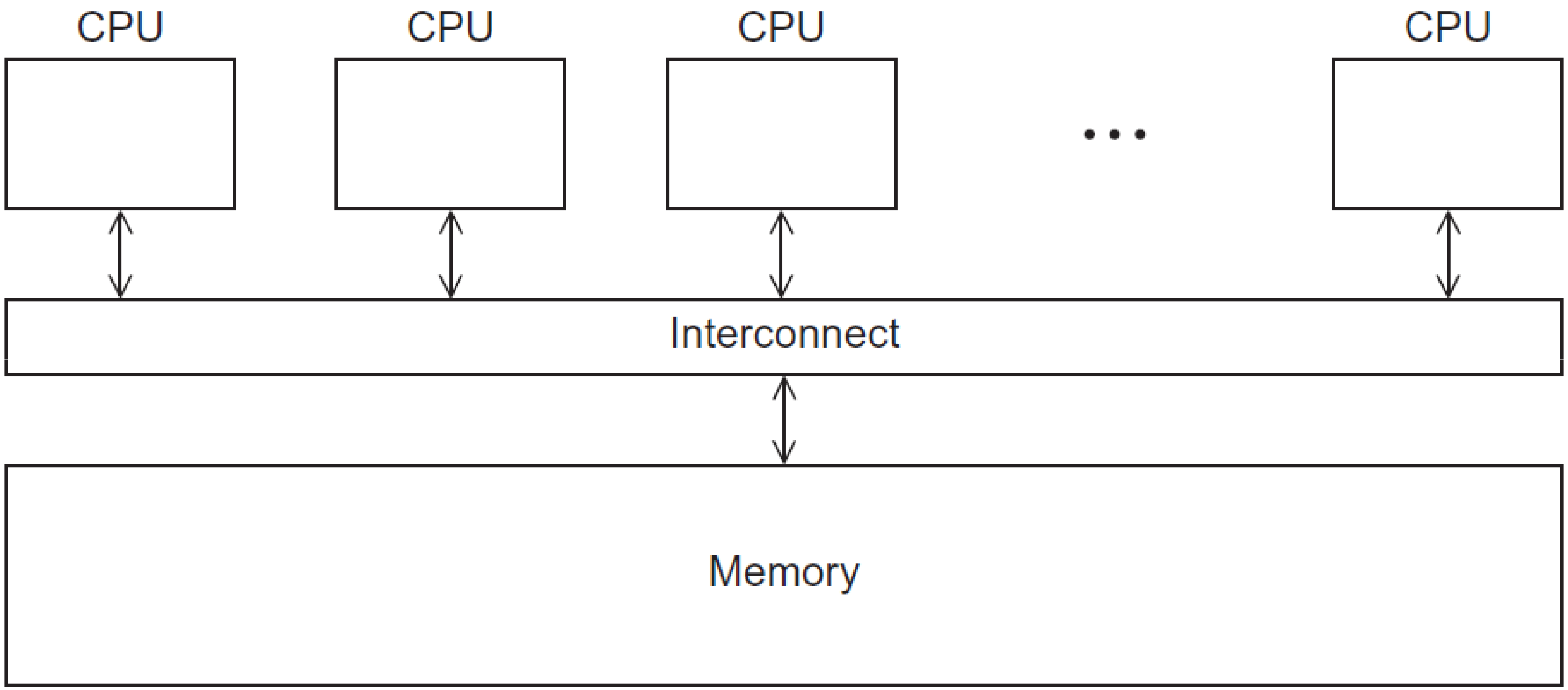
Unit-III

D.Venkata Vara Prasad

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.
- Similar to Pthreads

A shared memory system



OpenMP Vs Pthreads

- OpenMP & Pthreads are APIs for shared-memory programming
- The differences:

1. Pthreads requires that the programmer explicitly specify the behavior of each thread.

   OpenMP, allows the programmer to simply state a block of code should be executed in parallel, and the precise determination of the tasks and which thread should execute them is left to the compiler and the run-time system.

OpenMP Vs Pthreads

2. Pthreads is a library of functions that can be linked to a C program, so any Pthreads program can be used with any C compiler, provided the system has a Pthreads library

. OpenMP, requires compiler support for some operations, and hence it's possible that you may run across a C compiler that can't compile OpenMP programs into parallel programs

OpenMP Pogramming

- Designed to allow programmers to incrementally parallelize existing serial programs
- This is virtually impossible with MPI and fairly difficult with Pthreads.
- OpenMP provides a “directives-based” shared-memory API

Pragmas

- Special preprocessor instructions.
- 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.
- Pragmas in C and C++ start with

  # pragma

- **Pragmas** (preprocessor directives) are, one line in length, if a pragma won't fit on a single line, the newline needs to be "escaped" -i.e preceded by a backslash n
- Example, a "hello, world" program

example

```
#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 */
```

example

- To compile this with gcc we need to include the -*fopenmp option*
- To run the program, we specify the number of threads on the command line

example

$ 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

example

- In addition to a collection of directives, OpenMP consists of a library of functions and macros.
- The OpenMP header file is *omp.h.*
- we need to specify the number of threads on the command line.
- Line 9 we use the *strtol* function from *stdlib.h* to get the number of threads.
- In Line 11the first OpenMP directive to specify that the program should start some threads.

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.
- Threads are started or **forked** by a **process,** and they share the resources of the process that starts
- Ex: access to ***stdin*** and ***stdout***—but each thread has its own stack and program counter

clause

- We'll specify the number of threads on the command line
- 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 )
```

clause

- 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.

If compiler doesn't support OpenMP

- If the compiler doesn't support OpenMP, it will just ignore the parallel directive
- Attempt to include ***omp.h*** and the calls to ***omp_ get_ thread_num*** and ***omp_ get_ num_ threads*** will cause errors.
- check whether the preprocessor macro ***-OPENMP*** is defined.
- If *OPENMP* is defined we include omp.h and make calls to openmp functions.

If compiler doesn't support OpenMP

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

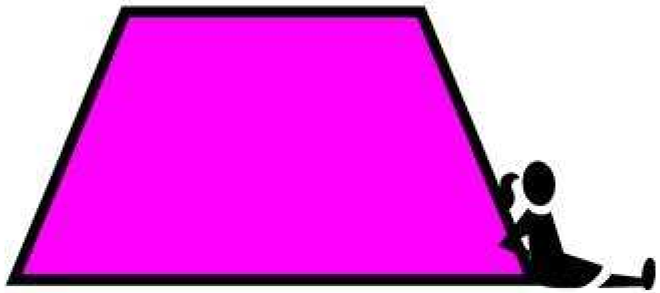
```
#ifdef _OPENMP
# include <omp.h>
#endif
```

If compiler doesn't support OpenMP

Also, instead of just calling the OpenMP functions, we can first check whether -OPENMP is defined

```
# 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
```

- If OpenMP isn't available, we assume that the Hello function will be single threaded



The Trapezoidal Rule

The Trapezoidal Rule

- The trapezoidal rule for estimating the area under a curve.
- If $y = f(x)$ is a function, and $a < b$ are real numbers, then we can
- Estimate the area between the graph of $f(x)$, the vertical lines $x = a$ and $x = b$,
- The x-axis by dividing the interval **[a,b] into n** subintervals and approximating the area over each subinterval by the area of a trapezoid

Serial Algorithm

- If each subinterval has the same length and if we define

```
/* 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

1) We identified two types of tasks:

   a) computation of the areas of individual trapezoids, and

   b) adding the areas of trapezoids.

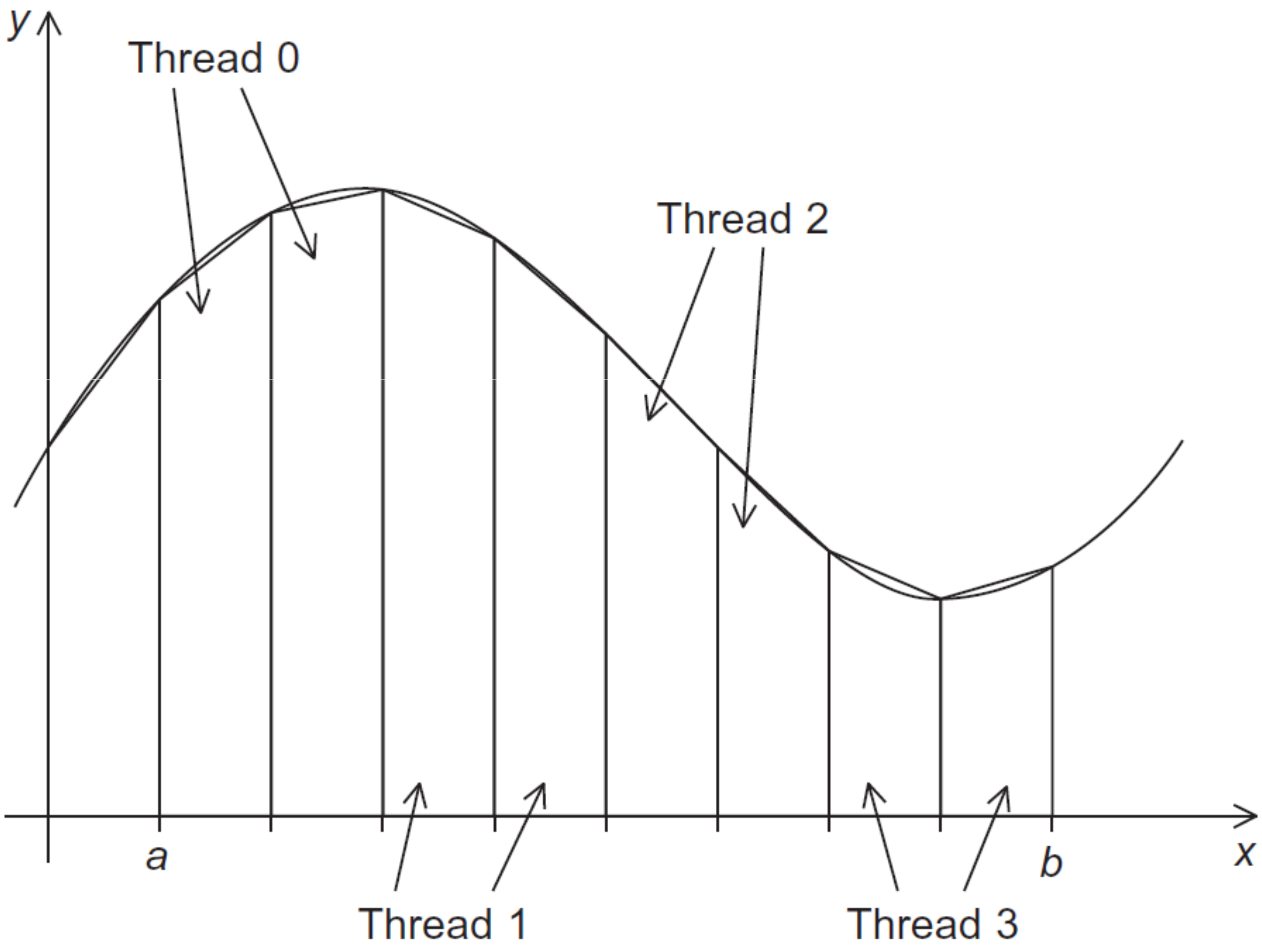
2) There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1b.

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).
- This partitioned the interval [a,b] into larger subintervals, and each thread simply applied the serial trapezoidal rule to its subinterval as shown Fig below.

Assignment of trapezoids to threads



Assignment of trapezoids to threads

- We use a shared variable for the sum of all the threads' results.
- Each thread can add its (private) result into the shared variable.

```
global_result += my result;
```

Assignment of trapezoids to threads

- This can result in an erroneous value for global result.
- If two (or more) threads attempt to execute this statement simultaneously, the result will be unpredictable.

Assignment of trapezoids to threads

- Ex: suppose that global_result has been initialized to 0, thread 0 has computed my_result = 1, and thread 1 has computed my result = 2. Furthermore, suppose that the threads execute the
- statement global_result += my_result according to the following timetable:

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 ;

- We see that the value computed by thread 0 (my result = 1) is overwritten by thread 1.

Assignment of trapezoids to threads

```
# pragma omp critical
global result += my result;
```

- This directive tells that the system needs to arrange the threads to have mutually exclusive access to the following structured block of code.
- Only one thread can execute the following structured block at a time.

Assignment of trapezoids to threads

- In Line 16 the ***parallel*** directive specifies that the ***Trap*** function should be executed by ***thread_count*** threads.
- In the ***Trap*** function, each thread gets its rank and the total number of threads in the team started by the ***parallel*** directive.

Assignment of trapezoids to threads

- Each thread determines the following:

1. The length of the bases of the trapezoids (Line 32)
2. The number of trapezoids assigned to each thread (Line 33)
3. The left and right endpoints of its interval (Lines 34 and 35, respectively)
4. Its contribution to global_result (Lines 36–41)

- The threads finish by adding in their individual results to global_result in Lines 43 and 44

```c
#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;
   double  a, b;
   int     n;
   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 */

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

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.
- Ex: A variable declared at the beginning of a C function has “function-wide” scope, that is, it can only be accessed in the body of the function.
- A variable declared at the beginning of ***a .c*** file but outside any function has “file-wide” scope.

Scope in OpenMP

- The scope of a variable refers to the set of threads that can access the variable in a parallel block.
- 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

- 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 Clause

- Ex: if A is an array of ***n ints***, the computation.

```
int sum = 0;
for (i = 0; i < n; i++)
sum += A[i];
```

- is a reduction in which the reduction operator is addition

The Reduction Clause

- In OpenMP it is possible to specify that the result of a reduction is a reduction variable.
- A reduction clause can be added to a parallel directive.
- The syntax of the reduction clause is

  reduction(<operator>: <variable list>)

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

The Reduction Clause

- In our example, we can modify the code as follows:

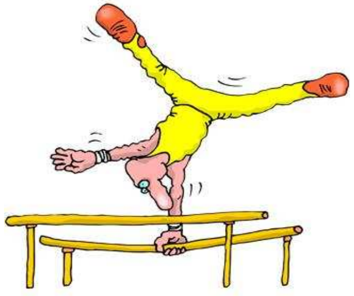
```
global_result = 0.0;
# pragma omp parallel num threads(thread count) \
    reduction(+: global result)
    global result += Local trap(double a, double b, int n);
```

The Reduction Clause

- When a variable is included in a reduction clause, the variable itself is shared.
- However, a private variable is created for each thread in the team.

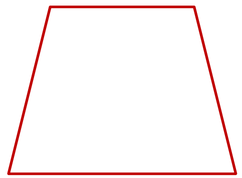
THE “PARALLEL FOR” DIRECTIVE

- 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.



THE “PARALLEL FOR” DIRECTIVE

```
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

- It is possible to parallelize a serial program that consists of one large ***for*** loop by just adding a single ***parallel for*** directive.

- It may be possible to incrementally parallelize a serial program that has many ***for*** loops by successively placing ***parallel for*** directives before each loop.

Caveats

- OpenMP will only parallelize ***for*** loops. It won't parallelize ***while loops*** or ***do-while*** loops
- OpenMP will only parallelize for loops that are in canonical form
- Loops in canonical form take one of the forms shown in fig below

Caveats

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

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

Data dependencies

- In loops in which the computation in one iteration depends on the results of one or more previous iterations.
- Ex: computes the first ***n*** fibonacci numbers:

Data dependencies

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

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 ];    parellel version
```

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

Data dependencies

- It appears that the run-time system assigned the computation of
- ***fibo[2], fibo[3], fibo[4],* and *fibo[5]*** to one thread
- while ***fibo[6], fibo[7]***, ***fibo[8],* and *fibo[9]*** were assigned to the other.

Data dependencies

- If the 1st thread completes its computation before other thread starts we get correct results.
- If the 1st thread not computed fibo(4), fibo(5) and the 2nd thread computes fibo(6) then the system has initialized 0's in fibo(4) and fibo(5).
- Then we may get wrong results as in case-2.

Data dependencies

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.

Estimating π

- One way to get a numerical approximation to π is

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

- We can implement this formula in ***serial code*** with

```
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 #1

- by default any variable declared before the loop—with the sole exception of the loop variable—is shared among the threads.

- So ***factor*** is shared and, for example, thread 0 might assign it the value 1, but before it can use this value in the update to ***sum***, thread 1 could assign it the value -1.

- to eliminating the loop-carried dependence in the calculation of ***factor***, we need to insure that each thread has its own copy of ***factor***.

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);
        }
```

OpenMP solution #2

- The ***private*** clause specifies that for each variable listed inside the parentheses, a private copy is to be created for each thread.
- Each of the thread will have its own copy of the variable factor.
- The updates of one thread to factor won’t affect the value of factor in another thread.

More About Loops in OpenMP: Sorting

Bubble sort

- Serial Bubble sort algorithm

```
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;
        }
```

Bubble sort

- stores ***n*** ints and the algorithm sorts them in increasing order
- The outer loop first finds the largest element in the list and stores it in a[n-1]
- then finds the next-to-the-largest element and stores it in a[n-2] and so on.
- the first pass is working with the full n-element list
- The second is working with
- all of the elements, except the largest; it’s working with an n-1-element list and so on.

Bubble sort

- Loop carried dependence is seen in the outer loop.
- the contents of the current iteration depends on the previous iteration of the outer loop.
- Loop carried dependence is seen in the inner loop also the elements compared in iteration i depends on the outcome of iteration i-1.

Serial Odd-Even Transposition Sort

- It is similar to bubble sort, but has more opportunities for parallelism.

```
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]);
```

Serial Odd-Even Transposition Sort

- The list a stores n integers, and the algorithm sorts them into increasing order.
- During an “even phase” (phase % 2 == 0), each odd-subscripted element, a[i], is compared to the element to its “left,” a[i-1], and if they’re out of order, they’re swapped.

Serial Odd-Even Transposition Sort

- During an “odd” phase, each odd-subscripted element is compared to the element to its right, and if they’re out of order, they’re swapped.

- A theorem guarantees that after ***n*** phases, the list will be sorted.

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

Serial Odd-Even Transposition Sort

Ex: suppose a = [9, 7, 8, 6].

- In phase 0 the inner loop will compare elements in the pairs .(9,7) and (8,6), and both pairs are swapped
- For phase 1 the list should be [7, 9, 6, 8]
- In phase 1 (9,6) should be compared and swapped

Odd-Even Transposition Sort
First OpenMP version

```
   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;
            }
         }
   }
```

Serial Odd-Even Transposition Sort First OpenMP version

- The inner ***for*** loops, however, don't appear to have any loop-carried dependences.

- Ex: In an even phase loop, variable i will be odd, so for two distinct values of i, say i = j and i = k, the pairs (j-1, j) and (k-1, k) will be be disjoint.

- The comparison and possible swaps of the pairs (a[j-1], a[j]) and (a[k-1], a[k]) can therefore proceed simultaneously.

Serial Odd-Even Transposition Sort
First OpenMP version

- Issues:
- ***parallel for*** directive has an implicit barrier at the end of the loop, so none of the threads will proceed to the next phase, ***phase p+1***, until all of the threads have completed the current phase***, phase p***.

- overhead associated with forking and joining the threads.

Second OpenMP Odd-Even Sort

- We can 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. This modification to the original OpenMP implementation is shown below.

- The ***for*** directive, unlike the ***parallel for*** directive, doesn’t fork any threads.

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;
            }
         }
   }
```

Second OpenMP Odd-Even Sort

- shows run-times for 1, 2, 3, and 4 threads on one of our systems when the input list contained 20,000 elements.
- Run-times for this second version of odd-even sort are in the second row of Table .

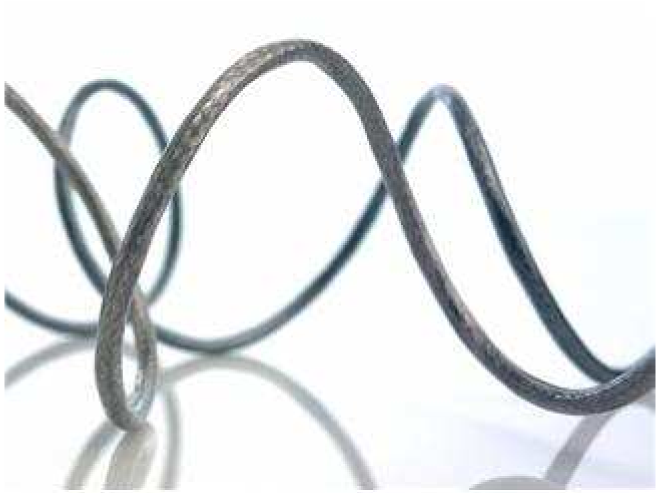
Second OpenMP Odd-Even Sort

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

Second OpenMP Odd-Even Sort

- When we're using two or more threads, the version that uses two ***for*** directives is at least 17% faster than the version that uses two ***parallel for*** directives.



Scheduling Loops

Scheduling Loops

- Most OpenMP implementations use roughly a block partitioning.

- If there are ***n*** iterations in the serial loop

- In the parallel loop the first n/ thread_count are assigned to thread 0

- The next n/ thread_count are assigned to thread 1, and so on.

The Schedule Clause

- Default Schedule
- we just add a ***parallel*** for directive with a ***reduction*** clause:

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

The Schedule Clause

- Cyclic schedule
- To get a cyclic schedule, we can add a schedule clause to the ***parallel*** for directive

```
    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

- The system assigns chunks of ***chunksize*** iterations to each thread in a round-robin fashion.
- As an example, suppose we have 12 iterations, 0, 1, : : : ,11, and three threads.
- Then if ***schedule(static,1)*** is used in the ***parallel for*** or ***for*** directive, we've already seen that the iterations will be assigned as

The Static Schedule Type

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

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

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

The Static Schedule Type

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

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

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

The Static Schedule Type

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

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

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

The Dynamic Schedule Type

- The iterations are 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 executes a chunk, and when a thread finishes a chunk, it requests another one.
- 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

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

Producers and Consumers

Queues

- A Queue is a list abstract data type in which new elements are inserted at the “rear” of the queue and elements are removed from the “front” of the queue.
- A queue can thus be viewed as an abstraction of a line of customers waiting to pay for their groceries in a supermarket.
- The elements of the list are the customers

Queues

- When a new entry is added to the rear of a queue, we sometimes say that the entry has been "enqueued."
- when an entry is removed from the front of a queue, we sometimes say that the entry has been "dequeued."

Queues

- 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

- A natural application is implementing message-passing on a shared memory system.
- 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();
```

Message-Passing

- The user specify the number of messages each thread should send.
- When a thread is done sending messages, it receives messages until all the threads are done, at which point all the threads quit.

Sending Messages

- Accessing a message queue to ***enqueue*** a message is a critical section.
- We need to have a variable that keeps track of the rear of the queue.
- When we ***enqueue*** a new message, we'll need to check and update the rear pointer.

Sending Messages

- If two threads try to do this simultaneously, we may lose a message that has been enqueued by one of the threads.
- The results of the two operations will conflict, and hence enqueueing a message will form a critical section.

Sending Messages

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

Receiving Messages

- The synchronization issues for receiving a message are a little different.
- Only the owner of the queue (that is, the destination thread) will ***dequeue*** from a given message queue.

Receiving Messages

- If we store two variables, enqueued and dequeued, then the number of messages in the queue is
- 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

- The implementation of the Done function specifies the termination.

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

- The above code leads to some problem

Termination Detection

- If thread u executes this code, it's entirely possible that some thread—call it thread v—will send a message to thread u after u has computed queue size = 0.
- After thread u computes queue size = 0, it will terminate and the message sent by thread v will never be received.
-

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

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.

Critical Sections

- Since enforcing mutual exclusion among threads serializes execution.
- This behavior of OpenMP—treating all critical blocks as part of one composite critical section.
- OpenMP does provide the option of adding a name to a critical directive

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

Critical Sections

- 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

- The ***lock*** data structure is shared among the threads that will execute the critical section.
- Before a thread enters the critical section, it attempts to set or lock the ***lock*** data structure by calling the lock function.
- . When the thread finishes the code in the critical section, it calls an ***unlock*** function, which relinquishes or unsets the lock and allows another thread to obtain the ***lock***.

Locks

- OpenMP has two types of locks: ***simple locks*** and ***nested locks***.
- A ***simple lock*** can only be set once before it is unset,
- A ***nested lock*** can be set multiple times by the same thread before it is unset.

Using Locks in the Message-Passing Program

- If we want to insure mutual exclusion in each individual message queue, we need to include a data member in our queue structure.

Using Locks in the Message-Passing Program

- When a thread tries to send or receive a message, it can only be blocked by a thread attempting to access the same message queue.
- since different message queues have different locks only one thread could send at a time, regardless of the destination.

CACHES, CACHE COHERENCE, & FALSE SHARING

Time	Event	Cache contents for CPU A	Cache contents for CPU B	Memory contents for location X
0				1
1	CPU A reads X	1		1
2	CPU B reads X	1	1	1
3	CPU A stores 0 into X	0	1	0

CACHES, CACHE COHERENCE

- two different processors have two different values for the same location.
- This difficulty is generally referred to as the *cache-coherence* problem.

Cache Coherence- Matrix-vector multiplication

- If A =(aij) is an mxn matrix and x is a vector with ***n*** components,
- The product y = Ax is a vector with ***m*** components.
- i^{th} component yi is found by forming the dot product of the i^{th} row of A with x:
-

Cache Coherence- Matrix-vector multiplication

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

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

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

Cache Coherence- Matrix-vector multiplication

- There are no loop-carried dependences in the outer loop,
- A and x are never updated and iteration i only updates y[i].
- we can parallelize this by dividing the iterations in the outer loop among the threads

Cache Coherence- 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];
    }
```

Cache Coherence- Matrix-vector multiplication

- If ***Tserial*** is the run-time of the serial program and ***Tparallel*** is the run-time of the parallel program.
- The efficiency E of the parallel program is the speedup S divided by the number of threads t

$$E = \frac{S}{t} = \frac{\left(\frac{T_{\text{serial}}}{T_{\text{parallel}}}\right)}{t} = \frac{T_{\text{serial}}}{t \times T_{\text{parallel}}}.$$

Where $S \leq t$, $E \leq 1$.

Cache Coherence- Matrix-vector multiplication

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

Cache Coherence- Matrix-vector multiplication

- The total number of floating point additions and multiplications is 64, 000, 000.
- It's clear that The 8, 000,000 x 8 system requires about 22% more time than the 8000 x 8000 system.
- The 8 x 8, 000,000 system requires about 26% more time than the 8000 x 8000 system.
- Both of these differences are at least partially attributable to cache performance