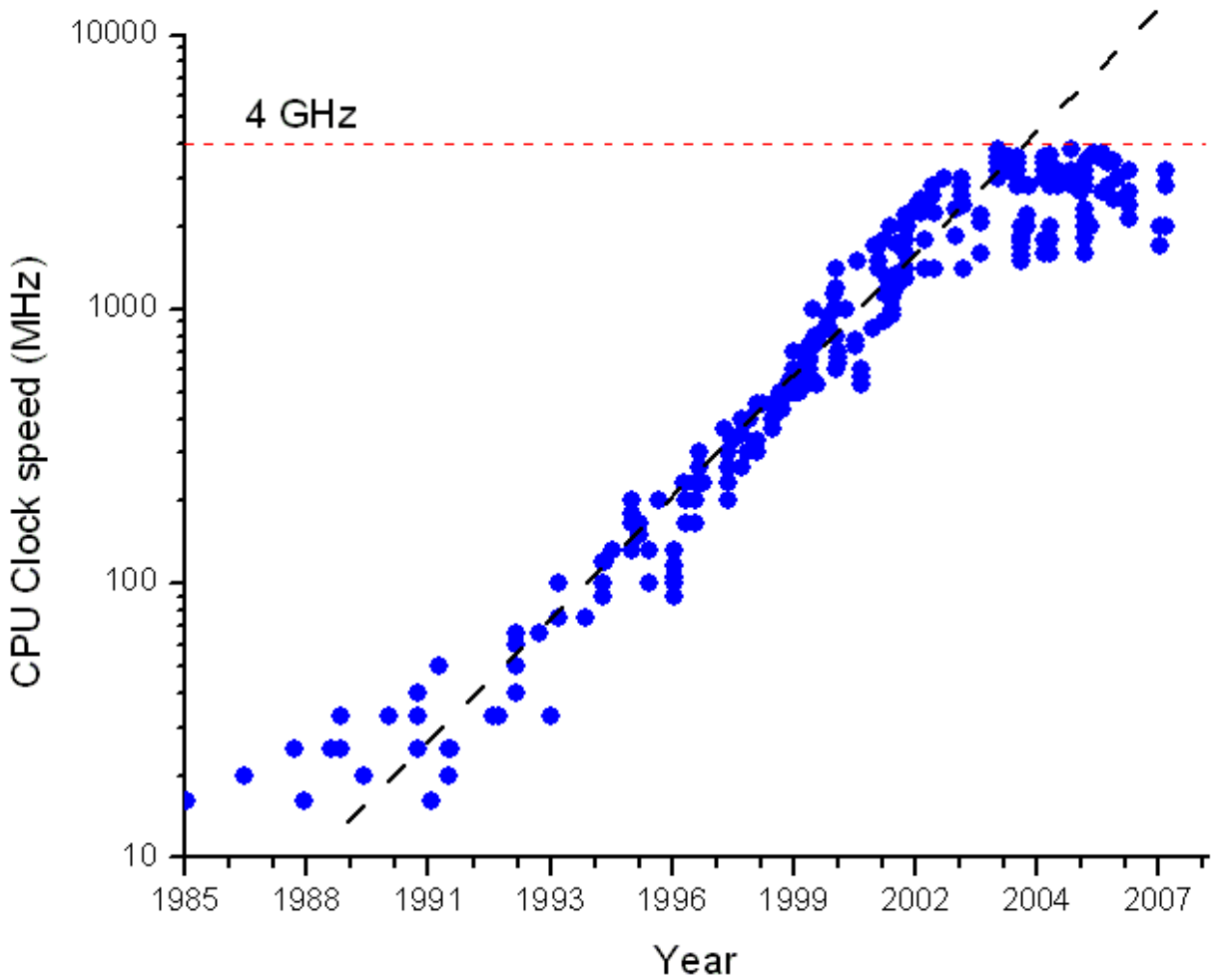
Parallel Programming: OpenMP + FORTRAN

John Burkardt
Virginia Tech
.....
FSU Department of Scientific Computing
"Introduction to Scientific Computing with FORTRAN"
.....
https://people.sc.fsu.edu/~jburkardt/presentations/...
openmp_2010_fsu.pdf

15 April 2010

INTRO: Clock Speed Ceiling

Computer processor speeds have hit physical limits.



INTRO: Parallel Programming

Sequential programming assumes the commands in a program are carried out one at a time. Sequential programs will never run faster than they did in 2002.

Parallel programming tries to take advantage of two facts:

- Many commands in a program could be carried out simultaneously
- It is possible to make multiple processors available to a program.

Sequential programming languages must be extended or replaced in order to make parallel programming possible.

INTRO: Extending Languages

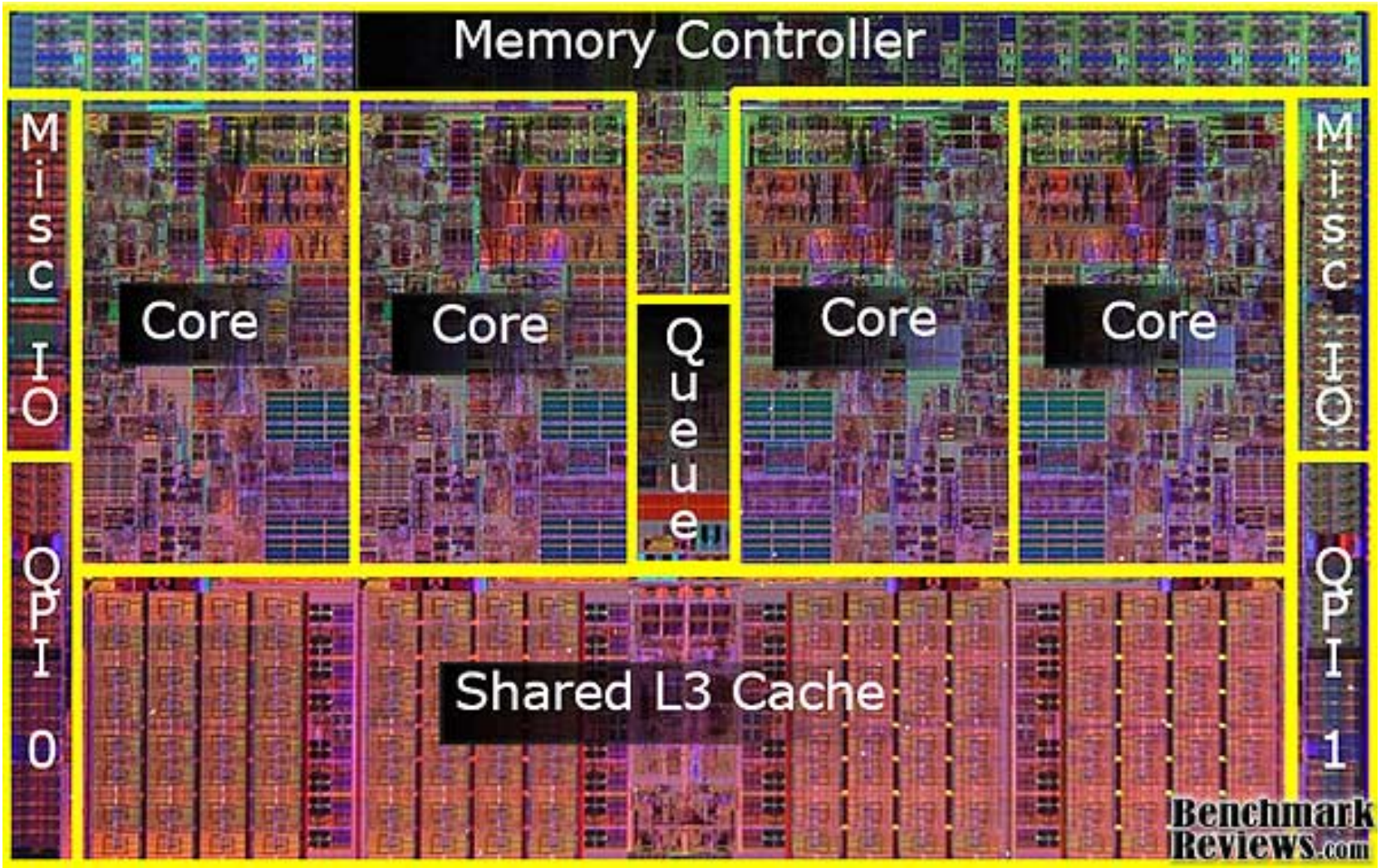
Creating a new parallel programming language is hard. Getting people to use it is almost impossible.

Language extensions offer a way to coax programmers and programs into parallel programming.

- **OpenMP** is a gentle modification to C and FORTRAN programs; a single sequential program can include parallel portions;
- **MPI** also works with C and FORTRAN; multiple copies of a program cooperate
- **MATLAB** has a Parallel Computing Toolbox from the MathWorks; there are also a free MPI-MATLAB, and a free "multicore" toolbox.

INTRO: We'll Look at OpenMP

We will look at how **OpenMP** can help to modify a program so it takes advantage of extra cores *(Intel Nehalem processor below)*.



HELLO: How Things Should Work

To feel comfortable with OpenMP, it's important to have a clear idea of how things happen.

We'll assume we have a quadcore computer, which means the processor includes 4 cores or mini-CPU's, each of which could be running a separate program. Each core has its own memory, and there is also a larger memory that all the cores can see.

Let's suppose that the user's program contains a **do** loop, involving 1000 iterations, and that the **i**-th iteration carries out some computation and stores the result in **x(i)**, and that these iterations could be done in any order, or simultaneously.

HELLO: How Things Should Work

To modify our program and run it with OpenMP, we would need to do the following things:

1. mark the **do** loop to run in parallel;
2. compile the program, indicating that we want to use OpenMP;
3. signal the number of parallel workers we want;
4. run the program in the ordinary way.

If we have used parallel programming correctly and wisely, then using 4 workers we could hope to run 4 times faster.

HELLO: What Goes On in the Program?

The next thing to wonder about is what is going on in the program.

A program compiled with OpenMP will begin execution in the same way as an ordinary sequential program.

However, at some point, the program will reach a block of statements that the user has marked as a parallel region. This parallel region contains a number of tasks that can be carried out in parallel.

At this point, the extra parallel workers are activated, the tasks are divided among them, and the program "waits" at the end of the block until all the tasks have been completed. Sequential execution resumes until another parallel region is encountered.

HELLO: Let's Say Hello!

Let's go through the steps of running a simple "Hello, World" program.

To mark a parallel region, we use comments of the form

```
!$omp parallel
  stuff to execute in parallel
!$omp end parallel
```

Anything inside the parallel region will be executed by all the *threads* (another name for parallel workers or processes or processors or cores).

HELLO: HELLO Source Code

```
program main

  implicit none
  write ( *, * ) 'A sequential hello to you!'

!$omp parallel

  write ( *, * ) '  Parallel hello''s to you!'

!$omp end parallel

  stop
end
```

HELLO: Making It Run

To make this program do its work, we do the following:

1. **compile**: gfortran -fopenmp hello.f90
2. **environment**: export OMP_NUM_THREADS=4
3. **run**: ./a.out

If you are using the Intel compiler, you would say:
ifort -openmp -parallel -fpp hello.f90

If you are using the C or T shells, you say:
setenv OMP_NUM_THREADS 4

HELLO: HELLO Output

```
A sequential hello to you!

  Parallel hello's to you!
  Parallel hello's to you!
  Parallel hello's to you!
  Parallel hello's to you!
```

HELLO: Helpful Functions

OpenMP provides a small number of useful functions:

- **omp_get_wtime()**, wall clock time;
- **omp_get_num_procs()**, number of processors available;
- **omp_get_max_threads()**, max number of threads available;
- **omp_get_num_threads()**, number of threads in use;
- **omp_get_thread_num()**, ID for this thread;

To use these functions, you need the statement:

```
use omp_lib
```

Let's redo HELLO, and use some of these functions.

HELLO: HELLO Again!

```
wtime = omp_get_wtime ( )
write ’  Available processors: ’, omp_get_num_procs ( )
write ’  Available threads     ’, omp_get_max_threads ( )
write ’  Threads in use        ’, omp_get_num_threads ( )
!$omp parallel private ( id )

  id = omp_get_thread_num ( )
  write ( *, * ) ’  Hello from process ’, id

  if ( id == 0 ) then
    write ’  Threads in use ’, omp_get_num_threads ( )
  end if

!$omp end parallel
wtime = omp_get_wtime ( ) - wtime
write ’  Wtime = ’, wtime
```

HELLO: HELLO Again Output

```
Available processors:                2
Available threads                    2
Threads in use                       1

  Hello from process                 0
  Hello from process                 1
  Threads in use                     2

Wtime =    0.732183E-03
```

HELLO: Private? What's That?

There's one item not explained in the previous example.
Why did I mark the beginning of the parallel region this way:

```
!$omp parallel private ( id )
```

OpenMP is based on the idea of *shared memory*. That is, even though multiple threads are operating, they are expected not to get in each other's way.

When variables are being computed, we have to make sure that

- only one thread sets a particular variable, or
- only one thread at a time sets a variable, and "puts it back" before another thread needs it, or
- *if the threads can't share the data*, then each thread gets its own private copy of the variable.

SAXPY: Tasks Inside the Parallel Region

The Hello example showed us how to print the same message 4 times. But how do we organize the workers to carry out different parts of a task?

Inside of a parallel region, you are allowed to have one or more "blocks" of code to be executed:

```
!$omp parallel
  PARALLEL TASK 1 (do loop on I)
  PARALLEL TASK 2 (sections)
  PARALLEL TASK 3 (a nested do loop on I and J)
!$omp end parallel
```

By default, the threads cooperate on task 1. When all are done, they move to task 2, and so on. (There's a "nowait" clause that can let fast threads move on.)

SAXPY: The DO Directive

The do directive indicates a given loop in the parallel region should be executed in parallel:

```
!$omp parallel
!$omp do
  ...parallelized DO loop here...
!$omp end do
  ..more blocks could follow...
!$omp end parallel
```

If the parallel region only contains a single DO loop, you can use a shortened directive:

```
!$omp parallel do
  ...ONE parallelized DO loop here...
!$omp end parallel do
```

SAXPY: DO Example

If loops are nested, you only parallelize one index!

```
!$omp parallel
  !$omp do
    do i = 1, nedge
      do j = 1, nface
        This nested loop is parallel in I
      end do
    end do
  !$omp end do
    do i = 1, nedge
    !$omp do
      do j = 1, nface
        This nested loop is parallel in J
      end do
    !$omp end do
    end do
$omp end parallel
```

SAXPY: The SECTION Directive

Another kind of block is described by the sections directive.

It's somewhat like a **case** statement. You simply have several sets of statements to execute, and you want each set of statements to be executed by exactly one thread.

The group is marked by sections, and each subgroup by **section**.

If there are more sections than threads, some threads will do several sections.

Any extra threads will be idle.

SAXPY: SECTION Syntax

```
!$omp parallel                  <-- inside "parallel"
  ...                           <-- other parallel tasks
  !$omp sections (nowait)       <-- optional nowait
    !$omp section
      code for section 1
    !$omp section
      code for section 2
                                <-- more sections
                                    could follow
  !$omp end sections
  ...                         <-- other parallel tasks
!$omp end parallel
```

SAXPY: Section Example

A Fast Fourier Transform (FFT) computation often starts by computing two tables, containing the sines and cosines of angles.

Sections could be used to set up these tables more quickly:

```
!$omp parallel
  !$omp sections
    !$omp section
      call sin_table ( n, s )
    !$omp section
      call cos_table ( n, c )
  !$omp end sections
!$omp end parallel
```

SAXPY: The WORKSHARE Directive

A third kind of task that can be included in a parallel region involves particular FORTRAN commands:

- array operations that use colon notation
- the **where** statement,
- the **forall** statement

To indicate that such an operation or block should be done in parallel, it is marked with the workshare directive.

SAXPY: Workshare for Colon and WHERE

```
!$omp parallel

  !$omp workshare
    y(1:n) = a * x(1:n) + y(1:n)
  !$omp end workshare

  !$omp workshare
    where ( x(1:n) /= 0.0 )
      y(1:n) = log ( x(1:n) )
    elsewhere
      y(1:n) = 0.0
    end where
  !$omp end workshare

!$omp end parallel
```

SAXPY: Workshare for FORALL

```
!$omp parallel
  !$omp workshare

    forall ( i = k+1:n, j = k+1:n )
      a(i,j) = a(i,j) - a(i,k) * a(k,j)
    end forall

  !$omp end workshare
!$omp end parallel
```

(This calculation corresponds to one of the steps of Gauss elimination or LU factorization)

SAXPY: OpenMP Program with DO

```
program main
  integer i
  integer, parameter :: n = 1000
  double precision :: s = 123.456
  double precision x(n), y(n)
  call random_number ( harvest = x(1:n) )
  call random_number ( harvest = y(1:n) )
!$omp parallel
  !$omp do
    do i = 1, n
      y(i) = y(i) + s * x(i)
    end do
  !$omp end do
!$omp end parallel
  stop
end
```

SAXPY: Program Comments

The SAXPY program can run in parallel, as long as every thread can "see" the value of **S**, and can grab its desired entries of **X** and **Y**, and can store the new value of **Y** without interfering with any other thread.

And all this works, and the threads "share" the variables just fine.

However, there's one variable that *cannot* be shared. It's a variable each thread needs and modifies. It's only needed temporarily, and its value before and after the loop is not important, but during the loop, it makes everything happen.

Each thread must get a private copy of the DO loop index **I**, so that they have a way of keeping track of what they are doing.

By default, OpenMP automatically makes the index of the parallelized loop a private variable. This is the second example of a private variable. It may be time to try to understand them!

PI: Data Conflicts

Sometimes OpenMP can't parallelize a loop because of the way data is handled.

The program might be written in such a way that the results of one iteration are used in the next iteration.

Sometimes, this is an inherent feature of the algorithm or problem (such as solving a differential equation) and that part of the algorithm might never run in parallel.

But often the code can be reworded so the difficulty goes away.

PI: Data Conflicts

Suppose we compute a table of factorials this way:

```
fact(0) = 1;
do i = 1, n
  fact(i) = fact(i-1) * i;
end do
```

Even though you might suspect this operation could take advantage of parallelism, it cannot be done if the code is written this way! Even if we had so many threads that we could assign each iteration to its own thread, only one thread can begin (number 1); only when it has computed **fact(1)** can thread 2 begin, and thread **n** has to wait

I invite you to figure out how to compute a table of N factorial values in log(N) parallel steps!

PI: Which of these loops are "safe"?

```
do i = 2, n - 1
   y(i) = ( x(i) + x(i-1) ) / 2          Loop #1
end do

do i = 2, n - 1
   y(i) = ( x(i) + x(i+1) ) / 2          Loop #2
end do

do i = 2, n - 1
   x(i) = ( x(i) + x(i-1) ) / 2          Loop #3
end do

do i = 2, n - 1
   x(i) = ( x(i) + x(i+1) ) / 2          Loop #4
end do
```

PI: Data Conflicts

Another difficulty arises because OpenMP shares a single memory space for multiple threads. What could go wrong with the following loop if all variables are shared (except for **i**)?

```
do i = 1, n
  t = sqrt ( a(i,1)**2 + a(i,2)**2 + a(i,3)**2 )
  x(i) = x(i) / t
end do
```

The variable **t** is a "convenience" or "temporary" variable. Its value is only briefly needed, but each iteration will put a different value into **t**. If we only have one variable called **t**, then thread 1 might store a value there, then thread 2 might store a different value, and only then might thread 1 come back to get the (wrong) value to use in the next statement.

PI: Data Conflicts

The **private** clause allows us to indicate that certain variables should be stored by each thread as local, private copies. As long as OpenMP is informed that this is what we want, the computation can proceed correctly:

```
!$omp parallel private(t,i)
  !$omp do
  do i = 1, n
    t = sqrt ( a(i,1)**2 + a(i,2)**2 + a(i,3)**2 )
    x(i) = x(i) / t
  end do
  !$omp end do
!$omp end parallel
```

(OpenMP made **i** private, but we must declare **t** too).

PI: Directive Clauses

Clauses are additional information included on a directive.

Inside of a parallel region, OpenMP assumes (unless we say otherwise) that all variables are shared, except for indices corresponding to parallelized DO loops.

Using the **private** clause, we can indicate that certain variables are actually to be stored privately by each thread. Typically, such variables are "temporaries" or other convenience variables whose values whose values change from one iteration to the next, but which aren't important once the loop is finished.

PI: Sequential Code

```
h = 1.0 / dble ( 2 * n )
q = 0.0
do i = 1, n
  x = h * dble ( 2 * i - 1 )
  q = q + 1.0 / ( 1.0 + x * x )
end do
q = 4.0 * q / dble ( n )
```

PI: Another Problem!

It's pretty clear that the convenience variable **x** must not be shared, but will be fine if we make it private.

But now there's another tricky issue. What is the status of the variable **q**? It is not a temporary variable. Its value is important outside the loop. On the other hand, just like a temporary variable, its value is altered on every iteration, by every iteration.

The variable **q** is a funny kind of mixture between shared and private. We'll use yet one more clause for that, and explain it later!

PI: Directive Clauses

```
  h = 1.0 / dble ( 2 * n )
  q = 0.0
!$omp parallel shared ( h, n ) private ( i, x ) &
!$omp reduction ( + : q )

!$omp do
  do i = 1, n
    x = h * dble ( 2 * i - 1 )
    q = q + 1.0 / ( 1.0 + x * x )
  end do
!$omp end do
!$omp end parallel
  q = 4.0 * q / dble ( n )
```

PI: Reduction Operations

We've just seen one example of a **reduction operation**.

These include:

- the sum of the entries of a vector,
- the product of the entries of a vector,
- the maximum or minimum of a vector,
- the Euclidean norm of a vector,

Reduction operations, if recognized, can be carried out in parallel.

The **OpenMP reduction** clause allows the compiler to set up the reduction correctly and efficiently. We must name the variable, and the type of reduction being performed. It's almost always of type "addition".

PI: The reduction clause

Any variable which will contain the result of a reduction operation must be identified in a **reduction** clause of the **OpenMP** directive.

Examples include:

- **reduction (+ : q)** (we just saw this)
- **reduction (+ : sum1, sum2, sum3)** , (several sums)
- **reduction (* : factorial)**, a product
- **reduction (max : pivot)** , maximum value)

Inside a parallel region, every variable is exactly one of the following: **shared**, **private** or **reduction**.

The MD Example: A Real Computation

The MD program simulates the behavior of a box full of particles.

The user picks the number of particles, and the number of time steps.

The program gives the particles random positions and velocities for time step 0.

In order to compute data for the next time step, it must sum up the force on each particle from all the other particles.

This operation is completely parallelizable.

Because this is a large computation, you are much more likely to see a speedup as you go from sequential to parallel execution.

The MD Example: Example Programs

Examples available at

http://people.sc.fsu.edu/~burkardt/f_src/f_src.html:

- **fft_open_mp** (Fourier transform)
- **md_open_mp** (molecular dynamics)
- **mxv_open_mp** (matrix times vector)
- **open_mp** (compute_pi, dot_product, hello, helmholtz)
- **quad_open_mp** (estimate integral)
- **satisfiability_open_mp** (seek solutions to logical formula)
- **sgefa_open_mp** (Gauss elimination)
- **ziggurat_open_mp** (random numbers)

The MD Example: References

References:

1. **Chandra**, *Parallel Programming in OpenMP*
2. **Chapman**, *Using OpenMP*
3. **Petersen, Arbenz**, *Introduction to Parallel Programming*
4. **Quinn**, *Parallel Programming in C with MPI and OpenMP*

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