Intermediate Programming Workshop

Basic Types of Parallelism

1. Threading- OpenMP, POSIX threads
   Often the simplest.
   Works primarily on loops.
   All processors involved must be able to see all memory used in the job.

2. Devices.  GPU's, Knights Landing
   Similar to old style co-processors.
   Memory management is a high barrier (must manage device memory transfers).

3. Message Passing (MPI)
   Workhorse of parallel computing.
   Allows distributing single job on arbitrary number of processors.
   No memory is shared, even on the same node.  All data must be sent via messages.

MPI Example

Basic Hello World code:

```fortran
program main
    USE MPI
    call MPI_INIT(ier)
    call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ier)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier)
    do i=0,numprocs
        print *, "Proc Num ", myid," Says Hello to Your World!!"
    enddo
    call MPI_FINALIZE(ier)
end
```

Almost all MPI codes have these basic calls.

Example serial code to calculate pi

```fortran
program compute_pi
!
! Karst compile line:  ifort -integer-size 64 -o pi.pi.f90
!
integer ::  i
integer(kind = 8) :: interval
real(kind=8) ::  width, partial, sum, pi
```
Example code threaded with OpenMP

program compute_pi

    ! USE OMP_LIB
    ! Karst compile line: ifort -integer-size 64 -openmp -par-num-threads=6 -o omp_pi omp_pi.f90
    !
    integer :: i
    integer(kind = 8) :: interval
    real(kind=8) :: width, partial, sum, pi
    integer, parameter :: nthreads = 6
    
    call OMP_SET_NUM_THREADS(nthreads)
    
    ! Sum of intervals equals one
    !
    interval = 2000000000000
    width = 1.d0 / interval
    sum = 0.d0
    !$OMP PARALLEL DO PRIVATE(i, partial), SHARED(width), REDUCTION(+: sum)
    do i = 1, interval
        partial = width * (i - 0.5d0)
        sum = sum + 4.d0 / (1.d0 + partial * partial)
    enddo
    pi = width * sum
    !$OMP END PARALLEL DO
    print *, "computed pi =", pi
    print *, "reference pi = 3.1415926535897932385"
    stop
end program compute_pi
Example code with MPI

    program compute_pi

    ! Karst compile line:  mpif90 -integer-size 64 -lmpi -o mpi_pi mpi_pi.f90
    !
    USE MPI
    !
    integer :: i
    integer(kind = 8) :: interval
    real(kind=8) :: width, partial, sum, pi
    integer :: ier, myid, numprocs
    !
    call MPI_INIT(ier)
    call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ier)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier)
    |
    interval = 2000000000000
    width = 1.d0 / interval
    sum = 0.d0
    do i = 1, interval
        partial = width * (i - 0.5d0)
        sum = sum + 4.d0 / (1.d0 + partial * partial)
    enddo
    pi = width * sum
    if (myid .EQ. 0) then
        print *, "computed pi =", pi
        print *, "reference pi = 3.1415926535897932385"
    endif
    call MPI_FINALIZE(ier)
    stop
end program compute_pi

This code gets the correct answer, but takes longer than the serial code. Why?

Each MPI rank acts independently. In effect, they are each running their own serial code. Must divide up the work and then combine the partial solutions each rank calculates at the end. The OpenMP version kept private copies of sum and then reduced them. In MPI, we need a separate variable for the partial results and a directive telling it where to combine them.

How do we separate the job?

The bulk of the job is processing each of the intervals. The simplest method is to simply divide up sections in proportion to the number of processors.
A better version

program compute_pi
!
! Karst compile line: mpif90 -integer-size 64 -Impi -o mpi_pi mpi_pi.f90
!
USE MPI
!
integer :: i
integer(kind = 8) :: interval
real(kind=8) :: width, partial, sum, pi
integer :: ier, myid, numprocs
integer :: local_interval, mybottom, mytop
real(kind=8) :: mypi
!
call MPI_Init(ier)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier)
!
interval = 2000000000000
local_interval = interval / numprocs
mybottom = (myid * local_interval) +1
mytop = (mybottom + local_interval) - 1
width = 1.d0 / interval
sum = 0.d0
do i = mybottom, mytop
    partial = width * (i - 0.5d0)
    sum = sum + 4.d0 / (1.d0 + partial * partial)
enddo
mypi = width * sum
!
call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0, MPI_COMM_WORLD,ier)
if (myid .EQ. 0) then
    print *, "computed pi =", pi
    print *, "reference pi = 3.1415926535897932385"
endif
!
call MPI_FINALIZE(ier)
stop
end program compute_pi

A more involved example

In the pi example, we knew the size of the intervals and the work to be done in each one at compile time. This helps equally divide the work between the ranks. Most problems are not that predictable.

Consider the function, \( y = \sin(x) \) between 0 and \( \pi \):
Let's add an attenuating force along the y axis. We will consider it as a step function where the force at zero is equal to one and the force at one is equal to zero.

```fortran
program compute_sin

! Karst compile line:  mpif90 -integer-size 64 -lmpi -o mpi_sin mpi_sin.f90
!
include "mpif.h"
!
integer :: i, interval, myy, mypos
real(kind=8) :: y, x, mysum, force, mysin, sum, temp
real(kind=8) :: realpos, realinterval
integer(kind = 4) :: ier, myid, numprocs
integer :: local_interval, mybottom, mytop
real(kind=8) :: normal=3141.6
!
call MPI_INIT(ier)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier)
!
interval = 314159265358
local_interval = interval / numprocs
mybottom = (myid * local_interval) +1
mytop = (mybottom + local_interval) - 1
sum = 0.d0
mysum = 0.d0
do i = mybottom, mytop
  realpos = real(i)
  realinterval = real(interval)
  temp = (realpos/realinterval) * 3.1416
  y = sin(temp) * 1000.
  myy = int(y)
  col_sum = 0.d0
  force = 1.d0
  do j = 1, myy
```
col_sum = (myy * force) + col_sum 
force = force - (real(j)/normal) 
enddo 
mysum = mysum + col_sum 
enddo 
call MPI_REDUCE(mysum,sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,0, MPI_COMM_WORLD,ier) 
if (myid .EQ. 0) then 
    print *, "computed  force =", sum 
endif 
call MPI_FINALIZE(ier) 
stop 
end program compute_sin 

Timing results when run on 48 processors: 

real  51m25.455s 
user  448m56.059s 
sys   42m49.484s 

A better distribution 

As you can see from the graph, the work is uneven. The ranks on the extremes have much less work to do than the central ranks. So, let’s try interleaving the columns of work between the ranks.

program compute_sin 
| 
! Karst compile line: Intel: mpif90 -integer-size 64 -lmpi -o better_sin better_sin.f90 
! GNU: mpif90 -integer-size 64 -lmpi -o better_sin better_sin.f90 
! 
!USE MPI 
include "mpif.h" 
| 
integer :: i, interval, myy, mypos 
real(kind=8) :: y, x, mysum, force, mysin, sum, temp 
real(kind=8) :: realpos, realinterval 
integer(kind = 4) :: ier, myid, numprocs 
real(kind=8) :: normal=3141.6 
| 
call MPI_INIT(ier) 
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ier) 
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier) 
| 
interval = 31415926535 
sum = 0.d0 
mysum = 0.d0 
do i= myid + 1, interval, numprocs
realpos = real(i)
realinterval = real(interval)
temp = (realpos/realinterval) * 3.1416
y = sin(temp) * 1000.
myy = int(y)
col_sum = 0.d0
force = 1.d0
    do j = 1, myy
        col_sum = (myy * force) + col_sum
        force = force - (real(j)/normal)
    enddo
    mysum = mysum + col_sum
enddo
call MPI_REDUCE(mysum,sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,MPI_COMM_WORLD,ier)
if (myid .EQ. 0) then
    print *, "computed  force =", sum
endif
call MPI_FINALIZE(ier)
stop
end program compute_sin

Results when run on 48 processors:

With the GNU compilers, a “normal compile” (above) returned a value and walltime of:
computed  force = -6.0785624158003328E+017
real 38m23.821s

Retrying and adding the –O2 compile switch to the compile line above returned:
computed  force = -6.0785624158003328E+017
real 32m51.685s

With the Intel Compilers, a “normal compile” (above) returned a value and walltime of:
computed  force = -6.078562415800333E+017
real 32m49.774s

Retrying and adding the –O2 compile switch to the compile line above returned:
computed  force = -6.078562415800333E+017
real 35m26.457s

OpenMP, Modern POSIX Threads

OpenMP is a directive based method of implementing POSIX threads. Unlike MPI, every variable, including data is shared between every thread, unless explicitly excluded. This is both an advantage in simplifying the effort, but also requires limiting parallelism to processors sharing a common memory.
program compute_sin
!
Karst compile line: Intel: ifort -openmp -integer-size 64 -o omp_sin omp_sin.f90
GNU: gfortran -fopenmp -fdefault-integer-8 -o omp_sin omp_sin.f90
!
USE OMP_LIB
!
integer :: i, interval, myy, mypos
integer, parameter :: nthreads = 16
real(kind=8) :: y, x, force, col_sum, sum, temp
real(kind=8) :: realpos, realinterval
real(kind=8) :: normal=3141.6
!
call OMP_SET_NUM_THREADS(nthreads)
interval = 3141592653
sum = 0.d0
!
!$OMP PARALLEL DO PRIVATE(i), REDUCTION(+:sum)
do i = 1, interval
   realpos = real(i)
   realinterval = real(interval)
   temp = (realpos/realinterval) * 3.1416
   y = sin(temp) * 1000.
   myy = int(y)
   col_sum = 0.d0
   force = 1.d0
   do j = 1, myy
      col_sum = (myy * force) + col_sum
      force = force - (real(j)/normal)
   enddo
   sum = sum + col_sum
enddo
!$OMP END PARALLEL DO
if (myid .EQ. 0) then
   print *, "computed force =", sum
endif
stop
end program compute_sin

Timing stats:
real  395m46.782s
user  4657m43.860s
sys   0m12.276s

This timing appears to be almost the same as a serial construct.
Now let’s privatize the variables by changing our parallel do command to:

!$OMP PARALLEL DO PRIVATE(i, temp, col_sum, force, y, realpos, realinterval, myy),
   REDUCTION(+:sum)
We rerun the code and get dramatically different results:

**New Timing Stats:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>15m18.805s</td>
</tr>
<tr>
<td>user</td>
<td>157m27.589s</td>
</tr>
</tbody>
</table>

OpenACC

OpenACC’s similarity to OpenMP is on purpose. OpenACC supports the equivalent of launching a hierarchy of threads on a remote device. Since the host may be its own device, a corollary to OpenMP may syntactically constructed. However, OpenACC’s primary use is to off load work to devices like GPU’s or Phi processors. Not all compilers support OpenACC because Intel sees it as a threat to their processor road map and actively discourages support from its vendor partners. We have both PGI and GNU compilers which support OpenACC.

    program compute_sin
    USE MPI
    USE OPENACC
    include "mpif.h"
    integer :: i, interval, myy, mypos
    integer, parameter :: nthreads = 4
    real(kind=8) ::  y, x, mysum, force, mysin, sum, temp
    real(kind=8) :: realpos, realinterval
    integer(kind = 4) :: ier, myid, numprocs
    real(kind=8) :: normal=3141.6
    !
    call MPI_INIT(ier)
    call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ier)
    call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier)
    !
    call OMP_SET_NUM_THREADS(nthreads)
    interval = 3141592653
    sum = 0.d0
    mysum = 0.d0
    do i= myid + 1, interval, numprocs
        realpos = real(i)
        realinterval = real(interval)
        temp = (realpos/realinterval) * 3.1416
        y = sin(temp) * 1000.
        myy = int(y)
col_sum = 0.d0
force = 1.d0

!$acc parallel loop reduction(+: col_sum)
do j = 1, myy
  col_sum = (myy * force) + col_sum
  force = force - (real(j)/normal)
enddo
!$acc end parallel loop
mysum = mysum + col_sum
enda do

call MPI_REDUCE(mysum,sum,1,MPI_DOUBLE_PRECISION,MPI_SUM,0, MPI_COMM_WORLD,ier)
if (myid .EQ. 0) then
  print *, "computed force =", sum
endif

Timing info to come...

Mixing MPI and OpenMP

Often, OpenMP is used inside MPI to operate on expensive DO (for) loops. Usually, the timing is actually slower than if it were possible to convert the threaded cores into MPI ranks. However, this is not always true. The key is to keep them all busy. An example in our code:

program compute_sin
  !
! Karst compile line: Intel: ifort -openmp -integer-size 64 -o omp_sin omp_sin.f90
!                        GNU: gfortran -fopenmp -fdefault-integer-8 -o omp_sin omp_sin.f90
! USE OMP_LIB
!
integer :: i, interval, myy, mypos
integer, parameter :: nthreads = 16
real(kind=8) :: y, x, force, col_sum, sum, temp
real(kind=8) :: realpos, realinterval
real(kind=8) :: normal=3141.6
!
call OMP_SET_NUM_THREADS(nthreads)
interval = 3141592653
sum = 0.d0
!$OMP PARALLEL DO PRIVATE(i), REDUCTION(+:sum)
do i= 1, interval
  realpos = real(i)
  realinterval = real(interval)
  temp = (realpos/realinterval) * 3.1416
  y = sin(temp) * 1000.
myy = int(y)
col_sum = 0.d0
force = 1.d0
do j = 1, myy
    col_sum = (myy * force) + col_sum
    force = force - (real(j)/normal)
endo
col_sum = sum + col_sum
endo
!$OMP END PARALLEL DO
if (myid .EQ. 0) then
    print *, "computed force =", sum
endif
stop
end program compute_sin

NOTE: This looks to be the most inviting form of parallelization. However, often the threads are
often under worked compared to expanding the MPI rank count. Note the time to run:
GNU unoptimized:  real  62m27.168s
Optimized with –O2:  real  60m33.625s

The PBS file only asks for the MPI ranks. Threads must be accounted for or else there will be
overloading of the CPU’s:
#PBS -l nodes=3:ppn=4,walltime=2:00:00
#PBS -m ae
#PBS -N test_pbs
#PBS -o mixed_sin.out
#PBS -e mixed_sin.err
#
cd $PBS_O_WORKDIR
#
time mpirun -np 12 -machinefile $PBS_NODEFILE mixed_sin

Sending and Receiving arbitrary pieces of work.

In the previous codes, we knew the data layout ahead of time. Often this is not true. One rank acts
like a manager and sends work to the rest of the universe. We can emulate that with our example.
Instead of preassuming which work a rank should do, rank 0 will pass work out with an MPI_SEND and
will collect partial results with an MPI_Recv. The workers will operate in reverse, they will receive
work and send their results.

MPI_SEND(variable_to_send, size, datatype, destination, tag, MPI_COMM_WORLD, error_term)

MPI_RECV(variable_to_send, size, datatype, sender, tag, MPI_COMM_WORLD, error_term)
To implement these in our code, an example would be:

```fortran
program compute_sin
!
! Karst compile line:
GNU:  mpif90 -default-integer-8 -o random_sin random_sin.f90
!
Intel:  mpif90 -integer-size 64 -lm -o random_sin random_sin.f90
!
!! USE MPI
include "mpif.h"
!
integer :: i, count, kill, interval, myy, mypos, work
real(kind=8) :: y, x, mysum, force, mysin, sum, temp
real(kind=8) :: realpos, realinterval
integer(kind=4) :: ier, myid, numprocs, new_source, new_tag
integer(kind=4) :: status(MPI_STATUS_SIZE)
real(kind=8) :: normal=3141.6
!
call MPI_INIT(ier)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ier)
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ier)
!
interval = 314159
count = 1
work = 1
kill = -1
sum = 0.d0
mysum = 0.d0
if (myid .EQ. 0) then
    do i= 1, numprocs - 1
        call MPI_SEND(count, 1, MPI_INTEGER, i, i, MPI_COMM_WORLD, ier)
        count = count + 1
    enddo
endif
if (myid .EQ. 0) then
    do while (work .LE. interval)
        call MPI_RECV(mysum, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, status, ier)
        new_source = status(MPI_SOURCE)
        new_tag = status(MPI_TAG)
        work = work + 1
        sum = sum + mysum
        if (count .LE. interval) then
            call MPI_SEND(count, 1, MPI_INTEGER, new_source, new_tag, MPI_COMM_WORLD, ier)
            count = count + 1
        else
            call MPI_SEND(kill, 1, MPI_INTEGER, new_source, new_tag, MPI_COMM_WORLD, ier)
        endif
    enddo
endif
```
enddo
else
mycount = 0
call MPI_RECV(mycount, 1, MPI_INTEGER, 0, myid, MPI_COMM_WORLD, status, ier)
do while  (mycount .NE. kill)
    realpos = real(mycount)
    realinterval = real(interval)
    temp = (realpos/realinterval) * 3.1416
    y = sin(temp) * 1000.
    myy = int(y)
    col_sum = 0.d0
    force = 1.d0
    do j = 1, myy
        col_sum = (myy * force) + col_sum
        force = force -(real(j)/normal)
    enddo
    mysum = mysum + col_sum
    call MPI_SEND(mysum, 1, MPI_DOUBLE_PRECISION, 0, myid, MPI_COMM_WORLD, ier)
    call MPI_RECV(mycount, 1, MPI_INTEGER, 0, myid, MPI_COMM_WORLD, status, ier)
enddo
endif
!
if (myid .EQ. 0) then
        print *, "computed    force =", sum
endif
call MPI_Finalize(ier)
stop
end program compute_sin

Note: Consider the volume of messages being traded between processors.

LAB

Let's go back to the pi code:

program compute_pi
!
! Karst compile line: ifort -integer-size 64 -o pi pi.f90
!
integer :: i
integer(kind = 8) :: interval
real(kind=8) :: width, partial, sum, pi
!
! Sum of intervals equals one
!
interval = 20000000000
width = 1.d0 / interval
sum = 0.d0
do i = 1, interval
    partial = width * (i - 0.5d0)
    sum = sum + (4.d0 / (1.d0 + partial * partial))
endo
pi = width * sum
print *, "computed pi =", pi
print *, "reference pi = 3.1415926535897932385"
stop
end program compute_pi

Our goal is to implement MPI parallelism (and/or even OpenMP) to reduce runtime.   Good luck!