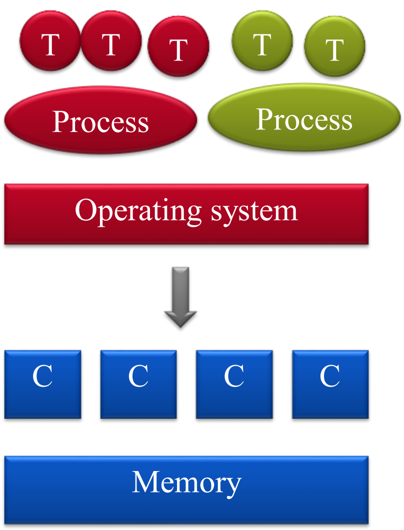
Shared memory programming with OpenMP

Mats Brorsson
Professor

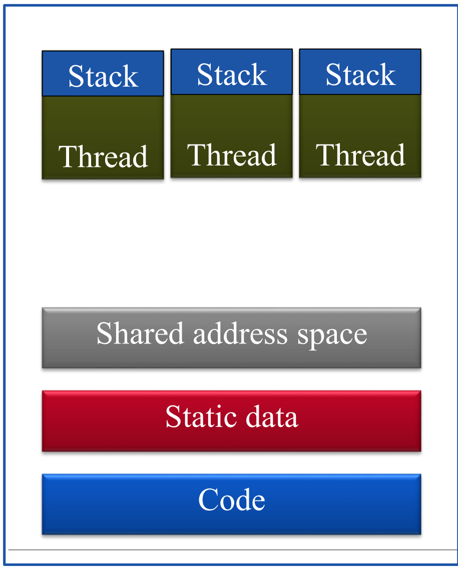
Shared memory Parallel Programming



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

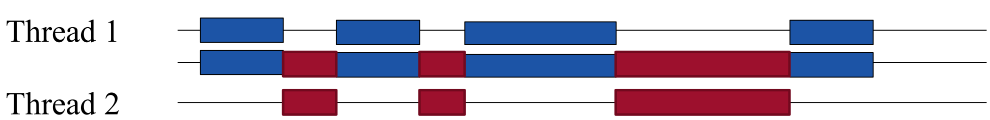


- The process is a container with capabilities and access rights to shared resources
  - Address space
  - Files
  - Code
  - Data
- Any program starts its execution as a single thread
- New threads can be created through OS calls

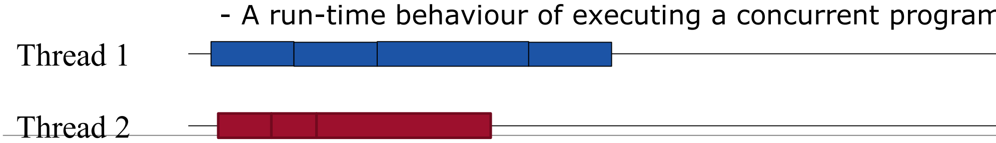
Concurrency vs Parallelism

As defined by Sun/Oracle:

- **Concurrency**: A condition that exists when at least two threads 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

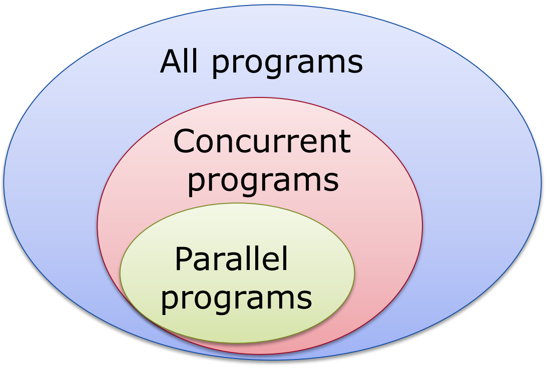


- **Parallelism**: A condition that arises when at least two threads are executing simultaneously.
  - A run-time behaviour of executing a concurrent program



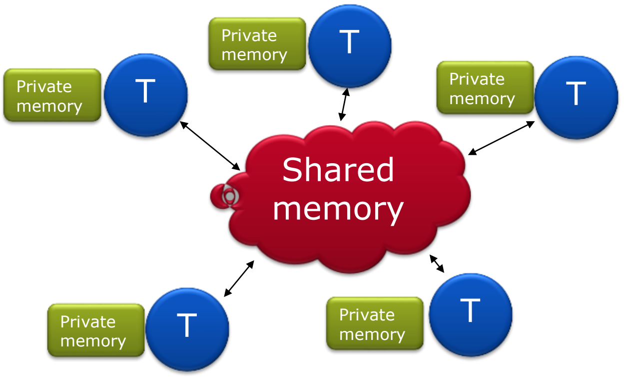
In other words…

- **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.



OpenMP

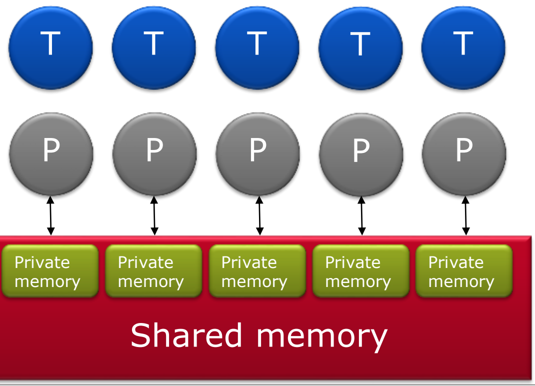
- A standardized (portable) way for writing concurrent programs for shared memory multiprocessors
  - For C/C++/Fortran



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

OpenMP

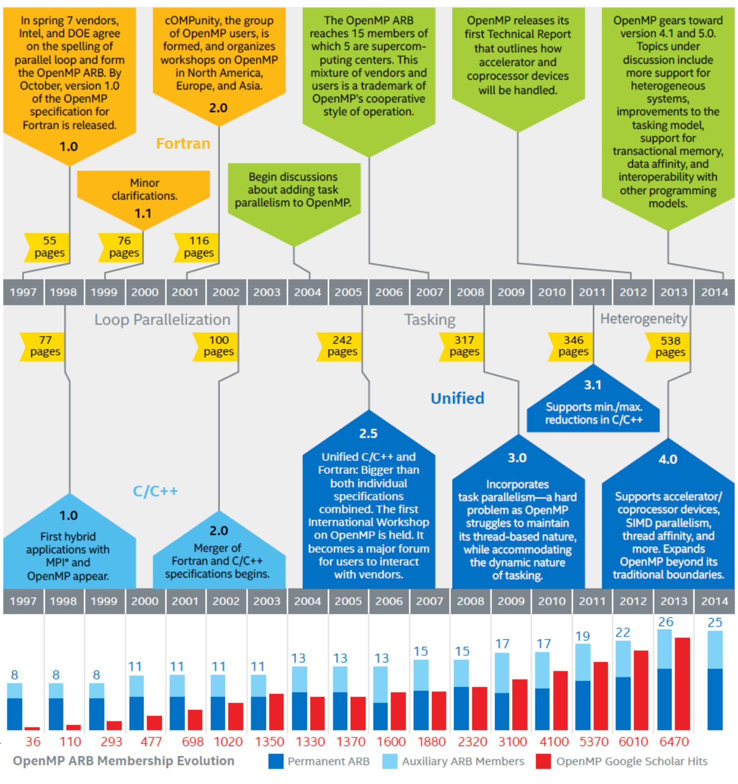
- A standardized (portable) way for writing concurrent programs for shared memory multiprocessors
  - For C/C++/Fortran



A more concrete model:

- Threads are scheduled on processors by the OS
- The private memory is located in the shared address space
- There are local memory to each processor
  - Caches
  - NUMA

The evolution of OpenMP



ROYAL INSTITUTE OF TECHNOLOGY

Agenda

Wednesday 20 Aug

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

Thursday 21 Aug

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

ROYAL INSTITUTE OF TECHNOLOGY

Acknowledgment

- Many slides are developed by Tim Mattson and others at Intel under the creative commons license
- Thanks!

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 4.0 and Accelerators

OpenMP* Overview:

C$OMP FLUSH

#pragma omp critical

C$OMP THREADPRIVATE(/ABC/)

CALL OMP_SET_NUM_THREADS(10)

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

!$OMP BARRIER

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

omp_set_lock(lck)

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

13

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

Proc1 | **Proc$_2$** | **Proc$_3$** | • • • | **ProcN**

Shared Address Space

14

OpenMP core syntax

ROYAL INSTITUTE OF TECHNOLOGY

- 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.

15

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

ROYAL INSTITUTE OF TECHNOLOGY

- Write a program that prints "hello world".

```
int main()
{



    int ID = 0;

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

}
```

16

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

17

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

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

```
#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)

18

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: when the program's outcome changes as the threads are scheduled differently.
- 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.

19

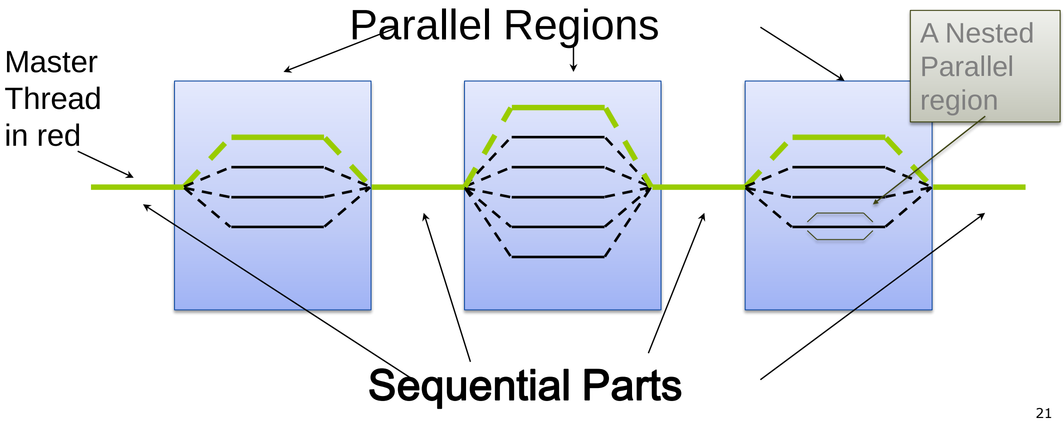
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.0 and Accelerators

20

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.



21

Thread Creation: Parallel Regions

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

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

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

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

22

Thread Creation: Parallel Regions

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

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

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

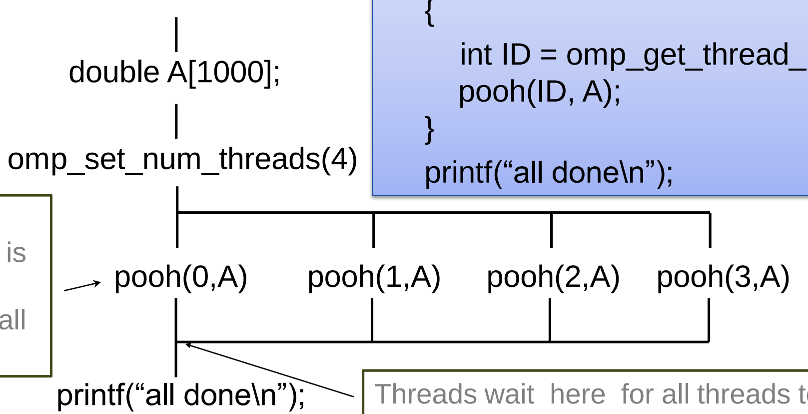
* The name "OpenMP" is the property of the OpenMP Architecture Review Board

23

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*)

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

24

What an OpenMP compiler does

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

- 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
- All known OpenMP implementations use a thread pool so full cost of threads creation and destruction is not incurred for reach parallel region.
- Only three threads are created because the last parallel section will be invoked from the parent thread.

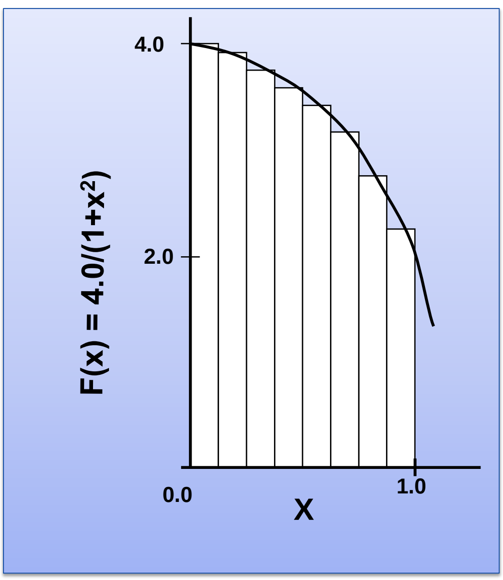
```
void thunk ()
{
   foobar ();
}

pthread_t tid[4];

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

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

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.

26

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 a parallel construct.
- Pay close attention to shared versus private variables.
- 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 Seconds since a fixed point in the past

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.0 and Accelerators

29

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

30

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);
      }
}
```

31

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

32

ROYAL INSTITUTE OF TECHNOLOGY

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.

33

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 4.0 and Accelerators

34

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

35

ROYAL INSTITUTE OF TECHNOLOGY

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

36

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

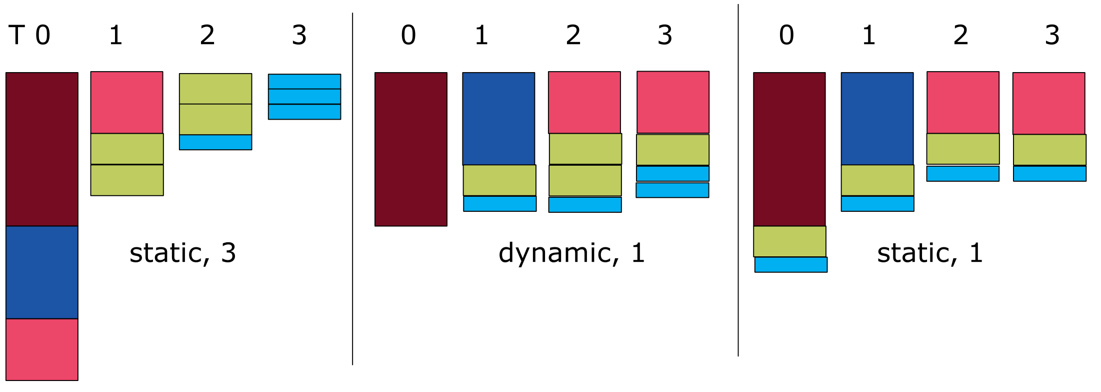
```
#pragma omp parallel
#pragma omp for
        for(i=0;i<N;i++)   { a[i] = a[i] + b[i];}
```

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.
  - 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)



loop work-sharing 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
AUTO	When the run-time can "learn" from previous executions of the same loop

Least work at runtime : scheduling done at compile-time

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

40

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

41

Working with loops

- Basic approach
  - Find compute intensive loops (use a profiler)
  - Make the loop iterations independent .. So they can safely execute in any order without loop-carried dependencies
  - 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

Remove loop carried dependence

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

42

ROYAL INSTITUTE OF TECHNOLOGY

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 loop of length NxM and then parallelize that.
- Useful if N is O(no. of threads) so parallelizing the outer loop makes balancing the load difficult

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]
}
```

44

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.

45

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;
```

46

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
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.

47

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.

48

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 4.0 and Accelerators

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

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();
}
```

53

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();
}
```

54

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.

55

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 omp ordered
              res += consum(tmp);
       }
```

56

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.

57

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.

58

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.

59

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.

60

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 4.0 and Accelerators

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.

63

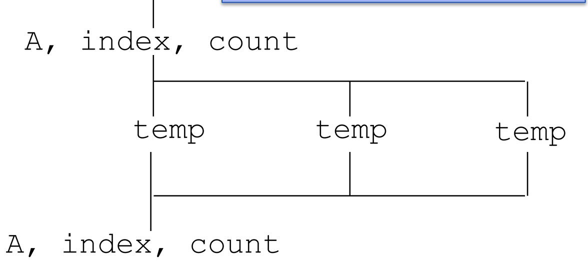
Data sharing: Examples

```
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



64

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.

65

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 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: 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 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.

70

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. <u>Good programming practice</u>!

Only the Fortran API supports default(private).

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

71

ROYAL INSTITUTE OF TECHNOLOGY

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.

73

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?

74

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.0 and Accelerators

75

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 berepresented by a closed-form expression at compiler time.
- 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]);
```

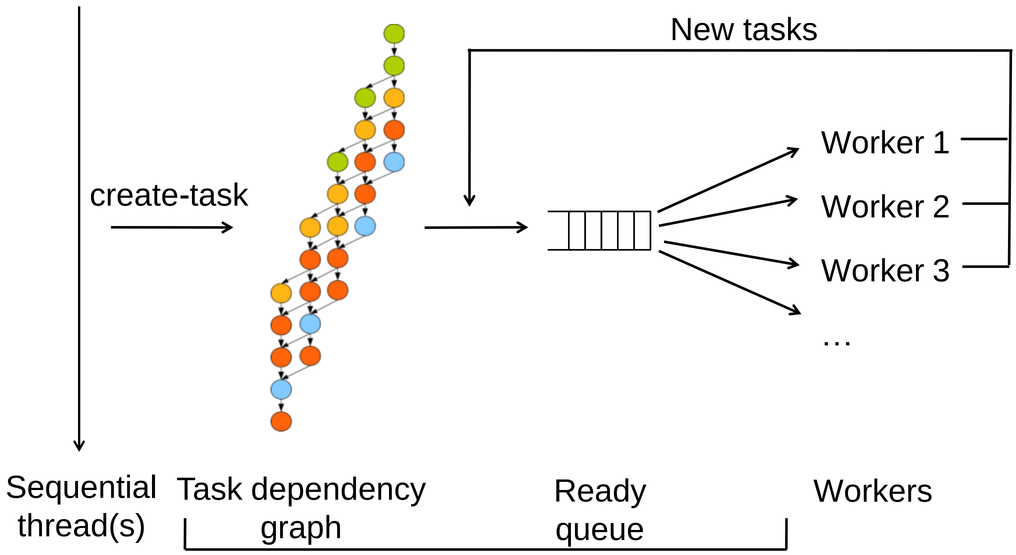
- Find out the length of list
- Copy pointer to each node in an array
- Process nodes in parallel with a for loop

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

78

Task parallelism



An example of task-parallelism

- The (naïve) sequential Fibon

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

        a = fib(n-1);

        b = fib(n-2);

        return a+b;
  }
}
```

<u>Parallelism in fib:</u>

- 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 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

```
int fib(int n){
 if( n<2 ) return n;
 else {
       int a,b;
#pragma omp task shared(a) if (n>30)
       a = fib(n-1);
#pragma omp task shared(b) if (n>30)
       b = fib(n-2);
#pragma omp taskwait
       return a+b;
 }
}
```

Starting code:

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

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!

83

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  )
```

84

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

85

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" .

86

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

87

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 ) ;
       }
   }
}
```

88

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

89

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

90

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

93

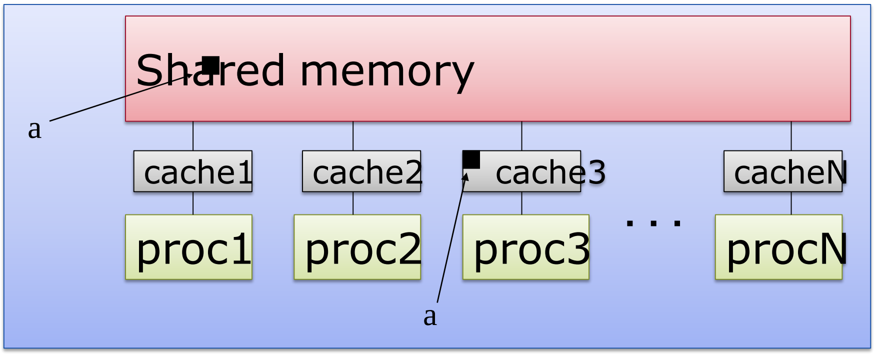
Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.0 and Accelerators

94

OpenMP memory model

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



- 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

97

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)

98

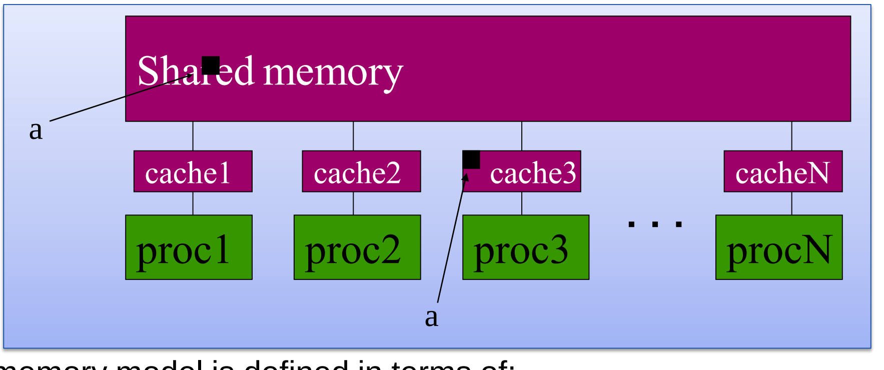
Example: producer-consumer pattern

Thread 0	Thread 1
`a = foo();`	`while (!flag);`
`flag = 1;`	`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.
- OpenMP has a `#pragma omp flush` directive which specifies an explicit flush operation
  - can be used to make the above example work
  - … but it's use is difficult and prone to subtle bugs

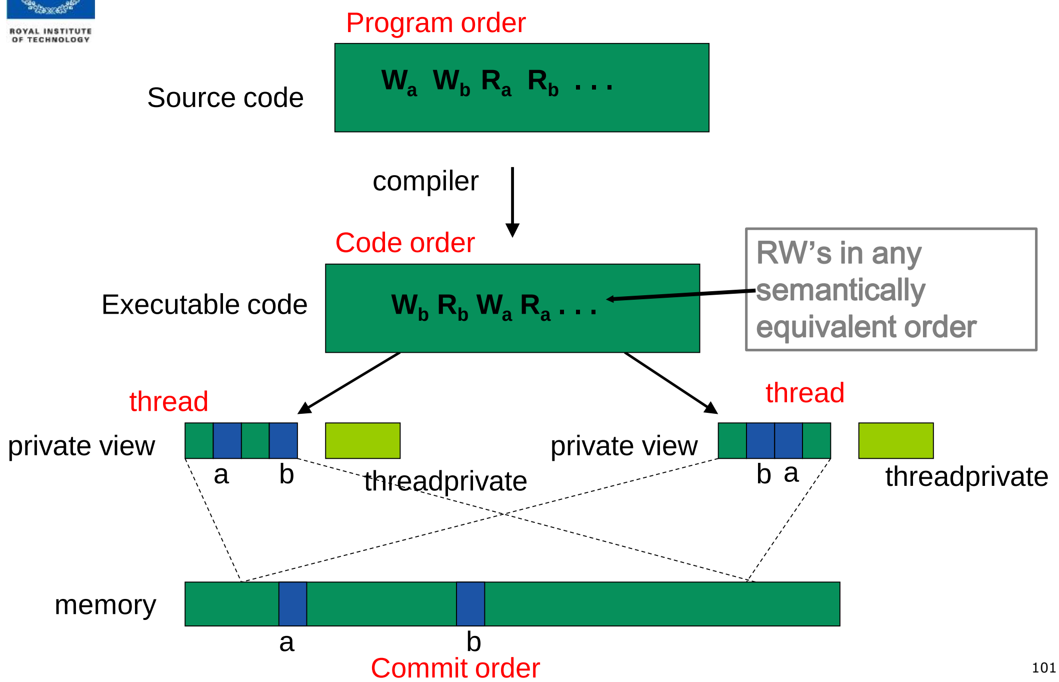
OpenMP memory model

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



- 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



101

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 follow ( ) other ops.
  - Possibilities include:
    - R→R, W→W, R→W, R→S, S→S, W→S

102

ROYAL INSTITUTE OF TECHNOLOGY

Consistency

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

ROYAL INSTITUTE OF TECHNOLOGY

OpenMP and Relaxed Consistency

- OpenMP defines consistency as a variant of <u>weak consistency</u>:
  - 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
      $S \rightarrow W$, $S \rightarrow R$, $R \rightarrow S$, $W \rightarrow S$, $S \rightarrow S$
- The Synchronization 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

105

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.

106

What is the Big Deal with Flush?

- Compilers routinely reorder instructions implementing a program
  - This helps better exploit the functional units, keep machine busy, hide memory latencies, etc.
- 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 flushes 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.

107

Exercise 10: producer consumer

- Parallelize the “prod_cons.c” program.
- This is a well known pattern called the producer consumer pattern
  - One thread produces values that another thread consumes.
  - Often used with a stream of produced values to implement “pipeline parallelism”
- The key is to implement pairwise synchronization between threads.

108

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 of 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);
}
```

109

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 flushes to force updates to and from memory

110

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
      {
         fill_rand(N, A);
         #pragma omp flush
         flag = 1;
         #pragma omp flush (flag)
      }
      #pragma omp section
      {
         #pragma omp flush (flag)
         while (flag != 1){
              #pragma omp flush (flag)
         }
         #pragma omp flush
         sum = Sum_array(N, A);
      }
   }
}
```

Use 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

111

Outline

- Introduction to OpenMP
- Creating Threads
- Synchronization
- Parallel Loops
- Synchronize single masters and stuff
- Data environment
- OpenMP Tasks
- Memory model
- Threadprivate Data
- OpenMP 4.0 and Accelerators

112

ROYAL INSTITUTE OF TECHNOLOGY

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).

113

ROYAL INSTITUTE OF TECHNOLOGY

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);
}
```

114

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()
}
```

115

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);
   }
}
```

116

Conclusion

- We have now covered the full sweep of the OpenMP specification (up to OpenMP 3.0)
  - 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 Compunity.

117