Introduction to OpenMP Programming

John Burkardt: burkardt@vt.edu
Advanced Research Computing
Virginia Tech
3:00-4:00pm, 27 October 2017
1100 Torgersen Hall

Slides available at
https://secure.hosting.vt.edu/www.arc.vt.edu/class_note/

OVERVIEW: Coverage

This course shows:

- many science problems include parallel opportunities;
- shared memory threads make a simple model of parallelism;
- OpenMP enables shared memory threads for C/C++/Fortran;
- loops can be easy to parallelize;
- OpenMP programs can run on your laptop, or an ARC system;
- it's easy to check whether your program runs faster.

No prior knowledge of OpenMP is assumed.

Experience writing or using C, C++ or Fortran programs is useful.

OVERVIEW: A Parallel Model

A simple model of parallelism is called SIMD, for "Single Instruction, Multiple Data".

On an ancient galley, each time the drum was struck, each prisoner had to carry out a stroke of the oar.

In SIMD parallelism, a fixed sequence of operations is to be carried out repeatedly, on a set of data. We assume several "cores" are available, each of which can simultaneously process a distinct subset of the data.

It is natural to hope for a great speedup in execution, if the cores can cooperatively divide up the tasks, carry out their work without interfering with each other, and combine their results at the end.

The results should be the same as if a single core had done the work in the ordinary, sequential fashion.

PARALLEL: Parallel Problems Obviously Exist

Many problems can be handled with some parallel approach:

- search for a matching item in a list, such as protein sequences;
- find the maximum value in an array;
- sort a set of items;
- smooth or sharpen or filter pixels in an image;
- iterative solution of linear equations;
- approximation of PDE solutions;

HARDWARE: Multicore Memory Processors

OpenMP is inspired by the development of chips containing a single chunk of memory and several processors or cores.

These cores could be running independent programs, in which case the memory is split into separate unshared regions.

But a single program could run, use all of the memory, and somehow orchestrate the cores to cooperate on one calculation.

Thus, OpenMP has a hardware restriction: how many cores share a single memory address space?

(Lovers of OpenMP have tried to get around this restriction by constructing NUMA machines, in which multiple chips are strung together and tricked into behaving like one object with enormous memory and thousands of cores.)

LOOP: OpenMP Concentrates on FOR and DO Loops

There are several ways that OpenMP allows you to create parallel programs. The simplest way in which OpenMP can implement parallelism is to change the way a loop is executed in a C/C++/Fortran program.

The idea is that:

- a loop can represent a lot of work (many iterations or many instructions, so parallelization is **effective**);
- each loop iteration is the same instructions, so parallelization is **easy**;
- if each loop iteration is independent, parallelization is **correct**.

If these items are true, then the loop iterations can be split into multiple threads and executed in parallel.

LOOP: Loop Iterations Are Divided Among Threads

For example, we might imagine that the sequential loop:

```
for ( i = 0; i < 1000; i++ )
{
  x[i] = x[i] + s * y[i];
}
```

could be split into two threads as:

```
Thread #0                          |                Thread #1
                                   |
for ( i = 0; i < 500; i++ )        |  for ( i = 500; i < 1000; i++ )
{                                  |  {
  x[i] = x[i] + s * y[i];          |    x[i] = x[i] + s * y[i];
}                                  |  }
```

It's easy to extend this to more threads.

LOOP: Indexing Can Be More Complicated

When multiple threads can interfere with each other.

```
for ( i = 1; i <= 1000; i++ )
{
  x[i]   = x[i] + sqrt ( y[i-1] ); <--   This is OK
}
for ( i = 1; i <= 1000; i++ )
{
  x[i] = x[i] + x[i-1];          <-- This will fail!
}
for ( i = 0; i <= 1000; i = i + 2 )
{
  z[i]   = sin ( i * pi );
  z[i+1] = cos ( i * pi );       <-- This is OK
}
```

One test: does the sequential version of the loop work exactly the same if we do the iterations in reverse order?

LOOP: "Left Hand Side" Variables Can Conflict

Problems occur if more than one loop iteration tries to write or modify the same variable, which occurs on the left hand side of a statement.

Here, we have a **y** vector; we'd want to add half of each entry to the corresponding "left" entry in **x** and half to the right.

```
for ( i = 1; i < n - 1; i++ )
{
  x[i-1] = x[i-1] + 0.5 * y[i];
  x[i+1] = x[i+1] + 0.5 * y[i];
}
```

Even in our simple two-thread model, this code will have the potential of conflicts. Suppose that **n**=1000. Thread #0 might try to execute the second addition for $i = 499$ while thread #1 is executing the first addition for $i = 501$.

LOOP: "Left/Right Hand Side" Variable Problems

Inside a loop that depends on the index **i**, it is "safe" to read and write vector entries indexed by **i**. But any read or write to entries with other indices is likely to cause problems for parallel execution.

This code is overwriting **x** by its cumulative sums, and will not perform correctly if attempted in parallel:

```
for ( i = 1; i < n; i++ )
{
  x[i] = x[i] + x[i-1];
}
```

For instance, if x = { 1, 1, 1, 1, 1 }, the result should be {1, 2, 3, 4, 5 }, but parallel execution might get {1, 2, 2, 2, 2}. That's what you would get in MATLAB with the statement:

```
x(2:5)= x(2:5)+ x(1:4);
```

LOOP: Another Example of Side Effects

Another dangerous practice involves temporary variables that are updated during the loop iteration.

For example, let's plot the function $y = x^2$ between 0 and 1, by evaluating the function at 1001 equally spaced points:

```
x = 0.0;
for ( i = 0; i <= 1000; i++ )
{
  y[i] = x * x;
  x = x + 0.001;
}
```

The above loop can't execute correctly in parallel, but we can easily fix it:

```
for ( i = 0; i <= 1000; i++ )
{
  x = i * 0.001;
  y[i] = x * x;
}
```

LOOP: A Summation

Here's a similar example, (approximating an integral) which is actually important enough that we will see how to fix it later:

```
n = 1000;
q = 0.0;
for ( i = 0; i < n; i++ )
{
  x = i / ( double ) n;
  q = q + x * x;
}
q = q / n;
```

In this loop, **x** is not the problem, it's **q**, which is being modified on every iteration. In our two thread parallel version, would we have two separate variables called **q**? If so, what do we do with them at the end? If there's just one variable, and the threads have to share it, then how do we avoid conflicts?

LOOP: Another Example of Left/Right Variables

Problems can occur if data appears on both the left and right hand side, and so is changed during the calculation.

Here is a sort of Gauss-Seidel iteration (for the -1,2,-1 matrix) for solving a linear system.

```
x[0] = ( b[0] + x[1] ) / 2.0;
for ( i = 1; i < n - 1; i++ )
{
  x[i] = ( b[i] + x[i-1] + x[i+1] ) / 2.0;
}
x[n-1] = ( b[n-1) + x[n-2] ) / 2.0;
```

Can you see that this loop will not get the same results if it is run in parallel by, say, two threads?

LOOP: Simple Rules for Parallel Loops

In summary,

- If we plan to run a loop in parallel, it should be written in such a way that the loop iterations would get the same results, even if they were executed in the reverse order, or any order;
- We need to avoid cases in which the same variable is modified by two different iterations of the loop;
- Some loops, like the integral approximation, use a single variable to collect results from all the iterations. If we want to use such methods, we need to come up with a special approach.

MODEL: OpenMP Enables Helper Threads

When part of a computational problem can be done in parallel, OpenMP can be used to "annotate" the corresponding computer program.

Parallel parts of the program are indicated by a **parallel** block.

Inside the parallel block, an OpenMP directive may be used to indicate that one or more loops should be handled by multiple threads.

By default, OpenMP assumes that all data is shared in common.

But in order for the threads to run different iterations, they each need their own separate loop index variable. Such a variable is said to be private.

SAXPY: Add a multiple of one vector to another

Now we are ready to consider how OpenMP can be used to parallelize a simple programming task.

Our example starts with an **n**-vector called **x** and adds to it the vector **y**, multiplied by the scalar **s**:

$$\vec{x} \leftarrow \vec{x} + s \cdot \vec{y}$$

We assume that the values of **x** and **y** are set by some formula, about which we don't really care that much.

Multiple threads can carry out this task, as long as they each know what index range to handle, can read values of **s**, **x**, and **y**, and can write updated values of **x** without interfering with each other.

SAXPY: C Example (Before)

```
int main ( )
{
  int i, n = 1000;
  double s = 1.23, x[1000], y[1000];




  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) ( ( i + 1 ) % 17 );
    y[i] = ( double ) ( ( i + 1 ) % 31 );
  }


  for ( i = 0; i < n; i++ )
  {
    x[i] = x[i] + s * y[i];
  }

  return 0;
}
```

SAXPY: C Example (After)

```
int main ( )
{
  int i, n = 1000;
  double s = 1.23, x[1000], y[1000];

# pragma omp parallel private ( i )    <-- Allow parallelism.
{                                          Variable i is private
# pragma omp for                       <-- Execute this loop using threads
  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) ( ( i + 1 ) % 17 );
    y[i] = ( double ) ( ( i + 1 ) % 31 );
  }

# pragma omp for                       <-- Execute this loop using threads
  for ( i = 0; i < n; i++ )
  {
    x[i] = x[i] + s * y[i];
  }
}                                      <-- End parallelism
  return 0;
}
```

COMMENT: FORK/JOIN Parallelism

OpenMP uses a kind of parallelism called **fork/join**. A single "master" thread is active until a parallel region is entered.

Inside a parallel region, all the threads will execute all the statements... which is NOT what you want, *unless you tell them to split up the work!*, using the statement

```
# pragma omp for
```

This tells the threads to split up the loop iterations of the next **for** loop, and execute them in parallel. This is where the main speedup of OpenMP comes from.

When a parallel region is exited, the helper threads become inactive and the master thread continues on its own (a "join").

COMMENT: Private Data

If multiple threads are executing different loop iterations at the same time, there is a potential for disaster.

Start with the loop counter **i**; each thread relies on the value of this variable to tell it what to do next. To avoid problems, we need to ensure that each thread gets a private copy of this variable that the other threads can't see or change.

By default, most variables will be shared, but it is the user's responsibility to specify, as part of the **parallel** statement, which variables in the parallel region should have the special **private** attribute.

I list the attributes of all variables in the region, just to be clear.

Typically, operations on vectors and arrays don't cause too many problems inside a loop, and such data can be shared. But often certain scalar calculations have to be handled specially so as not break parallelism.

COMMENT: Private Data

Suppose we wish to carry out a rotation operation on vectors X and Y:

```
[  C  S ] * [ X0 X1 X2 ... Xn-1 ]
[ -S  C ]   [ Y0 Y1 Y2 ... Yn-1 ]
```

A natural way to write this is:

```
for ( i = 0; i < n; i++ )
{
   t    =  c * x[i] + s * y[i];
   y[i] = -s * x[i] + c * y[i];
   x[i] =  t;
}
```

We need the temporary variable **t** to hold the new value of **x[i]**, until we've had time to use the old value of **x[i]** to update **y[i]**.

COMMENT: Private Data

To do this loop in parallel, **t** must be made private.

```
# pragma omp parallel \
  private ( i, t ) \
  shared ( c, s, x, y )
{
# pragma omp for
  for ( i = 0; i < n; i++ )
  {
     t    =  c * x[i] + s * y[i];
     y[i] = -s * x[i] + c * y[i];
     x[i] =  t;
  }
}
```

NESTING: Jacobi

Assume the vector x is initialized. How can we parallelize the following Jacobi iteration for solving $A * x = b$?

```
for ( k = 0; k < 1000; k++ )                    <-- Sequential
{
  y[0] = ( b[0] + x[1] ) / 2.0;
  for ( i = 1; i < n - 1; i++ )                 <-- Can be parallel
  {
    y[i] = ( b[i] + x[i-1] + x[i+1] ) / 2.0;
  }
  y[n-1] = ( b[n-1) + x[n-2] ) / 2.0;
  for ( i = 0; i < n; i++ )                     <-- Can be parallel
  {
    x[i] = y[i];
  }
}
```

NESTING: Jacobi

```
for ( k = 0; k < 1000; k++ )
{
# pragma omp parallel private ( i ) shared ( n, b, x, y )
  {
    y[0] = ( b[0] + x[1] ) / 2.0;
    # pragma omp for
    for ( i = 1; i < n - 1; i++ )
    {
      y[i] = ( b[i] + x[i-1] + x[i+1] ) / 2.0;
    }
    y[n-1] = ( b[n-1) + x[n-2] ) / 2.0;
    # pragma omp for
    for ( i = 0; i < n; i++ )
    {
      x[i] = y[i];
    }
  }
}
```

DENOISE: Salt and Pepper Noise

DENOISE: Salt and Pepper Noise

We propose to repair the damage to the image using a NEWS (North/East/West/South) median filter.

Each pixel has been assigned R, G and B values between 0 and 255.

For each component (R, G or B), compare the central pixel's value with those of its N, E, W and S neighbors, and replace by the median.

Not a perfect repair, but considerable improvement!

DENOISE: Salt and Pepper Noise

DENOISE: Original coding

```
for ( i = 1; i < m - 1; i++ )
{
  for ( j = 1; j < n - 1; j++ )
  {
    p[0] = color[i-1+ j   *m];     <-- north
    p[1] = color[i+1+ j   *m];     <-- south
    p[2] = color[i  +(j+1)*m];     <-- east
    p[3] = color[i  +(j-1)*m];     <-- west
    p[4] = color[i  + j   *m];     <-- central

    qsort ( p, 5, sizeof ( int ), int_cmp );

    color2[i+j*m] = p[2];          <-- replace by median
  }
}

for ( i = 1; i < m - 1; i++ )
{
  for ( j = 1; j < n - 1; j++ )
  {
    color[i+j*m] = color2[i+j*m];
  }
}
```

DENOISE: recoding with OpenMP

```
# pragma omp parallel \
  private ( i, j, p ) shared ( m, n, color, color2 )
{
  # pragma omp for
  for ( i = 1; i < m - 1; i++ )
  {
    for ( j = 1; j < n - 1; j++ )
    {
      p[0] = color[i-1+ j   *m];     <-- north
      p[1] = color[i+1+ j   *m];     <-- south
      p[2] = color[i  +(j+1)*m];     <-- east
      p[3] = color[i  +(j-1)*m];     <-- west
      p[4] = color[i  + j   *m];     <-- central

      qsort ( p, 5, sizeof ( int ), int_cmp );

      color2[i+j*m] = p[2];          <-- replace by median
    }
  }
  # pragma omp for
  for ( i = 1; i < m - 1; i++ )
  {
    for ( j = 1; j < n - 1; j++ )
    {
      color[i+j*m] = color2[i+j*m];
    }
  }
}
```

DENOISE: Timing on a 16-core machine

```
export OMP_NUM_THREADS=1 (or 2 or 4 or 8 or 16)

t1 = omp_get_wtime ( );
filter R, G and B
t2 = omp_get_wtime ( );
print t2-t1 seconds
```

Threads	Filter Time (seconds)
1	0.2564
2	0.1354
4	0.0699
8	0.0372
16	0.0395

REDUCTION: Integral

When we estimate an integral, the summation variable crosses the line between private and shared.

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

double f ( double x );

int main ( )
{
  double a = 1.0, ai, b = 100.0, bi, q, x;
  int i, n = 1000;

  q = 0.0;
  for ( i = 0; i < n; i++ )
  {
    ai = ( ( n - i     ) * a + ( i     ) * b ) / n;
    bi = ( ( n - i - 1 ) * a + ( i + 1 ) * b ) / n;
    x = 0.5 * ( ai + bi );
    q = q + ( bi - ai ) * f ( x );
  }
  printf ( "Integral estimate = %g\n", q );

  return 0;
}
```

REDUCTION: Integral

q is treated as a "reduction" variable. Each thread works on a private copy; on loop completion these are all summed to a single shared copy.

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

double f ( double x );

int main ( )
{
  double a = 1.0, ai, b = 100.0, bi, q, x;
  int i, n = 1000;

  # pragma omp parallel private ( ai, bi, i, x )
  {
    # pragma omp for reduction ( + : q )

    q = 0.0;
    for ( i = 0; i < n; i++ )
    {
      ai = ( ( n - i     ) * a + ( i     ) * b ) / n;
      bi = ( ( n - i - 1 ) * a + ( i + 1 ) * b ) / n;
      x = 0.5 * ( ai + bi );
      q = q + ( bi - ai ) * f ( x );
    }
  }
  printf ( "Integral estimate = %g\n", q );

  return 0;
}
```

RUN: Compiler Switches Activate OpenMP

GNU:

- **gcc** *myprog.c* **-fopenmp**
- **g++** *myprog.cpp* **-fopenmp**
- **gfortran** *myprog.f* **-fopenmp**
- **gfortran** *myprog.f90* **-fopenmp**

Intel:

- **icc** *myprog.c* **-openmp -parallel**
- **icpc** *myprog.cpp* **-openmp -parallel**
- **ifort** *myprog.f* **-openmp -parallel -fpp**
- **ifort** *myprog.f90* **-openmp -parallel -fpp**

Portland Group:

- **pgcc** *myprog.c* **-mp**
- **pgc++** *myprog.cpp* **-mp**
- **pgf77** *myprog.f* **-mp**
- **pgf95** *myprog.f90* **-mp**

RUN: Threads Versus Processors

OpenMP knows how many processors (cores) are available on the system.

However, when you want to run in parallel, you actually specify how many threads you want; this is the number of parallel tasks to be carried out at one time, and usually means how you want to "slice up" your loop.

Using 1 thread means sequential execution.

Asking for 2 threads means the work will be split into two chunks, and it will be done in parallel if there are at least two processors available.

Similarly, we can ask for 8 threads, but we might only have four processors. In that case, each processor will handle two threads.

It usually makes sense to ask for the number of threads to be the number of processors; *occasionally* you can get a speedup by having twice the number of threads.

RUN: Specifying the Number of Threads

When we run a program whose OpenMP directives have been activated, then OpenMP looks for the value of an environment variable called **OMP_NUM_THREADS** to determine the default number of threads.

You can query this value by typing:

```
echo $OMP_NUM_THREADS
```

A blank value is the same as 1. Usually, however, it's set to a sensible value, such as the number of cores available.

You can reset this environment variable using a command like:

```
export OMP_NUM_THREADS=4   <-- (No spaces around equal sign!)
```

and this new value will hold for any programs you run interactively.

RUN: Trying Different Numbers of THreads

Changing the number of threads is easy, and can be done at run time.

```
export OMP_NUM_THREADS=1
./myprog
export OMP_NUM_THREADS=2
./myprog
export OMP_NUM_THREADS=4
./myprog
export OMP_NUM_THREADS=8
./myprog
```

It's legal, but usually pointless, to define more threads than the number of available cores.

RUN: ARC Systems Available

You can run OpenMP jobs on ARC systems.

Since an OpenMP job requires shared memory, it can typically only run on one node of a system; on that node, however, it can ask for all the cores.

System	Nodes	Cores per node
BlueRidge	408	16
Cascades	196	32
DragonsTooth	48	24
Huckleberry	14	40
NewRiver	134	24

See, for instance,
https://secure.hosting.vt.edu/www.arc.vt.edu/computing/cascades/

RUN: Running OpenMP on ARC Systems

You need to request an account on a system, such as Dragonstooth.

You need to transfer your program file using **sftp** or **scp**.

```
sftp username@dragonstooth1.arc.vt.edu
put myprog.c
```

You can log into the login nodes dragonstooth1 or dragonstooth2 and run a small, short job interactively:

```
gcc -o myprog myprog.c -fopenmp  <-- Compile the program
export OMP_NUM_THREADS=24        <-- Set number of threads
./myprog                         <-- Run the program
```

RUN: Running OpenMP on ARC Systems

For production jobs, you must write a batch script, perhaps "myprog.sh", that sends your job for execution on a non-interactive compute node. This must also be copied to dragonstooth1 or dragonstooth2.

In order to run the job, you should be logged into dragonstooth, and you should be in the same directory as your program "myprog.c" and the script "myprog.sh".

To submit the job for execution, type:

```
qsub myprog.sh
122989.dt-scheduler.dt.arc.internal  <-- system response
                                     <-- Job number is 122989
```

Now you must wait, a short time or a long time, for the scheduler to decide when to run your job, and then for the job to run.

```
checkjob -v 122989      <-- to check status of job
qdel 122989             <-- to kill the job
showq -u username        <-- to show all your queued jobs
```

RUN: A Batch Script

The script *myprog.sh* might look something like this:

```
#! /bin/bash
#PBS -l walltime=00:05:00          <-- 5 minutes max
#PBS -l nodes=1:ppn=24             <-- request 1 node, all cores
#PBS -W group_list=dragonstooth
#PBS -q open_q                     <-- Use small ‘‘free’’ queue

cd $PBS_O_WORKDIR                  <-- Move to directory from
                                       which job was submitted.


module purge                       <-- Unload all modules
module load gcc                    <-- Load the gcc compiler

gcc -o myprog myprog.c -fopenmp    <-- Compile program
export OMP_NUM_THREADS=24          <-- Specify threads
time ./myprog                      <-- Run program and
                                       report required time
```

RUN: Running OpenMP on ARC Systems

Once your job has executed, the output will appear in the directory associated with the job, with a name constructed from the batch file name and the job number.

In our example this would be:

```
myprog.sh.o122989   <-- the output file
myprog.sh.e122989   <-- the error file
```

Since I would prefer to get just one output file, I often add to my batch file the following command to merge the two:

```
#PBS -j oe      <-- join the output and error files
```

in which case, all my information would be in one file:

```
myprog.sh.o122989   <-- the output AND error file
```

TIME: Simple Performance Analysis

The reason to use OpenMP is so your program runs faster. It's important to be able to reliably measure your program's speed as you ask for more and more threads.

A simple way to do this uses the **time** command:

```
export OMP_NUM_THREADS=1
time ./myprog
export OMP_NUM_THREADS=2
time ./myprog
export OMP_NUM_THREADS=6
time ./myprog
```

The output includes the total time, plus a breakdown into *user time* (your fault) and *sys time* (the system's fault). Usually, the lump sum value is good enough.

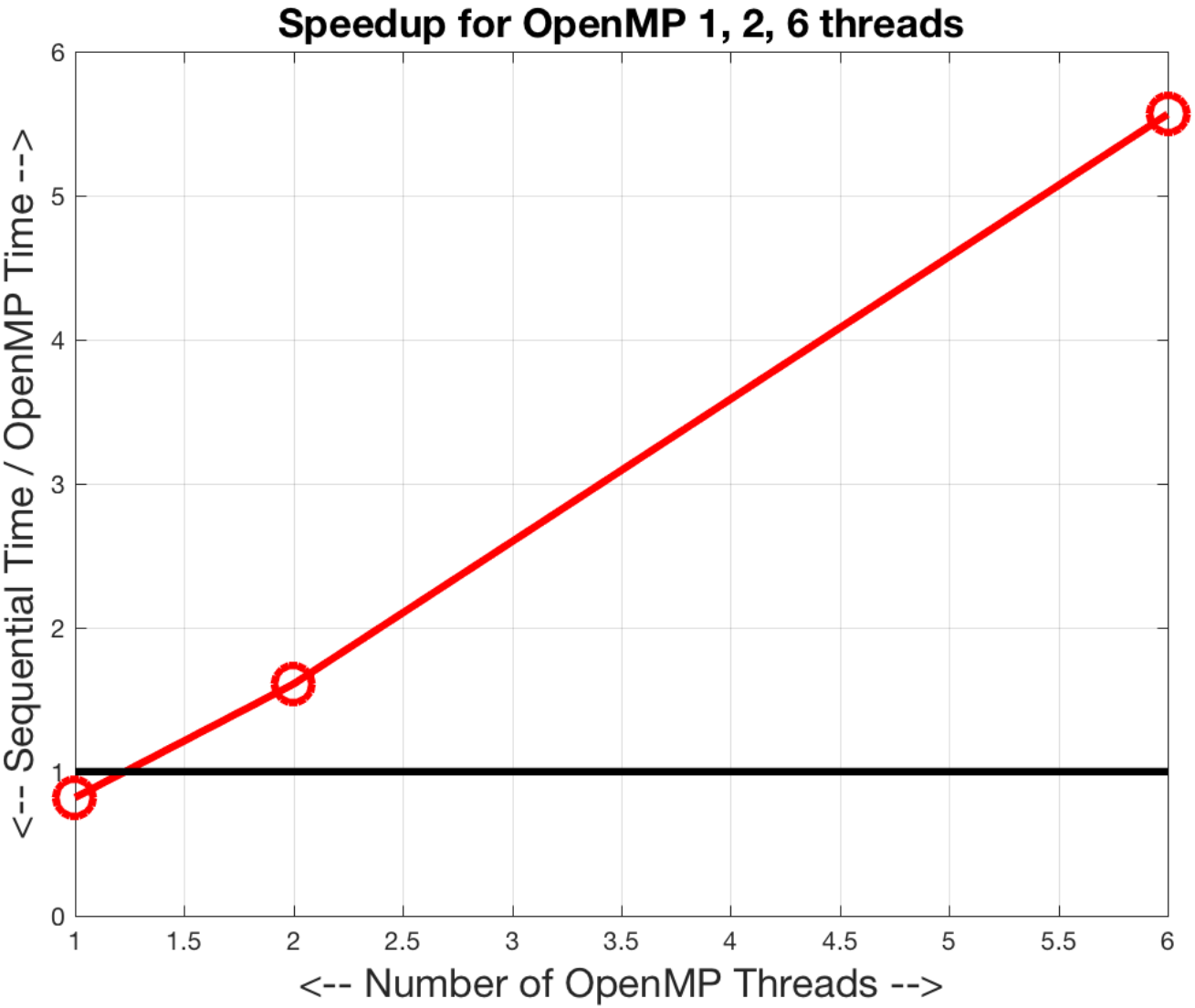
TIME: Output from TIME:

Here's an example of what **time** told me for a sample calculation:

```
Run sequentially (Compiled without OpenMP option.)
real    0m5.499s
user    0m5.492s
sys     0m0.004s

Run with 1 thread. (Compiled with OpenMP option.)
real    0m6.704s
user    0m6.694s
sys     0m0.009s

Run with 2 threads.
real    0m3.418s
user    0m6.823s
sys     0m0.009s

Run with 6 threads.
real    0m0.988s
user    0m5.902s
sys     0m0.015s
```

TIME: Is My Program Running Faster?

Plot the number of threads T as the X axis, and the ratio (time with 1 thread)/(time with T threads) on the Y axis, which is your speedup.



TIME: Alternative

While the **time** command is a good first tool for measuring program speed, it can only time the entire program.

You might be interested in finer measurements

- How long did this single function take to execute?
- Did this loop run faster after I parallelized it?

To answer such questions, there is a special OpenMP function called **omp_get_wtime()**

```
seconds = omp_get_wtime ( );
operations to time;
seconds = omp_get_wtime ( ) - seconds;
```

TIME: Alternative

While the **time** command is a good first tool for measuring program speed, it can only time the entire program.

You might be interested in finer measurements

- How long did this single function take to execute?
- Did this loop run faster after I parallelized it?

To answer such questions, there is a special OpenMP function called **omp_get_wtime()**

```
seconds = omp_get_wtime ( );
operations to time;
seconds = omp_get_wtime ( ) - seconds;
```

OpenMP + MPI?

We have noted that an OpenMP program must run on a single node, since each thread needs access to a common memory. That puts a hardware limit on how much parallelism you can ask for.

An MPI program can run on an arbitrary number of nodes: hundreds or thousands, if available.

However, an MPI program is really a collection of cooperating programs. It is perfectly posible for these individual programs to each be running on a separate node, using all the cores, under OpenMP.

In other words, a careful programmer can achieve good parallel performance by combining the simplicity and local speed of OpenMP with the global power of MPI.

CONCLUSION: References

- Parallel Programming in OpenMP, Chandra, et al;
- Using OpenMP; Chapman, et al;
- Parallel Programming in C with MPI and OpenMP, Quinn;
- OpenMP Tutorial, Blaise Barney, **https://computing.llnl.gov/tutorials/openMP/**