Recall domain decomposition

* We will partition the physical domain into pieces and assign each piece to a process. Each process will communicate with it neighboring domain using message passing.

* We developed a code to solve the Poisson equation using non-blocking sends and receives:

* The algorithm:

1. Initiate sends with MPI_ISEND
2. Initiate receives with MPI_IRECV
3. Wait for communication to complete with MPI_WAITALL
4. Unpack the buffers
5. Do Jacobi iteration
6. goto 1.

```
   DO iter = 1,4000

! post sends
   DO edge = 1,4
      CALL MPI_ISEND (buf(edge)%send,SIZE (buf(edge)%send(:)), &
                      MPI_DOUBLE_PRECISION,nghbr_list(edge),msg_tag, &
                      MPI_COMM_WORLD,send_req(req),ierr)
   ENDDO

! post receives
   DO edge = 1,4
      CALL MPI_IRECV (buf(edge)%recv,SIZE (buf(edge)%recv(:)), &
                      MPI_DOUBLE_PRECISION,nghbr_list(edge),msg_tag, &
                      MPI_COMM_WORLD,recv_req(req),ierr)
   ENDDO

! wait for messages to complete
   CALL MPI_WAITALL (nghbr_count,send_req,send_status,ierr)
   CALL MPI_WAITALL (nghbr_count,recv_req,recv_status,ierr)

!  unpack buffers
   DO edge = 1,4
      IF (edge == 1) alph(i_max+1,1:j_max) = buf(edge)%recv(:) ! east
      IF (edge == 2) alph(1:i_max,j_max+1) = buf(edge)%recv(:) ! north
      IF (edge == 3) alph(0,1:j_max)       = buf(edge)%recv(:) ! west
      IF (edge == 4) alph(1:i_max,0)       = buf(edge)%recv(:) ! south
   ENDDO

!  jacobi iteration
      DO j = 1,j_max
         DO i = 1,i_max
            tmpry(i,j) = 0.25*(alph(i+1,j)+alph(i,j+1)+ &
                               alph(i-1,j)+alph(i,j-1)-h*h*beta(i,j))
         ENDDO
      ENDDO
      alph(1:i_max,1:j_max) = tmpry(1:i_max,1:j_max)

   ENDDO ! iter
```

A numerical test

* This code was tested on Seaborg. The NERSC IBM SP RS/6000, named Seaborg, is a distributed memory computer with 6,080 processors. The machine is named in honor of Glenn Seaborg. Each processor has a peak performance of 1.5 GFlops. The processors are distributed among 380 compute nodes with 16 processors per node. Processors on each node have a shared memory pool of between 16 and 64 GBytes
* We use a 720X720 grid point mesh and perform 4000 Jacobi iterations.

* Time to run the basic code:

number of procs	wall time	speed up
1	78.92	-
4	22.67	3.5
16	2.69	29.3
32	1.72	45.9
64	1.11	71.1

Overlap computations and communication

* We will try to do some useful work while numbers move from one process to another.
* Rearrange the algorithm a bit:

1. Initiate sends with MPI_ISEND
2. Initiate receives with MPI_IRECV
3. Do Jacobi iteration on the interior of each block
4. Wait for communication to complete with MPI_WAITALL
5. Do Jacobi iteration along the perimeter of each block
6. goto 1.

* Overlap computations and communication:

number of procs	wall time	speed up
1	83.89	-
4	24.28	3.5
16	2.76	30.4
32	1.82	46.1
64	1.13	74.2

* Time to run the basic code:

number of procs	wall time	speed up
1	78.92	-
4	22.67	3.5
16	2.69	29.3
32	1.72	45.9
64	1.11	71.1

Send and receive with the same command

* Often with the domain decomposition approach each send is paired with a receive. Here we combine them into one command.
* Blocking sends and receives used in domain decomposition communication patterns may cause cyclic dependencies that lead to deadlock. When a send-receive operation is used, the communication subsystem prevents this problem.
* `MPI_SENDRECV` had the following syntax:

```
MPI_SENDRECV(sendbuf,sendcount,sendtype,dest,sendtag,
             recvbuf,recvcount,recvtype,sour,recvtag,
             comm,status)
```

Implementation of MPI_SENDRECV

* No need fpr MPI_WAITALL!
* The algorithm:

1. Load send buffers
2. MPI_SENDRECV
3. Unpack the buffers
4. Do Jacobi iteration
5. goto 1.

```
   DO iter = 1,4000

      DO edge = 1,4
         IF (nghbr_list(edge) /= -1) THEN
! load buffers
            IF (edge == 1) buf(edge)%send(:) = alph(i_max,1:j_max) ! east
            IF (edge == 2) buf(edge)%send(:) = alph(1:i_max,j_max) ! north
            IF (edge == 3) buf(edge)%send(:) = alph(1,1:j_max)     ! west
            IF (edge == 4) buf(edge)%send(:) = alph(1:i_max,1)     ! south
            buf(edge)%recv(:) = 0.0
            send_tag = (npe_wrld+1)*rnk_wrld + nghbr_list(edge) + 1
            recv_tag = (npe_wrld+1)*nghbr_list(edge) + rnk_wrld + 1
!  use MPI_SENDRECV
            CALL MPI_SENDRECV ( &
                           buf(edge)%send,SIZE (buf(edge)%send(:)), &
                           MPI_DOUBLE_PRECISION,nghbr_list(edge),send_tag, &
                           buf(edge)%recv,SIZE (buf(edge)%recv(:)), &
                           MPI_DOUBLE_PRECISION,nghbr_list(edge),recv_tag, &
                           MPI_COMM_WORLD,status,ierr)
         ENDIF
      ENDDO
!  unpack buffers
      DO edge = 1,4
         IF (nghbr_list(edge) /= -1) THEN
            IF (edge == 1) alph(i_max+1,1:j_max) = buf(edge)%recv(:) ! east
            IF (edge == 2) alph(1:i_max,j_max+1) = buf(edge)%recv(:) ! north
            IF (edge == 3) alph(0,1:j_max)       = buf(edge)%recv(:) ! west
            IF (edge == 4) alph(1:i_max,0)       = buf(edge)%recv(:) ! south
         ENDIF
      ENDDO
!  jacobi iteration
      DO j = 1,j_max
         DO i = 1,i_max
            tmpry(i,j) = 0.25*(alph(i+1,j)+alph(i,j+1)+ &
                               alph(i-1,j)+alph(i,j-1)-h*h*beta(i,j))
         ENDDO
      ENDDO
      alph(1:i_max,1:j_max) = tmpry(1:i_max,1:j_max)
   ENDDO ! iter
```

MPI_SENDRECV

* using MPI_SENDRECV:

number of procs	wall time	speed up
1	78.58	-
4	23.05	3.4
16	2.68	29.3
32	1.78	44.1
64	1.73	45.4

* Time to run the basic code:

number of procs	wall time	speed up
1	78.92	-
4	22.67	3.5
16	2.69	29.3
32	1.72	45.9
64	1.11	71.1

MPI Derived Datatypes

* Allows the user to associate a datatype with a noncontiguous block of memory.
* Eliminates user defined buffers
* The MPI_TYPE_VECTOR command creates a datatype which describes a group of elements separated by a constant stride
* `MPI_TYPE_VECTOR` has the following syntax:

```
MPI_TYPE_VECTOR(count,blocklength,stride,oldtype,newtype)
```

* For example we can define a 2X3 block like this:

```
MPI_TYPE_VECTOR
(3,2,7,MPI_INTEGER,blktype)
```

29	30	31	32	33	34	35
22	23	24	25	26	27	28
15	16	17	18	19	20	21
8	9	10	11	12	13	14
1	2	3	4	5	6	7

MPI Derived Datatypes

* After a new datatype is created it has to committed to the system
* `MPI_TYPE_COMMIT` has the following syntax:

```
MPI_TYPE_COMMIT(newtype,ierr)
```

MPI Derived Datatypes

* using MPI_SENDRECV with MPI derived datatype:

number of procs	wall time	speed up
1	82.02	-
4	24.02	3.4
16	2.59	31.7
32	1.85	44.3
64	1.23	66.7

* Time to run the basic code:

number of procs	wall time	speed up
1	78.92	-
4	22.67	3.5
16	2.69	29.3
32	1.72	45.9
64	1.11	71.1

Virtual topology

* The pattern of how processes in a parallel computer are connected is called topology.
* The best way to assign domain decomposition blocks to processes depends on the topology of the underlying hardware.
* MPI provides commands which allow process assignment that best takes advantage of a computer's topology.
* `MPI_CART_CREATE` returns a handle to a new communicator to which the cartesian topology information is attached. If reorder = true, the function may reorder the processes to choose a good embedding of the virtual topology onto the physical machine. `MPI_CART_CREATE` had the following syntax:

```
MPI_CART_CREATE(comm_old,ndims,dims,periods,reorder,
                comm_cart)
```

MPI Virtual topology

* using MPI_CART_CREATE

number of procs	wall time	speed up
1	85.2	-
4	24.2	3.5
16	2.58	33.0
32	2.41	35.6
64	1.48	57.6

* Time to run the basic code:

number of procs	wall time	speed up
1	78.92	-
4	22.67	3.5
16	2.69	29.3
32	1.72	45.9
64	1.11	71.1