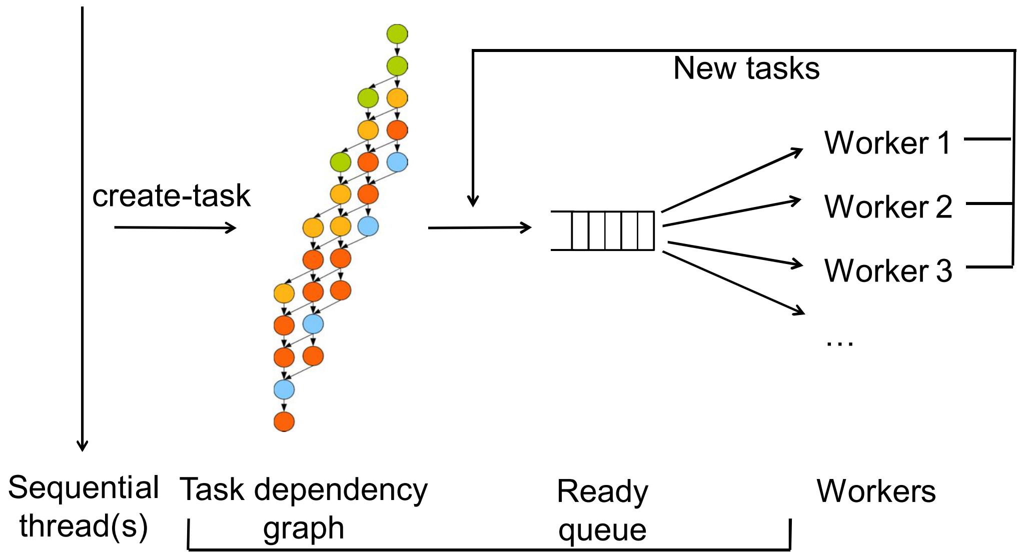
```
while (p != NULL) {
   p = p->next;
   count++;
}

p = head;
for(i=0; i<count; i++) {
   parr[i] = p;
   p = p->next;
}

#pragma omp parallel for
   for(i=0; i<count; i++)
      processwork(parr[i]);
```

- Find out the length of list
- Copy pointer to each node in an array
- Process nodes in parallel with a for loop

OpenMP tasks

- Introduced with OpenMP 3.0
- 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

Task parallelism


New tasks
create-task
Worker 1
Worker 2
Worker 3
…
Sequential thread(s)
Task dependency graph
Ready queue
Workers

An example of task-parallelism

Starting code, e.g. in main():

```
...
fib(N);
...
```

The (naïve) sequential Fibonacci calculation

```
int fib( int n ) {
 if( n<2 ) return n;
 else {
       int a,b;

       a = fib(n-1);

       b = fib(n-2);

       return a+b;
 }
}
```

Parallelism in fib:

- The two recursive calls are *independent* and can be computed in *any order* and *in parallel*
- It helps that fib is side-effect free, but disjoint side-effects are OK

The need for synchronization:

- The return statement must be executed after both recursive calls have been completed because of *data-dependence* on a and b.

A task-parallel fib in OpenMP 3.0

Starting code:

```
...
#pragma omp parallel
#pragma omp single
            fib(N);
...
```

```
int fib( int n ) {
  if ( n<2 ) return n;
  else {
        int a,b;
#pragma omp task shared(a) if (n>10)
        a = fib(n-1);
#pragma omp task shared(b) if (n>10)
        b = fib(n-2);
#pragma omp taskwait
        return a+b;
  }
}
```

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

Tasks and OpenMP

- Tasks have been fully integrated into OpenMP (3.0)

- Fundamental concept - OpenMP has always had tasks, we just never called them that.
  - Thread encountering a **`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!

`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  )
```

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
  - Control task granularity:
    Sequentialize execution when the cost of deferring the task is too large compared to the cost of executing the task code
  - Control cache and memory affinity

```
#pragma omp task shared(a) if (n>10)
            a = fib(n-1);
```

When/where are tasks completed?

- 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 directly in the current task, not to "descendants".

Example – pointer chasing on a list 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 – 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: Post-order 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

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

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)

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!

Data Sharing: tasks (OpenMP 3.0)

- 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

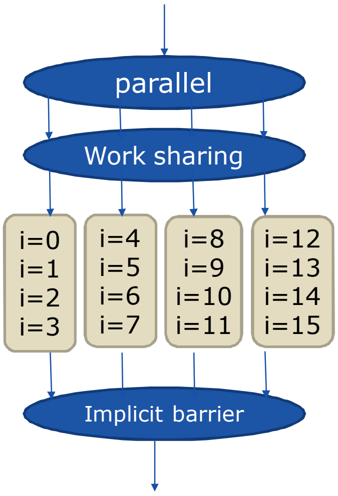
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- ➡ Recap of OpenMP basics
- Memory model
- Threadprivate Data
- OpenMP 4.x: Task Dependences, Accelerators, SIMD …
- Alternatives to OpenMP
- Summary

OpenMP basics: **for** loops

```
#pragma omp parallel
#pragma omp for
 for (i=0; i < 16; i++)
   c[i] = b[i] + a[i];
```

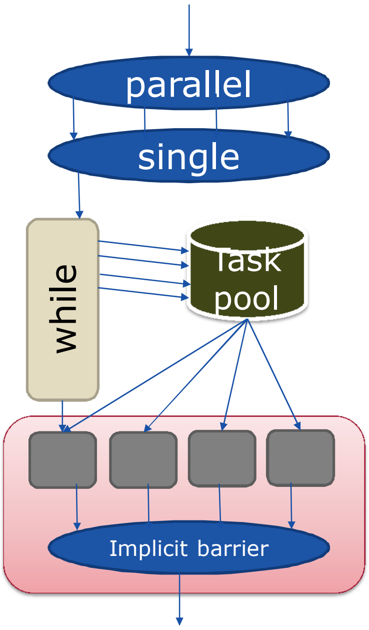
- Threads are assigned independent set of iterations
- Work-sharing construct


parallel
Work sharing
i=0
i=1
i=2
i=3
i=4
i=5
i=6
i=7
i=8
i=9
i=10
i=11
i=12
i=13
i=14
i=15
Implicit barrier

OpenMP basics: **tasks**

```
#pragma omp parallel
#pragma omp single
{ p = head;
  while( p ) {
  #pragma omp task firstprivate(p)
        process(p);
     p = p->next;
  }
}
```

- Threads are *task workers.*
- The code in a parallel region constitutes an *implicit task* for each thread in the team.
- When an implicit task is done, the worker fetches a new task from the task pool, if any.

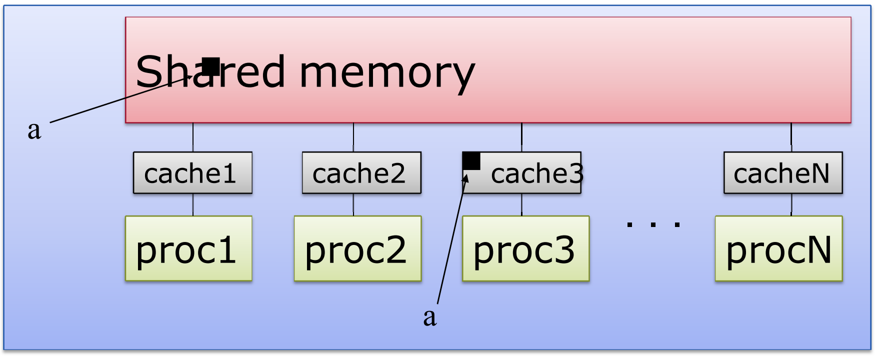

parallel
single
while
Task pool
Implicit barrier

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Recap of OpenMP basics
- Memory model
- Threadprivate Data
- OpenMP 4.x: Task Dependences, Accelerators, SIMD …
- Alternatives to OpenMP
- Summary

OpenMP Memory Model

- OpenMP supports a shared memory model.
- All threads share an address space, but it can get complicated:


Shared memory
a
cache1
cache2
cache3
cacheN
proc1
proc2
proc3
. . .
procN
a

- Multiple copies of data may be present in various levels of cache, or in registers.

OpenMP and Relaxed Consistency

OpenMP supports a **relaxed-consistency shared memory model**.

- Threads can maintain a temporary view of shared memory which is not consistent with that of other threads.
- These temporary views are made consistent only at certain points in the program.
- The operation which enforces consistency is called the **flush** operation.

Flush operation

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
  - All previous read/writes by this thread have completed and are visible to other threads
  - No subsequent read/writes by this thread have occurred
  - A flush operation is analogous to a fence in other shared memory API’s
    - Sometimes also referred to as a memory barrier

Flush and Synchronization

A **flush** operation is implied by OpenMP synchronizations, e.g.

- at entry/exit of parallel regions
- at implicit and explicit barriers
- at entry/exit of critical regions
- whenever a lock is set or unset

….

(but *not* at entry to worksharing regions
or entry/exit of master regions)

Example:

producer-consumer pattern

Thread 0

```
a = foo();
flag = 1;
```

Thread 1

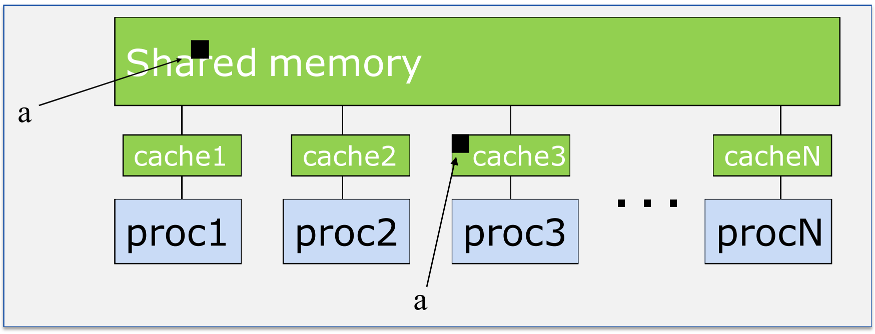
```
while (!flag) ;
b = a;
```

This is incorrect code!

- The compiler and/or hardware may re-order the reads/writes to `a` and `flag`, or `flag` may be held in a register.
  - NB - It might nevertheless work sometimes, depending on the OpenMP implementation, underlying hardware, scheduler decisions, … ☹
  - This is why concurrency bugs can go undetected for years…
- OpenMP provides the `#pragma omp` **`flush`** directive, which specifies an explicit flush operation
  - can be used to make the above example work
  - … but its use is difficult and prone to subtle bugs

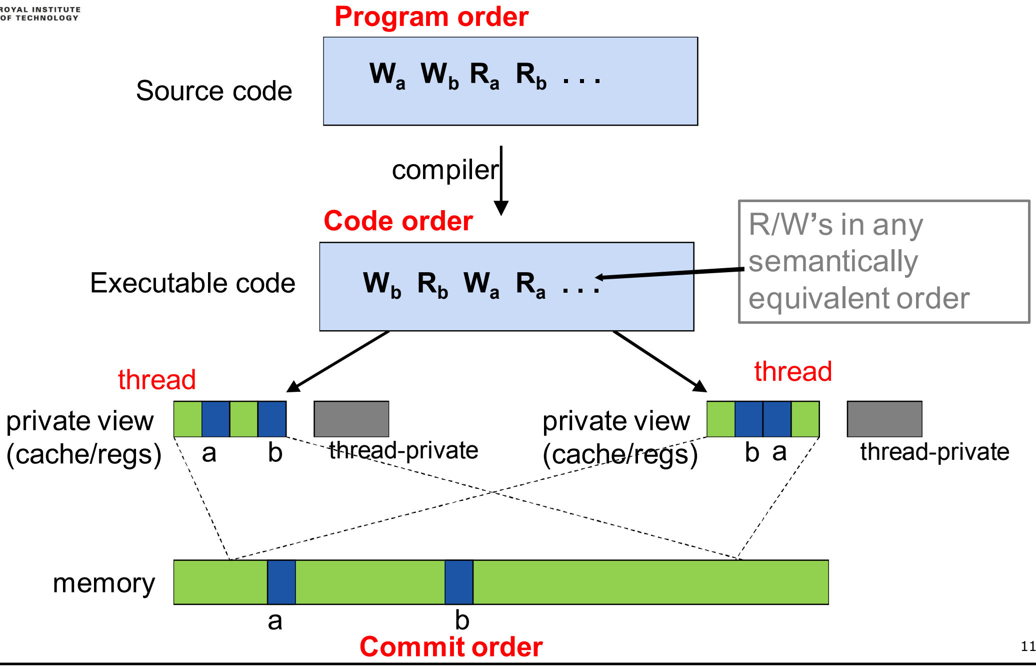
OpenMP memory model

- OpenMP supports a shared memory model.
- All threads share an address space, but it can get complicated:


Shared memory
a
cache1
cache2
cache3
cacheN
proc1
proc2
proc3
. . .
procN
a

- A memory model is defined in terms of:
  - **Coherence**: Behavior of the memory system when a single address is accessed by multiple threads.
  - **Consistency**: Orderings of reads, writes, or synchronizations (RWS) with various addresses and by multiple threads.

OpenMP Memory Model: Basic Terms



Program order

Source code W_a W_b R_a R_b . . .

compiler

Code order

Executable code W_b R_b W_a R_a . . .

R/W's in any semantically equivalent order

thread
private view
(cache/regs)
a
b
thread-private
thread
private view
(cache/regs)
b
a
thread-private
memory
a
b

Commit order

Consistency: Memory Access Re-ordering

- **Re-ordering**:
  - Compiler re-orders program order to the code order
  - Machine re-orders code order to the memory commit order
- At a given point in time, the "private view" seen by a thread may be different from the view in shared memory.
- Consistency Models define constraints on the orders of Reads (R), Writes (W) and Synchronizations (S)
  - … i.e. how do the values "seen" by a thread change as you change how ops (R,W,S) follow other ops.
  - Possibilities include:
    - R→R, W→W, R→W, R→S, S→S, W→S

Some Consistency Models

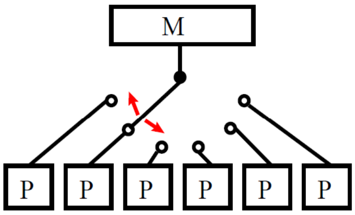
- **Sequential Consistency**:
  - In a multi-processor, ops (R, W, S) are sequentially consistent if:
    - They remain in program order for each processor.
    - They are seen to be in the same overall order by each of the other processors.
  - Program order = code order = commit order
  - The strongest consistency model available in practice, but not deterministic!

→

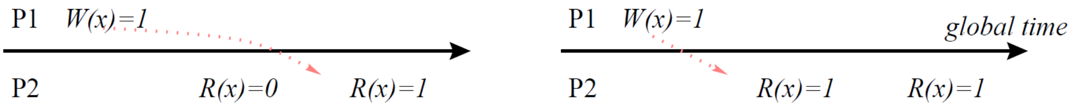
- **Relaxed consistency**:
  - Remove some of the ordering constraints for memory ops (R, W, S).

Sequential consistency [Lamport'79]

- + all memory accesses are ordered in *some* sequential order
- + all read and write accesses of a processor appear in program order
- + otherwise, arbitrary delays possible



Not deterministic:



OpenMP and Relaxed Consistency

OpenMP defines consistency as
a variant of weak consistency:

- S ops must be in sequential order across threads.
- Can not reorder S ops with R or W ops on the same thread
  - Weak consistency guarantees
    S→W, S→R , R→S, W→S, S→S

- The **S**ynchronization operation relevant to this discussion is **flush**.

Flush

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory with respect to the “flush set”.
- The **flush set** is:
  - “all thread visible variables” for a flush construct without an argument list.
  - a list of variables when the “flush(list)” construct is used.
- The action of **flush** is to guarantee that:
  - All R,W ops that overlap the flush set and occur prior to the flush complete before the flush executes
  - All R,W ops that overlap the flush set and occur after the flush don’t execute until after the flush.
  - Flushes with overlapping flush sets can not be reordered.

Memory ops: R = Read, W = write, S = synchronization

Synchronization: **flush** example

- Flush forces data to be updated in memory so other threads see the most recent value

```
double A;

A = compute();

flush(A);    // flush to memory to make sure other
             // threads can pick up the right value
```

Note: OpenMP’s flush is analogous to a **fence** in other shared memory API’s.

What is the Big Deal with Flush?

- Compilers (and processors with out-of-order execution) routinely reorder instructions implementing a program
  - This helps better exploit the functional units, keep machine busy, hide memory latencies, etc.
  - See only a single instruction stream, unaware of asynchronous side effects due to multithreading
- Compiler generally cannot move instructions:
  - past a **barrier**
  - past a **flush** on all variables
- But it can move them past a **flush** with a list of variables so long as those variables are not accessed
- Keeping track of consistency when **flush**es are used can be confusing … especially if "**flush**(list)" is used.

Note: the flush operation does not actually synchronize different threads. It just ensures that a thread's values are made consistent with main memory.

Exercise 10: **producer consumer**

- Parallelize the "prod_cons.c" program. →

  This is a well known coordination pattern called the **producer consumer** pattern
  - One thread produces data objects (e.g., values) that another thread consumes.
  - Often used with a stream of produced data to implement "pipeline parallelism"
- The key is to implement pairwise synchronization between threads.

Exercise 10: prod_cons.c

```
int main()
{
   double *A, sum, runtime;
   int flag = 0;

   A = (double *) malloc( N*sizeof(double) );

   runtime = omp_get_wtime();

   fill_rand(N, A);        // Producer: fill an array with data

   sum = Sum_array(N, A);  // Consumer: sum the array

   runtime = omp_get_wtime() - runtime;

   printf(" In %lf seconds, The sum is %lf \n", runtime, sum);
}
```

Pair-wise synchronization in OpenMP

OpenMP lacks synchronization constructs that work between pairs of threads.

- When this is needed, you have to build it yourself.

- **Pair-wise synchronization**
    - Use a shared *flag* variable
    - Reader spins waiting for the new *flag* value
    - Use **flush**es to force updates to and from memory

Exercise 10: producer consumer

```
int main()
{
   double *A, sum, runtime;
   int numthreads, flag = 0;
   A = (double *) malloc( N*sizeof(double) );

   #pragma omp parallel sections
   {
     #pragma omp section
      { // producer:
        fill_rand(N, A);
        #pragma omp flush
        flag = 1;
        #pragma omp flush(flag)
      }
      #pragma omp section
      { // consumer:
        #pragma omp flush(flag)
        while (flag != 1){
            #pragma omp flush(flag)
        }
        #pragma omp flush
        sum = Sum_array(N, A);
      }
    }
}
```

Use shared variable flag to signal when the "produced" value is ready

Flush forces refresh to memory. Guarantees that the other thread sees the new value of A

Flush needed on both "reader" and "writer" sides of the communication

Notice you must put the flush *inside* the while loop to make sure the updated flag variable is seen

---

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Recap of OpenMP basics
- Memory model
- ➡ Threadprivate Data
- OpenMP 4.x: Task Dependences, Accelerators, SIMD …
- Alternatives to OpenMP
- Summary

Data sharing: **Threadprivate**

- Makes **global data** private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables, static class members
- Different from making them **PRIVATE**
  - with **PRIVATE** global variables are masked.
  - **THREADPRIVATE** preserves global scope within each thread
- Threadprivate variables can be initialized using **COPYIN** or at time of definition (using language-defined initialization capabilities).

A **threadprivate** example (C)

Use threadprivate to create a counter for each thread.

```
int counter = 0;
#pragma omp threadprivate( counter )

int increment_counter()
{
    counter++;
    return (counter);
}
```

ROYAL INSTITUTE OF TECHNOLOGY

Data Copying: **Copyin**

You initialize threadprivate data using a **copyin** clause.

```
#define N 1000
int A[N];
#pragma omp  threadprivate(A)

/* Initialize the A array */
     init_data(N,A);

#pragma omp parallel copyin(A)
{
   … Now each thread sees threadprivate array A initialied
   … to the global value set in the subroutine init_data()
}
```

Allocate a new local copy of shared array A on each thread

The original shared array A is seen outside parallel regions

Each thread copies shared array A to its local copy

ROYAL INSTITUTE OF TECHNOLOGY

Data Copying: **Copyprivate**

Used with a **single** region to *broadcast* values of privates from one member of a team to the rest of the team.

```
#include <omp.h>
void input_parameters(int, int); // fetch values of input parameters
void do_work(int, int);

void main()
{
   int Nsize, choice;

   #pragma omp parallel private (Nsize, choice)
   {
        #pragma omp single copyprivate (Nsize, choice)
         {
             input_parameters( &Nsize, &choice );
          }
         do_work( Nsize, choice );
   }
}
```

Here the local values of Nsize and choice are copied into the other threads' copies

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Recap of OpenMP basics
- Memory model
- Threadprivate Data
- ➡ OpenMP 4.x: Task Dependences, Accelerators, SIMD ...
- Alternatives to OpenMP
- Summary

What is new in OpenMP 3.1-4.5

Lots…

- Task dependences
- Accelerator support (target construct, OpenMP 4.0)
- Taskloops (OpenMP 4.5)
- Read/write/update atomics
- Task priorities (OpenMP 4.5)
- SIMD support for loops (simd construct, OpenMP 4.0)
- Cancellation
- Vectorization support
- User-defined reducers

- Plus some odds and ends I’m not that familiar with…

OpenMP4.0 **Task dependences**

Task dependences

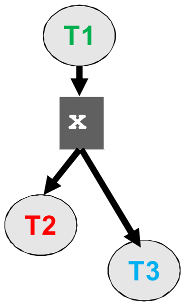
```
C/C++
#pragma omp task depend(dependency-type: list)
 ... structured block ...
```

- The **task dependence** is fulfilled when the predecessor task has completed
  - ***in*** dependency-type: the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **out** or **inout** clause.
  - ***out*** and ***inout*** dependency-type: The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **in**, **out**, or **inout** clause.
  - The list items in a **depend** clause may include array sections.

Concurrent execution with dependences

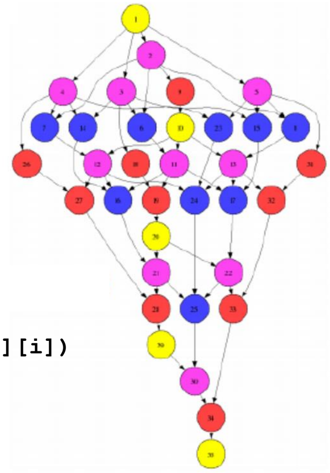
```
void process_in_parallel)
{
#pragma omp parallel
#pragma omp single
 {
   int x = 1;
   ...
   for (int i = 0; i < T; ++i) {
#pragma omp task shared(x, ...) depend(out: x) // T1
      preprocess_some_data(...);
#pragma omp task shared(x, ...) depend(in: x) // T2
      do_something_with_data(...);
#pragma omp task shared(x, ...) depend(in: x) // T3
      do_something_independent_with_data(...);
   }
 } // end omp single, omp parallel
}
```

- T1 must complete before T2 and T3 can be executed.
- T2 and T3 can be executed in parallel.



A more realistic example

```
void blocked_cholesky( int NB, float A[NB][NB] )
{
   int i, j, k;
   for (k=0; k<NB; k++) {
#pragma omp task depend(inout:A[k][k])
         spotrf (A[k][k]) ;
         for (i=k+1; i<NT; i++)
#pragma omp task depend(in:A[k][k]) depend(inout:A[k][i])
              strsm (A[k][k], A[k][i]);
            // update trailing submatrix
            for (i=k+1; i<NT; i++) {
              for (j=k+1; j<i; j++)
#pragma omp task depend(in:A[k][i],A[k][j]) depend(inout:A[j][i])
                   sgemm( A[k][i], A[k][j], A[j][i]);
#pragma omp task depend(in:A[k][i]) depend(inout:A[i][i])
              ssyrk (A[k][i], A[i][i]);
      }
   }
}
```

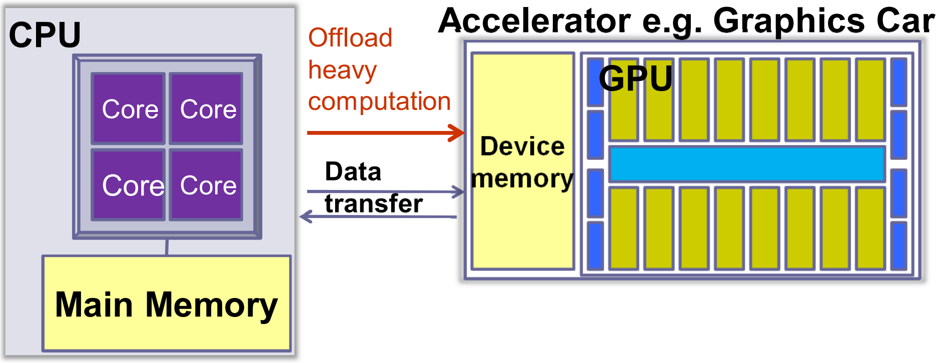


ROYAL INSTITUTE OF TECHNOLOGY

OpenMP 4.5: Tasks with **target** directive

Heterogeneous Systems

e.g. CPU-GPU based systems (→ more in lecture on CUDA programming)



CPU

Core

Core

Core

Core

Main Memory

Offload heavy computation

Data transfer

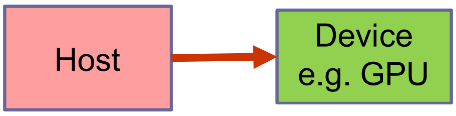
Accelerator e.g. Graphics Card

Device memory

GPU

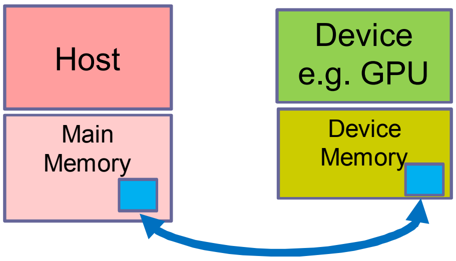
- Often, distributed memory
  - Accelerator (e.g., traditional GPU in graphics card) cannot access main memory
- Standard programming models for GPUs (CUDA, OpenCL) require explicit data transfer to/from device memory for operands
- High-level progr. frameworks (e.g. SkePU, OpenACC, OpenMP4.x) can generate data transfers automatically if the access mode (read, write, readandwrite) for each operand is given – by coherent software caching

OpenMP 4 Target Directive



- Transfer control from the host to a programmable accelerator device (e.g. GPU, Xeon-Phi, …)
  → *"offloading"* a computation to device
  - Default: *Synchronous* offloading,
    host waits for completion by device
- Syntax (C/C++)
  `#pragma omp target [clause[[,] clause],…]`
  structured-block
- Clauses
  - **device**(*scalar-integer-expression*)
  - **map**(alloc | to | from | tofrom: *list*) →
  - **if**(*scalar-expr*)
- OpenMP compiler attempts best-effort translation of *structured-block* to the target programming model (e.g., CUDA or OpenCL for GPU)
- If target is not present or not supported, *structured-block* is executed by the host (CPU)

Map Clause



- Map a variable or an array section to a device data environment
- Syntax:
  `map ( alloc | to | from | tofrom`: *list* **)**
- Map-types
  - **`alloc`**: allocate storage for corresponding variable
  - **`to`**: alloc and assign value of original variable to corresponding variable on entry
  - **`from`**: alloc and assign value of corresponding variable to original variable on exit
  - **`tofrom`**: default, both to and form

Example

```
void vec_mult( float *C, float *A,
               float *B, int N )
{
 int i;

 #pragma omp target map( to: A[0:N], B[0:N] ) \
                    map( from: C[0:N] )
 #pragma omp parallel for
 for (i=0; i<N; i++)
    C[i] = A[i] + B[i];

}
```

Variable N is implicitly mapped from the surrounding scope

Compiler will *best-effort* translate this OpenMP code into an equivalent kernel for the device

Note: **Map is not necessarily a Copy**.
On systems where host and device share memory, both see and modify the *same* memory locations.

Adapted from: E. Stotzer, OpenMP 4 Tutorial, IWOMP'14

Beyond OpenMP Map

- Other hardware and software abstractions have been proposed to automatically manage and optimize data transfers to/from device:
  - **CUDA Unified Memory** (for certain Nvidia GPUs)
    - Hardware distributed-shared memory (page-wise)
  - **"Smart Containers"**
    - Data abstractions (e.g. C++ STL-like Vector<..>) for operand data, transparently performing software caching on device(s)
    - *Lazy copying* – delay transfer and suppress redundant transfers. Considerable speedup for iterative computations using GPU.
    - E.g.: U. Dastgeer, C. Kessler: Smart containers and skeleton programming for GPU-based systems. *International Journal of Parallel Programming* 44(3):506-530, June 2016. DOI: 10.1007/s10766-015-0357-6.

Asynchronous Offloading

By packing the **target** directive in an omp **task** construct, the host can work concurrently with the device code

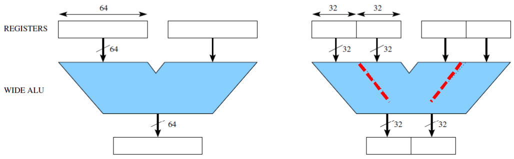
```
void vec_mult( float *C, float *A, float *B, int N )
{
 int i;
 #pragma omp task
 {
 #pragma omp target map( to: A[0:N], B[0:N] ) \
                    map( from: C[0:N] )
    #pragma omp parallel for
    for (i=0; i<N; i++)
       C[i] = A[i] + B[i];
 }
 #pragma omp task
 {
    //… some independent host code here
 }
 #pragma omp taskwait
}
```

OpenMP 4: **SIMD Loop vectorization and Multithreading+SIMD**

SIMD Instructions

SIMD: Single instruction stream, multiple data streams

SIMD instructions in modern microprocessors


REGISTERS
WIDE ALU
64
64
64
32
32
32
32
32
32

e.g. Intel MMX / SSE / SSE2 / SSE3
("Streaming SIMD Extensions")
for x86, x86_64

e.g. IBM/Motorola AltiVec for PowerPC

e.g. AVX, Xeon-Phi, ARM Neon, Cell SPE, …

also for DSP processors ...

Constraints:

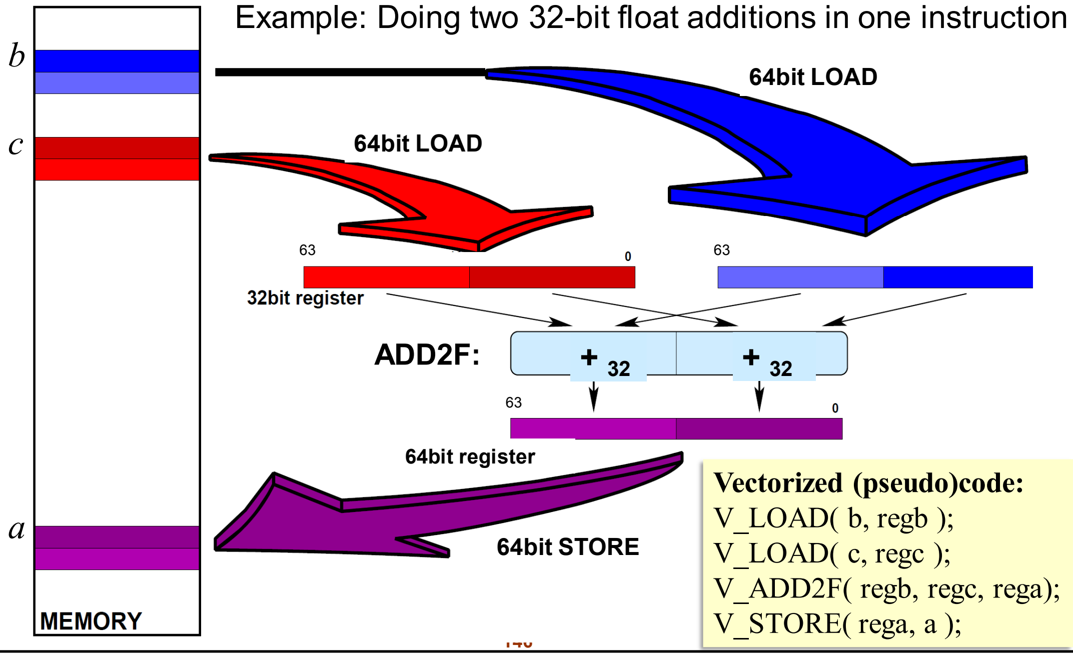
- Same instruction on smaller data type
- Contiguous storage of operands and results in memory and registers
- Alignment of addresses

Trend: CPU SIMD width is increasing: e.g. SSE (128bit), AVX (256bit), Xeon-Phi (512bit)

Using SIMD Instructions

Original code:
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];

Example: Doing two 32-bit float additions in one instruction


b
c
a
MEMORY
64bit LOAD
64bit LOAD
63
0
63
32bit register
ADD2F:
+ 32
+ 32
63
0
64bit register
64bit STORE

Vectorized (pseudo)code:
```
V_LOAD( b, regb );
V_LOAD( c, regc );
V_ADD2F( regb, regc, rega);
V_STORE( rega, a );
```

Using SIMD Instructions

- **Manual** vectorization …
  using SIMD intrinsics (low-level; architecture specific)
  or Fortran Array syntax (no such portable equivalent for C)
- Modern compilers can sometimes exploit SIMD instructions ***automatically*** – for straightline code and for (simple) loops.
  - Requires well-analyzable code and (basically) absence of *loop-carried* (cross-iteration) data dependences
    (details and exceptions omitted here for brevity)
- Certain program transformations e.g. loop unrolling may help:

```
void vector_add ( float  a[], b[], c[], unsigned int N )
{
    unsigned int i;
    for ( i = 0; i < N; i+ = 2 ) {
        a[i] = b[i] + c[i];
        a[i + 1] = b[i + 1] + c[i + 1];
    }
}
```

Vectorized (pseudo)code:

```
V_LOAD( b, regb );
V_LOAD( c, regc );
V_ADD2F( regb, regc, rega);
V_STORE( rega, a );
```

Using SIMD Instructions

- Automatic vectorization of loops can fail for many reasons
- Example: The compiler cannot statically decide
  if the loop's upper bound is loop-invariant or not,
  nor if there may be a loop-carried data dependence or not:

```
typedef struct {   // user-defined vector data type
      float *elem;
      unsigned int length;
} fvec;

void v_add ( fvec a, b, c )
{
   int i;
   for (i=0;  i < v->length;  i++ )
         a->elem[i] = b->elem[i] + c->elem[i];
}
```

Compiler cannot prove statically that the memory locations that *may* be denoted by v->length, a->elem[i] etc. in some execution will *never* overlap (possibility of pointer arithmetics in C

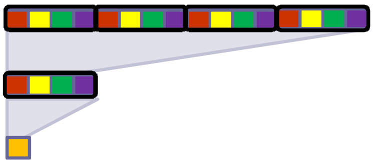
Loop Vectorization in OpenMP 4

- #pragma omp **simd** [clause [, clause]*]
  … *for-loop* (*nest*) …
- The OpenMP compiler cuts the loop into **chunks** each consisting of as many iterations as fit in one SIMD word of the target CPU,
- and generates SIMD instructions for each chunk of the loop.

```
typedef struct {   // user-defined vector data type
      float *elem;
      unsigned int length;
} fvec;

void v_add ( fvec a, b, c )
{
   int i;
#pragma omp simd
   for (i=0;  i < v->length;  i++ )
          a->elem[i] = b->elem[i] + c->elem[i];
}
```

SIMD loops with reduction

- Vectorization for reduction loops
  - 1. Summing element-wise up over chunks, using SIMD-add
  - 2. The final chunk is summed up sequentially (non-SIMD)



```
typedef struct {   // user-defined vector data type
      float *elem;
      unsigned int length;
} fvec;
void v_sum ( fvec a )
{
   int i;    float sum = 0.0;
#pragma omp simd reduction(+ : sum)
   for (i=0;  i < v->length;  i++ )
          sum += a->elem[i];
}
```

SIMD loop clauses

- safelen, simdlen, linear, aligned, collapse(*n*), private, lastprivate, …
- Details omitted here for brevity, see OpenMP 4.x documentation for reference

---

SIMD Worksharing Construct

- #pragma omp **for simd** [clause [, clause]*]
  … for-loop (nest) …
- Parallelize and vectorize the for-loop(s)
  - 1. multithread the loop in chunks, as in omp **for**
  - 2. vectorize each chunk, as in omp **simd**

```
…
void par_v_sum ( fvec a )
{
  int i;   float sum = 0.0;
#pragma omp for simd reduction(+ : sum)
  for (i=0;  i < v->length;  i++ )
       sum += a->elem[i];
}
```

Remark:
For better performance, the chunk size should be a multiple of the SIMD operation (i.e., register) length. (why?)

One can add a **simd** modifier to a **schedule** clause in order to enforce this by automatically rounding up the chunk size.
(Details omitted for brevity)

SIMD Remarks

- Also available in OpenMP 4:
  Constructs to help with vectorization of loops that involve user-defined functions (instead of only standard operators)
- Initialization of SIMD vectors from scalars
- For certain SIMD operations there is no good high-level support, e.g. instructions for permuting the elements within a SIMD vector
  - Still need to use intrinsic functions for that
- OpenMP worksharing constructs and some less common control flow constructs of C (e.g. setjmp/longjmp) are forbidden in SIMD regions
- Warning:
  Loop vectorization does not always improve performance.

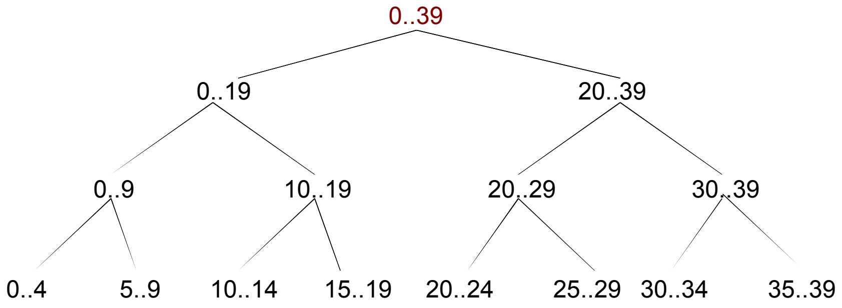
Details omitted here for brevity.

OpenMP4.5 **Taskloop construct**

Loops and tasks

- Loop scheduling used to be special (still is in OpenMP)
  - Often **static**:
    - iteration 0..9 on processor 0,
    - iteration 10..19 on processor 1,
    - ...
- Can be unified with task scheduling
  - Recursively divide iterations,
    making loop look like recursive divide and conquer
  - Run loop sequentially for small number of iterations

The task tree of a loop



0..39

0..19 20..39

0..9 10..19 20..29 30..39

0..4 5..9 10..14 15..19 20..24 25..29 30..34 35..39

Task loops (OpenMP 4.5)

```
#pragma omp taskloop [clause[[,] clause] ...] new-line
   for-loops
```

where *clause* is one of the following:

```
shared( list )
private( list )
default( shared | none )
firstprivate( list )
lastprivate( list )
grainsize( grain-size )
num_tasks( num-tasks )
collapse( n )
if( [taskloop:] scalar-expr )
final( scalar-expr )
priority( priority-value )
nogroup
untied
mergeable
```

When a thread encounters a **taskloop** construct, the construct partitions the associated loops into tasks for parallel execution of the loops' iterations.

OpenMP 3.1: **Read/Write Atomics**

Read/write/update atomics

Atomic was expanded in 3.1 to cover the full range of common scenarios where you need to protect a memory operation so it occurs atomically:

```
#pragma omp atomic [read | write | update | capture]
```

- **Atomic** can **protect loads**

```
#pragma omp atomic read
   v = x;
```

- **Atomic** can **protect stores**

```
#pragma omp atomic write
   x = expr;
```

- **Atomic** can **protect updates** to a storage location
  (this is the default behavior … i.e. when you don't provide a clause)

```
#pragma omp atomic update
   x++; or ++x; or x--; or –x;
   or x binop= expr; or x = x binop expr;
```

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Recap of OpenMP basics
- Memory model
- Threadprivate Data
- OpenMP 4.0: Task Dependences, Accelerators, SIMD …
- ➡ Alternatives to OpenMP
- Summary

Future development of OpenMP

Topics that are being addressed
at varying degree of maturity:

- Error handling
- Affinity
- Performance tool support

Some Alternatives to OpenMP

- **C11/C++11**
  - Threading built into the language
  - Async and lambda functions in C++ provide a task-like functionality
  - High-level parallel extensions of C++ e.g. SkePU, FastFlow, GrPPI
  - STL for aggregate data abstractions e.g. vector<...>
- **Intel Cilk Plus** (C/C++)
  - Task-centric approach
  - Array syntax for vector acceleration
- **Intel Threading Building Blocks** (TBB) for C++
  - Similar to OpenMP/Cilk tasks, plus some high-level constructs (dataparallel loops, reductions, pipeline)
- **Java fork-join framework**
  - Similar to OpenMP/Cilk tasks
- **MS TPL** (C#, .NET)
  - Similar to OpenMP/Cilk tasks
- **PGAS languages** e.g. X10, Chapel, UPC, Co-Array Fortran
  - If data locality matters (e.g. on clusters)

Summary

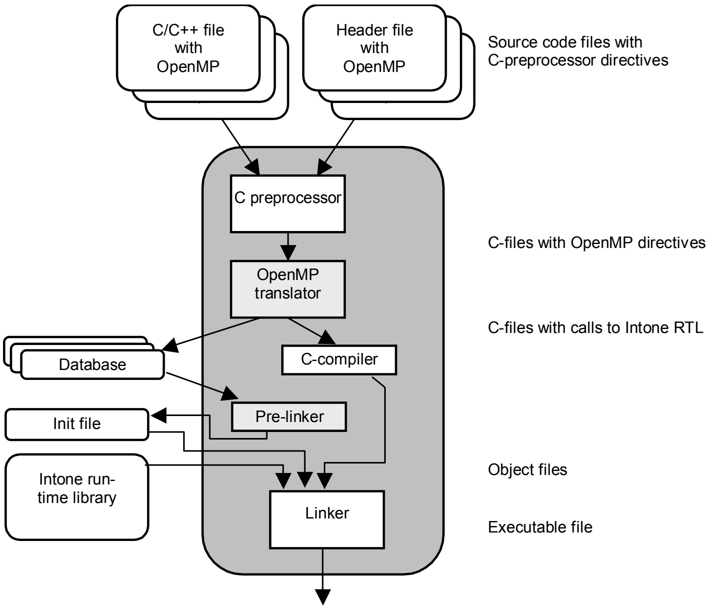
- OpenMP is the currently most widely spread shared memory programming model
  - With a higher abstraction level than explicit threading
- Widespread industrial support
  - Intel is particularly pushing OpenMP but even more Cilk Plus (and other models)
- Easy to get started
- Difficult to master
- Incremental parallelism
- Geared towards ”good enough” performance

Summary (2)

- We have covered much of OpenMP but omitted some minor details
- Download the spec to learn more …
  - www.openmp.org
  - the spec is filled with examples to support your continuing education.

APPENDIX

An Example OpenMP 1.0 Implementation


C/C++ file with OpenMP
Header file with OpenMP
Source code files with C-preprocessor directives
C preprocessor
C-files with OpenMP directives
OpenMP translator
C-files with calls to Intone RTL
C-compiler
Database
Pre-linker
Init file
Intone run-time library
Object files
Linker
Executable file

The OpenMP Translator

The OpenMP translator deals with three things:

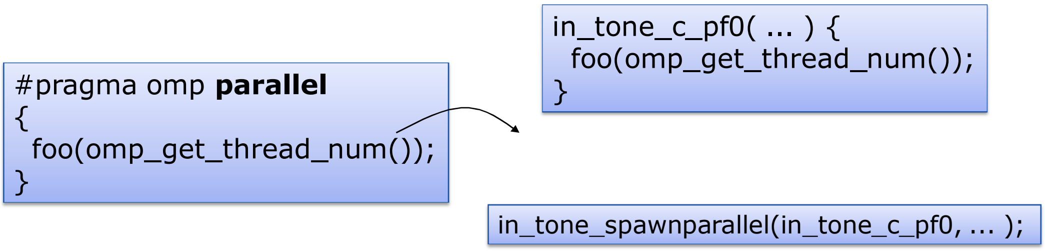
- Transformation of OpenMP constructs
  - Parses constructs
  - Performs some semantic and syntactic checks
  - Instruments the code with calls to the run-time library
- Handling of data clauses
  - Parses data clauses
  - Performs checks
  - Possibly alter variable declarations
- Instrumentation of OpenMP constructs
  - Interface to performance monitoring tools

The Run-Time Library

The run-time library deals with:

- Thread creation
- Thread synchronization
  - Locks
  - Barriers
- Work-sharing
  - Distribution of loop iterations among threads
- Memory consistency
  - Flush operations

The *parallel* OpenMP construct



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

```
in_tone_c_pf0( ... ) {
  foo(omp_get_thread_num());
}
```

```
in_tone_spawnparallel(in_tone_c_pf0, ... );
```

- The **parallel** construct forces threads to be created
  - The **parallel** region is executed in parallel
  - One level of parallelism is supported

What about shared variables?

```
int s, p1;
#pragma omp parallel private(p1)
{
   float p2;
   foo(omp_get_thread_num(), &p1);
}
```

- Variables with **global** scope
  are normally shared by all threads
- **Private** variables with a **global** scope
  are allocated on each thread’s stack during the parallel region and references are modified by the compiler
- **Stack allocated** variables that are **shared**
  are accessed through pointer references
- **Stack allocated** variables that are **private**
  are accessed through the stack pointer

Work-sharing constructs – the **for-loop**

```
#pragma omp for schedule(dynamic,2) lastprivate(lp1)
for (i = 0; i < MAX; i = i + 3)
{
   /* Body of the parallel loop */
}
```

- The run-time library’s work-sharing primitives directly support for-loops

- The for-loop is translated into:
  - A call to the run-time system that initializes the for loop
  - A while-loop that requests iterations until there are none left and does the work
  - A call to the run-time system ending the for-loop

The **single** construct

```
#pragma omp single nowait
{
   foo( );
}
#pragma omp single
{
   bar( );
}
```

- The **single** construct is treated as a for-loop with a single (1) iteration
- The **nowait** clause causes the compiler to *not* emit the code for the otherwise implicit barrier

The **section** construct

```
#pragma omp sections
{
  #pragma omp section
  {
    A( );
  }
  #pragma omp section
  {
    B( );
  }
}
```

- Each **section** is treated as an iteration and the **sections** construct is transformed to a for-loop

---

The **critical** construct

```
#pragma omp critical
{
    rdx = rdx + a;
    rdx2 = rdx2 *2;
}
```

- The critical section is enclosed with lock primitives

```
in__tone_set_lock(&in__tone_critical_lock_);
in__tone_global_flush( );
{
    rdx = rdx + a;
    rdx2 = rdx2 * 2;
}
in__tone_global_flush( );
in__tone_unset_lock(&in__tone_critical_lock_);
```

The **atomic** construct

```
…
#pragma omp atomic
  rdx = rdx + foo();
…
```

- The atomic update is replaced with a call to the run-time which does the actual update atomically:

```
…
in__tone_atomic_update(&rdx,
                              in_tone_type_f_float,
                              in_tone_op_plus, foo());
…
```

- Support for the final reduction of reduction variables is also implemented in a similar way

Acknowledgment

- Slides mostly based on Mats Brorsson's slides from 2014
  - With some minor updates
- Many slides were developed by Tim Mattson and others at Intel under the creative commons license
- Thanks!