4: Shared Memory Programming With OpenMP

John Burkardt
Information Technology Department
Virginia Tech

....

HPPC-2008
High Performance Parallel Computing Bootcamp

....

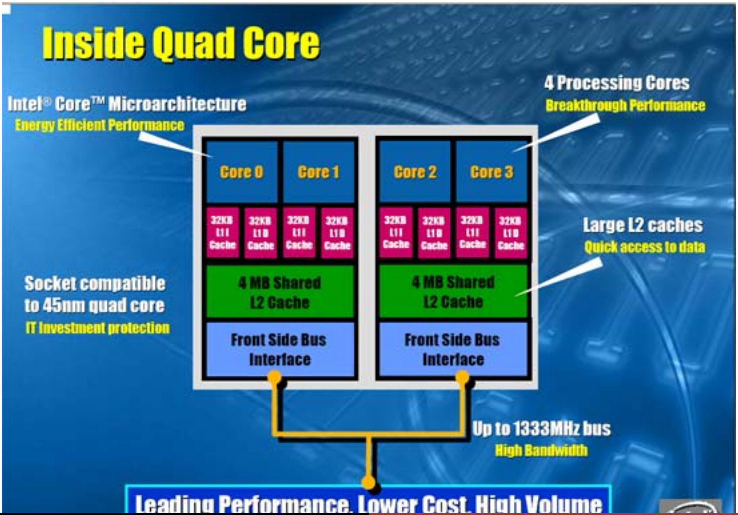
http://people.sc.fsu.edu/∼jburkardt/presentations/hppc_2008_lecture4.pdf

28 July - 02 August
2008

Shared Memory Programming with OpenMP

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 be combined with **MPI** if a distributed system is made up of multi-processor chips.

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 support OpenMP:

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

Mac users: Apple distributes old compilers. Get latest gcc from http://hpc.sourceforge.net/
You also need Apple Developer Tools (CodeWarrior).

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 C compilers, include the **openmp** and **parallel** switches:

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

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

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

Introduction: What Do Directives Look Like?

In C or C++, directives begin with the # preprocessor 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
```

One or more directives may appear, just before a block of code, which is typically 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
}
```

If the entire parallel region is a single **for** or **do** loop, or a single **sections** directive, the directives can be combined:

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

Introduction: What Do Directives Look Like?

There's overhead in starting up a parallel region. 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.

The most common clauses define a list of **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];
}
```

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 for \
  shared ( n, s, x, y ) \
  private ( i, t )

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**.

```
c$omp parallel do 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 **&**.

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

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

Introduction: What Do Directives Look Like?

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

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

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

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

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

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

Introduction: What Do Directives Do?

- indicate parallel sections of the code:
  # pragma omp parallel
- indicate code that only one thread can do at a time:
  # pragma omp critical
- suggest how the work is to be divided:
  # pragma omp parallel for schedule (dynamic)
- 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)
- force threads to wait til all are done:
  # pragma omp barrier

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 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.

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 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.

Shared Memory Programming with OpenMP

Sections

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
  {
    #pragma omp section
    {
      code for section 1
    }
    #pragma omp section
    {
      code for section 2
    }                           <-- more sections
  }                                 could follow
}
```

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 sections
                                 could follow

  !omp end sections
  ...                        <-- optional later work
!omp end parallel
```

Sections

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 immediately.

Sections

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 two threads are available:

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

Shared Memory Programming with OpenMP

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.

For convenience, there is a combined form, the **parallel do** or **parallel for** 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!

If you have **nested loops**, the order is significant! OpenMP splits up the outer loop, not the inner.

If you can write a pair of loops either way, you want to make sure the outer loop has a sizable iteration count!

```
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 its own 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, 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.

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 **parallel for** clause requests parallel execution of the following **for** loop.

The parallel section terminates at the closing brace 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: 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.

Shared Memory Programming with OpenMP

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.

VECTOR SUM: Sequential version

```
# 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 Regions and Reductions

To turn our program into an OpenMP program is easy:

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

But to turn our program into a **CORRECT** OpenMP program is not so easy!

This code cannot be guaranteed to run correctly on more than 1 processor!

VECTOR SUM: First OpenMP version

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

Critical Regions and Reductions

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 Regions and Reductions

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 Region and Reductions

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.

VECTOR SUM: Second OpenMP version

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

Critical Regions and Reductions

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.

VECTOR SUM: Third OpenMP version

```
# 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 Regions and Reductions

This code is correct, and efficient.

I've had to jump ahead and include some OpenMP clause 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 Regions and Reductions

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.

DOT PRODUCT: 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;
}
```

Critical Regions and Reductions: 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.

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

Critical Regions and Reductions: 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))

Shared Memory Programming with OpenMP

Data Conflicts and Data Dependence

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 only written during loop iteration **I**, then the array **A** can probably be shared.

Data Conflicts and Data Dependence

Private data is information each thread keeps separately.

A single variable name now refers to multiple copies for each thread.

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.

Data Conflicts and Data Dependence: PRIME SUM

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.

PRIME SUM: 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;
}
```

Data Conflicts and Data Dependence: 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.

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 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;
 }
```

Data Conflicts and Data Dependence

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!

The default treatment of **private** variables is that they have no value before or after the loop - they are purely temporary quantities.

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

Data Conflicts and 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.

Data Conflicts and Data Dependence

Suppose, for instance, we computed a table of 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.

Data Conflicts and Data Dependence

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.

The STEPS program: 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 Conflicts and Data 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.

The STEPS program: 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 Conflicts and 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!)*

Shared Memory Programming with OpenMP

OpenMP Environment

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.

OpenMP Environment

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

OpenMP Environment: 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.

OpenMP Environment: 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

OpenMP Environment: 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
```

OpenMP Environment: Functions

OpenMP environment 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 Environment: 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.
- **T_NUM** threads in the parallel sections.

OpenMP Environment: How Many Threads May I Use?

T_NUM

- 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_num)**
- can be checked by calling **t_num=omp_get_num_threads()**

OpenMP Environment: How Many Threads Should I Use?

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

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

Reasonable: one thread per processor.

```
  p_num = omp_get_num_procs ( );
  t_num = p_num;
  omp_set_num_threads ( t_num );
```

These three commands can be compressed into one.

OpenMP Environment: 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.

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

OpenMP Environment: 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 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";
```

OpenMP Environment: "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.

OpenMP Environment: Hiding Parallel Code in C++

```
# ifdef _OPENMP
  # include <omp.h>
# endif
# ifdef _OPENMP
   wtime = omp_get_wtime ( );
# endif
#pragma omp parallel for
   for ( i = 0; i < n; i++ ){
     Do a lot of work in parallel;  }
# ifdef _OPENMP
   wtime = omp_get_wtime ( ) - wtime;
   cout << "Work took " << wtime << " seconds.\n";
# else
   cout << "Elapsed time not measured.\n";
# endif
```

OpenMP Environment: Hiding Parallel Code in FORTRAN90

```
!$omp use omp_lib

!$omp wtime = omp_get_wtime ( )
!$omp parallel do
  do i = 1, n
      Do a lot of work in parallel;
  end do
!$omp end parallel do
!$omp wtime = omp_get_wtime ( ) - wtime
!$omp write ( *, * ) 'Work took', wtime, ' seconds.'
```

Shared Memory Programming with OpenMP

Parallel Control Structures, Loops

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

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

Parallel Control Structure, 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**, and it cannot be a **do** with no iteration limits.

Parallel Control Structure, 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.

Parallel Control Structures, 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.

Parallel Control Stuctures, No Loop, C/C++

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

Parallel Control Stuctures, No Loop, FORTRAN

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

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

Parallel Control Structures, 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.

Parallel Control Stuctures, FORTRAN90

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

```
!$omp parallel workshare
  where ( x(1:n) /= 0.0 )
    y(1:n) = log ( x(1:n) )
  elsewhere
    y(1:n) = 0.0
  end where
!$omp end parallel workshare
```

Parallel Control Stuctures, FORTRAN95

```
!$omp parallel workshare
  forall ( i = k+1:n,j = k+1:n )
    a(i,j) = a(i,j) - a(i,k) * a(k,j)
  end forall
!$omp end parallel workshare
```

(This calculation corresponds to one of the steps of Gauss elimination or LU factorization)

SATISFY: 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**.

SATISFY: Problem specification

What sets of 16 logical input values **X** will cause the following function to have the value **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) )
```

SATISFY: 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?

SATISFY: 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.

SATISFY: Program Design

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

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

SATISFY: 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
```

SATISFY: 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.

Shared Memory Programming with OpenMP

Data Classification (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 explicitly declare all variables in a loop.

Data Classification (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 Classification (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.

You should list all the variables that show up in this loop, and try to determine if they are **shared** or **private** or perhaps a **reduction** variable.

Also point out which variables are shared or private **by default**.

Data Classification (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 Classification (Private/Shared)

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

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.

Data Classification (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
```

In loop #2, we give the compiler freedom to do the calculation the best it can. Is this always the solution? In an actual computation, we might only compute the vector X one element at a time, so we would never have an actual array to process.

Please suggest how we would parallelize loop #1 or loop #2!

Data Classification (Private/Shared/Reduction)

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

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

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

Shared Memory Programming with OpenMP

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, by an operation such as

```
R = ( double ) SEED / 2147483647.0
```

Most random number generators hide the seed internal in static memory, initialized to a default value, which you can see or change only by calling the appropriate routine.

Examples: Random Numbers

Some system random number generators will work properly under OpenMP, but it's very important to test them. Initialize the seed to 123456789 (for example), and compute 20 random values sequentially. Repeat the process in parallel and compare.

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

If you want a real number returned, this is computed indirectly, by an operation such as

```
R = ( double ) SEED / 2147483647.0
```

Most random number generators hide the seed internal in static memory, initialized to a default value, which you can see or change only by calling the appropriate routine.

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

Make a parallel version of this program and compare the results. But even if you happen to get the same results, I still am not comfortable with this!

If you can, you should seek a random number function whose seed is an explicit argument.

Secondly, it seems to me you can't in general, hope to set up a random number generator that allows you to compute the "50th" random number immediately, because of the way they are set up.

So perhaps a compromise is this: use a parallel section, set a seed based on the thread index, and then start a loop.

Examples: Random Numbers

```
#omp pragma 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 )
  }
#omp pragma end parallel
```

Examples: Random Numbers

Do you see why I have made my choices this way?

Do you see why I am still unhappy with this setup? (we're not really emulating a sequential version.) (when you pick several seed arbitrary, it's actually possible for one sequence to overlap another)

After setting SEED, could I call srand (seed) and then use the system **rand()** function?

Note that, for MPI, there is at least one package, called **SPRNG**, which can generate random numbers that are guaranteed to be well distributed.

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

  end do
```

Shared Memory Programming with OpenMP

Conclusion

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

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

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

Exercises for this afternoon's hands on session will introduce you to OpenMP.

You'll write a simple program to do a sum.

You will take the FFT and molecular dynamics programs and try to make OpenMP versions, demonstrate a speedup, and investigate the dependence of that speedup on the number of processors and the problem size.

You will also work on an OpenMP program in which the parallelism is not expressed in a **for** or **do** loop.

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*