Chapter 5
Shared Memory Programing with OpenMP

软件学院　邵兵

2022年4月8日

Contents

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

1. WHAT IS OPENMP?

OpenMP

- An API for shared-memory parallel programming.
- MP = multiprocessing
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPU’s, all of which have access to main memory.

https://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf

https://computing.llnl.gov/tutorials/openMP/

OpenMP Is Not

- Meant for distributed memory parallel systems (by itself)
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, deadlocks, or code sequences that cause a program to be classified as non-conforming
- Designed to handle parallel I/O. The programmer is responsible for synchronizing input and output.

Release History

Date	Version
Oct 1997	Fortran 1.0
Oct 1998	C/C++ 1.0
Nov 1999	Fortran 1.1
Nov 2000	Fortran 2.0
Mar 2002	C/C++ 2.0
May 2005	OpenMP 2.5
May 2008	OpenMP 3.0
Jul 2011	OpenMP 3.1
Jul 2013	OpenMP 4.0
Nov 2015	OpenMP 4.5
Nov 2018	OpenMP 5.0

Three Components

The OpenMP 3.1 API is comprised of three distinct components:

- Compiler Directives (19)
- Runtime Library Routines (32)
- Environment Variables (9)

Compiler Directives (19)

No.	Directives Name	No.	Directives Name
1	# pragma omp parallel	11	# pragma omp barrier
2	# pragma omp for	12	# pragma omp taskwait
3	# pragma omp parallel for	13	# pragma omp taskyield
4	# pragma omp sections	14	# pragma omp atomic
5	# pragma omp parallel sections	15	# pragma omp flush
6	# pragma omp section	16	# pragma omp ordered
7	# pragma omp single	17	# pragma omp threadprivate
8	# pragma omp task	18	!$omp parallel workshare (Fortran only)
9	# pragma omp master	19	?
10	# pragma omp critical		

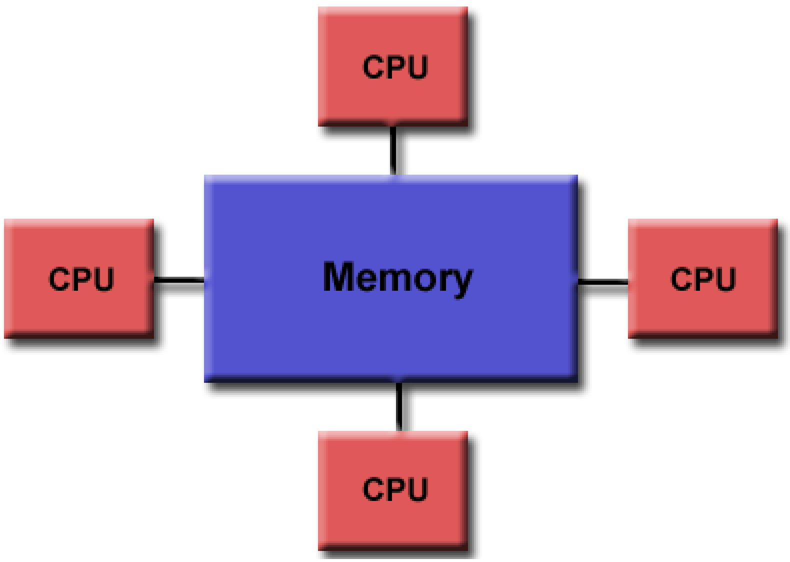
Runtime Library Routines (32)

no.	routine name	no.	routine name
1	omp_set_num_threads	17	omp_get_ancestor_thread_num
2	omp_get_num_threads	18	omp_get_team_size
3	omp_get_max_threads	19	omp_get_active_level
4	omp_get_thread_num	20	omp_in_final
5	omp_get_thread_limit	21	omp_init_lock
6	omp_get_num_procs	22	omp_destroy_lock
7	omp_in_parallel	23	omp_set_lock
8	omp_set_dynamic	24	omp_unset_lock
9	omp_get_dynamic	25	omp_test_lock
10	omp_set_nested	26	omp_init_nest_lock
11	omp_get_nested	27	omp_destroy_nest_lock
12	omp_set_schedule	28	omp_set_nest_lock
13	omp_get_schedule	29	omp_unset_nest_lock
14	omp_set_max_active_levels	30	omp_test_nest_lock
15	omp_get_max_active_levels	31	omp_get_wtime
16	omp_get_level	32	omp_get_wtick

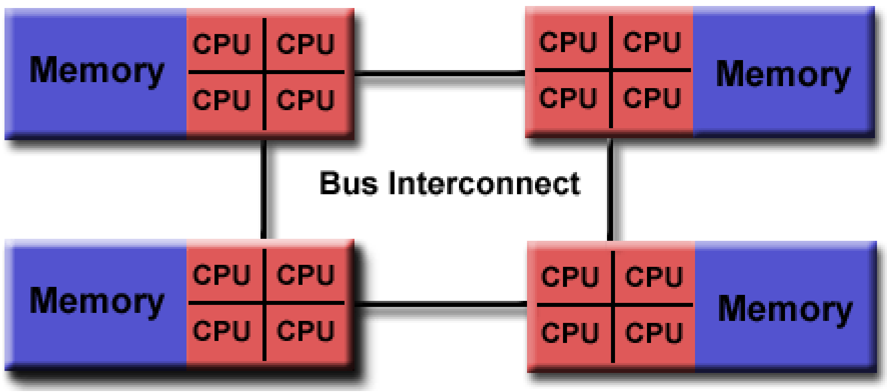
Environment Variables (9)

No.	Environment Variables Name
1	OMP_SCHEDULE
2	OMP_NUM_THREADS
3	OMP_DYNAMIC
4	OMP_PROC_BIND
5	OMP_NESTED
6	OMP_STACKSIZE
7	OMP_WAIT_POLICY
8	OMP_MAX_ACTIVE_LEVELS
9	OMP_THREAD_LIMIT

Shared Memory Model

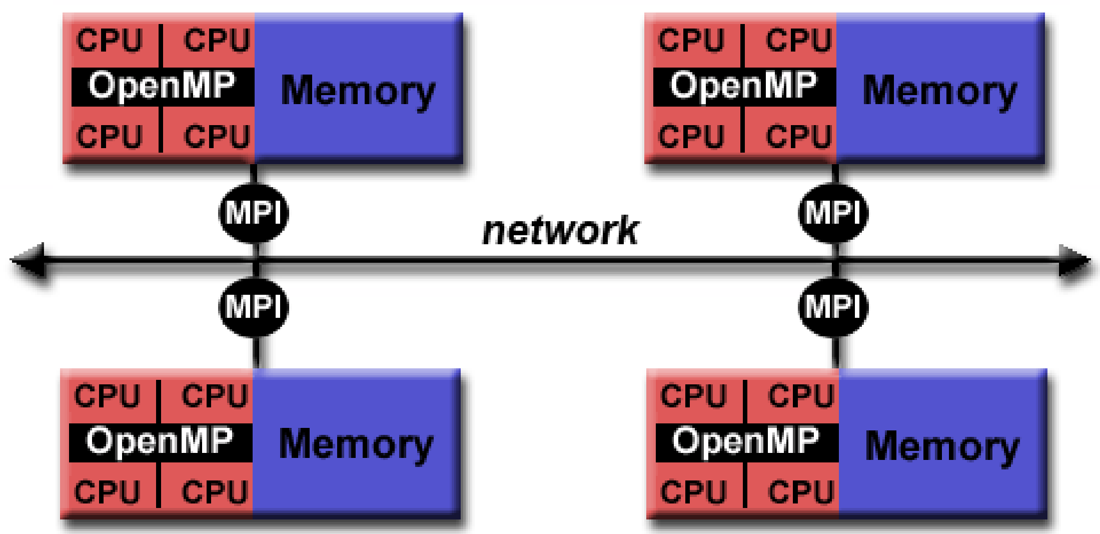


Uniform Memory Access



Non-Uniform Memory Access

Motivation for Using OpenMP in HPC



Hybrid OpenMP-MPI Parallelism

Hello World!

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
} /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   printf("Hello from thread %d of %d\n", my_rank,
         thread_count);
} /* Hello */
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Pragmas

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren’t part of the basic C specification.
- Compilers that don’t support the pragmas ignore them.

```
# pragma
```

```
gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

./omp_hello 4
```

running with 4 threads

compiling

```
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

possible outcomes

```
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

```
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4
```

Compiler Commands for Different Platforms

Compiler / Platform	Compiler Commands	OpenMP Flag
Intel Linux	icc icpc ifort	-qopenmp
GNU Linux IBM Blue Gene Sierra, CORAL EA	**gcc** g++ g77 gfortran	**-fopenmp**
PGI Linux Sierra, CORAL EA	pgcc pgCC pgf77 pgf90	-mp
Clang Linux Sierra, CORAL EA	clang clang++	-fopenmp

Compiler / Platform	Compiler Commands	OpenMP Flag
IBM XL Blue Gene	bgxlc_r, bgcc_r bgxlC_r, bgxlc++_r bgxlc89_r bgxlc99_r bgxlf_r bgxlf90_r bgxlf95_r bgxlf2003_r	-qsmp=omp
IBM XL Sierra, CORAL EA	xlc_r xlC_r, xlc++_r xlf_r xlf90_r xlf95_r xlf2003_r xlf2008_r	-qsmp=omp

Directives Format

Format:

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

Example:

```
#pragma omp parallel default(shared) private(beta, pi)
```

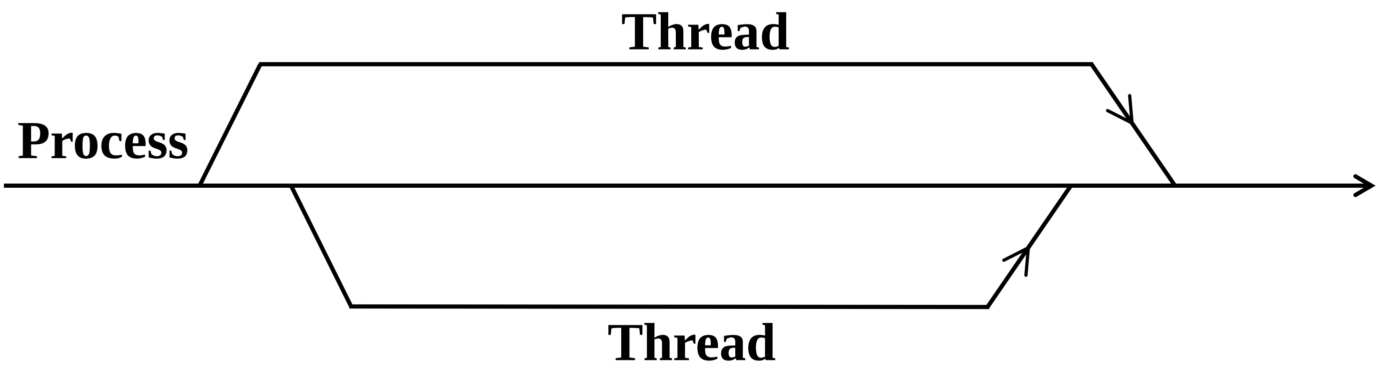
General Rules for Directives

- Case sensitive
- Directives follow conventions of the C/C++ standards for compiler directives
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

OpenMP Pragmas

- # pragma omp parallel
  - Most basic parallel directive.
  - The number of threads that run the following structured block of code is determined by the run-time system.

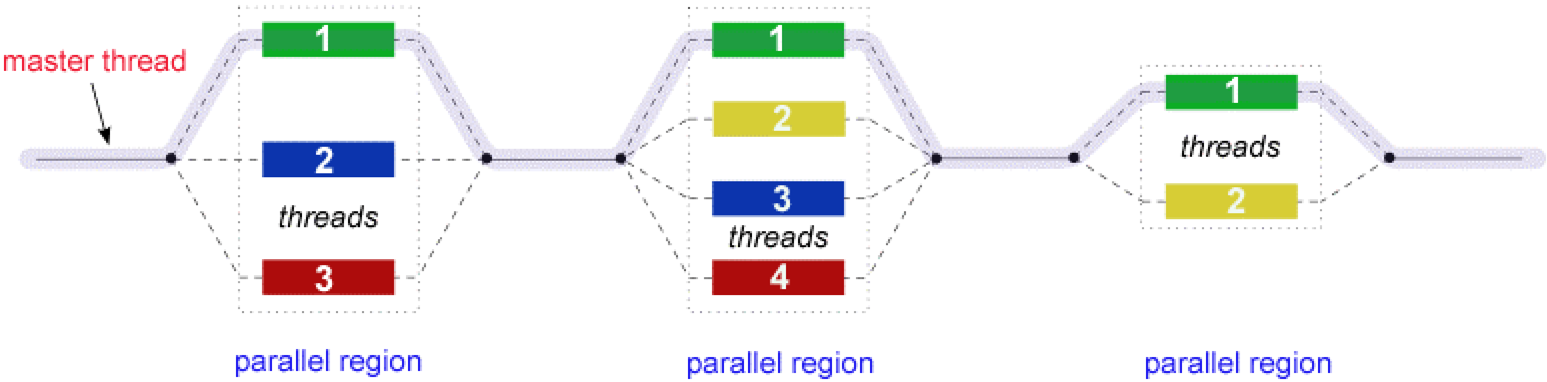
A Process Forking and Joining Two Threads



Some Terminology

- In OpenMP parlance the collection of threads executing the parallel block — the original thread and the new threads — is called a team, the original thread is called the master, and the additional threads are called slaves.

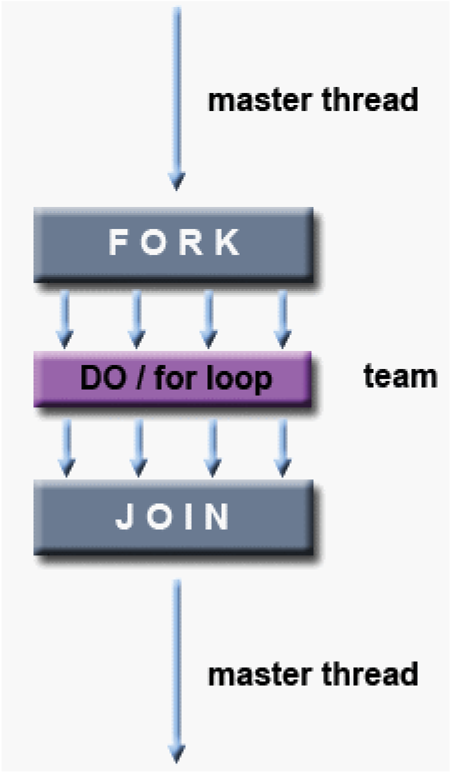
Fork - Join Model



- All OpenMP programs begin as a single process: the **master thread**.
- The master thread executes sequentially until the first **parallel region** construct is encountered.
- **FORK:** the master thread then creates a team of parallel ***threads***.
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads.
- **JOIN:** When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.
- The number of parallel regions and the threads that comprise them are arbitrary.

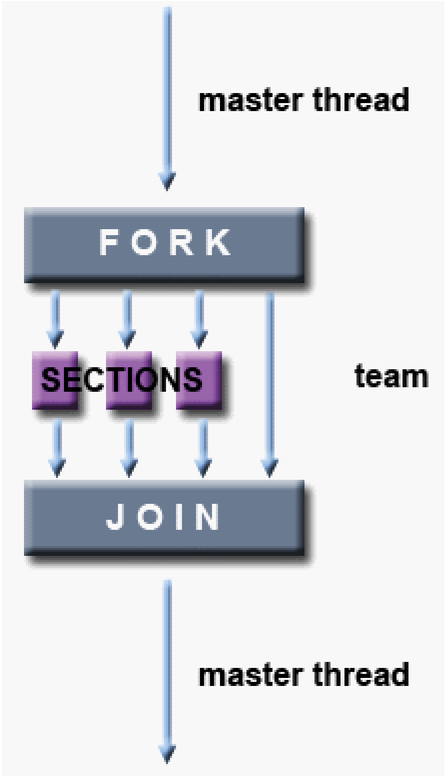
Types of Work-Sharing Constructs

DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".



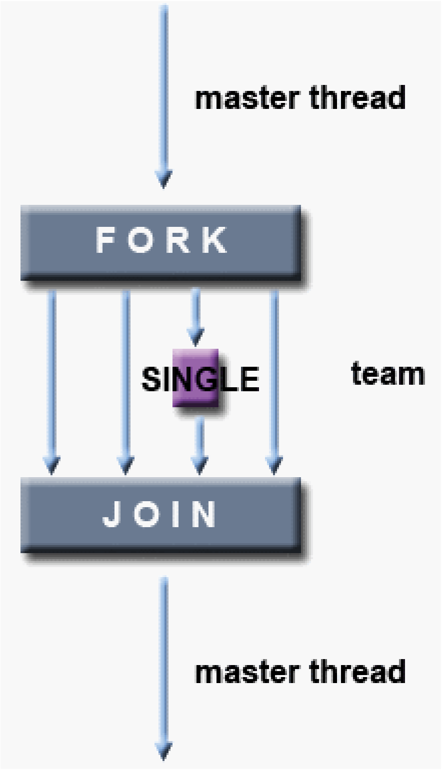
Types of Work-Sharing Constructs

SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".



Types of Work-Sharing Constructs

SINGLE - serializes a section of code



Clause

- Text that modifies a directive.
- The num_threads clause can be added to a parallel directive.
- It allows the programmer to specify the number of threads that should execute the following block.

```
# pragma omp parallel num_threads(thread_count)
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Of Note…

- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn't guarantee that this will actually start thread_count threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we're trying to start a lot of threads, we will almost always get the desired number of threads.

In Case Compilers Don’t Support OpenMP

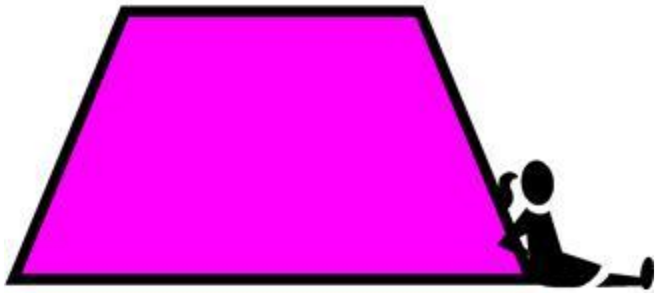
```
#include <omp.h>
```

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

In Case Compilers Don't Support OpenMP

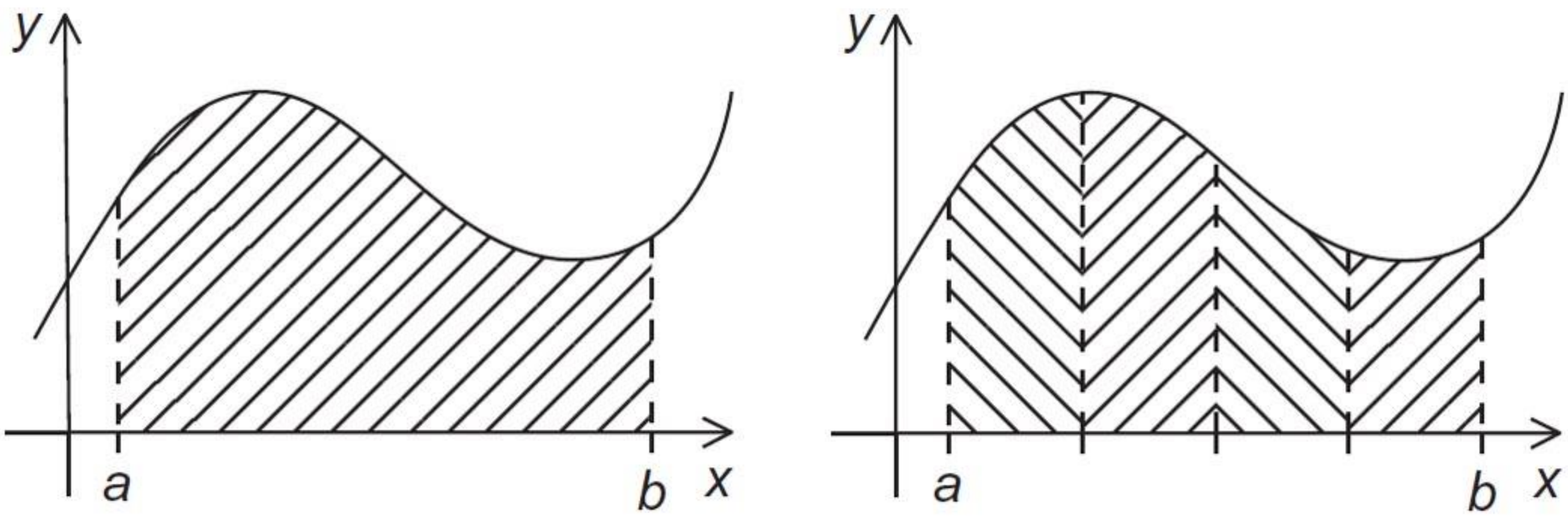
```
#ifdef _OPENMP
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();
#else
    int my_rank = 0;
    int thread_count = 1;
#endif
```



2. THE TRAPEZOIDAL RULE

北京航空航天大學
COLLEGE OF SOFTWARE
BEIHANG UNIVERSITY
软件学院

The Trapezoidal Rule



(a) (b)

Serial Algorithm

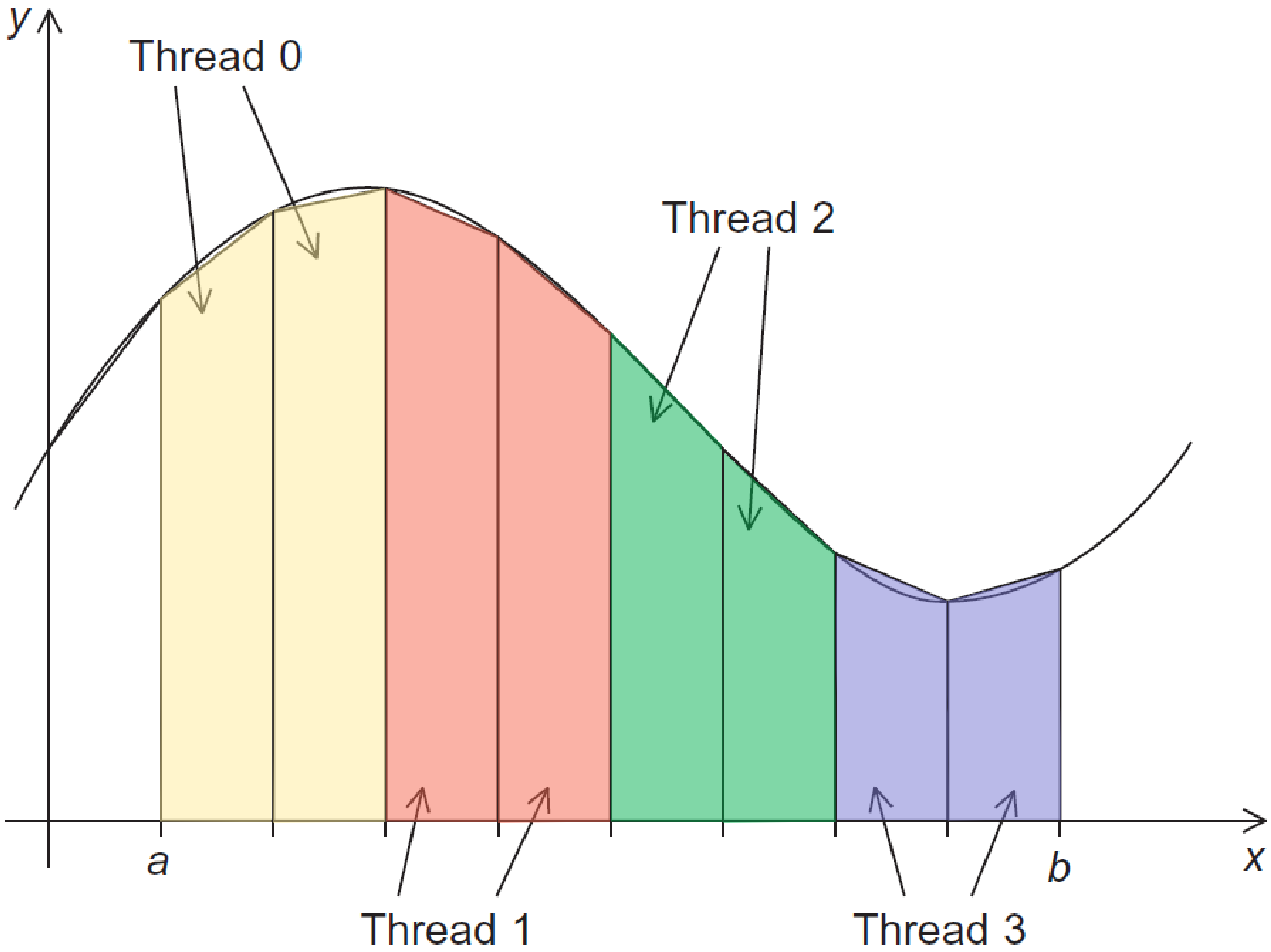
```
/* Input: a, b, n */
h = (b - a) / n;
approx = (f(a) + f(b)) / 2.0;
for (i = 1; i <= n - 1 ; i++) {
   x_i = a + i * h;
   approx += f(x_i);
}
approx = h * apprpx;
```

A First OpenMP Version

1. We identified two types of tasks:
   a. computation of the areas of individual trapezoids, and
   b. adding the areas of trapezoids.
2. There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1(b).
3. We assumed that there would be many more trapezoids than cores.

- So we aggregated tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).

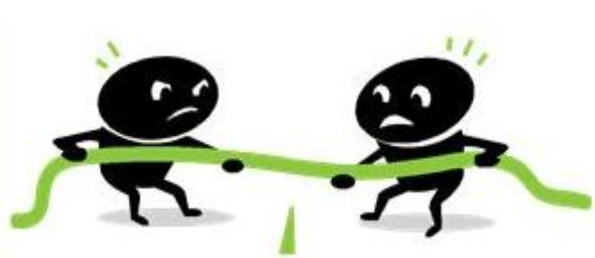
Assignment of Trapezoids to Threads



Time	Thread 0	Thread 1
0	`global_result = 0` to register	finish `my_result`
1	`my_result = 1` to register	`global_result = 0` to register
2	add `my_result` to `global_result`	`my_result = 2` to register
3	store `global_result = 1`	add `my_result` to `global_result`
4		store `global_result = 2`

Unpredictable results when two (or more) threads attempt to simultaneously execute :

```
global_result += my_result;
```



Mutual Exclusion

```
# pragma omp critical
    global_result += my_result;
```

only one thread can execute the following structured block at a time

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* global_result中存放全局和 */
   double  a, b;                 /* 左、右端点坐标            */
   int     n;                    /* 总的小梯形个数            */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
   if (n % thread_count != 0) Usage(argv[0]);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n", a, b,
         global_result);
   return 0;
}  /* main */
```

```c
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();         /* 获取当前线程号 */
   int thread_count = omp_get_num_threads(); /* 获取线程总数    */

   h = (b - a)/n;
   local_n = n / thread_count;
   local_a = a + my_rank * local_n * h;
   local_b = local_a + local_n * h;
   my_result = (f(local_a) + f(local_b)) / 2.0;
   for (i = 1; i <= local_n-1; i++) {
      x = local_a + i * h;
      my_result += f(x);
   }
   my_result = my_result * h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

SCOPE OF VARIABLES

Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.

- In OpenMP, the **scope** of a variable refers to the set of threads that can access the variable in a `parallel` block.

Scope in OpenMP

- A variable that can be accessed by all the threads in the team has shared scope.
- A variable that can only be accessed by a single thread has private scope.

- The default scope for variables declared before a `parallel` block is shared; The default scope for variables declared in the `parallel` block is private.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Scope in OpenMP

- With the default clause, OpenMP can modify variables' default scope.

- The value of a shared variable at the beginning of the `parallel` block is the same as the value before the block; After completion of the `parallel` block, the value of the variable is the value at the end of the block.

THE REDUCTION CLAUSE

We need this more complex version to add each thread's local calculation to get *global_result*.

```
void Trap(double a, double b, int n, double* global_result_p);
```

Although we'd prefer this.

```
double Trap(double a, double b, int n)
```

```
global_result = Trap(a, b, n);
```

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

If we use this, there's no critical section!

```
double Local_trap(double a, double b, int n);
```

If we fix it like this…

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
{
#  pragma omp critical
   global_result += Local_trap(double a, double b, int n);
}
```

… we force the threads to execute sequentially.

We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

```
   global_result = 0.0;
# pragma omp parallel num_threads(thread_count)
{
   double my_result = 0.0;      /* private */
   my_result += Local_trap(double a, double b, int n);
#  pragma omp critical
   global_result += my_result;
}
```

I think we can do better.

Neither do I.

I don't like it.

Reduction Operators

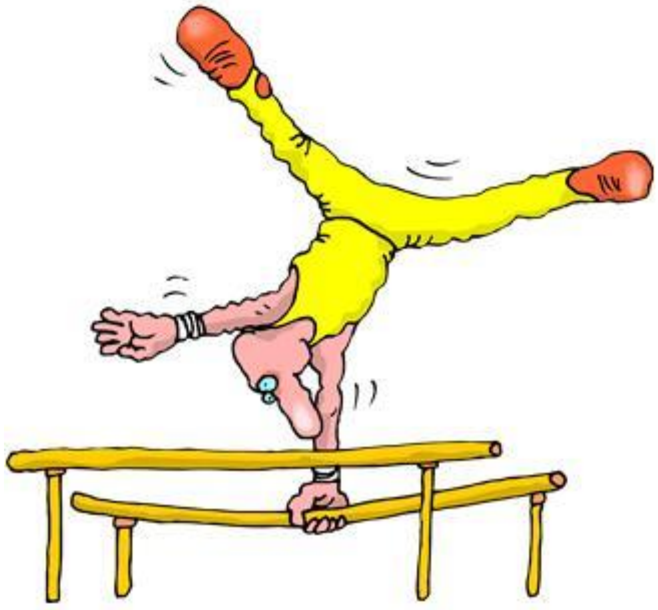
- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.

A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```

```
+, *, -, &, |, ^, &&, ||
```

```
global_result = 0.0;
# pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
global_result += Local_trap(double a, double b, int n);
```



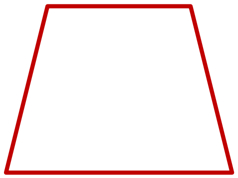
3. THE “PARALLEL FOR” DIRECTIVE

Parallel for

- Forks a team of threads to execute the following structured block.
- However, the structured block following the `parallel for` directive must be a for loop.
- Furthermore, with the `parallel for` directive the system parallelizes the for loop by dividing the iterations of the loop among the threads.

Sum of trapezoid areas $= h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2]$

```
h = (b - a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n - 1; i++)
  approx += f(a + i * h);
approx = h * approx;
```



```
  h = (b - a)/n;
  approx = (f(a) + f(b))/2.0;
# pragma omp parallel for num_threads(thread_count) \
     reduction(+: approx)
  for (i = 1; i <= n - 1; i++)
    approx += f(a + i * h);
  approx = h * approx;
```

Caveats

- The variable index must have integer or pointer type (e.g., it can't be a float).
- The expressions start, end, and incr must have a compatible type. For example, if index is a pointer, then incr must have integer type.
- The expressions start, end, and incr must not change during execution of the loop.
- During execution of the loop, the variable index can only be modified by the "increment expression" in the for statement.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Legal Forms for Parallelizable for Statements

$$
\text{for}\left(\text{index} = \text{start} \quad ; \quad \begin{array}{l} \text{index} < \quad \text{end} \\ \text{index} <= \text{end} \\ \text{index} >= \text{end} \\ \text{index} > \quad \text{end} \end{array} \quad ; \quad \begin{array}{l} \text{index}++ \\ ++\text{index} \\ \text{index}-- \\ --\text{index} \\ \text{index} += \text{incr} \\ \text{index} -= \text{incr} \\ \text{index} = \text{index} + \text{incr} \\ \text{index} = \text{incr} + \text{index} \\ \text{index} = \text{index} - \text{incr} \end{array}\right)
$$

北京航空航天大學

COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Data Dependencies

```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
   fibo[i] = fibo[i-1] + fibo[i-2];
```

note 2 threads

```
  fibo[0] = fibo[1] = 1;
# pragma omp parallel for num_threads(2)
  for (i = 2; i < n; i++)
     fibo[i] = fibo[i-1] + fibo[i-2];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes we get this

1 1 2 3 5 8 0 0 0 0

fibo[0] fibo[1] fibo[2] fibo[3] fibo[4] fibo[5] fibo[6] fibo[7] fibo[8] fibo[9]

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

What Happened?

1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a `parallel for` directive.
2. A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.

Data Dependence & Loop-Carried Dependence

- The dependence of the computation of fibo[6] on the computation of fibo[5] is called a data dependence.
- Since the value of fibo[5] is calculated in one iteration, and the result is used in a subsequent iteration, the dependence is sometimes called a loop-carried dependence.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Estimating π

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right) = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
  sum += factor/(2*k+1);
  factor = -factor;
}
pi_approx = 4.0*sum;
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

OpenMP Solution #1

loop dependency

```
  double factor = 1.0;
  double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
     reduction(+:sum)
  for (k = 0; k < n; k++) {
     sum += factor/(2*k+1);
     factor = -factor;
  }
  pi_approx = 4.0*sum;
```

OpenMP Solution #2

```
   double factor = 1.0;
   double sum = 0.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+:sum)
   for (k = 0; k < n; k++) {
      if (k % 2 == 0)
         factor = 1.0;
      else
         factor = -1.0;
      sum += factor/(2*k+1);
}
```

```
factor = (k%2 == 0)? 1.0: -1.0;
sum += factor/(2*k+1);
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

However, Things Still aren't Quite Right.

```
$With n = 1000 terms and 2 threads,
$Our estimate of pi = 2.97063289263385
$With n = 1000 terms and 2 threads,
$Our estimate of pi = 3.22392164798593
```

With n = 1000 terms and 2 threads twice

With n = 1000 terms and 1 threads once

```
$With n = 1000 terms and 1 threads,
$Our estimate of pi = 3.14059265383979
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

OpenMP Solution #3

```
  double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
     reduction(+:sum) private(factor)
  for (k = 0; k < n; k++) {
     if (k % 2 == 0)
        factor = 1.0;
     else
        factor = -1.0;
     sum += factor/(2*k+1);
}
```

Insures factor has private scope.

The Private Clause

The PRIVATE clause declares variables in its list to be private to each thread.

PRIVATE variables behave as follows:

1. A new object of the same type is declared once for each thread in the team
2. All references to the original object are replaced with references to the new object
3. Should be assumed to be uninitialized for each thread

The Default Clause

- Lets the programmer specify the scope of each variable in a block.

```
default(none)
```

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

The Default Clause

```
  double sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
     default(none) reduction(+:sum) private(k, factor) \
     shared(n)
  for (k = 0; k < n; k++) {
     if (k % 2 == 0)
        factor = 1.0;
     else
        factor = -1.0;
     sum += factor/(2*k+1);
}
```

编程人员需明确并行块中使用的每一个变量的作用域!

私有变量在线程栈上分配空间，共享变量在进程数据区分配空间。

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

MORE ABOUT LOOPS IN OPENMP: SORTING

Bubble Sort

```
for (list_length = n; list_length >= 2; list_length--)
    for (i = 0; i < list_length-1; i++)
        if (a[i] > a[i+1]) {
            tmp = a[i];
            a[i] = a[i+1];
            a[i+1] = tmp;
        }
```

Serial Odd-Even Transposition Sort

```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n ; i += 2)
         if (a[i-1] > a[i])
            tmp = a[i-1]; a[i-1] = a[i]; a[i] = tmp;
   else
      for (i = 1; i < n-1; i += 2)
         if (a[i] > a[i+1]) {
            tmp = a[i+1]; a[i+1] = a[i]; a[i] = tmp;
         }
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Serial Odd-Even Transposition Sort

Phase	**Subscript in Array**						
	0		1		2		3
0	9	↔	7		8	↔	6
	7		9		6		8
1	7		9	↔	6		8
	7		6		9		8
2	7	↔	6		9	↔	8
	6		7		8		9
3	6		7	↔	8		9
	6		7		8		9

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

First OpenMP Odd-Even Sort

```
for (phase = 0; phase < n; phase++) {
   if (phase % 2 == 0)    /* 偶数阶段 */
#     pragma omp parallel for num_threads(thread_count) \
         default(none) shared(a, n) private(i, tmp)
      for (i = 1; i < n; i += 2) {
         if (a[i-1] > a[i]) {
            tmp = a[i-1]; a[i-1] = a[i]; a[i] = tmp;
         }
      }
   else   /* 奇数阶段 */
#     pragma omp parallel for num_threads(thread_count) \
         default(none) shared(a, n) private(i, tmp)
      for (i = 1; i < n-1; i += 2) {
         if (a[i] > a[i+1]) {
            tmp = a[i+1]; a[i+1] = a[i]; a[i] = tmp;
         }
      }
}
```

Second OpenMP Odd-Even Sort

```
#  pragma omp parallel num_threads(thread_count) \
     default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i])
               tmp = a[i-1]; a[i-1] = a[i]; a[i] = tmp;
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1])
               tmp = a[i+1]; a[i+1] = a[i]; a[i] = tmp;
         }
   }
```

Odd-Even Sort with Two Parallel for Directives and Two for Directives.

(Times are in seconds. n=20,000)

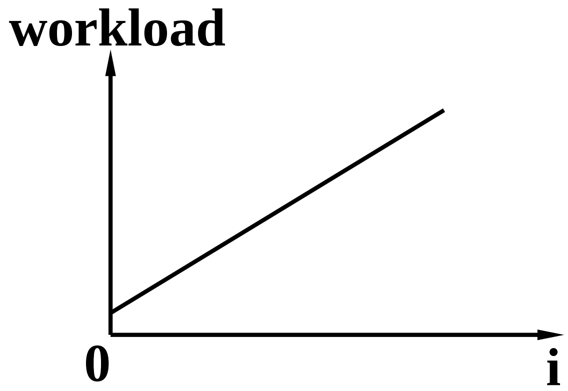
`thread_count`	1	2	3	4
Two `parallel for` directives	0.770	0.453	0.358	0.305
Two `for` directives	0.732	0.376	0.294	0.239
Speed up ↑	4.9%	17.0%	17.9%	21.6%

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

SCHEDULING LOOPS

We want to parallelize this loop.

```
sum = 0.0;
for (i = 0; i <= n; i++)
      sum += f(i);
```




```
double f(int i) {
    int j , start = i *(i + 1)/2, finish = start + i;
    double return_val = 0.0;
    for (j = start ; j <= finish ; j++) {
      return_val += sin(j);
    }
    return return_val ;
} /* f */
```

Our definition of function *f*.

Results

- f(i) calls the sin function *i* times.
- Assume the time to execute f(2i) requires approximately twice as much time as the time to execute f(i).

- n = 10,000
  - one thread
  - run-time = 3.67 seconds.

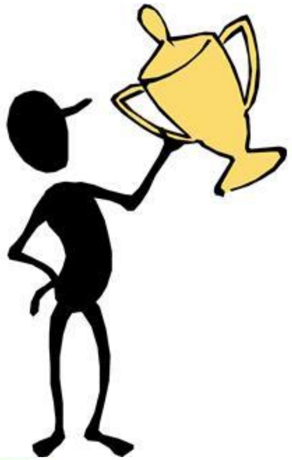
Assignment of work using cyclic partitioning.

Thread	Iterations
0	0, n/t, 2n/t, ...
1	1, n/t+1, 2n/t+1, ...
⋮	⋮
t-1	t-1, n/t+t-1, 2n/t+t-1, ...

Results

- n = 10,000
  - two threads
  - default assignment
  - run-time = 2.76 seconds
  - speedup = 1.33

- n = 10,000
  - two threads
  - cyclic assignment
  - run-time = 1.84 seconds
  - speedup = 1.99



航天大學 软件学院

The Schedule Clause

- Default schedule:

```
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction(+: sum)
for (i = 0; i <= n; i++)
    sum += f(i);
```

- Cyclic schedule:

```
sum = 0.0;
# pragma omp parallel for num_threads(thread_count) \
    reduction (+: sum) schedule(static, 1)
for (i = 0; i <= n; i++)
    sum += f(i);
```

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

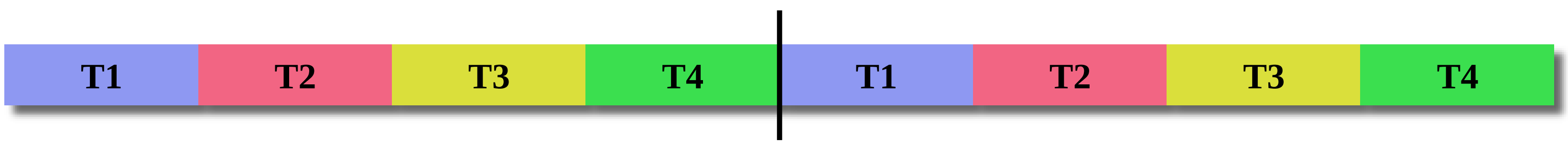
Schedule (type, chunksize)

- Type can be:
  - **static**: the iterations can be assigned to the threads before the loop is executed.
  - **dynamic** or **guided**: the iterations are assigned to the threads while the loop is executing.
  - **auto**: the compiler and/or the run-time system determine the schedule.
  - **runtime**: The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.
- The chunksize is a positive integer.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

The Static Schedule Type

- Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads.
- If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.



平均分配。块大小固定

北京航空航天大学
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

The Static Schedule Type

twelve iterations, 0, 1, . . . , 11, and three threads

`schedule(static,1)`

Thread 0 : 0, 3, 6, 9
Thread 1 : 1, 4, 7, 10
Thread 2 : 2, 5, 8, 11

`schedule(static,2)`

Thread 0 : 0, 1, 6, 7
Thread 1 : 2, 3, 8, 9
Thread 2 : 4, 5, 10, 11

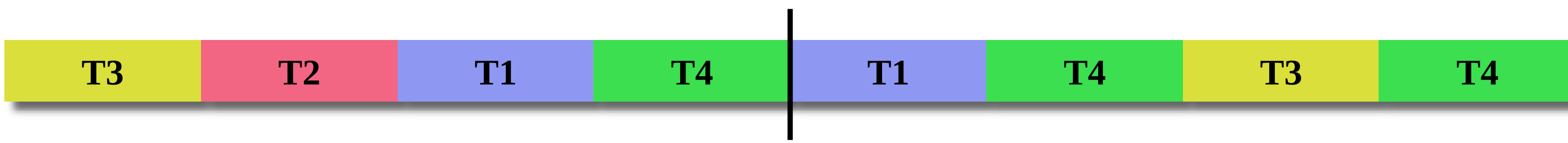
`schedule(static,4)`

Thread 0 : 0, 1, 2, 3
Thread 1 : 4, 5, 6, 7
Thread 2 : 8, 9, 10, 11

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

The Dynamic Schedule Type

- Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads;
- When a thread finishes one chunk, it is dynamically assigned another.
- The default chunk size is 1.



能者多劳。块大小固定

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

The Guided Schedule Type

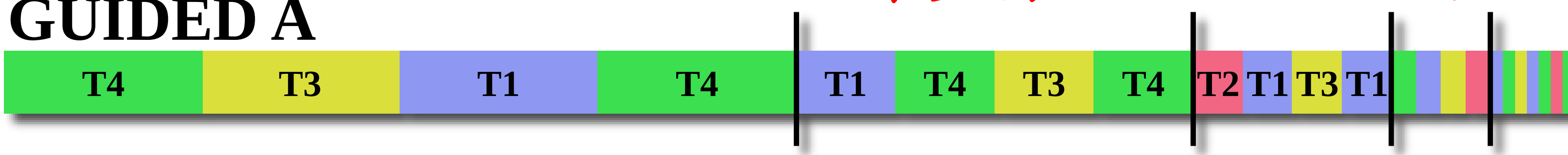
- Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.
- The size of the initial block is proportional to:
  `number_of_iterations/number_of_threads.`
  Subsequent blocks are proportional to
  `number_of_iterations_remaining/number_of_threads`
- The chunk parameter defines the minimum block size.
- The default chunk size is 1.

The Guided Schedule Type

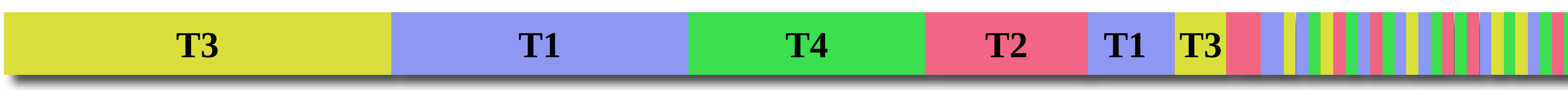
- Note: compilers differ in how GUIDED is implemented as shown in the "Guided A" and "Guided B" examples below.

能者多劳。块大小逐轮递减

GUIDED A



GUIDED B



能者多劳。块大小逐次递减

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

The Runtime Schedule Type

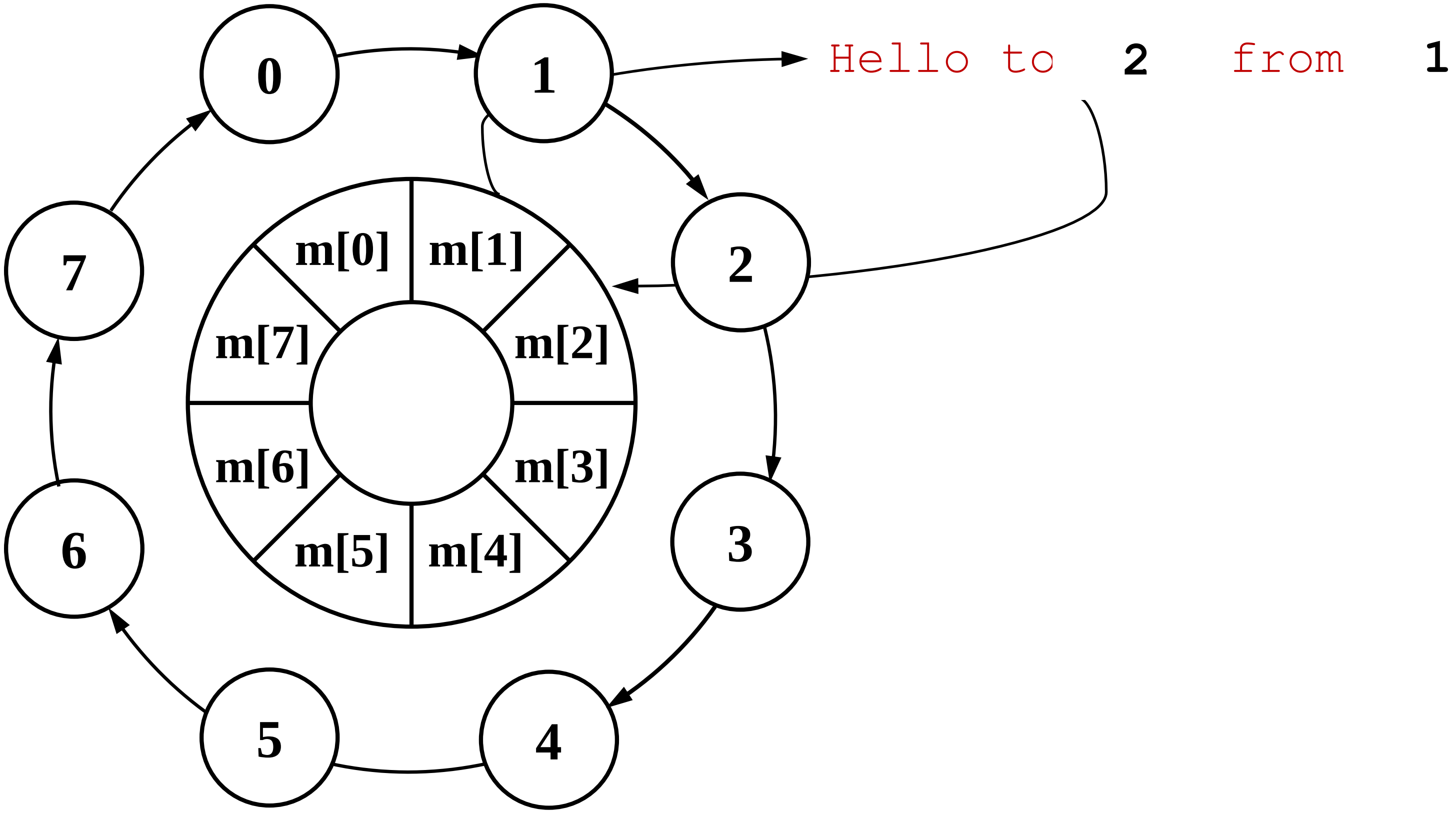
- The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop.
- The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.

4. PRODUCERS AND CONSUMERS

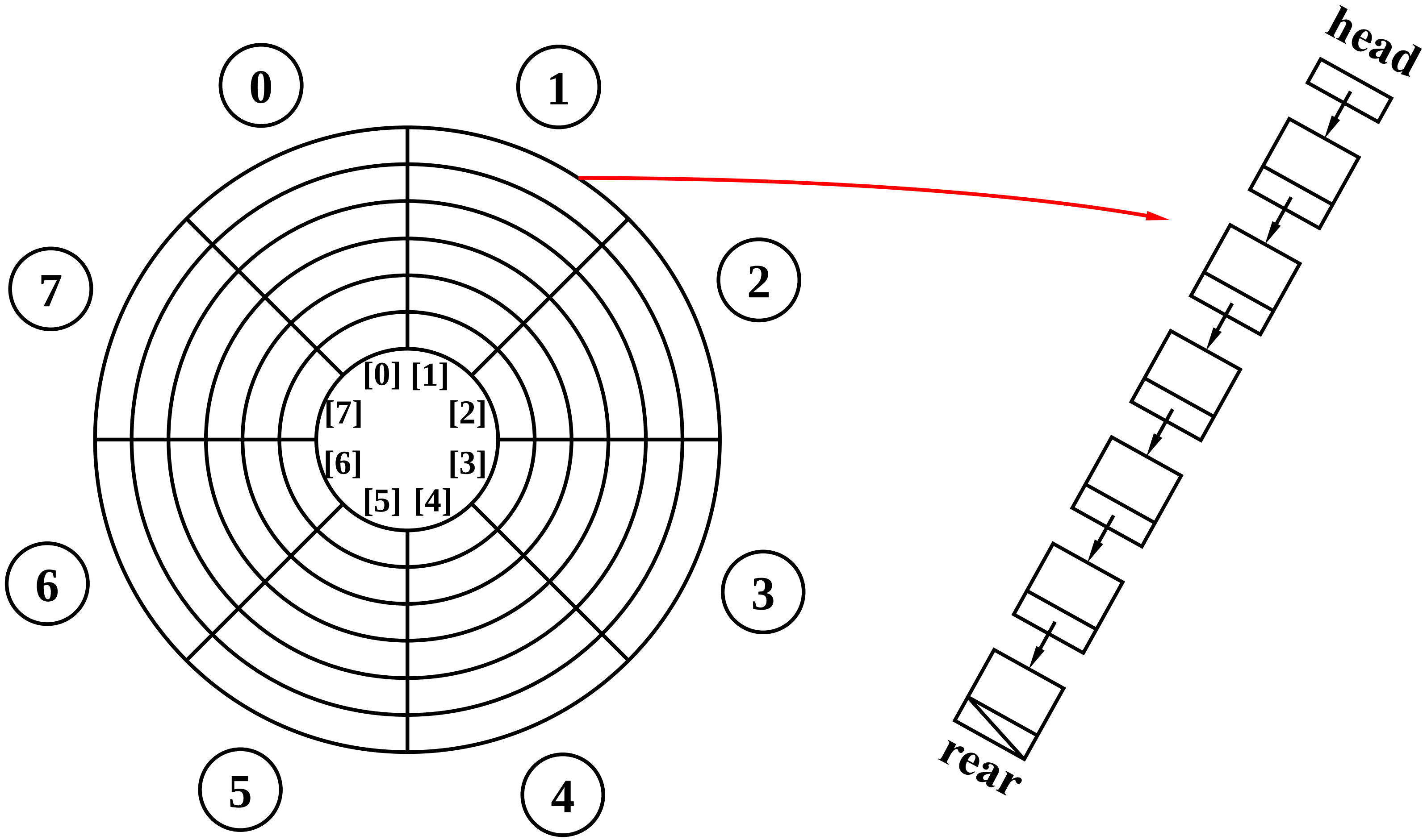
Queues

- A natural data structure to use in many multithreaded applications. –FIFO (First In First Out)
- For example, suppose we have several “producer” threads and several “consumer” threads.
  - Producer threads might “produce” requests for data.
  - Consumer threads might “consume” the request by finding or generating the requested data.

Do you remember this? – Chapter4



We change this model as…



Message-Passing

- Each thread could have a shared message queue, and when one thread wants to “send a message” to another thread, it could enqueue the message in the destination thread’s queue.
- A thread could receive a message by dequeuing the message at the head of its message queue.

Message-Passing

```
for (sent_msgs=0; sent_msgs < send_max; sent_msgs++) {
    Send_msg();
    Try_receive();
}
while (!Done())
    Try_receive();
```

Sending Messages

```
  mesg = random();
  dest = random() % thread_count ;
# pragma omp critical
  Enqueue(queue, dest, my_rank, mesg);
```

Receiving Messages

```
queue_size = enqueued - dequeued;
if (queue_size == 0)
   return;
else if (queue_size == 1)
#  pragma omp critical
   Dequeue(queue, &src, &mesg);
else
   Dequeue(queue, &src, &mesg);
Print_message(src, mesg);
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Termination Detection

```
queue_size = enqueued - dequeued ;
if (queue_size == 0 && done_sending == thread_count)
    return TRUE ;
else
    return FALSE ;
```

each thread increments this after completing its for loop

Startup (1)

- When the program begins execution, a single thread, the master thread, will get command line arguments and allocate an array of message queues: one for each thread.
- This array needs to be shared among the threads, since any thread can send to any other thread, and hence any thread can enqueue a message in any of the queues.

Startup (2)

- One or more threads may finish allocating their queues before some other threads.
- We need an explicit barrier so that when a thread encounters the barrier, it blocks until all the threads in the team have reached the barrier.
- After all the threads have reached the barrier all the threads in the team can proceed.

```
# pragma omp barrier
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

The `Atomic` Directive (1)

- Unlike the `critical` directive, it can only protect critical sections that consist of a single C assignment statement.

```
# pragma omp atomic
```

- Further, the statement must have one of the following forms:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

The `Atomic` Directive (2)

- Here <op> can be one of the binary operators

  `+, *, -, /, &, ^, |, <<` or `>>`

- Many processors provide a special load-modify-store instruction.
- A critical section that only does a load-modify-store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general critical sections.

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Critical Sections

- OpenMP provides the option of adding a name to a critical directive:

```
# pragma omp critical(name)
```

- When we do this, two blocks protected with critical directives with different names can be executed simultaneously.
- However, the names are set during compilation, and we want a different critical section for each thread's queue.

Locks

- A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section.

Locks

```
/* Executed by one thread */
Initialize the lock data structure ;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure ;
Critical section ;
Unlock or unset the lock data structure ;
. . .
/* Executed by one thread */
Destroy the lock data structure ;
```

Using Locks in the Message-Passing Program

```
# pragma omp critical
/* q_p = msg_queues[dest] */
Enqueue(q_p, my_rank, mesg);
```

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock );
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```

Using Locks in the Message-Passing Program

```
# pragma omp critical
  /* q_p = msg_queues[my_rank] */
  Dequeue (q_p, &src, &mesg );
```

```
/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock );
Dequeue(q_p, &src, &mesg);
omp_unset_lock(&q_p->lock);
```

Some Caveats

1. You shouldn’t mix the different types of mutual exclusion for a single critical section.
2. There is no guarantee of fairness in mutual exclusion constructs.
3. It can be dangerous to “nest” mutual exclusion constructs.

5. CACHE-COHERENCE & THREAD-SAFETY

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Cache-Coherence, and False Sharing

- Recall that chip designers have added blocks of relatively fast memory to processors called cache memory.
- The use of cache memory can have a huge impact on shared-memory.
- A write-miss occurs when a core tries to update a variable that's not in cache, and it has to access main memory.

Matrix-Vector Multiplication

$$y_i = a_{i,0}x_0 + a_{i,1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

$$\begin{array}{|c|c|c|c|}\hline a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ \hline a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i,0} & a_{i,1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline \end{array} \bullet \begin{array}{|c|}\hline x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{n-1} \\ \hline \end{array} = \begin{array}{|c|}\hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i,0}x_0 + a_{i,1}x_1 + \cdots + a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline \end{array}$$

```
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

OpenMP Matrix-Vector Multiplication

```
# pragma omp parallel for num_threads(thread_count) \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
}
```

write-miss

read-miss

OpenMP Run-times and efficiencies

Threads	Matrix Dimension					
	8,000,000×8		8,000×8,000		8×8,000,000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

(times are in seconds)

北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Pthreads Matrix-Vector Multiplication

```
void *Pth_mat_vect(void* rank){
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++){
write-miss →  y[i] = 0.0;
      for (j = 0; j < n; j++)
read-miss  →     y[i] += A[i][j]*x[j];
   }

      return NULL;
} /* Pth_mat_vect */
```

Pthreads Run-times and Efficiencies

Threads	Matrix Dimension					
	8,000,000×8		8,000×8,000		8×8,000,000	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

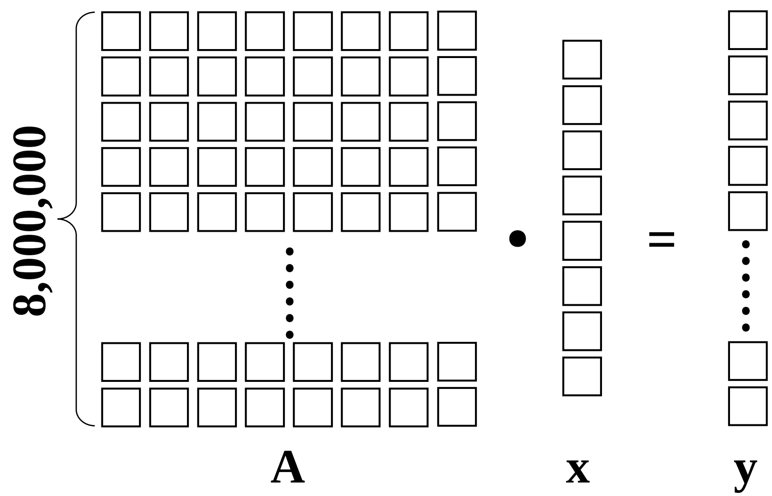
(times are in seconds)

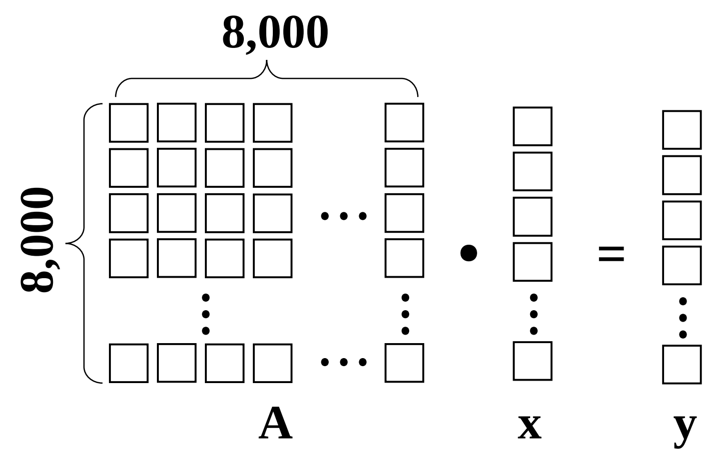
北京航空航天大學
COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

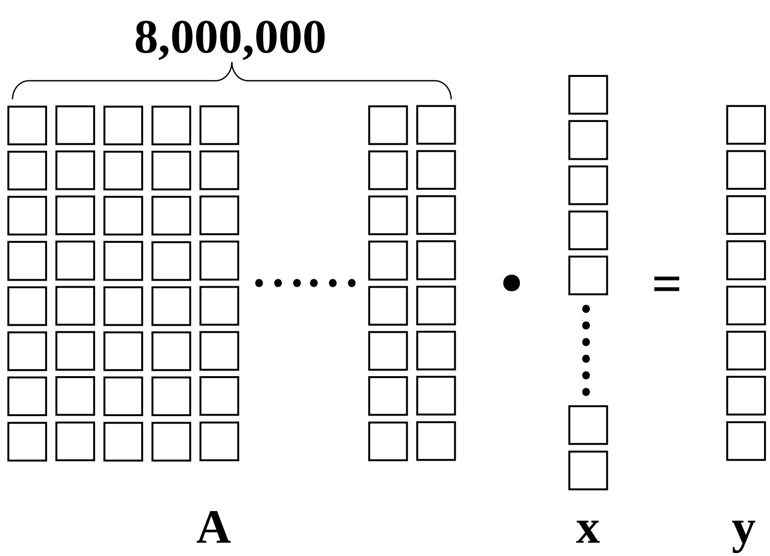
False Sharing

The threads aren’t sharing anything (except a cache line), but the behavior of the threads with respect to memory access is the same as if they were sharing a variable.

8,000,000×8 vs 8,000×8,000 vs 8×8,000,000







Thread-Safety

- A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.

Example

- Suppose we want to use multiple threads to "tokenize" a file that consists of ordinary English text.
- The tokens are just contiguous sequences of characters separated from the rest of the text by white-space — a space, a tab, or a newline.

Simple Approach

- Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.
- The first line goes to thread 0, the second goes to thread 1, . . . , the tth goes to thread t, the t+1st goes to thread 0, etc.

Simple Approach

- We can serialize access to the lines of input using semaphores in Pthreads program.
- After a thread has read a single line of input, it can tokenize the line using the strtok function.

The Strtok Function

- The first time it's called the string argument should be the text to be tokenized.
  - Our line of input.
- For subsequent calls, the first argument should be NULL.

```
char* strtok(
      char*          string      /* in/out */
      const char*    separators  /* in     */);
```

The Strtok Function

- The idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens taken from the cached copy.

Multi-Threaded Tokenizer (1)

```
void* Tokenize(void* rank){
   long my_rank = (long)rank;
   int count;
   int next = (my_rank + 1)%thread_count;
   char *fg_rv;
   char my_line[MAX];
   char *my_string;

   sem_wait(&sems[my_rank]);
   fg_rv = fgets(my_line, MAX, stdin);
   sem_post(&sems[next]);
   while (fg_rv != NULL){
      printf("Thread %ld > my_line = %s", my_rank,
            my_line);
```

北京航空航天大學 COLLEGE OF SOFTWARE BEIHANG UNIVERSITY 软件学院

Multi-Threaded Tokenizer (2)

```
      count = 0;
      my_string = strtok(my_line, " \t\n");
      while (my_string != NULL) {
         count++;
         printf("Thread %ld > string %d = %s\n", my_rank,
              count, my_string);
         my_string = strtok(NULL, " \t\n");
      }
      sem_wait(&sems[my_rank]);
      fg_rv = fgets(my_line, MAX, stdin);
      sem_post(&sems[next]);
   }
   return NULL;
} /* Tokenize */
```

Running with One Thread

- It correctly tokenizes the input stream.

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

Running with Two Threads

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

Oops!

What Happened?

- strtok caches the input line by declaring a variable to have static storage class.
- This causes the value stored in this variable to persist from one call to the next.
- Unfortunately for us, this cached string is shared, not private.

What Happened?

- Thus, thread 0's call to strtok with the third line of the input has apparently overwritten the contents of thread 1's call with the second line.
- So the strtok function is not thread-safe.
- If multiple threads call it simultaneously, the output may not be correct.

Thread-Safety

```
void Tokenize(
      char*  lines[]       /* in/out */,
      int    line_count    /* in     */,
      int    thread_count  /* in     */) {
   int my_rank, i, j;
   char *my_token;

#  pragma omp parallel num_threads(thread_count) \
      default(none) private(my_rank, i, j, my_token, saveptr) \
      shared(lines, line_count)
   {
      my_rank = omp_get_thread_num();
#     pragma omp for schedule(static, 1)
      for (i = 0; i < line_count; i++) {
         printf("Thread %d > line %d = %s", my_rank, i, lines[i]);
         j = 0;
         my_token = strtok_r(lines[i], " \t\n", &saveptr);
         while ( my_token != NULL ) {
            printf("Thread %d > token %d = %s\n", my_rank, j, my_token);
            my_token = strtok_r(NULL, " \t\n", &saveptr);
            j++;
         }
      } /* for i */
   }  /* omp parallel */
}  /* Tokenize */
```

Other Unsafe C Library Functions

- Regrettably, it's not uncommon for C library functions to fail to be thread-safe.
- The random number generator random in stdlib.h.
- The time conversion function localtime in time.h.

"Re-entrant" (Thread Safe) Functions

- In some cases, the C standard specifies an alternate, thread-safe, version of a function.

```
char* strtok_r(
      char*        string      /* in/out */,
      const char*  separators, /* in     */,
      char**       saveptr_p   /* in/out */);
```

可重入函数主要用于多任务环境中，一个可重入的函数简单来说就是可以被中断的函数。也就是说，可以在这个函数执行的任何时刻中断它，转入OS调度下去执行另外一段代码，而返回控制时不会出现什么错误；

而不可重入的函数由于使用了一些系统资源，比如全局变量区，中断向量表等，所以它如果被中断的话，可能会出现问题。这类函数是不能运行在多任务环境下的。

——百度百科

Concluding Remarks (1)

- OpenMP is a standard for programming shared-memory systems.
- OpenMP uses both special functions and preprocessor directives called pragmas.
- OpenMP programs start multiple threads rather than multiple processes.
- Many OpenMP directives can be modified by clauses.

Concluding Remarks (2)

- A major problem in the development of shared memory programs is the possibility of race conditions.
- OpenMP provides several mechanisms for insuring mutual exclusion in critical sections.
  - Critical directives
  - Named critical directives
  - Atomic directives
  - Simple locks

Concluding Remarks (3)

- By default most systems use a block-partitioning of the iterations in a parallelized for loop.
- OpenMP offers a variety of scheduling options.
- In OpenMP the scope of a variable is the collection of threads to which the variable is accessible.
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.

Concluding Remarks (4)

- Some C functions cache data between calls by declaring variables to be static, causing errors when multiple threads call the function.
- This type of function is not thread-safe.

编程人员应确保线程安全性。

如无法保证，宁可保守地采用串行化的实现方案！

Clauses / Directives Summary

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO /for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
COPYPRIVATE				●		
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Clauses / Directives Summary

The following OpenMP directives do not accept clauses:

- MASTER
- CRITICAL
- BARRIER
- ATOMIC
- FLUSH
- ORDERED
- THREADPRIVATE

Implementations may (and do) differ from the standard in which clauses are supported by each directive.