Load balancing vs Communication
This is the eternal problem in parallel computing. The basic approaches to this problem include:

▶ Data partitioning - moving different parts of the data set across several nodes
▶ Task partitioning - give separate tasks to different nodes
Definition of Terms

- node - a box usually containing processors, local memory, disks and network connection
- cluster - a group of nodes networked together
- speedup: \( S_p = \frac{T_1}{T_p} \)
- efficiency: \( \frac{S_p}{p} = \frac{T_1}{pT_p} \)

\( T_i \) is the execution time for \( i \) processors, \( p \) is the number of processors
Speedup

- Adding more processors does not always improve the speed a code runs.
- Usually, better speedup can be found by increasing the problem size, at least to a point.
- The non-parallel part of a code generally scales linearly with the problem size. The parallel part usually scales as problem size to some power.
- Generally increasing the problem size without increasing the node number helps performance.
Good parallel algorithms run faster when more nodes are available. In the best case, doubling the number of nodes decreases the execution time by a factor of two. One way to consider scaling of a code is Amdahl’s law

\[
\frac{1}{\alpha + \frac{1-\alpha}{p}}
\]

where \( \alpha \) is the portion of the code which cannot be parallelized and \( p \) is the number of processors. This is a simplification, but *Speedup is limited by the slowest portion of the code.*
Amdahl’s Law
Communication between nodes takes a great deal of time.

Typically you can do thousands of computation in the time it takes to pass the simplest message.

The time it takes for a message to be passed is limited by bandwidth $b$ and latency $l$. To pass a message of size $s$, you need

$$\frac{s}{b} + l$$

(Assuming $b$, $l$, and $s$ are in consistent units.)
Introduction to OpenMP

OpenMP is as a set of simple program additions to make codes run efficiently on shared memory computers. The formal API for OpenMP is only about 50 pages long, and contains compiler directives and library functions.

http://www.llnl.gov/computing/tutorials/openMP/
OpenMP Threads

OpenMP uses threads for parallel programming

- Forks and joins are used for most of the internal programming
- Speedup is achieved by the operating system splitting the threads across multiple CPUs.
- New threads are created explicitly by the program directives dynamically.
Forks and Joins
Goals of OpenMP - from LLNL

- Standardization
- Lean and Mean - only 3-4 directives
- Ease of use
- Portability - F77, F90, F95, C, C++
OpenMP Programming Model - from LLNL

- Shared Memory, thread based
- Explicit Parallelism
- Fork-Join Model
- Compiler Directives
- Nested Parallelism Support - in most implementations
- Dynamic Threads
- Not tied to I/O
Explicit Parallelism

- You must tell the computer what sections of code to parallelize using compiler directives.
- The compiler directives vary between languages, but are ignored when OpenMP flags are not set with the compiler.
- Codes written with OpenMP can run easily on serial machines.
Environment and Library Routines

- Some environmental variables are needed to make the code execute using the correct number of threads
- Some library routines allow the programmer to set and access system variables
Not Message Passing

This is NOT a set of message passing routines. Instead, you give directives to the compiler of what parts of the code can be executed in parallel. In some ways, OpenMP is a set of directives to tell the compiler how to more efficiently handle loops.
General Syntax

Fortran:

!$OMP <directive>
   do useful stuff
!$OMP end <directive>

C/C++:

#pragma omp <directive-name> clause
   {do useful stuff in a
    structured block}
A Trivial Example

Basic Code

```fortran
program trivial
  print *, 'Hello World!'
end program
```

% gfortran trivial.f90
% ./a.out
Hello World!

OMP Additions

```fortran
program trivial

  !$OMP PARALLEL
  print *, 'Hello World!'
  !$OMP END PARALLEL

end program trivial
```

% gfortran trivialOpenMP.f90
% ./a.out
Hello World!

What went wrong?
Execution of the Trivial Example

% gfortran trivialOpenMP.f90 -fopenmp
% ./a.out
   Hello World!
   Hello World!
   Hello World!
   Hello World!
   Hello World!
   Hello World!
   Hello World!
% export OMP_NUM_THREADS=3
% ./a.out
   Hello World!
   Hello World!
   Hello World!
   Hello World!
   Hello World!
   Hello World!
program trivial1
    implicit none
    integer :: OMP_GET_THREAD_NUM, OMP_GET_MAX_THREADS
    integer :: tid, nthreads
!$OMP PARALLEL PRIVATE(nthreads, tid)
    tid = OMP_GET_THREAD_NUM()
    nthreads = OMP_GET_MAX_THREADS()
    print *, 'Hello World! from ', tid, nthreads
!$OMP END PARALLEL
end program

Note the PRIVATE key word, indicating that all threads have their own copy of the variable.
% gfortran -fopenmp trivial1.f90
% ./a.out
Hello World! from 0 1
Hello World! from 2 1
Hello World! from 3 1
Hello World! from 4 1
Hello World! from 1 1
Hello World! from 7 1
Hello World! from 5 1
Hello World! from 6 1
program trivial2
  implicit none
  integer :: OMP_GET_THREAD_NUM, OMP_GET_MAX_THREADS
  integer :: tid, nthreads
  nthreads = OMP_GET_MAX_THREADS()
!$OMP PARALLEL PRIVATE(tid)
  tid = OMP_GET_THREAD_NUM()
  print *, 'Hello World! from ', tid, nthreads
!$OMP END PARALLEL
end program

Note that nthreads is outside of the OMP directives
Thread ID (2)

% gfortran -fopenmp trivial2.f90
% ./a.out

Hello World! from 5 8
Hello World! from 0 8
Hello World! from 1 8
Hello World! from 2 8
Hello World! from 7 8
Hello World! from 3 8
Hello World! from 4 8
Hello World! from 6 8
Parallelizing Loops

To parallelize a loop, you need to help the compiler figure out the most efficient way to use threads. There are simple defaults, but giving it more details can help efficiency. The basic directives are:

```c
!$OMP PARALLEL
!$OMP DO

some parallel loop

!$OMP END DO
!$OMP END PARALLEL
```
A Simple OMP Example

omptest1

program omptest1
    integer, parameter :: n = 10000
    integer, parameter :: dble = selected_real_kind(15,307)
    real(kind=dble), dimension(n) :: a
    integer :: i, j

    !$OMP PARALLEL
    !$OMP DO
    do j = 1, 100000
        do i = 1, n
            a(i) = log(real(i)) + j
        enddo
    enddo
    !$OMP END DO
    !$OMP END PARALLEL
    print *, a(1)
end program omptest1
Results

omptest1

![Chart showing the relationship between OMP_NUM_THREADS and Seconds for 'real' values. The y-axis represents Seconds ranging from 0 to 40, and the x-axis represents OMP_NUM_THREADS ranging from 0 to 8. The chart shows a decrease in Seconds as the number of threads increases, stabilizing at around 5 Seconds for 5 or more threads.]
Results

omptest1

- Graph showing the relationship between OMP_NUM_THREADS and Seconds for 'real' and 'user' time.
Results

omptest1

<table>
<thead>
<tr>
<th>OMP_NUM_THREADS</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>ideal</td>
</tr>
</tbody>
</table>
Results

omptest1

![Graph showing efficiency versus OMP_NUM_THREADS]
Combining Directives

You do not have to have a separate directive on each line. For example,

```plaintext
!$OMP PARALLEL
!$OMP DO
!$OMP PRIVATE(NTHREADS, TID)
```

Becomes

```plaintext
!$OMP PARALLEL DO PRIVATE(NTHREADS, TID)
```
Because the loops are executing separately, you may wish to combine the results from different threads to a final answer. You need to use reduction to make this work.

```bash
$!OMP PARALLEL PRIVATE(X) REDUCTION(+:SUM)
```
Numerical Integration

\[ y = \frac{4}{1+x^2} \]
Numerical Integration

Integrating

\[
\pi = \int_{-1/2}^{1/2} \frac{4}{1 + x^2} \, dx
\]  

(1)

We can approximate this integral using Simpson’s algorithms

- Input the number of partitions to be used
- Divide the domain into \( n \) partitions
- Evaluate the function at each partition
- Multiply the function evaluation times the width of the function to find a differential area
- Add the differential areas together
- Output the result
Parallel Integration

In parallel, the problem is nearly the same.

- Have processing element (PE) zero, get the number of partitions
- Determine the number of PEs: \( m \)
- Divide the domain into \( \frac{n}{m} \) partitions on each PE
- Evaluate the function at each partition
- Multiply the function evaluation times the width of the function to find a differential area
- Add the differential areas together across all the PEs
- On PE zero, output the result
Simple Code to Calculate PI

```fortran
program reduce
  integer :: i, num_steps
  double precision :: x, pi, step, sum
  sum = 0.0d0; nsteps = 10000
  step = 1.0d0 / double(nsteps)
  do i = 1, nsteps
    x = (double(i) + 0.5d0) * step
    sum = sum + 4.0d0 / (1.0d0 + x**2)
  enddo
  pi = step * sum
  print *, "Estimate of Pi with ", nsteps, " steps is ", pi
end program reduce
```

$ ./reduce
  Estimate of Pi with 10000 steps is 3.1413926444243838
Simple Code to Calculate PI

```fortran
program reduceOMP
    integer :: i, num_steps
    double precision :: x, pi, step, sum
    sum = 0.0d0; num_steps = 10000
    step = 1.0d0 / double(num_steps)
!$OMP PARALLEL DO
    do i = 1, num_steps
        x = (double(i) + 0.5d0) * step
        sum = sum + 4.0d0 / (1.0d0 + x**2)
    enddo
!$OMP END PARALLEL DO
    pi = step * sum
    print *, "Estimate of Pi with ", num_steps, " steps is ", pi
end program reduceOMP
```

$ gfortran -fopenmp reduceOMP.f90 -o reduceOMP
$ ./reduceOMP
    Estimate of Pi with 10000 steps is 7.558335781770253

What happened?
OpenMP Modifications

```fortran
program reduceOMP2

  integer :: i, num_steps
  double precision :: x, pi, step, sum
  sum = 0.0d0 ; nsteps = 100000000
  step = 1.0d0 / double(nsteps)

  !$OMP PARALLEL DO PRIVATE(X) REDUCTION(+:SUM)
  do i = 1, nsteps
    x = (double(i) + 0.5d0) * step
    sum = sum + 4.0d0 / (1.0d0 + x*x)
  enddo

  !$OMP END PARALLEL DO
  pi = step * sum

  print *, "Estimate of Pi with ", nsteps, " steps is ", pi
end program reduceOMP2
```

```
$ gfortran -fopenmp reduceOMP2.f90 -o reduceOMP2
$ ./reduceOMP2
  Estimate of Pi with 10000 steps is 3.1413926444243732
```
Results
reduceOMP2

![Graph showing the relationship between OMP_NUM_THREADS and Seconds with a downward trend.]
Results
reduceOMP2

Graph showing the relationship between OMP_NUM_THREADS and time in seconds, labeled as 'real' and 'user'.
Results

reduceOMP2

![Graph showing speedup against OMP_NUM_THREADS]

- Speedup increases with the number of threads
- The ideal line indicates perfect speedup
- The graph shows the actual speedup compared to the ideal speedup
Results

reduceOMP2

![Graph showing the relationship between Efficiency and OMP_NUM_THREADS.](image)
Loop Splitting

One of the key ideas to remember is that loops often contain several operations that can be split. Taking an example from the Patterns in Parallel Programming book, imagine we have a loop with two functions:

- **BIG COMPUTATION** - a big computation the executes independently on each element in the loop
- **COMBINE** - an element that cannot be parallelized and must execute in order
Loop Splitting

do i = 1, nsteps
    x = BIG_COMPUTATION(i)
    call COMBINE(x,answer)
enddo

can be split into

do i = 1, nsteps
    x(i) = BIG_COMPUTATION(i)
enddo
do i = 1, nsteps
    call COMBINE(x(i),answer)
enddo
Using OpenMP in Loop Splitting

!$OMP PARALLEL DO PRIVATE(I)
do i = 1, nsteps
  x(i) = BIG_COMPUTATION(i)
enddo
!$OMP END PARALLEL DO

!$OMP PARALLEL DO
  do i = 1, nsteps
    call COMBINE(x(i),answer)
  enddo
Controlling Loops

There are many options for controlling the execution of threads.

```latex
%!$OMP DO SCHEDULE(TYPE,integer)
```

- `schedule(static[,chunk])` - groups of size `chunk` statically assigned in a round-robin fashion
- `schedule(dynamic[,chunk])` - threads dynamically grab work as it is completed
- `schedule(guided[,chunk])` - chunk size is reduced automatically during iteration toward a minimum level of `chunk`
- `schedule(runtime)` - checks the OMP_SCHEDULER environmental variable
integer, parameter :: chunk = 10

 !$OMP PARALLEL PRIVATE(i,j,z,c,it) DEFAULT(SHARED)
 !$OMP DO SCHEDULE(DYNAMIC,CHUNK)
   do i = 1, n
     do j = 1, n
       ...

...
Controlling Loops

setenv OMP_SCHEDULE static
11.477u 0.012s 0:08.24 139.3%

setenv OMP_SCHEDULE dynamic
11.239u 0.006s 0:05.67 198.0%

setenv OMP_SCHEDULE guided
11.453u 0.005s 0:06.52 175.6%

setenv OMP_SCHEDULE static,20
11.439u 0.028s 0:05.89 194.3%

no omp
11.280u 0.004s 0:11.28 100.0%