# Advanced MPI

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

Parallel Algorithms, 2009





## The Need For Derived Datatypes

- Optimal message construction for mixed data types (our examples thus far have been of a uniform type, contiguous in memory - not exactly real world conditions).
- It might be tempting to send messages of different type separately - but that incurs considerable overhead (especially for small messages) leading to inefficient message passing.
- Type casting or conversion is hazardous, and best avoided.





## Derived Datatypes

A derived datatype consists of two things:

- A sequence of primitive types
- A sequence of integer (byte) displacements, **not** necessarily positive, distinct, or ordered.

The **type map** is this pair of sequences,

$$\text{typemap} = \{(\text{type}_0, \text{disp}_0), (\text{type}_1, \text{disp}_1), \ldots, (\text{type}_{N-1}, \text{disp}_{N-1})\}, \quad (1)$$

with the **type signature** being the sequence of primitive types

$$\text{typesig} = \{\text{type}_0, \text{type}_1, \ldots, \text{type}_{N-1}\}, \quad (2)$$

taken together with a base memory address, the type map specifies a communication buffer.





## Datatype Constructors

This is a sampling of the most-commonly used routines that are available (there are many more ...) in rough order of increasing complexity:

**MPI_TYPE_DUP**

```
MPI_TYPE_DUP(oldtype,newtype)
```

oldtype (IN), datatype (handle)
newtype (OUT), copy of type (handle)

- Simple duplication (more useful for library writers)

## MPI_TYPE_CONTIGUOUS

```
MPI_TYPE_CONTIGUOUS(count,oldtype,newtype)
```

- **count** (IN), replication count (int)
- **oldtype** (IN), old datatype (handle)
- **newtype** (OUT), new datatype (handle)

- duplication and replication (by concatenation) of datatypes.





## MPI_TYPE_VECTOR

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

- **count** (IN), number of blocks (int)
- **blocklen** (IN), number elements in each block (int)
- **stride** (IN), spacing (in elements) between start of each block (int)
- **oldtype** (IN), old datatype (handle)
- **newtype** (OUT), new datatype (handle)

- Replication of datatype into equally spaced (equal stride = extent of oldtype) blocks





## MPI_TYPE_CREATE_HVECTOR

```
MPI_TYPE_CREATE_HVECTOR(count,blocklen,stride,
            oldtype,newtype)
```

- **count** (IN), number of blocks (int)
- **blocklen** (IN), number elements in each block (int)
- **stride** (IN), spacing (in bytes) between start of each block (int)
- **oldtype** (IN), old datatype (handle)
- **newtype** (OUT), new datatype (handle)

- replicate a datatype into equally spaced locations, separated by byte stride (bytes for HVECTOR, extents of the old datatype for VECTOR).





## MPI_TYPE_INDEXED

```
MPI_TYPE_INDEXED(count,array_blocklen,
        array_disp,oldtype,newtype)
```

- **count** (IN), number of blocks (int)
- **array_blocklen** (IN), number of elements per block (int array)
- **array_disp** (IN), displacements (in elements) for each block (int array)
- **oldtype** (IN), old datatype (handle)
- **newtype** (OLD), new datatype (handle)

- Indexed allows the user to specify a noncontiguous data layout where separations between blocks is not the same (unequal strides).

### MPI_TYPE_CREATE_STRUCT

```
MPI_TYPE_CREATE_STRUCT(count,array_blocklen,
       array_disp, array_type,newtype)
```

- **count** (IN), number of blocks (int)
- **array_blocklen** (IN), number of elements per block (int array)
- **array_disp** (IN), displacements (in elements) for each block (int array)
- **array_type** (IN), type of elements in each block (handle array)
- **newtype** (OUT), new datatype (handle)

- the most general type constructor, allowing each block to consist of replications of different datatypes





... and many more ... `MPI_TYPE_CREATE_INDEXED_BLOCK` (constant blocksize, arbitrary displacements), `MPI_TYPE_CREATE_HINDEXED`(block displacements specified in Bytes) ... ...





## Datatype Accessors

Routines to determine information on derived datatypes (they will work on predefined datatypes as well, of course):

### MPI_TYPE_GET_EXTENT

```
MPI_TYPE_GET_EXTENT(datatype, lb, extent)
```

- **datatype** (IN), datatype on which to return info (handle)
- **lb** (OUT), lower bound of datatype (int)
- **extent** (OUT), extent of datatype (int)

- “size” of the datatype, i.e. use `MPI_TYPE_GET_EXTENT` for MPI types, rather than C’s `sizeof(datatype)`





### MPI_TYPE_SIZE

```
MPI_TYPE_SIZE(datatype, size)
```

- **datatype** (IN), datatype on which to return info (handle)
- **size** (OUT), datatype siz, in bytes (int)

- total size, in Bytes, of entries in datatype signature

## Committed Datatypes

A derived datatype must be **committed** before use, once committed, a derived datatype can be used as input for further datatype construction.

**MPI_COMMIT**

`MPI_COMMIT(datatype)`

datatype (INOUT), datatype to be committed (handle)





and a routine to free up a datatype object:

**MPI_TYPE_FREE**

`MPI_TYPE_FREE(datatype)`

datatype (INOUT), datatype to be freed (handle)

and there are routines for greater control (and more complexity) ...
`MPI_GET_ADDRESS` (find the address of a location in memory),
`MPI_GET_ELEMENTS` (number of primitive elements received),
`MPI_TYPE_CREATE_RESIZED` (the ability to resize an existing user defined datatype),
`MPI_TYPE_GET_TRUE_EXTENT` (overlook “artificial” extents)...





## A Derived Datatype Example

```
double a[100][100]; /* matrix, order 100 */
int disp[100],blocklen[100],i,dest,tag;
MPI_Datatype upperTri; /* upper triangular part of the matrix */
...
for (i=0,i<=99;i++) {
  disp[i] = 100*i+i;
  blocklen[i] = 100-i;
}
MPI_Type_indexed(100,blocklen,disp,MPI_DOUBLE,&upperTri); /* create datatype */
MPI_Type_commit(&upperTri);
MPI_Send(a,1,upperTri,dest,tag,MPI_COMM_WORLD);
```

- A handle to a derived datatype can appear in sends/receives (including collective ops).
- Note that the predefined MPI datatypes are just special cases of a derived datatype. For example, `MPI_FLOAT` is a predefined handle to a datatype with type map $\{(\text{float}, 0)\}$.





## Packing it In

**MPI_PACK**

```
MPI_PACK(in_buffer,in_count,datatype,
         out_buffer,out_size,pos,comm)
```

in_buffer (IN), input buffer (choice)
in_count (IN), number of input components (int)
datatype (IN), datatype of each input component (handle)
out_buffer (OUT), output buffer (choice)
out_size (IN), output buffer size, in bytes (int)
pos (INOUT), current positionin buffer, in bytes (int)
comm (IN), communicator for packed messages (handle)

**MPI_UNPACK**

`MPI_UNPACK(in_buffer,in_size,pos,out_buffer, out_count,datatype,comm)`

- in_buffer (IN), input buffer (choice)
- in_size (IN), input buffer size, in bytes (int)
- pos (INOUT), current position in buffer, in bytes (int)
- out_buffer (OUT), output buffer (choice)
- out_count (IN), number of components to unpack (int)
- datatype (IN), datatype of each input component (handle)
- comm (IN), communicator for packed messages (handle)





These routines (`MPI_PACK, MPI_UNPACK`) allow you to fill a buffer with non-contiguous data in a streamlined fashion - the following routine will tell you how much space the message will occupy, if you want to manage your buffers:

**MPI_PACK_SIZE**

`MPI_PACK_SIZE(in_count, datatype, comm, size)`

- in_count (IN), count argument to packing call (int)
- datatype (IN), datatype argument to packing call (handle)
- comm (IN), communicator argument to packing call (handle)
- size (OUT), upper bound on size of packed message, in bytes (int)

The data format used for packed data is implementation dependent.





# An Example of Message Packing

```
int my_i,pos=0;
char a[100],buff[110];
MPI_Status status;
...
if (myrank == 0) {
   MPI_Pack(&my_i,1,MPI_INT,buff,110,&pos,MPI_COMM_WORLD);
   MPI_Pack(a,100,MPI_CHAR,buff,110,&pos,MPI_COMM_WORLD);
   MPI_Send(buff,pos,MPI_PACKED,1,0,MPI_COMM_WORLD);
}
else {
   MPI_Recv(buff,110,MPI_PACKED,1,0,MPI_COMM_WORLD,&status);
   MPI_Unpack(buff,110,&pos,&my_i,1,MPI_INT,MPI_COMM_WORLD);
   MPI_Unpack(buff,110,&pos,a,100,MPI_CHAR,MPI_COMM_WORLD);
}
...
```





# Derived Datatypes vs. Pack/Unpack

- The data format used for packed data is implementation dependent.
- Messages are the same size
- May take longer to access non-contiguous memory of derived types
- Packing executes a function call for each packed item, and possibly additional memory-to-memory copies (packing has to copy the data, derived types need to store the layout). Most implementations can expect better performance from derived types.

## MPI Communicators

- Provides a separate communication space, especially useful for libraries and modules (can use their own numbering scheme).
- If you are uncomfortable dealing with multiple spaces for communications, just use a single one - the pre-defined `MPI_COMM_WORLD`.





- Two types of communicators:
  1. **intra-communicator** - for comms within a group of processes. Can also have a topology describing the process layout.
  2. **inter-communicator** - for comms between two disjoint groups of processes. No topology.

| Functionality | Intra- | Inter- |
|---|---|---|
| Number of groups involved | 1 | 2 |
| Communication Safety | Y | Y |
| Collective Ops | Y | Y(MPI-2) |
| Topologies | Y | N |
| Caching | Y | Y |





## More Communication Domains

- You can think of a communicator as an array of links to other communicators.
- Each intra-group communication domain consists of a set of communicators such that:
  - the links form a complete graph in which each communicator is linked to all communicators in the set (including itself)
  - the links have consistent indices, for each communicator the i-th link points to the communicator for process i.
- Each process holds a complete list of group members - not necessarily a scalable design.





## Key Group Routines

**MPI_COMM_GROUP**

`MPI_COMM_GROUP(comm,group)`

- comm (IN), communicator (handle)
- group (OUT), group corresponding to comm (handle)

- obtain the group handle for a given communicator - new groups have to be built from old ones (they can not be built from scratch)
- returned handle can then be used as input to `MPI_GROUP_INCL`, `MPI_COMM_CREATE`, `MPI_GROUP_RANK`.

## MPI_GROUP_INCL

`MPI_GROUP_INCL(group, n, ranks, newgroup)`

- **group** (IN), group (handle)
- **n** (IN), number of elements in array ranks (and size of newgroup) (int)
- **ranks** (IN), ranks of processes in group to appear in newgroup (int array)
- **newgroup** (OUT), new group derived from input, in order defined by ranks (handle)

- creates a new group whose i-th process had ranks[i] in the old group
- n=0 results in newgroup having the value `MPI_GROUP_EMPTY`.





## MPI_GROUP_EXCL

`MPI_GROUP_EXCL(group, n, ranks, newgroup)`

- **group** (IN), group (handle)
- **n** (IN), number of elements in array ranks (and size of newgroup) (int)
- **ranks** (IN), ranks of processes in group to appear in newgroup (int array)
- **newgroup** (OUT), new group derived from input, in order defined by ranks (handle)

- newgroup created from group by deleting processes with ranks ranks[0]...ranks[n-1]
- n=0 newgroup is identical to group





## MPI_GROUP_RANK

`MPI_GROUP_RANK(group,rank)`

- **group** IN, group (handle)
- **rank** OUT, rank of the calling process in group (int)

- returns the rank of the calling process in group
- if calling process is not a member of group, `MPI_UNDEFINED` is returned.

## MPI_GROUP_SIZE

`MPI_GROUP_SIZE(group,size)`

- **group** (IN), group (handle)
- **size** (OUT), number of processes in group (int)





## MPI_GROUP_FREE

`MPI_GROUP_FREE(group)`

- **group** (INOUT), group (handle)

- mark group fro deallocation
- handle group is set to `MPI_GROUP_NULL`

# Key Communicator Routines

## MPI_COMM_CREATE

```
MPI_COMM_CREATE(comm,group,newcomm)
```

- **comm** (IN), communicator (handle)
- **group** (IN), group, a subset of the group of comm
- **newcomm** (OUT), new communicator (handle)

- must be executed by all processes in comm
- returns `MPI_COMM_NULL` to processes not in group





Our old friend, but in a new context ...

## MPI_COMM_RANK

```
MPI_COMM_RANK(comm, rank)
```

- **comm** (IN), communicator (handle)
- **rank** (OUT), rank of the calling process in group of comm (int)

- if comm is an intra-communicator, rank is the rank of the calling process
- rank is relative to the group associated with comm





Primary API call for forming new communicators:

## MPI_COMM_SPLIT

```
MPI_COMM_SPLIT(comm, color, key, newcomm)
```

- **comm** (IN), communicator (handle)
- **color** (IN), control of subset assignment (int)
- **key** (IN), control of rank assignment (int)
- **newcomm** (OUT), new communicator (handle)





`MPI_COMM_SPLIT(comm, color, key, newcomm)`:

- partitions group associated with comm into disjoint subgroups, one for each value of color.
- a collective call, but each process can provide its own color and key
- a color of `MPI_UNDEFINED` results in a newcomm of `MPI_COMM_NULL`
- for same key values, rank in new communicator is relative to ranks in the old communicator
- a very useful call for breaking a single communicator group into a user controlled number of subgroups. Multigrid, linear algebra, etc.

# Master/Server Example Using Group/Communicator Routines

We can use the communicator and group routines to lay out a simple code for performing master/worker tasks:

- Master is process zero, rest are workers
- Create a group of workers by eliminating server process
- Create communicator for workers
- Master/worker task code





```
1  int ServerTask, myRank, myWorkerRank;
2  MPI_Comm comm_workers;
3  MPI_Group group_world, group_workers;
4
5  MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
6
7  ServerTask = 0;
8  MPI_Comm_group(MPI_COMM_WORLD,&group_world);
9  MPI_Group_excl(group_world,1,ServerTask,&group_workers);
10 MPI_Comm_create(MPI_COMM_WORLD,&group_workers,&comm_workers);
11 MPI_Group_free(&group_workers);   /* if no longer needed */
12
13 if (myRank == ServerTask) {
14    RunServer();
15 } else {
16    MPI_Comm_rank(comm_workers,&myWorkerRank);
17    WorkerBees();
18 }
19 ...
```





# Virtual Topologies

- An extra, optional attribute for an intra-communicator
- Convenient naming mechanism for processes in a group
- Many applications can benefit from a 2d or 3d topological communication pattern
- Possible mapping of runtime processes to available hardware
- “Virtual” topology is all that we will discuss - machine independent
- Two main topology types in MPI - Cartesian (grid) and graphs - while graphs are the more general case, majority of applications use regular grids





# Topology Benefits

Key benefits of MPI topologies:

- Applications have specific communication patterns (e.g. a 2D Cartesian topology suits 4-way nearest neighbor communications)
- Topologies are advisory to the implementation - topological aspects of the underlying hardware may offer performance advantages to various communication topologies

# Key Topology Routines

## MPI_CART_CREATE

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

- comm_old (IN), input communicator (handle)
- ndims (IN), dimensions in Cartesian grid (int)
- dims (IN), processes in each dimension (int array)
- periods (IN), periodic (true) in each dim (logical array)
- reorder (IN), ranks may be reordered (true) or not (logical)
- comm_cart (OUT), comm. with new topology (handle)

- Must be called by all processes in the group, extras will end up with `MPI_COMM_NULL`.





## MPI_CART_COORDS

```
MPI_CART_COORDS(comm,rank,maxdims,coords)
```

- comm (IN), communicator with Cartesian structure (handle)
- rank (IN), rank of a process within group comm (int)
- maxdims (IN), length of vector coord in the calling program (int)
- coords (OUT), array containing Cartesian coordinates of specified process (int array)

- rank to coordinates translator (the inverse of `MPI_CART_RANK`)





## MPI_CART_RANK

```
MPI_CART_RANK(comm,coords,rank)
```

- comm (IN), communicator with Cartesian structure (handle)
- coords (IN), specifies the Cartesian coordinates of a process (int array)
- rank (OUT), rank of specified process (int)

- coordinates to rank translator (the inverse of `MPI_CART_COORDS`).





## MPI_CART_SUB

```
MPI_CART_SUB(comm,remain_dims,newcomm)
```

- comm (IN), communicator with Cartesian structure (handle)
- remain_dims (IN), i-th entry = true, then i-th dimension is kept in the subgrid (array of logicals)
- newcomm (OUT), communicator containing subgrid that includes calling process (handle)

- A collective routine to be called by all processes in comm
- Partitions communicator group into subgroups that form lower dimensional Cartesian subgrids

### MPI_CARTDIM_GET

```
MPI_CARTDIM_GET(comm,ndims)
```

- **comm** (IN), communicator with Cartesian structure (handle)
- **ndims** (OUT), number of dimensions of the structure (int)





### MPI_CART_GET

```
MPI_CART_GET(comm,maxdims,dims,periods,coords)
```

- **comm** (IN), communicator with Cartesian structure (handle)
- **maxdims** (IN), length of vector dims, periods, coords in calling program (int)
- **dims** (OUT), number processes in each Cartesian dim (int array)
- **periods** (OUT), periodicity in each dim (logical array)
- **coords** (OUT), coordinates of calling process in structure (int array)





### MPI_CART_SHIFT

```
MPI_CART_SHIFT(comm,direction,displ,
               rank_source,rank_dest)
```

- **comm** (IN), communicator with Cartesian structure (handle)
- **direction** (IN), coordinate dimensions of shift (int)
- **displ** (IN), displacement (>0 for up, <0 down) (int)
- **rank_source** (OUT), rank of source process (int)
- **rank_dest** (OUT), rank of destination process (int)

- direction has range [0,..,ndim-1] (e.g. for 3D from 0 to 2)
- if destination is out of bound, a negative value is returned (MPI_UNDEFINED), which implies no periodicity in that direction.





## Cartesian Topology Example

Simple example to illustrate Cartesian topology:

- Construct a 2D, 4x4 grid
- Treat without periodic boundaries (e.g. as a domain decomposition with fixed boundaries)
- Construct list of SENDRECV pairs for each process in the grid

```
1   #include "mpi.h"
2   #include <stdio.h>
3   #define SIZE 16
4   #define UP    0
5   #define DOWN  1
6   #define LEFT  2
7   #define RIGHT 3
8
9   int main(int argc,char **argv)
10  {
11    int numtasks, rank, source, dest, outbuf, i, tag=1,
12      inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
13      nbrs[4], dims[2]={4,4},
14      periods[2]={0,0}, reorder=0, coords[2];    /* not periodic, no reordering */
15
16    MPI_Request reqs[8];
17    MPI_Status stats[8];
18    MPI_Comm cartcomm;
19
20    MPI_Init(&argc,&argv);
21    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
22
23    if (numtasks == SIZE) {
24      MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
25      MPI_Comm_rank(cartcomm, &rank);
26      MPI_Cart_coords(cartcomm, rank, 2, coords);
27      MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);    /* s/r +1 shift in rows */
28      MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]); /* s/r +1 shift in cols */
```



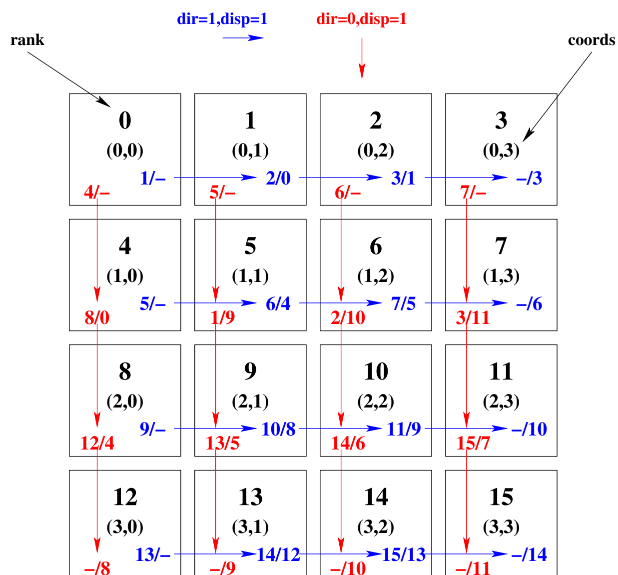


```
29      outbuf = rank;
30
31      for (i=0; i<4; i++) {
32        dest = nbrs[i];
33        source = nbrs[i];
34        MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,
35                  MPI_COMM_WORLD, &reqs[i]);
36        MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,
37                  MPI_COMM_WORLD, &reqs[i+4]);
38      }
39
40      MPI_Waitall(8, reqs, stats);
41
42      printf("rank= %3d coords= %3d %3d  neighbors(u,d,l,r)= %3d %3d %3d %3d\n",
43             rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],
44             nbrs[RIGHT]);
45      printf("rank= %3d                   inbuf(u,d,l,r)=     %3d %3d %3d %3d\n",
46             rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);
47    }
48    else
49      printf("Must specify %d processors. Terminating.\n",SIZE);
50
51    MPI_Finalize();
52  }
```





# Cartesian Topology Example Illustrated








# Running The Topology Example

```
1   [bono:~/d_mpi-samples]$ qsub -q debug -lnodes=8:ppn=2,walltime=00:15:00 -I
2   qsub: waiting for job 566107.bono.ccr.buffalo.edu to start
3   qsub: job 566107.bono.ccr.buffalo.edu ready
4
5   ############################PBS Prologue############################
6   PBS prologue script run on host c15n28 at Tue Sep 18 13:50:40 EDT 2007
7   PBSTMPDIR is /scratch/566107.bono.ccr.buffalo.edu
8   [c15n28:~]$ cd $PBS_O_WORKDIR
9   [c15n28:~/d_mpi-samples]$ module load mpich/gcc-3.4.6/ch_p4/1.2.7p1
10  [c15n28:~/d_mpi-samples]$ mpiexec ./mpi-cart-ex
11  rank=  2 coords=  0  2  neighbors(u,d,l,r)= -1  6  1  3
12  rank=  2                inbuf(u,d,l,r)=     -1  6  1  3
13  rank=  8 coords=  2  0  neighbors(u,d,l,r)=  4 12 -1  9
14  rank=  8                inbuf(u,d,l,r)=      4 12 -1  9
15  rank= 14 coords=  3  2  neighbors(u,d,l,r)= 10 -1 13 15
16  rank= 14                inbuf(u,d,l,r)=     10 -1 13 15
17  rank=  3 coords=  0  3  neighbors(u,d,l,r)= -1  7  2 -1
18  rank=  3                inbuf(u,d,l,r)=     -1  7  2 -1
19  rank=  5 coords=  1  1  neighbors(u,d,l,r)=  1  9  4  6
20  rank=  5                inbuf(u,d,l,r)=      1  9  4  6
21  rank=  7 coords=  1  3  neighbors(u,d,l,r)=  3 11  6 -1
22  rank=  7                inbuf(u,d,l,r)=      3 11  6 -1
23  rank= 15 coords=  3  3  neighbors(u,d,l,r)= 11 -1 14 -1
24  rank= 15                inbuf(u,d,l,r)=     11 -1 14 -1
25  rank=  6 coords=  1  2  neighbors(u,d,l,r)=  2 10  5  7
26  rank=  6                inbuf(u,d,l,r)=      2 10  5  7
27  rank= 10 coords=  2  2  neighbors(u,d,l,r)=  6 14  9 11
28  rank= 10                inbuf(u,d,l,r)=      6 14  9 11
```

```
29  rank=  12 coords=   3   0   neighbors(u,d,l,r)=   8  -1  -1  13
30  rank=  12                   inbuf(u,d,l,r)=       8  -1  -1  13
31  rank=  11 coords=   2   3   neighbors(u,d,l,r)=   7  15  10  -1
32  rank=  11                   inbuf(u,d,l,r)=       7  15  10  -1
33  rank=   0 coords=   0   0   neighbors(u,d,l,r)=  -1   4  -1   1
34  rank=   0                   inbuf(u,d,l,r)=      -1   4  -1   1
35  rank=   1 coords=   0   1   neighbors(u,d,l,r)=  -1   5   0   2
36  rank=   1                   inbuf(u,d,l,r)=      -1   5   0   2
37  rank=   4 coords=   1   0   neighbors(u,d,l,r)=   0   8  -1   5
38  rank=   4                   inbuf(u,d,l,r)=       0   8  -1   5
39  rank=   9 coords=   2   1   neighbors(u,d,l,r)=   5  13   8  10
40  rank=   9                   inbuf(u,d,l,r)=       5  13   8  10
41  rank=  13 coords=   3   1   neighbors(u,d,l,r)=   9  -1  12  14
42  rank=  13                   inbuf(u,d,l,r)=       9  -1  12  14
```





# Process Startup

- Single most confusing aspect of MPI for most new users
- Implementation dependent! with many implementation specific options, flags, etc.
- Consult the documentation for the MPI implementation that you are using.





# Some Examples Using MPI Task Launchers

**SGI Origin/Altix (intra-machine)**:

```
mpirun -np <np> [options] <progname> [progname options]
```

**MPICH-1 ch_p4 device**:

```
mpirun -machinefile <filename> -np <np> [options] <progname> [args]
```

**Sun HPC Tools**:

```
mprun -l ''nodename [nproc] [,nodename [nproc],...] [options] <executable> [args]
```

**IBM AIX POE**:

```
poe ./a.out -nodes [nnodes] -tasks_per_node [ntasks] [options]
```

**OSC's PBS/Torque based mpiexec**:

```
mpiexec [-pernode] [-kill] [options] <executable> [args]
```





# Getting Implementation Info from MPI

**MPI_GET_VERSION**

`MPI_GET_VERSION(version,subversion)`

- version (OUT), version number (int)
- subversion (OUT), subversion number (int)

- Not exactly critical for programming, but a nice function for determining what version of MPI you are using (especially when the documentation for your machine is poor).

## Where am I running?

### MPI_GET_PROCESSOR_NAME

`MPI_GET_PROCESSOR_NAME(name, resultlen)`

name (OUT), A unique specifier for the actual node (string)

resultlem (OUT), Length (in printable chars) of the reslut in name (int)

- returns the name of the processor on which it was called at the moment of the call.
- name should have storage that is at least `MPI_MAX_PROCESSOR_NAME` characters long.





## Timing & Synchronization

### MPI_WTIME

`MPI_WTIME()`

- double precision value returned representing elapsed wall clock time from some point in the past (origin guaranteed not to change during process execution time).
- A portable timing function (try finding another!) - can be high resolution, provided it has some hardware support.





Testing the resolution of `MPI_WTIME`:

### MPI_WTICK

MPI_WTICK()

- double precision value returned which is the resolution of `MPI_WTIME` in seconds.
- hardware dependent, of course - if a high resolution timer is available, it should be accessible through `MPI_WTIME`.





Common `MPI_Wtime` usage:

```
double time0,time1;
...
time0 = MPI_Wtime();
...
/* code to be timed */
...
time1 = MPI_Wtime();
printf(''Time interval = %f seconds\n'',time1-time0);
```

# More About MPI Error Codes

**MPI_ERROR_STRING**

```
MPI_ERROR_STRING(errorcode, string, resultlen)
```

errorcode (IN), Error code returned by an MPI routine (int)

string (OUT), Text that corresponds to errorcode (string)

resultlen (OUT), Length (in printable chars) of result returned in string (int)

- Most error codes in MPI are implementation dependent
- `MPI_ERROR_STRING` provides information on the type of MPI exception that occurred.
- argument string must have storage that is at least `MPI_MAX_ERROR_STRING` characters.





# MPI Profiling Hooks

- The MPI profiling interface is designed for authors of profiling tools, such that they will not need access to a particular implementation's source code (which a vendor may not wish to release).
- Many profiling tools exist:
  1. **Vampir** (Intel, formerly Pallas), now called *Intel Trace Analyzer and Visualizer*
  2. **HPMCount** (IBM AIX)
  3. **jumpshot** (MPICH)
  4. *SpeedShop, cvperf* (SGI)
- Consult your profiling tools of choice for detailed usage.





# MPI-2 Features

I will not attempt to fully cover MPI-2 extensions - in the slides that follow I will just give a broad outline of the new features:

- Dynamic process management (routines to create new processes)
- One-sided communications (put/get)
- Parallel I/O
- Additional language bindings (`C++`)
- Extended collective operations (non-blocking, inter-communicator)





# Dynamic Process Management

- An MPI-1 application is static - no processes can be added (or removed) after it has started.
- MPI-2 introduces a spawning call for dynamic execution (MPMD):

## MPI_COMM_SPAWN

```
MPI_COMM_SPAWN(command,argv,maxprocs,info,root,
               comm,intercomm,array_err)
```

| | |
|---|---|
| command | (IN), name of spawned program (string at root) |
| argv | (IN), arguments to command (string array) |
| maxprocs | (IN), maximum number processes to start (int) |
| info | (IN), key-value pairs where and how to start processes (handle) |
| root | (IN), rank of process in which previous arguments are examined (int) |
| comm | (IN), intra-communicator for group of spawning process (handle) |
| intercomm | (OUT), inter-communicator between original and new group |
| array_err | (OUT), one error code per process (int array) |





# Some Notes on MPI_COMM_SPAWN

Things to watch out for when using dynamic task management in MPI:

- Not supported in all implementations
- The attribute `MPI_UNIVERSE_SIZE` of `MPI_COMM_WORLD` gives a useful upper limit on the number of tasks (query using `MPI_Comm_get_attr`)
- Interaction with runtime system generally not visible to application, and not specified by MPI standard
- Static view in which all processes are started at once is still preferred method (for performance if not simplicity)





# One-sided Communication

- extends communication mechanisms of MPI through **RMA** (Remote Memory Access).
- three communication calls:
  - `MPI_PUT` remote write
  - `MPI_GET` remote read
  - `MPI_ACCUMULATE` remote update
- does **not** provide a shared memory programming model or support for direct shared-memory programming.
- Uses memory *windows* and all RMA communications are non-blocking.





# MPI I/O

- a programming interface for I/O
- parallel in the sense of I/O performed by a parallel application, but *cooperative* also, in the sense that many processes concurrently access a single file.
- does **not** specify a filesystem, should be able to interact with a variety of filesystems.
- provides support for asynchronous I/O, strided access, and control over physical file layout on storage devices.

# MPI C++ Bindings

The C++ interface for MPI consists mainly of a small set of classes with a lightweight functional interface to MPI:

- Most C++ bindings for MPI functions are member functions of MPI classes
- All MPI classes, constants, and functions are declared as part of an MPI **namespace**
- Rather than MPI_ prefix (as for C and Fortran), MPI functions in C++ have an MPI:: prefix





# MPI namespace

An abbreviated definition of the MPI namespace:

```
namespace MPI { // MPI-1
  class Comm {...};
  class Intracomm  : public Comm   {...};
  class Graphcomm  : public Intracomm   {...};
  class Cartcomm   : public Intracomm   {...};
  class Intercomm  : public Comm        {...};
  class Datatype                        {...};
  class Errhandler                      {...};
  class Exception                       {...};
  class Group                           {...};
  class Op                              {...};
  class Request                         {...};
  class Prequest   : public Request     {...};
  class Status                          {...};
  // MPI-2
  class File                            {...};
  class Grequest   : public Request     {...};
  class Info                            {...};
  class Win                             {...};
};
```





# C++ MPI Semantics

Construction/Destruction:

```
MPI::<CLASS>()
~MPI::<CLASS>()
```

Copy/Assignment

```
MPI::<CLASS>(const MPI::<CLASS>& data)
MPI::<CLASS>& MPI::<CLASS>::operator=(const MPI::<CLASS>& data)
```





# C++ Data Types

| MPI datatype | C++ datatype |
|---|---|
| MPI::CHAR | char |
| MPI::SHORT | signed short |
| MPI::INT | signed int |
| MPI::LONG | signed long |
| MPI::SIGNED_CHAR | signed char |
| MPI::UNSIGNED_CHAR | unsigned char |
| MPI::UNSIGNED_SHORT | unsigned short |
| MPI::UNSIGNED | unsigned int |
| MPI::UNSIGNED_LONG | unsigned long int |
| MPI::FLOAT | float |
| MPI::DOUBLE | double |
| MPI::LONG_DOUBLE | long double |
| MPI::BOOL | bool |
| MPI::COMPLEX | Complex<float> |
| MPI::DOUBLE_COMPLEX | Complex<double> |
| MPI::LONG_DOUBLE_COMPLEX | Complex<long double> |
| MPI::BYTE | |
| MPI::PACKED | |