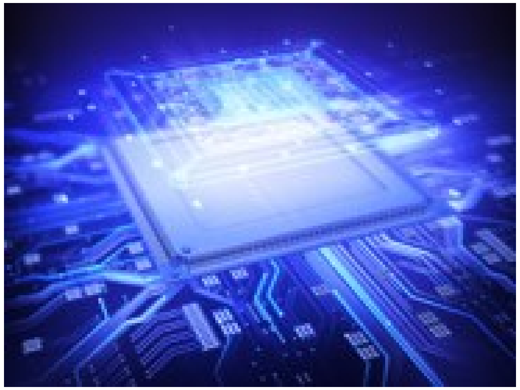
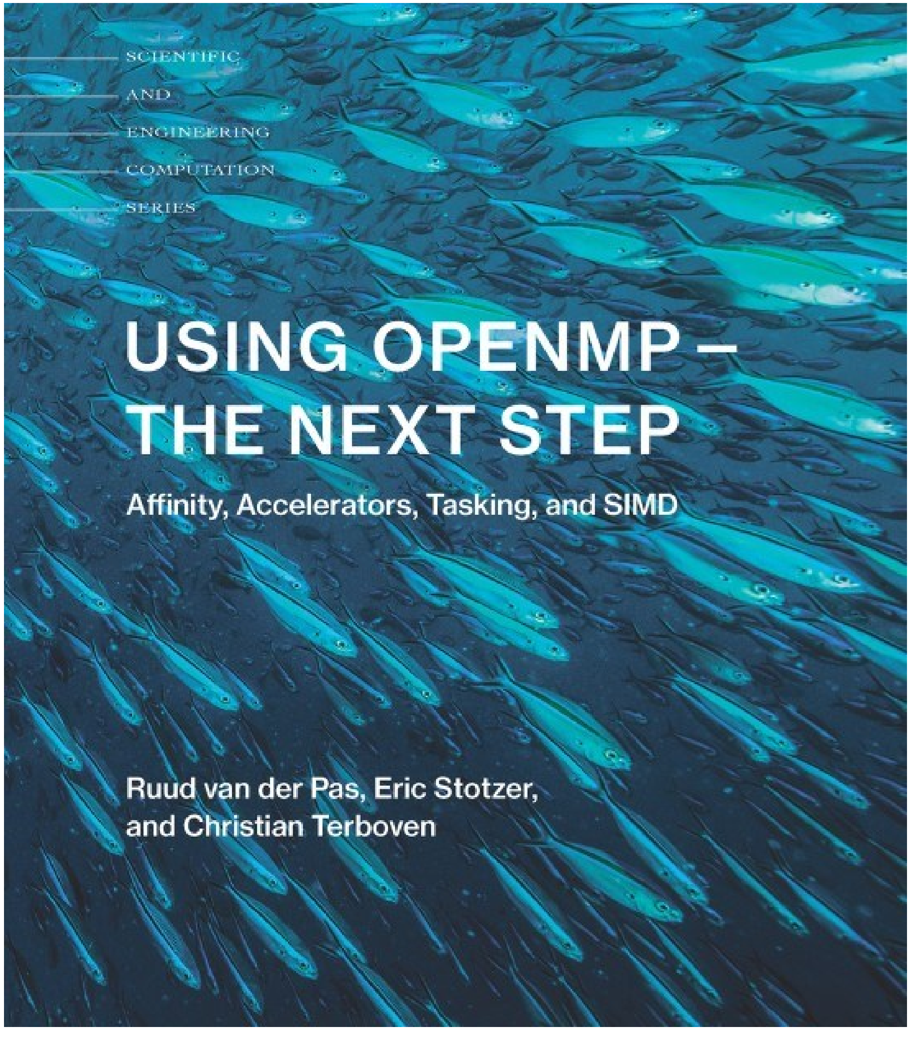
Multicore Processors: Architecture & Programming

OpenMP

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com



A Good Book


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES
USING OPENMP –
THE NEXT STEP
Affinity, Accelerators, Tasking, and SIMD
Ruud van der Pas, Eric Stotzer,
and Christian Terboven

Small and Easy Motivation

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


int main() {



  // Do this part in parallel


  printf( "Hello, World!\n" );


  return 0;
}
```

Small and Easy Motivation

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

int main() {

  omp_set_num_threads(16);

  // Do this part in parallel
  #pragma omp parallel
  {
    printf( "Hello, World!\n" );
  }

  return 0;
}
```

Simple!

Serial Program:

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

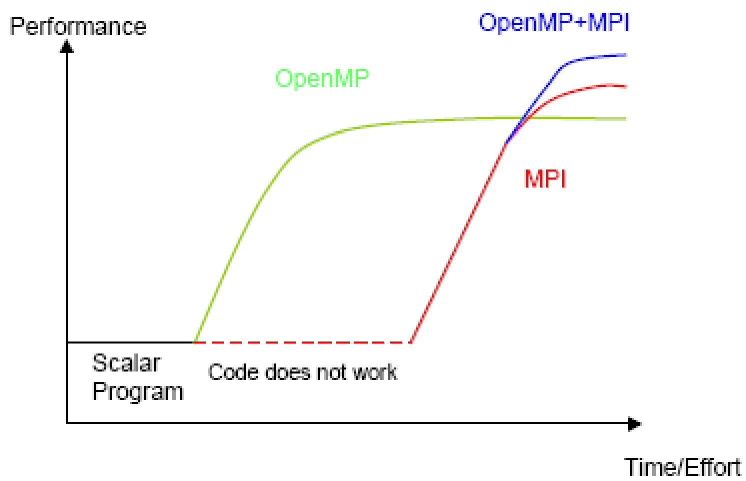
Parallel Program:

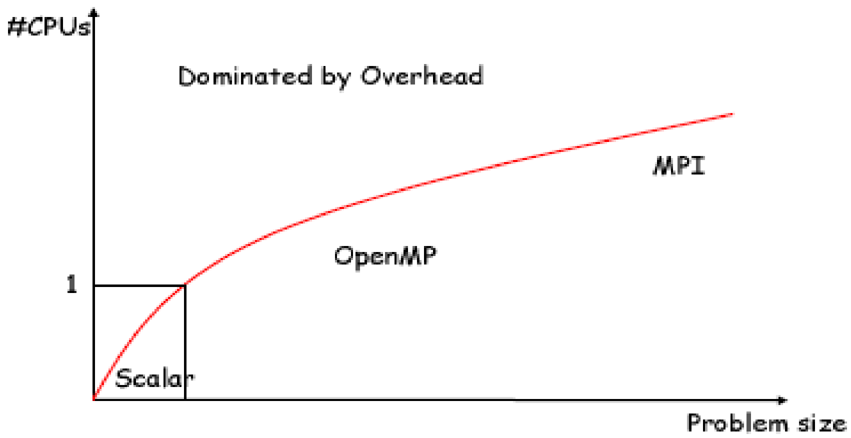
```
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence

OpenMP is a small API that hides cumbersome threading calls with simpler *directives*

Interesting Insights About OpenMP





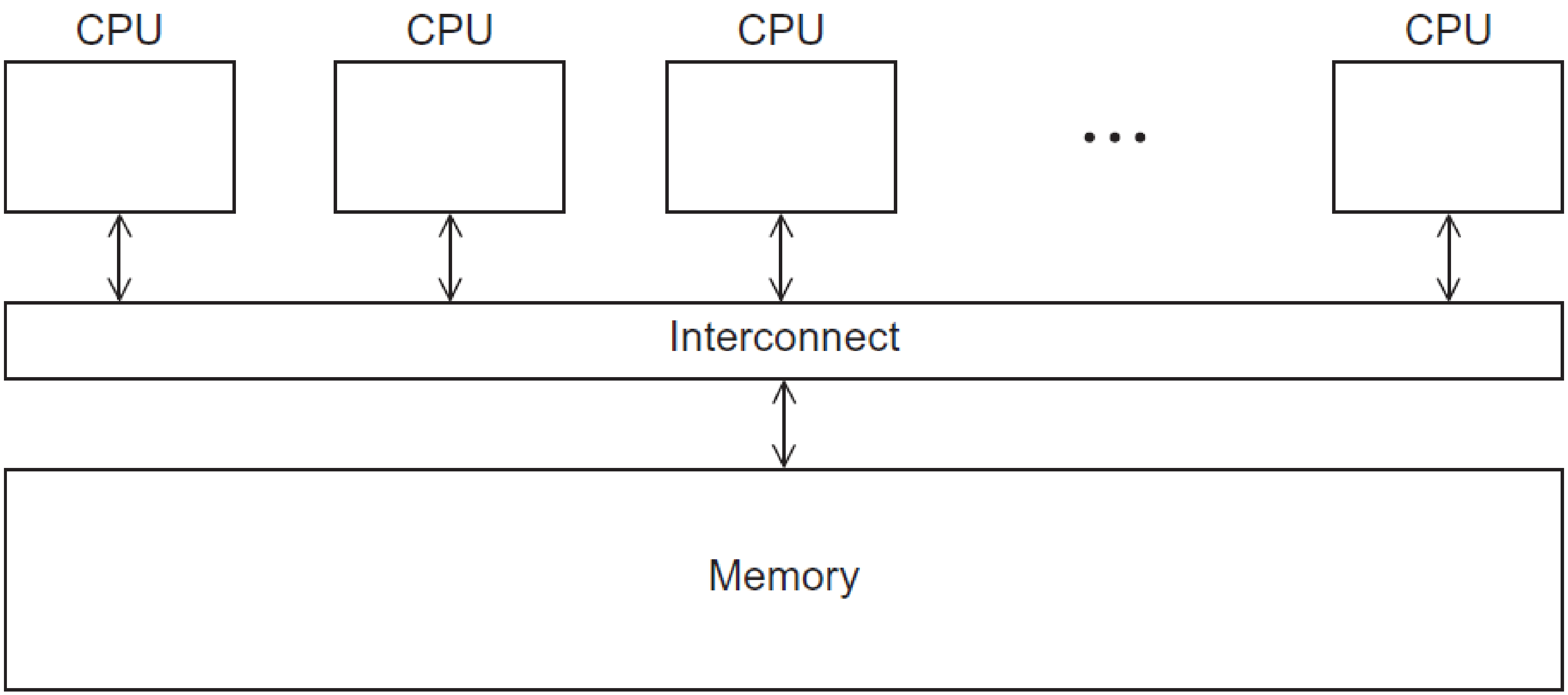
These insights are coming from HPC folks though!

Source: www.sdsc.edu/~allans/cs260/lectures/OpenMP.ppt

OpenMP

- An API for shared-memory parallel programming.
- Designed for systems in which each thread 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 → shared memory architecture

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
```

```
./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 Implementation in GCC

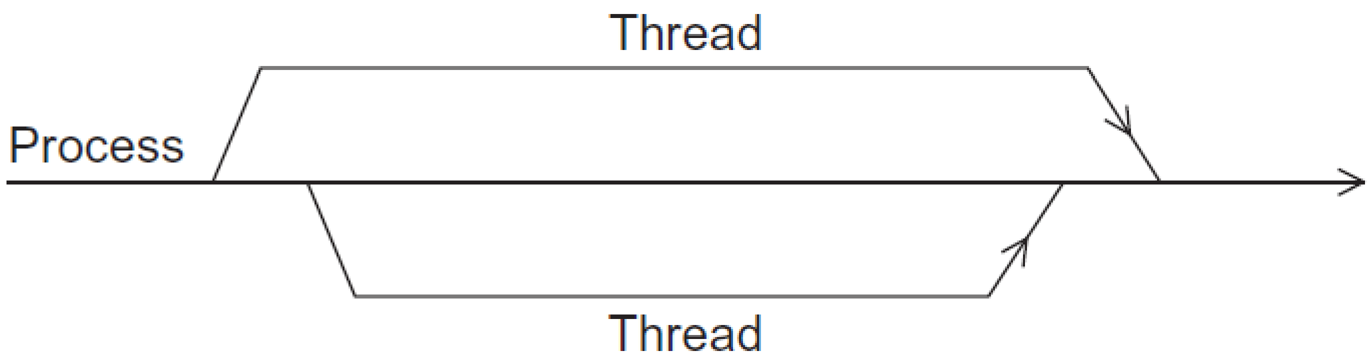
GCC version	OpenMP version
4.2	2.5
4.4	3.0
4.7	3.1
4.9	4.0
6	4.5
9	5.0 (basic)
11	5.0 (more features)
12	5.0 extended + some 5.1

OpenMp pragmas

- # pragma omp parallel

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

A process forking and joining two threads



clause

pragma omp parallel **num_threads (thread_count)**

- A clause is a 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.

Of note...

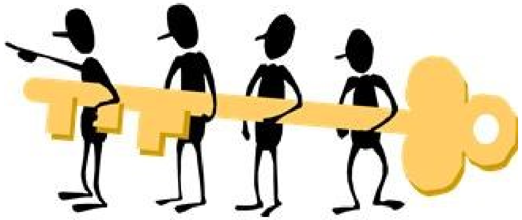
- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn't guarantee that this will actually start thread_count threads.

However

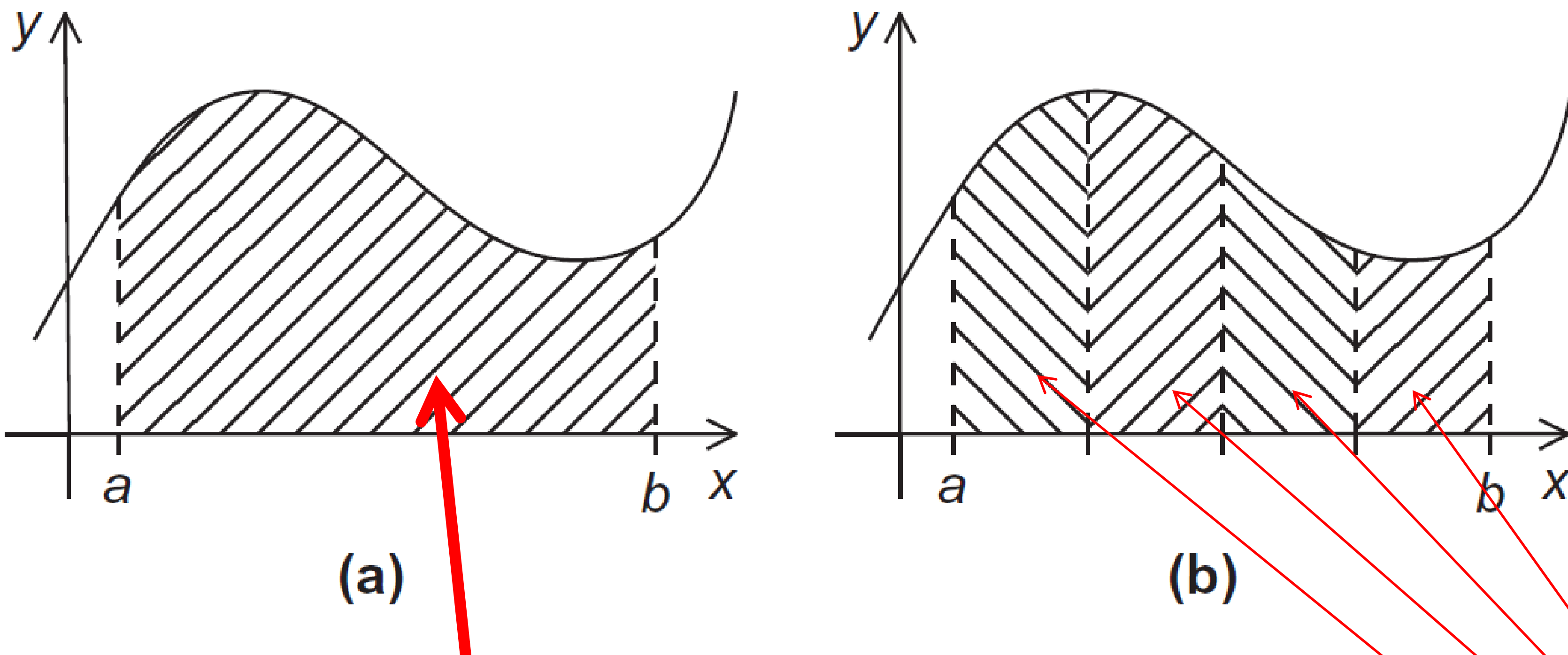
- Unless we're trying to start a lot of threads, we will almost always get the desired number of threads.

Some terminology

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



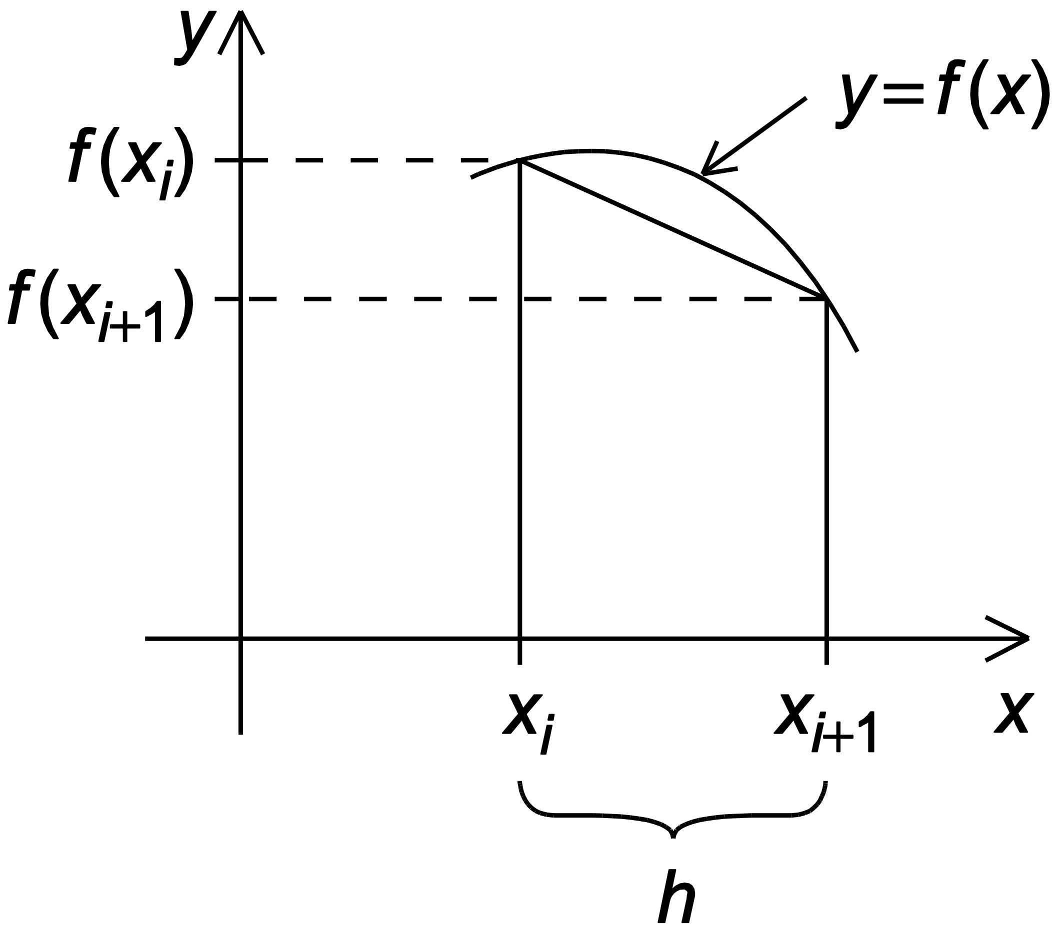
Example: The Trapezoidal Rule



To find this area

We approximate it with trapezoids.

One trapezoid



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

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

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.

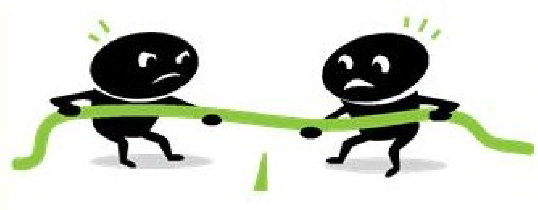
- So we aggregated tasks by assigning a contiguous block of trapezoids to each thread.

- Each thread must add its local sum to the global sum. That is, each thread must execute: 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:

global_result += my_result ;

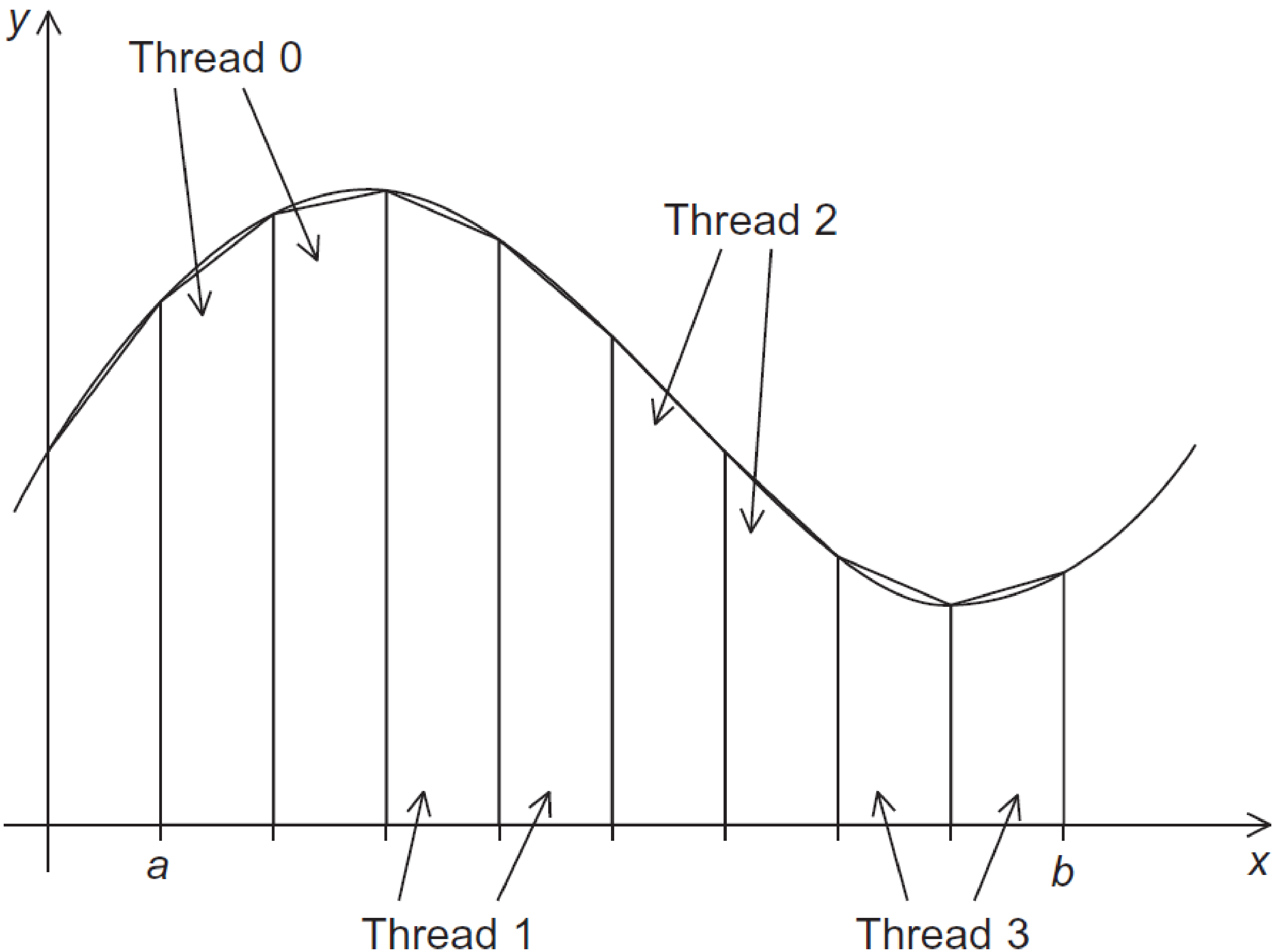


Mutual exclusion

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

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

Assignment of trapezoids to threads



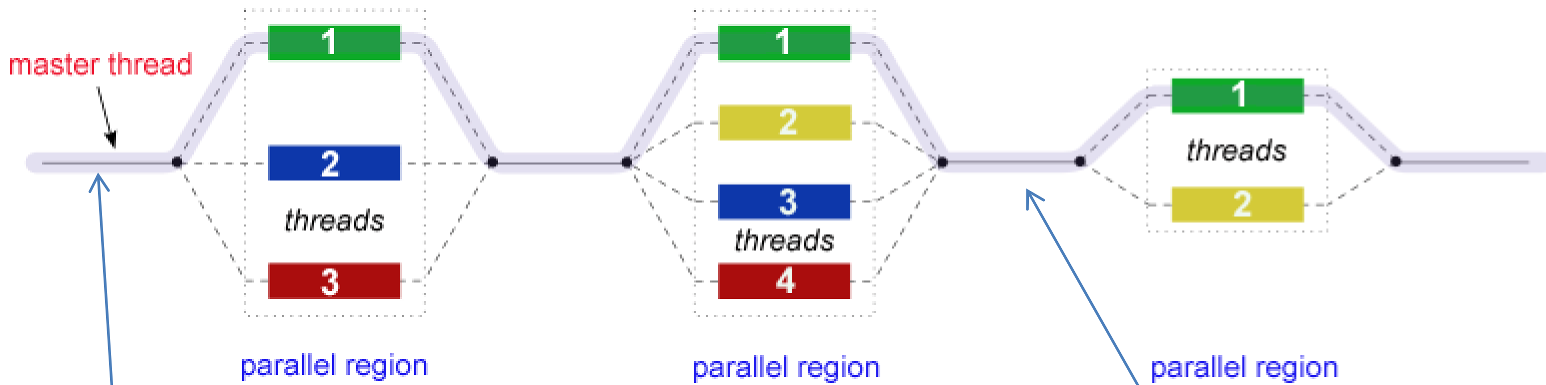
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

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

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* 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 */
```

OpenMP Parallel Programming

1. Start with a *parallelizable* algorithm
   - loop-level parallelism is necessary
2. Implement serially
3. Test and Debug
4. Annotate the code with parallelization (and synchronization) directives
5. Hope for linear speedup
6. Test and Debug

OpenMP uses the fork-join model of parallel execution.



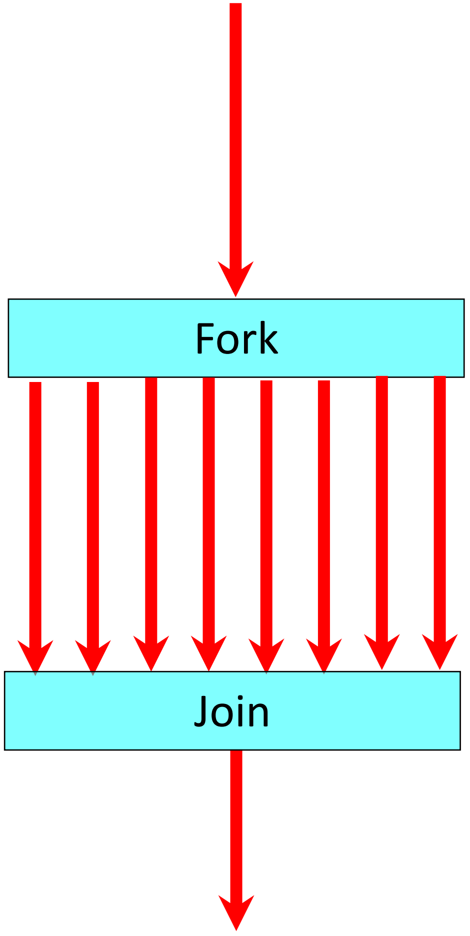
All OpenMP programs begin with a single thread: **master thread** (ID = 0)

FORK: the master thread then creates a team of parallel ***threads***.

JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate

Programming Model - Threading

```
int main() {
  // serial region
  printf("Hello…");
  // parallel region
  #pragma omp parallel
  {
    printf("World");
  }
  // serial again
  printf("!");
}
```



We didn't use omp_set_num_threads(), what will be the output?

What we learned so far

- #include <omp.h>
- gcc -fopenmp …
- omp_set_num_threads(x);
- omp_get_thread_num();
- omp_get_num_threads();
- #pragma omp parallel [num_threads(x)]
- #pragma omp critical

The concept of 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.

```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();
```

shared

```c
   return 0;
}  /* main */
```

private

```c
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 */
```

```
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 */
```

Do you remember the trapezoidal?

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);
```

How about this:

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

and we use 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.

It is now slower than a version with single thread!

How can we fix this?

We can avoid this problem by:

1. declaring a private variable inside the parallel block
2. 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;
   }
```

Can we do better?

Reduction operators

- 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>)
```

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

Be careful of:

- subtraction
- floating points

And the code becomes:

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

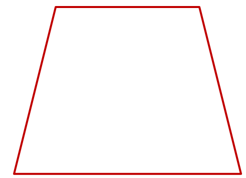
How Does OpenMP Do it?

- The reduction variable is shared
- OpenMP create a local variable for each thread
- When the parallel block ends, the values in the private variables are combined into the shared variable.

#pragma omp parallel for

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

```
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;
```

In a loop that is parallelized with *parallel for* the default scope of a loop variable is private

Legal forms for parallelizable *for statements*

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

Number of iterations MUST be known prior to the loop execution.

OpnMP won't parallelize while loops or do-while loops.

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

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

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

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.

Question

Do we have to worry about the following:

```
#pragma omp parallel for num_threads(2)
  for( i =0 ; i < n; i++) {
      x[i] = a + i*h;
      y[i] = exp(x[i]);
  }
```

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

```
        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;
```

Is this a good solution?

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);
        }
```

How about this one?

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);
      }
```

ensures factor has private scope.

The default clause

```
default(none)
```

- Lets the programmer specify the scope of each variable in a block.
- With this clause the compiler requires 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-carried dependency in inner loop

Loop-carried dependency in outer loop

What can we do?

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]) 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

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

Serial Odd-Even Transposition Sort

No dependence in inner loops

```
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]);
```

Outer-loop carried dependence

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;
            }
         }
   }
```

What if a thread proceeds from phase p to phase p+1 before other threads?

Performance issue:
For each outer iteration, OpenMP may fork-join threads.
Repeated overhead per iteration.
Can we do better?

Second OpenMP Odd-Even Sort

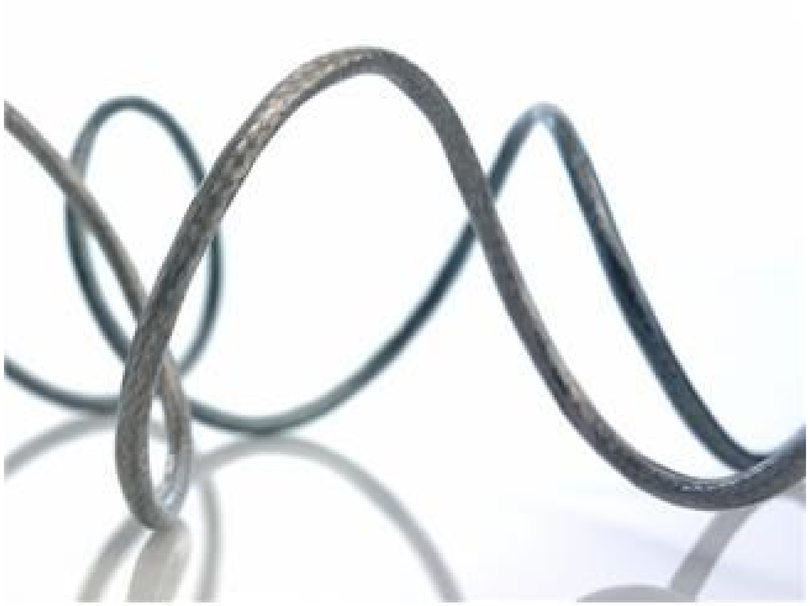
```
#   pragma omp parallel num_threads(thread_count) \
       default(none) shared(a, n) private(i, tmp, phase)
    for (phase = 0; phase < n; phase++) {
       if (phase % 2 == 0)
#         pragma omp for
          for (i = 1; i < n; i += 2) {
             if (a[i-1] > a[i]) {
                tmp =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

for directive does not fork any threads. But uses whatever threads that have been forked before in the enclosing parallel block.

(Times are in seconds.)

Array of 20,000 elements

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

Take a look at this:

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

- Usually, the default for many OpenMP implementations is to parallelize the above iterations as block of consecutive n/thread_count iterations to each thread.
- What if f(i) has latency that increases with i? What is the best schedule then?

Example of function *f*.

```
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 */
```

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

Wouldn't this be better? (why?)

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.

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

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

Example: 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

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 default chunksize of 1 is used.

The Guided Schedule Type

- Each thread also executes a chunk (total iterations/ num threads), 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.

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

Approximation: #remaining iterations / number of threads

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

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.
- Example:

  export OMP_SCHEDULE ="static,1"

Keep in mind:

- There is an overhead in using the schedule directive
- The overhead is higher in dynamic than static schedules
- The overhead of guided is the greatest of all three.
- So: if we get satisfactory performance without schedule then don't use schedule.

Rules of thumb

- If each iteration requires roughly the same amount of computation → default is best
- If the cost of each iteration increases/decreases linearly as the loop executes → static with small chunksize
- If the cost cannot be determined → you need to try several schedules: schedule(runtime) and try different options with OMP_SCHEDULE

Question

Can we parallelize the following loop? If yes, do it. If not, why not?

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

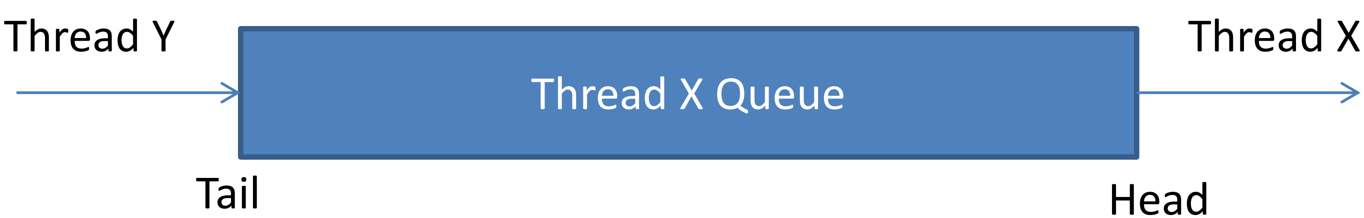
PRODUCERS AND CONSUMERS

Queues

- A natural data structure to use in many multithreaded applications.
- The two main operations: enqueue and dequeue
- 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.

Example of Usage: 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.



Example of Usage: Message-Passing

Each thread executes the following:

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

while (!Done())
    Try_receive();
```

Send_msg()

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

Try_receive()

```
    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);
```

When queue size is 1, dequeue affects the tail pointer.

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.
- Each thread allocates its queue in the array.

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.

```
# pragma omp barrier
```

Managing Mutual Exclusion

- critical directive
- atomic directive
- locks

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.

The Atomic Directive

- Higher performance than critical
- 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;
```

Must not reference X

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

Locks

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

Locks: main actions

```
/* 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;
```

Locks: main actions

```
void omp_init_lock(omp_lock_t *      lock_p);

void omp_set_lock(omp_lock_t *       lock_p);

void omp_unset_lock(omp_lock_t *     lock_p);

void omp_destroy_lock(omp_lock_t *   lock_p);
```

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

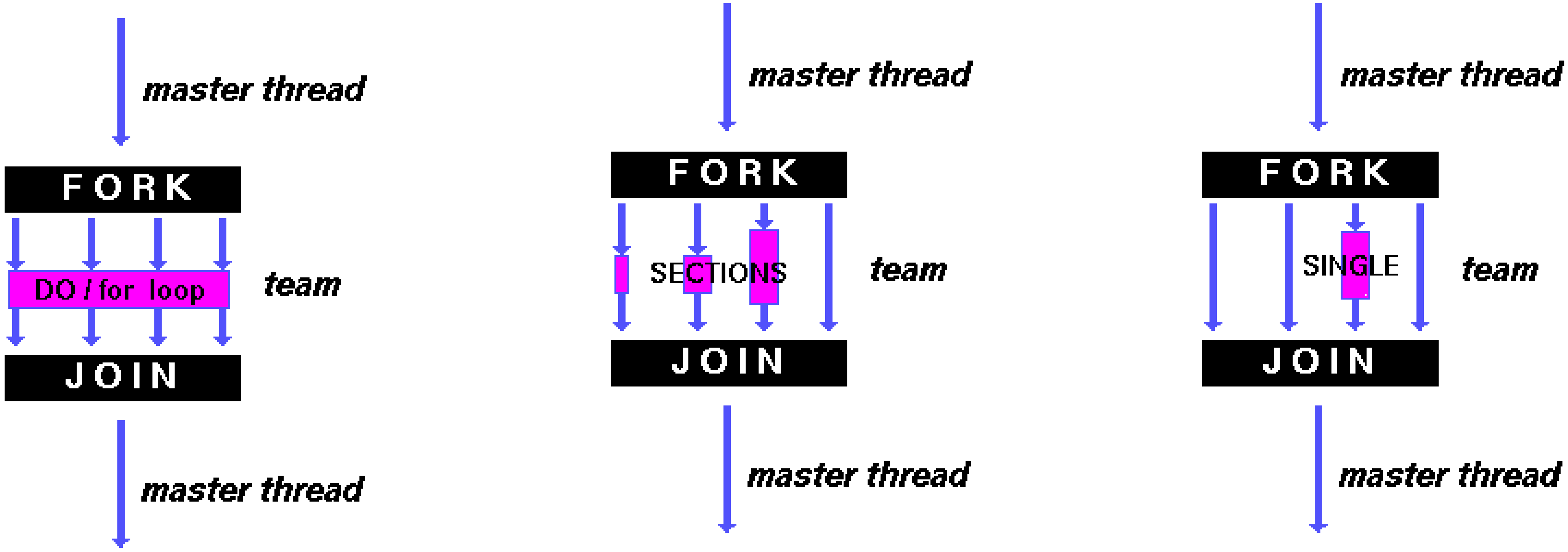
```
#  pragma omp critical
   /* q_p = msg_queues[my_rank] */
   Dequeue(q_p, &src, &mesg);
```

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

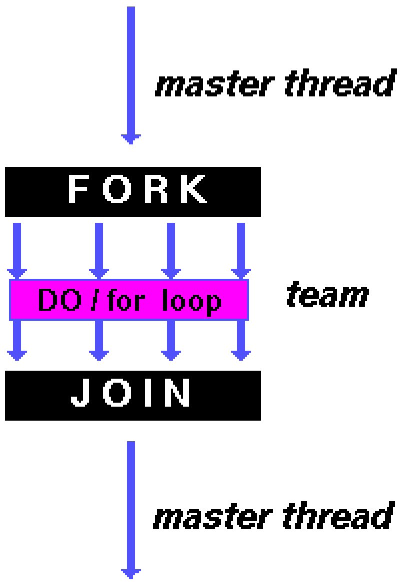
Some Caveats

1. You shouldn't mix the different types of mutual exclusion for a single critical section.
   - i.e. do not mix atomic and critical for the same variable update
2. There is no guarantee of fairness in mutual exclusion constructs.
   - A thread can be blocked forever!
3. It can be dangerous to "nest" mutual exclusion constructs.

Dividing Work Among Threads

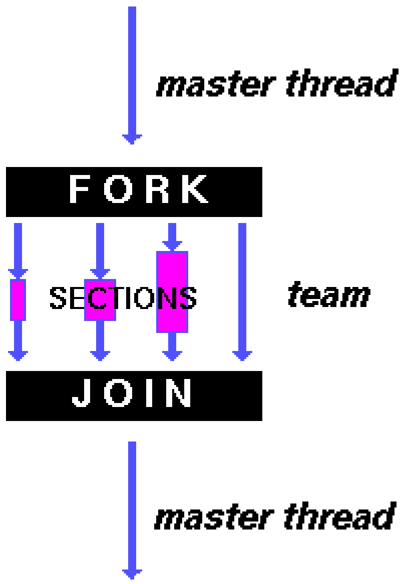


Dividing Work Among Threads



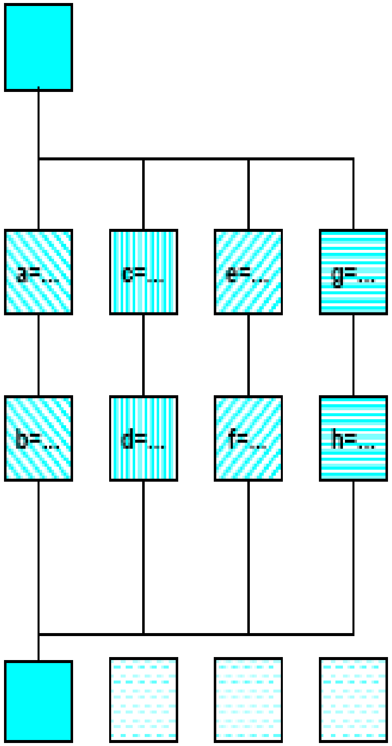

```
#pragma omp parallel for
for_loop
```

Dividing Work Among Threads



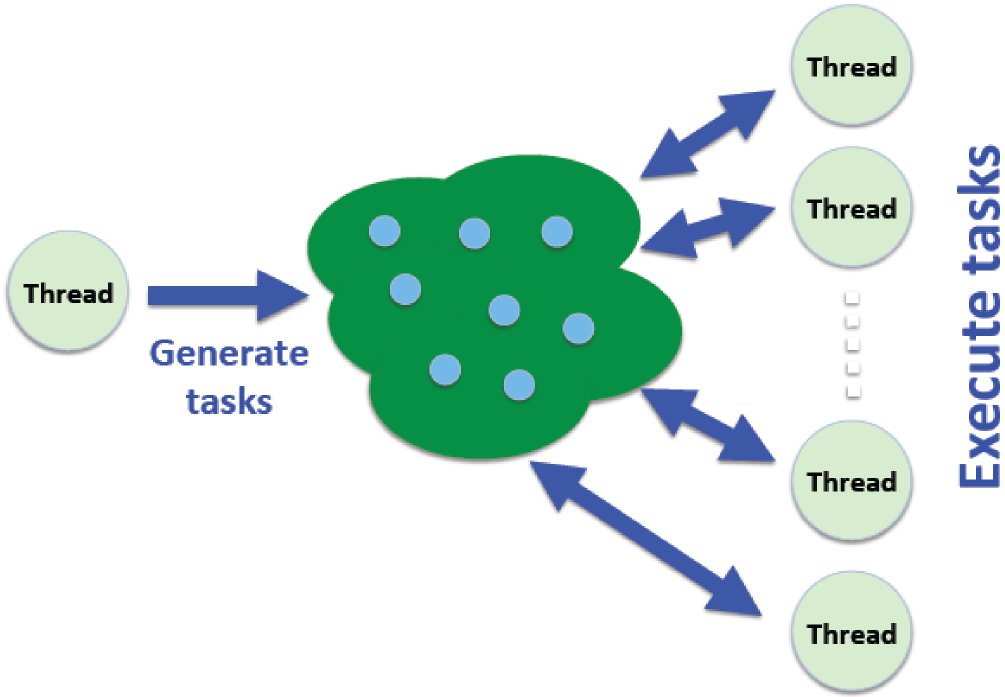

```
#pragma omp parallel
#pragma omp sections
{
   #pragma omp section
         structured_block

   #pragma omp section
         structured_block
}
```

```
#pragma omp parallel
{
      #pragma omp sections
      {      { a=...;
               b=...; }
            #pragma omp section
               { c=...;
               d=...; }
            #pragma omp section
               { e=...;
               f=...; }
            #pragma omp section
               { g=...;
               h=...; }
      } /*omp end sections*/
} /*omp end parallel*/
```



Tasks

- Feature added to version 3.0 of OpenMP
- A task is: an independent unit of work
- A thread is assigned to perform a task.



Source:
Ruud van der Pass
SC'13

Tasks Example

```
#pragma omp parallel {

    #pragma omp single {
            node *p = head_of_list;
            while (p) {
                #pragma omp task private(p)
                    process(p);
              p = p->next;
             } // end while
       } //end pragma single
}// end pragma parallel
```

Implicit barrier
At that point the threads start executing the tasks.

Two Task Synchronization Constructs That We Will Use

#pragma omp barrier

#pragma omp taskwait

- explicitly waits on the completion of child tasks

Example:
Write a program that prints either "A Race Car" or "A Car Race"

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

int main(int argc, char *argv[]) {

          printf("A ");
          printf("race ");
          printf("car ");

   printf("\n");
   return(0);
}
```

```
$ cc -fast hello.c
$ ./a.out
A race car
$
```

What will this program print ?

Source:
Ruud van der Pass
SC'13

Example:
Write a program that prints either "A Race Car" or "A Car Race"

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

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("A ");
          printf("race ");
          printf("car ");

   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Source:
Ruud van der Pass
SC'13

Example:
Write a program that prints either "A Race Car" or "A Car Race"

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

int main(int argc

   #pragma omp parallel
   {
     #pragma omp single
      {
            printf("A ");
            printf("race ");
            printf("car ");
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Source:
Ruud van der Pass
SC'13

Example:
Write a program that prints either "A Race Car" or "A Car Race"

```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
      {
          printf("A ");
          #pragma omp task
           {printf("race ");}
          #pragma omp task
           {printf("car ");}
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Source:
Ruud van der Pass
SC'13

Example:
Write a program that prints either "A Race Car" or "A Car Race"

```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
      {
          printf("A ");
          #pragma omp task
           {printf("race ");}
          #pragma omp task
           {printf("car ");}
         printf("is fun to watch ");
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

```
A is fun to watch race car
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch car race
```

What will this program print using 2 threads ?

Source:
Ruud van der Pass
SC'13

Example:
Write a program that prints either "A Race Car" or "A Car Race"

```
int main(int argc, char

  #pragma omp parallel
  {
    #pragma omp single
     {
        printf("A ");
        #pragma omp task
          {printf("car ");}
        #pragma omp task
          {printf("race ");}
        #pragma omp taskwait
        printf("is fun to watch ");
     }
   } // End of parallel region

   printf("\n");return(0);
}
```

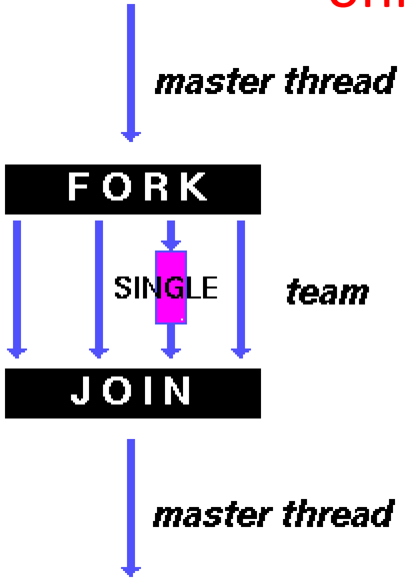
What will this program print using 2 threads ?

```
A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
```

Source:
Ruud van der Pass
SC'13

Dividing Work Among Threads

Specifies that the enclosed code is to be executed by only one thread in the team.




```
#pragma omp single [clause ...]

   structured_block
```

Performance Issues

Performance

- Easy to write OpenMP but hard to write an efficient program!
- 5 main causes of poor performance:
  - Sequential code
  - Communication
  - Load imbalance
  - Synchronisation
  - Compiler (non-)optimisation.

Sequential code

- Amdahl's law: Limits performance.
- Need to find ways for parallelising it!
- In OpenMP, all code outside of parallel regions and inside MASTER, SINGLE and CRITICAL directives is sequential.
  - This code should be as small as possible.

Communication

- On Shared memory machines, communication = increased memory access costs.
  - It takes longer to access data in main memory or another processor's cache than it does from local cache.
- Memory accesses are expensive!

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

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];
   }
```

Worse performance relative to 8000x8000 is mainly due to cache performance

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

Even though the number of operations is the same!

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];
    }
```

Far more write-misses than the other two.

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

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];
    }
```

Far more read-misses than the other two.

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

Data affinity

- Data is cached on the cores which access it.
  - Must reuse cached data as much as possible.

- Write code with good *data affinity*:
  - Ensure the same thread accesses the same subset of program data as much as possible.

- Try to make these subsets large, contiguous chunks of data.
  - Will avoid false sharing and other problems.

- The manner in which the memory is accessed by individual threads has a major influence on performance
  - If each thread accesses a distinct portion of data consistently through the program, the threads will probably make excellent use of memory.
  - This improvement includes good use of thread-local cache.

Load imbalance

- Load imbalance can arise from both communication and computation.

- Worth experimenting with different scheduling options
  - **runtime** clause is handy here

- If none are appropriate, may be best to do your own scheduling!

Synchronisation

- Barriers can be very expensive
- Avoid barriers via:
  - *Careful* use of the NOWAIT clause. A recommended strategy is:
    - Parallelise at the outermost level possible.
    - May require re-ordering of loops /indices.
  - Choice of CRITICAL / ATOMIC / lock routines may impact performance.

Compiler (non-)optimisation

- Sometimes the addition of parallel directives can inhibit the compiler from performing sequential optimisations.
- Symptoms:
  - 1-thread parallel code has longer execution and higher instruction count than sequential code.
- Can sometimes be cured by making shared data private, or local to a routine.

Performance Tuning

- My code is giving me poor speedup. I don't know why. What do I do now?
- A:
  - Say "this machine/language is a heap of junk"
  - Give up and go back to your laptop
- B:
  - Try to classify and localise the sources of overhead.
    - What type of problem is it and where in the code does it occur
  - Fix problems that are responsible for large overheads first.
  - Iterate

Performance Tuning: Timing the OpenMP Performance

- A standard practice is to use a standard operating system command.
- For example

  $time   ./a.out

  – The "real", "user", and "system" times are then printed after the program has finished execution.

  – For example

```
$ time   ./prog
real     5.4   Elapsed time
user     3.2
sys      1.0
```

$$\left.\begin{array}{ll}\text{user} & 3.2\\ \text{sys} & 1.0\end{array}\right\}\text{CPU time}$$

  – These three numbers can be used to get initial information about the performance.

Performance Tuning: Timing the OpenMP Performance

- A common cause for the difference between the wall-clock time of 5.4 seconds and the CPU time is a processor sharing too high a load on the system.
- If sufficient processors are available (i.e., not being used by other users), your elapsed time should be less than the CPU time.
- The omp_get_wtime() function provided by OpenMP is useful for measuring the elapsed time of blocks of source code.
  - Elapsed wall clock time in seconds (returns double)
  - Time is measured from some "time in the past".

```
t_start = omp_get_wtime();
#pragma omp parallel
{
  .....
}
t_taken = omp_get_wtime() - t_start;
```

Performance Tuning: Avoid Parallel Regions in Inner Loop

- Another common technique to improve the performance is to move parallel regions out of the innermost loops.
- **Why?**
  - Otherwise, we repeatedly incur the overheads of the parallel construct (i.e. creating threads).
  - By moving the parallel construct outside of the loop nest, the parallel construct overheads are minimized.

Performance Tuning: Overlapping Computation and I/O

- This helps avoid having all but one processors wait while the I/O is handled.
- A general rule for MIMD parallelism in general is to overlap computation and communications so that the total time taken is less that the sum of the times to do each of these.
- However, this general guideline might not always be possible.

Exercises

Problem 1

There is a problem with the following code. Please explain the problem in 1-2 lines. Can we fix this problem? If yes, show how (either by writing few lines of code or by explaining what you will do.). If no, explain why not.

```
#pragma omp parallel for
for (int i = 1; i < N; i++)
{
 A[i] = B[i] – A[i – 1]; //A and B are arrays of int
}
```

Problem 1

```
#pragma omp parallel for
for (int i = 1; i < N; i++)
{
 A[i] = B[i] - A[i - 1]; //A and B are arrays of int
}
```

The code as above has loop carried dependency.

We can fix this problem

A[1] = B[1] - A[0]
A[2] = B[2] - A[1] = B[2] – B[1] + A[0]
A[3] = B[3] - A[2] = B[3] -B[2] + B[1] - A[0]
So the loop body will be:

```
factor = 1  //before the outer for-loop but after the program
{  int sum = 0;
   for(j = i; j >0; j--)
      { sum += factor*B[j]; factor = -factor; }
  A[i] = (i%2 == 0? sum - A[0]: sum + A[0]);
}
```

Problem 2

Suppose we have the following code snippet. Assume all variables and arrays have been declared and initialized earlier.

```
1.      #pragma omp parallel
2.      {
3.
4.              for (i = 1 ; i < M; i += 2 )
5.              {
6.                  D[i] = x * A[i] + x * B[i];
7.               }
8.
9.              #pragma omp for
10.             for (i = 0; i < N; i++)
11.            {
12.                C[i] = k * D[i];
13.             }
14.     } // end omp parallel
```

Assume M = N. What do we have to write in line 3 above to get as much performance as possible from the above code? You must not have race condition and at the same time get the best performance.

Problem 2

Suppose we have the following code snippet. Assume all variables and arrays have been declared and initialized earlier.

```
#pragma omp parallel
{
    for (i = 1 ; i < M; i += 2 )
    {
        D[i] = x * A[i] + x * B[i];
    }
    #pragma omp for
    for (i = 0; i < N; i++)
    {
        C[i] = k * D[i];
    }
} // end omp parallel
```

Repeat problem a, but assume M = 2N.

Problem 2

Suppose we have the following code snippet. Assume all variables and arrays have been declared and initialized earlier.

```
1.      #pragma omp parallel
2.      {
3.
4.              for (i = 1 ; i < M; i += 2 )
5.              {
6.                  D[i] = x * A[i] + x * B[i];
7.              }
8.
9.              #pragma omp for
10.             for (i = 0; i < N; i++)
11.             {
12.                 C[i] = k * D[i];
13.             }
14.     } // end omp parallel
```

If M = 16, N = 8, and the multicore on which we run the code has 16 cores.
How many threads do you think OpenMP runtime will create when executing line 1?
Assume no other processes/threads are using the multicore except the above code.

Problem 3

Suppose we have the following code snippet is running on a four-core processor:

```
1.   #pragma omp parallel for num_threads(8)
2.   for(int i = 0; i <N; i++){
3.    for(int j = i; j < N; j++){
4.          array[i*N + j] = (j-i)!;
5.    }
6.   }
```

How many iterations will each thread execute from the outer loop (line 2)? Justify

Problem 3

Suppose we have the following code snippet is running on a four-core processor:

```
1.    #pragma omp parallel for num_threads(8)
2.    for(int i = 0; i <N; i++){
3.     for(int j = i; j < N; j++){
4.            array[i*N + j] = (j-i)!;
5.     }
6.    }
```

Will each thread execute the same number of iterations of the inner loop (line 3)? Explain.

Problem 3

Suppose we have the following code snippet is running on a four-core processor:

```
1.    #pragma omp parallel for num_threads(8)
2.    for(int i = 0; i <N; i++){
3.     for(int j = i; j < N; j++){
4.            array[i*N + j] = (j-i)!;
5.     }
6.    }
```

Is there a possibility that (line 4) is a critical section? Justify.

Problem 4

Suppose we have the following code snippet is running on an eight core processor:

```
#pragma omp parallel for
for(int i = 0; i <N; i++){
    for(int j = i; j < N; j++){
        array[i*N + j] = sin(i) + cos(j);
    }
}
```

Given the code above, which **schedule** will be better in terms of performance: static? Or Dynamic? And Why?

Problem 4

Suppose we have the following code snippet is running on an eight core processor:

```
1.   #pragma omp parallel for
2.   for(int i = 0; i <N; i++){
3.       for(int j = i; j < N; j++){
4.           array[i*N + j] = sin(i) + cos(j);
5.       }
6.   }
```

If we substitute line 1 with #pragma omp for will we get same? Better? or worse performance than the original code? Justify your answer.

Problem 4

Suppose we have the following code snippet is running on an eight core processor:

```
1.   #pragma omp parallel for
2.   for(int i = 0; i <N; i++){
3.        for(int j = i; j < N; j++){
4.            array[i*N + j] = sin(i) + cos(j);
5.        }
6.   }
```

If we move the pragma statement from the outer loop, line 1, to the inner loop, before line 3, will that give better performance? Justify your answer.

Problem 4

Suppose we have the following code snippet is running on an eight core processor:

```
1.  #pragma omp parallel for
2.  for(int i = 0; i <N; i++){
3.          for(int j = i; j < N; j++){
4.                  array[i*N + j] = sin(i) + cos(j);
5.          }
6.  }
```

Assume *array[]* is an array of int, cache block is 64 bytes, and N is 4096.
For each statement below, to be used in line 1 above, indicate whether there may be false sharing and why. Assume we have one thread per outer loop iteration.

- #pragma omp parallel for schedule(static, 1)
- #pragma omp parallel for schedule(static, 8)
- #pragma omp parallel for schedule (static, 16)
- #pragma omp parallel for schedule (dynamic)

Conclusions

- OpenMP is a standard for programming shared-memory systems.
- The main concept to parallelize a program with OpenMP is how to have independent for-loops.
- OpenMP (4.5 and up) started supporting heterogeneous computing (i.e. offloading tasks to GPUs, FPGAs, …).

http://www.openmp.org/