OpenMP

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

www.openmp.org

OpenMP

- An API for shared-memory parallel programming.
- MP = multiprocessing (shared mem)
- 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.
- Use mainlly for multi core programming

OpenMP

- Pthreads the programmer explicitly specify the behavior of each thread
  - Library that is supported by any C compiler
  - Low level
- OpenMP allows the programmer to simply state that a block of code should be executed in parallel, which thread to execute the task is left for the compiler and run time system
  - OpenMP requires compiler support
  - High level

Pragmas

- #pragmas are part of standrd C
- Special preprocessor instructions. Have to be one line length
- 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
```

compiling

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

- Set of pragmas
- Each pragmas has “attributes” called clause

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.

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 )
```

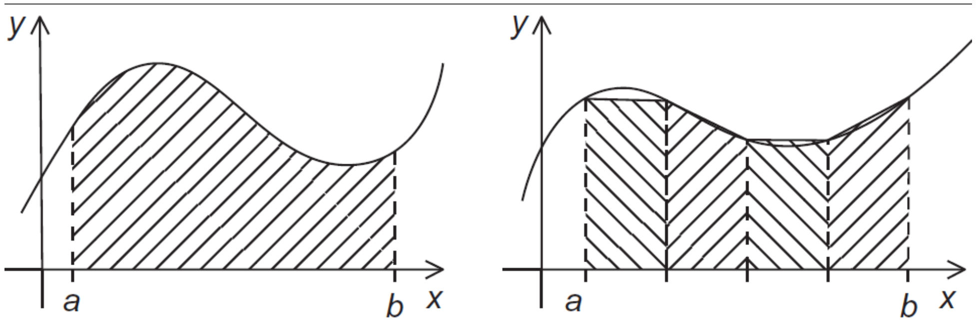
Some terminology

- 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

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



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 (and a single thread to each core).

Mutual exclusion

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

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <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 */
```

Reduction clause

- A reduction 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.

We need this more complex function call on the trapezoid function 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;
   }
```

A reduction clause can be added to a parallel directive.

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

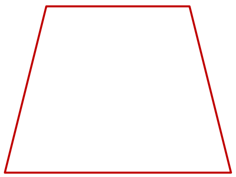
+, *, -, &, |, ^, &&, ||

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

The pragma : Parallel for

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

```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;
```

The pragma : Parallel for

- Does only work with for loops (no while, do-while)
- Only for loops which number of iterations:
  - Defined in the for statement
  - Know prior to execution of loop
- No break or return inside of the loop. The compiler will complain
- recursive calls with data dependencies will give inconsistent results without compiler warning

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, can't be correctly parallelized by OpenMP.

Data dependencies

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

There is data dependence on x[i] BUT there is no problem since same thread will calculate x[i] and use it

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

```
        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.

By default any variable declared before the loop (exception of the loop variable) is shared among threads

The default clause

```
default(none)
```

- Forces the programmer to specify the scope of each variable in the block 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);
     }
```

Bubble Sort

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

Serial Odd-Even Transposition Sort

```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n; i += 2)
         if (a[i−1] > a[i]) 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

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;
            }
         }
   }
```

We need to be sure that threads finish phase p before starting p+1, which is provided by the parallel for directive: a barrier is inplicit at the en of the for loop

Overhead of creating and joining the 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;
            }
         }
   }
```

pragma omp for. Parallelize the for loop with 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

The pragma : Parallel task

- The OpenMP **task** pragma can be used to explicitly define a task.
- The **task** pragma can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms for which other OpenMP workshare constructs are inadequate. The **task** directive only takes effect if you specify the **-qsmp** compiler option.

```
int fib(int n) {
        int i, j;
        if (n<2)
                return n;
        else {
                #pragma omp task shared(i)
                i=fib(n-1);
                #pragma omp task shared(j)
                j=fib(n-2);
                #pragma omp taskwait
                return i+j;
        }
}
```

The pragma : Parallel section

- Different threads carry different kind of work. Allowing the definition of different regions each of which will be executed by one of the threads
- Each thread executes one code block at a time , each code block executed only once
- Suppose you have blocks of code that can be executed in parallel (i.e. No dependencies). Usually for function calls and subroutines

The pragma : Parallel section

```
#pragma omp parallel
{
        #pragma omp sections
        {
                #pragma omp section
                (void) functionA();
                #pragma omp section
                (void) functionB();
        }
}
void functionA(){
  printf("In funcA:this section is executed by thread %d\n",
omp_get_thread_num());
}
void functionB(){
        printf("In funcB:this section is executed by thread %d\n",
omp_get_thread_num());
}
```

Limits the parallelization to two threads
If only one thread is available then the two functions will executed sequentially
No assumptions about specific order
May lead to Load balancing problem

Task vs section

- The difference between tasks and sections is in the time frame that their code would execute.
- Sections are enclosed within the sections construct and (unless the nowait clause was specified) threads would not leave it until all sections have been executed:

```
[ sections ]
Thread 0: -------< section 1 >---->*------
Thread 1: -------< section 2      >*------
Thread 2: ------------------------>*------
... *
Thread N-1: ---------------------->*------
```

- Here N threads encounter a sections construct with two sections, the second taking more time than the first. The first two threads execute one section each. The other N-2 threads simply wait at the implicit barrier at the end of the sections construct (show here as *).

Task vs section

- Tasks are queued and executed whenever possible. The run-time is also allowed to move task between threads, even in the mid of their lifetime.
- That means that one task might start executing in one thread, then at some scheduling point it might be migrated by the runtime to another thread.
- Still tasks and sections are in many ways similar.

The pragma : Parallel single

- Block executed by one thread only

```
#pragma omp parallel shared(a,b) private(i){
  #pragma omp single
  {
    a=10;
  } //only one thread init shared variable a.
  #pragma omp for
  for(i=0;i<n:i++){
    b[i]=a;
  }
}
```

SCHEDULING LOOPS CLAUSE

We want to parallelize this loop.

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

If f(i) execution time depends on size of i then if we assign work to threads linearly some we will have an unbalanced thread work

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 */
```

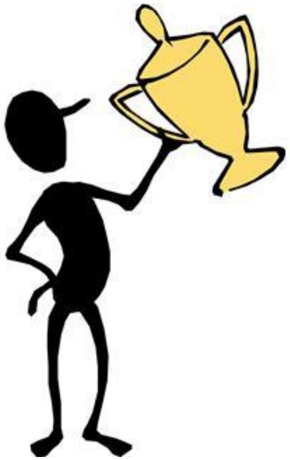
Our definition of function *f*.

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



The Schedule Clause

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

```
      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. Group of iterations executed consecutively loop. If omited is total_iterations/thread_count

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)
```

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

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 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.

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.

OpenMP Synchronization Pragmas

#pragma omp barrier
when a thread encounters this barrier, it blocks until all other threads reach this barrier also

#pragma omp atomic
protect critical sections that contains only one line of C statement (++x, x++, ..)

#pragma omp critical(name)
two critical blocks with different names can be executed together

Locks

void omp_init_lock(omp_lock_t* lock_p) //init lock

void omp_set_lock(omp_lock_t* lock_p) //sets the lock if succed then the thread proceed else it will block

void omp_unset_lock(omp_lock_t* lock_p) //unset the lock

void omp_destroy_lock(omp_lock_t* lock_p)

Synchronization

prevents multiple threads from accessing the critical sections code at same time

```
int count;
void Trick(){
        #pragma omp critical
                count++;
}
```

The Atomic Directive (1)

```
# pragma omp atomic
```

- Unlike the critical directive, it can only protect critical sections that consist of a single C assignment statement.
- Further, the statement must have one of the following forms:

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

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

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

Locks

```
#pragma omp critical
Enqueue(q_p, my_rank, msg);

Replaced by:
Omp_set_lock(&qp->lock);
Enqueue(q_p, my_rank, msg);
Omp_unset_lock(&qp->lock);
```

Which method is faster

Potentially atomic directive works faster

But when using several atomics directives than is better to use critical(names) directives to avoid one atomic stopping all of them

In general atomic and critical directives perform equally fast

Locks should be used when mutual exclusion is needed for a data structures rather than a block of code

Example

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

Matrix-vector multiplication

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

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

```
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)

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

OpenMP pragma directives	Description
#pragma omp atomic	Identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.
#pragma omp parallel	Defines a parallel region to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive.
#pragma omp for	Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel.
#pragma omp parallel for	Shortcut combination of **omp parallel** and **omp for** pragma directives, used to define a parallel region containing a single **for** directive.
#pragma omp ordered	Work-sharing construct identifying a structured block of code that must be executed in sequential order.
#pragma omp section, #pragma omp sections	Work-sharing construct identifying a non-iterative section of code containing one or more subsections of code that should be run in parallel.
#pragma omp parallel sections	Shortcut combination of **omp parallel** and **omp sections** pragma directives, used to define a parallel region containing a single **sections** directive.
#pragma omp single	Work-sharing construct identifying a section of code that must be run by a single available thread.
#pragma omp master	Synchronization construct identifying a section of code that must be run only by the master thread.
#pragma omp critical	Synchronization construct identifying a statement block that must be executed by a single thread at a time.
#pragma omp barrier	Synchronizes all the threads in a parallel region.
#pragma omp flush	Synchronization construct identifying a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.
#pragma omp threadprivate	Defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.