Shared Memory Programming With OpenMP

ISC 5316: Applied Computational Science II

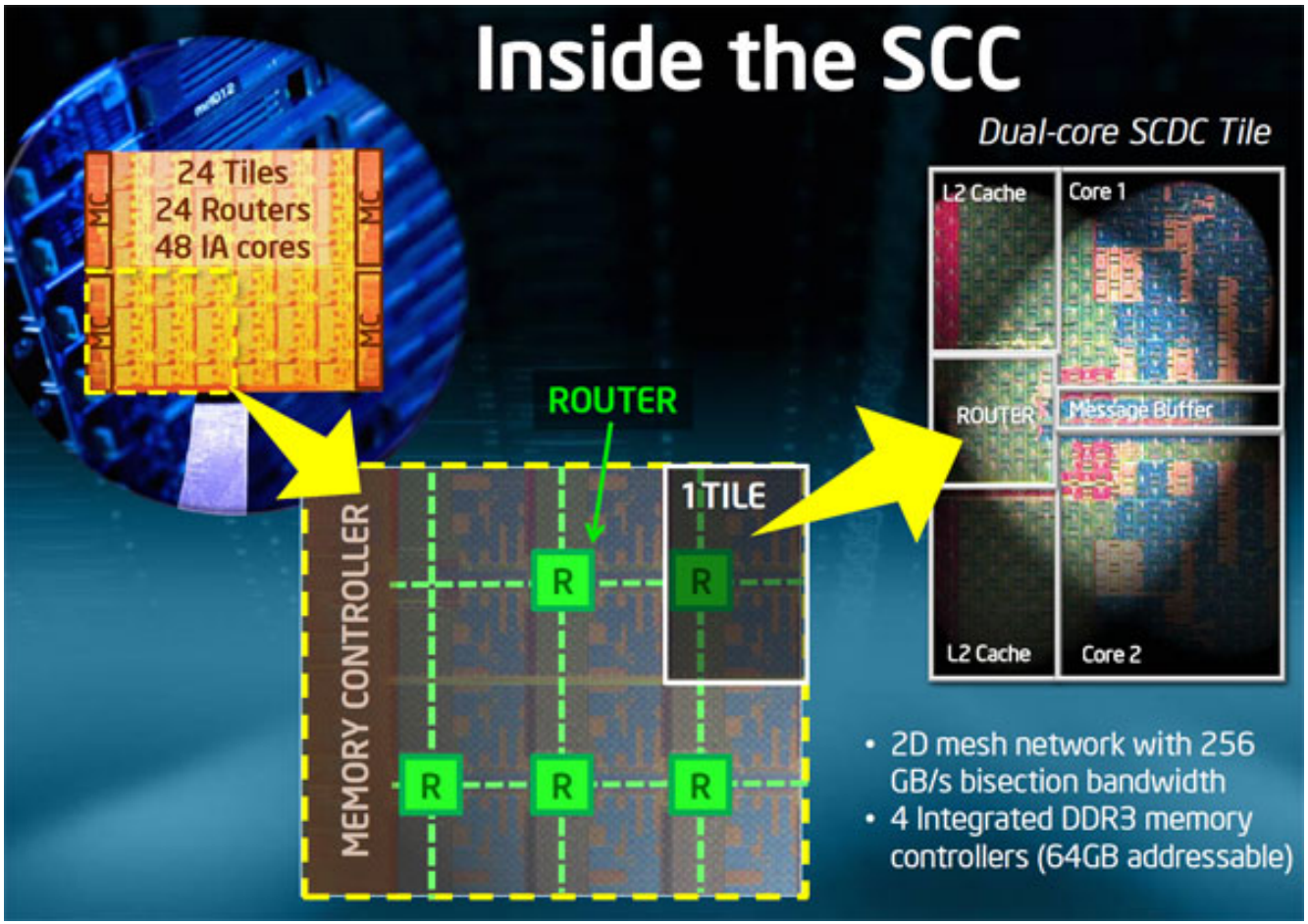
.....

John Burkardt
Department of Scientific Computing
Florida State University
https://people.sc.fsu.edu/~jburkardt/presentations/...
...openmp_2011_acs2.pdf

18 & 20 October 2011
Lab on 25 October 2011, due 01 November 2011

INTRO: OLD LANGUAGES=>OpenMP=>New Hardware

OpenMP is a bridge between yesterday's programming languages and tomorrow's multicore chips (4 cores when I first gave this talk, 48 cores now here at FSU!)



INTRO: Hardware Options

OpenMP runs a user program on shared memory systems:

- 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 (SGI system)
- multiple multicore chips using other schemes (Intel's Cluster OpenMP)

OpenMP, which you can think of as running on one tightly-coupled chip, can be combined with MPI, which runs on multiple, loosely-networked chips.

INTRO: OpenMP Limitations

OpenMP is limited by the shared memory hardware.

An OpenMP program can only handle problems that fit on the chip or the coupled chips, over which memory can be shared.

If an MPI program running on 5 chips needs more memory, it can easily be run on 100 chips, getting 20 times more memory.

An OpenMP program usually runs on a single chip, with a fixed amount of memory. If multiple chip OpenMP is available, the number of chips that can participate is limited, and there is a noticeable performance cost.

So **MPI** is better when big memory is needed.

INTRO: The OpenMP Idea

OpenMP assumes that a user program (already written) includes some **for** or **do** loops with the property that the iterations do not need to be carried out in sequence.

Assuming the iterations can actually be carried out simultaneously, OpenMP will automatically divide up the iterations among the available processors to get the work done faster.

We can imagine that each core in a processor is assigned some work. In fact, it is often the case that a single core can execute more than one independent set of calculations.

INTRO: Threads

To avoid thinking too much about hardware, OpenMP thinks of the user job as being divided into **threads** of execution.

Each thread is an independent but "obedient" entity. It has access to the shared memory. It has "private" space for its own 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 one **master thread** executing.

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

INTRO: 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 into **chunks** (that's really the term!); each chunk is assigned to a thread until all the work is done.

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

The helper threads go idle until the next parallel section.

INTRO: How OpenMP is Used

The user is responsible for indicating the loops to parallelize, and for indicating what variables in a loop must be handled in a special way so that the processors don't interfere with each other.

To indicate parallel loops, 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. It runs faster *(we hope!)*.

INTRO: Compiler Support

Compiler writers support OpenMP:

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

Mac users need Apple Developer Tools (CodeWarrior).

Shared Memory Programming with OpenMP

DIRECT: What Do Directives Look Like?

In C or C++, OpenMP 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.

DIRECT: The Parallel Region is Defined by a Directive

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

```
# pragma omp parallel
{
   do things in parallel here, if directed!
}
```

Inside the parallel region are **for** loops. The ones to be done in parallel are immediately preceded by an OpenMP **for** directive.

DIRECT: In C/C++, Brackets Delimit the Parallel Region

```
# pragma omp parallel
{  <-- use curly bracket to delimit the parallel region
    # pragma omp for
    for ( i = 0; i < n; i++ )
    {
      do things in parallel here.
    }

    for ( j = 1; j < 5; j++ )
    {
      This one is NOT parallel!
    }

    # pragma omp for
    for ( k = 99; k <= 1000; k++ )
    {
      This one parallel too.
    }
}  <-- End of parallel region
```

DIRECT: One-Block Parallel Region

If no brackets are used, the parallel region is the smallest block of code following the parallel directive.

For example, a single **for** or **do** loop constitutes a block of code, and need not be delimited:

```
# pragma omp parallel
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
    do things in parallel here
  }
```

DIRECT: The WAIT / NOWAIT Clauses

If a thread completes its portion of a loop, it ordinarily waits at the end of that loop for the other threads to complete. The **nowait** clause indicates that any thread finished with the current loop may safely start its part of the next loop immediately.

```
# pragma omp parallel
{
# pragma omp for nowait
  for ( j = 0; j < edges; j++ )
  {
    parallel loop 1
  }
# pragma omp for
  for ( k = 0; k < faces; k++ )
  {
    parallel loop 2
  }
}
```

DIRECT: Clauses

CLAUSES are additional information included on a directive.

The most common clause indicates which variables inside a parallel region must be treated as **private** or **shared** 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];
  }
}
```

DIRECT: 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];
  }
}
```

DIRECT: FORTRAN77

FORTRAN77 directives begin with the string **c$omp** and this string must begin in column 1!

```
c$omp parallel private ( i, j )
```

Directives longer than 72 characters **must** continue on a new line.

The continuation line also begins with the **c$omp** marker **AND** a continuation character in column 6, such as **&**.

These weird rules are inherited from the miserable rules in FORTRAN77 about where you can put things and where you can't. (They do not apply to FORTRAN90 and later.)

DIRECT: FORTRAN77 Example

```
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          <-- Must terminate the do area

c$omp end parallel  <-- Must terminate parallel area
```

DIRECT: FORTRAN90

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

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

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

The continued line must also be "commented out" with the **!$omp** marker.

DIRECT: FORTRAN90 Example

```
!$omp parallel &
!$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
```

DIRECT: What Do Directives Do?

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

SECTIONS: Tasks That Can Be Done Simultaneously

The easiest kind of parallelism to understand involves a set of jobs which can be done in any order.

Often, the number of tasks is small (2 to 5, say), and known in advance. It's possible that each task, by itself, is not suitable for processing by multiple threads.

We may try to speed up the computation by working on all the tasks at the same time, assigning one thread to each.

SECTIONS: Syntax for C/C++

```
# pragma omp parallel                    <-- inside "parallel"
{
  # pragma omp sections nowait <--optional "nowait" allows
  {                                        a fast thread to move on.
    # pragma omp section
    {
      code for section 1
    }
    # pragma omp section
    {
      code for section 2
    }
    ...   <-- more parts allowed in section
  }       <-- section ends.
  ...   <-- more work allowed in parallel region
}
```

SECTIONS: Syntax for FORTRAN90

```
!$omp parallel                    <-- inside "parallel"
  ...                            <-- optional initial work
  !$omp sections nowait <-- optional nowait
    !$omp section
      code for section 1
    !$omp section
      code for section 2
    ...                          <-- more parts allowed in section
  !$omp end sections
  ...                 <-- more work allowed in parallel region
!$omp end parallel
```

SECTIONS: How Sections are Executed

Each section will be executed by one thread.

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

Any extra threads will be idle.

The end of the sections block is a barrier, or synchronization point. Idle threads, and threads which have completed their sections, wait here for the others.

If the **nowait** clause is added to the **sections** directive, then idle and finished threads move on to the next piece of parallel work (if any) in the parallel region.

SECTIONS: Example

Notice that, if the program is executed sequentially, (ignoring the directives), then the sections will simply be computed one at a time, in the given order.

A Fast Fourier Transform program needs to compute two tables, containing the sines and cosines of angles. Sections could be used if at least two threads are available:

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

LOOPS: FOR and DO

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.

If a parallel region is just one loop, a combined directive is allowed, the **parallel do** or **parallel for** directive. *(I find this confusing, and try not to use it!)*

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

LOOPS: How OpenMP Parallelizes Loops

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

It's as though 2, 4, 8 or more programs are running at the same time, each executing a different subset of the iterations.

If you have **nested loops**, the order can be significant! OpenMP splits up the loop that immediately follows the directive **# pragma omp for**.

If you can write a pair of loops either way, you want to make sure the outer loop has a sizable iteration count! The following example will not take advantage of more than 3 threads!

```
# pragma omp for
  for ( i = 0; i < 3; i++ )
    for ( j = 0; j < 100000; j++ )
      a[i][j] = x[i] * sin ( pi * y[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 also 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!

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.

A **SAXPY** computation adds a multiple of one vector to another. Each iteration is

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

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

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

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).

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
# pragma omp 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 **# pragma omp parallel** directive opens a parallel region.

The **# pragma omp for** directive marks that loop as something inside the parallel region that can actually be done in parallel.

In this example, the parallel region terminates at the closing brace of the **for** loop block. We could use curly brackets to define a larger parallel region.

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$omp** (F77) or **!$omp** (F90).

The **!$omp parallel** directive opens a parallel region, which must be explicitly closed with a corresponding **!$omp end parallel**.

Within the parallel region, the **!$omp do** directive marks the beginning of a loop to be executed in parallel. The end of the loop must be marked with a corresponding **!$omp end do**.

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

LOOPS: QUIZ: 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.

CRITICAL: Critical Regions and Reductions

Critical regions of a code contain operations that should not be performed by more than one thread at a time.

A common cause of critical regions occurs when several threads want to modify the same variable, perhaps in a summation:

```
total = total + x[i]
```

To see what a critical region looks like, let's consider the following program, which computes the maximum entry of a vector.

CRITICAL: Sequential Version of Vector Sum

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

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

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

  total = 0.0;
  for ( i = 0; i < n; i++ )
  {
    total = total + x[i];
  }
  cout << "Sum = " << total << "\n";
  return 0;
}
```

CRITICAL: Making an OpenMP Version

To create an OpenMP version might seem easy:

- add the statement # **include** <**omp.h**>
- add the directive # **pragma omp parallel** before the **for** loop
- then add the directive # **pragma omp for** before the **for** loop
- compile, say with **g++ -fopenmp vector_sum.cpp**

But the results of this program are likely to be incorrect. The problem arises because we are asking several threads to work on a single variable, reading and writing it all at the same time.

CRITICAL: OpenMP Version 1

```
# include <cstdlib>
# include <iostream>
# include <omp.h>
using namespace std;

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

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

  total = 0.0;
# pragma omp parallel
# pragma omp for
  for ( i = 0; i < n; i++ )
  {
    total = total + x[i];
  }
  cout << "Sum = " << total << "\n";
  return 0;
}
```

CRITICAL: Synchronization Problems

The problem is one of **synchronization**. Because more than one thread is reading and writing the same data, it is possible for information to be mishandled.

When OpenMP uses threads to execute the iterations of a loop:

- the statements in a particular iteration of the loop will be carried out by one thread, in the given order
- but the statements in different iterations, carried out by different threads, may be "interleaved" arbitrarily.

CRITICAL: TOTAL Becomes Ambiguous

The processors must work on local copies of data.

```
P0: read TOTAL, X1
                        P1: read TOTAL, X2

P0: local TOTAL = TOTAL + X1
P0: write TOTAL
                        P1: local TOTAL = TOTAL + X2
                        P1: write TOTAL
```

If X = [10,20], what is TOTAL at the end?

CRITICAL: The TOTAL Update is Critical

As soon as processor 0 reads **TOTAL** into its local memory, no other processor should try to read or write **TOTAL** until processor 0 is done.

The update of **TOTAL** is called a **critical** region.

The OpenMP **critical** clause allows us to indicate that even though we are inside a **parallel** section, the critical code may only be performed by one thread at a time.

Fortran codes also need to use an **end critical** directive. C/C++ codes simply use curly braces to delimit the critical region.

CRITICAL: OpenMP Version 2

```
# include <cstdlib>
# include <iostream>
# include <omp.h>
using namespace std;

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

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

  total = 0.0;
  # pragma omp parallel
  # pragma omp for
  for ( i = 0; i < n; i++ )
  {
    # pragma omp critical
    {
      total = total + x[i];
    }
  }
  cout << "Sum = " << total << "\n";
  return 0;
}
```

CRITICAL: Correct...but Slower!

This is code is correct, and it uses OpenMP.

However, it runs no faster than sequential code! That's because our critical region is the entire loop. So one processor adds a value, than waits. The other processor adds a value and waits. Nothing really happens in parallel!

Here's a better solution. Each processor keeps its own local total, and we only have to combine these at the end.

CRITICAL: OpenMP Version 3

```
# include <cstdlib>
# include <iostream>
# include <omp.h>
using namespace std;
int main ( int argc, char *argv[] )
{
  int i, id, n = 1000;
  double total, total_local, x[1000];

  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
  }
  total = 0.0;
  # pragma omp parallel private ( id, total_local )
  {
    id = omp_get_thread_num ( );
    total_local = 0.0;
    # pragma omp for
    for ( i = 0; i < n; i++ )
    {
      total_local = total_local + x[i];
    }
    # pragma omp critical
    {
      total = total + total_local;
    }
  }
  cout << "Sum = " << total << "\n";
  return 0;
}
```

CRITICAL: A Little Mysterious

The process here is subtle. Inside the parallel region, there are multiple variables called **total_local**. Each thread is going to be assigned a subset of the vector entries to add up. Each thread computes its own sum in a variable called **total_local**. Presumably, this task takes the most time.

Once the thread finishes its part of the addition, it wanders towards the **critical** region. If no other thread is busy messing with total, this thread can add its contribution. Otherwise, it waits just before the critical region until it is free.

It's something like using a bathroom on an airplane!

CRITICAL: Success

This code is correct, and efficient.

I've had to jump ahead and include some OpenMP clauses and function calls you won't recognize yet.

Can you see where and why the **nowait** clause might be useful?

However, without understanding the details, it is not hard to see that the **critical** clause allows us to control the modification of the **TOTAL** variable, and that the **private** clause allows each thread to work on its own partial sum until needed.

CRITICAL: Reduction Clause

Simple operations like summations and maximums, which require a critical section, come up so often that OpenMP offers a way to hide the details of handling the critical section.

OpenMP offers the **reduction** clause for handling these special examples of critical section programming.

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

The variable summing the individual products is going to cause conflicts - delays when several threads try to read its current value, or errors, if several threads try to write updated versions of the value.

CRITICAL: Sequential dot product

```
# 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;
}
```

CRITICAL: Reduction Examples

The vector 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.

The OpenMP **reduction** clause allows the compiler to set up the reduction correctly and efficiently.

The **reduction** clause is included with the **for** or **do** directive associated with the loop where the reduction occurs.

CRITICAL: OpenMP Version

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

CRITICAL: The reduction clause

Variables which are the result of a reduction operator must be identified in a **reduction** clause of the **for** or **do** directive.

Reduction clause examples include:

- **# omp for reduction (+ : xdoty)** : we just saw this;
- **# omp for reduction (+ : sum1, sum2, sum3)** : several sums in one loop;
- **# omp for reduction (* : factorial)**: a product;
- **!$omp do reduction (max : pivot)** : maximum or minimum value (Fortran only);)

CONFLICTS: "Shared" Data

Shared data is data that can be safely shared by threads during a particular parallel operation, without leading to conflicts or errors.

By default, OpenMP will assume all data is shared.

A variable that is only "read" can obviously be shared. (Although in some cases, delays might occur if several threads want to read it at the same time).

Some variables may be shared even though they seem to be written by multiple threads;

An example is an array **A**.
If entry **A[I]** is never changed except during loop iteration **I**, then the array **A** can probably be shared.

CONFLICTS: "Private" Data

Private data is information each thread keeps separately.

A single variable name now refers to all the copies.

Simple examples:

- the iteration index of the loop, **i**
- temporary variables

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

A temporary variable **x_inv**, defined by
x_inv = 1.0 / x[i]
would also have to be private, even though **x** would not be.

CONFLICTS: PRIME SUM Example

The **PRIME SUM** program illustrates 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.

CONFLICTS: Sequential Version of Prime Sum

```
# 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;
}
```

CONFLICTS: Handling Conflicts!

Data conflicts will occur in **PRIME SUM** if all the data is shared during a parallel execution. We can't share a variable if two threads want to put different numbers into it.

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.

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

CONFLICTS: OpenMP Version

```
# 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 private ( i, prime, j ) shared ( n )

  # pragma omp for 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;
}
```

CONFLICTS: Controlling Data Access

The **shared**, **private** and **reduction** clauses allow us to specify how every variable is to be treated in the following loop.

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

By default, **private** variables have no value before or after the loop - they are purely temporary quantities.

If you need to initialize your private variables, or need to save the value stored by the very last iteration of the loop, OpenMP offers the **firstprivate** and **lastprivate** clauses.

CONFLICTS: Data Dependence

Data Dependence is an obstacle to parallel execution. Sometimes it can be repaired, and sometimes it is unavoidable.

In a loop, the problem arises when the value of a variable depends on results from a previous iteration of the loop.

Examples where this problem occurs include the solution of a differential equation or the application of Newton's method to a nonlinear equation.

In both examples, each step of the approximation requires the result of the previous approximation to proceed.

CONFLICTS: A Dependent Calculation

For example, suppose we computed factorials this way:

```
fact[0] = 1;
for ( i = 1; i < n; i++ )
{
  fact[i] = fact[i-1] * i;
}
```

We can't let OpenMP handle this calculation. The way we've written it, the iterations must be computed sequentially.

The variable on the right hand side, **fact[i-1]**,
is not guaranteed to be ready,
unless the previous iteration has completed.

CONFLICTS: The STEPS Program

The **STEPS** program illustrates an example of data dependence. Here, 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.

CONFLICTS: 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
```

CONFLICTS: We Can Fix this Dependence

In this example, the data dependence is simply a consequence of a common programming pattern. It's not hard to avoid the dependence once we recognize it.

Our options include:

- 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.

CONFLICTS: OpenMP Version

```
program main

  use omp_lib

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

  total = 0.0
!$omp parallel private ( i, j, x, y ) shared ( m, n )
  !$omp do 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
  !$omp end do
!$omp end parallel

  stop
end
```

CONFLICTS: Calling a Function

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!)*

RUN: From Source Code to Executable

Strictly speaking, compilation takes one or more source code files, say **myprog.c**, and translates them into "object" files, **myprog.o**.

Compiler errors have to do with syntax, (as well as "include" files that can't be found).

Linking joins the object files, along with compiled libraries to create an executable file **a.out**.

Linking errors have to do with calls to functions or routines that cannot be found.

RUN: Compiling

A typical compile-only command is:

```
cc -c myprog.c
cc -c graphics.c
```

Typical link-only commands include:

```
cc myprog.o graphics.o -lnag
```

A one-shot compile-and-link command would be

```
cc myprog.c graphics.c -lnag
```

RUN: Linking and Loading

Linking brings together your compile program, system libraries and external libraries. Once all the information is available, the loader can create a single executable program. Typically, this will be called **a.out**. It's best to rename the executable to something meaningful:

```
cc myprog.c graphics.c -lnag
mv a.out myprog
```

or you can do *everything* in one shot:

```
cc -o myprog myprog.c graphics.c -lnag
```

The executable program can be run by typing its (full) name. But the current directory is symbolized by "dot", so you can type

```
./myprog
```

RUN: Compiling an OpenMP Program

When you compile an OpenMP source code file, some errors may be reported because

- the compiler you invoked doesn't actually support OpenMP;
- the compiler does support OpenMP, but you didn't include the appropriate compiler switch.

So if your program doesn't compile, that doesn't necessarily mean you made a *programming* error.

You might be seeing complaints about a missing include file, or unrecognized function names.

RUN: Directives are Invisible by Default

On the other hand, if you didn't call any OpenMP functions, then any compiler will compile your code...because all the directives look like comments.

So if your program does compile, that doesn't necessarily mean you're actually using an OpenMP compiler!

RUN: Activate the Directives with a Switch

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

GNU compilers need the **fopenmp** switch:

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

RUN: Intel Compiler Switches

Intel C compilers need the **openmp** and **parallel** switches:

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

Intel Fortran compilers also require the **fpp** switch:

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

RUN: Executing the Program

Once you have an executable OpenMP program, you can run it interactively, the same way you would any executable program.

```
./a.out
```

The only thing you need to do is make sure that you have defined the number of threads of execution - that is, "how parallel" you want to be.

RUN: Specifying the Number of Threads

OpenMP looks for the value of an environment variable called **OMP_NUM_THREADS** to determine the default number of threads. You can query this value by typing:

```
echo $OMP_NUM_THREADS
```

A blank value means it hasn't been defined, so it's essentially 1.

You can redefine this environment variable using the command

```
export OMP_NUM_THREADS=4
```

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

This value only represents the default. That means your program can use the default, or request a specific number of threads by making a function call internally.

RUN: Trying Different Numbers of THreads

Suppose you want to compile, link and run on the lab computer?

We can presume we've already got the program file, compiled it and linked it, and called it...let's say we called it **md**.

If we are simply going to run the program on the command line, changes we make to **OMP_NUM_THREADS** will affect the program. We could experiment with 1, 2, 4 and 8 threads by:

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

RUN: Using a Shell Script

It is often convenient to put a group of useful commands into a *shell script*, which we might call *run_md.sh*:

```
#!/bin/bash
  export OMP_NUM_THREADS=1
  ./md
  export OMP_NUM_THREADS=2
  ...
  ./md
```

The first line indicates that this is a BASH shell script.

We execute the shell script by "feeding it" to BASH:

```
bash run_md.sh
```

which has the disadvantage that we must wait until the script has finished before we get a command prompt again.

RUN: Running in the Background

If we expect **md** to take a long time to run, we could instead run the shell script "in the background", using an ampersand, in which case we get our command prompt back immediately.

```
bash run_md.sh &
```

but probably we want the program output to go to a file then, so we can be typing new commands without interference:

```
bash run_md.sh > output.txt &
```

RUN: Running in the Background with NOHUP

If we expect **md** to run a really really long time, then it's likely we'll want to log out before it finishes. Normally, that would kill the job, even though it's running in the background.

However, we can specify that this job runs in the background, writes its output to a file, and continues to run even if we log out, by using the **nohup** command:

```
nohup bash run_md.sh > output.txt &
```

What I have told you is just an outline of what is possible if you want to run long jobs on a system. Naturally, if you try these operations, you are going to want to learn more about how to monitor or kill a background job!

RUN: Running on a Cluster

If you have a really big job, or lots of them, or you need a lot of parallel threads, you are not going to want to mess around on our lab system. Instead, you want to try to run your program on a cluster, such as the FSU HPC system.

Obviously, you need to use **sftp** to transfer files to the system, and **ssh** to log in. The environment there is similar, so we can compile our program in the same way. But running the program is quite different. Now we are going to use a sophisticated queueing system which will handle our job.

To use the queue, we must prepare a script file, which might be called *md_batch.sh*.

RUN: HPC Batch Script

```
#!/bin/bash                          << Job uses BASH shell

#MOAB -N md                          << Job name is "MD"
#MOAB -q classroom                   << Run job in this queue
#MOAB -l nodes=1:ppn=8               << Want 1 node, 8 processors.
#MOAB -l walltime=00:02:00           << Request 2 minutes of time.
#MOAB -j oe                          << Join output and error files

cd $PBS_O_WORKDIR                    << Move to directory

export OMP_NUM_THREADS=8             << Number of threads <= PPN
./md > output.txt                    << Finally!
```

RUN: Using the HPC Batch Script

Briefly, once you have a compiled version of **md** on the HPC, and your batch script file *md_batch.sh*, you "submit" the job with the command

```
msub md_batch.sh
```

and then you wait for the job to complete. You can check the job's program using the command

```
showq
```

and, the way this script was written, the interesting results will show up in the file *output.txt*.

We will go over the details of HPC job execution during the lab.

ENVIRON: Customizing the OpenMP Environment

We have already run across the mysterious variable **OMP_NUM_THREADS**.

I told you it specifies how many parallel threads of execution there will be. You can set this variable externally, in the Unix environment, or you can fiddle with it inside the program as it runs.

This variable is one example of a set of OpenMP environment variables.

It's time to take a look at these variables, how they influence the way your program is run, and how your program can access and modify their values.

ENVIRON: Environment Variables and Function Calls

OpenMP uses internal data which can be of use or interest.

In a few cases, the user can set some of these values by means of a Unix environmental variable.

There are also functions the user may call to get or set this information.

ENVIRON: Information You Can Set or Get

You can **set**:

- maximum number of threads - most useful!
- details of how to handle loops, nesting, and so on

You can **get**:

- number of "processors" (=cores) are available
- individual thread id's
- maximum number of threads
- wall clock time

ENVIRON: Variables

If you are working on a UNIX system, you can "talk" to OpenMP by setting certain environment variables.

The syntax for setting such variables varies slightly, depending on the shell you are using.

Many people use this method in order to specify the number of threads to be used. If you don't set this variable, your program runs on one thread.

ENVIRON: Variables

There are just 4 OpenMP environment variables:

- **OMP_NUM_THREADS**, maximum number of threads
- **OMP_DYNAMIC**, allows dynamic thread adjustment
- **OMP_NESTED**, allows nested parallelism, default 0/FALSE
- **OMP_SCHEDULE**, determines how loop work is divided up

ENVIRON: Variables

Determine your shell by:

echo $SHELL

Set the number of threads in the Bourne, Korn and BASH shells:

export OMP_NUM_THREADS=4

In the C or T shells, use a command like

setenv OMP_NUM_THREADS 4

To verify:

echo $OMP_NUM_THREADS

ENVIRON: Functions

OpenMP environment functions include:

- **omp_set_num_threads (t)**
- **t = omp_get_num_threads ( )**
- **p = omp_get_num_procs ( )**
- **id = omp_get_thread_num ( )**
- **wtime = omp_get_wtime()**

ENVIRON: How Many Threads May I Use?

A **thread** is one of the "workers" that OpenMP assigns to help do your work.

There is a limit of

- 1 thread in the sequential sections.
- **OMP_NUM_THREADS** threads in the parallel sections.

ENVIRON: How Many Threads May I Use?

The number of threads

- has a default for your computer.
- can be initialized by setting **OMP_NUM_THREADS** before execution
- can be reset by calling **omp_set_num_threads(t)**
- can be checked by calling **t=omp_get_num_threads()**

ENVIRON: How Many Threads Should I Use?

If **OMP_NUM_THREADS** is 1, then you get no parallel speed up at all, and probably actually slow down.

You can set **OMP_NUM_THREADS** much higher than the number of processors; some threads will then "share" a processor.

Reasonable: one thread per processor.

```
p = omp_get_num_procs ( );
t = p;
omp_set_num_threads ( t );
```

These three commands can be compressed into one.

ENVIRON: Which Thread Am I Using?

In any parallel section, you can ask each thread to identify itself, and assign it tasks based on its index.

```
!$omp parallel
  id = omp_get_thread_num ( )
  write ( *, * ) 'Thread ', id, ' is running.'
!$omp end parallel
```

ENVIRON: How Much Time Has Passed?

You can take "readings" of the wall clock time before and after a parallel computation.

```
wtime = omp_get_wtime ( );
# pragma omp parallel
# pragma omp for
for ( i = 0; i < n; i++ )
{
  Do a lot of work in parallel;
}
wtime = omp_get_wtime ( ) - wtime;

cout << "Work took " << wtime << " seconds.\n";
```

ENVIRON: "Hiding" Parallel Code

OpenMP tries to make it possible for you to have your sequential code and parallelize it too. In other words, a single program file should be able to be run sequentially or in parallel, simply by turning on the directives.

This isn't going to work so well if we have these calls to **omp_get_wtime** or **omp_get_proc_num** running around. They will cause an error when the program is compiled and loaded sequentially, because the OpenMP library will not be available.

Fortunately, you can "comment out" all such calls, just as you do the directives, or, in C and C++, check whether the symbol **_OPENMP** is defined.

ENVIRON: Hiding Parallel Code in C++

```
# ifdef _OPENMP
  # include <omp.h>
   wtime = omp_get_wtime ( );
# endif

   # pragma omp parallel
   # pragma omp for
   for ( i = 0; i < n; i++ )
   {
     Do a lot of work, possibly in parallel;
   }

# ifdef _OPENMP
   wtime = omp_get_wtime ( ) - wtime;
   cout << "Work took " << wtime << " seconds.\n";
# else
   cout << "Elapsed time not measured.\n";
# endif
```

ENVIRON: Hiding Parallel Code in F90

```
!$ use omp_lib

!$ wtime = omp_get_wtime ( )

!$omp parallel
  !$omp do
  do i = 1, n
    Do a lot of work, possibly in parallel;
  end do
  !$omp end do
!$omp parallel

!$ wtime = omp_get_wtime ( ) - wtime
!$ write ( *, * ) 'Work took', wtime, ' seconds.'
```

CONTROL: C Loops

```
# pragma omp parallel
  # pragma omp for
  for ( i = ilo; i <= ihi; i++ )
  {
    C/C++ code to be performed in parallel
  }
```

```
!$omp parallel
  !$omp do
  do i = ilo, ihi
    FORTRAN code to be performed in parallel
  end do
  !$omp end do
!$omp end parallel
```

CONTROL: FORTRAN Loops

FORTRAN Loop Restrictions:

The loop must be a **do** loop of the form;

```
do i = low, high (, increment)
```

The limits **low**, **high** (and **increment** if used), cannot change during the iteration.

The program cannot jump out of the loop, using an **exit** or **goto**.

The loop cannot be a **do while**.

The loop cannot be an "infinite" **do** (no iteration limits).

CONTROL: Loops

C Loop Restrictions:

The loop must be a **for** loop of the form:

```
for ( i = low; i < high; increment )
```

The limits **low** and **high** cannot change during the iteration;

The **increment** (or decrement) must be by a fixed amount.

The program cannot **break** from the loop.

CONTROL: No Loop

It is possible to set up parallel work without a loop.

In this case, the user can assign work based on the ID of each thread.

For instance, if the computation models a crystallization process over time, then at each time step, half the threads might work on updating the solid part, half the liquid.

If the size of the solid region increases greatly, the proportion of threads assigned to it could be increased.

CONTROL: No Loop, C/C++

```
# pragma omp parallel
{
  id = omp_get_thread_num ( );
  if ( id % 2 == 0 )
  {
    solid_update ( );
  }
  else
  {
    liquid_update ( );
  }
}
```

CONTROL: No Loop, FORTRAN

```
!$omp parallel
  id = omp_get_thread_num ( )
  if ( mod ( id, 2 ) == 0 ) then
    call solid_update ( )
  else if ( mod ( id, 4 ) == 1 ) then
    call liquid_update ( )
  else if ( mod ( id, 4 ) == 3 ) then
    call gas_update ( )
  end if
!$omp end parallel
```

(Now we've added a gas update task as well.)

CONTROL: WORKSHARE

FORTRAN90 expresses implicit vector operations using colon notation.

OpenMP includes the **WORKSHARE** directive, which says that the marked code is to be performed in parallel.

The directive can also be used to parallelize the FORTRAN90 **WHERE** and the FORTRAN95 **FORALL** statements.

Unfortunately, I have not yet found any FORTRAN compiler that has implemented the WORKSHARE directive!

CONTROL: FORTRAN90

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

```
!$omp parallel
  !$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
```

CONTROL: FORTRAN95

```
!$omp parallel
  !$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)

CONTROL: Parallel Computing Without Loops

OpenMP is easiest to use with loops.

Here is an example where we get parallel execution without using loops.

Doing the problem this way will make OpenMP seem like a small scale version of **MPI**.

CONTROL: Problem specification

What values of **X** make **F(X)** evaluate **TRUE**?

```
f(x) = (  x(1)  ||  x(2)  ) && ( !x(2)  || !x(4)  ) &&
       (  x(3)  ||  x(4)  ) && ( !x(4)  || !x(5)  ) &&
       (  x(5)  || !x(6)  ) && (  x(6)  || !x(7)  ) &&
       (  x(6)  ||  x(7)  ) && (  x(7)  || !x(16) ) &&
       (  x(8)  || !x(9)  ) && ( !x(8)  || !x(14) ) &&
       (  x(9)  ||  x(10) ) && (  x(9)  || !x(10) ) &&
       ( !x(10) || !x(11) ) && (  x(10) ||  x(12) ) &&
       (  x(11) ||  x(12) ) && (  x(13) ||  x(14) ) &&
       (  x(14) || !x(15) ) && (  x(15) ||  x(16) )
```

CONTROL: Problem specification

Sadly, there is no clever way to solve a problem like this in general. You simply try every possible input.

How do we generate all the inputs?

Can we divide the work among multiple processors?

CONTROL: Algorithm Design

There are $2^{16} = 65,536$ distinct input vectors.

There is a natural correspondence between the input vectors and the integers from 0 to 65535.

We can divide the range [0,65536] into **T_NUM** distinct (probably unequal) subranges.

Each thread can generate its input vectors one at a time, evaluate the function, and print any successes.

CONTROL: Program Design

```
#pragma omp parallel
{
  T = omp_get_num_threads ( );
  ID = omp_get_thread_num ( );
  ILO = ( ID     ) * 65535 / T;
  IHI = ( ID + 1 ) * 65535 / T;

  for ( I = ILO; I < IHI; I++ )
  {
    X[0:15] <= I              (set binary input)
    VALUE = F ( X )           (evaluate function)
    if ( VALUE ) print X
  end
}
```

CONTROL: FORTRAN90 Implementation

```
  thread_num = omp_get_num_threads ( )
  solution_num = 0
!$omp parallel private ( i, ilo, ihi, j, value, x ) &
!$omp shared ( n, thread_num ) &
!$omp reduction ( + : solution_num )
  id = omp_get_thread_num ( )
  ilo = id * 65536 / thread_num
  ihi = ( id + 1 ) * 65536 / thread_num

  j = ilo
  do i = n, 1, -1
    x(i) = mod ( j, 2 )
    j = j / 2
  end do

  do i = ilo, ihi - 1
    value = circuit_value ( n, x )
    if ( value == 1 ) then
      solution_num = solution_num + 1
      write ( *, '(2x,i2,2x,i10,3x,16i2)' )  solution_num, i - 1, x(1:n)
    end if
    call bvec_next ( n, x )
  end do
!$omp end parallel
```

CONTROL: Observations

I wanted an example where parallelism didn't require a **for** or **do** loop. The loop you see is carried out entirely by one (each) thread.

The "implicit loop" occurs when when we begin the parallel section and we generate all the threads.

The idea to take from this example is that the environment functions allow you to set up your own parallel structures in cases where loops aren't appropriate.

DATA: Private/Shared

The very name "shared memory" suggests that the threads share one set of data that they can all "touch".

By default, OpenMP assumes that all variables are to be shared – with the exception of the loop index in the **do** or **for** statement.

It's obvious why each thread will need its own copy of the loop index. Even a compiler can see that!

However, some other variables may need to be treated specially when running in parallel. In that case, you must explicitly tell the compiler to set these aside as **private** variables.

It's a good practice to declare all variables in a loop.

DATA: Private/Shared

```
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
```

DATA: Private/Shared

I've had to cut this example down a bit. So let me point out that **coord** and **f** are big arrays of spatial coordinates and forces, and that **f** has been initialized already.

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

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

List all the variables in this loop, and try to determine if they are **shared** or **private** or perhaps a **reduction** variable.

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

DATA: 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
```

DATA: Private/Shared

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

  !$ 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: 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.

DATA: Private/Shared/Reduction

Suppose in FORTRAN90 we need the maximum of a vector.

```
x_max = - huge ( x_max )              ---+
do i = 1, n                             |
  x_max = max ( x_max, x(i) )           |  Loop #1
end do                                  |
                                      ---+

x_max = maxval ( x(1:n) )             --->  Loop #2
```

How could we parallelize loop #1 or loop #2!

DATA: Private/Shared/Reduction

In loop 1, the reduction variable **x_max** will automatically be initialized to the minimum real number.

```
!$omp parallel private ( i ) shared ( n, x )

  !$ omp do reduction ( max : x_max )
  do i = 1, n
    x_max = max ( x_max, x(i) )
  end do
  !$ omp end do
!$omp end parallel

!$omp parallel
  !$ omp workshare
  x_max = maxval ( x(1:n) )
  !$ omp end workshare
!$omp end parallel
```

DATA: DEFINE'd Variables in C/C++

In C and C++, it is common to use a **#define** statement. This can look almost like a declaration statement, with a variable name and its value. It is actually a preprocessor directive, and the "variable" is really a text string to be replaced by the given value.

By convention, defined variables are CAPITALIZED.

A typical defined variable is actually a constant, that is, a number. And this means that even though it may look like a variable, it is not appropriate nor necessary to include a defined variable in the **shared** or **private** clauses of an OpenMP directive!

DATA: DEFINE'd Variables in C/C++

Do NOT put the defined variable **N** in the shared clause!

```
# define N 100

# pragma omp parallel shared ( x, y ) \
  private ( i, xinv )

  # pragma omp for
  for ( i = 0; i < N; i++ )
  {
    xinv = 1.0 / x[i];
    y[i] = y[i] * xinv;
  }
```

DATA: FORTRAN Parameters

In FORTRAN, it is common to use a **parameter** statement to define constants such as π or $\sqrt{2}$.

The important thing about a parameter is that, although it looks like a "variable", it is a **constant**. At least for some compilers, this means that it is neither appropriate nor necessary to include a FORTRAN parameter in the **shared** or **private** clauses of an OpenMP directive!

DATA: FORTRAN Parameters

Do NOT put the parameters **pi** or **n** in the shared clause!

```
  integer, parameter :: n = 100
  real, parameter :: pi = 3.14159265

!$omp parallel shared ( c, s ) private ( angle, i )
  !$omp do
  do i = 1, n
    angle = ( i - 1 ) * pi / n
    c(i) = cos ( angle )
    s(i) = sin ( angle )
  end do
  !$ omp end do
!$omp end parallel
```

EXAMPLES: The Index of the Maximum Entry

In Gauss elimination, the K-th step involves finding the row index **P** of the largest element on or below the diagonal in column K of the matrix.

What's important isn't the maximum value, but its index.

That means that we can't simply use OpenMP's **reduction** clause.

Let's simplify the problem a little, and ask:

Can we determine the index of the largest element of a vector in parallel?

EXAMPLES: The Index of the Maximum Entry

The **reduction** clause can be thought of as carrying out a critical section for us. Since there's no OpenMP reduction clause for *index of maximum value*, we'll have to do it ourselves.

We want to do this in such a way that, as much as possible, all the threads are kept busy.

We can let each thread find the maximum (and its index) on a subset of the vector, and then have a cleanup code (and *now* we use the critical section!) which just compares each thread's results, and takes the appropriate one.

EXAMPLES: The Index of the Maximum Entry

```
all_max = 1
!$omp parallel private ( i,id,i_max ) shared ( n,p_num,x )
  id = omp_get_thread_num ( );
  i_max = id + 1;
  do i = id + 1, n, p_num
    if ( x(i_max) < x(i) ) then
      i_max = i;
    end if
  end do
  !$omp critical
    if ( x(all_max) < x(i_max) ) then
      all_max = i_max
    end if
  !$omp end critical
!$omp end parallel
```

EXAMPLES: Random Numbers

Random numbers are a vital part of many algorithms. But you must be sure that your random number generator behaves properly.

It is acceptable (but hard to check) that your parallel random numbers are at least "similarly distributed."

It would be ideal if you could generate the same stream of random numbers whether in sequential or parallel mode.

EXAMPLES: Random Numbers

Most random number generators work by repeatedly "scrambling" an integer value called the seed. One kind of scrambling is the linear congruential generator:

```
SEED = ( A * SEED + B ) modulo C
```

If you want a real number returned, this is computed indirectly, as a function of the updated value of the SEED;

```
float my_random ( int *SEED )
  *SEED = ( A * *SEED + B ) modulo C
  R = ( double ) *SEED / 2147483647.0
  return R
```

EXAMPLES: Random Numbers

Many random number generators have you set the seed first:

```
seed = 123456789;
srand48 ( seed );
```

This value of **SEED** is stored somewhere in "static" memory, where the generator can get to it as needed.

When you call the random number generator, it gets a copy of the seed, updates it, writes the updated seed back to static memory, and then returns the random number you asked for:

```
x = drand48 ( );    <-- Hidden calculations
                        involve SEED.
```

EXAMPLES: Random Numbers

For typical random number calculations, **SEED** determines everything.

For parallel computations, it is dangerous to use an algorithm which has hidden variables that are stored statically.

It's important to test. Initialize SEED to 123456789, say, compute 20 values sequentially; repeat in parallel and compare.

Random number generators using hidden seeds may or may not work *correctly* in parallel.

They may work *inefficiently*, if multiple processors contend for access to a single shared seed.

EXAMPLES: Random Numbers

```
# include ...stuff...
int main ( void )
{
  int i;
  unsigned int seed = 123456789;
  double y[20];

  srand ( seed );
  for ( i = 0; i < 20; i++ )
  {
    y[i] = ( double ) rand ( ) / ( double ) RAND_MAX;
  }
  return 0;
}
```

EXAMPLES: Random Numbers

There are random number generators which use a seed value, but which have you pass the seed as an argument.

This means there is no internal memory in the random number generator to get confused when multiple processes are involved.

It allows you to assign separate (and different!) seeds to each thread, presumably resulting in distinct random sequences.

We can do this using a parallel section, setting a seed based on the thread ID.

EXAMPLES: Random Numbers

```
# pragma omp parallel private ( i, id, r, seed )
{
  id = omp_get_thread_num ( );
  seed = 123456789 * id
  for ( i = 0; i < 1000; i++ )
  {
    r = my_random ( seed );
    (do stuff with random number r )
  }
}
```

EXAMPLES: Random Numbers

For the MPI system of parallel programming, generating distinct sets of random numbers is also a big issue.

However, in that case, there is at least one well-tested package, called **SPRNG**, developed at FSU by Professor Michael Mascagni, which can generate distinct random numbers for multiple processes.

EXAMPLES: Carry Digits

Suppose vectors X and Y contain digits base B, and that Z is to hold the base B representation of their sum. (Let's assume for discussion that base B is 10).

Adding is easy. But then we have to carry. Every entry of Z that is B or greater has to have the excess subtracted off and carried to the next higher digit. This works in one pass of the loop only if we start at the lowest digit.

And adding 1 to 9,999,999,999 shows that a single carry operation could end up changing every digit we have.

EXAMPLES: Carry Digits

```
do i = 1, n
  z(i) = x(i) + y(i)
end do
overflow = .false.
do i = 1, n
  carry = z(i) / b
  z(i) = z(i) - carry * b
  if ( i < n ) then
    z(i+1) = z(i+1) + carry
  else
    overflow = .true.
  end if
end do
```

EXAMPLES: Carry Digits

In the carry loop, notice that on the I-th iteration, we might write (modify) both **z[i]** and **z[i+1]**.

In parallel execution, the value of **z[i]** used by iteration I might be read as 17, then iteration I-1, which is also executing, might change the 17 to 18 because of a carry, but then iteration I, still working with its temporary copy, might carry the 10, and return the 7, meaning that the carry from iteration I-1 was lost!

99% of carries in base 10 only affect at most two higher digits. So if we were desperate to use parallel processing, we could use repeated carrying in a loop, plus a temporary array **z2**.

EXAMPLES: Carry Digits

```
do
  !$omp parallel
    !$omp workshare
    z2(1)   = mod ( z(1)  , b )
    z2(2:n) = mod ( z(2:n), b ) + z(1:n-1) / b
    z(1:n)  = z2(1:n)
    done = all ( z(1:n-1) / b == 0 )
    !$omp end workshare
  !$omp end parallel

  if ( done )
    exit
  end if

end do
```

CONCLUSION: Clauses We Skipped

Although OpenMP is a relatively simple programming system, there is a lot we have not covered.

The **single** clause allows you to insist that only one thread will actually execute a block of code, while the others wait. (Useful for initialization, or print out).

The **schedule** clause, which allows you to override the default rules for how the work in a loop is divided.

There is a family of functions that allow you to use a **lock** variable instead of a **critical** clause. Locks are turned on and off by function calls, which can be made anywhere within the code.

CONCLUSION: Nested Parallelism

In nested parallelism, a parallel region contains smaller parallel regions. A thread coming to one of these nested regions can then fork into even more threads. Nested parallelism is only supported on some systems.

OpenMP has the environment variable **OMP_NESTED** to tell if nesting is supported, and functions to determine how nesting is to be handled.

CONCLUSION: Parallel Debugging

Debugging a parallel programming can be quite difficult.

If you are familiar with the Berkeley **dbx** or Gnu **gdb** debuggers, these have been extended to deal with multithreaded programs.

There is also a program called **TotalView** with an intuitive graphical interface.

However, I have a colleague who has worked in parallel programming for years, and who insists that he can always track down every problem by using **print** statements!

He's not as disorganized as that sounds. When debugging, he has each thread write a separate log file of what it's doing, and this gives him the evidence he needs.

CONCLUSION: Tuesday's Lab Exercises

Exercises for the laboratory session will introduce you to OpenMP.

You'll write a simple program to estimate an integral.

You will make OpenMP versions of FFT, molecular dynamics, and heat equation programs, using directives on just one or two loops.

You will investigate (a little) the speedup as you increase the number of processors, or make other changes in the codes.

CONCLUSION: Weekend Headache

For Tuesday, write a program, using OpenMP, that computes a matrix product, using three loops.

- Loop 1 sets A(I,J) = $\sin(\frac{2\pi(i-1)(j-1)}{n}) + \cos(\frac{2\pi(i-1)(j-1)}{n})$
- Loop 2 sets B(I,J) = A(I,J) / n
- Loop 3 sets C = A * B.

Note that I and J loop from 1 to N (FORTRAN indexing).
C/C++ users replace the factor $(i-1)(j-1)$ by ij.

Print the value of C(n,n) or C[n-1][n-1];
Print the elapsed time in seconds to carry out all three loops.
Run with n = 500 on 1, 2, 4 and 8 threads.

If you don't have it finished, we'll work on it in the lab!

CONCLUSION:

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/`