5: Introductory OpenMP

John Burkardt
Information Technology Department
Virginia Tech

.....

FDI Summer Track V:
Parallel Programming

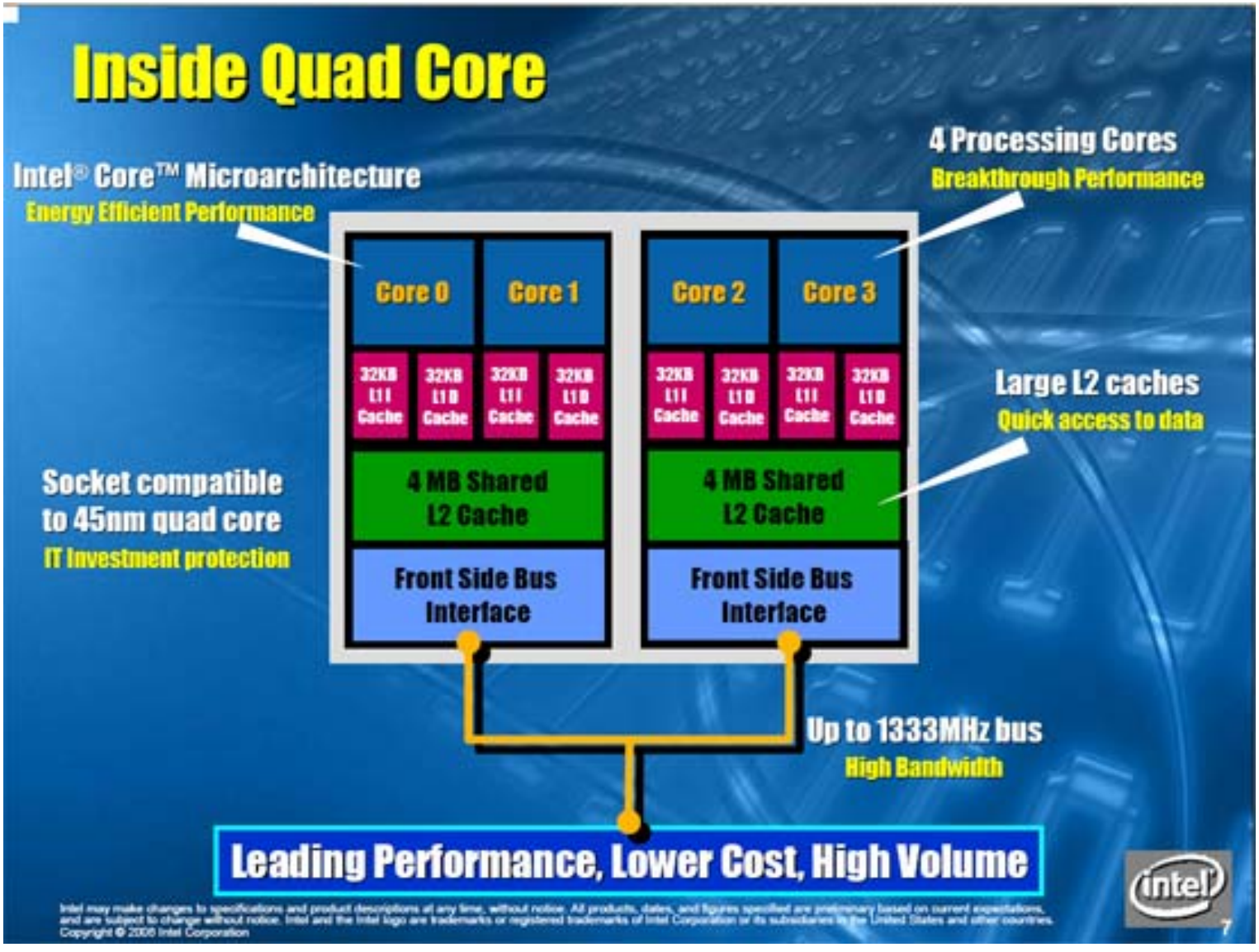
.....

https://people.sc.fsu.edu/~jburkardt/presentations/...
openmp1_2008_vt.pdf

10-12 June 2008

Introduction

OpenMP is a bridge between yesterday's programming languages and tomorrow's multicore chips.



Introduction: Where OpenMP is Used

OpenMP runs a user program on any 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)

OpenMP can also be combined with **MPI**, but that's an advanced topic!

Introduction: How OpenMP is Used

The user inserts OpenMP "directives" in a program.

The user compiles the program with OpenMP directives enabled.

The number of "threads" is chosen by an environment variable or a function call.

(Usually set the number of threads to the number of processors)

The user runs the program.

Introduction: Compiler Support

Compiler writers **do** support OpenMP:

- 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

You build a parallel version of your program by telling the compiler to activate the OpenMP directives.

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

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

Introduction: Compilation with Intel Compilers

For the Intel compilers, include the **openmp** switch:

- **icc -openmp -parallel myprog.c**
- **icpc -openmp -parallel myprog.C**

Fortran programs also require the **fpp** switch:

- **ifort -fpp -openmp -parallel myprog.f**
- **ifort -fpp -openmp -parallel myprog.f90**

Introduction: What Do Directives Look Like?

In C or C++, directives begin with the # preprocessor comment character and the string **pragma omp**.

```
#pragma omp parallel for private ( i, j )
```

In Fortran, directives begin with the **c** or **!** comment character and the string **$omp**.

```
!$omp parallel do private ( i, j )
```

Introduction: Long Directive Lines

By the way, you may find it necessary to write a directive that doesn't fit comfortably on one line.

In C and C++, you can end the first line with a final double backslash, which "escapes" the new line, or you can simply start another **#pragma omp** line.

In FORTRAN77, your directives can't be longer than 72 characters, and the next line must also be "commented out" with the **c$omp** marker and column 6 of the continuation line must have a continuation character in it such as **&**.

In FORTRAN90, you need to use the continuation character **&** to continue to the next line, and the next line must also be "commented out" with the **!$omp** marker.

Introduction: What Do Directives Do?

- indicate parallel sections of the code
- indicate code that only one thread can do at a time
- suggest how the work is to be divided
- mark variables that must be shared "carefully"
- suggest how some results are to be combined into one
- force threads to wait til all are done

Introduction: Threads

OpenMP assigns pieces of a computation to **threads**.

Each thread is an independent but "obedient" entity. It has access to the shared memory. It has "private" space for record keeping.

We usually assume that each core corresponds to one thread;

An OpenMP program begins with one **master thread** executing.

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

Introduction: Fork and Join

The program proceeds in sequential execution until it encounters a region that the user has marked as a **parallel section**

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

The work is divided up, and carried out.

The master thread waits as each helper thread completes its work and is able to **join** the master thread.

The helper threads go back on unemployment until the next parallel section.

Loops

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

In the simplest cases, all the user has to do is mark the loop with a **parallel** directive.

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

Loops: Default Behavior

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!

Each thread has its own private copy of the loop index.

All the other variables are shared, and open for "contention".

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

Loops: 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. Loosely, this is the meaning of the term **shared** data.

For instance, in the **SAXPY** task, each iteration is

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

You can imagine that any number of processors could cooperate in a calculation like this, with no interference.

We will start by assuming that all data can be shared, and wait for reality to correct us!

This is **OpenMP**'s default assumption, as well.

Loops: Sequential Version

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

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

  s = 123.456;

  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
    y[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
  }

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

Loops: The SAXPY task

The SAXPY task adds a multiple of vector **X** to vector **Y**.

The arrays **X** and **Y** can be shared, because only the thread associated with loop index **I** needs to look at the **I**-th entries.

Each thread will need to know the value of **S** but they can all agree on what that value is. (They "share" the same value).

This is a "perfect" parallel application: no private data, no memory contention.

SAXPY is a common low level task, as in Gauss elimination.

Loops: SAXPY with OpenMP Directives

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

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

  s = 123.456;

  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
    y[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
  }

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

Loops: C Syntax

We've included the <**omp.h**> file, but this is only needed to refer to predefined constants, or call **OpenMP** functions.

The **#pragma omp** string is a marker that indicates to the compiler that this is an **OpenMP** directive.

The **parallel for** clause requests parallel execution of a **for** loop.

The parallel section terminates at the closing of the **for** loop block.

Loops: Fortran Syntax

The **include 'omp_lib.h'** command is only needed to refer to predefined constants, or call **OpenMP** functions.

In FORTRAN90, try **use omp_lib** instead.

The marker string is **c$omp** or **!$omp**.

The **parallel do** clause requests parallel execution of a **do** loop.

In Fortran, *but not C*, the end of the parallel loop must also be marked. A **c$omp end parallel** directive is used for this.

Loops: SAXPY with OpenMP Directives

```
      program main

      include 'omp_lib.h'

      integer i, n
      double precision x(1000), y(1000), s

      n = 1000
      s = 123.456

      do i = 1, n
        x(i) = rand ( )
        y(i) = rand ( )
      end do

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

      stop
      end
```

Loops: Which of these loops are "safe"?

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

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

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

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

Loops: How To Think About Threads

To visualize parallel execution, suppose 4 threads will execute the 1,000 iterations of the SAXPY loop.

OpenMP might assign the iterations in chunks of 50, so thread 1 will go from 1 to 50, then 201 to 251, then 401 to 450, and so on.

Then you also have to imagine that the four threads each execute their loops more or less simultaneously.

Even this simple model of what's going on will suggest some of the things that can go wrong in a parallel program!

Loops: The SAXPY loop, as OpenMP might think of it

```
if ( thread_id == 0 ) then
  do ilo = 1, 801, 200
    do i = ilo, ilo + 49
      y(i) = y(i) + s * x(i)
    end do
  end do
else if ( thread_id == 1 ) then
  do ilo = 51, 851, 200
    do i = ilo, ilo + 49
      y(i) = y(i) + s * x(i)
    end do
  end do
else if ( thread_id == 2 ) then
  do ilo = 101, 901, 200
    do i = ilo, ilo + 49
      y(i) = y(i) + s * x(i)
    end do
  end do
else if ( thread_id == 3 ) then
  do ilo = 151, 951, 200
    do i = ilo, ilo + 49
      y(i) = y(i) + s * x(i)
    end do
  end do
end if
```

Loops: Comments

What about the loop that initializes **X** and **Y**?

The problem here is that we're calling the **rand** function.

Normally, inside a parallel loop, you can call a function and it will also run in parallel. However, the function cannot have *side effects*.

The **rand** function is a special case; it has an internal "static" or "saved" variable whose value is changed and remembered internally.

Getting random numbers in a parallel loop requires care. We will leave this topic for later discussion.

REDUCTION

OpenMP's default behavior for parallelizing loops only works for simple cases.

The user can help **OpenMP** to handle more complicated cases by using the appropriate directives.

Computing a dot product is an example where help is needed.

The variable summing the individual products is going to cause conflicts.

Inefficiencies (delays) occur as several processors want to read the same value.

Errors occur if several processors try to write their updated versions back to the single data location.

REDUCTION: Sequential Version

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

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

  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
    y[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
  }

  xdoty = 0.0;
  for ( i = 0; i < n; i++ )
  {
    xdoty = xdoty + x[i] * y[i];
  }
  printf ( "XDOTY = %e\n", xdoty );
  return 0;
}
```

REDUCTION: Examples of reduction operations

The dot product is one example of a **reduction operation**.

Other examples;

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

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

In **OpenMP**, a **reduction** declaration allows the compiler to set up the reduction correctly and efficiently.

REDUCTION: with OpenMP directives

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

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

  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
    y[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
  }

  xdoty = 0.0;
#pragma omp parallel for reduction ( + : xdoty )
  for ( i = 0; i < n; i++ )
  {
    xdoty = xdoty + x[i] * y[i];
  }
  printf ( "XDOTY = %e\n", xdoty );
  return 0;
}
```

REDUCTION: 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))

Private Data

OpenMP assumes that most of the problem data is **shared**, that is, there is a single copy, to which all threads have access.

If data is only "read", a single copy can obviously be shared, although delays might occur if several threads want to read at the same time.

Some data may be shared even though it is written. An example is an array **A**. If entry **A[I]** is only written during loop iteration **I**, then the array **A** can probably remain shared.

Private Data

A common example of **private** data, which cannot be shared, would be the temporary variables often used for convenience during a loop iteration.

For instance, it's common to create variables called **im1** and **ip1** equal to the loop index decremented and incremented by 1.

If many threads run the loop at the same time, they're all going to have different ideas of what **im1** and **ip1** should be, but only one place to put these values!

Private Data: The PRIME SUM task

The PRIME SUM task will illustrate the concept of private and shared variables.

Our task is to compute the sum of the prime numbers from 1 to N.

A natural formulation stores the result in **TOTAL**, then checks each number I from 2 to N.

To check if the number I is prime, we ask whether it can be evenly divided by any of the numbers J from 2 to $I - 1$.

We can use a temporary variable **PRIME** to help us.

Private Data: 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;
}
```

Private Data: Eliminating Data Conflicts!

Let's imagine we parallelize the **I** loop.

So each thread:

- works on an integer **I**
- initializes **PRIME** to be TRUE
- checks whether any **J** can divide **I** and resets **PRIME** if necessary;
- If **PRIME** ends up TRUE, add **I** to **TOTAL**.

The variables **J**, **PRIME** and **TOTAL** represent possible data conflicts that we must resolve.

Private Data: 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 for private ( i, prime, j ) shared ( n )
# pragma omp reduction ( + : total )
  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;
}
```

Private Data

By default, all variables in a loop are shared except for the main loop index.

We can override the defaults for one or all the variables.

Every variable in a loop is either **private**, or **shared** or **reduction**.

We didn't have to declare that **i** was private...but we did have to declare that **j** was private!

Data Dependence

When we discussed the example of a differential equation, we pointed out that the usual approach requires you to compute the approximate solution one step at a time.

Even though you can write the procedure as a loop, the problem is that each iteration of the loop depends on a result that is not available until the previous iteration is complete.

This is an example of data dependence. A data dependent calculation cannot be done in parallel.

In the **STEPS** program, we will look at a simpler case of data dependence, one which can be "cured".

Data Dependence

For the STEPS task, we evaluate a function at equally spaced points in the unit square.

Start **(X,Y)** at (0,0), increment **X** by **DX**. If **X** exceeds 1, reset to zero, and increment **Y** by **DY**.

This is a natural way to "visit" every point.

This simple idea won't work in parallel without some changes.

Each thread will need a private copy of **(X,Y)**.

...but, much worse, the value **(X,Y)** is data dependent.

Data Dependence: Sequential Version

```
program main

  integer i, j, m, n
  real dx, dy, f, total, x, y

  total = 0.0
  y = 0.0
  do j = 1, n
    x = 0.0
    do i = 1, m
      total = total + f ( x, y )
      x = x + dx
    end do
    y = y + dy
  end do

  stop
end
```

Data Dependence

To make the loop iterations independent,

- precompute **X(1:M)** and **Y(1:N)** in arrays.
- or notice $X = I/M$ and $Y = J/N$

The first solution, converting some temporary scalar variables to vectors and precomputing them, may be able to help you parallelize a stubborn loop. The second solution is simple and saves us a separate preparation loop and extra storage.

Data Dependence: With OpenMP directives

```
program main

  use omp_lib

  integer i, j, m, n
  real f, total, x, y

  total = 0.0
!$omp parallel do private ( i, j, x, y ) shared ( m, n ) reduction ( + : total )
  do j = 1, n
    y = j / real ( n )
    do i = 1, m
      x = i / real ( m )
      total = total + f ( x, y )
    end do
  end do
!$omp end parallel do

  stop
end
```

Data Dependence

Another issue pops up in the **STEPS** program. What happens when you call the function **f(x,y)** inside the loop?

Notice that **f** is not a variable, it's a function, so it is not declared private or shared.

The function might have internal variables, loops, might call other functions, and so on.

OpenMP works in such a way that a function called within a parallel loop will also participate in the parallel execution. We don't have to make any declarations about the function or its internal variables at all.

Each thread runs a separate copy of **f**.

(But if **f** *includes static or saved variables, trouble!)*

SUMMARY

I hope you have started to become familiar with some of the basic concepts of **OpenMP**.

We will come back to **OpenMP** in more detail for:

- a few more useful features of the language;
- some obstacles to parallelization;
- how to get or set the number of threads
- parallel sections that are not loops
- how to measure the improvement in performance.