OpenMP Shared Memory Programming

John Burkardt
Information Technology Department
Virginia Tech

.....

FDI Summer Track V:
Using Virginia Tech High Performance Computing
https://people.sc.fsu.edu/~jburkardt/presentations/...
openmp_2009_vt.pdf

26-28 May 2009

OpenMP Shared Memory Programming

- **Introduction**
- Threads
- Directives
- The SAXPY Example
- The DOT_PRODUCT Example
- The PRIME_SUM Example
- The MD Example
- OpenMP on VT's SGI Cluster
- OpenMP Utility Functions
- Conclusion

Introduction: 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.

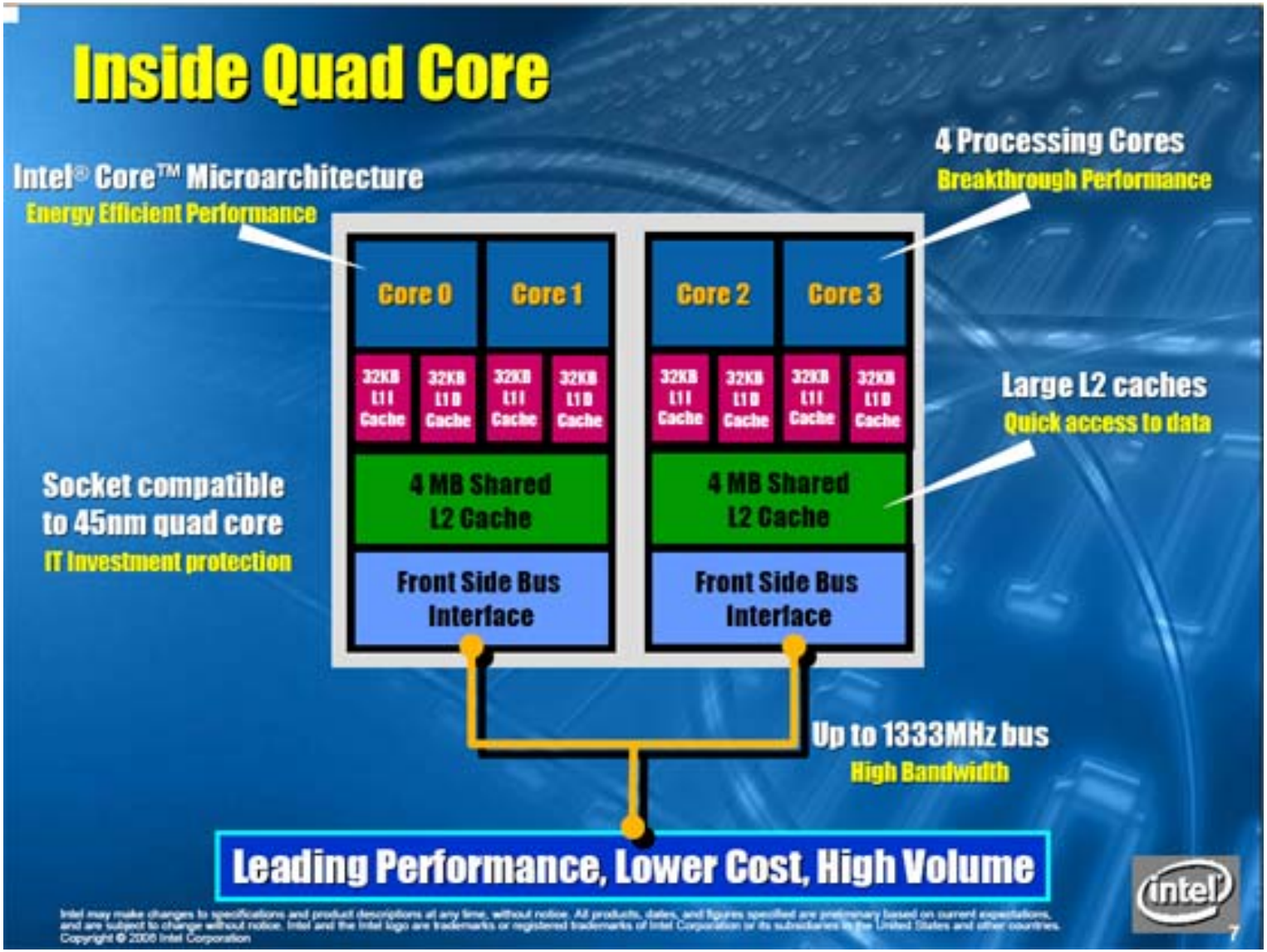
Introduction: A Shared Memory System

A shared memory system might be:

- a single core chip (older PC's, sequential execution)
- a multicore chip (such as your laptop?)
- multiple single core chips in a **NUMA** system
- multiple multicore chips in a **NUMA** system (VT SGI system)

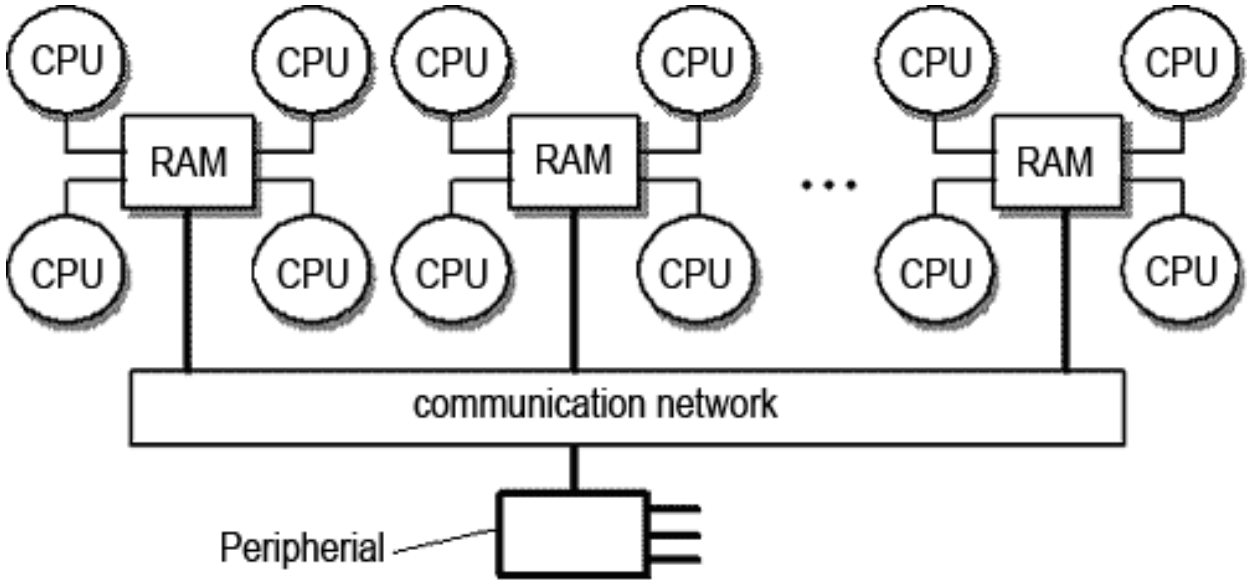
Introduction:

OpenMP can run on a single multicore processor:



Introduction: NUMA Shared Memory

VT's SGI ALTIX systems use the NUMA model.



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

Introduction: Steps in Using OpenMP

An OpenMP user must edit, compile and run:

- insert "directives" in a C or FORTRAN program;
- compile the program with OpenMP switches;
- set an environment variable for number of threads;
- run the program.

Introduction: Compiler Support

OpenMP compilers:

- Gnu **gcc/g++** 4.2, **gfortran** 2.0;
- IBM **xlc**, **xlf**
- Intel **icc**, **ifort**
- Microsoft Visual C++ (2005 Professional edition)
- Portland C/C++/Fortran, **pgcc**, **pgf95**
- Sun Studio C/C++/Fortran

Introduction: Compilation with Gnu Compilers

For the GNU compilers, include the **fopenmp** switch:

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

Introduction: Compilation with Intel Compilers

Intel compilers require the **openmp** and **parallel** switches.
Fortran programs also need the **fpp** switch:

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

Introduction: Compilation with IBM Compilers

For the IBM compilers, include the **omp** switch:

- xlc_r -**qsmp=omp** myprog.c
- xlC_r -**qsmp=omp** myprog.cpp
- xlf_r -**qsmp=omp** myprog.f
- xlf_r -**qsmp=omp** myprog.f90

OpenMP Shared Memory Programming

- Introduction
- **Threads**
- Directives
- The SAXPY Example
- The DOT_PRODUCT Example
- The PRIME_SUM Example
- The MD Example
- OpenMP on VT's SGI Cluster
- OpenMP Utility Functions
- Conclusion

Threads

OpenMP runs a program in parallel by dividing one task into several subtasks, each of which is assigned to a separate **threads**.

Each thread is an independent but "obedient" entity. It has access to the shared memory. It has "private" space for a small amount of working data.

We usually ask for one thread per available core:
ask for fewer, some cores are idle;
ask for more, some cores will run several threads, (probably slower).

An OpenMP program begins with just one thread running, the **master thread**.

The other threads begin in **idle** mode, waiting for work.

Threads: Fork and Join

The program encounters a **parallel** directive inserted by the user, which indicates the beginning of a parallel region.

The master thread activates the idle threads. (Technically, the master thread **forks** into multiple threads.)

Chunks of work are assigned to each thread, until it is complete.

The end of the parallel region is an implicit **barrier**. Program execution will not proceed until all threads have exited the parallel region and **joined** the master thread. (This is called "synchronization".)

The helper threads go idle until the next parallel section.

Threads: Sections

Inside a parallel region, the user has to help OpenMP by indicating what kind of work is to be parallelized.

The easiest situation occurs if there are actually several independent tasks to be done. OpenMP's term for this is **sections**.

For instance, the following three calculations could be treated as sections and done in parallel:

```
a = matrix_multiply ( b, c );
e = inverse ( b );
f = eigenvalues ( b );
```

However, these sorts of parallel sections are not the most common application for OpenMP.

Threads: Loops

OpenMP is ideal for parallel execution of **for** or **do** loops.

It's really as though we had a huge number of parallel sections, which are all the same except for the iteration counter **I**.

To execute a loop in parallel requires a **parallel** directive, followed by a **for** or **do** directive.

We'll look at a simple example of such a loop to get a feeling for how **OpenMP** works.

Threads: How Loops Are Handled

OpenMP assigns "chunks" of the index range to each thread.

It's as though 20 programs (threads) are running at the same time.

In fact, that's exactly what is happening!

If you apply OpenMP to a nested loop, only the outer loop is parallelized. If the outer loop index is very small, you may want to try to invert the loops, or apply the OpenMP directive to the inner loop.

```
for ( i = 0; i < 3; i++ )
  for ( j = 0; j < 100000; j++ )
```

Loops: Default Behavior

When OpenMP splits up the loop iterations, it has to decide what data is **shared** (in common), and what is **private** (each thread gets a separate variable of the same name).

Each thread automatically gets a private copy of the loop index.

In FORTRAN only, each thread automatically gets a private copy of the loop index for any loops nested inside the main loop. In C/C++, nested loop indices are not automatically "privatized".

By default, all other variables are shared.

A simple test: if your loop executes correctly even if the iterations are done in reverse order, things are probably going to be OK!

Threads: Shared and Private Data

In the ideal case, each iteration of the loop uses data in a way that doesn't depend on other iterations and doesn't interfere with them. Loosely, this is the meaning of the term **shared** data.

A nice example is the **SAXPY** computation, which adds a multiple of one vector to another:

y(i) = s * x(i) + y(i)

OpenMP Shared Memory Programming

- Introduction
- Threads
- **Directives**
- The SAXPY Example
- The DOT_PRODUCT Example
- The PRIME_SUM Example
- The MD Example
- OpenMP on VT's SGI Cluster
- OpenMP Utility Functions
- Conclusion

Introduction: What Do Directives Look Like?

In C or C++, directives begin with the # comment character and the string **pragma omp** followed by the name of the directive.

```
# pragma omp parallel
# pragma omp sections
# pragma omp for
# pragma omp critical
```

Directives appear just before a block of code, which is delimited by { **curly brackets** } or the body of a **for** statement.

Introduction: What Do Directives Look Like?

The **parallel** directive begins a *parallel region*.

```
# pragma omp parallel
{
  do things in parallel here
}
```

Inside the parallel region, you can have any number of sections and loops.
You don't make the whole program one big parallel region because some things must only happen once (such as I/O), and in each parallel region, you must manage and declare private variables.

Introduction: What Do Directives Look Like?

Inside the parallel region, you might have sections:

```
# pragma omp parallel
{
  # pragma omp sections
  {
    # pragma omp section
    { a = matrix multiply ( b, c ); }
    # pragma omp section
    { e = inverse ( b ); }
    # pragma omp section
    { f = eigenvalues ( b ); }
  }
}
```

Introduction: What Do Directives Look Like?

If you have several loops in a row, try to include them all in one parallel region:

```
!$omp parallel
  !$omp do
    do i = 1, nedge
      parallel loop 1
    end do
  !$omp end do
  !$omp do
    do j = 1, nface
      parallel loop 2
    end do
  !$omp end do
!$omp end parallel
```

Introduction: What Do Directives Look Like?

The end of each loop normally forces all threads to wait. If there are several loops in one parallel region, you can use a **nowait** command to let a fast thread move on to the next one.

```
!$omp parallel
!$omp do nowait
  do i = 1, nedge
    parallel loop 1
  end do
!$omp end do
!$omp do
  do j = 1, nface
    parallel loop 2
  end do
!$omp end do
!$omp end parallel
```

Introduction: What Do Directives Look Like?

CLAUSES are additional information included on a directive.

There are clauses to define lists of **private** or **shared** variables.

(When multiple threads are running a loop, each thead gets its own copy of the private variables.)

```
# pragma omp parallel shared (n,s,x,y) private (i,t)

# pragma omp for
for ( i = 0; i < n; i++ )
{
  t = tan ( y[i] / x[i] );
  x[i] = s * x[i] + t * y[i];
}
```

Introduction: Long Directive Lines

You may often find that the text of a directive becomes rather long.

In C and C++, you can break the directive at a convenient point, interrupting the text with a backslash character, \, and then continuing the text on a new line.

```
# pragma omp parallel \
  shared ( n, s, x, y ) \
  private ( i, t )


# pragma omp for
for ( i = 0; i < n; i++ )
{
  t = tan ( y[i] / x[i] );
  x[i] = s * x[i] + t * y[i];
}
```

Introduction: What Do Directives Look Like?

FORTRAN77 directives begin with the string **c$omp**.

Directives longer than 72 characters must continue on a new line. with another **c$omp** marker **AND** a continuation character in column 6, such as **&**.

```
c$omp parallel
c$omp&  shared ( n, s, x, y )
c$omp&  private ( i, t )

c$omp do
do i = 1, n
  t = tan ( y(i) / x(i) )
  x(i) = s * x(i) + t * y(i)
end do
c$omp end do
c$omp end parallel
```

Introduction: What Do Directives Look Like?

FORTRAN90 directives begin with the string **!$omp**.

Long lines may be continued using a terminal **&**.

The continued line also begins with the **!$omp** marker.

```
!$omp parallel &
!$omp   shared ( n, s, x, y ) &
!$omp   private ( i, t )

!$omp do
do i = 1, n
  t = tan ( y(i) / x(i) )
  x(i) = s * x(i) + t * y(i)
end do
!$omp end do
!$omp end parallel
```

Introduction: What Do Directives Do?

- begin a parallel section of the code:
  # pragma omp parallel
- mark variables that must be kept private:
  # pragma omp parallel private (x, y, z)
- suggest how some results are to be combined into one:
  # pragma omp parallel reduction (+ : sum)
- indicate code that only one thread can do at a time:
  # pragma omp critical
  # pragma omp end critical
- force threads to wait til all are done:
  # pragma omp barrier

Introduction: What Do Directives Do?

- Work to be done in a loop:
  # pragma omp for
- Work to be done in a loop; when done, don't wait!:
  # pragma omp for nowait
- suggest how loop work is to be divided:
  # pragma omp for schedule (dynamic)
- Work has been divided into user-defined "sections":
  # pragma omp sections
- Work to be done using FORTRAN90 implicit loops:
  !$omp workshare

OpenMP Shared Memory Programming

- Introduction
- Threads
- Directives
- **The SAXPY Example**
- The DOT_PRODUCT Example
- The PRIME_SUM Example
- The MD Example
- OpenMP on VT's SGI Cluster
- OpenMP Utility Functions
- Conclusion

The SAXPY Example

OpenMP is most often used to parallelize loops.

A loop is probably OK for parallel execution if, even in sequential mode, the iterations could be performed in any order.

An example is the **saxpy** loop, adding a multiple of one vector to another.

```
for { i = 0; i < n; i++ }
{
  y[i] = a * x[i] + y[i];
}
```

Loops: SAXPY with OpenMP Directives

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

double *random_vector ( int n );

int main ( int argc, char *argv[] )
{
  int i, n = 1000;
  double *x, *y, s;

  s = 123.456;
  x = random_vector ( n );
  y = random_vector ( n );

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

The SAXPY Example

We have just seen our first examples of **directives**.

An OpenMP directive has the form of a comment. So unless the compiler is told to pay attention to them, the compiled code runs sequentially.

The **parallel** directive indicates the beginning of a section of code to be executed in parallel. It applies to the next statement, or loop, or group of statements that have been surrounded by curly brackets.

The SAXPY Example

```
# pragma omp parallel
{
# pragma omp for
  for ( i = 0; i < n; i++ )
  {
    loop #1 is inside the parallel section
  }
# pragma omp for
  for ( j = 0; j < m; j++ )
  {
    loop #2 is inside the parallel section
  }
}   <-- parallel section ends
```

The SAXPY Example

The **for** directive indicates that the following statement begins a loop, and that this loop should be done in parallel.

The iterations of a parallel loop are split up among the threads.

Loops are often nested;
Normally, a programmer only parallelizes one loop in a nest.

The SAXPY Example

```
# pragma omp parallel
{
# pragma omp for
  for ( i = 0; i < m; i++ ) {
    for ( j = 0; j < n; j++ ) {
      Parallelization is on outer index I.
    }
  }
  for ( i = 0; i < m; i++ ) {
    # pragma omp for
    for ( j = 0; j < n; j++ ) {
      Parallelization is on inner index J.
    }
  }
}
```

The SAXPY Example

The **private** directive indicates that the following variables are not to be shared.

In a loop, the loop index variable must be made private.

This gives each thread its own variable called **I**, which it can use to keep track of the iterations it is performing.

The SAXPY Example: Compile and Load

The compilation step might be:

```
gcc -c -fopenmp saxpy.c
gcc saxpy.o random_vector.o
mv a.out saxpy
```

We assume "random_vector" was precompiled.
(It does NOT have to be compiled with the OpenMP switch.)

The SAXPY Example: Run

Set the number of threads in Bourne/Korne/Bourne-Again shells:

```
export OMP_NUM_THREADS=4
```

or, in the C or T shells:

```
setenv OMP_NUM_THREADS 4
```

and now run the program:

```
./saxpy
```

The SAXPY Example: Timing

Check total wall clock execution time versus thread numbers:

```
export OMP_NUM_THREADS=1
time ./saxpy
export OMP_NUM_THREADS=2
time ./saxpy
export OMP_NUM_THREADS=4
time ./saxpy
export OMP_NUM_THREADS=8
time ./saxpy
```

You control the number of threads externally to the program. The best number is usually equal to the number of cores or coprocessors available.

OpenMP Shared Memory Programming

- Introduction
- Threads
- Directives
- The SAXPY Example
- **The DOT_PRODUCT Example**
- The PRIME_SUM Example
- The MD Example
- OpenMP on VT's SGI Cluster
- OpenMP Utility Functions
- Conclusion

DOT_PRODUCT

Most programs don't run in parallel so easily as SAXPY did.

Sometimes the threads must cooperate in computing a single value.

Reduction operations are simple calculations which can be carried out in parallel if you warn OpenMP in advance.

Reduction operations include:

- maximum or minimum
- norm
- sum
- product

and some other less frequent ones.

A **reduction** directive warns OpenMP that a particular variable is the result of a reduction operation.

DOT_PRODUCT: With OpenMP Directives

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

int main ( int argc, char *argv[] )
{
  int i, n = 1000;
  double *x, *y, xdoty;

  x = random_vector ( n );
  y = random_vector ( n );

  xdoty = 0.0;

# pragma omp parallel \
  shared ( n, x, y ) \
  private ( i ) \
  reduction ( + : xdoty )

# pragma omp for
  for ( i = 0; i < n; i++ )
  {
    xdoty = xdoty + x[i] * y[i];
  }
  printf ( "XDOTY = %e\n", xdoty );
  return 0;
}
```

DOT_PRODUCT: The reduction clause

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

Reduction clause examples include:

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

DOT_PRODUCT: The shared clause

Another OpenMP directive allows you to declare the variables which are to be shared.

Every variable in a loop should be exactly one of **shared**, **private** or **reduction**.

You don't have to explicitly declared shared variables; that is the default status for variables in a loop.

(However, in FORTRAN codes, the loop index will have private status by default)

It can be helpful to declare all your variables.

OpenMP Shared Memory Programming

- Introduction
- Threads
- Directives
- The SAXPY Example
- The DOT_PRODUCT Example
- **The PRIME_SUM Example**
- The MD Example
- OpenMP on VT's SGI Cluster
- OpenMP Utility Functions
- Conclusion

PRIME_SUM

Most programs don't run in parallel so easily as SAXPY did.

The PRIME_SUM program illustrates the biggest concern for OpenMP programs:

How can we eliminate data conflicts between threads?

Conflicts can occur because the loop

- uses a temporary variable whose value changes;
- computes a sum, product, or maximum during the loop;
- needs to read and write values in the same data item

PRIME_SUM: A Sequential Version

```
# include <cstdlib>
# include <iostream>
using namespace std;

int main ( int argc, char *argv[] )
{
  int i, j, total;
  int n = 1000;
  bool prime;

  total = 0;
  for ( i = 2; i <= n; i++ )
  {
    prime = true;

    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = false;
        break;
      }
    }
    if ( prime )
    {
      total = total + i;
    }
  }
  cout << "PRIME_SUM(2:" << n << ") = " << total << "\n";
  return 0;
}
```

PRIME_SUM: What Data Cannot be Shared?

In PRIME_SUM, a given thread, carrying out iteration **I**:

- works on an integer **I**
- initializes **PRIME** to be TRUE
- checks if any **J** divides **I** and resets **PRIME** if necessary;
- adds **I** to **TOTAL** if **PRIME** is TRUE.

If multiple threads are running with different values of I at the same time, then the variables **I**, **J**, **PRIME** and **TOTAL** represent possible data conflicts.

PRIME_SUM: With OpenMP Directives

```
# include <cstdlib>
# include <iostream>
# include <omp.h>
using namespace std;
int main ( int argc, char *argv[] )
{
  int i, j, total, n = 1000, total = 0;
  bool prime;

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

# pragma omp for
  for ( i = 2; i <= n; i++ )
  {
    prime = true;
    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = false;
        break;
      }
    }
    if ( prime ) { total = total + i; }
  }
  cout << "PRIME_SUM(2:" << n << ") = " << total << "\n";
  return 0;
}
```

OpenMP Shared Memory Programming

- Introduction
- Threads
- Directives
- The SAXPY Example
- The DOT_PRODUCT Example
- The PRIME_SUM Example
- **The MD Example**
- OpenMP on VT's SGI Cluster
- OpenMP Utility Functions
- Conclusion

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

```
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
```

MD: With OpenMP Directives

```
!$omp parallel &
!$omp shared ( n, coord, f ) &
!$omp private ( i, j, k, d, dif )

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

Data Classification (Private/Shared/Reduction)

In the previous example, the variable **D** looked like a reduction variable.

But that would only be the case if the loop index **K** was executed as a **parallel do**.

We could work very hard to interchange the order of the I, J and K loops, or even try to use nested parallelism on the K loop.

But these efforts would be pointless, since the loop runs from 1 to 3, a range too small to get a parallel benefit.

- Introduction
- Threads
- Directives
- The SAXPY Example
- The DOT_PRODUCT Example
- The PRIME_SUM Example
- The MD Example
- **OpenMP on VT's SGI Cluster**
- OpenMP Utility Functions
- Conclusion

OpenMP at VT: File Transfer/Compilation

Virginia Tech has 3 clusters of SGI ALTIX 3700 machines using Intel processors.

For this class, we will have access to the cluster known as **inferno2**.

inferno2 has a total of 128 CPU's; however, an individual job is only allowed access to at most 12 CPU's.

Users transfer files, compile programs, and submit jobs from one of the two head nodes, known as **charon1** and **charon2**.

Files are transferred with the **sftp** program:

```
sftp my_name@charon1.arc.vt.edu
put hello.c
get hello_output.txt
quit
```

OpenMP at VT: Creating an Executable Program

Assuming your files are available on **charon**, you need to login using the **ssh** program so that you can compile the files and submit your jobs.

```
ssh my_name@charon1.arc.vt.edu
```

We need to use the Intel compilers. To access OpenMP, the C compilers need two extra switches, and the Fortran compilers need three!

```
icc -openmp -parallel hello.c
icpc -openmp -parallel hello.cpp
ifort -openmp -parallel -fpp hello.f
ifort -openmp -parallel -fpp hello.f90
```

Batch jobs: Ready to Run

The compile command creates an executable program called **a.out**. It's probably best to rename it using the **mv** command:

```
mv a.out hello
```

Once you have created the executable, you are *almost* ready to go!

However, on the SGI system, *interactive jobs are not allowed.*

Instead, you put your job into a queue with the jobs requested by other users, so they can be run in an orderly fashion.

This is called the **batch** or **queueing** system.

Batch jobs: The Job Script File

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

The job script file describes the account information, time limits, the number of processors you want, input files, and the program to be run.

The job script file can look pretty confusing, but the good news is that there are only a few important lines!

Batch jobs: Example Job Script File

```
#!/bin/bash
#PBS -lwalltime=00:00:30
#PBS -lncpus=4
#PBS -W group_list=sgiusers
#PBS -q inferno2_q
#PBS -A hpcb0001

cd $PBS_O_WORKDIR

export OMP_NUM_THREADS=4

./hello
```

Batch jobs: Important items in job script file

In this job script, the important items are:

- **walltime** lists your job time limit in seconds
- **ncpus=4** asks for 4 processors. 12 is the maximum.
- **hpcb0001** is the account under which you are running.
- **export OMP_NUM_THREADS=4** sets the number of threads. This should match the value of **ncpus**!
- **./hello &> hello_output.txt** runs your program and saves the output to a particular file.
- **./hello** would also work; in this case, the queueing system will save the output for you.

Batch jobs: Submit the job script

To run your job, you use the **qsub** command to send your job script file:

You submit the job, perhaps like this:

```
qsub hello.sh
```

The queueing system responds with a short message:

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

The important information is your job's ID **111484**.

Batch jobs: Wait for the script to run

Your job probably won't execute immediately. To check on the status of ALL the jobs for everyone, type

```
showq
```

Since the **showq** command lists each job by number and username, you can check for just your job number:

```
showq | grep 111484
```

If for some reason you want to kill your job, you can type

```
qdel 111484
```

Batch jobs: Output files

When your job is done, the queueing system gives you two files:

- an *output* file, such as **hello.o111484**
- an *error* file, such as **hello.e111484**

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

Otherwise, the interesting information is in the output file, which 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 your home directory when the program is completed.

Batch jobs: Examining the output

To see our output file, we type:

```
more hello_output.txt
```

OpenMP output from different processes may be "shuffled":

```
HELLO
  FORTRAN90/OpenMP version
  The number of processors available:
  OMP_GET_NUM_PROCS () =      118
  The number of threads is        4
  This is process         1
  This is process         2
  This is process         0
  This is process         3
HELLO
  Normal end of execution.
```

OpenMP Shared Memory Programming

- Introduction
- Threads
- Directives
- The SAXPY Example
- The DOT_PRODUCT Example
- The PRIME_SUM Example
- The MD Example
- **OpenMP Utility Functions**
- Conclusion

OpenMP Utility Functions

OpenMP functions include:

- **omp_set_num_threads (t_num)**
- **t_num = omp_get_num_threads ( )**
- **p_num = omp_get_num_procs ( )**
- **t_id = omp_get_thread_num ( )**
- **wtime = omp_get_wtime()**

OpenMP Utility Functions: Number of Processors

The function **omp_get_num_procs ( )** returns the number of processors (or coprocessors or cores) that are available to the program.

This is telling you something about the hardware.

It is also suggesting the maximum parallel speedup you can expect.

On **inferno2** there may be as many as 128 processors "available"; however, most jobs are only actually allowed a maximum of 12!

OpenMP Utility Functions: Number of Threads Available

By default, the number of threads of execution is 1 (no parallelism!).

By setting the environment variable **OMP_NUM_THREADS**, the user can set the maximum level of parallelism.

To find out the number of threads inside the program, call **omp_get_num_threads ( )**.

You must call this function inside a parallel section; otherwise the answer will be 1!

OpenMP Utility Functions: Maximum Number of Threads

If you just want to know the maximum number of threads available, the simplest way is to call **omp_get_max_threads ( )**.

This gives you the value of **OMP_NUM_THREADS**, whether you are in a parallel or non-parallel region.

OpenMP Utility Functions: Which Thread am I?

Inside a parallel region, you can call **omp_get_thread_num ( )** to tell you which thread is executing this iteration of the loop.

You probably want to store the result in a private variable!

```
t_id = omp_get_thread_num ( )
write ( *, * ) 'Thread ', t_id, ' is running.'
```

OpenMP Utility Functions: How Much Time Has Elapsed

If you want to time a piece of parallel code, you do this **outside** of the parallel section. You call **omp_get_wtime** which returns a double precision real number representing a reading of the elapsed wall clock time.

```
wtime = omp_get_wtime ( )
!$ omp parallel
  STUFF TO TIME
!$ omp end parallel
wtime = omp_get_wtime ( )
write ( *, * ) '  Section took ', wtime, ' seconds.'
```

- Introduction
- Threads
- Directives
- The SAXPY Example
- The DOT_PRODUCT Example
- The PRIME_SUM Example
- The MD Example
- OpenMP Utility Functions
- **Conclusion**

CONCLUSION: OpenMP Has a Future

This has been a very brief introduction to the power of OpenMP.

OpenMP is a simple method for adding parallelism.

OpenMP is limited by the number of cores available on a particular shared memory system.

But systems with 10 or 100 cores are already scheduled to appear soon.

CONCLUSION: OpenMP is Flexible

OpenMP can be used on a desktop;

It can be used on any cluster that behaves like a shared memory machine.

It can be used in combination with MPI; MPI copies a program to each node of a cluster. On each node, OpenMP is able to exploit the multiple cores available.

CONCLUSION: OpenMP is Flexible

If you already have a program written, it is easy to add OpenMP directives to one part of the program at a time, and gradually create a parallel version.

At the same time, the original sequential program can be run simply by a compiler switch.

OpenMP offers an easy, portable and reversible path to parallel programming.