Parallel Programming with OpenMP

John Burkardt
Information Technology Department
Virginia Tech

.....

https://people.sc.fsu.edu/~jburkardt/presentations/...
openmp_2010_vt.pdf

.....

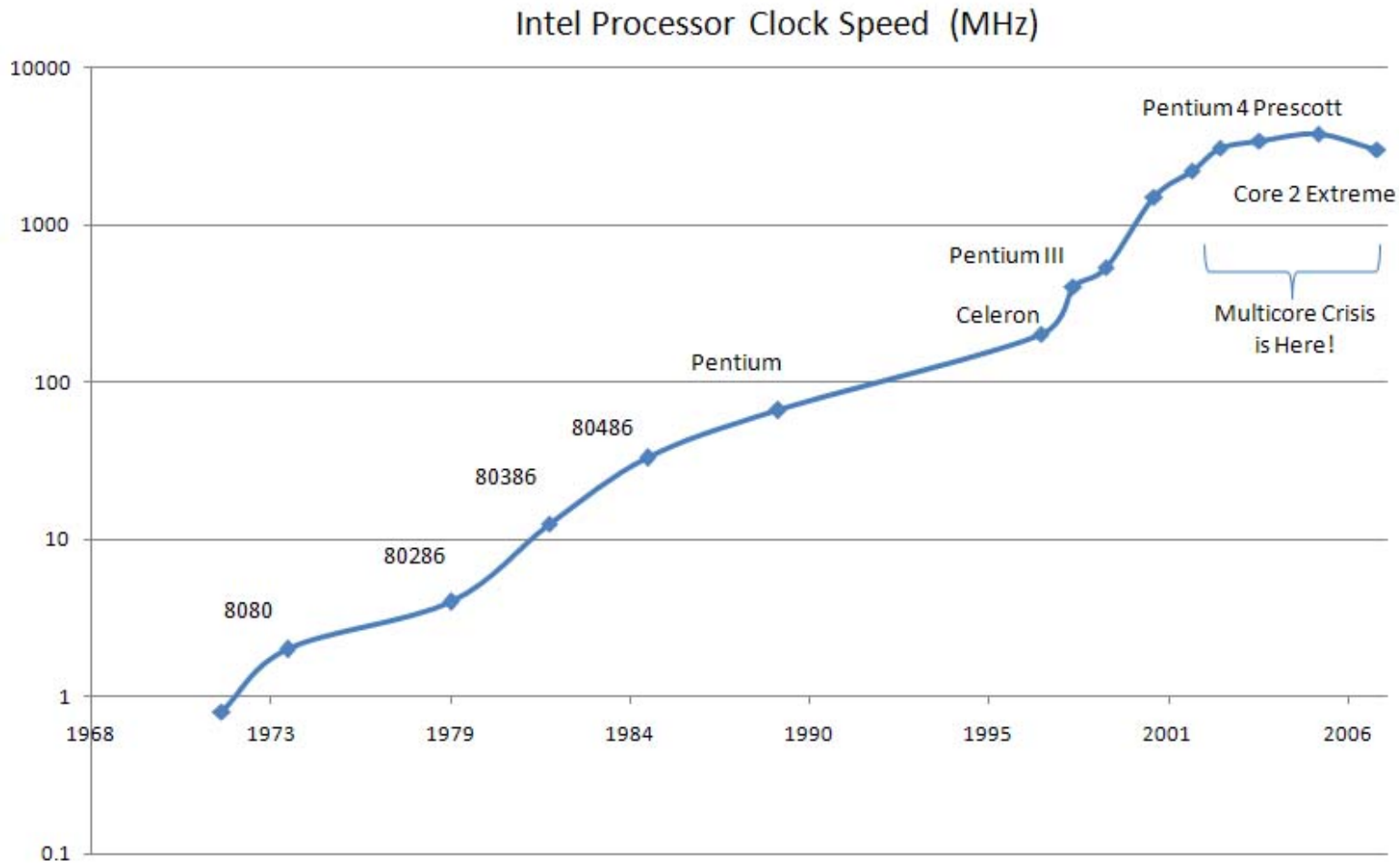
Virginia Tech Parallel Programming Bootcamp
1010 Torgersen Hall

10 August 2010

Parallel Programming With OpenMP

INTRO: Clock Speed Ceiling

CPU speeds have hit physical limits (about 4 GigaHertz).



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 takes advantage of certain facts:

- Many program steps can be carried out simultaneously;
- Multiple processors are available to execute one program;
- Processors can cooperate under one program and one memory;
- Processors can communicate, running separate programs and memory.

Old languages have been updated, and new languages invented, so that the programmer can use these new ideas.

INTRO: Extending Languages

In some cases, parallel programming appears as a small modification to an existing language. In other cases, existing languages appear as a small part of an extensive new parallel programming framework.

- **OpenMP** is a gentle modification to C and FORTRAN; a single program 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;
- **CUDA**, **OpenCL** and **DirectCompute** are programming frameworks that include a programming language (usually C) but also interfaces that talk directly with the underlying hardware, usually a GPU.

INTRO: OpenMP

OpenMP runs a user program in parallel.

Parallelism comes from multiple cooperating **threads** of execution.

These threads cooperate on **parallel sections** of a user program.

This happens on a **shared memory** system, where every thread can see and change any data item.

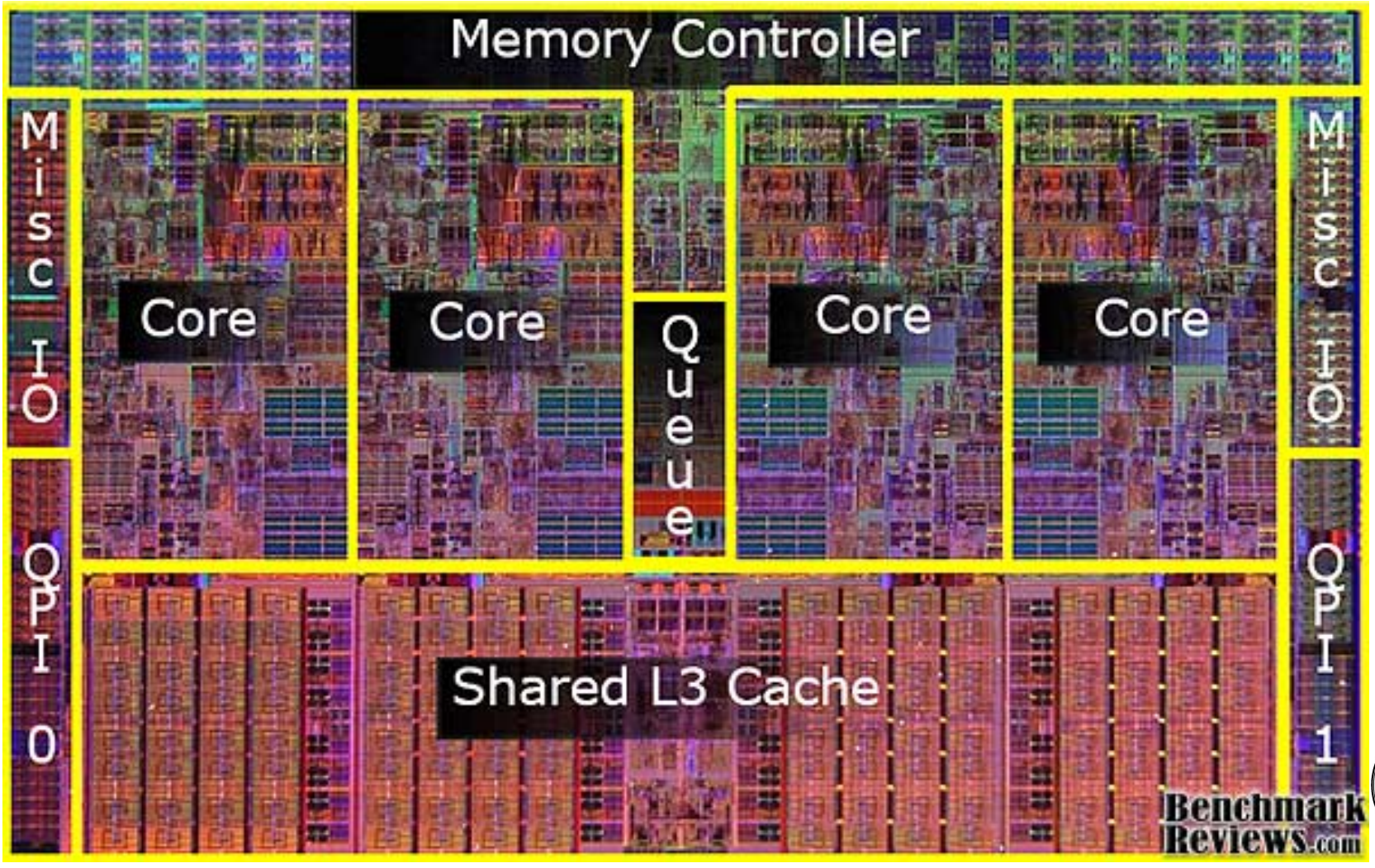
INTRO: A Shared Memory System

A shared memory system might be:

- one core, one memory (older PC's, sequential execution)
- multicore, one memory (your laptop; VT Ithaca system)
- multicore, multiple memory **NUMA** system (VT SGI system)

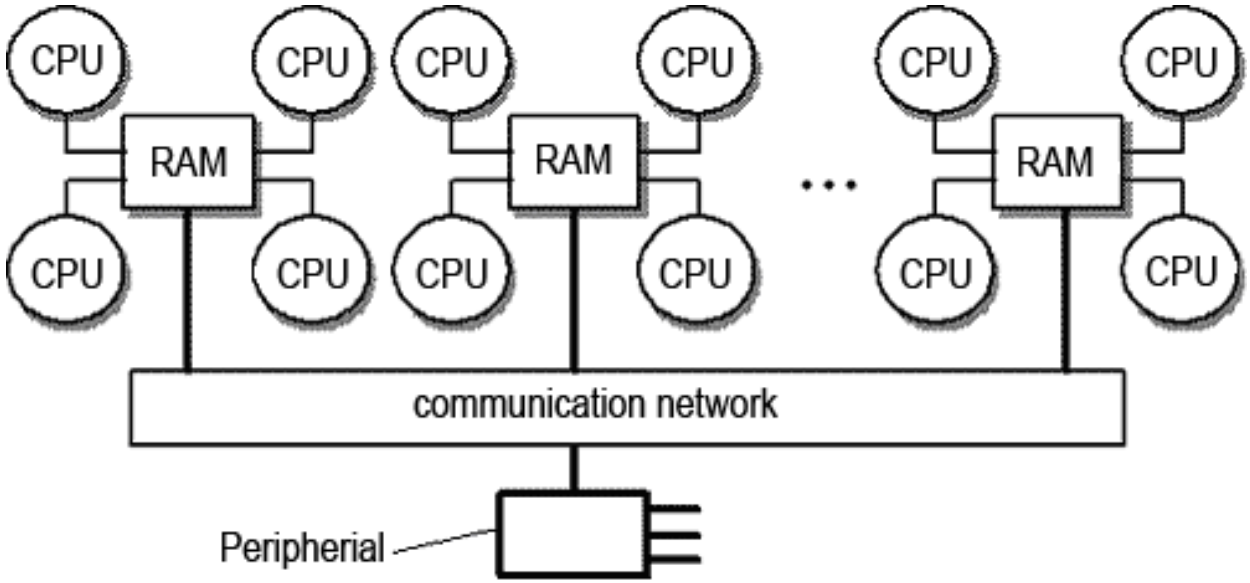
INTRO: Multicore Shared Memory

On VT's Ithaca system, OpenMP can run a single program on a pair of Nehalem quadcore processors sharing one memory.



INTRO: NUMA Shared Memory

VT's SGI ALTIX systems use the NUMA model, with as many as 128 dual-core processors; OpenMP programs can run here as well.



On a NUMA system, a very fast communication network and special memory addressing allows multiple memories to be shared (although "far" memory can be slower to access.)

Parallel Programming With OpenMP

PARALLEL: Programming Without Parallelism

If you've programmed in a "traditional" style, you probably have a mental image of how a program works.

At one level, you regard the program as a list of calculations to be carried out. Variables are little boxes with names on the outside, and changeable values inside. The processor executes the program by reading the next statement, which usually requires getting numbers out of some boxes, performing a calculation, and puting the result into another box.

The program may have loops, and conditional statements, and calls to functions, but as the processor executes the program, there is never any doubt about what statement is being executed, and what the next statement is.

We call this a **sequential** or **serial** or **non-parallel** model of computation.

PARALLEL: Adding Parallelism

A typical program will contain loops, that is, sequences of operations to be repeated many times. Sometimes each iteration of the loop is an independent computation. This is a common example where OpenMP can be used.

OpenMP provides a language for the user to mark such loops and other sections of code as parallelizable. It responds to such remarks by activating multiple cooperating cores to share the work of those calculations.

Before we look at the OpenMP language, let's consider some simple calculations that might be parallelizable.

PARALLEL: A Parallelizable Loop

Suppose a teacher has 150 students in a class, has given 16 quizzes, and that every student has (amazingly) taken every quiz. An averaging program might look like this:

```
for ( s = 0; s < 150; s++ ) {
  av[s] = 0.0;
  for ( q = 0; q < 16; q++ ) {
    av[s] = av[s] + g[s][q];
  }
  av[s] = av[s] / 16.0;
}
```

```
do s = 1, 150
  av(s) = 0.0;
  do q = 1, 16
    av(s) = av(s) + g(s,q)
  end do
  av(s) = av(s) / 16.0
end do
```

Each student's average can be computed in parallel.

PARALLEL: Almost Parallelizable Loop

Suppose we want the maximum score on test 5:

```
mx = 0.0;
for ( s = 0; s < 150; s++ ) {
  if ( mx < g[s][4] ) {
    mx = g[s][4];
  }
}
```

```
mx = 0.0;
do s = 1, 150

  mx = max ( mx, g(s,5) )

end do
```

Notice that we are computing a single value **mx**.
Can such a computation be done in parallel?

PARALLEL: A Non-Parallelizable Loop

Consider the following procedure to compute the base 2 digits of an integer **n**.

- If **n** is even, write a 0, otherwise 1;
- Divide **n** by 2;
- If **n** is zero, stop;
- Otherwise repeat the loop to compute the next digit.

PARALLEL: A Non-Parallelizable Loop

```
n = some value;
i = - 1;
do {
  i = i + 1;
  d[i] = n % 2;
  n = n / 2;
} while ( 0 < n )
```

```
n = some value
i = -1
do while ( 0 < n )
  i = i + 1
  d(i) = mod ( n, 2 )
  n = n / 2
end do
```

Parallelization tests:

- Do we know there will be an iteration 17?
- If so, could we compute iteration 17 immediately?

PARALLEL: Dependent Variables

There are two parallelization problems with this loop.

First, it's a **while** loop rather than a **for** or **do** loop, which means that when we begin the loop, we can't divide up the work because we don't know how much there is. We can't start iteration 17 because there may not be one! (Test 1 fails.)

Second, and more seriously, iteration **i** of the loop needs to "read" the value of **n** in order to do its job. But the iteration **i-1** of the loop just changed the value of **n**. So the only iteration that can be begin immediately is iteration 0. Only when it is complete can iteration 1 begin. And iteration 17 can't begin until 16, 15, 14, ... have completed. (Test 2 fails.)

PARALLEL: A Simple Loop to Parallelize

Suppose you had to calculate the norm of a vector by hand. If there were 100 entries to square and sum, that's a lot of work. If a friend offered to help, you could certainly figure out a way to cooperate on the task, and finish in about half the time.

```
norm = 0.0

do i = 1, n
  xsq = x(i) * x(i)
  norm = norm + xsq
end do

norm = sqrt ( norm )
```

OpenMP can speed up exactly these kinds of calculations.

PARALLEL: How OpenMP Sees Your Loop

We could imagine two cooperating processors executing the program something like this:

```
                    norm = 0.0

IF ( id == 0 ) THEN      ELSE   IF ( id == 1 ) THEN
  do i = 1, n/2            |      do i = n/2 + 1, n
    xsq = x(i) * x(i)      |        xsq = x(i) * x(i)
    norm = norm + xsq      |        norm = norm + xsq
  end do                   |      end do
                         END IF

                  norm = sqrt ( norm )
```

But we must handle the variables in these loops in a special way, in order to get the right answer!

PARALLEL: New Variables

We might need a couple new variables:

- Assume that each processor has an ID, so it knows what part of the code to execute.
- Assume each processor knows how many processors there are, so that we know whether to break the loop into halves, thirds,...

PARALLEL: Shared Variables

Some variables won't cause any problems. They can be shared. For instance, both processors will want to know the value of **N**. Both processors will want to know values of the array **X**.

Since both processors only "read" the values of these variables, we don't anything to worry about. As we will see, it is when both processors try to "write" to a variable that the problems arise!

PARALLEL: Private Variables

But consider the simple variable **I**, used as a loop counter. Processor 0 will want **I** to have values like 1, 2, 3, ... up to 50, while processor 1 will expect I to contain values like 51, 52, ..., 100. They cannot share the variable **I**. What if we somehow make two copies of this variable, so that each processor can have its own private copy?

The variable **XSQ** is just like **I**; that is, each processor will be trying to put different information into it, so we must give them private copies of **XSQ** as well.

Note that both private variables **I** and **XSQ** are simply conveniences used during the loop. After the loop is completed, we don't expect them to contain any interesting information.

PARALLEL: Reduction Variables

This is not true for our remaining variable, **NORM**. This variable seems to stand on the borderline between a shared and private variable. It has a value before the loop, and we want to know its value after the loop. But both processors will try to modify it during the loop.

A variable like this is called a reduction variable. Essentially, we make give each processor a private copy during the loop, and when the loop is over, we combine these private results into a final shared result.

That's actually how you and your friend would share the work of the norm calculation if you had to do it by hand!

PARALLEL: Summary

We have had an informal introduction to some of the ideas that OpenMP uses in order to make it possible to execute a program in parallel.

Now we will look at some actual programs, and start to learn the rules for what can be parallelized, how you classify variables in a loop, and what sorts of changes you make to your program so that it can use OpenMP.

HELLO: How Things Should Work

Now let's assume we have a program including some portion that can be parallelized, and look at how OpenMP can be involved.

We'll assume we have a quadcore computer; we will want to harness all 4 cores to cooperate in executing our program. Each core will be in charge of a "thread" of execution.

(We will use the words *core*, *thread*, *worker*, and *processor* as though they meant the same thing: an entity capable of executing a sequential program.)

There will be one "master thread" which is always running. It executes the sequential portions of the program. When it comes to a parallel region, it calls the extra threads for help.

HELLO: Shared Memory

In order for the processors to cooperate, each needs a numeric ID; they will also need some "private" memory space that no other processor can get to. That's where it can keep its ID, the value of the loop index it is executing, and possibly a few other temporary variables (examples coming!).

However, most program data sits in one place, accessible for reading or writing by all processors. This feature is so important that OpenMP is denoted as shared-memory programming.

The cores in a desktop machine can always cooperate in this way; some computer clusters can also use shared memory. When a cluster cannot use shared memory it is called a distributed memory cluster, and parallel programming is done with MPI instead.

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 loops or other code that is 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. execute the program.

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

HELLO: Let's Say Hello!

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

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

```
#pragma omp parallel          !$omp parallel
{
   parallel commands            parallel commands
}                               !$omp end parallel
```

Work in the parallel region is shared by the available *threads*.

The default behavior is that each thread carries out all the commands. (This is not what we will want to do in general!)

HELLO: HELLO in C++ and F90

```
void main ( ){
cout << "Sequential hi!\n";
#pragma omp parallel
{
cout << "Parallel hi!\n";
}
return;
}
```

```
program main
print *, 'Sequential hello!'
!$omp parallel

print *, 'Parallel hello!'
!$omp end parallel
stop
end
```

HELLO: HELLO in C and F77

```
void main ( ){
printf ( "Sequential hi!\n" );
#pragma omp parallel
{
printf ( "Parallel hi!\n" );
}
return;
}
```

```
      program main
      print*,'Sequential hi!'
c$omp parallel

      print*,'Parallel hi!'
c$omp end parallel
      stop
      end
```

HELLO: Comments

The parallel directive marks parallel regions in the code.

- In C and C++, the directive is used just before the block of statements. Braces {} may be used to create a block.
- In Fortran, the beginning **and** end of the parallel region must be marked;
- The parallel directive can include further information about shared and private variables;
- Other directives may be inserted into the parallel region to indicate loops that can be divided up.

The form of the directive makes it look like a comment.

So if we compile the program in the usual way, it runs sequentially.

HELLO: Making It Run

To compile the program with OpenMP and run it:

1. **Compile with C or C++**:
   - gcc -fopenmp hello.c
   - g++ -fopenmp hello.C
   - icc -openmp -parallel hello.c
   - icpc -openmp -parallel hello.C
2. **or compile with FORTRAN**:
   - gfortran -fopenmp hello.f90
   - ifort -openmp -parallel -fpp hello.f90
3. **Set the number of OpenMP threads**:
   - export OMP_NUM_THREADS=4 *(Bourne, Korn, Bash)*
   - setenv OMP_NUM_THREADS 4 *(C or T shell)*
4. **Run the program**:
   - ./a.out

HELLO: HELLO Output

The print in the sequential region is done by the master thread; The print in the parallel region is done by every thread.

```
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!
```

Of course usually we will want to use OpenMP to divide work up, rather than doing the same thing several times! We will learn how to do that in a few minutes.

HELLO: The Number of Threads

We got 4 printouts because we had 4 threads.

We asked for them by setting the environment variable **OMP_NUM_THREADS** to 4 before the program ran.

If your program has access to 8 cores (hardware), it is always legal to ask for anywhere from 1 to 8 threads (software).

In some cases, you can ask for more than 8 threads; then some cores "double up" and keep track of more than one thread. Usually, this reduces performance.

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:

```
# include "omp.h"      C / C++
include "omp_lib.h"    FORTRAN77
use omp_lib            FORTRAN90
```

Let's redo HELLO using these functions.

HELLO: HELLO Again!

```
wtime = omp_get_wtime ( );
cout<<"Available processors:"<<omp_get_num_procs()<<"\n";
cout<<"Available threads    "<<omp_get_max_threads()<<"\n";
cout<<"Threads in use       "<<omp_get_num_threads()<<"\n";
#pragma omp parallel private ( id  )
{
  id = omp_get_thread_num ( );
  cout << "  Hello from process " << id << "\n";
  if ( id == 0 )
  {
    cout<<"Threads in use"<<omp_get_num_threads ( ) << "\n";
  }
}
wtime = omp_get_wtime ( ) - wtime;
cout << "  Wtime = " << wtime << "\n";
```

HELLO: HELLO Again!

```
wtime = omp_get_wtime ( )
print*,'  Available processors: ', omp_get_num_procs ( )
print*,'  Available threads     ', omp_get_max_threads ( )
print*,'  Threads in use        ', omp_get_num_threads ( )
!$omp parallel private ( id )
  id = omp_get_thread_num ( )
  print *, '  Hello from process ', id
  if ( id == 0 ) then
    print*,'  Threads in use ', omp_get_num_threads ( )
  end if
!$omp end parallel
wtime = omp_get_wtime ( ) - wtime
print *, '  Wtime = ', wtime
```

HELLO: Making It Run

Now compile the hello program, but let's also change the number of threads from 4 to 2:

- export OMP_NUM_THREADS=2 *(Bourne, Korn, Bash)*
- setenv OMP_NUM_THREADS 2 *(C or T shell)*

HELLO: HELLO Again Output

```
Available processors:    4  <-  OpenMP knows we have 4
Available threads        2  <-- We asked for 2 threads
Threads in use           1  <-- In sequential region

  Hello from process     0
  Hello from process     1
  Threads in use         2   <-- In parallel region

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:

```
#pragma omp parallel private ( id )
!$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 needs its own private copy of the variable.

SAXPY: Work That Can Be Divided

If you simply put statements inside a parallel region, (as we saw in the Hello example), then all the threads execute them.

Rather than **repeating** the work, we would like to look at how we can divide work among the threads. Then all the work gets done, and each thread's work is just a portion of the total, and we're done faster.

The kind of work we can divide up comes in three varieties:

- **loops**;
- **sections**, an explicit list of tasks;
- **workshare**, (Fortran only).

Each loop, set of sections, or workshare is considered a **block**.

SAXPY: Blocks of Tasks

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

```
!$omp parallel
  PARALLEL BLOCK 1 (a loop on I)
  ...a few statements executed by all threads...
  PARALLEL BLOCK 2 (explicit list of tasks)
  PARALLEL BLOCK 3 (a nested loop on I and J)
!$omp end parallel
```

By default, no thread moves to the next block until all are done the current one.

This waiting can be cancelled by the **(nowait)** clause on a block.

SAXPY: The FOR or DO Directive

```
#pragma omp parallel
{
#pragma omp for
  for ( i = 0; i < n; i++ ) {
      ...for loop body...
  }
..more blocks could follow...
}
```

```
!$omp parallel
  !$omp do
    ...do loop body...
  !$omp end do
    ..more blocks could follow...
!$omp end parallel
```

SAXPY: The FOR or DO Directive

A parallel region of just one loop can use a short directive:

```
#pragma omp parallel for
for ( i = 0; i < n; i++ ) {
      ...for loop body...
}
```

- -

```
!$omp parallel do
do i = 1, n
  ...do loop body...
end do
!$omp end parallel do
```

(But I think it's better to use separate statements.)

SAXPY: Nested Loops

If loops are nested, you only parallelize one index!

```
#pragma omp parallel
{
  #pragma omp for (nowait)   ¡– an option
    for ( i = 0; i < m; i++ ) {
      for ( j = 0; j < n; j++ ) {
        This nested loop is parallel in I
      }
    }

    for ( i = 0; i < m; i++ ) {
    #pragma omp for
      for ( j = 0; j < n; j++ ) {
        This nested loop is parallel in J
      }
    }
}   <-- End of parallel region
```

SAXPY: Nested Loops

```
!$omp parallel
  !$omp do
    do i = 1, m
      do j = 1, n
        This nested loop is parallel in I
      end do
    end do
  !$omp end do (nowait)   <-- nowait goes here in Fortran!
    do i = 1, m
    !$omp do
      do j = 1, n
        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; each section is marked by a section directive and will be executed by **exactly one thread**.

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

If there are more threads than sections, the extras will be idle.

SAXPY: SECTION Syntax

```
!$omp parallel                    <-- inside "parallel"
  ...                             <-- earlier blocks
  !$omp sections<-- begin sections block
    ...                           <-- all threads will do
    !$omp section
      code for section 1  <-- only 1 thread will do
    !$omp section
      code for section 2  <-- only 1 thread will do
                                  <-- more sections could follow
  !$omp end sections              <-- end sections block; optiona
  ...                             <-- later blocks
!$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.

```
#pragma omp parallel
{
   cout << "Hey!\n";
  #pragma omp sections  <-- optional (nowait)
  {
    cout << "Hello!\n";
    #pragma omp section
      s = sin_table ( n );
    #pragma omp section
      cout << "Hi!\n";
      c = cos_table ( n );
  }
}
```

How many times will "Hey!", "Hello" and "Hi" appear?

SAXPY: The WORKSHARE Directive

A third kind of task that can be included in a parallel region involves any of three kinds of 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.

Unfortunately, the **workshare** directive does not actually seem to have been implemented, at least not in the Gnu and Intel compilers that I have seen.

In any case, here's how it is supposed to work!

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: Add a multiple of one list to another

The SAXPY program will demonstrate how we can combine the **parallel** directive, which makes multiple workers available, and the **for** or **do** directive, which tells OpenMP that the following work should be divided among the available workers.

It is this kind of procedure that we expect will speed up our execution!

SAXPY: Program with Parallel Loop

```
int main ( ) {
  int i, n = 1000;
  double x[1000], y[1000], s = 123.456;
  x = random_vector ( n );
  y = random_vector ( n );
#pragma omp parallel private ( i )
#pragma omp for
  for ( i = 0; i < n; i++ )
  {
      y[i] = y[i] + s * x[i];
  }
  return 0;
}
```

SAXPY: Program with Parallel Loop

```
program main
  integer i
  integer, parameter :: n = 1000
  double precision :: s = 123.456
  double precision x(n), y(n)
  call random_vector ( n, x )
  call random_vector ( n, y )
!$omp parallel private ( i )
  !$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

This program shows how threads can cooperate on a calculation.

Imagine that 4 threads would each do 250 iterations, for instance.

Is it clear these calculations are independent? What is the test?

One variable gives us a little trouble, though. In a shared memory system, there's only one copy of each variable. This means the variable **I** could have only one value. But each thread needs a separate copy of **I**, to keep track of its iterations.

The **parallel** directive can include a private() clause, which provides a lists of those variables which must be stored privately by each thread for the duration of this parallel region.

When a parallel region begins, the private variables do not "remember" values they may already have had, and when the region ends, their values are lost.

PI: Data Conflicts

When multiple workers are handling data, there are many possibilities for *data conflicts*; basically, this means the orderly reading and writing of data that occurs in a sequential calculation has broken down when we parallelized it.

When the value of a variable cannot be shared in the default way, there are OpenMP directives that can sometimes fix the problem.

The issues we will concentrate on here involve *temporary variables* and *reduction variables* and they will both be illustrated by a simple program that estimates an integral (whose correct value is $\frac{\pi}{4}$).

PI: Sequential Version

```
n = 100000;
h = 1.0 / ( double ) ( n );
q = 0.0;
x = - h;

for ( i = 0; i < n + 1; i++ )
{
  x = x + h;
  q = q + 1.0 / ( 1.0 + x * x );
}

q = q / ( double ) ( n + 1 );
```

PI: Sequential Version

```
n = 100000
h = 1.0 / dble ( n )
q = 0.0
x = - h

do i = 1, n + 1
  x = x + h
  q = q + 1.0 / ( 1.0 + x * x )
end do

q = q / dble ( n + 1 )
```

PI: The Problem of Variable X

The first thing to realize about this loop is that it looks like it cannot be parallelized. We cannot begin the 17th iteration, for instance, without knowing the values of **X** and **Q**, which were modified by every one of the previous iterations.

Let's try to address the problem of **X** first.

X is really a convenience variable, or temporary variable. We don't need its value before or after the loop. The first loop iteration sets **X** to 0, the second to **H**, the third to 2*__H__, and the 17th to 16*__H__. So we could calculate **X** immediately by:

```
x = i * h;          ...or...          x = ( i - 1 ) * h
```

Then, if we make **X** a private variable, **X** is no longer a problem.

PI: The Problem of Variable Q!

Q is also a problem.

If **Q** is shared, we will have problems. Each worker reads, adds, and writes **Q** in an orderly way. But at run time, the operations of multiple workers are interleaved in an unpredictable way.

Suppose **Q** starts at 0, and consider the several ways in which the six worker operations can be interleaved, although each worker does its steps in order. What are possible values for **Q** at the end?

```
Worker 1 reads Q.
Worker 1 adds 10.
Worker 1 updates Q.
```

```
Worker 2 reads Q.
Worker 2 adds 20.
Worker 2 updates Q.
```

PI: The Problem of Variable Q!

So could we make **Q** a private variable?

Private variables are temporary; their values are not saved once we leave the loop. But the point of this loop is to compute **Q** and return its value, so making **Q** private is the wrong approach.

Q is a sort of intermediate variable, not exactly private or shared; it's called a reduction variable. An orderly way to compute its value allows each worker to compute a private version of **Q** that we never need to know about. At the end, these are joined into the shared variable **Q**.

By declaring **Q** to be a **reduction** variable, OpenMP has enough information to create the temporary hidden private copies, and to combine them at the end.

PI: The Reduction clause

Any variable which contains the result of a reduction operator must be identified in a **reduction** clause of the **parallel** directive.

Each worker operates on a private copy of the variable, and these results are combined into a shared variable at the end.

Reduction clause examples must also include the type of operation:

- **reduction (+ : xdoty)** ; xdoty is a sum;
- **reduction (+ : sum1, sum2, sum3)** , several sums;
- **reduction (* : factorial)**, a product;
- **reduction (&& : b)** , logical AND in C/C++;
- **reduction (.and. : b)** , logical AND in FORTRAN;
- **reduction (max : p)** , max / min (FORTRAN only);

PI: Directive Clauses

```
  n = 100000;
  h = 1.0 / ( double ) ( n );
  q = 0.0;
#pragma omp parallel shared ( h, n ) private ( i, x ) \
reduction ( + : q )   <-- note how we continued the directive!

#pragma omp for
  for ( i = 0; i < n + 1; i++ )
  {
    x = i * h;
    q = q + 1.0 / ( 1.0 + x * x );
  }
  q = q / ( double ) ( n );
```

PI: Directive Clauses

```
  n = 100000
  h = 1.0 / dble ( n )
  q = 0.0
!$omp parallel shared ( h, n ) private ( i, x ) &
!$omp reduction ( + : q ) <-- continued directive

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

PI: Reduction Operations

Each variable in a parallel region is assumed to have a type.

The available types are:

- shared, most variables;
- private, for loop indices and temporary variables;
- reduction, for sums, products and so on.

Unless a user declares the type of a variable in the **parallel** directive, it is given the default type, which is usually **shared**.

In FORTRAN, the **do** loop index is private by default.

The MD Example: A Real Computation

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

The user chooses the number of particles and time steps.

The particles get random positions and velocities for time step 0.

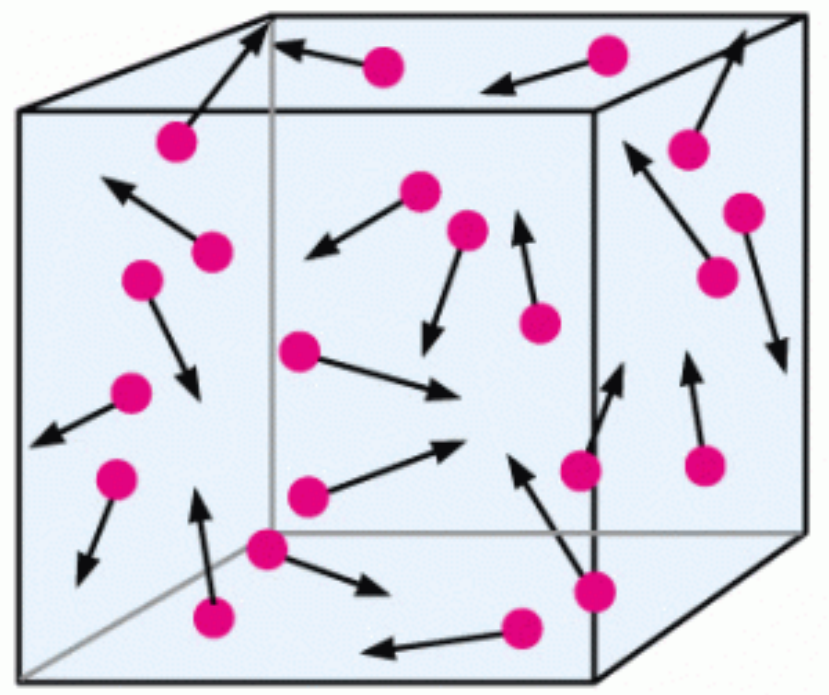
To compute data for the next time step, we compute the force on each particle from all the other particles.

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.

MD: A Molecular Dynamics Simulation

Compute positions and velocities of **N** particles over time;
The particles exert a weak attractive force on each other.



The MD Example

```
for ( i = 0; i < n; i++ ) {
  for ( j = 0; j < n; j++ ) {
    d = 0.0;
    for ( k = 0; k < 3; k++ ) {
      dif[k] = coord[k][i] - coord[k][j];
      d = d + dif[k] * dif[k];
    }
    for ( k = 0; k < 3; k++ ) {
      f[k][i] = f[k][i] - dif[k] * pfun ( d ) / d;
    }
  }
}
```

The MD Example

```
do i = 1, n
  do j = 1, n
    d = 0.0
    do k = 1, 3
      dif(k) = coord(k,i) - coord(k,j)
      d = d + dif(k) * dif(k)
    end do
    do k = 1, 3
      f(k,i) = f(k,i) - dif(k) * pfun ( d ) / d
    end do
  end do
end do
```

The MD Example: Private/Shared/Reduction?

This example comes from a molecular dynamics (MD) program.

The variable **n** is counting particles, and where you see a 3, that's because we're in 3-dimensional space.

The array **coord** contains spatial coordinates; the force array **f** has been initialized to 0.

The mysterious **pfun** is a function that evaluates a factor that will modify the force.

Which variables in this computation should be declared **shared** or **private** or **reduction**?

Which variables are shared or private **by default**?

The MD Example: QUIZ

```
for ( i = 0; i < n; i++ ) {    <-- I?  N?
  for ( j = 0; j < n; j++ ) {    <-- J?
    d = 0.0;                          <-- D?
    for ( k = 0; k < 3; k++ ) {      <-- K?
      dif[k] = coord[k][i] - coord[k][j];  <-- DIF?
      d = d + dif[k] * dif[k];                COORD?
    }
    for ( k = 0; k < 3; k++ ) {
      f[k][i] = f[k][i] - dif[k] * pfun ( d ) / d;
    }      <-- F?  PFUN?
  }
}
```

The MD Example: QUIZ

```
do i = 1, n                    <-- I?   N?
  do j = 1, n                  <-- J?
    d = 0.0                    <-- D?
    do k = 1, 3                <-- K
      dif(k) = coord(k,i) - coord(k,j)  <-- DIF?
      d = d + dif(k) * dif(k)                COORD?
    end do
    do k = 1, 3
      f(k,i) = f(k,i) - dif(k) * pfun ( d ) / d
    end do                     <-- F?, PFUN?
  end do
end do
```

The MD Example: Some Puzzling Cases

The variable **D** might look like a reduction variable.

But that would only be the case if the parallel loop index was **K**; only in that case would the computation of the value **D** be carried out by multiple workers.

It's true that it is computed as a reduction operation, but it is **not a parallel reduction**.

DIF is our first example of a private variable that is an array. The clue that **DIF** must be private is that it is computed for convenience, its value changes from one iteration to the next, and we don't have any need for its value afterwards.

The MD Example: OpenMP Version

```
#pragma omp parallel shared ( coord, f, n ) \
  private ( d, dif, i, j, k )
#pragma omp for
  for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ ) {
      d = 0.0;
      for ( k = 0; k < 3; k++ ) {
        dif[k] = coord[k][i] - coord[k][j];
        d = d + dif[k] * dif[k];
      }
      for ( k = 0; k < 3; k++ ) {
        f[k][i] = f[k][i] - dif[k] * pfun ( d ) / d;
      }
    }
  }
```

The MD Example: OpenMP Version

```
!$omp parallel shared ( coord, f, n ) &
!$omp private ( d, dif, i, j, k )
!$omp do
  do i = 1, n
    do j = 1, n
      d = 0.0
      do k = 1, 3
        dif(k) = coord(k,i) - coord(k,j)
        d = d + dif(k) * dif(k)
      end do
      do k = 1, 3
        f(k,i) = f(k,i) - dif(k) * pfun ( d ) / d
      end do
    end do
  end do
!$omp end do
```

DIRECTIVES

In a short class like this, it is only possible to discuss some of the basic OpenMP directives. You should, however, know that there are many more directives available to help you manage certain common problems that arise in parallel programming.

We will mention some of the more useful ones, and then proceed to a moderately complicated program which illustrates how a number of these advanced directives can be useful.

DIRECTIVES: Scheduling

Scheduling is the way that the iterations of a parallel loop are assigned to the available threads.

The default schedule is **static**; iterations are divided into consecutive "chunks"; the first thread is assigned the first chunk, and so on.

On the **for** or **do** directive, you can include the clause schedule(static,10) to use static scheduling; now the iterations are dealt out round robin, in groups of 10, into chunks; the first thread then gets the first chunk, as before.

The schedule(dynamic,5) clause assigns chunks of 5 iterations to each thread initially, and holds back all the remaining iterations. As soon as a thread finishes, it is assigned another chunk of 5 more, until the work is done.

DIRECTIVES: Scheduling

You only need to worry about scheduling in situations where you know that some iterations of the loop will take much more time than others.

To give each thread the same number of iterations, but to shuffle them up better, the static schedule, with a smaller chunk size, is enough.

The dynamic option deals most effectively with unbalanced work, but at the price of more checks and communication.

As an example where scheduling might help, consider a calculation to count the number of primes.

DIRECTIVES: Scheduling

```
# pragma omp parallel \
  shared ( n ) \
  private ( i, j, prime ) \
  reduction ( + : total )

# pragma omp for schedule ( static, 20 )
  for ( i = 2; i <= n; i++ ) {
    prime = 1;
    for ( j = 2; j < i; j++ ) {
      if ( i % j == 0 ) {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
```

DIRECTIVES: Critical

A **critical** region is a portion of a parallel region that may be executed by all threads, but only one thread at a time.

```
  y_max = -100.0;
#pragma omp parallel for...
  for ( i = 0; i < n; i++ ){
    y = sin ( x[i] );
#pragma omp critical
    if ( y_max < y ) {
      y_max = y;
      i_max = i;
    }
  }
```

```
  y_max = - 100.0
!$omp parallel do ...
  do i = 1, n
    y = sin ( x(i) )
!$omp critical
    if ( y_max < y ) then
      y_max = y
      i_max = i
    end if
!$omp end critical
  end do
!$omp end parallel do
```

(If we only needed **y_max**, we could use a **max** reduction in FORTRAN.)

DIRECTIVES: Barrier

A **barrier** is a position in a parallel region at which every thread pauses execution, until all threads have reached that point.

```
#pragma omp parallel ... {
  id=omp_get_thread_num();
  if ( id == 0 ) {
    printf ( "Enter X:\n" );
    sscanf ( "%f", &x );
  }
  #pragma omp barrier
  printf("X+ID=%d\n",x+id);
}
```

```
!$omp parallel ...
  id=omp_get_thread_num()
  if ( id == 0 ) then
    print *, 'Enter X'
    read ( *, * ) x
  end if
!$omp barrier
  print *, 'X+ID=', x+id
!$omp end parallel
```

Some directives, such as the **for** or **do** directive, have an implicit barrier at the loop termination; this can be cancelled by using the nowait clause.

DIRECTIVES: the NOWAIT Clause

Some directives include an implicit barrier; the nowait clause cancels this. You may insert an explicit barrier where you need it:

```
#pragma omp parallel ...{
  #pragma omp for ( nowait )
  for ( i = 0; i < n; i++ )
    a[i] = sqrt ( x[i] );
  #pragma omp for ( nowait )
  for ( i = 0;i < n; i++ )
    b[i] = cos ( y[i] );
  #pragma barrier
  #pragma omp for
  for ( i = 0; i < n; i++ )
    c[i] = a[i] + b[n-1-i];
}
```

```
!$omp parallel ...
!$omp do
  a(i) = ...
!$omp end do ( nowait )
!$omp do
  b(i) = ...
!$omp end do ( nowait )
!$omp barrier
!omp do
  c(i) = ...
!$omp end do
!$omp end parallel
```

DIRECTIVES: Master and Single

The **master** directive: only the master thread (#0) executes this.

The **single** directive: only the first thread to get here executes this.

```
#pragma omp parallel ... {
  id=omp_get_thread_num();
  #pragma master {
    printf ( "Enter X:\n" );
    sscanf ( "%f", &x );
  }
  #pragma omp barrier
  printf("X+ID=%d\n",x+id);
}
```

```
!$omp parallel ...
  id=omp_get_thread_num()
!$omp master
    print *, 'Enter X'
    read ( *, * ) x
!$omp end master
!$omp barrier
  print *, 'X+ID=', x+id
!$omp end parallel
```

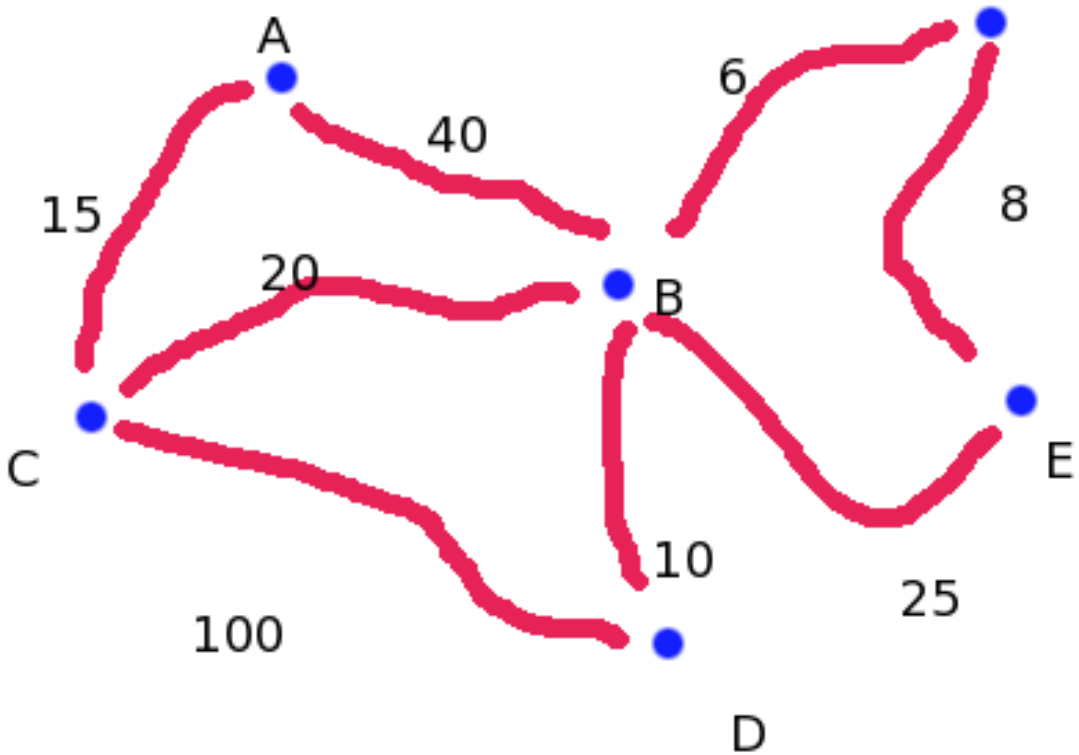
DISTANCE: A Classic Problem

Anyone doing highway traveling is familiar with the difficulty of determining the shortest route between points A and B. From a map, it's easy to see the distance between neighboring cities, but often the best route takes a lot of searching.

A **graph** is the abstract version of a network of cities. Some cities are connected, and we know the length of the roads between them. The cities are often called *nodes* or *vertices* and the roads are *links* or *edges*. Whereas cities are described by maps, we will describe our abstract graphs using a *one-hop distance matrix*, which is simply the length of the direct road between two cities, if it exists.

DISTANCE: An Intercity Map

Here is a simple map of 6 cities with the intercity highway distance.



DISTANCE: An Intercity One-Hop Distance Matrix

Supposing we live in city A, our question is, *"What is the shortest possible distance from A to each city on the map?"*

Instead of a map, we use a "one-hop distance" matrix **OHD[I][J]**:

	A	B	C	D	E	F
A	0	40	15	∞	∞	∞
B	40	0	20	10	25	6
C	15	20	0	100	∞	∞
D	∞	10	100	0	∞	∞
E	∞	25	∞	∞	0	8
F	∞	6	∞	∞	8	0

where ∞ means there's no direct route between the two cities.

DISTANCE: The Shortest Distance

The map makes it clear that it's possible to reach every city from city A; we just have to take trips that are longer than "one hop". In fact, in this crazy world, it might also be possible to reach a city faster by taking two hops rather than the direct route. (Look at how to get from city A to city B, for instance!)

We want to use the information in the map or the matrix to come up with a **distance** vector, that is, a record of the shortest possible distance from city A to all other cities.

A method for doing this is known as *Dijkstra's algorithm*.

DISTANCE: Dijkstra's algorithm

Use two arrays, **connected** and **distance**.
Initialize **connected** to false except for A.
Initialize **distance** to the one-hop distance from A to each city.
Do N-1 iterations, to connect one more city at a time:

1. Find I, the unconnected city with minimum **distance[I]**;
2. Connect I;
3. For each unconnected city J, see if the trip from A to I to J is shorter than the current **distance[J]**.

The check we make in step 3 is:
distance[J] = min (**distance[J]**, **distance[I]** + **ohd[I][J]**)

DISTANCE: A Sequential Code

connected[0] = true;

```
for ( i = 1; i < n; i++ ){
  connected[i] = false;
}
for ( i = 0; i < n; i++ ){
  distance[i] = ohd[0][i];
}
for ( step = 1; step < n; step++ )
{
  find_nearest ( distance, connected, &md, &mv );

  connected[mv] = true;

  update_distance ( mv, connected, ohd, distance );
}
```

DISTANCE: Parallelization Concerns

The main program includes a loop, but it is **not** parallelizable! Each iteration relies on the results of the previous one.

However, during each iteration, we also use loops to carry out the following operations, which are expensive and parallelizable:

- **find_nearest** searches for the nearest unconnected node;
- **update_distance** checks the distance of each unconnected node to see if it can be reduced.

These operations can be parallelized by using a parallel region. But we will need to be careful to synchronize and combine computations.

DISTANCE: Startup

When the parallel region begins, we can assign the cities S through E to thread T.

```
# pragma omp parallel private ( ... ){
  my_id = omp_get_thread_num ( );
  nth = omp_get_num_threads ( );
  my_s = (   my_id       * n ) / nth;
  my_e = ( ( my_id + 1 ) * n ) / nth - 1;

  # pragma omp single {
    cout << "  Using " << nth << " threads.\n";
  }
  for ( my_step = 1; my_step < n; my_step++ )
  {
  ---(we will fill this in next)---
  }
}
```

DISTANCE: FIND_NEAREST

Each thread T uses **find_nearest** to search its range of cities for the nearest unconnected one.

But now each thread will have returned such an answer. The answer we want is the node that corresponds to the smallest distance returned by the threads.

DISTANCE: FIND_NEAREST

```
# pragma omp single
{
  md = 1000000;
  mv = -1;
}
find_nearest ( my_s, my_e, distance, connected,
  &my_md, &my_mv );
# pragma omp critical
{
  if ( my_md < md )  {
    md = my_md;
    mv = my_mv;
  }
}
# pragma omp barrier
```

DISTANCE: UPDATE_DISTANCE

We have found the nearest unconnected city.

We need to connect it.

Knowing the minimum distance to this city, we check whether this decreases our estimated minimum distances to unconnected cities.

But we must make sure that we finish each step before moving on to the next.

DISTANCE: UPDATE_DISTANCE

```
# pragma omp single
{
  connected[mv] = true;
  cout << "Connecting node " << mv << "\n";
}

# pragma omp barrier

update_distance ( my_s, my_e, mv, connected, ohd,
  distance );

# pragma omp barrier
```

DISTANCE: Comments

This example illustrates how many of the new directives we just discussed can be used for a good purpose.

One reason this example seems more complicated is that we are not using parallel loops, but are organizing all the actions ourselves. A parallel loop will automatically pause at the end until all threads have caught up, before proceeding. In this example, we had to take care of that ourselves each time.

When we modified the sequential program, we had to introduce a few shared variables and many private ones. To help distinguish them, we tried to use the prefix **my** on the private variables, to suggest that copies of this variable "belong" to each thread.

WHERE: Desktop?

The most important thing you need for parallel programming with OpenMP is a machine with multiple cores.

(Of course you can run an OpenMP program on a one-core machine. And it's a good way to develop and check a program. But when you pay for a sports car, you look for a long straight road where you can drive fast!)

Manufacturers of PC's, Mac's and "Linux boxes" have been quietly moving up to dual-core and quad-core machines, and it's possible to get an 8-core desktop.

WHERE: Desktop?

Even if you plan to do your main work on a cluster, a multi-core desktop can be used for development. To do so, you need a compiler for your language, and it must be recent enough to support OpenMP.

The Intel family of compilers includes OpenMP support. They are available for Windows, Linux and Mac OSX. They include the Intel Math Kernel Library "MKL", a parallelized, highly optimized set of mathematical functions. The compilers are not free, but are heavily discounted to academic users.

The Gnu compilers are free, and come with OpenMP support. These compilers expect a Unix-style command line interface, so a PC user would have to install something like the Cygwin Unix emulator.

WHERE: System X?

A cluster machine seems like the logical place to go for a lot of cores. However, OpenMP also requires that the memory be shared, addressable as a single logical unit.

Virginia Tech's **System X** has 1100 nodes, each an Apple PowerMac G5, with two cores sharing a memory chip. But memory of separate nodes cannot be shared so an OpenMP program can only run on one node, and use at most 2 cores.

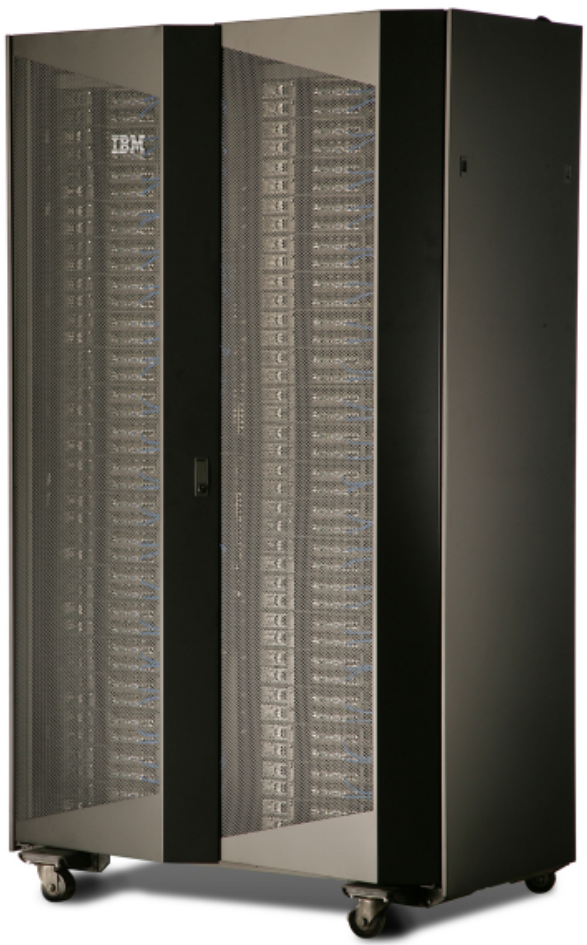
WHERE: SGI: Inferno/Inferno2/Cauldron?

Virginia Tech's SGI clusters have shareable memory.

- **Inferno** has 20 cores, is designed for interactive work, especially debugging, and users should ask for 2 to 4 cores.
- **Inferno2** has 128 cores, is accessible through a queuing system. No more than 16 cores per job.
- **Cauldron** has 64 cores, is accessible through a queueing system. A program can get all 64 cores.

WHERE: Ithaca?

Virginia Tech's IBM iDataplex **Ithaca** has 84 nodes, each of which has 8 cores (= 672 cores). Unfortunately, only the memory on a single node can be shared, so a maximum of 8 cores for OpenMP programs.



WHERE: Ithaca Software

- **abaqus**, version 6.9-2;
- **ansys**, finite element analysis;
- **ASreml**, maximum likelihood fit of mixed models;
- **atlas**, "automatically tuned linear algebra software";
- **bwa**, Burrows Wheeler sequence alignment;
- **CFX**, computational fluid dynamics;
- **fftw**, version 3.2.2, fast Fourier Transforms;
- **fluent**, version 12.0.16, fluid dynamics;
- **gaussian**, version 09, computational chemistry;
- **lapack**, version 3.2.1, linear algebra;
- **mathematica**, (no parallelism);
- **matlab**, version R2010a, with Parallel Computing Toolbox;
- **R**, version 2.11.0, the statistical package;
- **scalapack**, linear algebra for parallel processing;
- **star-ccm**, version 5.02, fluid dynamics.

Where: Getting an Account

Accounts on the Virginia Tech clusters are provided at no charge. Go to the website **http://www.arc.vt.edu/index.php**.

Under the item **Services & Support** select **User Accounts**. On the new page, you will see headings for

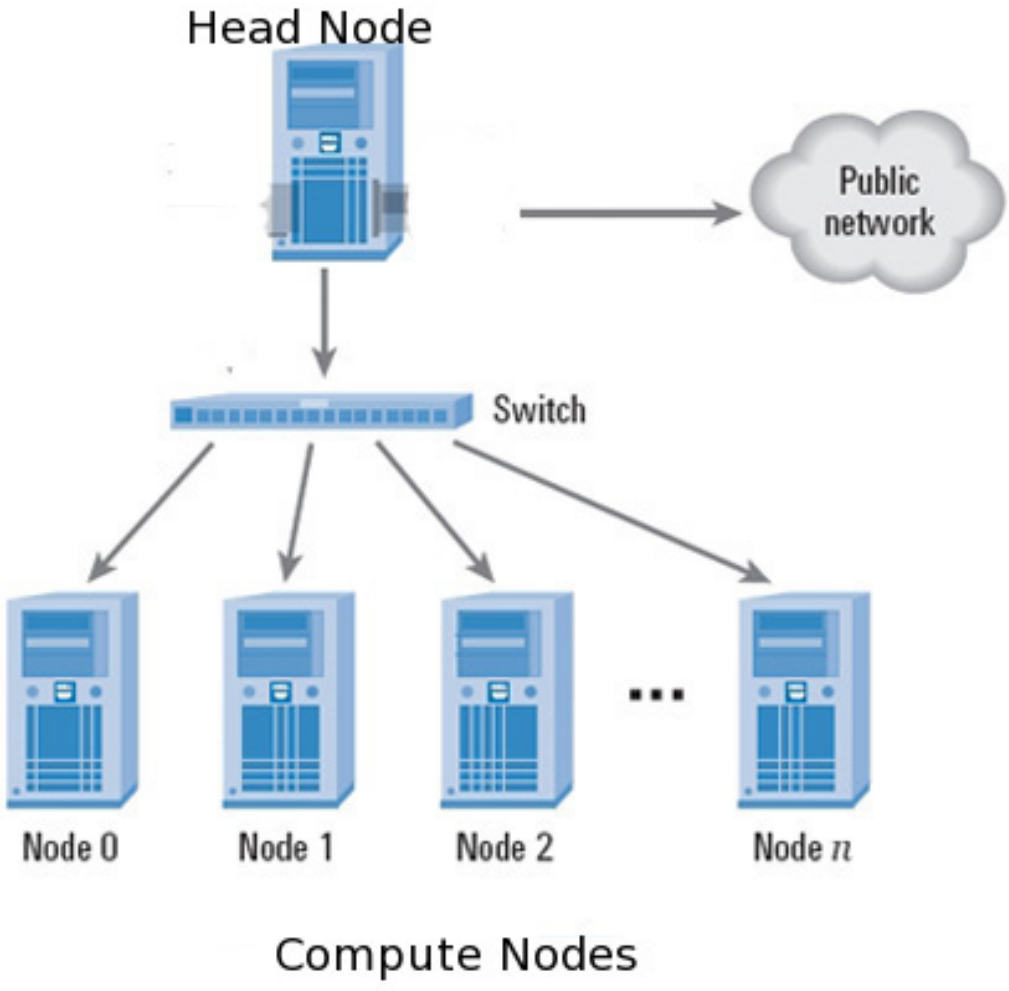
- **System X Allocation Requests**
- **SGI Account Requests**
- **Ithaca Account Requests**

You'll need to fill out the **ARC Systems Account Request Form**.

If you're interested in an account simply to experiment with OpenMP, say so. You do not have to make a lengthy description of your research until you have found out whether OpenMP and the cluster you've chosen will work for you.

BATCH: Typical Cluster

A cluster computer typically is divided into a few "head nodes", and lots of "compute nodes".



BATCH: Typical Cluster

When you are given an account on the cluster, this means you can log into the head node, and that you have file space there. You can transfer files between your desktop and your head node directory. You can compile programs on the head node, but you should not run your program there!

The compute nodes are designed to do nothing but run big programs. The only program that talks to the compute nodes is the queueing system. When you want to execute a job on the compute nodes, you log into the head node and send a message to the queueing system describing your needs.

Transfer files using **sftp** or **scp**;
Log in using the **ssh** program or **putty**.

BATCH: Executing Jobs on the Clusters

To run a program on a cluster requires several steps:

- *transfer* program text to an interactive "head node";
- *log in* to the interactive "head node";
- *compile* your program to make the executable;
- *execute* your program indirectly, using a batch file, which communicates with the "compute nodes";
- *wait* for your program to run, then examine the results.

BATCH: File Transfer, Login, Compilation

Transfer the file "hello.c" to the headnode:

```
sftp ithaca2.arc.vt.edu
cd project
put hello.c
quit
```

Login and make the executable program "hello":

```
ssh ithaca2.arc.vt.edu
cd project
icc -openmp -parallel hello.c
mv a.out hello
```

BATCH: The Job Script File

To run the executable program **hello** on the cluster, you write a "job script", or "batch file", which might be called *hello.sh*.

The job script includes two parts:

- *queue parameters*, the account information, time limits, the number of processors you want. These lines always begin with **#PBS**
- *system commands*, the same commands you would type if you could execute the program interactively.

Batch jobs: Example Job Script File

```
#!/bin/bash
#PBS -l walltime=00:00:30
#PBS -l nodes=1:ppn=8
#PBS -W group_list=ithaca
#PBS -q ithaca_q
#PBS -A hpcb0001
#PBS -j oe

cd $PBS_O_WORKDIR

export OMP_NUM_THREADS=8

./hello
```

Batch jobs: Important items in job script file

- **!/bin/bash** selects the shell (must be first line!);
- **-l walltime=00:00:30** requests hours:minutes:seconds time;
- **-l nodes=1:ppn=8** asks for 1 node, and all 8 processors;
- **-W group_list=ithaca** specifies your group;
- **-q ithaca_q** requests that the job run on the Ithaca queue;
- **-A hpcb0001** change this to your account number;
- **-j oe** joins the output and error logs into one file;
- **cd $PBS_O_WORKDIR** starts execution in the same directory from which this batch file was submitted;
- **export OMP_NUM_THREADS=8** sets the number of OpenMP threads;
- **./hello** runs your program.

Batch jobs: Submit the job script

To run a job, you log into the head node. We'll assume you've moved to a directory containing your executable and the job script.

The **qsub** command asks the queueing system to process your job script:

```
qsub hello.sh
```

The queueing system responds with a short message:

```
111484.queue.tcf-int.vt.edu
```

The important information here is your job's ID **111484** which can be used to determine the status of the job, to kill the job, and which will also be used to name the output file of results.

Batch jobs: Wait for the script to run

Your job probably won't execute immediately. If the system is busy, or you've asked for a long execution time, you should probably log out and check back later.

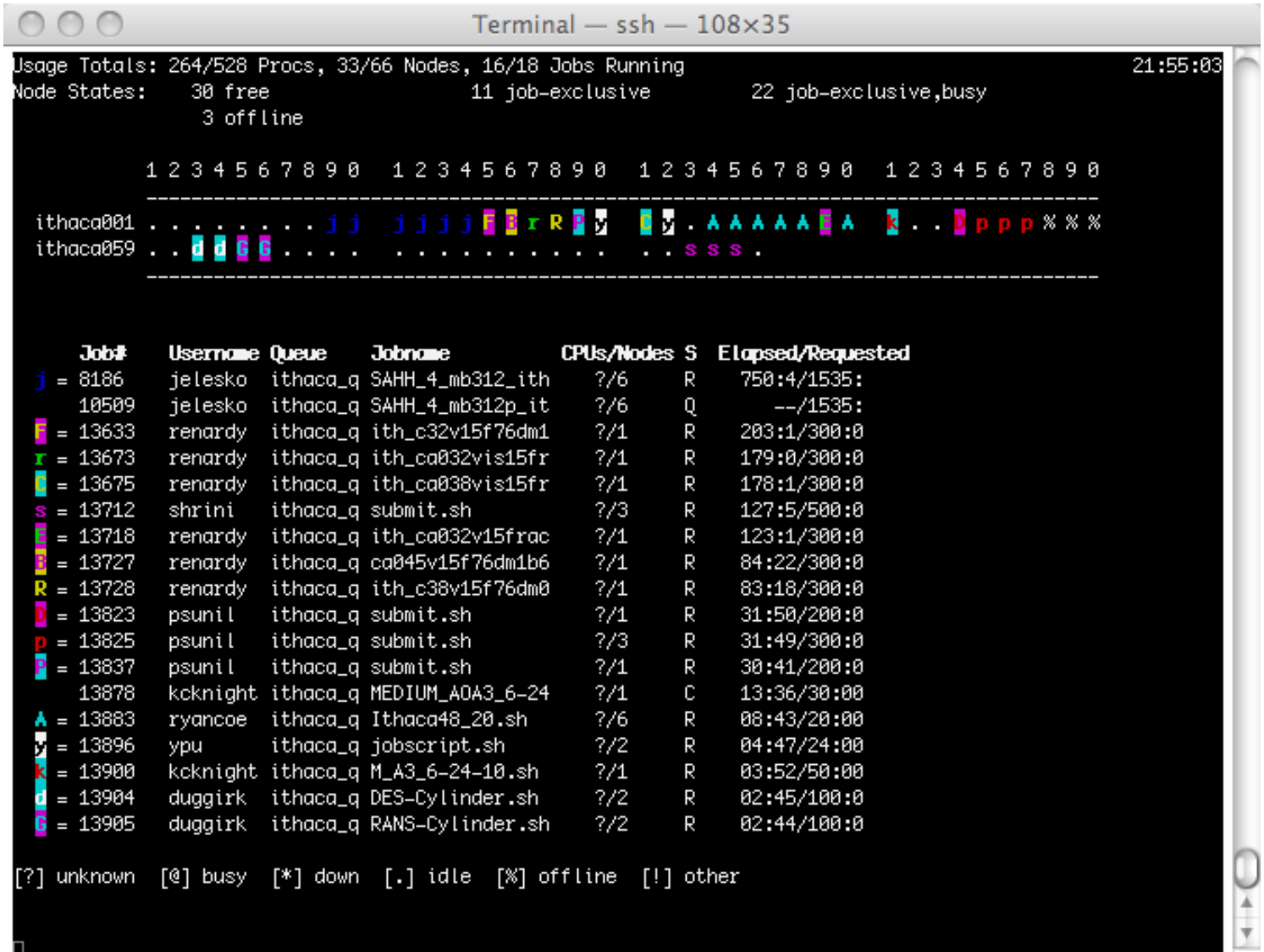
Some useful queue commands include:

`showq`	*status of all jobs and all queues for everyone*
`qstat -a`	*status of all jobs for this machine's queues*
`showq | grep burkardt`	*status of my jobs*
`showq | grep 111484`	*status of job 111484*
`qdel 111484`	*kill job 111484*
`showstart 111484`	*estimated start time for job*
`qstat -f 111484`	*all information about job*

Status is "Q" for queued, "R" for running, "C" for completed.

Batch jobs: PBSTOP

The **pbstop** program on Ithaca can show the queue status.



Batch jobs: Output files

Your results are returned in a file named **hello.o111484**.

If your program failed unexpectedly, this file contains messages explaining the sudden death of your program.

Otherwise, it contains all the data which would have appeared on the screen if you'd run the program interactively.

Of course, if your program also writes data files, these simply appear in the directory where the program ran.

Batch jobs: Examining the output

To see the output, type:

```
more hello.o111484
```

If you want a copy back on your desktop, you can use the **sftp** program to retrieve a copy:

```
sftp ithaca2.arc.vt.edu
cd project      <-- perhaps move to a subdirectory on Ithaca
get hello.o111484
quit
```

Batch jobs: Scrambled Output

Individual thread output lines may be "scrambled":

```
Thread 3 says 'Hello, world!'
Thread 1 says 'Hello, world!'
Thread 0 says 'Hello, world!'
Thread 2 says 'Hello, world!'
```

In C++ programs, output characters can get shuffled!

```
This is proc  This is process 2
ess 0
This is process 1
```

It's probably a good idea to have just the master process do printing, unless you're trying to debug the program.

Conclusion: The Future, 2, 4, 8, ... 4096 cores

The potential performance improvement of an OpenMP depends on hardware. Dual and quadcore shared-memory systems are common, and 8 core systems are not hard to find.

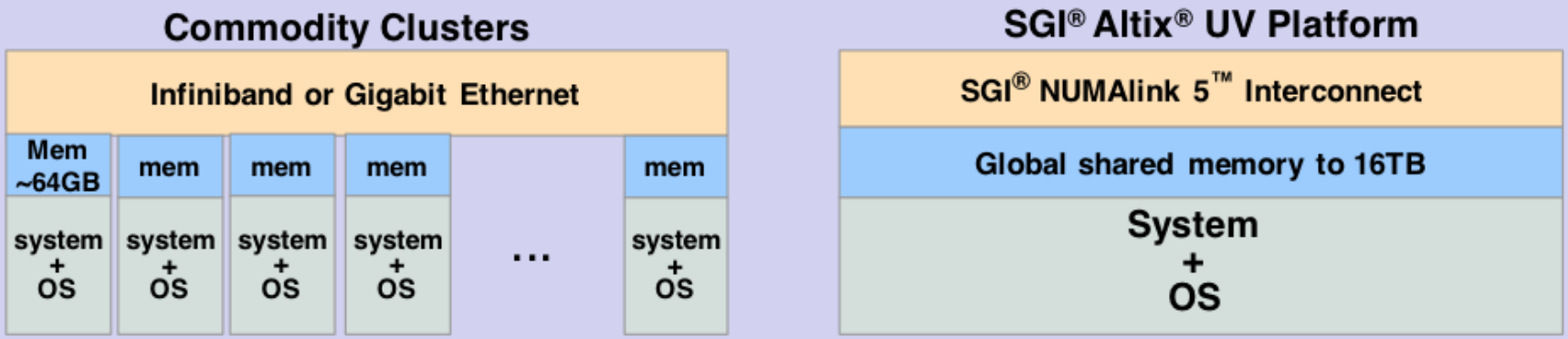
Since CPU's can't run faster, the future will involve multicore systems. The number of cores available on desktop systems is expected to double about every two years, following the trends of supercomputers.

For instance, SGI has announced a shared memory system called Altix UV, allowing as many as 2,048 cores.

The Pittsburgh Supercomputing Center, which offers free computing access to any researcher with an NSF grant, is about to offer a 4,096 core Altix UV system.

Conclusion: Distributed vs Shared Memory

Comparison of Distributed-Memory Cluster to Shared-Memory SGI Altix UV
Source: SGI



- Each system has own memory and OS
- Nodes communicate over commodity interconnect
- Inefficient cross-node communication creates bottlenecks
- Coding required for parallel code execution

- All nodes operate on one large shared memory space
- Eliminates data passing between nodes
- Big data sets fit entirely in memory
- Less memory per node required
- Simpler to program
- High Performance, Low Cost, Easy to Deploy

Conclusion: Example Programs

http://people.sc.fsu.edu/~jburkardt/c_src/c_src.html
or
http://people.sc.fsu.edu/~jburkardt/f_src/f_src.html

- **dijkstra_open_mp** (minimum distance)
- **fft_open_mp** (Fourier transform)
- **hello_open_mp** (Hello, world!)
- **md_open_mp** (molecular dynamics)
- **mxv_open_mp** (matrix times vector)
- **open_mp** (compute_pi, dot_product, helmholtz)
- **prime_open_mp** (count prime numbers)
- **quad_open_mp** (estimate integral)
- **satisfy_open_mp** (seek solutions to logical formula)
- **sgefa_open_mp** (Gauss elimination)
- **ziggurat_open_mp** (random numbers)

Conclusion: Exercises

This afternoon, we will have a hands-on session, in which you are encouraged to try some exercises involving OpenMP programming.

If you have a multicore laptop, and a compiler that supports OpenMP, you can do the work there instead of on Ithaca.

The exercises are described at:

http://people.sc.fsu.edu/~jburkardt/vt2/bootcamp_2010.html

- **hello** (Hello, world!)
- **quad2d** (approximate integration in 2D)
- **heated_plate** (heat equation)
- **schedule** (counting primes)

Conclusion: 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/`