LSU CENTER FOR COMPUTATION & TECHNOLOGY

LSU INFORMATION TECHNOLOGY SERVICES

Using OpenMP

Shaohao Chen

High performance computing @ Louisiana State University

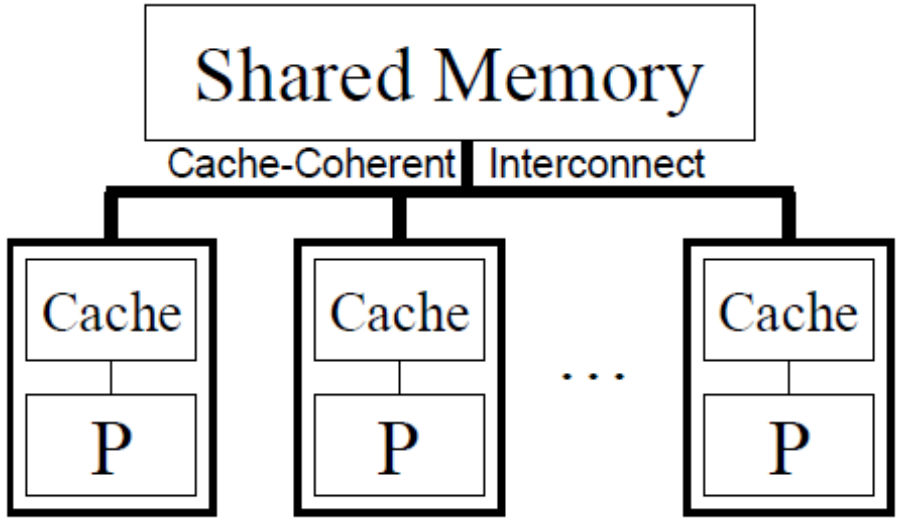
Outline

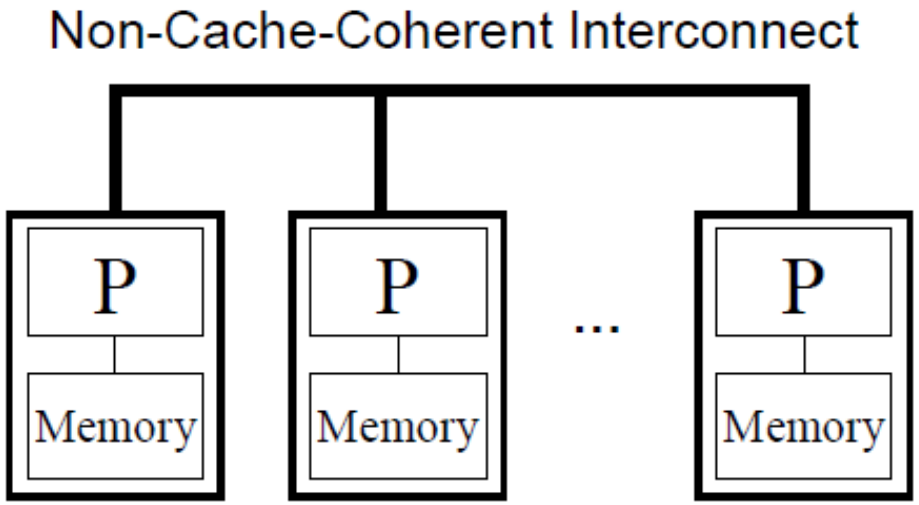
- Introduction to OpenMP
- OpenMP Language Features
  - Parallel constructs
  - Work-sharing constructs
  - Synchronization constructs
  - Basic and advanced clauses
- Optimization for performance

I. Introduction to OpenMP

- OpenMP (Open Multi-Processing) is an API (application programming interface) that supports multi-platform shared memory multiprocessing programming.
- Supporting languages: C, C++, and Fortran
- Consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
- For most processor architectures and operating systems: Linux, Solaris, AIX, HP-UX, Mac OS X, and Windows platforms.
- The latest version is OpenMP 4.0, which supports accelerators. Most features mentioned in this training are within OpenMP 3.0 .
- Official website: http://openmp.org/wp/

Distributed and shared memory

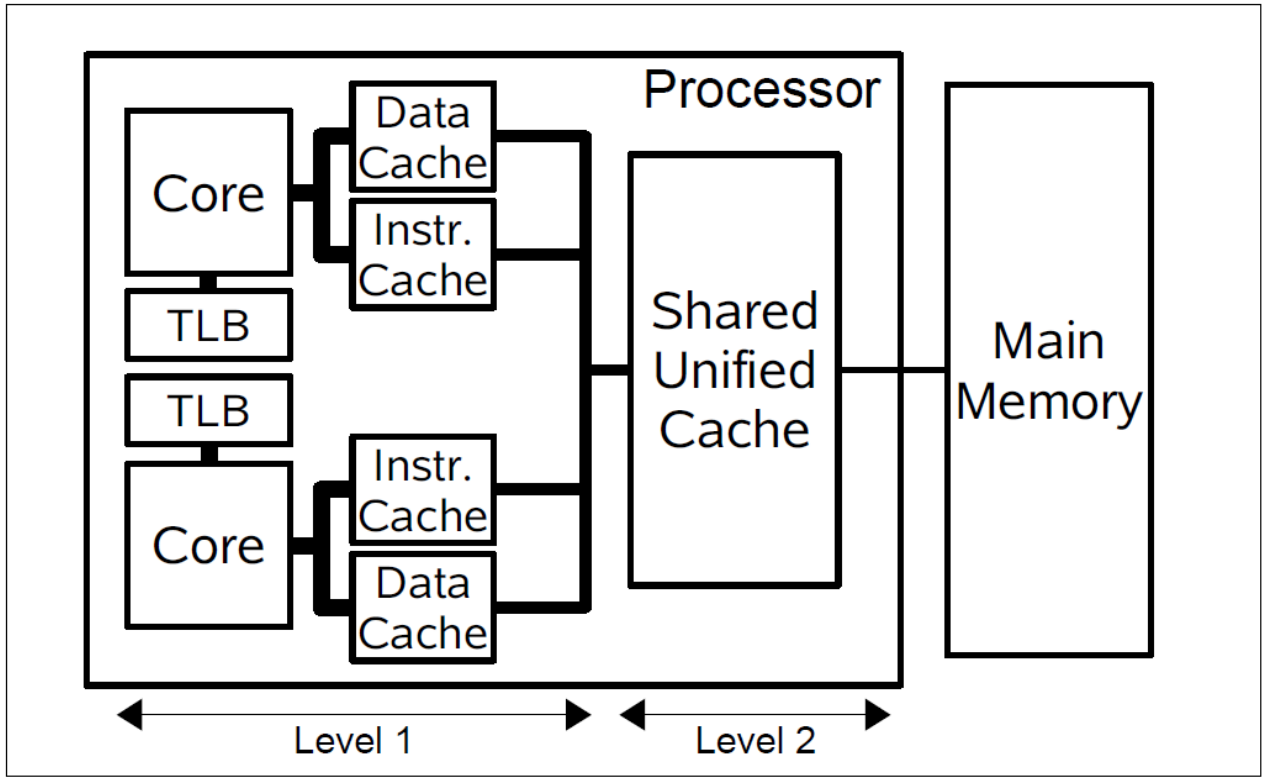




- Shared memory
- A single multicore compute node
- Open Multi-processing (OpenMP)

- Distributed memory system
- Mutli compute nodes
- Message Passing Interface (MPI)

Multicore processor with shared memory

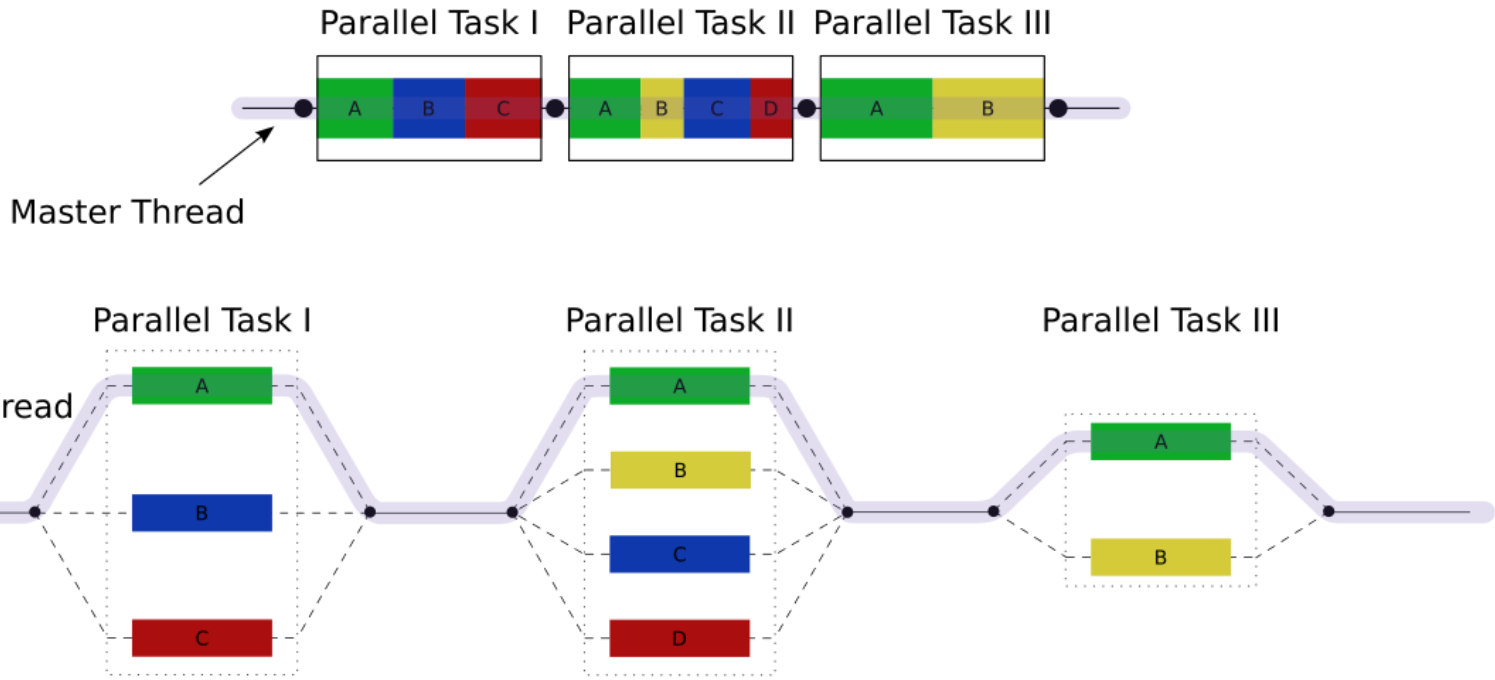


- OpenMP can be applied to a multicore processor with shared memory.
- Works are spread to multi threads and each thread is assigned to one core.
- Data is copied into cache from main memory.

Parallelism of OpenMP

- Multithreading: a master thread forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors (or cores).

- Fork-join model:



The first OpenMP program: Hello world!

- Hello world in C language

```
#include <omp.h>
int main() {
int id;
#pragma omp parallel private(id)
{
   id = omp_get_thread_num();
   if (id%2==1)
     printf("Hello world from thread %d, I am odd\n", id);
   else
     printf("Hello world from thread %d, I am even\n", id);
  }
}
```

- Hello world in Fortran language

```
program hello
   use omp_lib
   implicit none
   integer i
   !$omp parallel private(i)
   i = omp_get_thread_num()
   if (mod(i,2).eq.1) then
      print *,'Hello from thread',i,', I am odd!'
   else
      print *,'Hello from thread',i,', I am even!'
   endif
   !$omp end parallel
end program hello
```

OpenMP directive syntax

- In C/C++ programs

```
#pragma omp directive-name [clause[[,] clause]. . . ]
```

- In Fortran programs

```
!$omp directive-name [clause[[,] clause]. . . ]
```

- Directive-name is a specific keyword, for example parallel, that defines and controls the action(s) taken.
- Clauses, for example private, can be used to further specify the behavior.

Compile and run OpenMP programs

Compile C/C++/Fortran codes

```
> icc/icpc/ifort  -openmp  name.c/name.f90  -o  name
> gcc/g++/gfortran  -fopenmp  name.c/name.f90  -o  name
> pgcc/pgc++/pgf90  -mp  name.c/name.f90  -o  name
```

Run OpenMP programs

```
> export OMP_NUM_THREADS=20        # set number of threads
> ./name
> time ./name                      # run and measure the time.
```

II. OpenMP language features

- Parallel Construct
- Work-Sharing Constructs

  Loop Construct

  Sections Construct

  Single Construct

  Workshare Construct (Fortran only)
- Basic clauses

  shared, private, lastprivate, firstprivate,

  default, nowait, schedule

- Synchronization constructs

  Barrier Construct

  Master Construct

  Critical Construct

  Atomic Construct
- Advanced clauses:

  reduction, if, num_thread
- Nested parallelism

- Construct : An OpenMP executable directive and the associated statement, loop, or structured block, not including the code in any called routines.

Parallel construct

- Syntax in C/C++ programs

```
#pragma omp parallel [clause[[,] clause]. . . ]
    …… code block ……
```

- Syntax in Fortran programs

```
!$omp parallel [clause[[,] clause]. . . ]
    …… code block ……
!$omp end parallel
```

- Parallel construct is used to specify the computations that should be executed in parallel.
- A team of threads is created to execute the associated parallel region.
- The work of the region is replicated for every thread.
- At the end of a parallel region, there is an implied barrier that forces all threads to wait until the work inside the region has been completed.

- Clauses supported by the parallel construct

• **if***(scalar-expression)*	(C/C++)
• **if***(scalar-logical-expression)*	(Fortran)
• **num_threads***(integer-expression)*	(C/C++)
• **num_threads***(scalar-integer-expression)*	(Fortran)
• **private***(list)*	
• **firstprivate***(list)*	
• **shared***(list)*	
• **default(none|shared)**	(C/C++)
• **default(none|shared|private)**	(Fortran)
• **copyin***(list)*	
• **reduction***(operator:list)*	(C/C++)
• **reduction***({operator|intrinsic procedure name}:list)*	(Fortran)

Work-sharing constructs

Functionality	Syntax in C/C++	Syntax in Fortran
Distribute iterations	**#pragma omp for**	**!$omp do**
Distribute independent works	**#pragma omp sections**	**!$omp sections**
Use only one thread	**#pragma omp single**	**!$omp single**
Parallelize array syntax	**N/A**	**!$omp workshare**

- Many applications can be parallelized by using just a parallel region and one or more of work-sharing constructs, possibly with clauses.

- The parallel and work-sharing (except single) constructs can be combined.
- Following is the syntax for combined parallel and work-sharing constructs,

Combine parallel construct with …	Syntax in C/C++	Syntax in Fortran
Loop construct	**#pragma omp parallel for**	**!$omp parallel do**
Sections construct	**#pragma omp parallel sections**	**!$omp parallel sections**
Workshare construct	**N/A**	**!$omp parallel workshare**

Loop construct

- The loop construct causes the iterations of the loop immediately following it to be executed in parallel.

- Syntax in C/C++ programs

```
#pragma omp for [clause[[,] clause]. . . ]
    …… for loop ……
```

- Syntax in Fortran programs

```
!$omp do [clause[[,] clause]. . . ]
    …… do loop ……
[!$omp end do]
```

- The terminating **!$omp end do** directive in Fortran is optional but recommended.

- Distribute iteration in a parallel region

```
#pragma omp parallel for shared(n,a) private(i)
{
   for (i=0; i<n; i++)
         a[i]=i+1;
}     /*-- End of parallel region --*/
```

- shared clause: All threads can read from and write to the variable.
- private clause: Each thread has a local copy of the variable.
- The maximum iteration number n is shared, while the iteration number i is private.
- Each thread executes a subset of the total iteration space i = 0, . . . , n − 1
- The mapping between iterations and threads can be controlled by the schedule clause.

- Two work-sharing loops in one parallel region

```
#pragma omp parallel shared(n,a,b) private(i)
{
   #pragma omp for
   for (i=0; i<n; i++)   a[i] = i;
   //  there is an implied barrier

   #pragma omp for
   for (i=0; i<n; i++)   b[i] = 2 * a[i];
}     /*-- End of parallel region --*/
```

- The distribution of iterations to threads could be different for the two loops.
- The implied barrier at the end of the first loop ensures that all the values of a[i] are updated before they are used in the second loop.

Sections construct

- Syntax in C/C++ programs

```
#pragma omp sections [clause[[,] clause]. . . ]
{
[#pragma omp section ]
…… code block 1 ……
[#pragma omp section
…… code block 2 ……  ]
. . .
}
```

- Syntax in Fortran programs

```
!$omp sections [clause[[,] clause]. . . ]
[!$omp section ]
…… code block 1 ……
[!$omp section
…… code block 2 ……  ]
. . .
!$omp end sections
```

- The work in each section must be independent.
- Each section is distributed to one thread.

- Example of parallel sections

```
#pragma omp parallel sections
{
    #pragma omp section
    (void) funcA();
     #pragma omp section
    (void) funcB();
}  /*-- End of parallel region --*/
```

- Although the sections construct can be generally used to get threads to perform different tasks independently, its most common use is probably to execute function or subroutine calls in parallel.
- There is a load-balancing problem, if the works in different sections are not equal.

Single construct

- Syntax in C/C++ programs

```
#pragma omp single [clause[[,] clause]. . .
…… code block  ……
```

- Syntax in Fortran programs

```
!$omp single [clause[[,] clause]. . . ]
…… code block  ……
!$omp end single
```

- The code block following the single construct is executed by one thread only.
- The executing thread could be any thread (not necessary the master one).
- The other threads wait at a barrier until the executing thread has completed.

- Example of the single construct

```
#pragma omp parallel shared(a,b) private(i)
{
  #pragma omp single
  {
     a = 10;
  }
  /* A barrier is automatically inserted here */
  #pragma omp for
  for (i=0; i<n; i++)
     b[i] = a;

} /*-- End of parallel region --*/
```

- Only one thread initializes the shared variable a.
- If the single construct is omitted here, multiple threads could assign the value to a at the same time, potentially resulting in a memory problem.
- The implicit barrier at the end of the single construct ensures that the correct value is assigned to the variable a before it is used by all threads.

Workshare construct

- Workshare construct is only available for Fortran.

- Syntax in Fortran programs

```
!$omp workshare [clause[[,] clause]. . . ]
…… code block ……
!$omp end workshare
```

- Units of works within the block are executed in parallel in a manner that respects the semantics of Fortran array operations.
- For example, if the workshare directive is applied to an array assignment statement, the assignment of each element is a unit of work.

- Example of workshare construct

```
!$OMP PARALLEL SHARED(n,a,b,c)
!$OMP WORKSHARE
    b(1:n) = b(1:n) + 1
    c(1:n) = c(1:n) + 2
    a(1:n) = b(1:n) + c(1:n)
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

- These array operations are parallelized.
- There is no control over the assignment of array updates to the threads.
- The OpenMP compiler must generate code such that the updates of b and c have completed before a is computed.

Lastprivate clause

- private clause: The values of data can no longer be accessed after the region terminates.
- lastprivate clause: The sequentially last value is accessible outside the region.
- For loop construct, “last” means the iteration of the loop that would be last in a sequential execution.
- For sections construct, “last” means the lexically last sections construct.
- Lastprivate clause is not available for parallel construct.

```
#pragma omp parallel for private(i) lastprivate(a)
for (i=0; i<n; i++) {
      a = i+1;
      printf("Thread %d has a value of a = %d for i = %d\n", omp_get_thread_num(),a,i);
}    /*-- End of parallel for --*/
printf(“After parallel for: i = %d , a = %d\n", i, a);
```

- Alternative code with shared clause

```
#pragma omp parallel for private(i) private(a) shared(a_shared)
for (i=0; i<n; i++) {
   a = i+1;
   if ( i == n-1 ) a_shared = a;
}   /*-- End of parallel for --*/
```

- All behavior of the lastprivate clause can be reproduced by the shared clause, but the lastprivate clause is more recommended.
- A performance penalty is likely to be associated with the use of lastprivate, because the OpenMP library needs to keep track of which thread executes the last iteration.

Firstprivate clause

- private clause: Preinitialized value of variables are not passed to the parallel region.
- firstprivate clause: Each thread has a preinitialized copy of the variable. This variable is still private, so threads can update it individually.
- Firstprivate clause is available for parallel, loop, sections and single constructs.

```
int i, vtest=10, n=20;
#pragma omp parallel for private(i) firstprivate(vtest) shared(n)
for(i=0; i<n; i++) {
        printf("thread %d: initial value = %d\n", omp_get_thread_num(), vtest);
        vtest=i;
}
printf("value after loop = %d\n", vtest);
```

Default clause

- The default clause is used to give variables a default data-sharing attribute.
- It is applicable to the parallel construct only.
- Syntax in C programs

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

- Syntax in Fortran programs

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

- An example: declares all variables to be shared, with the some exceptions.

```
#pragma omp for default(shared) private(a,b,c)
```

- If default(none) is specified, the programmer is forced to specify a data-sharing attribute for each variable in the construct.

Nowait clause

- If the nowait clause is added to a construct, the implicit barrier at the end of the associated construct will be suppressed. When a thread is finished with the work associated with the parallelized for loop, it continues and no longer waits for the other threads to finish.
- Note, however, that the barrier at the end of a parallel region cannot be suppressed.

- An example for C program

```
#pragma omp for nowait
for (i=0; i<n; i++)
{
.....
} // no barrier here
```

- An example for Fortran program

```
!$OMP DO
.....
!$OMP END DO NOWAIT   ! no barrier here
```

Schedule clause

- Specifies how iterations of the loop are assigned to the threads in the team.
- Supported on the loop construct only.
- The iteration space is divided into chunks. Chunk represents the granularity of workload distribution, a contiguous nonempty subset of the iteration space.

- Syntax **schedule(kind [,chunk_size])**

- The static schedule works best for regular workloads and is the default on many OpenMP compilers.
- The dynamic and guided schedules are useful for handling poorly balanced and unpredictable workloads.
- There is a performance penalty for using dynamic and guided schedules.

- Schedule kind

kind	description
static	The chunks are assigned to the threads statically in a round-robin manner, in the order of the thread number. If *chunk_size* is not specified, the chunk size is approximately equal to the total number of iteration divided by the number of threads.
dynamic	The chunks are assigned to threads as the threads request them. The last chunk may have fewer iterations than chunk size. If *chunk_size* is not specified, it defaults to 1.
guided	The chunks are assigned to threads as the threads request them. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations, divided by the number of threads, decreasing to 1. For a *chunk_size* of "k" (k > 1), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations (with a possible exception for the last chunk to be assigned, which may have fewer than k iterations). When no *chunk_size* is specified, it defaults to 1.
runtime	The schedule and (optional) chunk size are set through the OMP_SCHEDULE environment variable.

- Example of schedule clause:

The workload in the inner loop depends on the value of the outer loop iteration variable i. Therefore, the workload is not balanced, and the static schedule is probably not the best choice. Dynamic or guided schedules are required.

```
#pragma omp parallel for default(none) schedule(runtime) private(i,j) shared(n)
for (i=0; i<n; i++)
{
   printf("Iteration %d executed by thread %d\n", i, omp_get_thread_num());
   for (j=0; j<i; j++)
        system("sleep 1");
}
```

Barrier construct

- A barrier is a point in the execution of a program where threads wait for each other: no thread in the team of threads it applies to may proceed beyond a barrier until all threads in the team have reached that point.

- Syntax in C/C++ programs

```
#pragma omp barrier
```

- Syntax in Fortran programs

```
!$omp barrier
```

Two important restrictions apply to the barrier construct:

- Each barrier must be encountered by all threads in a team, or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in the team.

- Example of barrier construct:

A thread waits at the barrier until the last thread in the team arrives.

```
#pragma omp parallel private(TID)
{
    TID = omp_get_thread_num();
    if (TID < omp_get_num_threads()/2 )  system("sleep 3");
   bt1 =  time(NULL);
   printf("Thread %d before barrier at %s \n", omp_get_thread_num(), ctime(&t1) );
    #pragma omp barrier
    t2 =  time(NULL);
    printf("Thread %d after barrier at %s \n", omp_get_thread_num(), ctime(&t2) );
} /*-- End of parallel region --*/
```

- Illegal use of the barrier
- The barrier is not encountered by all threads in the team, and therefore this is not illegal.

```
#pragma omp parallel
{
   if ( omp_get_thread_num() == 0 ){
      .....
      #pragma omp barrier  // Correction: the barrier should be out of the if-else region
   }
   else{
      .....
      #pragma omp barrier
   }
}  /*-- End of parallel region --*/
```

- Also, a barrier should not be in a work-sharing construct, a critical section, or a master construct.

- A dead lock situation

```
work1(){
   /*-- Some work performed here --*/
   #pragma omp barrier   // Correction: remove this barrier
}
work2(){
   /*-- Some work performed here --*/
}

main(){
   #pragma omp parallel sections
   {
      #pragma omp section
      work1();
      #pragma omp section
      work2();
   }   // An implicit barrier
}
```

- If executed by two threads, this program never finishes.
- Thread1 executing work1 waits forever in the explicit barrier, which thread2 will never encounter.
- Thread2 executing work2 waits forever in the implicit barrier at the end of the parallel sections construct, which thread1 will never encounter.
- Note: Do not insert a barrier that is not encountered by all threads of the same team.

Master construct

- The master construct defines a block of code that is guaranteed to be executed by the master thread only.
- It does not have an implied barrier on entry or exit. In the cases where a barrier is not required, the master construct may be preferable compared to the single construct.

- Syntax in C/C++ programs

```
#pragma omp master
…… code block …..
```

- Syntax in Fortran programs

```
!$omp master
…… code block …..
!$omp end master
```

- The master construct is often used (in combination with barrier construct) to initialize data.

- Incorrect use of master construct

- This code fragment implicitly assumes that variable Xinit is available to all threads after it is initialized by the master thread. This is incorrect. The master thread might not have executed the assignment when another thread reaches it.

```
int Xinit, Xlocal;
#pragma omp parallel shared(Xinit) private(Xlocal)
{
    #pragma omp master // correct version 1: use single construct instead, #pragma omp single
    {
        Xinit = 10;
    }
    // correct version 2: insert a barrier here, #pragma omp barrier
    Xlocal = Xinit;    /*-- Xinit might not be available for other threads yet --*/
}    /*-- End of parallel region --*/
```

Critical construct

- The critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously.
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.

- Syntax in C/C++ programs

```
#pragma omp critical [(name)]
…… code block …..
```

- Syntax in Fortran programs

```
!$omp critical [(name)]
…… code block …..
!$omp end critical [(name)]
```

- The code block is executed by all threads, but only one at a time executes the block.

- Example 1 of critical construct: Avoiding garbled output

A critical region helps to avoid intermingled output when multiple threads print from within a parallel region.

```
#pragma omp parallel private(TID)
{
  TID = omp_get_thread_num();
  #pragma omp critical (print_tid)
  {
      printf("Thread %d : Hello, ",TID);
      printf(“world!\n");
  }
} /*-- End of parallel region --*/
```

Race condition

- Race conditions arise when the result depends on the sequence or timing of processes or threads, for example, when multithreads read or write the same shared data simultaneously.
- Example: two threads each want to increment the value of a shared integer variable by one.

Correct sequence

Thread 1	Thread 2		value
			0
read value		←	0
Increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Incorrect sequence

Thread 1	Thread 2		value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

- Example of data racing: sums up elements of a vector

Multithreads can read and write the shared data sum simultaneously.

A data race condition arises!

If a thread reads sum before sum is updated by another thread, the final result of sum is wrong!

```
sum = 0;
 #pragma omp parallel for shared(sum,a,n) private(i)
  for (i=0; i<n; i++)
  {
     sum = sum + a[i];
  }  /*-- End of parallel for --*/
printf("Value of sum after parallel region: %f\n",sum);
```

- A partially parallel scheme to avoid data racing

Step 1: Calculate local sums in parallel

Thread 1	Thread 2	……	Thread m
a_0	a_m	……	a_{n-m-1}
+	+		+
a_1	a_{m+1}	……	a_{n-m}
+	+		+
⋮	⋮		⋮
+	+		+
a_{m-1}	a_{2m-1}	……	a_n
=	=		=
LS_1	LS_2	……	LS_m

m: number of threads

n: array length

LS: local sum

Step 2: Update total sum sequentially

Thread 1	Thread 2	……	Thread m
Read initial S			
$S = S + LS_1$			
Write S			
	Read S		
	$S = S + LS_2$		
	Write S		
		……	
			Read S
			$S = S + LS_m$
			Write S

m: number of threads

LS: local sum

S: total sum

- Example 2 of critical construct: sums up the elements of a vector

The critical region is needed to avoid a data race condition when updating variable sum.

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(sumLocal)
{
    sumLocal = 0;
    #pragma omp for
    for (i=0; i<n; i++)  sumLocal += a[i];
    #pragma omp critical (update_sum)
    {
       sum += sumLocal;
       printf("TID=%d: sumLocal=%d sum = %d\n", omp_get_thread_num(), sumLocal, sum);
    }
}  /*-- End of parallel region --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Atomic construct

- The atomic construct also enables multiple threads to update shared data without interference.
- It is applied only to the (single) assignment statement that immediately follows it.
- If a thread is atomically updating a value, then no other thread may do so simultaneously.

	C/C++ programs	Fortran programs
• Syntax	**#pragma omp atomic** …… a single statement …..	**!$omp atomic** …… a single statement ….. **!$omp end atomic**
• Supported operators	+, *, -, /, &, ^, |, <<, >>.	+, *, -, /, .AND., .OR., .EQV., .NEQV. .

- Example 1a of atomic construct: sums up the elements of a vector (fast version)

The atomic construct ensures that no updates are lost when multiple threads update the variable sum.

Atomic construct can be an alternative to the critical construct in this case.

```
sum = 0;
#pragma omp parallel shared(n,a,sum) private(sumLocal)
{
    sumLocal = 0;
    #pragma omp for
    for (i=0; i<n; i++)  sumLocal += a[i];
    #pragma omp atomic
    sum += sumLocal;
}  /*-- End of parallel region --*/
printf("Value of sum after parallel region: %d\n",sum);
```

- Example 1b of atomic construct: sums up the elements of a vector (slow version)

- The atomic construct avoids the data racing condition. Therefore this code gives a correct result.
- But the additions are performed sequentially and there is additional performance penalty for atomic.
- This code is even slower than a normal serial code!

```
sum = 0;
 #pragma omp parallel for shared(n,a,sum) private(i) // Optimization: use reduction instead of atomic
 for (i=0; i<n; i++)
 {
      #pragma omp atomic
       sum += a[i];
 }  /*-- End of parallel for --*/
printf("Value of sum after parallel region: %d\n",sum);
```

- Example 2 of atomic construct: sums up the values of functions

The atomic construct does not prevent multiple threads from executing the function *bigfunc* parallelly.

It is only the update to the memory location of the variable sum that will occur atomically.

```
sum = 0;
 #pragma omp parallel for shared(n,a,sum) private(i)
 for (i=0; i<n; i++)
 {
      #pragma omp atomic
       sum = sum + bigfunc();
 }  /*-- End of parallel for --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Reduction clause

- There is a much easier way to implement the summation operation using reduction clause.

```
#pragma omp parallel for default(none) shared(n,a) private(i) reduction(+:sum)
 for (i=0; i<n; i++)
       sum += a[i];
 /*-- End of parallel reduction --*/
```

- An OpenMP compiler will generate a roughly equivalent machine code for the two cases: using critical construct and using reduction clause, meaning that their performance is almost the same.
- The result sum will be shared and it is not necessary to specify it explicitly as “shared”.
- The order in which thread-specific values are combined is unspecified. Therefore, where floating-point data are concerned, there may be numerical differences between the results of a sequential and parallel run, or even of two parallel runs using the same number of threads.

- Operators and statements supported by the reduction clause

	C/C++	Fortran
Typical statements	x = x *op* expr x *binop* = expr x = expr *op* x (except for subtraction) x++ ++x x-- --x	x = x *op* expr x = expr *op* x (except for subtraction) x = *intrinsic* (x, expr_list) x = *intrinsic* (expr_list, x)
op could be	+, *, -, &, ^, |, &&, or ||	+, *, -, .and., .or., .eqv., or .neqv.
binop could be	+, *, -, &, ^, or |	N/A
Intrinsic function could be	N/A	max, min, iand, ior, ieor

If clause

- The if clause is supported on the parallel construct only and is used to specify conditional execution.

```
#pragma omp parallel if (n > 5) default(none)  private(TID) shared(n)
{
   TID = omp_get_thread_num();
   #pragma omp single
   {
       printf("Number of threads in parallel region: %d\n", omp_get_num_threads());
   }
   printf("Print statement executed by thread %d\n",TID);
}  /*-- End of parallel region --*/
```

- it is sometimes necessary to test whether there is enough work in the region to warrant its parallelization.

Num_threads clause

- The num_threads clause is supported on the parallel construct only and can be used to specify how many threads should be in the team executing the parallel region

```
omp_set_num_threads(4);
#pragma omp parallel if (n > 5) num_threads(n) default(none) shared(n)
{
   #pragma omp single
   {
      printf("Number of threads in parallel region: %d\n", omp_get_num_threads());
   }
   printf("Print statement executed by thread %d\n", omp_get_thread_num());
} /*-- End of parallel region --*/
```

Nested parallelism

- If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team and becomes the master of that new team.

```
#pragma omp parallel private(TID)
{
   TID = omp_get_thread_num();
   #pragma omp parallel num_threads(2) firstprivate(TID)
   {
      printf("Outer thread number: %d. Inner thread number: %d.\n", TID,omp_get_thread_num());
   }   /*-- End of inner parallel region --*/
}   /*-- End of outer parallel region --*/
```

- The function omp_get_thread_num() returns the thread number of the current parallel region.
- The thread number of the first lever can be passed on to the second level by fristprivate clause.

III. Optimization for performance

- It may be possible to quickly write a correctly functioning OpenMP program, but not so easy to create a program that provides the desired level of performance.
- The most intuitive implementation is often not the best one when it comes to performance, but the parallel inefficiency is not directly visible simply by inspecting the source.
- Programmers have developed some rules of thumb on how to write efficient code.

❑ Optimize serial code

- Memory access patterns: rowwise for C and columnwise for Fortran.
- Lower data precision if possible
- Common subexpression elimination
- Loop unrolling and jam
- Loop fusion and fission
- Loop tiling
- If-statement collapse

❑ Cases for optimizing OpenMP parallel codes will be introduced at the following slides.

Case 1: A reduced number of barriers

```
#pragma omp parallel shared(n,a,b,c,d,sum) private(i)
{
   #pragma omp for nowait
   for (i=0; i<n; i++)  a[i] += b[i];

   #pragma omp for nowait
   for (i=0; i<n; i++) c[i] += d[i];
   #pragma omp barrier

   #pragma omp for nowait reduction(+:sum)
   for (i=0; i<n; i++) sum += a[i] + c[i];
}  /*-- End of parallel region --*/
```

- Use the nowait clause where possible, carefully inserting explicit barriers at specific points in the program as needed.
- Here vectors a and c are independently updated. Therefore a thread that has finished its work in the first loop can safely enter the second loop.
- The barrier ensures that a and c have been updated before they are used.

Case 2: Avoid Large Critical Regions

```
#pragma omp parallel shared(a,b) private(c,d)
{
.....
  #pragma omp critical
  {
    a += 2 * c;
    c = d * d;   // Optimization: move this line out of critical region
  }
}  /*-- End of parallel region --*/
```

- The more code contained in the critical region, the greater the likelihood that threads have to wait to enter it, and the longer the potential wait times.
- The first statement is protected by the critical region to avoid a data race of the shared variable a.
- The second statement however involves private data only. There is no data race. It should be removed from the critical region.

Case 3: Maximize Parallel Regions

- Overheads are associated with starting and terminating a parallel region.
- Large parallel regions offer more opportunities for using data in cache and provide a bigger context for other compiler optimizations.
- The code in the right panel is better, because it has fewer implied barriers, and there might be potential for cache data reuse between loops.

```
#pragma omp parallel for
for (.....){
/*-- Work-sharing loop 1 --*/
}
#pragma omp parallel for
for (.....){
/*-- Work-sharing loop 2 --*/
}
```

```
#pragma omp parallel
{
#pragma omp for /*-- Work-sharing loop 1 --*/
{ ..... }

#pragma omp for  /*-- Work-sharing loop 2 --*/
{ ..... }
}
```

Case 4: Avoid Parallel Regions in Inner Loops

- In the left panel, the overheads of the parallel region are incurred n^2 times.
- The code in the right panel is better, because the parallel construct overheads are minimized.

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

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

False sharing

- Cache coherence mechanism: When a cache line is modified by one processor, other caches holding a copy of the same line are notified that the line has been modified elsewhere. At such a point, the copy of the line on other processors is invalidated.
- False sharing: When two or more threads update different data elements in the same cache line simultaneously, they interfere with each other.
- Typically, the computing results in false sharing cases are still correct.
- Note that a modest amount of false sharing does not have a significant impact on performance. However, if some or all of the threads update the same cache line frequently, performance degrades.
- False sharing is likely to significantly impact performance under the following conditions:

  1. Shared data is modified by multiple threads.
  2. The access pattern is such that multiple threads modify the same cache line(s).
  3. These modifications occur in rapid succession.

Case 5: Avoid false sharing (I)

- Example I of false sharing case:

```
#pragma omp parallel for shared(Nthreads,a) schedule(static,1)
for (int i=0; i<Nthreads; i++)   a[i] += i;        // Optimization: use a[i][0] instead of a[i]
```

- Each thread has its own copy of a[i], thus there is no data race and the computing result is correct.
- However, all elements of a accesses to the same cache line, which results in false sharing and thus degrades the performance.

- This case can be optimized by array padding: Accesses to different elements a[i][0] are now separated by a cache line. As a result, the update of an element no longer affects other elements.

Case 6: Avoid false sharing (II)

- Example II of false sharing:

```
#pragma omp parallel shared(a,b)   // Optimization: variable a should be private.
{
   a = b + 1;
   ......
}
```

- Variable b is not modified, thus it does not cause false sharing.
- However, the shared variable a is modified by multi threads, thus it causes false sharing.
- If there are a number of such initializations, they could reduce program performance. In a more efficient implementation, variable a should be declared and used as a private variable instead.

Appendix A: OpenMP built-in functions

- Enable the usage of OpenMP functions:

C/C++ program: include omp.h .

Fortran program: include omp_lib.h or use omp_lib module.

- List of OpenMP functions:

omp_set_num_threads(integer) : set the number of threads

omp_get_num_threads(): returns the number of threads

omp_get_thread_num(): returns the number of the calling thread.

omp_set_dynamic(integer|logical): dynamically adjust the number of threads

omp_get_num_procs(): returns the total number of available processors when it is called.

omp_in_parallel(): returns true if it is called within an active parallel region. Otherwise, it returns false.

Appendix B: OpenMP runtime variables

OMP_NUM_THREADS : the number of threads (=integer)

OMP_SCHEDULE : the schedule type (=kind,chunk . Kind could be static, dynamic or guided)

OMP_DYNAMIC : dynamically adjust the number of threads (=true | =false).

KMP_AFFINITY : for intel compiler, to bind OpenMP threads to physical processing units.

(=compact | =scatter | =balanced).

Example usage: export KMP_AFFINITY= compact,granularity=fine,verbose .

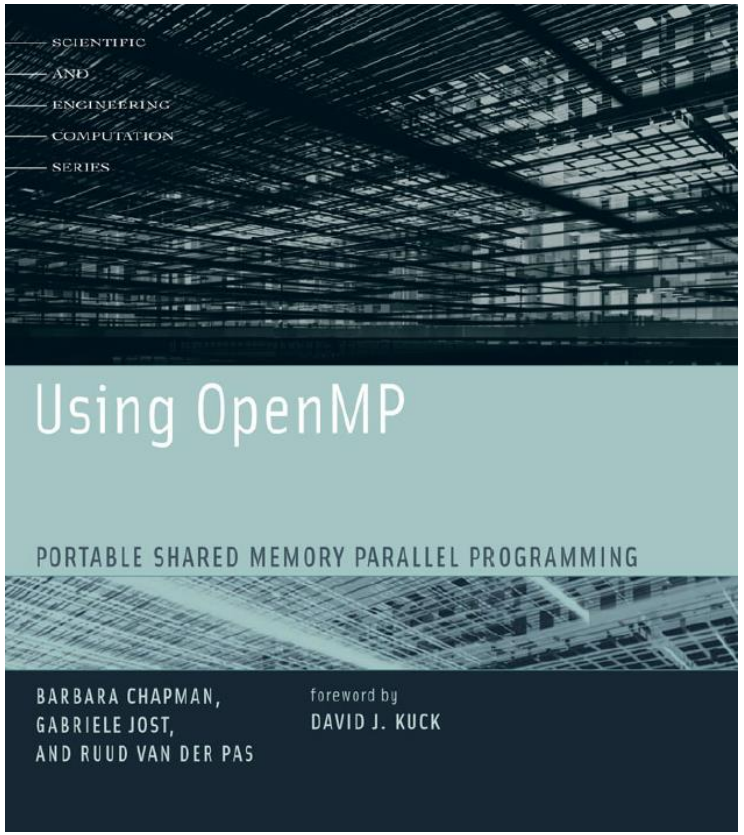
Exercises

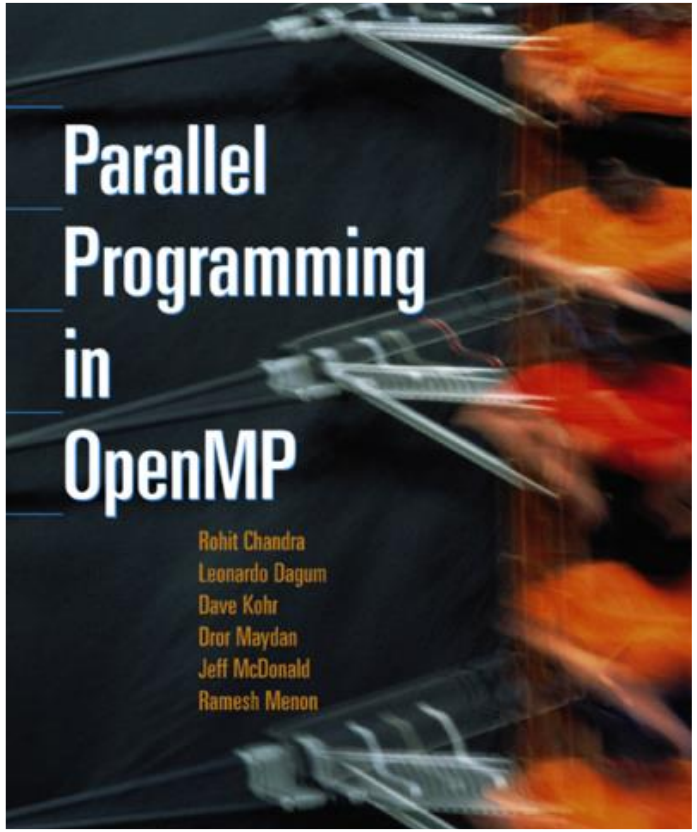
- Use OpenMP to parallelize the following programs.

1. Add a scalar multiple of a real vector to another real vector, s = a*x + y.
2. Multiply two squared matrices, C=A*B.
3. Calculate the value of Pi using numerical integration,

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

References





- Wiki: http://en.wikipedia.org/wiki/OpenMP