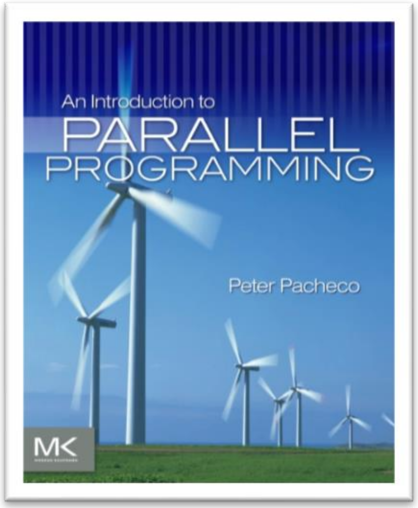


Section4-Shared Memory Programming with OpenMP

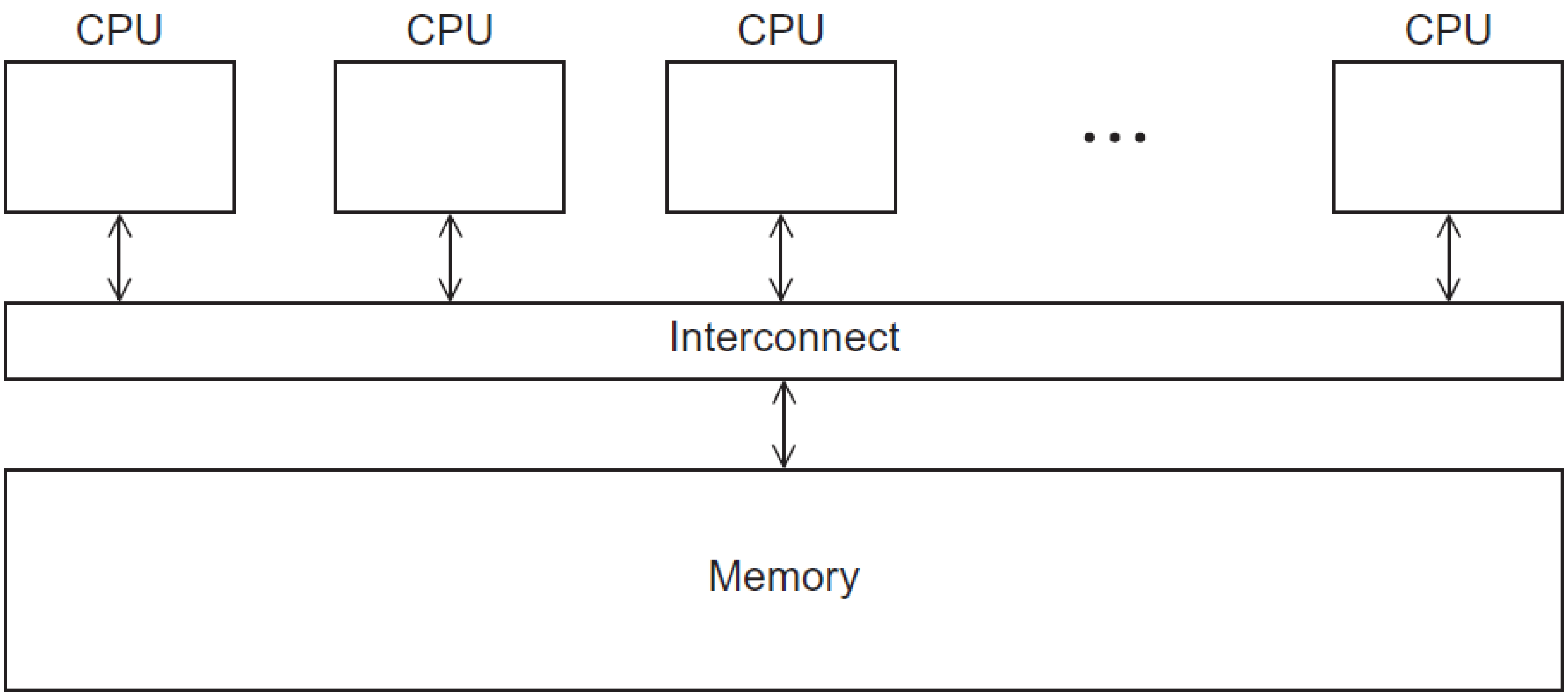
Roadmap

- Writing programs that use OpenMP.
- Using OpenMP to parallelize many serial ***for*** loops with only small changes to the source code.
- Task parallelism.
- Explicit thread synchronization.
- Standard problems in shared-memory programming.

OpenMP

- An API for shared-memory parallel programming.(http://openmp.org/wp/)
- 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.

A shared memory system



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
```

```c
#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 */
```

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

compiling

```
./omp_hello 4
```

running with 4 threads

possible outcomes

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

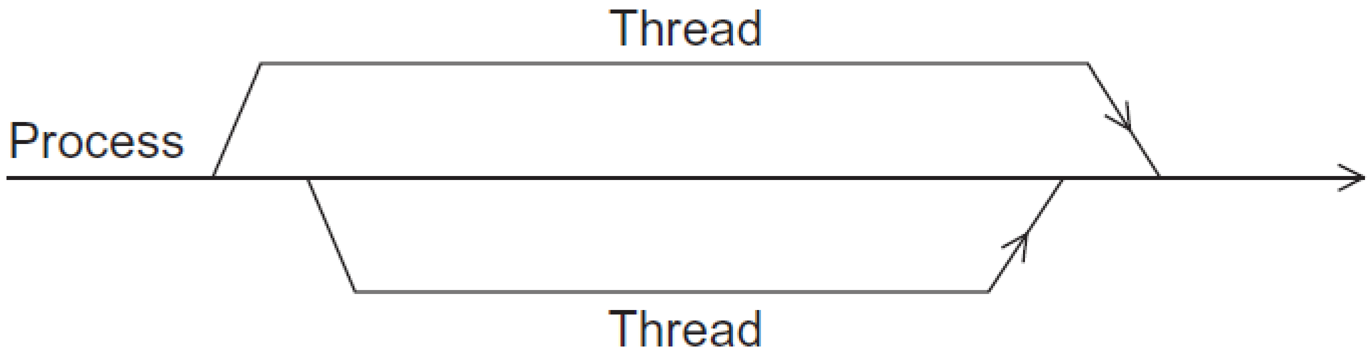
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

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

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.
  - If there are no other threads started, the system will typically run one thread on each available core.

A process forking and joining two threads



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 )
```

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.

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.

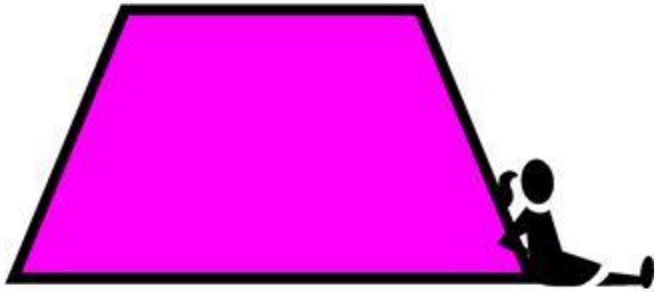
In case the compiler doesn't support OpenMP

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

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

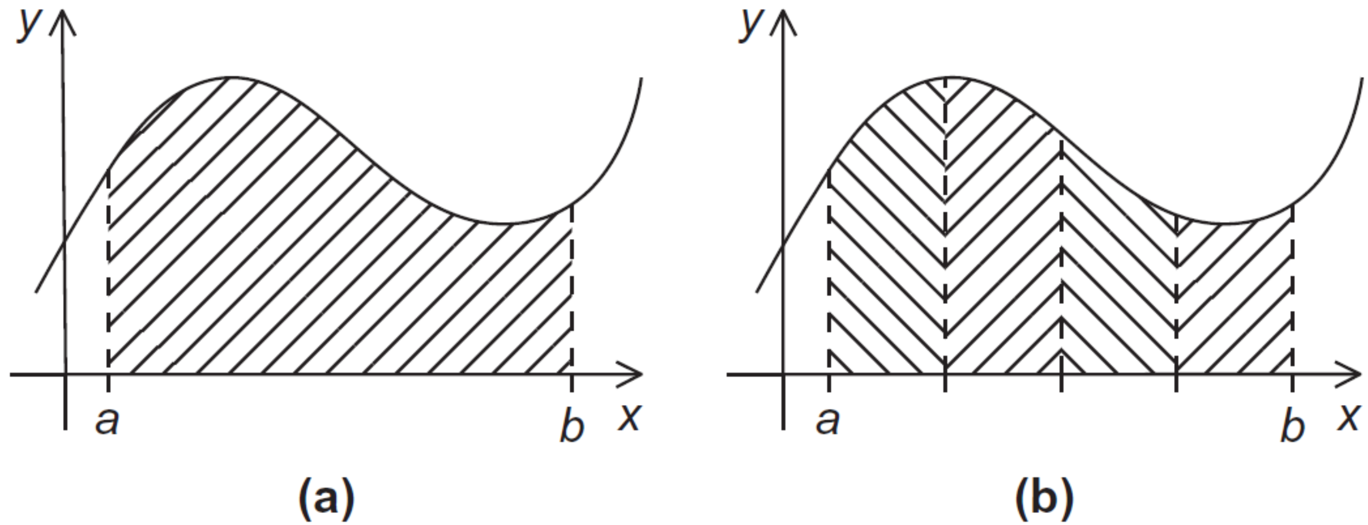
In case the compiler doesn't support OpenMP

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



THE TRAPEZOIDAL RULE

The trapezoidal rule



$$h = \frac{b-a}{n}$$

$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})].$$

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

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*approx;
```

A First OpenMP Version

PCAM method:

1) We identified two types of tasks:

t1: computation of the areas of individual trapezoids, and

t2: adding the areas of trapezoids.

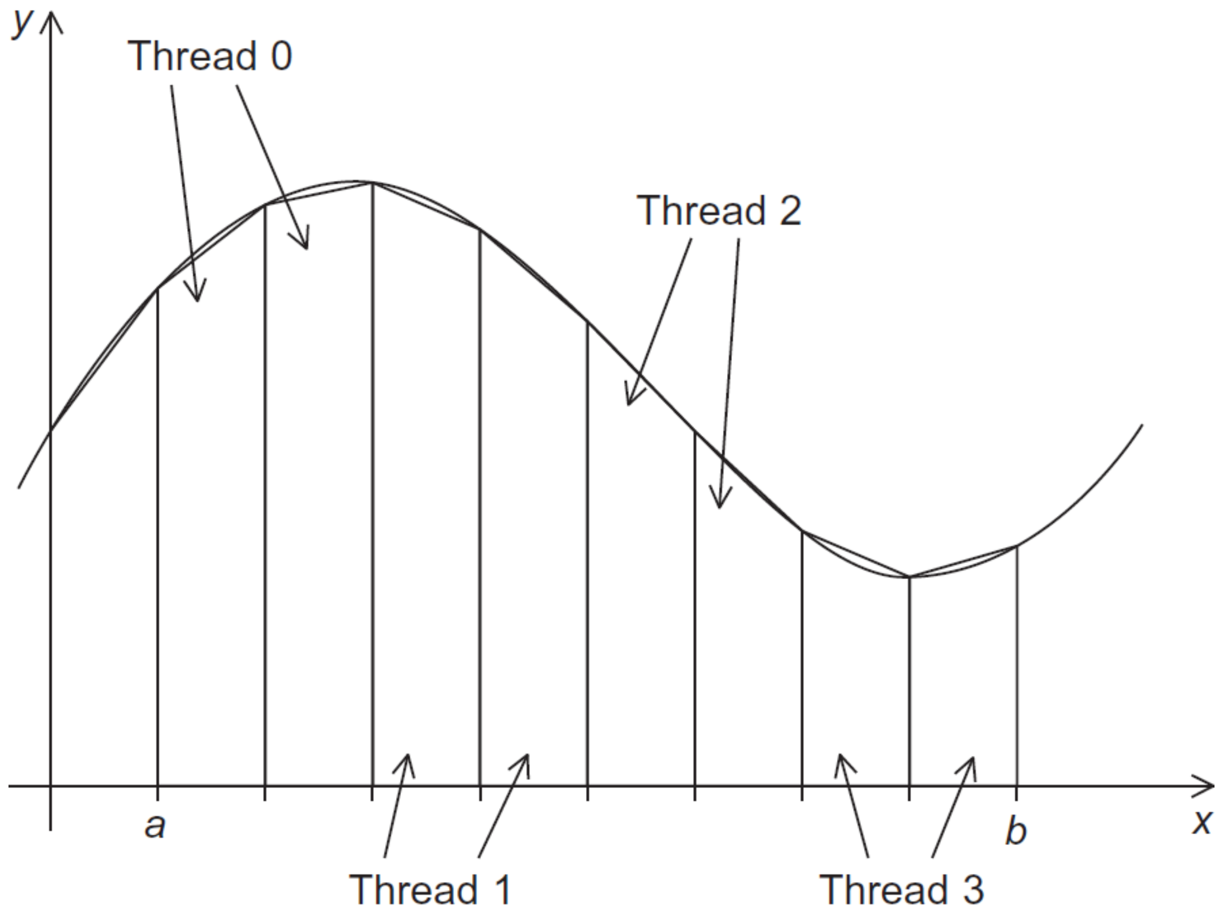
2) There is no communication among the tasks in t1, but each task in t1communicates with t2.

A First OpenMP Version

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



global_result += my_result ;

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:

This is an example of **Race Condition**

Mutual exclusion

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

critical section

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

```
#include <stdio.h>
#include <stdlib.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;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  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();
   int thread_count = omp_get_num_threads();

   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 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);
```

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 don’t like it.
Neither do I.
I think we can do better.

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.

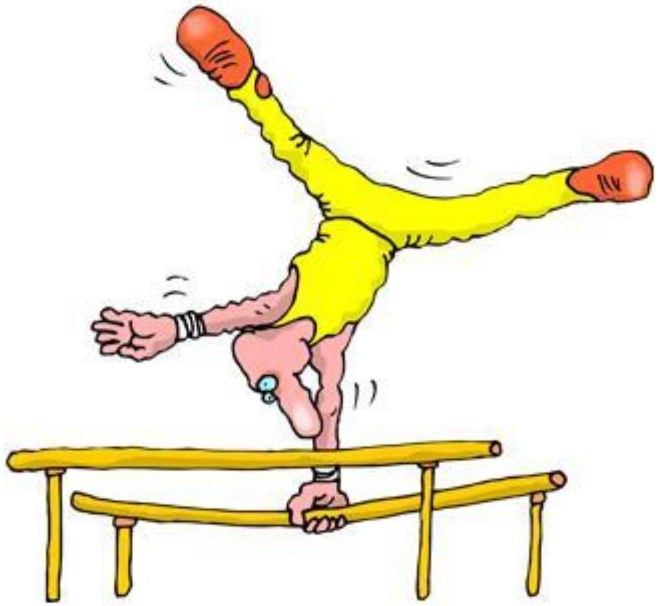
```
reduction(<operator>: <variable list>)
```

+, *, -, &, |, ˆ, &&, ||

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

note…

- The use of subtraction is a bit problematic, since subtraction isn't associative or commutative.
- If a reduction variable is a float or a double, the results may differ slightly when different numbers of threads are used (floating point arithmetic isn't associative).

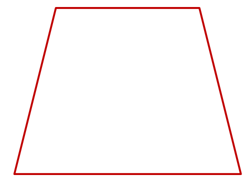


THE “PARALLEL FOR” DIRECTIVE

Parallel for

- OpenMP can parallelize a serial program that consists of ***for*** loop by "just" adding a **parallel for** directive

```c
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;
```



```c
   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;
```

note…

- the structured block following the **parallel for** directive must be a for loop.
- Forks a team of threads to execute the following structured block.
- the system parallelizes the for loop by dividing the iterations of the loop among the threads.
- with a **parallel for** directive, the default scope of the loop variable is ***private***
- Both **parallel** directive and **parallel for** directive have an implicit barrier at the end of the loop

Caveats

- It seems to be too easy, but…
  - only parallelize ***for*** loops. It won't parallelize ***while*** loops or ***do-while*** loops.
  - only parallelize **for loops** that are in **canonical form**
    - ***for*** loop is a structured block
    - the number of iterations can be determined from the ***for*** statement itself and prior to execution of the loop

Legal forms for parallelizable for statements

$$
\textbf{for}\left(\begin{array}{lllll}
 & & & & \texttt{index++} \\
 & & & & \texttt{++index} \\
 & & \texttt{index < end} & & \texttt{index--} \\
 & & \texttt{index <= end} & & \texttt{--index} \\
\texttt{index = start} & ; & \texttt{index >= end} & ; & \texttt{index += incr} \\
 & & \texttt{index > end} & & \texttt{index -= incr} \\
 & & & & \texttt{index = index + incr} \\
 & & & & \texttt{index = incr + index} \\
 & & & & \texttt{index = index - incr}
\end{array}\right)
$$

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.

Caveats

- 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.
- If a *for* loop fails to satisfy one of the above rules, the compiler will simply reject it.

Examples

```
1   int Linear_search(int key, int A[], int n) {
2      int i;
3      /* thread_count is global */
4   #  pragma omp parallel for num_threads(thread_count)
5      for (i = 0; i < n; i++)
6         if (A[i] == key) return i;
7      return -1;  /* key not in list */
8   }
```

The gcc compiler reports:

Line 6: error: invalid exit from OpenMP structured block

Data dependencies

- The dependence of the computation of between data is called a **data dependence.**
- A loop in which the results of one or more iterations depend on other iterations. The dependence is called a **loop-carried dependence.**

Data dependencies

- An example of computing the first n **fibonacci** numbers
- Fibonacci Sequence
  - 0、1、1、2、3、5、8、13、21、……
  - $F_0=0$，$F_1=1$，$F_n=F_{n-1}+F_{n-2}$, (n>=2, n∈N*)

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 ];
```

but sometimes we get this

1 1 2 3 5 8 13 21 34 55

this is correct

1 1 2 3 5 8 0 0 0 0

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 dependencies

- Another example

```
for (i = 0; i < n; i++) {
    x[i] = a + i*h;
    y[i] = exp(x[i]);
}
```

There is a data dependence between computations of y[i] and x[i].

```
#  pragma omp parallel for num_threads(thread_count)
   for (i = 0; i < n; i++) {
      x[i] = a + i*h;
      y[i] = exp(x[i]);
   }
```

No problem. Why?

Data dependencies

- we only need to worry about loop-carried dependences.
- How to detect a loop-carried dependence?
  - We should look for variables that are read or written in one iteration, and written in another.

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;
```

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

replace the code

```
sum += factor/(2*k+1);
factor = -factor;
```

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

OpenMP solution #2

```
        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 is shared

OpenMP solution #2

```
      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 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);
    }
```

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;
        }
```

loop dependency:

- The outer Loop
- The inner loop

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]) Swap(&a[i−1],&a[i]);
   else
      for (i = 1; i < n−1; i += 2)
         if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

- Parallelizing the outer for loop isn't an option
- The inner for loops don't have any loop-carried dependences

Odd-Even Transposition Sort

Phase	Subscript in Array 0		1		2		3
0	9	$\leftrightarrow$	7		8	$\leftrightarrow$	6
	7		9		6		8
1	7		9	$\leftrightarrow$	6		8
	7		6		9		8
2	7	$\leftrightarrow$	6		9	$\leftrightarrow$	8
	6		7		8		9
3	6		7	$\leftrightarrow$	8		9
	6		7		8		9

First OpenMP Odd-Even Sort

```
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        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
#        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;
            }
         }
   }
```

The OpenMP implementation *may* fork and join thread count threads on *each* pass

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;
            }
         }
   }
```

- Fork our team of thread count threads *before* the outer loop with a parallel directive.
- Use a for directive, which tells OpenMP to parallelize the for loop with the existing team of threads.

Odd-even sort with two parallel for directives and two for directives.
(Times are in seconds.)

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

SCHEDULING LOOPS

We want to parallelize this loop.

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

Thread	Iterations
0	$0,\ n/t,\ 2n/t,\ \ldots$
1	$1,\ n/t+1,\ 2n/t+1,\ \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1,\ n/t+t-1,\ 2n/t+t-1,\ \ldots$

Assignment of work
using cyclic partitioning.

```
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*.

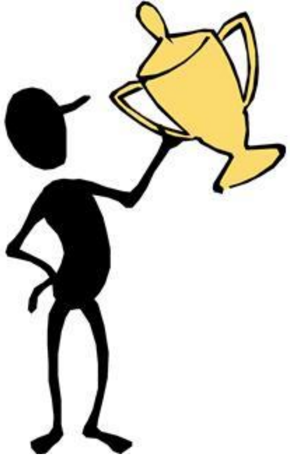
Results

- 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.

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



In OpenMP, assigning iterations to threads is called **scheduling**, and the schedule clause can be used to assign iterations in either a **parallel for** or a **for** directive.

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);
```

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 schedule is determined at run-time.
- The chunksize is a positive integer.

The Static Schedule Type

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

```
schedule(static,1)
```

$$\begin{aligned} \text{Thread } 0: &\quad 0, 3, 6, 9 \\ \text{Thread } 1: &\quad 1, 4, 7, 10 \\ \text{Thread } 2: &\quad 2, 5, 8, 11 \end{aligned}$$

The Static Schedule Type

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

```
schedule(static,2)
```

$$
\begin{array}{ll}
\text{Thread } 0: & 0, 1, 6, 7 \\
\text{Thread } 1: & 2, 3, 8, 9 \\
\text{Thread } 2: & 4, 5, 10, 11
\end{array}
$$

The Static Schedule Type

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

```
schedule(static,4)
```

$$\begin{array}{ll} \text{Thread } 0: & 0,1,2,3 \\ \text{Thread } 1: & 4,5,6,7 \\ \text{Thread } 2: & 8,9,10,11 \end{array}$$

The chunksize can be omitted. If it is omitted, the chunksize is approximately total_iterations/thread_count.

The Dynamic Schedule Type

- The iterations are also broken up into chunks of chunksize consecutive iterations.
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- This continues until all the iterations are completed.
- The chunksize can be omitted. When it is omitted, a chunksize of 1 is used.

The Guided Schedule Type

- Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
- However, in a guided schedule, as chunks are completed the size of the new chunks decreases.
- If no chunksize is specified, the size of the chunks decreases down to 1.
- If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.

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.

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.
- Suppose we have a parallel **for** directive in a program and it has been modified by schedule(runtime).

```
$ export OMP_SCHEDULE="static,1"
```

About Schedule

- There *is* some overhead associated with the use of a schedule clause.
- The overhead is greater for dynamic schedules than static schedules.
- The overhead associated with guided schedules is the greatest of the three.
- If we're getting satisfactory performance without a schedule clause, we should go no further

About Schedule

- If each iteration of the loop requires roughly the same amount of computation, then it’s likely that the default distribution will give the best performance.
- If the cost of the iterations decreases (or increases) linearly as the loop executes, then a static schedule with small chunksizes will probably give the best performance.
- If the cost of each iteration can’t be determined in advance, then it may make sense to explore a variety of scheduling options.
  - The schedule(runtime) clause can be used here, and the different options can be explored by running the program with different assignments to the environment variable OMP_SCHEDULE.

PRODUCERS AND CONSUMERS

Queues

- A parallel problem that isn't amenable to parallelization using a parallel **for** or **for** directive
- Can be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket.
- A natural data structure to use in many multithreaded applications.
- 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.

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

Enqueueing a message forms a critical section

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

Receiving Messages

If there are at least two messages in the queue, a call to Dequeue can't possibly conflict with any calls to Enqueue, To avoid this, we can implement Try receive as follows:

```
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);
```

Termination Detection

Done function:

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

May cause problems, add a counter done sending, and each thread increments this after completing its for loop.

```
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
```

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 >>

  Notes:<expression> must not reference x.
- 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.

Examples

```
#       pragma omp atomic
        x += y++;
```

- A thread's update to x will be completed before any other thread can begin updating x.
- However, the update to y may be unprotected and the results may be unpredictable.

Critical Sections

- OpenMP treats all critical blocks as part of one composite critical section
- 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.
  - **simple** locks and **nested** locks
  - an OpenMP simple lock is omp_lock_t
  - an OpenMP nested lock is omp_nest_lock_t

```
void omp_init_lock(omp_lock_t*    lock_p   /* out */);
void omp_set_lock(omp_lock_t*     lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*   lock_p   /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p   /* in/out */);
```

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);
```

Comparison of critical directives, atomic directives and locks

- In general, the atomic directive has the potential to be the fastest
- However, the OpenMP specification allows the atomic directive to enforce mutual exclusion across *all* atomic directives in the program

```
#  pragma omp atomic            #  pragma omp atomic
   x++;                            y++;
```

  - Even if x and y are unrelated memory locations, it's possible that if one thread is executing x++, then no thread can simultaneously execute y++.

- Little difference between the performance of critical sections protected by a critical directive, and critical sections protected by locks
- If you can’t use an atomic directive, but you can use a critical directive, you probably should
- The use of locks should probably be reserved for situations in which mutual exclusion is needed for a data structure rather than a block of code.

Some Caveats

1. You shouldn't mix the different types of mutual exclusion for a single critical section.

```
#  pragma omp atomic          #  pragma omp critical
   x += f(y);                    x = g(x);
```

2. There is no guarantee of fairness in mutual exclusion constructs.

```
while(1) {
   . . .
#     pragma omp critical
      x = g(my_rank);
   . . .
}
```

- Some thread 1 can block forever waiting to executex = g(my rank), while the other threads repeatedly execute the assignment.

3.It can be dangerous to "nest" mutual exclusion constructs.

```
#   pragma omp critical
    y = f(x);
    . . .
    double f(double x) {
#      pragma omp critical
       z = g(x);  /* z is shared */
       . . .
    }
```

deadlock

```
#   pragma omp critical(one)
    y = f(x);
    . . .
    double f(double x) {
#      pragma omp critical(two)
       z = g(x);  /* z is global */
       . . .
    }
```

- However, it’s not difficult to come up with examples when naming won’t help.

Time	Thread u	Thread v
0	Enter crit. sect. one	Enter crit. sect. two
1	Attempt to enter two	Attempt to enter one
2	Block	Block

Caches

- Recall that chip designers have added blocks of relatively fast memory to processors called cache memory.
- The design of cache memory takes into consideration the principles of *temporal and spatial locality*: if a processor accesses main memory location *x* at time *t*, then it is likely that at times close to *t*, it will access main memory locations close to *x*.
- The use of cache memory can have a huge impact on shared-memory.

Cache-Coherence

- **Cache coherence:** When the caches of multiple processors store the same variable, an update by one processor to the cached variable is "seen" by the other processors.
- Cache coherence is enforced at the "cache-line level." That is, each time any value in a cache line is written, if the line is also stored in another processor's cache, the entire *line* will be invalidated

False Sharing

- Suppose two threads with separate caches access different variables that belong to the same cache line.
- Further suppose at least one of the threads updates its variable. Then even though neither thread has written to a variable that the other thread is using, the cache controller invalidates the entire cache line and forces the threads to get the values of the variables from main memory.
- 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, hence the name *false sharing*.

- Pragmas

```
__declspec (align(64)) int thread1_global_variable;
```

- Padding

```
struct ThreadParams
{
  // For the following 4 variables: 4*4 = 16 bytes
  unsigned long thread_id;
  unsigned long v; // Frequent read/write access variable
  unsigned long start;
  unsigned long end;
  // expand to 64 bytes to avoid false-sharing
  // (4 unsigned long variables + 12 padding)*4 = 64
  int padding[12];
};
```

Thread-local copies of data

```
struct ThreadParams
    {// For the following 4 variables: 4*4 = 16 bytes
        unsigned long thread_id;
        unsigned long v; //Frequent read/write access variable
        unsigned long start;
        unsigned long end;
    };
    void threadFunc(void *parameter)
    {
        ThreadParams *p = (ThreadParams*) parameter;
        // local copy for read/write access variable
        unsigned long local_v = p->v;
        for(local_v = p->start; local_v < p->end; local_v++)
        {                    // Functional computation            }
        p->v = local_v;  // Update shared data structure only once
    }
```

Cache-Miss

- A write-miss occurs when a core tries to update a variable that's not in cache, and it has to access main memory.
- A read-miss occurs when a core tries to read a variable that's not in cache, and it has to access main memory.

Matrix-vector multiplication

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

$$\begin{array}{|c|c|c|c|}\hline a_{00} & a_{01} & \cdots & a_{0,n-1} \\ \hline a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline \end{array} \quad \begin{array}{|c|}\hline x_0 \\ \hline x_1 \\ \hline \vdots \\ \hline x_{n-1} \\ \hline \end{array} = \begin{array}{|c|}\hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline \end{array}$$

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

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];
    }
```

Run-times and efficiencies of matrix-vector multiplication (times are in seconds)

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	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

write-misses

read-misses of x

Analysis

- When the program is run with the 8, 000,000*8 input, it has far more cache write-misses than either of the other inputs(Line 4)
- When the program is run with the 8*8,000,000 input, it has far more cache read-misses than either of the other inputs(Line 6)
- The two-thread efficiency of the program with the 8*8,000,000 input is more than 20% less than the efficiency of the program with the 8,000,000*8 and the 8000*8000 inputs. So do the four-thread efficiency.

- Four threads: with the 8*8,000,000 input, each thread is assigned two components
- Cache line in our example is 64 bytes. a single cache line will store eight **double**s.
- As cache coherence is enforced at the “cache-line level, every write to some element of y will invalidate the line in the other processor’s cache.
- Each thread will update each of its components 8,000,000 times.
- Due to the update, all the threads will have to reload y *many* times ---- **False Sharing** appear

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) \
      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(lines[i], " \t\n");
         while ( my_token != NULL ) {
            printf("Thread %d > token %d = %s\n", my_rank, j, my_token);
            my_token = strtok(NULL, " \t\n");
            j++;
         }
      } /* for i */
   } /* omp parallel */

}  /* Tokenize */
```

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

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.

Concluding Remarks (4)

- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.