Shared Memory Programming with OpenMP

ISC5316-01: Applied Computational Science II

.....

John Burkardt
Department of Scientific Computing
Florida State University

http://people.sc.fsu.edu/~jburkardt/presentations/...
...acs2_openmp_2013.pdf

03/08 October 2013

Shared Memory Programming with OpenMP

1. **Serial Programs Can't Accelerate**
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. Timing and Other Functions
8. PRIME Example
9. Private and Shared Variables
10. Reduction Operations
11. Using Random Numbers
12. SATISFY Example

SERIAL: von Neumann Architecture

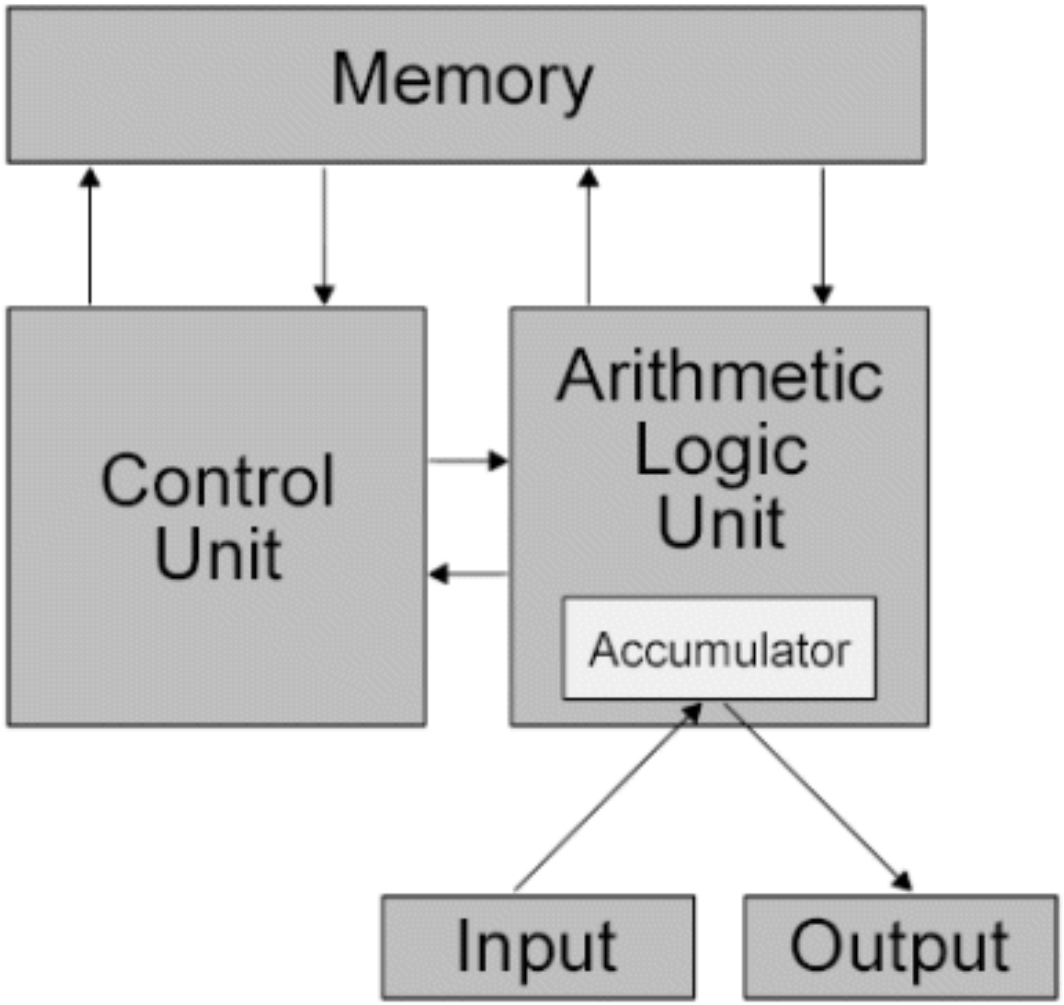
For fifty years, we have used a simple model of the computer called the von Neumann computer, consisting of memory (input data and intermediate results), a processor (carrying out arithmetic and logical operations), an input/output device, and a clock.

The von Neumann computer runs a program, also stored in memory.

The clock is surprisingly important, providing synchronization. Roughly speaking, nothing happens faster than a single clock tick.

In particular, no matter how many instructions there are in the program, just one instruction is carried out each tick. The von Neumann computer thus carries out sequential or serial execution of the program.

SERIAL: von Neumann Architecture



SERIAL: Years of Faster and Faster Clocks

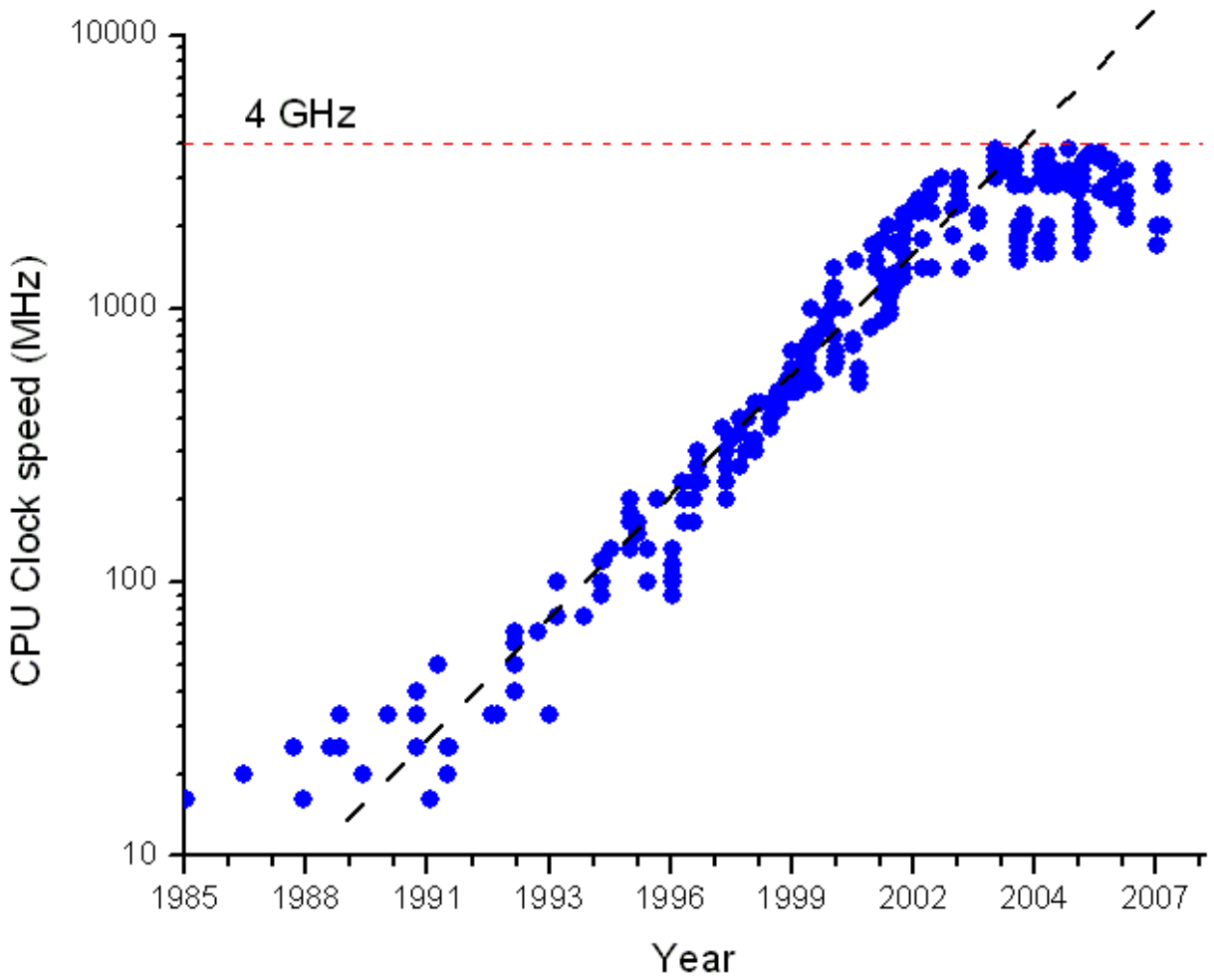
Logically, this is a beautiful and simple model. And we have kept if for so long, and designed computers and algorithms and programming languages around it, because over the years, computers have been fast enough to solve the problems we are interested in.

The problems we are interested in have grown enormously over the years, and the only reason computers were able to keep up was because it was possible to relentlessly increase the clock speed (of course, we had to make faster processors and memory access as well.)

The clock speed increased because of the increasing sophistication and miniaturization of computer chips.

Sometime around 2005, the clock speed curve hit an immovable ceiling.

SERIAL: We've Hit the Ceiling



SERIAL: Multicore Processors

The are simple alternatives to the von Neumann model of computation.

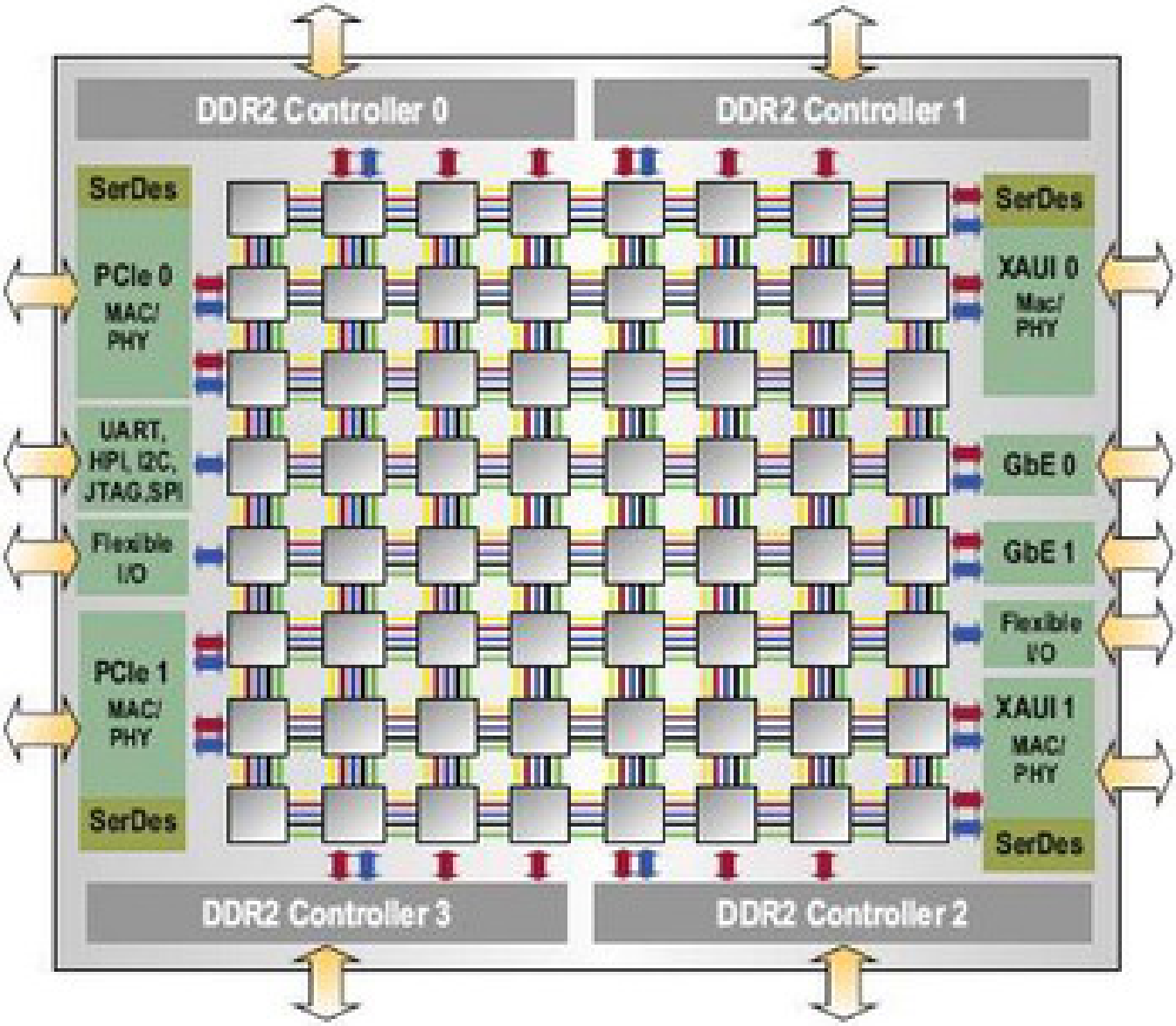
Most of these involve the idea of parallel computing, that is, of carrying out more than one instruction or operation at a time.

The clock can't tick faster, but we can do more work each clock cycle.

One hardware advance involved communication: it was possible for two different computers to share information during a computation. This fact led to the development of MPI (next week's topic!).

Another advance was the development of processors with multiple cores, which led to OpenMP (this week's topic!).

SERIAL: Tilera, One Processor, 64 Cores



SERIAL: Shared Memory Parallelism

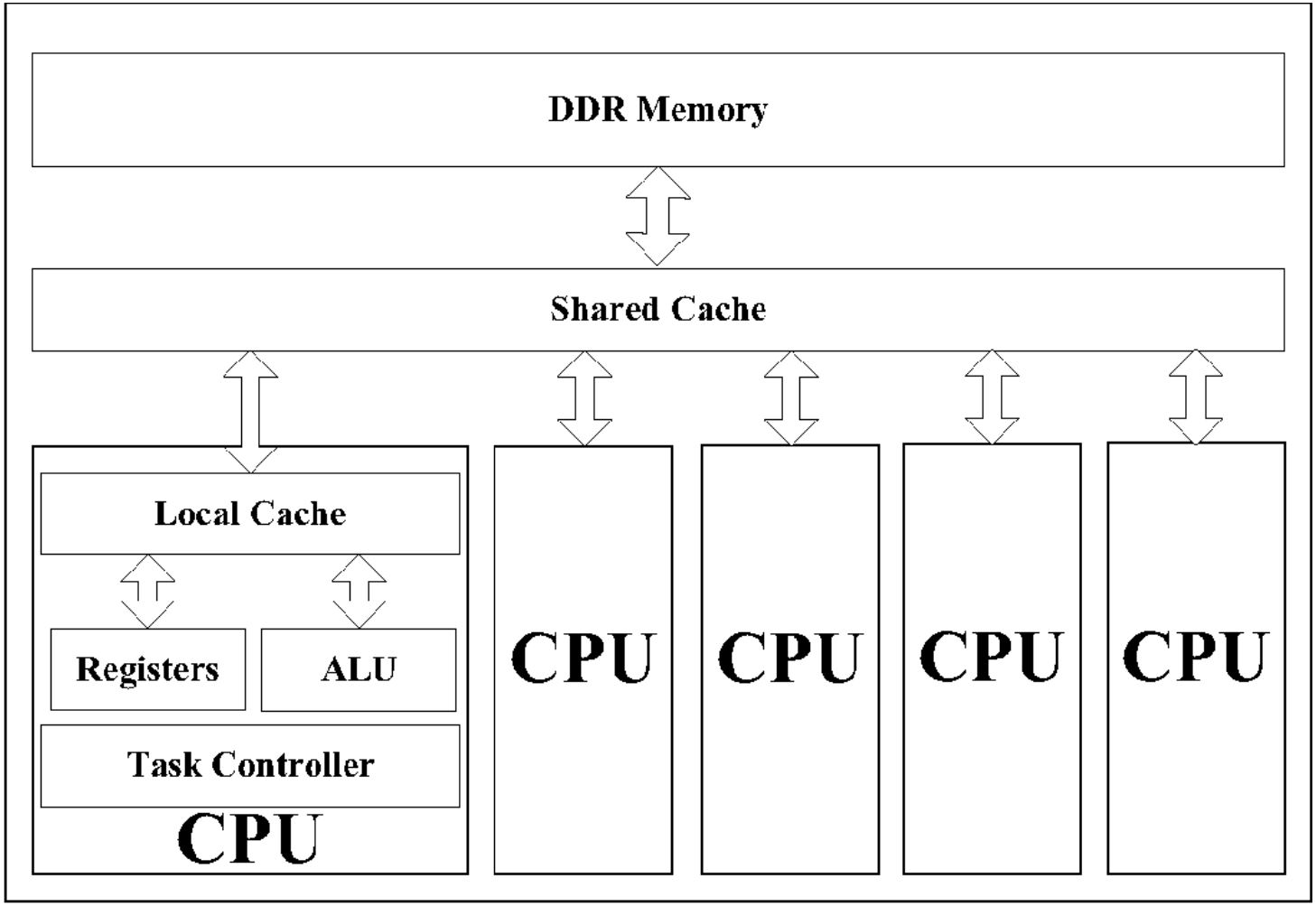
A multicore processor generalizes the von Neumann model. Instead of one processor doing the work, the processor has many cores to which it can assign independent or cooperative work, simultaneously.

If the cores can cooperate on one task, we get done faster.

But how can we write a new kind of computer program that can correctly control multiple, cooperating cores?

We'll assume that the cores live in a shared memory space, so that any core can read or write any piece of data.

SERIAL: The "non Neumann" Architecture



SERIAL: OpenMP Controls Multicore Processors

Suppose we have a list of numbers, and we simply want a new list containing the squares of each number.

Our program should read something like:

```
Get a number from the list in locations A through B.
Square the number.
Insert the number into list in locations C through D.
```

For our parallel program, we need to:

- decide how many cores will be involved;
- determine limits A:B and C:D for each core;
- give each core a copy of the program, and the values of limits.

I hope you can see that, if we can generate these instructions, the program will work correctly.

OpenMP is a system that exactly allows us to write programs that will be correctly executed on multiple cores; if done correctly, such a parallel program can run a hundred times faster if we have 100 cores available.

Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. **Parallel Algorithms Are Available**
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. Timing and Other Functions
8. PRIME Example
9. Private and Shared Variables
10. Reduction Operations
11. Using Random Numbers
12. SATISFY Example

ALG: An Algorithm Need Not be "Step by Step"

We think of algorithms as step-by-step procedures, in which step 1 must be completed before step 2 can be started, but this is **not** an essential feature of an algorithm!

The step-by-step approach is called *serial* or *sequential* programming.

We look at a problem like

$$S = X0 + X1 + X2 + X3 + X4 + X5 + X6 + X7$$

and can only see a serial approach:

$$\begin{aligned}
(\text{clock} = 1)\ S &= X0 \\
(\text{clock} = 2)\ S &= S + X1 \\
(\text{clock} = 3)\ S &= S + X2 \\
&\ldots \\
(\text{clock} = 8)\ S &= S + X7
\end{aligned}$$

For many algorithms, multiple steps can be going on at the same time. The point of parallel programming is to identify those opportunities.

ALG: Adding N Numbers Doesn't Need N Steps

If we have four cores, and room to store 4 temporary sums, and if all the cores can work at the same time, we can do the addition in 4 clock cycles (we are really working on a binary tree here):

$$(\text{clock} = 1)\ S0 = X0, S1 = X2, S2 = X4, S3 = X6$$
$$(\text{clock} = 2)\ S0 = S0 + X1, S1 = S1 + X3, S2 = S2 + X5, S3 = S3 + X7$$
$$(\text{clock} = 3)\ S0 = S0 + S1, S2 = S2 + S3$$
$$(\text{clock} = 4)\ S0 = S0 + S2$$

Assuming "unlimited resources" ($n/2$ cores!), we can add n numbers in $1 + log_2(n)$ rather than n cycles, so 1,000 numbers could be added in about 10 steps rather than 1,000, and 1,000,000 numbers could be added in about 20 steps rather than 1,000,000!

Of course, typically we don't have thousands of cores available, but it is still true that we really can solve problems faster if we can find, describe, and implement an appropriate parallel algorithm.

ALG: Parallel Problems Obviously Exist

Are there classes of problems we can solve in parallel?

Search: if you need to:

- find the joker in a deck of cards;
- find the triangle in a mesh that contains a given point;
- find proteins in a database that are similar to a sample protein.

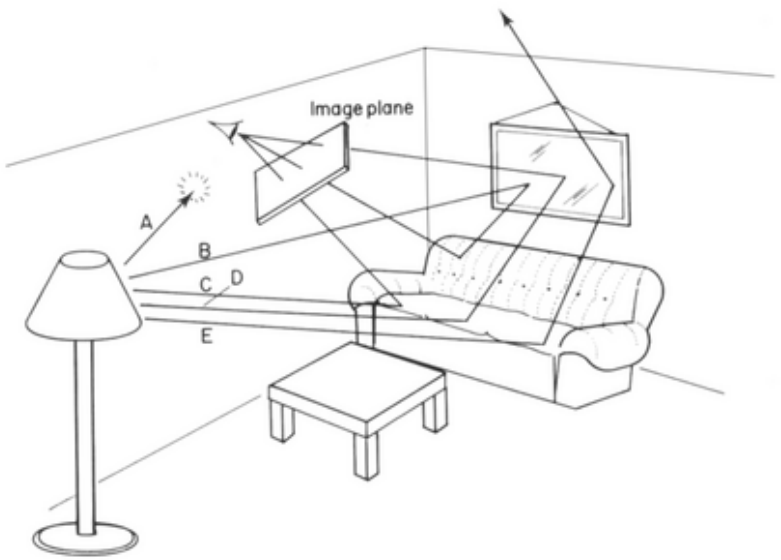
then you can divide up the deck, or the mesh, or the database, among the available cores, and run until one of the cores finds a match.

Sort: each core can take a portion of the items and sort them, returning the sorted set to the "master" core. The master then simply has to merge the sorted items.

ALG: Image Processing

Image Processing: An image is simply an array of pixels. We could be searching for patterns (a tumor, a star, a camouflaged missile battery).

We could be trying to "render" an image, that is, to simulate the look of a model with a given light source. A standard technique called **ray tracing** is to send out thousands of light rays from the source, letting them hit objects in the model, reflect and bounce off other objects, before exiting the model. We can do this in parallel by giving each core its own set of random light rays to track.



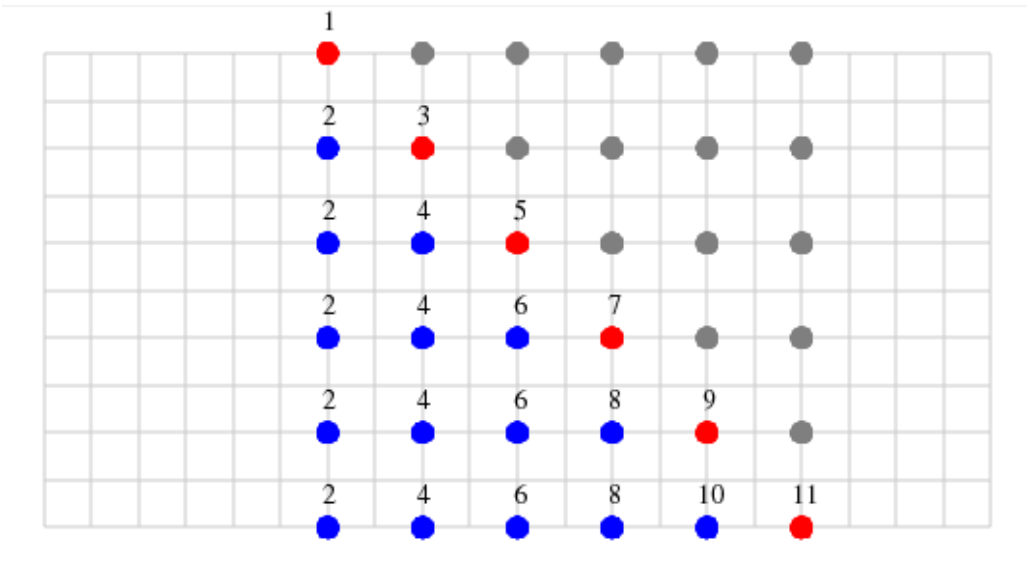
ALG: Gauss Elimination

Gauss Elimination: Suppose we have a linear system of the form:

$$A * x = b$$

Gauss elimination can produce the answer. The k-th step seeks a pivot row, which is used to eliminate elements in the matrix:

- Find $A_{p,k}$, the largest entry on or below the diagonal $A_{k,k}$;
- Swap rows p and k;
- Add a multiple of row k to rows $k+1$ through n to eliminate the entry in column k;



ALG: Jacobi Iteration

Jacobi Iteration: Suppose we have a linear system of the form:

$$A * x = b$$

where A is a symmetric positive definite matrix. Then Jacobi iteration can be used to estimate the solution. On the k-th step of Jacobi iteration, we compute the solution estimate x^k by

$$A_{i,i}\ x_i^k = b_i - \sum_{j=1;j\neq i}^{n} A_{i,j}\ x_j^{k-1}$$

or, in other words:

$$x_i^k = (b_i - \sum_{j=1;j\neq i}^{n} A_{i,j}\ x_j^{k-1})/A_{i,i}$$

so we can update all the entries in x^k at the same time.

ALG: An Ordinary Differential Equation (OOPS!)

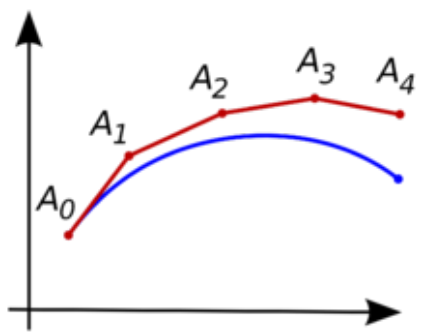
An ordinary differential equation in time: Suppose we have a differential equation of the form:

$$u'(t) = f(u, t)$$
$$u(t = 0) = u_0$$

We can define a discretized problem, in which we seek approximate values of $u()$ at the sequence of points $t_1, t_2, \ldots$ equally spaced by dt. We might do this with a simple Euler method:

$$u_i = u_{i-1} + dt * f(u_{i-1}, t_{i-1})$$

But such an approach cannot be parallelized! Here, the problem is easy to see. The calculation of u_i cannot begin until the calculation of u_{i-1} is complete. But that can't begin until u_{i-2} is computed, and so on.

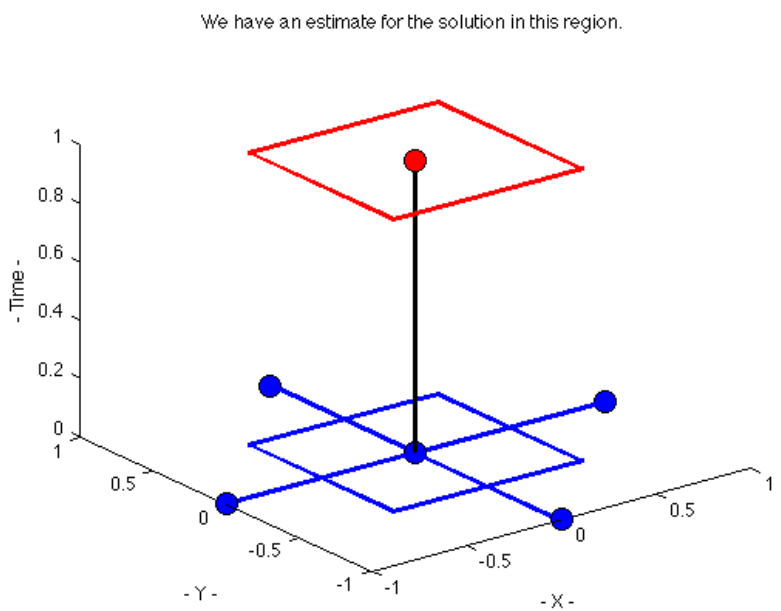


ALG: Partial Differential Equations

A partial differential equation in time and space: Suppose we have a differential equation of the form:

$$\frac{\partial u}{\partial t} + \nu\left(\frac{\partial^2 u}{\partial^2 x} + \frac{\partial^2 u}{\partial^2 y}\right) = f(u, t, x, y)$$

with corresponding initial and boundary conditions. If we discretize space and time, our solution is an array $u(t_i, x_j, y_k)$. If we use a forward difference for the time derivative, we relate each value $u(t_i, x_j, y_k)$ to known values at the previous time t_{i-1}. Given data at one time step, we can compute all the data at the next time step in parallel.



ALG: Newton's Method (OOPS!)

Solving F(X)=0 Using Newton's Method: Suppose we have a scalar equation $f(x) = 0$ to be solved for x. Newton's method produces a sequence of presumably improved estimates:

$$x^{i+1} = x^i - f(x^i)/f'(x^i)$$

However, we can't do this problem in parallel, since we can't start working on x^{17} until x^{16} has been computed.

We might be able to give each core a different starting point. Or, if x is actually an m-dimensional vector, we might be able to solve the related linear systems in parallel.

Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. **The Parallel Loop**
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. Timing and Other Functions
8. PRIME Example
9. Private and Shared Variables
10. Reduction Operations
11. Using Random Numbers
12. SATISFY Example

LOOP: Terminology

So far, I have been trying to use the word **core** to describe the computing "thingies" that work in parallel. But "core" is a hardware term, and OpenMP is a software system, and so OpenMP calls the things that cooperate **processors**.

This annoys people who build computer hardware; but let's adopt the OpenMP term from now on. In order for an OpenMP program to run in parallel, it needs to have access to multiple cooperating processors, and we don't care if these are bits of computer hardware, or clerks with abacuses, or angels.

LOOP: OpenMP Concentrates on FOR and DO Loops

There are several ways that OpenMP allows you to create parallel programs. However, the simplest and most useful one concentrates entirely on operations that are carried out in loops - these are **for** loops in C/C++ and **do** loops in Fortran.

The idea is that

- a loop represents a lot of work;
- each step is the same instructions, so we can parallelize;
- each step is independent of the others (not always true!)

If these items are true, we can break the loop work into sections to be executed independently.

LOOP: Loop Iterations Are Divided Among Processors

For example, we might imagine that the sequential loop:

```
for ( i = 0; i < 1000; i++ )
{
  x[i] = x[i] + s * y[i];
}
```

could be parallelized for two processors as:

```
Processor #0

for ( i = 0; i < 500; i++ )
{
  x[i] = x[i] + s * y[i];
}
```

```
Processor #1

for ( i = 501; i < 1000; i++ )
{
  x[i] = x[i] + s * y[i];
}
```

It's easy to extend this to any number of processors.

LOOP: Indexing Can Be More Complicated

We can use more complicated formulas and indices. It's just important that distinct iterations in a loop don't try to set the same variable.

```
for ( i = 1; i <= n; i++ )
{
  x[i]   = sqrt ( y[i-1) );
}

for ( i = 0; i <= 1000; i = i + 2 )
{
  z[i]   = sin ( i * pi );
  z[i+1] = cos ( i * pi );
}
```

Ask yourself: can processors executing the loop simultaneously, but for different values of **i**, interfere with each other?

LOOP: "Right Hand Side" Variables Are Usually OK

When looking for interference, we never have to worry about variables that only appear on the right hand side. They represent memory that is "read", but not modified:

```
for ( i = 0; i <= 1000; i = i + 2 )
{
  y[i] = y[i] + x[i-1] - 2.0 * x[i] + x[i+1] + z[99] + w;
}
```

LOOP: "Left Hand Side" Variables Can Conflict

Problems occur if more than one loop iteration tries to write or modify the same variable, which occurs on the left hand side of a statement.

Here, we have a **y** vector; we'd want to add half of each entry to the corresponding "left" entry in **x** and half to the right.

```
for ( i = 1; i < n - 1; i++ )
{
  x[i-1] = x[i-1] + 0.5 * y[i];
  x[i+1] = x[i+1] + 0.5 * y[i];
}
```

Even in our simple two-processor model, this code will have the potential of conflicts. Suppose that **n**=1000. Processor #0 might try to execute the second addition for $i = 499$ while processor #1 is executing the first addition for $i = 501$.

LOOP: "Left/Right Hand Side" Variable Problems

Sometimes a variable occurs on both the left and right hand side. Since this means that the variable's value changes during the loop execution, it means we can't safely run it in parallel.

This code is overwriting **x** by its cumulative sums:

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

Note that if we try to compute
$\mathbf{x[2] = x[2] + x[1]}$,
the value will depend on whether we have already executed the statement
$\mathbf{x[1] = x[1] + x[0]}$.

LOOP: Another Example of Side Effects

Aside from vector references depending on the loop index, another common programming practice that can interfere with parallel programming involves temporary variables that are updated during the loop iteration.

For example, let's plot the function $y = x^2$ between 0 and 1, by evaluating the function at 1001 equally spaced points:

```
x = 0.0;
for ( i = 0; i <= 1000; i++ )
{
  y[i] = x * x;
  x = x + 0.001;
}
```

This loop (as written) can't execute in parallel.

But it's easy to come up with an equivalent loop that avoids this problem.

LOOP: Another Example of Left/Right Variables

Problems can occur if data appears on both the left and right hand side, and so is changed during the calculation.

Here is a sort of Gauss-Seidel iteration for solving a linear system. Why does the loop model fail here?

```
for ( i = 1; i < n - 1; i++ )
{
  x[i] = ( b[i] + x[i-1] + x[i+1] ) / 2.0;
}
```

LOOP: A Summation

Here's another example, (approximating an integral) which is actually important enough that we will see how to fix it later:

```
n = 1000;
q = 0.0;
for ( i = 0; i < n; i++ )
{
  x = i / ( double ) n;
  q = q + x * x;
}
q = q / n;
```

In this loop, **x** is not the problem, it's **q**, which is being modified on every iteration. In our two processor parallel version, would we have two separate variables called **q**? If so, what do we do with them at the end? If there's just one variable, and the processors have to share it, then how do we avoid conflicts?

LOOP: Simple Rules for Parallel Loops

In summary,

- If we want to run a loop in parallel, it should be written in such a way that the loop iterations would get the same results, even if they were executed in the reverse order, or any order;
- Moreover, we need to avoid cases in which the same variable is modified by two different iterations of the loop;
- Some loops, like the integral approximation, use a single variable to collect results from all the iterations. If we want to use such methods, we need to come up with a special approach.

Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. **SAXPY Example: Vector Addition**
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. Timing and Other Functions
8. PRIME Example
9. Private and Shared Variables
10. Reduction Operations
11. Using Random Numbers
12. SATISFY Example

SAXPY: A Basic Linear Algebra Operation

Our simple example starts with an **n**-vector called **x** and adds to it the vector **y**, multiplied by the scalar **s**:

$$\vec{x} \leftarrow \vec{x} + s \cdot \vec{y};$$

We assume that the values of **x** and **y** are set by some formula, about which we don't really care that much.

If we have multiple processors, then, all we are asking is that they divide up the range of vector indices, and then carry out the arithmetic for their part of the work.

SAXPY: C Example (Before)

```
int main ( )
{
  int i, n = 1000;
  double s = 1.23, x[1000], y[1000];

  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) ( ( i + 1 ) % 17 );
    y[i] = ( double ) ( ( i + 1 ) % 31 );
  }




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

SAXPY: C Example (After)

```
# include <omp.h>

int main ( )
{
  int i, n = 1000;
  double s = 1.23, x[1000], y[1000];

  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) ( ( i + 1 ) % 17 );
    y[i] = ( double ) ( ( i + 1 ) % 31 );
  }

# pragma omp parallel

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

SAXPY: F90 Example (Before)

```
program main


  integer, parameter :: n = 1000

  integer i
  double precision :: s = 1.23
  double precision x(n), y(n)

  do i = 1, n
    x(i) = mod ( i, 17 )
    y(i) = mod ( i, 31 )
  end do



  do i = 1, n
    x(i) = x(i) + s * y(i)
  end do



  stop
end
```

SAXPY: F90 Example (After)

```
program main

  use omp_lib

  integer, parameter :: n = 1000

  integer i
  double precision :: s = 1.23
  double precision x(n), y(n)

  do i = 1, n
    x(i) = mod ( i, 17 )
    y(i) = mod ( i, 31 )
  end do

!$omp parallel

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

!$omp end parallel

  stop
end
```

SAXPY: The Changes

Notice that our OpenMP program looks exactly the same as our original program, except for the OpenMP directives.

These directives, in fact, look like comments to the compiler. In other words, if you simply compile the OpenMP program in the usual way, it will compile and run just as it did before.

You might think this is not a big accomplishment, but what it means is that, as you modify your program to create an OpenMP version, you can always run the program in sequential mode as a check.

Moreover, we can work on our program one section at a time. There may be many loops in our program, but OpenMP will only parallelize the loops we select. So it's easy to experiment and to work in steps.

Now let's try to find out a little bit more about the directives that we added to the program.

Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. **Basic OpenMP Directives**
6. Compiling, Linking, Running
7. Timing and Other Functions
8. PRIME Example
9. Private and Shared Variables
10. Reduction Operations
11. Using Random Numbers
12. SATISFY Example

DIRECT: The Parallel Region is Defined by a Directive

OpenMP includes a small number of functions and symbolic constants, which must be declared.

Therefore, a C or C++ program that uses OpenMP should usually have the following include statement at the beginning of the file:

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

Every Fortran77 subroutine or function that uses OpenMP should have the following include statement:

```
      include 'omp_lib.h'
```

A FORTRAN90 subroutine or function using OpenMP can use the Fortran77 include statement, or reference the OpenMP module:

```
  use omp_lib
```

DIRECT: The Parallel Region is Defined by a Directive

The C/C++ **parallel** directive begins a *parallel region*.

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

Typically, this parallel region will contain one or more **for** loops; these loops may be selected for parallel execution if the user indicates so.

As in many cases in C and C++, the curly brackets are optional; if omitted, the **parallel** statement applies only to the very next block of code. If you have several successive **for** loops that should all be parallel in the same way, you are better off using the curly brackets.

DIRECT: The Parallel Region is Defined by a Directive

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

```
!$omp parallel
  do things in parallel here, if directed!
!$omp end parallel
```

The parallel region must be closed with an **end parallel** directive; thus in Fortran you always explicitly declare where the parallel region ends.

Typically, this parallel region will contain one or more **do** loops; these loops may be selected for parallel execution if the user indicates so.

DIRECT: Variables in the Parallel Region

All variables inside the parallel region will be classified as:

- **shared**, leave this in shared memory;
- **private**, each core needs a private copy;
- **reduction**, special treatment.

Except for loop indices, all variables are assumed to be shared.

It's important that no variable be misclassified. To override the defaults, or to declare some variables explicitly, add the **private()** or **shared()** clause to your **parallel** directive.

Our example doesn't need to specify this information, so we'll come back to discuss this in a later example!

DIRECT: PRIVATE and SHARED Clauses

The **private()** and **shared()** clauses modify the **parallel** directive.

In C, this might appear like this:

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

or:

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

while in FORTRAN90, the same information would look like this:

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

or:

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

DIRECT: Parallel Loops are Marked by Directives

Loops in the parallel region will **not** be executed in parallel, until you mark that loop with a **for** directive in C/C++, or a **do** directive in FORTRAN.

If a parallel region has five loops, you can mark any or all of them.

A nested loop only requires one directive.

```
# pragma omp for
for ( i = 0; i < m; i++ )
{
      ...xxx...
 }
# pragma omp for
for ( i = 0; i < 1000; i++ )
{
      ...xxx...
 }
```

DIRECT: Nested Loops

A nested loop should only be marked by one directive.

```
# pragma omp for
for ( i = 0; i < m; i++ )
{
   for ( j = 0; j < n; j++ )
   {
      ...xxx...
   }
 }
```

```
!$ omp do
do i = 1, m
  do j = 1, n
     ...xxx...
  end do
end do
!$ omp end do
```

DIRECT: The Loop Directive in the SAXPY Example

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

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

DIRECT: SAXPY Example in C

```
# include <omp.h>          <-- OpenMP Definitions

int main ( )
{
  int i, n = 1000;
  double s = 1.23, x[1000], y[1000];

  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) ( ( i + 1 ) % 17 );
    y[i] = ( double ) ( ( i + 1 ) % 31 );
  }

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

DIRECT: SAXPY Example in F90

```
program main

  use omp_lib                 <-- OpenMP Definitions

  integer, parameter :: n = 1000

  integer i
  double precision :: s = 1.23
  double precision x(n), y(n)

  do i = 1, n
    x(i) = mod ( i, 17 )
    y(i) = mod ( i, 31 )
  end do

!$omp parallel              <-- Begin parallel region

!$omp do                      <-- Next loop is parallel
  do i = 1, n
    x(i) = x(i) + s * y(i)
  end do
!$omp end do                  <-- End of that loop

!$omp end parallel            <-- End of parallel region

  stop
end
```

DON'T DESPAIR, WE'RE HALFWAY THERE!

Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. **Compiling, Linking, Running**
7. Timing and Other Functions
8. PRIME Example
9. Private and Shared Variables
10. Reduction Operations
11. Using Random Numbers
12. SATISFY Example

RUN: OpenMP Directives Are Ignored

Because the OpenMP directives look like comments, your program will run sequentially if you compile it the usual way -
even after you have added OpenMP directives!

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

`./a.out` *runs as a sequential program*

RUN: Creating and Running an Executable

The compile statement results in the creation of what is called the *executable* program - that is, it's ready to run.

By default, the executable is stored in a file with the peculiar name of *a.out*. To avoid confusion, this should be renamed to something sensible:

```
mv a.out myprog
```

The executable program can be run by typing its name, preceding by `./`, which is actually shorthand for the current directory:

```
./myprog
```

RUN: OpenMP Directives Can Be Activated

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

GNU compilers do this with the **-fopenmp** switch:

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

This time, we choose a different name for the executable:

```
mv a.out myprog_omp
```

RUN: Intel Compiler Switches

Intel C compilers need 2 switches, **openmp** and **parallel**:

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

Intel Fortran compilers need 3 switches, **openmp** and **parallel** and **fpp**:

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

RUN: Threads Versus Processors

OpenMP knows how many processors (cores) are available on the system.

However, when you want to run in parallel, you actually specify how many threads you want; this is the number of parallel tasks to be carried out at one time, and usually means how you want to "slice up" your loop.

Using 1 thread means sequential execution.

Asking for 2 threads means the work will be split into two chunks, and it will be done in parallel if there are at least two processors available.

Similarly, we can ask for 8 threads, but we might only have four processors. In that case, each processor will handle two threads.

It usually makes sense to ask for the number of threads to be the number of processors; *occasionally* you can get a speedup by having twice the number of threads.

RUN: Specifying the Number of Threads

When we run a program whose OpenMP directives have been activated, then 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 is the same as 1. Usually, however, it's set to a sensible value, such as the number of cores available.

You can reset this environment variable using a command like:

```
export OMP_NUM_THREADS=4   <-- (No spaces around equal sign!)
```

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

RUN: Trying Different Numbers of THreads

Changing the number of threads is easy, and can be done at run time. Suppose our executable program is called **myprog_omp**.

We could experiment with 1, 2, 4 and 8 threads by:

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

These commands have **no effect** on a sequential program. Only a program compiled with the OpenMP switches will notice that there are multiple threads available.

RUN: Trying Different Numbers of Threads

We know a little bit about how to convert a sequential program into an OpenMP parallel program (include file, parallel and do/for directives).

We know how to compile;

We know how to run;

We know how to set the number of threads, which make the program "more parallel".

But the point of parallel programming is to run faster and to do that, we need to measure the time it takes a program to run.

How do we time our OpenMP programs?

Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. **Timing and Other Functions**
8. PRIME Example
9. Private and Shared Variables
10. Reduction Operations
11. Using Random Numbers
12. SATISFY Example

TIME: Measuring Wall Clock Time

Parallel programming does the same amount of work as sequential programming, and in fact, it might even do more. So it's not a good idea, when making comparisons, to measure the amount of work.

It's not a good idea to measure the total CPU time (how much time was used by all the processors), because this will also, almost surely, be larger than the CPU time for a sequential run.

There are two important quantities in parallel programming:

- the **elapsed wall clock time** - how long did you wait for a result?
- the **relative speedup**: the wall clock time using 1 processor divided by the wall clock time using **p** processors.

TIME: Measuring Wall Clock Time

There are several things to keep in mind when measuring performance:

- your parallel program may run slower on small versions of your problem;
- parallelizing one big thing is much better than parallelizing many small things;
- some programs run too quickly, or work on too little data, to be worth parallelizing;
- if you let your problem get big enough, it will suddenly slow down drastically when it reaches the memory limit.

TIME: OpenMP Functions

OpenMP includes the following functions:

- **omp_set_num_threads (t)** : set the number of threads;
- **t = omp_get_num_threads ( )** : get the number of threads;
- **t = omp_get_max_threads ( )** : get the maximum number of threads;
- **p = omp_get_num_procs ( )** : how many processors are there?
- **id = omp_get_thread_num ( )** : which thread is executing?
- **wtime = omp_get_wtime()** : how much time has elapsed?

At the moment, we are goint to need the timing function!

TIME: How Much Time Has Passed?

The function **omp_get_wtime()** returns, as a double precision real number, the current reading of the wall clock.

You read the wall clock time before and after a parallel computation. The difference gives you the measured time.

```
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";
```

TIME: The SAXPY Example

```
# include <omp.h>

int main ( )
{
  int i, n = 1000;
  double wtime, s = 1.23, x[1000], y[1000];

  wtime = omp_get_wtime ( );  <-- Start the clock

# pragma omp parallel
  {  <-- parallel region begins with this bracket...
# pragma omp for
    for ( i = 0; i < n; i++ )
    {
      x[i] = ( double ) ( ( i + 1 ) % 17 );
      y[i] = ( double ) ( ( i + 1 ) % 31 );
    }

# pragma omp for
    for ( i = 0; i < n; i++ )
    {
      x[i] = x[i] + s * y[i];
    }
  }  <-- and parallel region ends with this bracket.
  wtime = omp_get_wtime ( ) - wtime;  <-- Stop the clock
  printf ( "%g seconds.\n", wtime );
  return 0;
}
```

TIME: the SAXPY Example!

```
program main

  use omp_lib

  integer, parameter :: n = 1000

  integer i
  double precision :: s = 1.23
  double precision wtime, x(n), y(n)

  wtime = omp_get_wtime ( );  <-- Start the clock.

!$omp parallel

!$omp do
  do i = 1, n
    x(i) = mod ( i, 17 )
    y(i) = mod ( i, 31 )
  end do
!$omp end do

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

!$omp end parallel
  wtime = omp_get_wtime ( ) - wtime  <-- Stop the clock.
  write ( *, * ) wtime, ' seconds.'
  stop
end
```

TIME: Run SAXPY with Timings

Now our example program is tiny (as far as the number of instructions) and also as far as the amount of work.

So to start with, let's increase the value of **n** to 100,000.

Secondly, we are going to insert the timing calls (this should involve four new lines of code.)

I will recompile the program with OpenMP enabled, and run with 1, 2, 4, 8 and 16 threads.

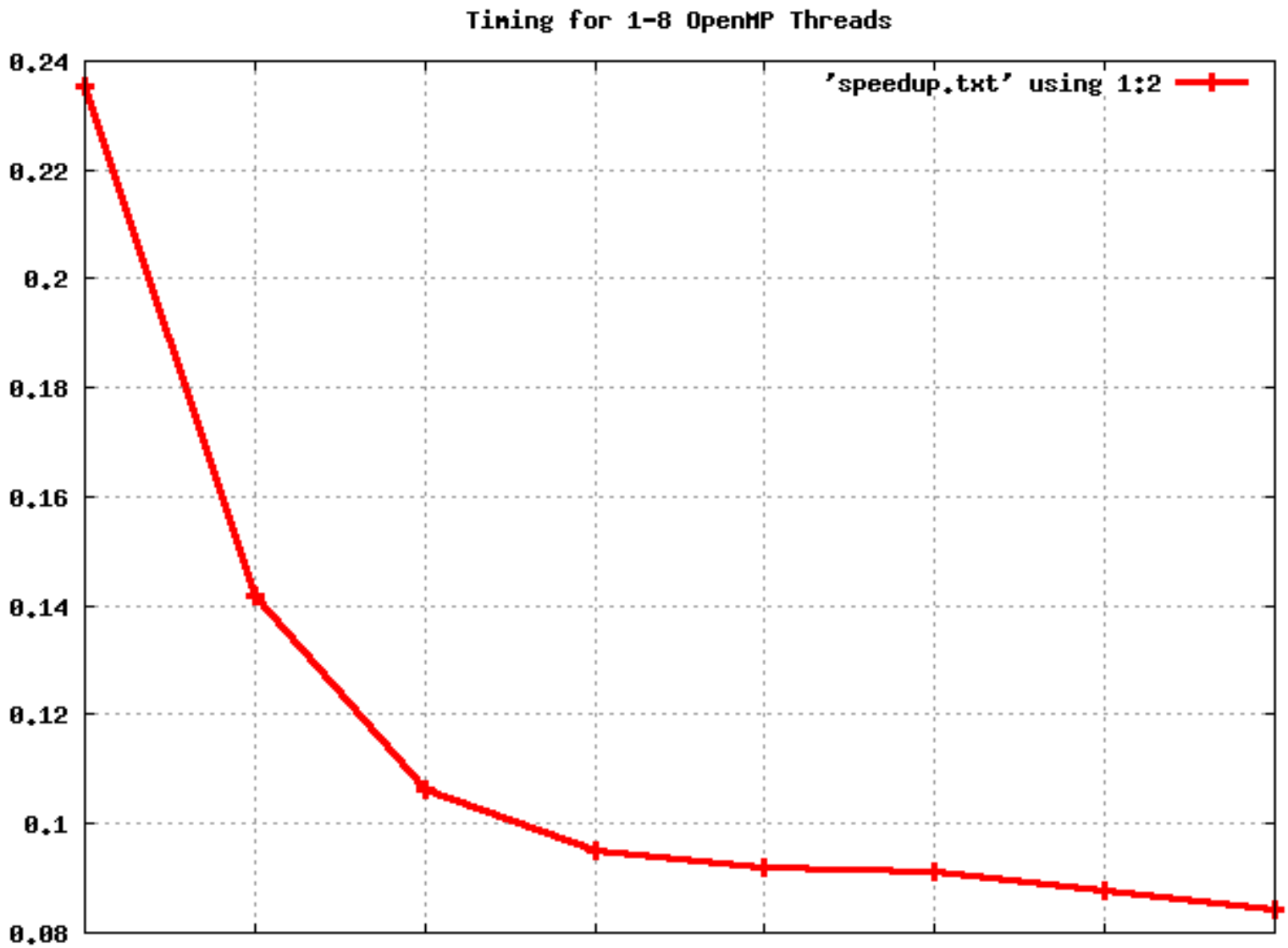
If I plot the **time**, I am likely to get a hyperbola. If I plot the speedup, that is, **p**-processor time divided by the 1-processor time, I should get a diagonal line (if things are going well.). Lines that are diagonal (or not) are much easier to understand and judge than hyperbolas!

TIME: GNUPLOT Commands for Time Plot

```
gnuplot
  set term png

  set output 'timing1.png'
  set style data linespoints
  set title "Timing for 1-8 OpenMP Threads"
  set grid
  plot 'speedup.txt' using 1:2 lw 3

  set output 'timing2.png'
  set style data linespoints
  set title "Timing for 1-8 OpenMP Threads"
  set grid
  set yrange [0:*]                          <-- Important!
  plot 'speedup.txt' using 1:2 lw 3

  quit
```

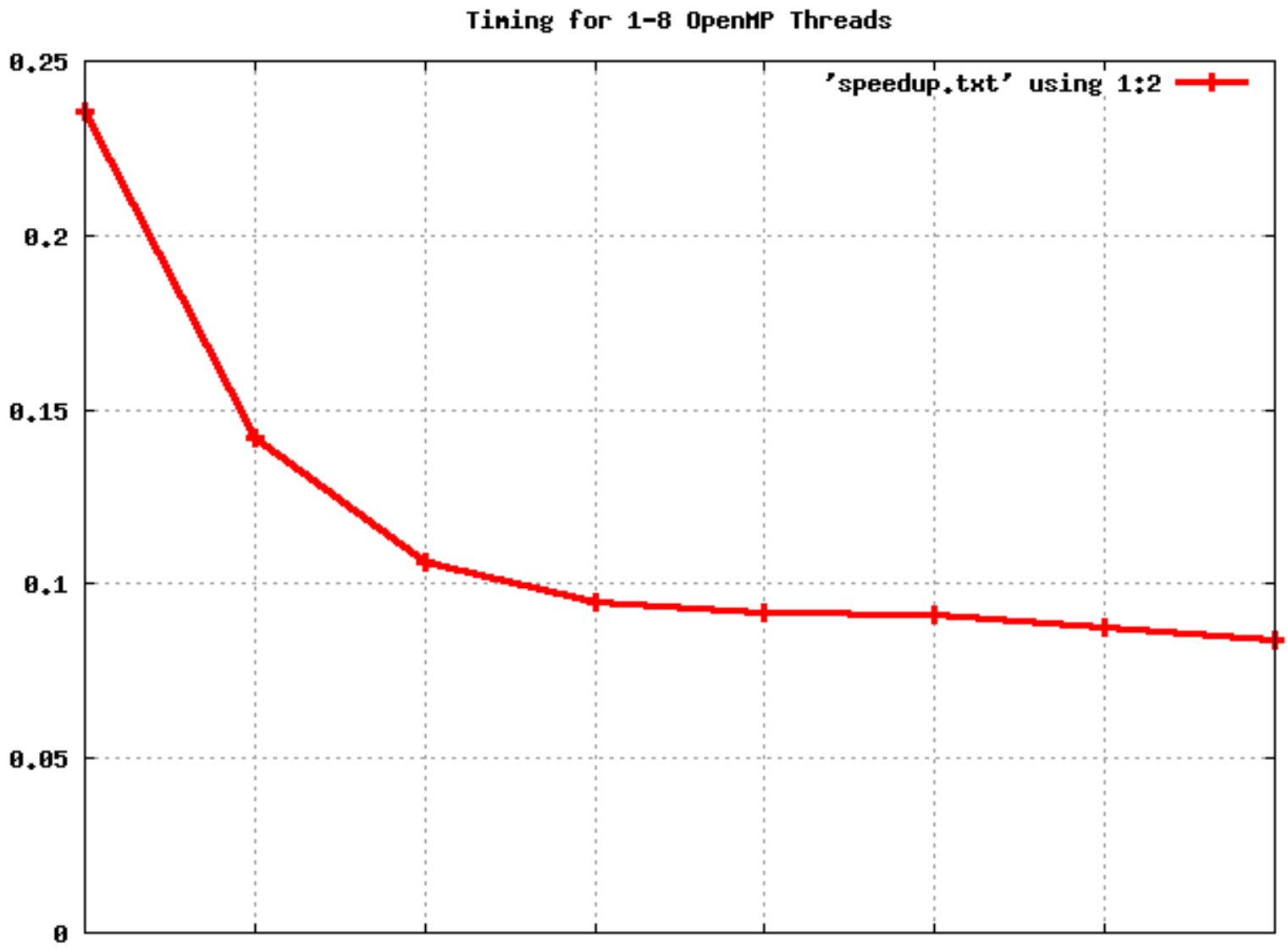
TIME: Wall Clock Time as a Function of Threads

This graph incorrectly suggests that our time is dropping to zero.



TIME: Wall Clock Time as a Function of Threads

This graph forces the Y axis to start at 0.

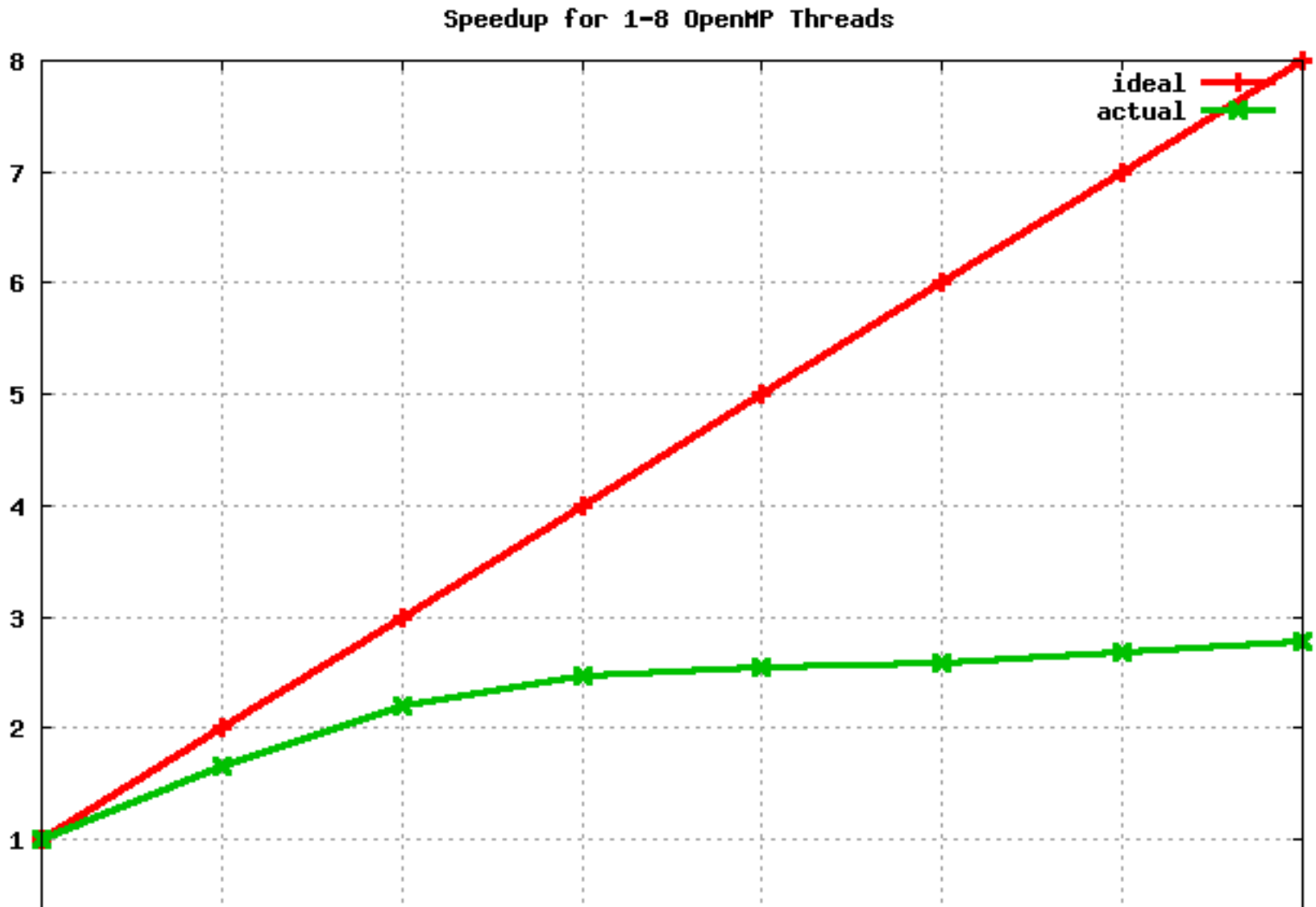


TIME: GNUPLOT Commands for Speedup Plot

```
gnuplot
  set term png

  set output 'speedup.png'
  set style data linespoints
  set title "Speedup for 1-8 OpenMP Threads"
  set grid
  plot 'speedup.txt' using 1:1 title 'ideal' lw 3, \
       'speedup.txt' using 1:3 title 'actual' lw 3
  quit
```

TIME: Speedup Plot

This graph uses the same data, but is much easier to understand (and to be sad about - our performance is not great!)



Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. Timing and Other Functions
8. **PRIME Example**
9. Private and Shared Variables
10. Reduction Operations
11. Using Random Numbers
12. SATISFY Example

PRIME: Counting the Number of Primes

The timing function is very useful:

- it can make you happy (things speed up)!
- it can puzzle you (why don't they speed up more?)
- it can anger you (what? things slowed down???)

Let's look at a simple program to count the prime numbers. We'll turn it into an OpenMP program, run it with 1 thread and 2 threads, and ask what information is coming back to us from the timing results.

PRIME: The Serial C Code

```
n = n_lo;

while ( n <= n_hi )
{
  total = 0;

  for ( i = 2; i <= n; i++ )
  {
    prime = 1;

    for ( j = 2; j < i; j++ )
    {
      if ( i % j == 0 )
      {
        prime = 0;
        break;
      }
    }
    total = total + prime;
  }
  printf ( "  %8d  %8d\n", n, total );
  n = n * n_factor;
}
```

PRIME: Counting the Number of Primes

Where is the loop to parallelize?

What are two reasons we cannot parallelize the **j** loop?

If we have two processes (or "threads") running parts of the **i** loop at the same time, which variables must we make copies of so the threads don't interfere with each other?

What variable needs to be handled in a special way?

If we run this code on 2 threads, will it run (about) twice as fast?

PRIME: The OpenMP C Code

```
  n = n_lo;

  while ( n <= n_hi )
  {
    wtime = omp_get_wtime ( );
# pragma omp parallel shared ( n ) private ( i, j, prime )

# pragma omp for reduction ( + : total )
    total = 0;
    for ( i = 2; i <= n; i++ )
    {
      prime = 1;

      for ( j = 2; j < i; j++ )
      {
        if ( i % j == 0 )
        {
          prime = 0;
          break;
        }
      }
      total = total + prime;
    }
    wtime = omp_get_wtime ( ) - wtime;
    printf ( "  %8d  %8d  %12f\n", n, total, wtime );
    n = n * n_factor;
  }
```

PRIME: Compare Results

Here are timings using 1 thread and 2 threads.

N	Pi	Time(1)	Time(2)
1	0	0.000077	0.000120
2	1	0.000002	0.000006
4	3	0.000002	0.000003
8	7	0.000002	0.000003
16	13	0.000002	0.000003
32	24	0.000003	0.000003
64	42	0.000007	0.000006
128	73	0.000019	0.000014
...	..	....	....
4096	1269	0.006119	0.009021
8192	2297	0.023119	0.016182
16384	4197	0.077573	0.057229
32768	7709	0.283234	0.206660
65536	14251	1.051483	0.771272
131072	26502	5.166276	4.116094
262144	49502	18.337702	13.203991

PRIME: Compare Results

We could probably guess that the program runs faster, even without calling the timer routine.

But it's when we look at the timing results that we realize that something is not working the way we expect.

It turns out that OpenMP takes the loop running from 2 to **n** and assigns the first half of the indices to the first thread, the second to the second thread. Ordinarily, there's no reason to think that's a bad idea.

But the work only gets done in half the time if each thread gets the same amount of work. Does that happen here?

If we notice that the work load varies a lot, then there we can modify the program, or use more sophisticated OpenMP instructions that will help us try to distribute the work better.

Shared Memory Programming with OpenMP

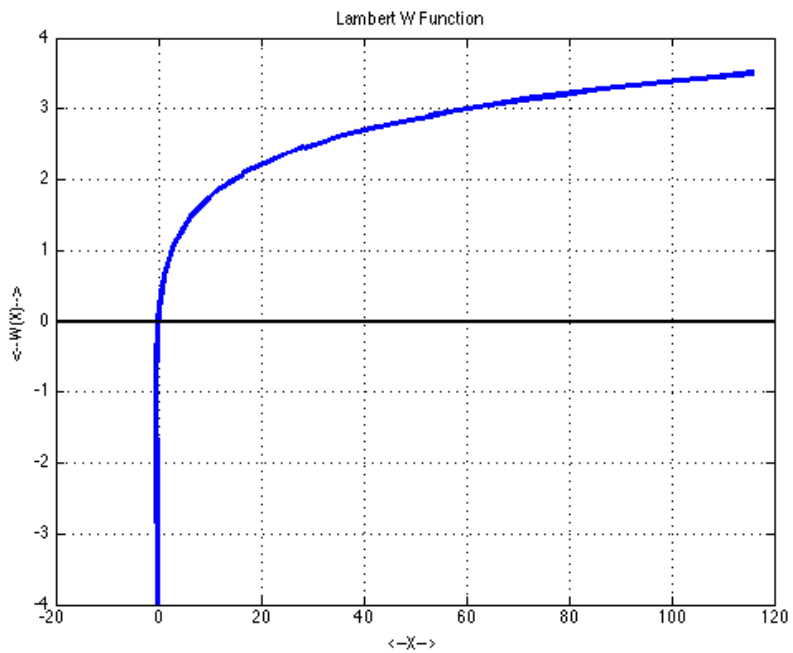
1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. Timing and Other Functions
8. PRIME Example
9. **Private and Shared Variables**
10. Reduction Operations
11. Using Random Numbers
12. SATISFY Example

PRIV: Lambert's W Function

Lambert's W function is the solution of the equation

$$w(x)e^{w(x)} = x$$

It's might not be obvious why this function needed to be invented, but there was a reason. It's really not obvious how to evaluate the function, but there is an algorithm. As you might expect, $w(x)$ is more difficult to compute than a polynomial or the sine function, but these complications are typical of what you will encounter in scientific computing. And they point out some problems that OpenMP will have to work around!



PRIV: Code to Compute W for N values of X

```
for ( i = 0; i < 8; i++ )
{
  x = x_vec[i];            <-- x_vec contains test values.
  w = x + log ( x );       <-- Initial guess for w(x).

  it = 0;

  while ( 1 )
  {
    if ( 100 < it )
    {
      break;
    }

    if ( fabs ( ( x - w * exp ( w ) ) ) <
      tol * fabs ( ( w + 1.0 ) * exp ( w ) ) )
    {
      break;
    }
    a = w * exp ( w ) - x;
    b = ( w + 1.0 ) * exp ( w )
      - ( w + 2.0 ) * ( w * exp ( w ) - x ) / ( 2.0 * w + 2.0 );
    w = w - a / b;
    it = it + 1;
  }
  printf ( "  %8.4f  %3d  %14g  %14g  %8.2e\n",
    x, it, w, w_vec[i], fabs ( w - w_vec[i] ) );   <-- Compare with exact.
}
```

PRIV: Lambert's W Function

We want to compute a table of 100,000 values for $w(x)$ over the range $1 \leq x \leq 100$. We'd like to use OpenMP to do so.

Do you see some real problems here? This computation is much messier than the "saxpy" example. In that example, the only variables that appeared on the left hand side were vector entries, indexed by the loop index. So there was no possibility of a conflict. Each loop iteration was working with completely distinct data.

Here, however, notice the variables **x, w, it, a, b** which all appear on the left hand side of equations inside the loop. These are all potential conflicts.

We presumably will fix the problem with **x** and **w** by storing them in arrays for our table. But how do we deal with **it, a, b**? Remember, if these variables each represent a single shared location in memory, then all the processors can be putting and reading stuff from the same location, which will cause total confusion!

PRIV: Code to Compute W

We are used to the loop index **i** being "private"; each process has its own copy. But now the variables **a, b, it** must also be stored this way.

```
# pragma omp parallel shared ( n, tol, w, x ) private ( a, b, i, it )
{
# pragma omp for
  for ( i = 0; i < n; i++ )
  {
    x[i] = ( ( n - i ) * 1.0 + i * 100.0 ) / n;
    w[i] = x[i] + log ( x[i] );

    it = 0;

    while ( 1 )
    {
      if ( 100 < it )
      {
        break;
      }

      if ( fabs ( ( x[i] - w[i] * exp ( w[i] ) ) ) <
        tol * fabs ( ( w[i] + 1.0 ) * exp ( w[i] ) ) )
      {
        break;
      }
      a = w[i] * exp ( w[i] ) - x[i];
      b = ( w[i] + 1.0 ) * exp ( w[i] )
        - ( w[i] + 2.0 ) * ( w[i] * exp ( w[i] ) - x[i] ) / ( 2.0 * w[i] + 2.0 );
      w[i] = w[i] - a / b;
      it = it + 1;
    }
  }
}
```

PRIV: 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.

Usually, such variables are temporary or convenience variables, whose values are not needed before or after the loop is executed.

PRIV: Private/Shared

What variables here are private?

Note that **pfun** is the name of a user function.

```
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
```

PRIV: 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**.

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

PRIV: 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
```

PRIV: 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
```

PRIV: The SAXPY Example

Remember the SAXPY example?

We didn't specify any **shared()** or **private()** information there, but the programs compiled, ran correctly, and ran faster, under OpenMP. Are there rules for when we have to specify this information?

Indeed. By default, OpenMP will assume that the index of any loop marked with a **for** or **do** directive is **private** (which it must be). In FORTRAN, it will also assume that the indices of any loops nested inside such a loop are also private - but in C/C++, it does not make this assumption.

By default, all other variables are assume to be **shared**. Therefore, although it can be useful to indicate the status of all variables, you really only have to indicate any non-loop index variables that are private, and any reduction variables.

Just for review, here is the SAXPY example, with the extra clauses that it turned out we didn't actually need.

PRIV: The SAXPY Example

```
# include <omp.h>

int main ( )
{
  int i, n = 1000;
  double s = 1.23, x[1000], y[1000];

  for ( i = 0; i < n; i++ )
  {
    x[i] = ( double ) ( ( i + 1 ) % 17 );
    y[i] = ( double ) ( ( i + 1 ) % 31 );
  }

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

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

PRIV: The SAXPY Example

```
program main

  use omp_lib

  integer, parameter :: n = 1000

  integer i
  double precision :: s = 1.23
  double precision x(n), y(n)

  do i = 1, n
    x(i) = mod ( i, 17 )
    y(i) = mod ( i, 31 )
  end do

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

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

!$omp end parallel

  stop
end
```

Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. Timing and Other Functions
8. PRIME Example
9. Private and Shared Variables
10. **Reduction Operations**
11. Using Random Numbers
12. SATISFY Example

REDUCTION: Integral of Lambert Function

Recall the Lambert function $w(x)$ and suppose that we wish to estimate the integral of this function over the range $1 \leq x \leq 100$.

For simplicity, let's also assume that we have rewritten the computation so that there is now a C or FORTRAN function called **w(x)** which carries out the evaluation of the Lambert function for any value of x.

We'll use a simple approximation that divides $[1, 100]$ into n subintervals $[a, b]$, and sums the products of $w(x)$ at the midpoints multiplied by the length of the subintervals.

$$\int_1^{100} w(x)dx \approx \sum_{i=1}^{n} w(\frac{a_i + b_i}{2}) \, (b_i - a_i)$$

REDUCTION: Is Q Private? Shared?

What kind of variable is Q? It can't be shared, because every process is trying to update it. It can't be private, because it has a value before the loop, and after the loop.

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

int main ( );
double w ( double x );

int main ( )
{
  double a;
  double b;
  int i;
  int n = 1000;
  double q;
  double x;

  q = 0.0;
  for ( i = 0; i < n; i++ )
  {
    a = ( ( n - i     ) * 1.0 + ( i     ) * 100.0 ) / n;
    b = ( ( n - i - 1 ) * 1.0 + ( i + 1 ) * 100.0 ) / n;
    x = 0.5 * ( a + b );
    q = q + ( b - a ) * w ( x );
  }

  printf ( "Q = %g\n", q );

  return 0;
}
```

REDUCTION: Q is a Reduction Variable

Q has an intermediate status between private and shared, called a **reduction variable**, because a single result is formed from contributions of all the processes. To indicate such a variable, a reduction clause must be added to the **for** or **do** loop where the variable is computed.

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

int main ( );
double w ( double x );

int main ( )
{
  double a;
  double b;
  int i;
  int n = 1000;
  double q;
  double x;

# pragma omp parallel shared ( n ) private ( a, b, i, x )

# pragma for reduction ( + : q )
  q = 0.0;
  for ( i = 0; i < n; i++ )
  {
    a = 1.0 + ( 100.0 - 1.0 ) * ( double ) ( i     ) / n;
    b = 1.0 + ( 100.0 - 1.0 ) * ( double ) ( i + 1 ) / n;
    x = 0.5 * ( a + b );
    q = q + ( b - a ) * w ( x );
  }

  printf ( "Q = %g\n", q );

  return 0;
}
```

REDUCTION: The Reduction Clause

A reduction operation occurs when you

- sum a set of numbers;
- compute the dot product of two vectors;
- compute the product of a set of numbers;
- find the maximum of a set of numbers.

The OpenMP **reduction** clause is used to indicate variables that are used to store such computations.

REDUCTION: The Reduction clause

The **reduction** clause modifies a **for** or **do** directive.

Reduction clause examples include:

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

If a variable occurs in a reduction clause, it cannot also occur in a private or shared clause for that parallel region!

Within a parallel region, a variable is private, shared or reduction.

REDUCTION: Norm Example

A dot product is another example of reduction:

```
# pragma omp parallel private ( i ) shared ( n, x, y )
  dot = 0.0;
  # pragma for reduction ( + : dot )
  for ( i = 0; i < n; i++ )
  {
    dot = dot + x[i] * y[i];
  }
```

Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. Timing and Other Functions
8. PRIME Example
9. Private and Shared Variables
10. Reduction Operations
11. **Using Random Numbers**
12. SATISFY Example

RANDOM: Lambert Integral by Monte Carlo

Now suppose that we wish to approximate the integral of the Lambert W function from 1 to 100, but instead of using a quadrature rule, we want to use a Monte Carlo approach.

For this simple problem, the advantages of a Monte Carlo approach aren't obvious, but for problems with complicated geometries, or probabilistic components, or simulations, sometimes this is the only way to do computations.

Our approximation selects n sample points x_i from $[1, 100]$ at random, and computes the estimate as

$$\int_1^{100} w(x)dx \approx \frac{100-1}{n} \sum_{i=1}^{n} w(x_i)$$

REDUCTION: The Sequential Code

Each time we call **drand()**, we get a new sample point. We also get a new value of **seed**, which is actually what controls the computation.

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

int main ( );
double w ( double x );
double drand ( int *seed );

int main ( )
{
  double a;
  double b;
  int i;
  int n = 1000;
  double q;
  int seed = 123456789;
  double u;
  double x;

  q = 0.0;
  for ( i = 0; i < n; i++ )
  {
    u = drand ( &seed );          <-- some random number generator.
    x = 1.0 + ( 100.0 - 1.0 ) * u;
    q = q + w ( x );
  }
  q = q * ( 100.0 - 1.0 ) / n;
  printf ( "Q = %g\n", q );

  return 0;
}
```

RUN: Parallel Execution Needs Multiple Seeds

If we want this calculation to run in parallel, then we want each process to be able to call **drand()** and get a distinct set of random numbers. That means each process needs a separate, distinct, private value of **seed**. Let's call this quantity **my_seed**.

A simple idea would be to set

```
my_seed = seed + id
```

where **seed** is our original seed value and **id** is the identifier or index of the process. This will give us distinct seeds.

This idea will work, but look carefully at the details in the revised code:

REDUCTION: Changes for Parallel Code

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

int main ( );
double w ( double x );
double drand ( int *seed );

int main ( )
{
  double a;
  double b;
  int i;
  int my_seed;
  int n = 1000;
  double q;
  int seed = 123456789;
  double u;
  double x;

  q = 0.0;
# pragma omp parallel shared ( n, seed ) private ( i, my_seed, u, x )
{
  my_seed = seed + omp_thread_num ( );  <-- omp_thread_num() gets the id.

# pragma omp for reduction ( q : + )
  for ( i = 0; i < n; i++ )
  {
    u = drand ( &my_seed );
    x = 1.0 + ( 100.0 - 1.0 ) * u;
    q = q + w ( x );
  }
}
  q = q * ( 100.0 - 1.0 ) / n;
  printf ( "Q = %g\n", q );

  return 0;
```

Shared Memory Programming with OpenMP

1. Serial Programs Can't Accelerate
2. Parallel Algorithms Are Available
3. The Parallel Loop
4. SAXPY Example: Vector Addition
5. Basic OpenMP Directives
6. Compiling, Linking, Running
7. Timing and Other Functions
8. PRIME Example
9. Private and Shared Variables
10. Reduction Operations
11. Using Random Numbers
12. **SATISFY Example**

SATISFY: The Logical Satisfaction Problem

Logicians and electric circuit designers both worry about the logical satisfaction problem. When logicians describe this problem, they assume that they have n logical variables b_1 through b_n, each of which can be **false=0** or **true=1**.

Moreover, they have a formula involving these variables, as well as logical operators such as **and**, **or**, **not**.

```
f = b_1 AND ( b_2 OR ( b_1 AND NOT b_5 ) OR ...
```

Their simple question is, what values of the variables b will make the formula have a value that is **true**, that is, what values satisfy this formula?

SATISFY: The Brute Force Approach

While there are some cases of the satisfaction problem that allow for an efficient approach, there is an obvious brute force approach that has three advantages:

- it always finds all the solutions;
- it is easy to program;
- it is easy to do in parallel.

As you can imagine, the brute force approach is brutally simple:

1. generate every possibility
2. check if it's a solution

We can easily parallelize this approach if we can figure out how to generate all the possibilities, and divide this up among the processors.

SATISFY: Logical Vector = Binary Number

But we are searching all possible logical vectors of dimension n. A logical vector, containing TRUE or FALSE, can be though of as a binary vector, which can be thought of as an integer:

```
FFFF = (0,0,0,0) = 0
FFFT = (0,0,0,1) = 1
FFTF = (0,0,1,0) = 2
FFTT = (0,0,1,1) = 3
...
TTTT = (1,1,1,1) = 15
```

so we can parallelize this problem by computing the total number of possibilities, which will be 2^n, and dividing up the range among the processors.

SATISFY: Logical Vector = Binary Number

For example, we might have 1,024 possibilities, and 10 processors.

You should convince yourself that it is worthwhile figuring out a formula that a formula that will automatically compute the ranges for you:

```
lo = floor ( (   p       * 1024 ) / 10 )
hi = floor ( ( ( p + 1 ) * 1024 ) / 10 )
```

giving us the table:

Processor P	Start	Stop before
0	0	102
1	102	204
2	204	307
3	307	409
...	...	...
9	921	1024

SATISFY: A Simple C Code

```
/*
  Compute the number of binary vectors to check.
*/
  ihi = 1;
  for ( i = 1; i <= n; i++ )
  {
    ihi = ihi * 2;
  }
  printf ( "\n" );
  printf ( "  The number of logical variables is N = %d\n", n );
  printf ( "  The number of input vectors to check is %d\n", ihi );
  printf ( "\n" );
  printf ( "   #       Index    ---------Input Values-------------------------\n" );
  printf ( "\n" );
/*
  Check every possible input vector.
*/
  solution_num = 0;

  for ( i = 0; i < ihi; i++ )  Make this loop parallel!
  {
    i4_to_bvec ( i, n, bvec );

    value = circuit_value ( n, bvec );

    if ( value == 1 )
    {
      solution_num = solution_num + 1;

      printf ( "  %2d  %10d:  ", solution_num, i );
      for ( j = 0; j < n; j++ )
      {
        printf ( " %d", bvec[j] );
      }
      printf ( "\n" );
    }
  }
```

SATISFY: Issues

What issues do we face in creating a parallel version?

- We must define a range for each processor;
- The only input is **n**;
- Each processor works completely independently;
- Each processor prints out any solution it finds;
- The only common output variable is **solution_num**.

SATISFY: A Simple C/OPENMP Code

```
# pragma omp parallel \
  shared ( ihi, n, thread_num ) \
  private ( bvec, i, id, ihi2, ilo2, j, solution_num_local, value ) \
  reduction ( + : solution_num )
  {
    id = omp_get_thread_num ( );

    ilo2 = (   id       * ihi ) / thread_num;
    ihi2 = ( ( id + 1 ) * ihi ) / thread_num;
/*
  Check every possible input vector.
*/
    solution_num_local = 0;

    for ( i = ilo2; i < ihi2; i++ )
    {
      i4_to_bvec ( i, n, bvec );
      value = circuit_value ( n, bvec );
      if ( value == 1 )
      {
        solution_num_local = solution_num_local + 1;
        printf ( "  %2d  %8d  %10d:  ", solution_num_local, id, i );
        for ( j = 0; j < n; j++ )
        {
          printf ( " %d", bvec[j] );
        }
        printf ( "\n" );
      }
    }
    solution_num = solution_num + solution_num_local;
  }
```

CONCLUSION

OpenMP has a simple set of rules that allow a programmer to experiment on individual program loops.

The programmer must be careful to identify a parallel region, to mark the loops in that region that are to be run in parallel, and to classify the variables within the parallel region.

A program can be compiled with or without the OpenMP option.

A program can be run with 1, 2, or many OpenMP threads.

Program improvement is measured by comparing wall clock time.

Lab exercises will ask you to run or modify several OpenMP programs.