OpenMP*: An Introduction

Md Abdullah Shahneous Bari
Stony Brook University

Courtesy:

Barbara Chapman
Stony Brook University

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

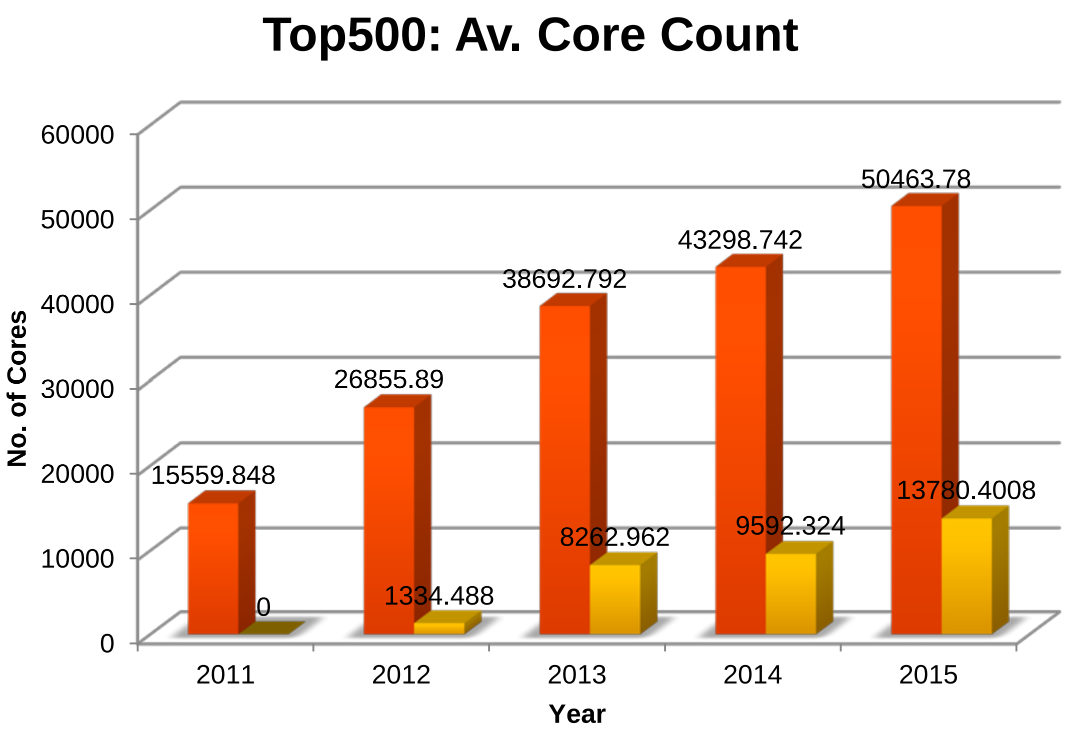
Agenda

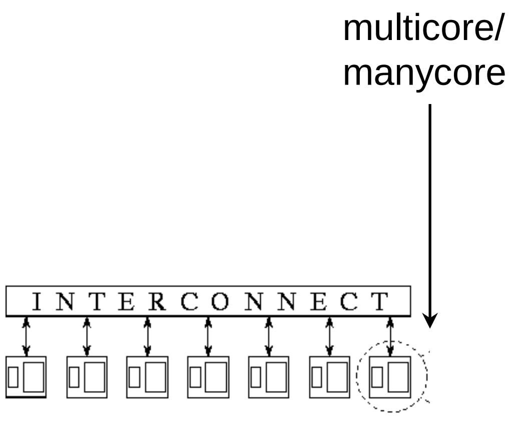
- Background
- Parallel Regions
- Sharing Work
- Handling Data
- Synchronization
- Tasks

Agenda

- **Background**
- Parallel Regions
- Sharing Work
- Handling Data
- Synchronization
- Tasks

Large-Scale Computers Today



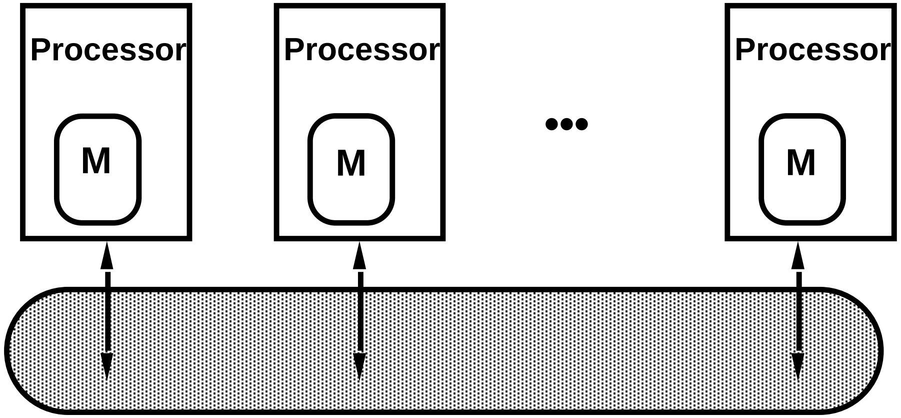


- HPC clusters today have nodes that are increasingly powerful parallel systems in their own right
  - The compute capacity of the nodes continues to grow
- It is important that application codes exploit the nodes as fully as possible

Parallelism In HPC Clusters

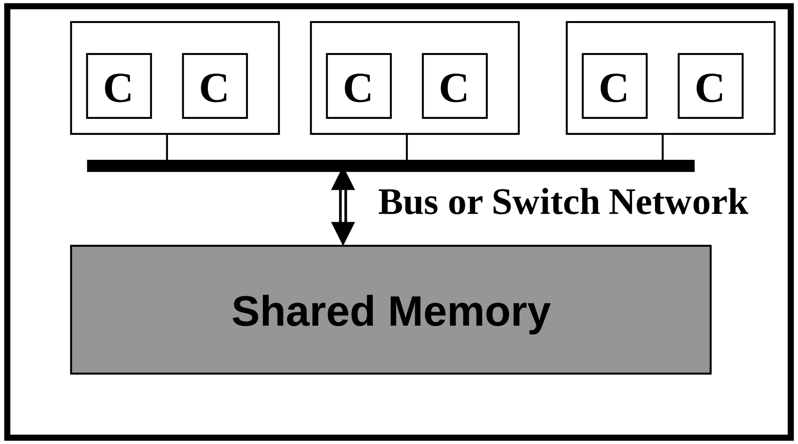
- Internode parallelism requires data to be exchanged across a fast network
- Intra-node parallelism exploits multiple cores and their shared memory
  - Today's nodes often also configured with accelerators, without shared memory (but potentially with unified memory)

Cluster Architecture



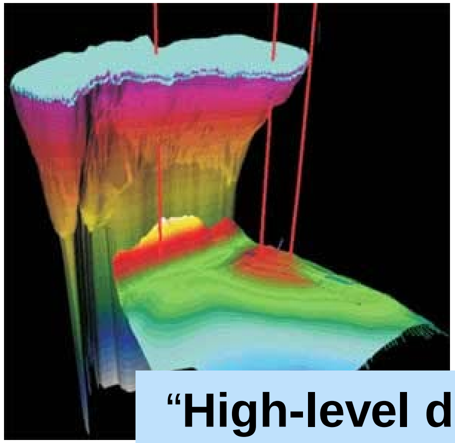
Network for Data Exchange

Shared Memory Architecture



The OpenMP API

- Industry standard providing directives (pragmas) to create parallel Fortran, C and C++ programs
  - Directives are instructions to a compiler
  - API also has library routines and environment variables
- Specification by OpenMP Architecture Review Board (ARB)
  - Members from industry, government labs, academia
  - OpenMP is compiled, so needs significant on-going support



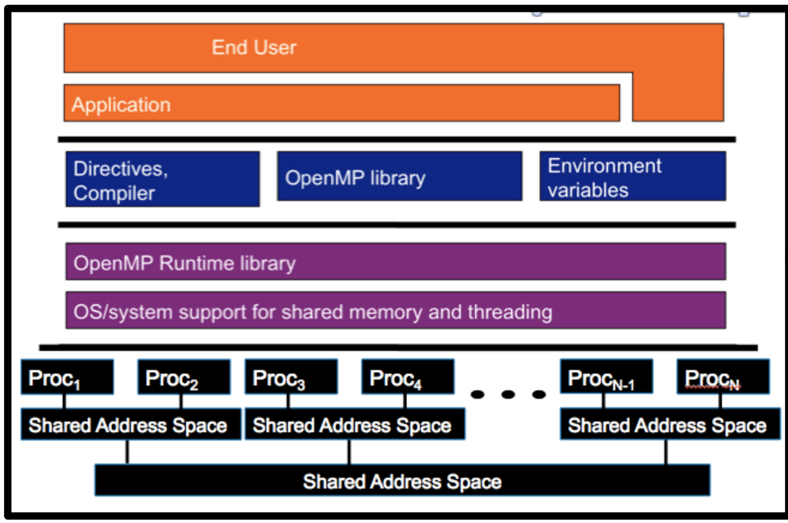
```
#pragma omp parallel
#pragma omp for schedule(dynamic)
    for (i=0; i<N; i++){
        A[i] = sqrt( A[i] );
    } /* implicit barrier here */
```

"High-level directive-based multi-language parallelism that is performant, productive and portable"

http://www.openmp.org/

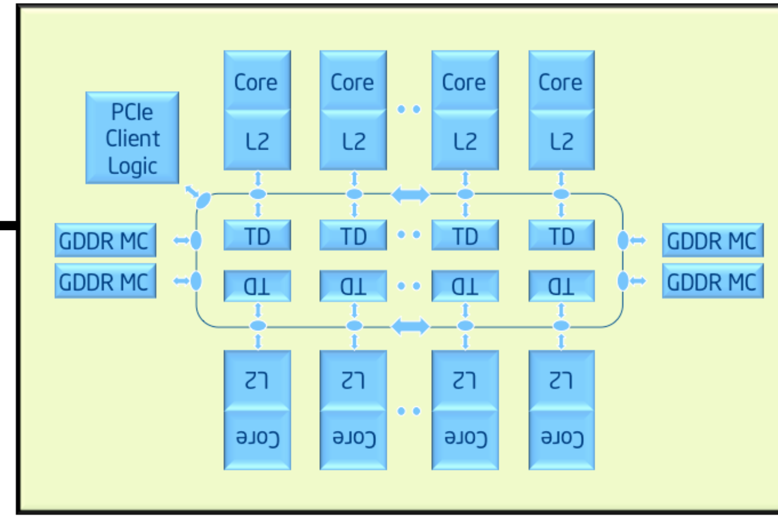
Where Does OpenMP Run?

Supported (since OpenMP 4.0) with target, teams, distribute, and other constructs

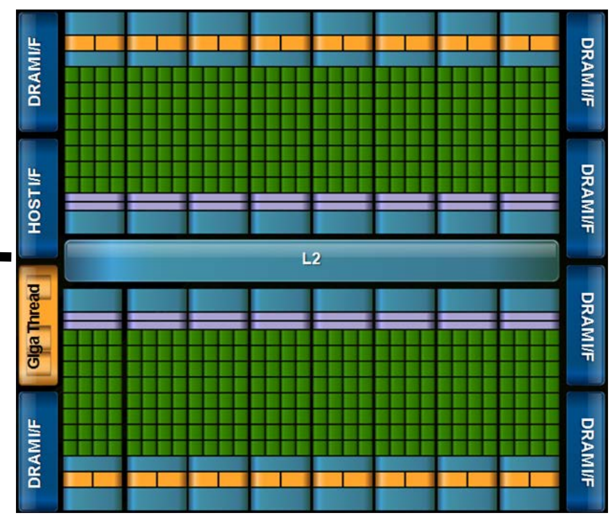


Host

OpenMP 4.5



Target Device: Intel® Xeon Phi™ coprocessor



Target Device: GPU

Basic components of a parallel programming environment

- Team of workers
- Work division among worker
- Sharing and accessing data among workers
- Synchronization among workers

How Does OpenMP Work?

- Teams of OpenMP threads are created to perform the computation in a code
  - **Work is divided** among the threads, which run on the different cores
  - The threads collaborate **by sharing variables**
  - Threads **synchronize** to order accesses and prevent data corruption
  - **Structured programming** is encouraged to reduce likelihood of bugs

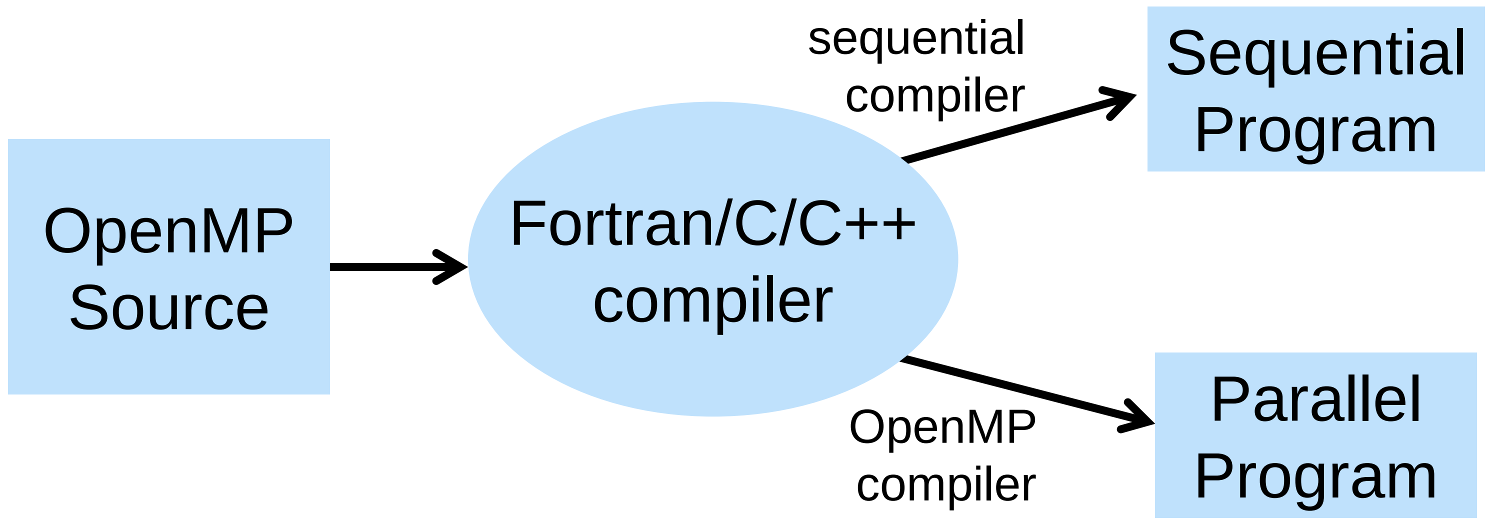
- Most Fortran/C/C++ compilers implement OpenMP
  - Use compiler “flag”, sometimes a specific **optimization level**
- Alternatives:
  - MPI
  - POSIX thread library is lower level
  - Automatic parallelization is higher level (user does nothing)
    - But usually successful on simple codes only

What Does the User Have to Do?

- Starting point is most often MPI or sequential program code
- Application developer must decide how the work can be divided up among multiple threads
  - Identify parallelism and needed synchronization
  - Getting this right is the **user's responsibility**!
  - Insert OpenMP constructs that represent the strategy
- Getting good performance requires an understanding of implications of chosen strategy
  - Translation introduces overheads
  - Data access pattern might affect performance
- **Sometimes, non-trivial rewriting of code is needed to accomplish desired results**

User makes strategic decisions; compiler figures out details

OpenMP Usage



Info on several compilers used in some known HPC centers

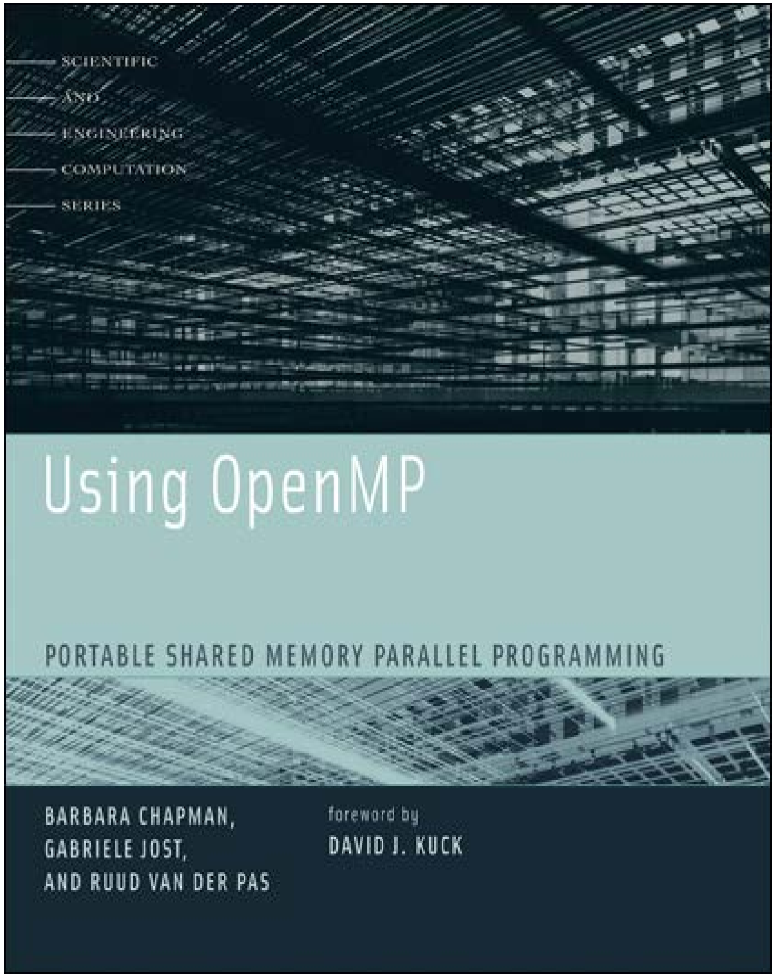
Compiler Name	Compiler Version	OpenMP version	OpenMP flag	C/C++/Fortran compiler
Cray Compilers (cce) [cori, bluewaters, edison]	8.5.X	Most of 4.0	-h omp (None is needed, OpenMP default)	cc, CC (crayc++), ftn
GNU Compiler Collection (gcc) [cori, bluewaters, Edison, stampede 2]	6.3.0	4.5	-fopenmp	gcc, g++, gfortran
Intel Compilers [cori, bluewaters, Edison, stampede 2]	17.0.X	4.5	-qopenmp	icc, icpc, ifort
PGI Compilers [bluewaters]	16.9.0	3.1	-mp=nonuma	pgcc, pgc++, pgfortran (pgf77, pgf90)

Resources

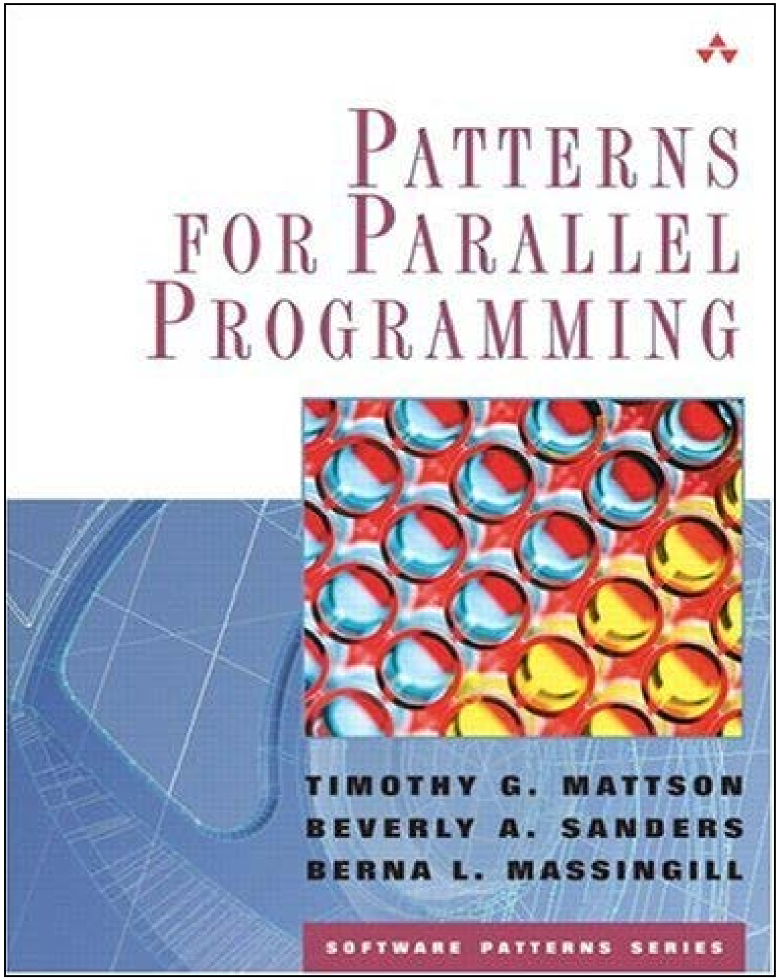
http://www.openmp.org

- We can only give an overview today
  - **We won't cover all features**
- Lots of information available at ARB's website
  - Specifications, technical reports, **summary cards** for downloading
  - Tutorials and publications; links to other tutorials
- Tutorials also at:
  - Supercomputing conferences
  - Annual OpenMPCon, IWOMP workshop
  - Some user sites, e.g. NERSC

Books about OpenMP

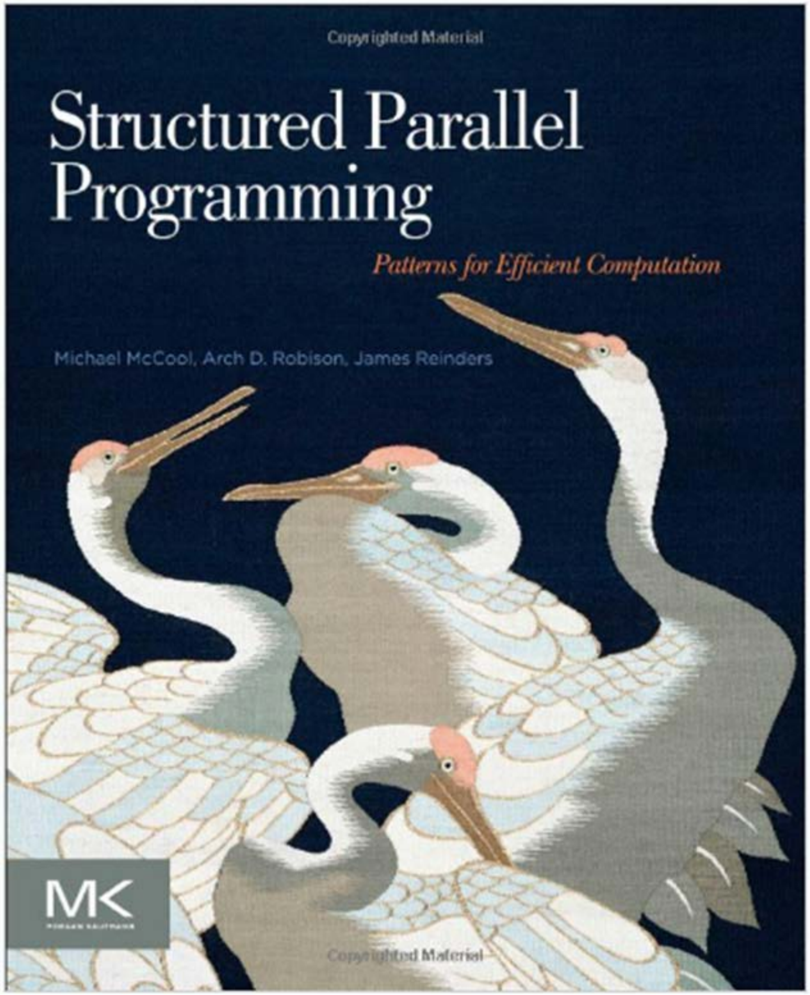


- A book about OpenMP by a team of authors at the forefront of OpenMP's evolution.

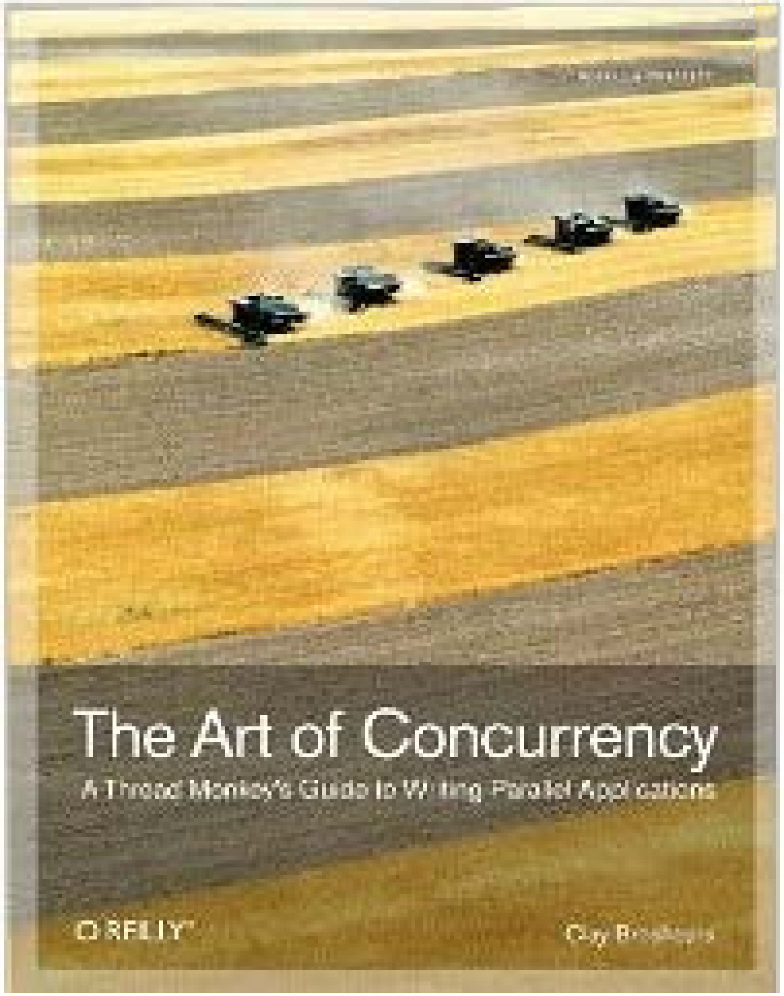


- A book about how to "think parallel" with examples in OpenMP, MPI and java

Background Reference Material



This book explores key patterns with
Cilk, TBB, OpenCL, and OpenMP
(by McCool, Robison, and Reinders)



An introduction to and overview of
multithreaded programming in general
(by Clay Breshears)

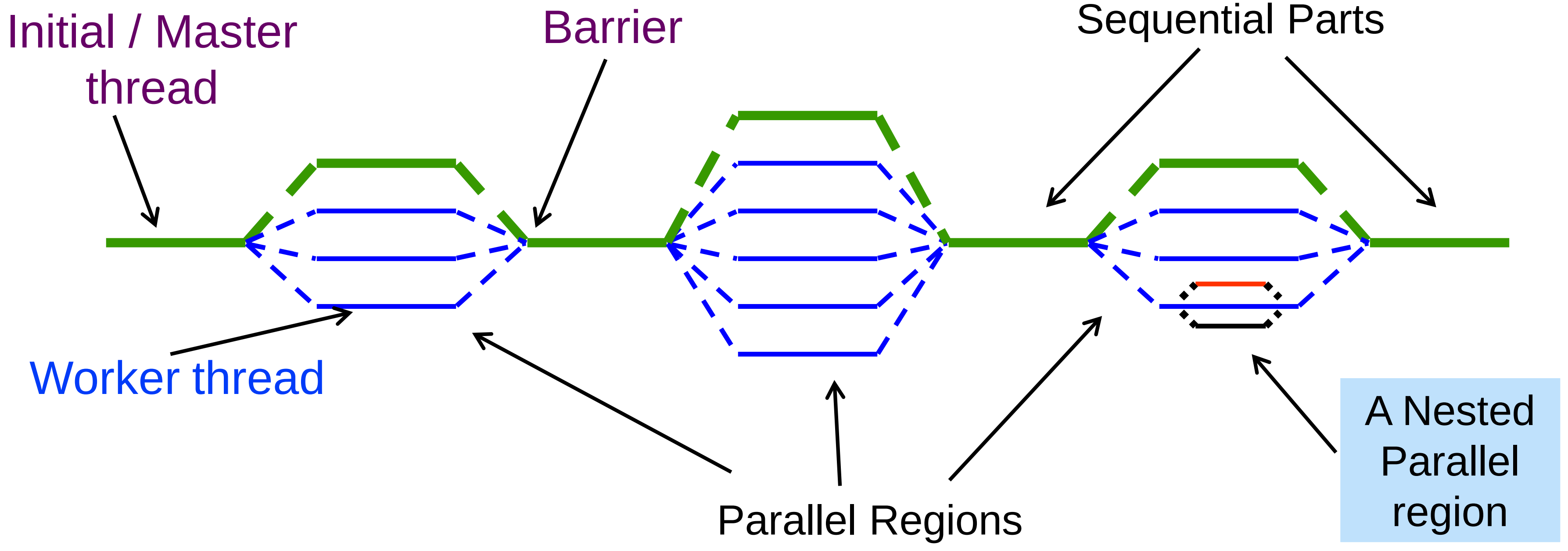
Other books by James Reinders, especially on Xeon Phi multicore programming

Agenda

- Background
- **Parallel Regions**
- Sharing Work
- Handling Data
- Synchronization
- Tasks

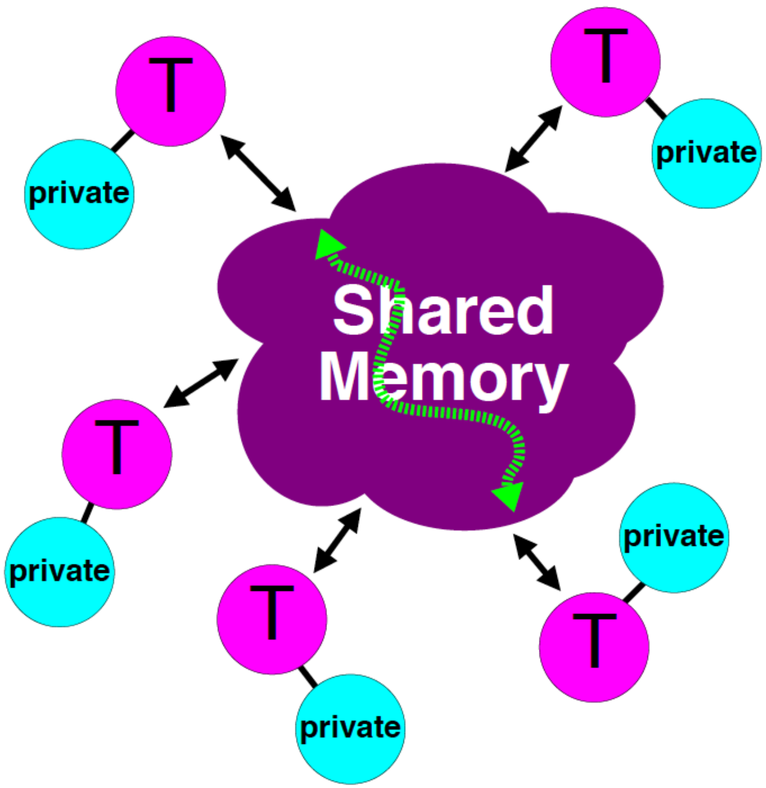
OpenMP Fork-Join Execution Model

- Execution starts with single thread (the initial / master thread)
- Master thread spawns multiple worker threads as needed, together they form a team
- Parallel region is a block of code executed by all threads in a team simultaneously



Number of threads in a team may be dynamically adjusted

OpenMP Memory Model



- All threads access the same, globally shared memory
- Data can be shared or private
  - **Shared** – only one instance of data
    - Threads can access data simultaneously
    - Changes are visible to all threads
      - Not necessarily immediately
  - **Private** - Each thread has copy of data
    - No other thread can access it
    - Changes only visible to the thread owning the data
- OpenMP has **relaxed-consistency** shared memory model
  - Threads may have a *temporary* view of shared memory that is not consistent with that of other threads
  - These temporary views are made consistent at certain places in code

OpenMP Syntax

- Most OpenMP constructs are compiler directives
  - For C and C++, they are pragmas with the form:

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

  - For Fortran, the directives may have fixed or free form:

```
*$OMP construct [clause [clause]…]
C$OMP construct [clause [clause]…]
!$OMP construct [clause [clause]…]
```

- Include C OpenMP header file and the Fortran OpenMP lib module

```
#include <omp.h>    (c)
use omp_lib         (Fortran)
```

- Most OpenMP constructs apply to a **"structured block"**.
  - No spaghetti code, please
  - A block of one or more statements: no arbitrary branching in and out, but it's OK to have a STOP or an exit() within the block

Clauses are all optional and allow the user to provide additional instructions to the implementation

Defining Parallelism In OpenMP

- First step is to specify the **parallel region(s)**
  - A team of threads will be created to execute parallel region; it is terminated at the end of region
    - Threads are managed by OpenMP runtime
    - Threads in team are numbered consecutively, starting from 0; the **master thread has thread ID 0**
  - Thread adjustment is only done before entering a parallel region
  - Parallel regions can be nested; nesting is disabled by default
  - An "if" clause can be used to guard the parallel region; if the condition evaluates to "false", the code is executed serially

OpenMP Team := Master + Workers

Thread Creation: Parallel Regions

- You create threads in OpenMP with the parallel construct.
- A **runtime function** can be used to request a specific number of threads to execute a parallel region; here, we request 4 threads:

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

Thread Creation: Parallel Regions

- Here, an **environment variable** is used to set the team size:

Runtime function returning a thread ID

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

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

terminal@ubuntu:~

```
$ export OMP_NUM_THREADS = 4
```

Shell

Environment variable to initialize number of threads in a parallel region

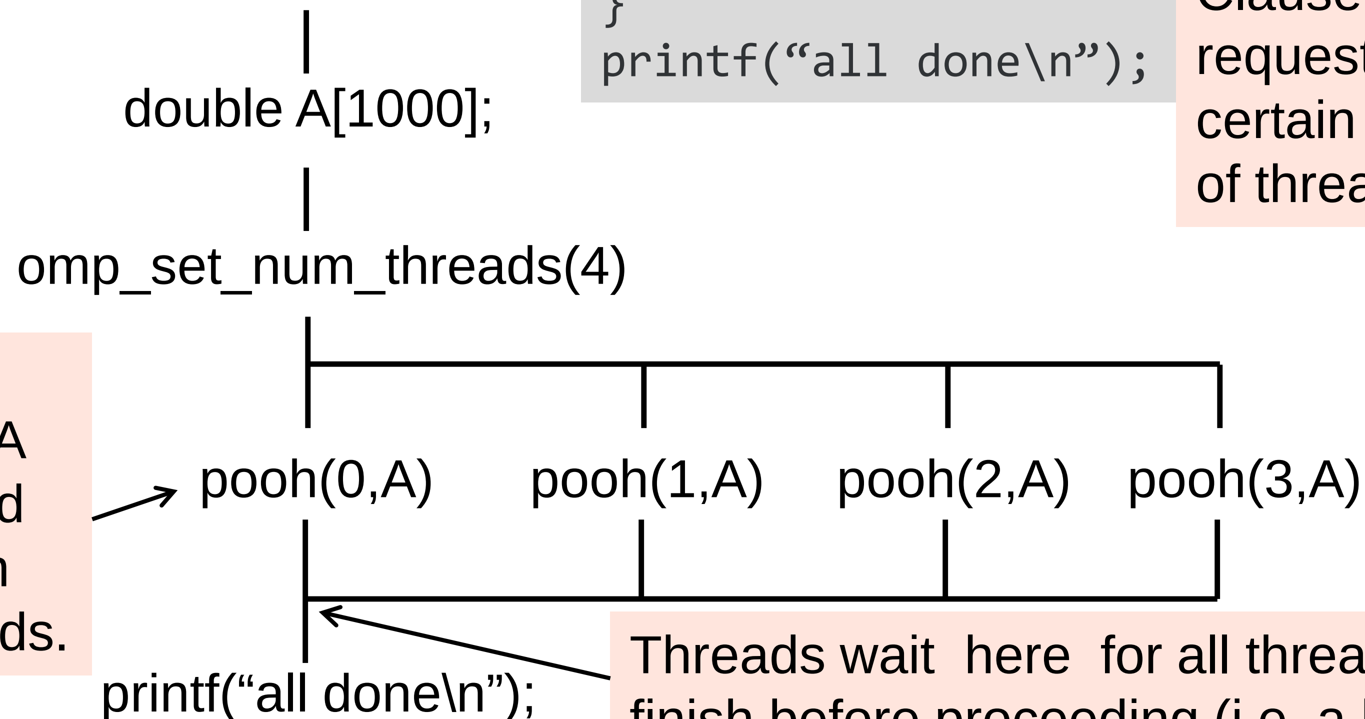
- Each thread calls pooh(ID,A) for `ID` = `0` to 3

Thread Creation: Parallel Regions

- Here a **clause** is used to specify team size
- Each thread executes the same code:

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

Clause to request a certain number of threads



A single copy of A is shared between all threads.

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

Scope of OpenMP Parallel Region

A parallel region can span multiple source files

poo.f

```
      call whoami
C$OMP PARALLEL
      call whoami
C$OMP END PARALLEL
```

bar.f

```
      subroutine whoami
      external omp_get_thread_num
      integer iam, omp_get_thread_num
      iam = omp_get_thread_num()
C$OMP CRITICAL
      print*,'Hello from ', iam
C$OMP END CRITICAL
      return
      end
```

Parallel **construct**; Lexical extent of parallel region

The parallel **region** is the text of the construct plus any code called from inside the construct

Orphan constructs outside lexical extent of parallel region

Example: A Multi-threaded “Hello world” Program

- Write a multithreaded program where each thread prints “hello world”

```
int main(int argc, char *argv[])
{
    int ID = 0;
    printf("hello(%d)", ID);
    printf("world(%d)\n", ID);
    return 0;
}
```

Example: A Multi-threaded "Hello world" Program

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

```
#include <omp.h>
int main(int argc, char *argv[])
{

    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("hello(%d)", ID);
        printf("world(%d)\n", ID);
    }
   return 0;
}
```

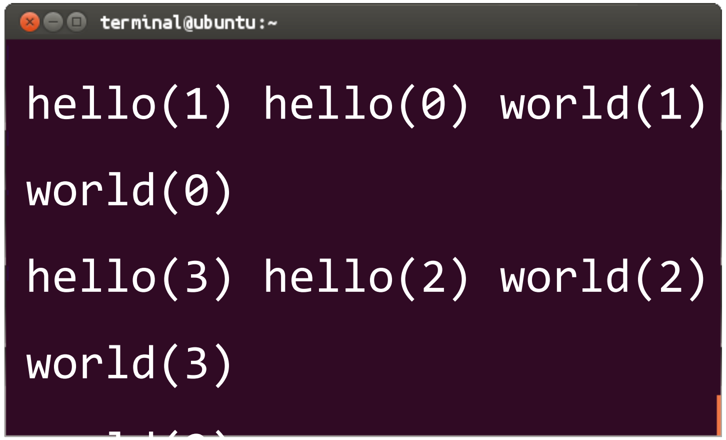
OpenMP include file

Parallel region with default number of threads

End of the parallel region. Threads wait at a barrier

Runtime library function to return a thread ID.

Sample Output:

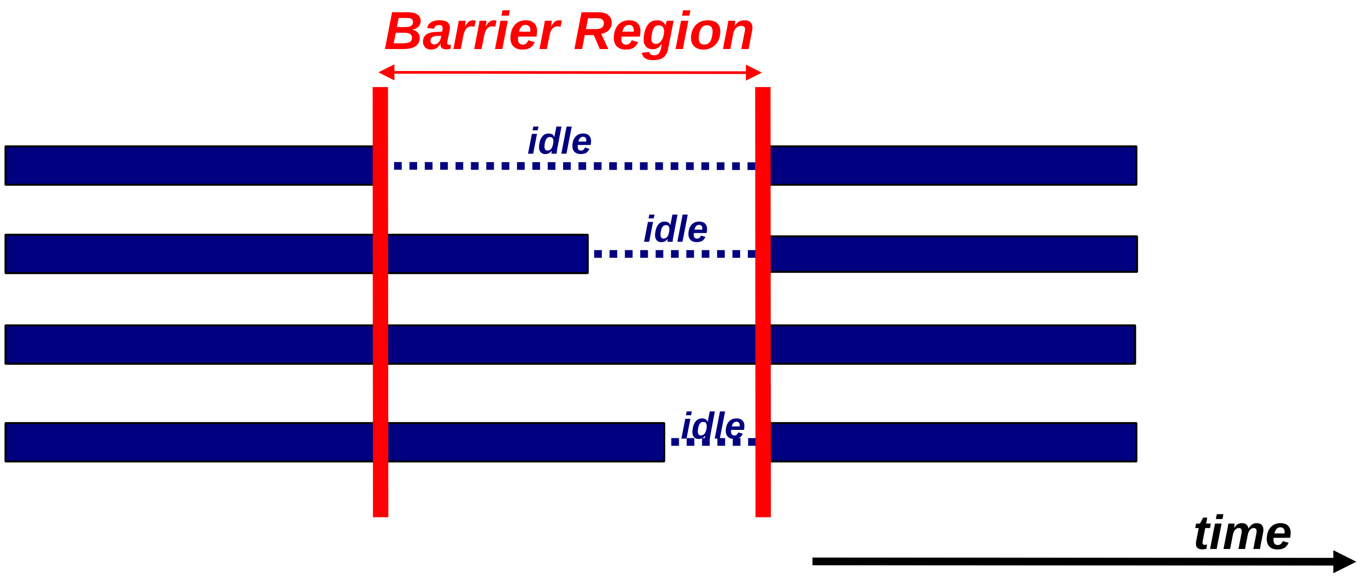


Programming in Pthreads vs. OpenMP

```c
#include <pthread.h>
#define DEFAULT_NUM_THREADS 4

/* encapsulate multiple args to a thread */
typedef struct args {
    int id;        /* this thread's number */
} args_t;

/* function that is run inside each thread */
void *do_hello_world(void *arg)
{
    args_t *ap = (args_t *) arg; /* unpack incoming args */
    printf("Hello from thread %d\n", ap->id);    /* ACTUAL WORK */
    return NULL;
}

int main(int argc, char *argv[])
{
    int i, num_threads = DEFAULT_NUM_THREADS;
    pthread_t *thread_pool;
    args_t *thread_args;

    if (argc > 1) {
        num_threads = atoi(argv[1]);
        if (num_threads < 0) {
            num_threads = DEFAULT_NUM_THREADS;
        }
    }
    thread_pool = (pthread_t *) malloc(num_threads *
                                   sizeof(*thread_pool));
    thread_args = (args_t *)    malloc(num_threads *
                                   sizeof(*thread_args));
    /* create and run threads: pass id of thread to each */
    for (i = 0; i < num_threads; i += 1) {
        thread_args[i].id = i;
        pthread_create(&thread_pool[i], NULL, do_hello_world,
                       (void *) &thread_args[i]);
    }
    /* wait for all threads to finish */
    for (i = 0; i < num_threads; i += 1) {
        pthread_join(thread_pool[i], NULL);
    }
    free(thread_args);
    free(thread_pool);
    return 0;
}
```

```c
int main(int argc, char *argv[]) {
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("hello from thread %d\n", ID);
  }
  return 0;
}
```

Explicit Barriers



Barrier syntax in OpenMP:

```
#pragma omp barrier
```

```
!$omp barrier
```

- A barrier is automatically inserted at the end of each parallel region. We can also add barriers to the code using this directive.

Using The Master Thread Only

- The **master** construct denotes a structured block that is only executed by the master thread. The other threads just skip it.
- There is no barrier at the end of the master construct.

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

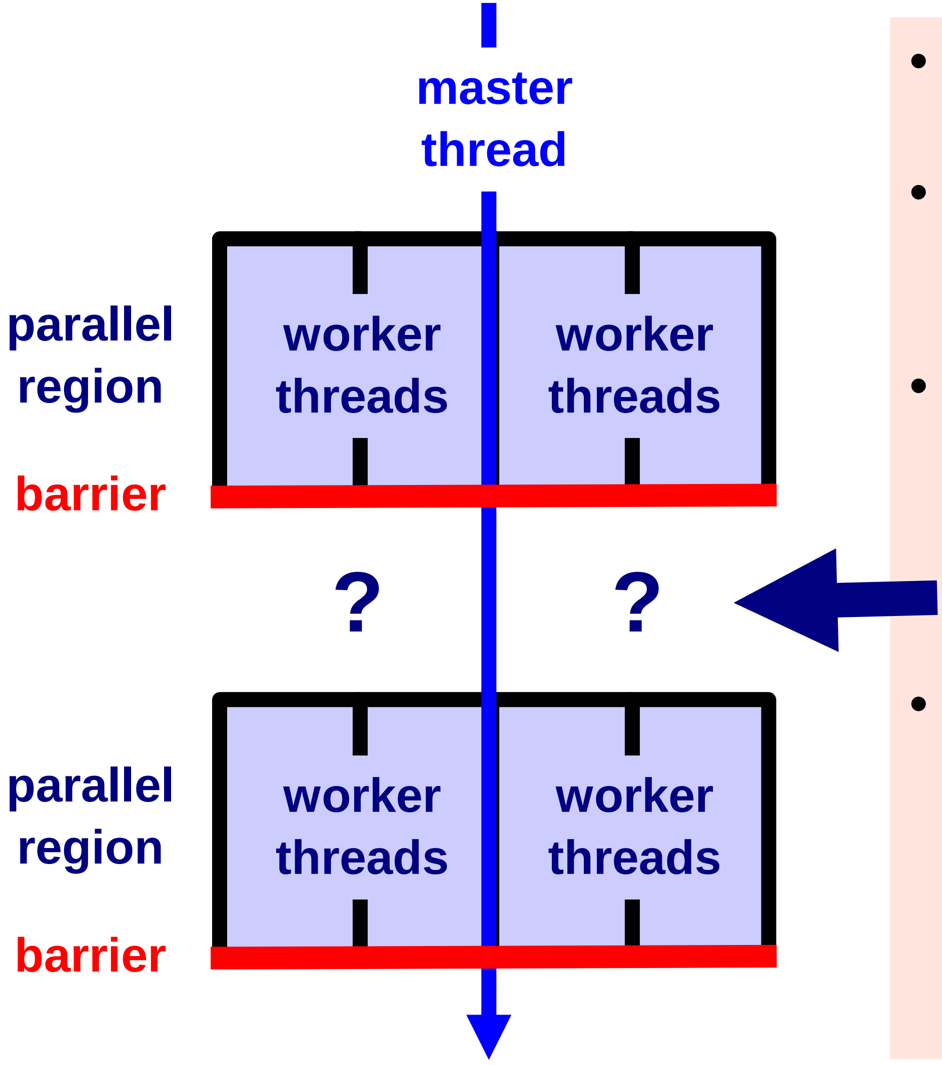
OpenMP Features We Have Seen So Far

- Header file, directives, environment variable and runtime library routines

```c
#include <omp.h>
int main(int argc, char *argv[])
{
   #pragma omp parallel
   {
      int ID = omp_get_thread_num();
      printf("hello(%d)", ID);
      printf("world(%d)\n", ID);
   }
   return 0;
}
```

```c
#pragma omp master
{
   printf("hello(%d)", ID);
}
#pragma omp barrier
```

Controlling Threads: Environment Variables



- OMP_THREAD_LIMIT
  - Upper limit on #threads
- OMP_NUM_THREADS
  - Initialize # threads in team
- OMP_DYNAMIC
  - Enable adjustment?
  - TRUE or FALSE
- OMP_WAIT_POLICY
  - Controls idle thread behavior
  - ACTIVE or PASSIVE
  - No runtime routine for this

- Runtime library routines can be used to override some of the initial values (whether set by environment variable or implementation)

Controlling Threads: Runtime Library Routines

- To use a known, fixed number of threads in a program, you can (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>
int main(int argc, char *argv[])
{
    int num_threads;
    omp_set_dynamic(0);
    omp_set_num_threads(omp_get_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);
    }
    return 0;
}
```

Disable dynamic adjustment of the number of threads.

Request as many threads as you have processors.

Only one thread retrieves the value on behalf of all threads

The system may still give you fewer threads than requested. If the precise # matters, test for it and respond accordingly.

omp_get_max_threads() returns max # threads available to form a team

Example Environment Variable Defaults

OpenMP Environment Variable	Cray Compiler 8.5 Defaults
OMP_NUM_THREADS	1
OMP_THREAD_LIMIT	4 times the number of available processors
OMP_DYNAMIC {TRUE | FALSE}	TRUE
OMP_NESTED {TRUE | FALSE}	FALSE
OMP_MAX_ACTIVE_LEVELS	1023
OMP_WAIT_POLICY [ACTIVE | PASSIVE]	ACTIVE
OMP_SCHEDULE "schedule,[chunk]"	STATIC, 0
OMP_STACKSIZE "size [B|K|M|G]"	128 MB
Be careful when relying on defaults (they are compiler dependent)	

Performance Tips

- Experiment to find the best number of threads on your system
- Put as much code as possible inside parallel regions
  - Amdahl's law: **If 1/s of the program is sequential, then you cannot ever get a speedup better than s**
  - So if 1% of a program is serial, speedup is limited to 100, no matter how many processors it is computed on
- Have large parallel regions
  - Minimize overheads: starting and stopping threads, moving data into cache
  - Directives can be "orphaned"; procedure calls inside regions are fine
- Run-time routines are your friend
  - Usually very efficient and allow maximum control over thread behavior
- Barriers are expensive
  - With large numbers of threads, they can be slow
  - Depends in part on HW and on implementation quality
  - Some threads might have to wait a long time if load not balanced

Agenda

- Background
- Parallel Regions
- **Sharing Work**
- Handling Data
- Synchronization
- Tasks

Worksharing Constructs: Loops

- Worksharing constructs divide the execution of the enclosed code region among the members of the team of threads
- The "for"/ "do" worksharing construct splits up loop so that each thread in team gets adjacent loop iterations
  - Each thread gets one or more "chunks" of iterations

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

Barrier at end of parallel loop

`$omp do` in Fortran

By default, all threads wait at a barrier at the end of the "`omp for`". Use "`nowait`" clause to remove the barrier.

```
#pragma omp for nowait
```

"`nowait`" is useful between two consecutive, independent parallel loops.

`$omp end do nowait` in Fortran

Work Sharing Loops and Scheduling

Sequential code

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

OpenMP parallel region

```
#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;
  for (i=istart; i<iend; i++) {
    a[i] = a[i] + b[i];
  }
}
```

OpenMP parallel region and a work-sharing for-construct

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

OpenMP Schedule Clause

- The schedule clause affects how loop iterations are mapped onto threads
- schedule (static | dynamic | guided [, chunk])
- schedule (auto | runtime)

static	Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion
dynamic	Fixed portions of work; size is controlled by the value of chunk. When a thread finishes, it starts on the next portion of work
guided	Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially
auto	The compiler (or runtime system) decides what is best to use; choice could be implementation dependent
runtime	Iteration scheduling scheme is set at runtime via environment variable OMP_SCHEDULE or runtime library call

Reduction Operations

- Many calculations combine values into a single accumulation variable at some point, e.g. to determine an overall error
- Such a so-called **reduction** leads to a true dependence between loop iterations
- Reductions are common and prevent us from parallelizing loops

E.g., to calculate the L2-norm error, we sum up the contributing *diffs*:

```
double err_2_norm(size_t N, double *x, double *y)
{
    double sum = 0;
    for (size_t i = 0; i < N; i++) {
        double diff = fabs(x[i] - y[i]);
        sum += diff * diff;
    }
    return sqrt(sum);
}
```

$$|z| = \sqrt{\sum_{k=1}^{n} |x_k - y_k|^2}$$

Reductions

```
reduction ( operator: list )
```
C/C++

```
reduction ( [operator | intrinsic] : list)
```
Fortran

- Inside a parallel or work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “operator” (e.g. 0 for “+”).
  - Local copies are updated
  - Local copies are reduced into a single value and combined with the original global value.
- Variables in “list” must be shared in enclosing parallel region

```
double err_2_norm(size_t N, double *x, double *y)
{
    double sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for (size_t i = 0; i < N; i++) {
        double diff = fabs(x[i] - y[i]);
        sum += diff * diff;
    }
    return sqrt(sum);
}
```

Result variable is shared by default

Predefined Reductions

- 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.

User Defined Reductions (version 4.0)

- For mathematically associative and commutative operations
- Declare the reduction operator
  - Name, type, combiner function, initialization of local copies
  - Use special identifiers omp_in for value to be combined, omp_out for resulting combined value, omp_priv to initialize private copy

```
int my_mul(int a, int b) { return a * b; }

#pragma omp declare reduction(mul_id : int : omp_out *= omp_in) \
    initializer(omp_priv = 1)
```

- Use the reduction operator in a `reduction` clause

```
#pragma omp parallel for reduction(mul_id : prod_par)
    for (i = 0; i < ARRAY_SIZE; i += 1) {
        prod_par = my_mul(prod_par, array[i]);
    }
```

Parallelizing Multiple Loops in Nest

- Allows parallelization of multiple loops in perfectly nested loop nests without using nested parallelism

```
#pragma omp parallel for private(i,j) collapse(2)\
            reduction(+:sum)
    for ( i = 0; i < 8; i++) {
        for ( j = 0; j < 1000; j++) {
            sum += A[i][j];
        }
    }
```

Number of loops to be parallelized, counting from the outside

- Compiler forms and parallelizes a single loop of length NxM.
- Useful if N is O (no. of threads) so parallelizing the outer loop makes balancing the load difficult.

Working with Loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent, so they can safely execute in any order without loop-carried dependencies
  - Insert the appropriate OpenMP directive(s) and test
  - Now tune: reduce synchronization; ensure data locality, optimize cache behavior

Note: loop index "i" is private by default

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

Modify this to get independent loops

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

Limitations of Parallel For / Do

```
#pragma omp parallel
{
    …
    while (my_pointer != NULL) {
        do_independent_work(my_pointer);
        my_pointer = my_pointer->next;
    } // End of while loop
    …
}
```

To use a for or do construct, loops must be countable.

To parallelize this loop, it is necessary to first count the number of iterations and then rewrite it as a *for* loop.

Or we can use tasks. More on this later…

Worksharing Constructs: Sections

- 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.

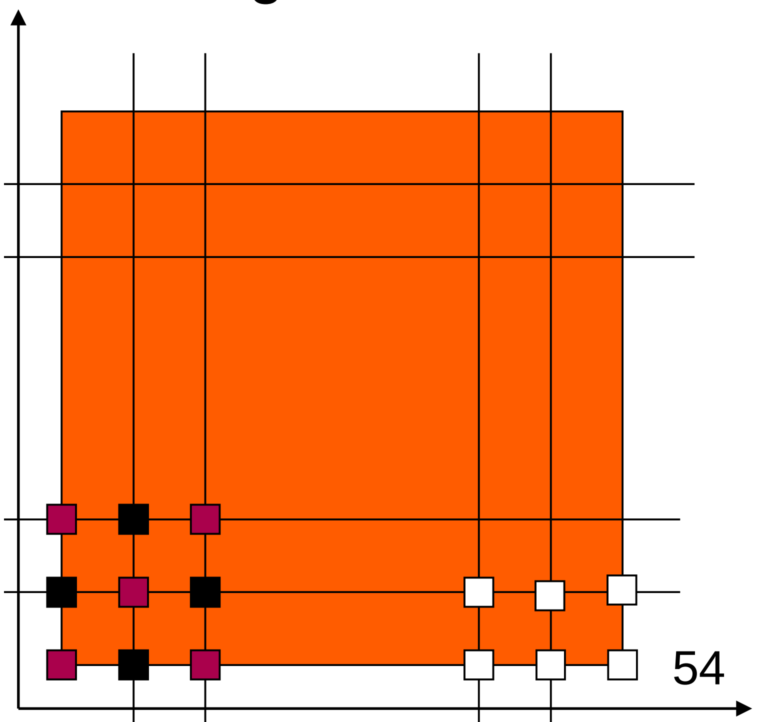
Work-Sharing Constructs: Single

- The **single** construct denotes a block of code that is executed by only one thread
- A barrier is implied at the end of the single block

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

Exercise: Red-Black Method in Parallel

- Grid points partitioned into two sets like a chess board
  - “colored” red and black
- Update in two steps
  - Compute new values on “red” points using current values on neighboring “black” points
  - Compute new values on “black” points using current values on neighboring “red” points
- To parallelize, consider:
  - Which loops to parallelize?
  - What loop schedules?
  - Needed synchronization?



Red-Black Method

```
for some number of timesteps/iterations {// update red points
    for (i=1; i<n; i+=2 ) ←parallel
        for (j=1; j<n; j+=2 ) ←parallel
            grid [i][j] = 0.25 *
                           ( grid[i-1][j] + grid[i+1][j]
                           + grid[i][j-1] + grid[i][j+1] );
    for (i=2; i<n; i+=2 ) ←parallel
        for( j=2; j<n; j+=2 ) ←parallel
            grid [i][j] = 0.25 *
                           ( grid[i-1][j] + grid[i+1][j]
                           + grid[i][j-1] + grid[i][j+1] );
}
```

Each loop nest here updates half of the red points. The points used to compute the updates are all black points.

A similar pair of loops update the black points, using red points.

Exercise: Red Black Method

```
/* Parallelizing the outer loop is not possible, so we will parallelize one level below */
for (n=0; n < ITERNUM; n++) {
    #pragma omp parallel {
        /* Update red points */
        /* @TODO: Insert OpenMP pragma here, hint: omp for, collapse, schedule, nowait */
        for (i = 1; i < DSIZE_X - 1; i += 2)
            for (j = 1; j < DSIZE_Y - 1; j += 2)
                grid[i][j] = 0.25 * (grid[i-1][j] + grid[i+1][j] +
                                     grid[i][j-1] + grid[i][j+1]);
        /* @TODO: Insert OpenMP pragma here, hint: omp for, collapse, schedule, nowait */
        for (i = 2; i < DSIZE_X - 1; i += 2)
            for (j = 2; j < DSIZE_Y - 1; j += 2)
                grid[i][j] = 0.25 * (grid[i-1][j] + grid[i+1][j] +
                                     grid[i][j-1] + grid[i][j+1]);
        /* Update black points */
        /* @TODO:Insert OpenMP pragma here, hint:omp for, collapse, schedule, nowait */
        for (i = 1; i < DSIZE_X - 1; i += 2)
            for (j = 2; j < DSIZE_Y - 1; j += 2)
                grid[i][j] = 0.25 * (grid[i-1][j] + grid[i+1][j] +
                                     grid[i][j-1] + grid[i][j+1]);
        /* @TODO:Insert OpenMP pragma here, hint:omp for, collapse, schedule, nowait */
        for (i = 2; i < DSIZE_X - 1; i += 2)
            for (j = 1; j < DSIZE_Y - 1; j += 2)
                grid[i][j] = 0.25 * (grid[i-1][j] + grid[i+1][j] +
                                     grid[i][j-1] + grid[i][j+1]);
    }}
```

Performance Tips

- Is there enough work to amortize overheads?
  - May not be worthwhile for very small loops (if clause can control this)
  - Might be overcome by choosing different loop, rewriting loop nest or collapsing loop nest
- Best choice of schedule might change with system, problem size
  - Experimentation may be needed
- Minimize synchronization
  - Use nowait where possible
- Locality
  - Most large systems are NUMA
  - Be prepared to modify your loop nests
  - Change loop order to get better cache behavior
- If performance is bad, look for false sharing
  - We talk about this in part 2 of the tutorial
  - Occurs frequently, performance degradation can be catastrophic

Agenda

- Background
- Parallel Regions
- Sharing Work
- **Handling Data**
- Synchronization
- Tasks

What About The Data?

- There is only one instance of **shared** data
  - Threads can read and write the data simultaneously unless protected through a specific construct
  - All changes are visible to all threads (not necessarily immediately)
- Each thread has its own copy of **private** data
  - No other thread can access it *in any way*
    - Changes only visible to the thread owning the data
- Most, but not all, variables are shared by default
  - Shared by default: Global variables; Fortran: COMMON blocks, SAVE variables, MODULE variables; C: File scope variables, static
  - Private by default: Stack (local) variables in sub-programs called from parallel regions; Automatic variables in a statement block
  - **Tasks** have different defaults
- The default status can be modified with:
  - DEFAULT (PRIVATE | SHARED | NONE)

Private Clause

- private(var) creates a local copy of var for each thread.
  - The value is uninitialized
  - Private copy is *not* storage-associated with the original
  - The original is undefined at the end

```
#pragma omp parallel for private(tmp)
    for(i=0; i<n; i++){
        tmp= 2.0*a[i];
        a[i] = tmp;
        b[i] = c[i]/tmp;
    }
```

Without the private clause, all threads would update `tmp` at roughly the same time

`a` and `c` are shared data; values defined prior to loop

- Parallel loop variable is private by default

Firstprivate Clause

- Variables initialized from a shared variable
- C++ objects are copy-constructed

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

Each thread gets its own copy of `incr` with an initial value of 0

Value of `incr` is unspecified after parallel region

OpenMP Data Environment

```
double a[size][size], b=4;
#pragma omp parallel private (b)
{   ....  }
```

shared data a[size][size]			
private data b = ?	private data b = ?	private data b =?	private data b =?
T0	T1	T2	T3

Private variable b becomes undefined on exit from region

- Private variables are undefined on entry and exit of the parallel region
- A private variable within a parallel region has *no* storage association with the same variable outside of the region
- **Firstprivate** initialize private data; **lastprivate** causes variable outside region to be updated after end of region

Recap: Private and Firstprivate Data

- Consider this example of PRIVATE and FIRSTPRIVATE

```
int a=1, b=1, 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 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 undefined.

Parallel Random Number Generator (RNG)

```
void seed(double low_in, double hi_in)
{
 /* Sets the per thread range and seed (random_last) */
}
double drandom()
{
    random_last = (mult_n  * random_last)% PMOD);
    rand_num = (random_last /PMOD)*(rand_hi-rand_low)+rand_low;
    return rand_num;
}
```

```
#pragma omp parallel
{
    seed(low, high);
    for (int i=0; i<10; i++)
         drandom();
    ...
}
```

- Seed is used and updated each time a number is generated
- Need to retain value across different parallel regions
- Want values to be local to thread

Threadprivate Data

- To retain the last seed for each thread in our parallel RNG code across different parallel regions, we can declare this variable to be threadprivate:

```
unsigned long long random_last = 0;
#pragma omp threadprivate(random_last)
void seed(double low_in, double hi_in)
   . . . .
```

- Threadprivate makes global data private to a thread
  - Fortran: `COMMON` blocks
  - C: File scope and static variables
- Threadprivate variables persist between parallel regions!
  - They can be initialized using `COPYIN` or `DATA` statements; values can be broadcast from one thread to the others using `COPYPRIVATE`

Performance and Correctness Tips

- There is one version of shared data
  - Keeping data shared reduces overall memory consumption
- Private data is stored locally, so use of private variables can increase efficiency
  - Avoids false sharing
  - May make it easier to parallelize loops
  - But private data is no longer available after parallel regions ends

- It is an error if multiple threads update the same variable at the same time (a data race)
- It is a good idea to use “default none” while testing code
- Putting code into a subroutine / function can make it easier to write code with many private variables
  - Local / automatic data in a procedure is private by default

Agenda

- Background
- Parallel Regions
- Sharing Work
- Handling Data
- **Synchronization**
- Tasks

OpenMP Synchronization

- Synchronization enables the user to
  - Control the ordering of executions in different threads
  - Ensure that at most one thread executes operation or region of code at any given time (mutual exclusion)
- High level synchronization:
  - barrier
  - critical section
  - Atomic
  - ordered
- Low level synchronization:
  - flush
  - locks (both simple and nested)

Barrier: Explicit and Implicit

- 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_calc3(id);
}
```

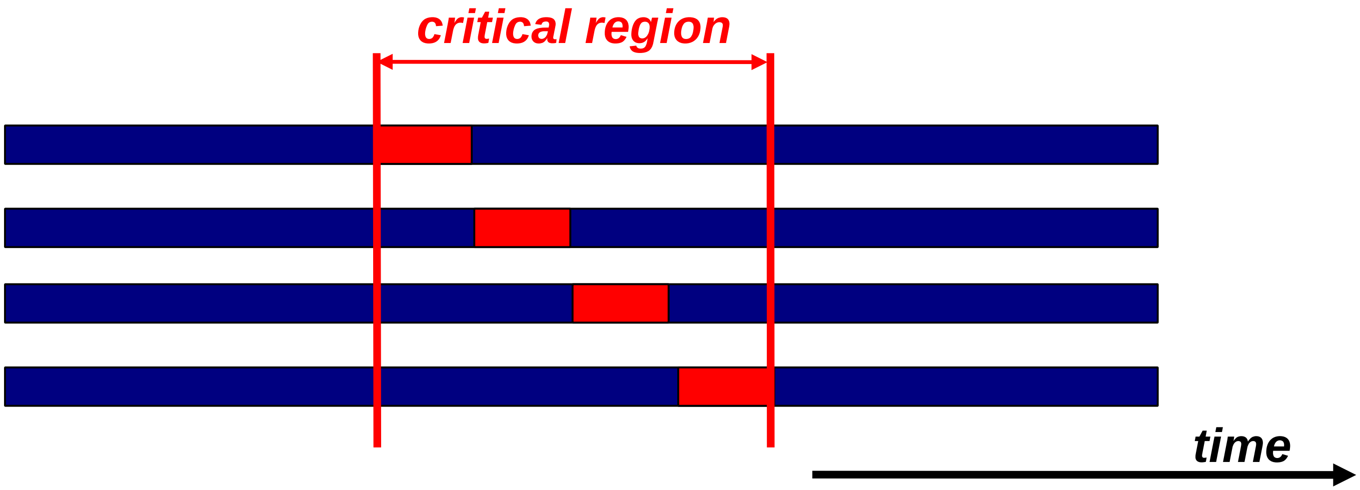
implicit barrier at the end of a for work-sharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Mutual Exclusion

- Code may only be executed by at most one thread at any given time
- Could lead to long wait times for other threads
  - Atomic updates for individual operations
  - Critical regions and locks for structured regions of code



Why Is this needed?

- When multiple threads attempt to manipulate the same data item, the results can often be unexpected if proper care is not taken.
- Consider a joint account, deposited by both the wife and the husband

```
/*Thread 1:the wife deposits $1500 */
   ourBalance = ourBalance + $1500
/*Thread 2:the husband deposits $1000 */
   ourBalance = ourBalance + $1000
```

  - Assuming the initial balance is $500, the final value of `ourBalance` is exepected to be $3000.
  - However, depending on the schedule of the threads, the value of `ourBalance` could be $3000, $1500 or $2000!

How could this happen?

ourBalance = $500

```
/* T1: The wife */

T1: t1 = ourBalance
T1: t1+ = 1000
T1: ourBalance = t1
```

```
/*T2: The husband */

T2: t2= ourBalance
T2: t2+= 1500
T2: ourBalance = t2
```

Concurrent execution

```
/*schedule 1*/
T1: t1 = ourBalance /*500*/
T1: t1 +=1000          /*1500*/
T1: ourBalance = t1 /*1500*/
T2: t2 = ourBalance /*1500*/
T2: t2+= 1500          /*3000*/
T2: ourBalance= t2 /*3000*/
```

```
/*schedule 2*/
T1: t1 = ourBalance /*500*/
T2: t2 = ourBalance /*500*/
T1: t1 += 1000          /*1500*/
T2: t2 += 500          /*2000*/
T1: ourBalance = t1 /*1500*/
T2: ourBalance = t2 /*2000*/
```

Interrupting the modifications to the shared data is dangerous!

Solution: Critical Section

- Critical section: a code segment that must be executed by only one thread at any time.
  - Thread cooperates by acquiring a lock before accessing the corresponding data
    - Pthreads: pthread_mutex_lock, pthread_mutex_unlock
    - OpenMP: omp critical or omp_set_lock, omp_unset_lock
- Mutex-locks have two states: locked and unlocked.
  - Lock::Acquire(): wait until lock is unlocked, then set it to locked
  - Lock::Release(): release the lock to unlocked state

T1: pthread_mutex_lock(&cs_mutex) T1: t1 = ourBalance T1: t1+ = 1000 T1: ourBalance = t1 T1: pthread_mutex_unlock(&cs_mutex)	T2: pthread_mutex_lock(&cs_mutex) T2: t2= ourBalance T2: t2+= 1500 T2: ourBalance = t2 T2: pthread_mutex_lock(&cs_mutex)

Atomic

- Atomic is a special case of mutual exclusion
- It usually applies only to the update of a memory location

```
#pragma omp parallel private(b)
{
     b=do_it(i);
     tmp = big_ugly();
     #pragma omp atomic
          x=x+tmp
}
```

Mutual Exclusion: critical and atomic

```
long balance[NUM_ACCOUNTS] = {INIT_BALANCE, INIT_BALANCE};
long transaction[NUM_TRANSACTION] = {10, 20, 30, -40, -50, 80, -10, -50, 100,
90};
#pragma omp parallel for
for(i=0; i<NUM_TRANSACTION ; i++) {
    balance[i%2] += transaction[i];
}
```

Parallelized ***without*** proper synchronization

```
long balance[NUM_ACCOUNTS] = {INIT_BALANCE, INIT_BALANCE};
long transaction[NUM_TRANSACTION] = {10, 20, 30, -40, -50, 80, -10, -50, 100,
90};
#pragma omp parallel for
for(i=0; i<NUM_TRANSACTION ; i++) {
    #pragma omp critical
    balance[i%2] += transaction[i];
}
```

Parallelized and using ***omp critical*** update

```
long balance[NUM_ACCOUNTS] = {INIT_BALANCE, INIT_BALANCE};
long transaction[NUM_TRANSACTION] = {10, 20, 30, -40, -50, 80, -10, -50, 100,
90};
#pragma omp parallel for
for(i=0; i<NUM_TRANSACTION ; i++) {
    #pragma omp atomic
    balance[i%2] += transaction[i];
}
```

Parallelized and using ***omp atomic*** update

Data Races / Race Condition

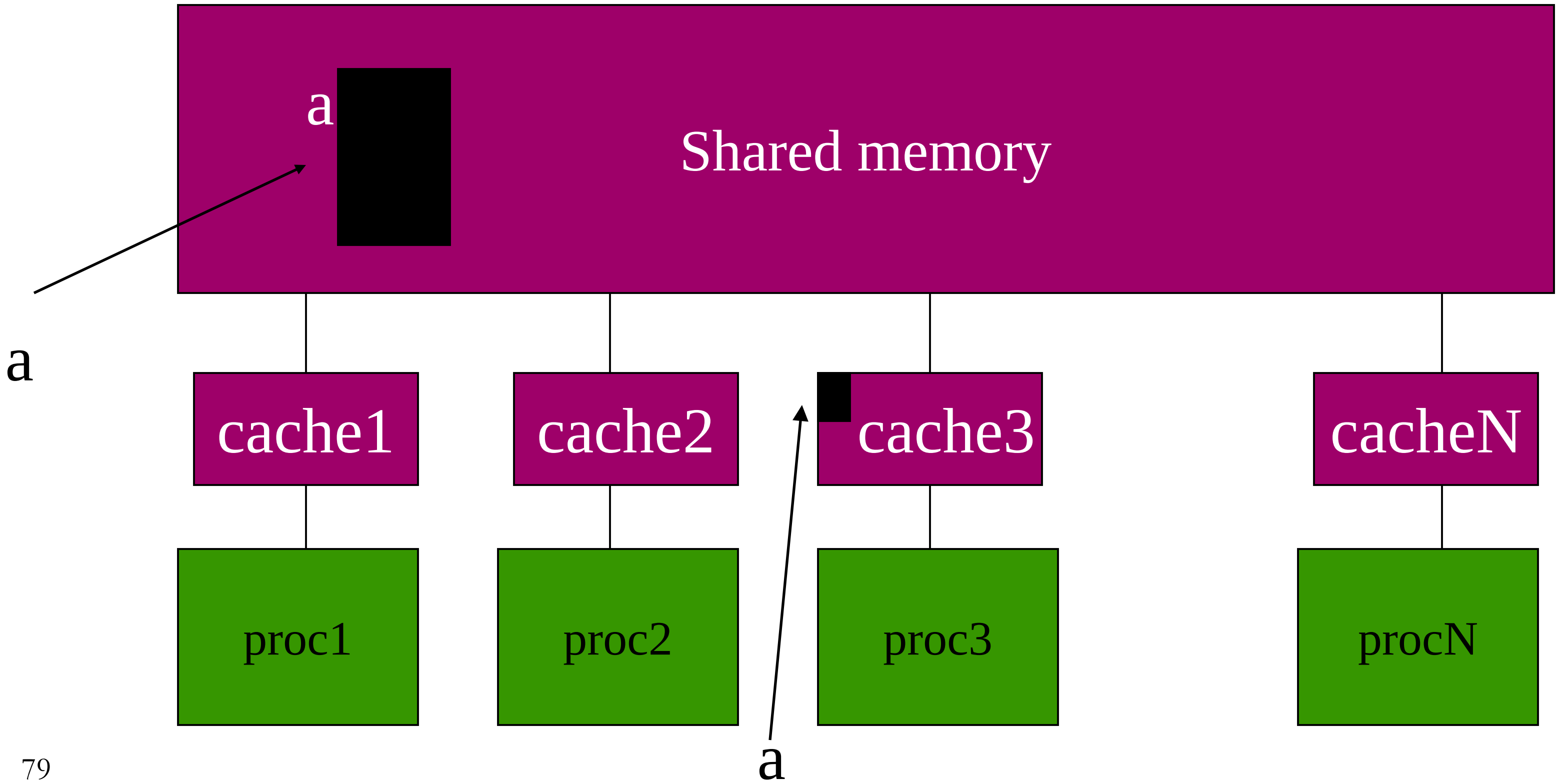
- Data race occurs when two ore more threads access and update shared data “more or less” concurrently
  - One thread writes and one or more threads read or write same memory location at about the same time
  - Outcome depends on relative ordering of operations and may differ between runs
- User is expected to avoid race conditions
  - insert synchronization constructs as appropriate, or
  - privatize data
- Some tools exist to detect data races at runtime
  - e.g. Intel Thread Checker, Oracle Solaris Studio Thread Analyzer

Care with Synchronization

- Recall that a thread's temporary view of memory may vary from shared memory
  - Value of shared objects updated at synchronization points
  - User must be aware of the point at which modified values are (guaranteed to be) accessible
- Compilers routinely reorder instructions that implement a program
  - Helps exploit the functional units, keep machine busy
- Compiler cannot move instructions past a barrier
  - Also not past a flush on all variables
  - But it can move them past a flush on a set of variables so long as those variables are not accessed

Updates to Shared Data

- Blocks of data are fetched into cache lines
- Values may temporarily differ from other copies of data within a parallel region



Updates to Shared Data

Thread A

```
X = 0
:
:
X = 1
:
:
```

Thread B

```
while (X == 0)
{
     "wait"
}
```

If shared variable X is kept within a register, the modification may not be immediately visible to the other thread(s)

The Flush Directive

- Flushing is what creates a consistent view of shared data: it causes a thread to write data back to main memory and retrieve new values of updated variables
- It is automatically performed on a number of constructs
- The flush construct allows the programmer to define a point where a thread makes its variable values consistent with main memory
  - Caution: it does not enable a thread to retrieve values updated by another thread unless that thread also performs a flush
  - It also does not synchronize threads
  - Its use is tricky: be sure you understand it

What Else Does Flush Influence?

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

Compilers reorder instructions to better exploit the functional units and keep the machine busy

- Flush prevents the compiler from doing the following:
  - Reorder read/writes of variables in a flush set relative to a flush.
  - Reorder flush constructs when flush sets overlap.
- A compiler CAN do the following:
  - Reorder instructions NOT involving variables in the flush set relative to the flush.
  - Reorder flush constructs that don't have overlapping flush sets.

Implied Flush

Flushes are implicitly performed during execution:

- In a *barrier* region
- At *exit from* worksharing regions, unless a nowait is present
- At *entry to and exit from* parallel, critical, ordered and parallel worksharing regions
- During omp_set_lock and omp_unset_lock regions
  - During omp_test_lock, omp_set_nest_lock, omp_unset _nest_lock and omp_test_nest_lock regions, if the region causes the lock to be set or unset
- Immediately *before and after* every task scheduling point
- At *entry to and exit from* atomic regions, where the list contains only the variable updated in the atomic construct
- But *not* on entry to a worksharing region, or entry to/exit from a master region,

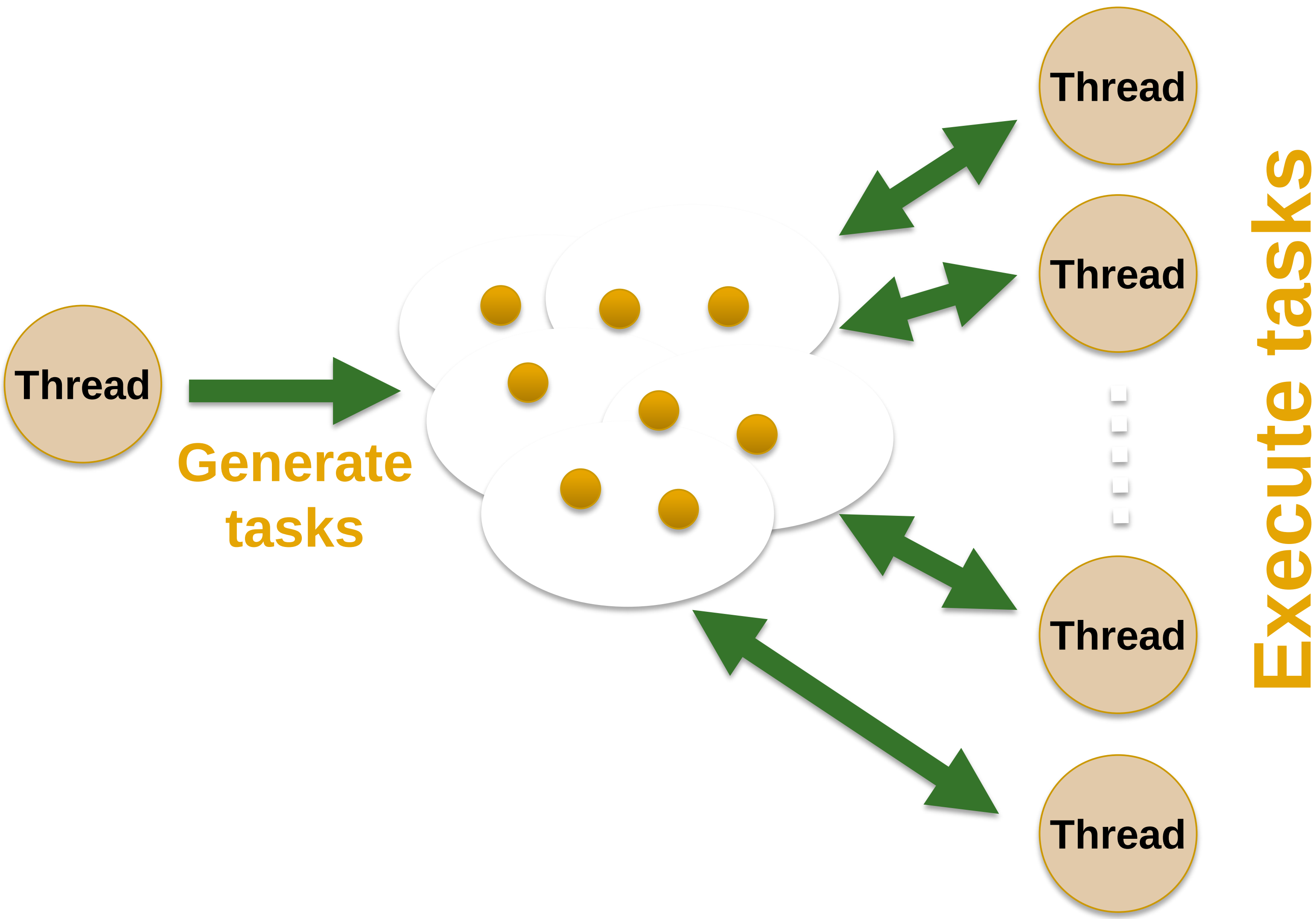
Agenda

- Background
- Parallel Regions
- Sharing Work
- Handling Data
- Synchronization
- **Tasks**

Tasking In OpenMP

- Tasking was introduced in OpenMP 3.0
- Until then it was impossible to efficiently implement certain types of parallelism
  - Recursive algorithms
  - Linked lists, ...
- The initial functionality was very simple by design
  - The idea was (and still) is to augment tasking as we collectively gain more insight and experience

The Tasking Concept In OpenMP



The Tasking Construct

Define a task:

```
#pragma omp task
```

```
!$omp task
```

- A **task** is a specific instance of executable code and its data environment
- A task is generated when a thread encounters a task/taskloop construct or a parallel construct. Comprised of a task region and data environment.
- A task region consists of all code encountered during the execution of a task.
- The data environment consists of all the variables associated with the execution of a given task. It is constructed from the data environment of the generating task at the time the task is generated.

Tasking - Who Does What And When ?

- Assumption: all tasks can execute independently
- When any thread encounters a **task/taskloop construct**, one or more new tasks generated
  - Tasks can be nested (but not for the faint of heart)
- Execution of a generated task is carried out by one of the threads in the current team
  - This is subject to the thread's availability and thus could be immediate or deferred until later
- Completion of the task can be guaranteed using a **task synchronization** construct
  - a **taskwait** or a **barrier** construct

Tasking Example

```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
      {
         printf(“A “);
         #pragma omp task
          {printf("race ");}
         #pragma omp task
          {printf("car ");}
        printf(“is fun to watch “);
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Tasking Example

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch car race
$
```

Tasking Example

```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
      {
         printf(“A “);
         #pragma omp task
           {printf("car ");}
         #pragma omp task
           {printf("race ");}
         #pragma omp taskwait
         printf(“is fun to watch “);
      }
    } // End of parallel region

    printf("\n");return(0);
}
```

What will this program print using 2 threads ?

Tasking Example

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
$
A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
$
```

Tasks are executed first now

Task Completion

Explicit wait on the completion of child tasks:

```
#pragma omp taskwait
```

```
!$omp taskwait
```

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

Does not include descendents of child tasks

Clauses On The Task Directive

if(*scalar-expression*)	if false, create an undeferred task: encountering thread must suspend the encountering task region, immediately execute the current task region until it is completed. Helps avoid small tasks.
untied	any thread can resume after suspension
default(shared | none)	
private(*list*)	
firstprivate(*list*)	
shared(*list*)	
final(*scalar-expression*)	if true, the generated task is a final task
mergeable	if the task is an undeferred task or an included task, the implementation may generate a merged task

Default Data-Sharing Attributes for Tasks

```c
int a;
void foo()
{
    int b, c;

    #pragma omp parallel private(b)
    {
        int d; static int e;
        #pragma omp task
        {
            int f;
            // Scope of a: shared
            // Scope of b: firstprivate
            // Scope of c: shared
            // Scope of d: firstprivate
            // Scope of e: shared
            // Scope of f: private
        }
    }
}
```

Variables that are shared in enclosing context (a, c, e) are shared in task.

Variables that are not shared in enclosing context (b, d) are firstprivate in task.

Local variables (f) are private to task.

Task Scheduling Points In OpenMP

- Whenever a thread reaches a **task scheduling point**, it may suspend the current task in order to execute a different task bound to the current team
- Task scheduling points are implied at:
  - The point immediately following the generation of an explicit task
  - After the last instruction of a task region
  - In taskwait and taskyield regions
  - In implicit and explicit barrier regions
- The implementation may insert task scheduling points in untied tasks
- The user may define additional scheduling points

Tied and Untied Tasks

- Default: Tasks are tied to the thread that first executes them
  - Tasks created with the untied clause are never tied to a thread
  - Take care with some constructs, e.g. thread ids, locks
- This affects execution behavior after a *task switch* at a task scheduling point
- If the suspended task region is for a tied task, the initially assigned thread resumes execution of the suspended task subsequently
  - If it is untied, any thread may resume its execution

Taskyield

```
#pragma omp taskyield
```

```
!$omp taskyield
```

- The taskyield directive specifies that the current task can be suspended in favor or execution of a different task
- Hint to the runtime

```
#include <omp.h>
void something_useful();
void something_critical();
void foo(omp_lock_t * lock, int n)
{
   for(int i = 0; i < n; i++)
   #pragma omp task
   {
      something_useful();
      while( !omp_test_lock(lock) ) {
      #pragma omp taskyield
      }
      something_critical();
      omp_unset_lock(lock);
   }
}
```

The waiting task may be suspended here so that the executing thread can perform other work.

Final clause

```
#pragma omp task final(expr)
```

```
!$omp task final(expr)
```

- For recursive problems that perform task decomposition
  - stop task creation at a certain depth exposes
  - enough parallelism while reducing overhead.
- Warning: Merging the data environment may have side-effects

```
void foo(bool arg)
{
   int i = 3;
   #pragma omp task final(arg) firstprivate(i)
         i++;
   printf(“%d\n”, i); // will print 3 or 4 depending on arg
}
```

Loop parallelization with tasks

- Recall: loop (for/do) construct distributes loop iterations among *encountering* threads.
- **taskloop** construct:
  - distributes loop iterations among tasks generated by the construct
  - implicit **taskgroup** region surrounds loop by default
  - tasks are scheduled onto threads like any other task

The taskloop construct

```c
void long_running(void);
void loop_body(int i, int j);

void parallel_work(void)
{
   int i, j;
   // generates 1 task
#pragma omp task
   long_running();

   // generates 20 tasks
#pragma omp taskloop private(j) num_tasks(20) nogroup
   for (i = 0; i < 10000; i++) {
      for (j = 0; j < i; j++) {
         loop_body(i, j);
      }
   }
}
```


Thanks