Introduction to OpenMP

Shaohao Chen

ORCD at MIT

Outline

- A brief overview of OpenMP
- Constructs and clauses
- Parallel loops
- Data race
- OpenMP in Numpy

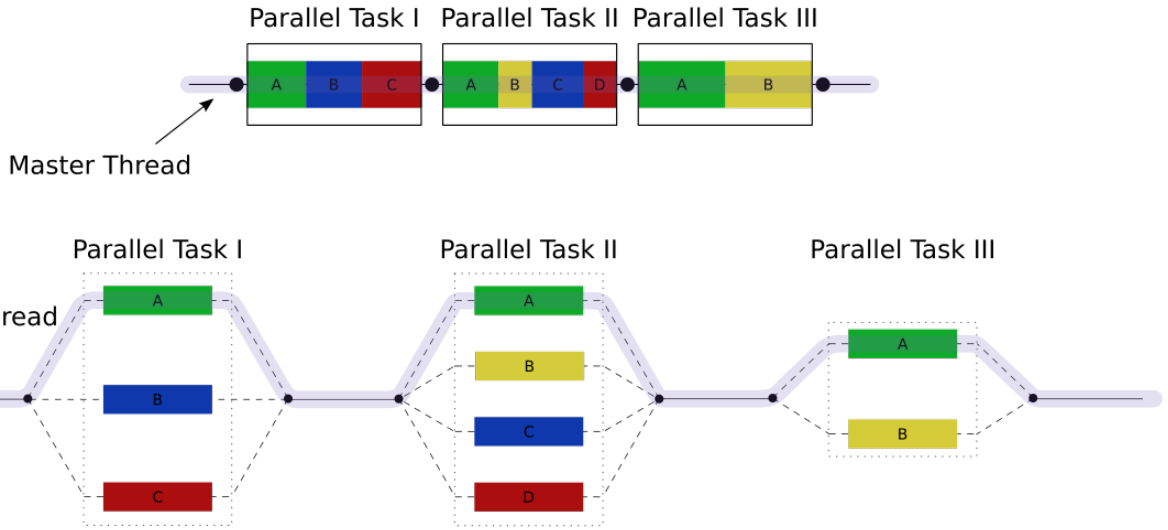
OpenMP

- 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.
- OpenMP supports accelerators.

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:



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

The first OpenMP program: Hello world!

- Hello world in C

```
#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

```
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/Fortran codes

```
gcc -fopenmp  name.c -o  name
gfortran  -fopenmp  name.f90  -o  name
```

Run OpenMP programs

```
export OMP_NUM_THREADS=8       # set number of threads
./name
time ./name                    # run and measure the time.
```

OpenMP programming

- Parallel Construct
- Work-Sharing Constructs
- Clauses
- Data race: atomic, reduction

➢ 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 computation inside the region has been completed.

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 iterations in a parallel region

```
#pragma omp parallel for shared(n,a) private(i)
   for (i=0; i<n; i++)
         a[i] = i + n;
```

- shared clause: All threads can read from and write to a shared variable.
- private clause: Each thread has a local copy of a private 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+1;
   //  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.

Exercise 1

- SAXPY in OpenMP:

The SAXPY program is to add a scalar multiple of a real vector to another real vector: s = a*x + y.

1. Provided a serial SAXPY code, parallelize it using OpenMP directives.
2. Compare the performance between serial and OpenMP codes.

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

- The most common use of the sections construct is probably to execute function or subroutine calls in parallel.

- There is a load-balancing problem, if the amount of work 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.

- An 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.

Data race

- A data race condition arises when multithreads read or write the same shared data simultaneously.
- Example: two threads each increases 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 race: sums up elements of a vector

Different threads read and write the shared data sum simultaneously.

A data race condition arises!

The final result of *sum* could be incorrect!

```
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);
```

Atomic construct

- The atomic construct allows multiple threads to safely update a shared variable.
- The memory update (such as write) in the next instruction will be performed atomically. It does not make the entire statement atomic. Only the memory update is atomic.
- It is applied only to the (single) assignment statement that immediately follows it.

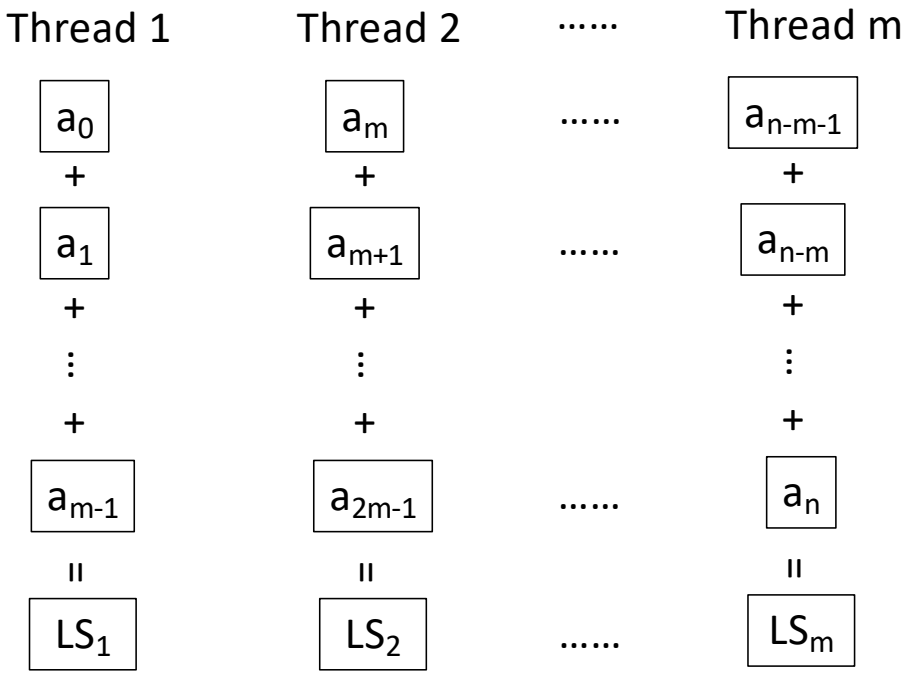
	C/C++ programs	Fortran programs
• Syntax	**#pragma omp atomic** …… a single statement …..	**!$omp atomic** …… a single statement ….. **!$omp end atomic**
• Supported operators	+, *, -, /, &, ^, |, <<, >>.	+, *, -, /, .AND., .OR., .EQV., .NEQV. .

- The first try to solve the data-race problem: use *atomic* (correct but slow)
- The atomic construct avoids the data racing condition. Therefore, the result is correct.
- But all elements are added sequentially, and there is performance penalty for using *atomic*, because the system coordinates all threads.
- This code is even slower than a 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);
```

- A partially parallel scheme to avoid data race

Step 1: Calculate local sums in parallel



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

- The second try to solve the data-race problem: use *atomic* (correct and fast)

- Each thread adds up its local sum.
- The *atomic* is only applied for adding up local sums to obtain the total 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 atomic
    sum += sumLocal;
}  /*-- End of parallel region --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Reduction clause

- The third try to solve the data-race problem: use *reduction* (correct, fast and simple)

```
#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 --*/
```

- The reduction variable is protected to avoid data race.
- The partially parallel scheme is applied behind the scene.
- The reduction variable is shared by default and it is unnecessary to specify it explicitly.

- 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

Exercise 2

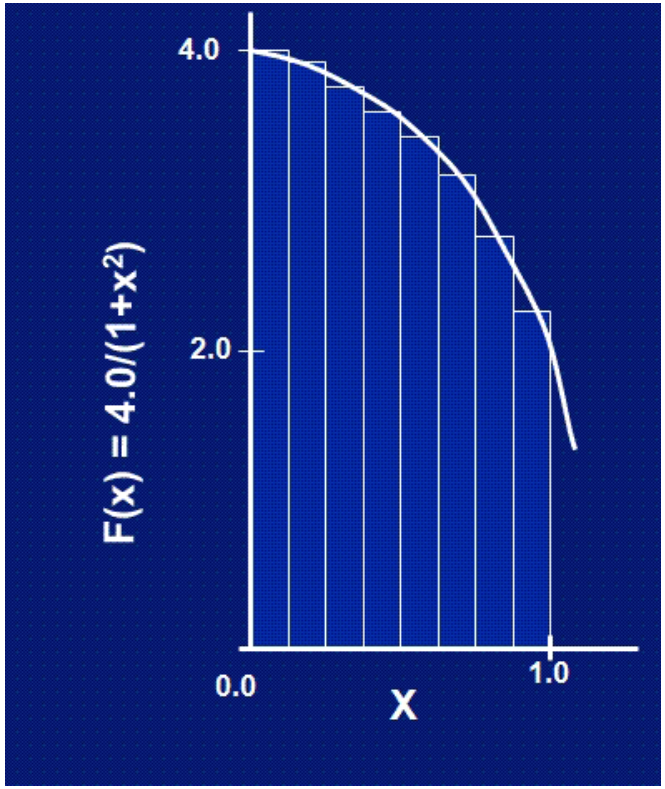
- Compute the value of pi:

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

The integration can be numerically approximated as the sum of a number of rectangles.

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

1. Provided the serial code for computing the value of pi, parallelize it using OpenMP directives.
2. Compare the performance between seral and OpenMP codes.



Using OpenMP in Numpy

❑ OpenMP-enabled Libraries

- OpenBLAS: BLAS = Basic Linear Algebra Subprograms
- MKL: Linear algebra libraries optimized for intel CPUs.

❑ Numpy

- Install Numpy with precompiled OpenBLAS or MKL

```
pip install numpy
```

- Numpy calls OpenMP-enabled routines under the hood.

```
import numpy as np
n = 10000
a = np.random.rand(n,n)
b = np.random.rand(n,n)
c = np.matmul(a, b)
```

- Run a python program with multiple threads

```
export OMP_NUM_THREADS=8
pyton name.py
```

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. False otherwise.

Appendix B: OpenMP runtime variables

OMP_NUM_THREADS : the number of threads (=integer)

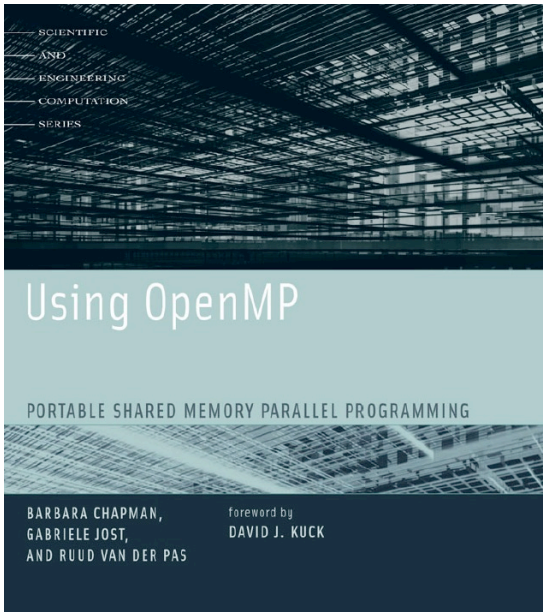
OMP_SCHEDULE : the schedule type (=kind,chunk . Kind could be static, dynamic or guided)

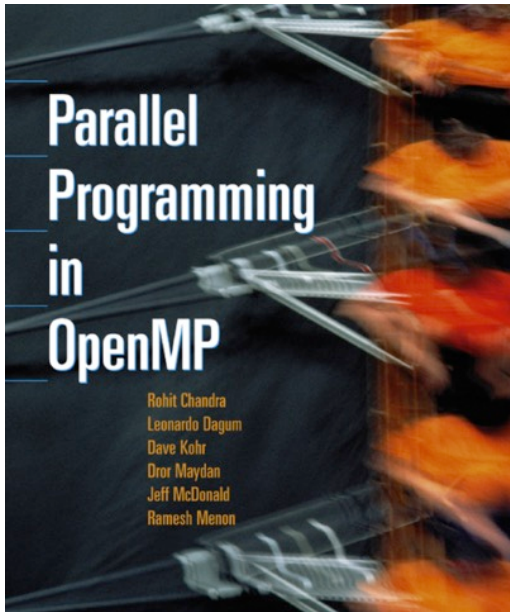
OMP_DYNAMIC : dynamically adjust the number of threads (=true | =false).

KMP_AFFINITY : only for intel compiler, to bind OpenMP threads to physical processing units.
(=compact | =scatter | =balanced).
Example usage: export KMP_AFFINITY= compact,granularity=fine,verbose .

Further information

- References





- OpenMP official website: http://openmp.org/wp/