Distributed Programming & MPI
MPI paradigm

- Single program, multiple data (SPMD)
  - One program, multiple processes (ranks)
  - Processes communicate via messages
    - An MPI message is a collection of fixed-size data elements
    - Underlying mechanism (e.g., sockets) is implementation-dependent
  - Multiple processes may run on the same node
    - They do NOT share an address space!
    - But intra-node communication will be faster than inter-node
  - Processes are grouped into communicators
    - May be in multiple communicators simultaneously
    - Default communicator: MPI_COMM_WORLD (all processes)
Message-Passing Interface (MPI)

- **MPI** is a standardized software library interface
  - Available online: http://www.mpi-forum.org/docs/
  - **MPI-1** released in 1994 after Supercomputing ‘93
  - **MPI-2** (1996) added one-sided operations and parallel I/O
  - **MPI-3** (2012) improved non-blocking and one-sided operations
    - Also added tooling interface
  - Latest version (**MPI-3.1**) approved June 2015
  - **MPI-4** draft published in November 2020

- Several widely-used implementations
  - OpenMPI and MPICH (on our cluster)
  - MVAPICH / MVAPICH2 (higher performance)
## MPI-3.1 support

### Status of MPI-3.1 Implementations

<table>
<thead>
<tr>
<th>Feature</th>
<th>MPICH</th>
<th>MVAPICH</th>
<th>Open MPI</th>
<th>Cray MPI</th>
<th>Tianhe MPI</th>
<th>Intel MPI</th>
<th>IBM BG/Q MPI 1</th>
<th>IBM PE MPICH 2</th>
<th>IBM Platform</th>
<th>SGI MPI</th>
<th>Fujitsu MPI</th>
<th>MS MPI</th>
<th>MPC</th>
<th>NEC MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Nbrhood collectives</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RMA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Shared memory</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tools Interface</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>*</td>
<td>Q4'16</td>
<td>✓</td>
</tr>
<tr>
<td>Comm-creat group</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>F08 Bindings</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>New Datatypes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Large Counts</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Matched Probe</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>NBC I/O</td>
<td>✓</td>
<td>Q3'16</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>Q2'16</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

MPI development

- MPI involves more than just a library (unlike pthreads)
  - Compiler wrapper (mpicc / mpiCC / mpif77)
    - Still need to #include <mpi.h>
  - Program launcher (mpirun)
  - Job management integration (salloc / sbatch)
    - Do not use srunt!
    - SLURM tasks = MPI processes (set with “-n” switch)
- System admins use modules to ease setup
  - Command: module load mpi (for OpenMPI)
  - Populates your shell environment w/ MPI paths
    - To use MPICH (needed for P4): module load mpi/mpich-3.2.1
Basic MPI functions

```c
int MPI_Init (int *argc, char ***argv)
int MPI_Finalize ()

int MPI_Comm_size (MPI_Comm comm, int *size)
int MPI_Comm_rank (MPI_Comm comm, int *rank)

double MPI_Wtime ()
int MPI_Barrier (MPI_Comm comm)
```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int mpi_rank;
    int mpi_size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);

    printf("Hello from process %2d / %d\n", mpi_rank+1, mpi_size);

    MPI_Finalize();

    return 0;
}
Copy `/shared/cs470/mpi-hello` to your home folder.

Build with "make"
- Don’t forget to “module load mpi” first!

Run locally (don’t do this normally!)
- `mpirun ./hello`

Run on cluster
- `salloc mpirun ./hello`
- `salloc -n 4 mpirun ./hello`
- `salloc -n 16 mpirun ./hello`
- `salloc -N 4 mpirun ./hello`

**IMPORTANT:** the “-n” and “-N” parameters should be for `salloc`, not `mpirun`
MPI conventions

• Identifiers start with “MPI_”
  – Also, first letter following underscore is uppercase

• MPI must be initialized and cleaned up
  – MPI_Init and MPI_Finalize
  – For MPI_Init, you should just “pass through” argc and argv
  – No MPI calls before MPI_Init or after MPI_Finalize!

• Task parallelism is based on rank / process ID
  – MPI_Comm_rank and MPI_Comm_size
  – Rank 0 is often considered to be special (the "supervisor" process)

• I/O is asymmetrical
  – All ranks may write to stdout (or stderr)
  – Usually, only rank 0 can read stdin
Point-to-point messages

- MPI an **explicit** message-passing paradigm
  - You (the developer) decide how to split up data
  - You manage memory allocation manually
  - You decide how to send data between processes
  - Most direct mechanism: **point-to-point** messages

```c
int MPI_Send (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
```

- recv count must be equal to or higher than send count
- must match

```c
int MPI_Recv (void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Status *status)
```

- must match
- must match
- (* unless ignored by MPI_Recv)
Generic receiving

- All parameters are required for `MPI_Send`
- `MPI_Recv` allows for some ambiguity
  - `count` is the *maximum* count (actual could be lower)
  - `src` can be `MPI_ANY_SOURCE` and `tag` can be `MPI_ANY_TAG`
- The `status` parameter provides this info
  - Pointer to `MPI_Status` struct that is populated by `MPI_Recv`
  - After receive, access members `MPI_SOURCE` and `MPI_TAG`
  - Use `MPI_Get_count` to calculate true count
  - If you don't need any of these, pass `MPI_IGNORE_STATUS`

**Postel’s Law**: “Be conservative in what you do; be liberal in what you accept from others.”
# MPI datatypes

<table>
<thead>
<tr>
<th>C data type</th>
<th>MPI data type</th>
<th>Size on cluster (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>MPI_CHAR</td>
<td>1</td>
</tr>
<tr>
<td>unsigned char</td>
<td>MPI_UNSIGNED_CHAR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>MPI_SHORT</td>
<td>2</td>
</tr>
<tr>
<td>unsigned short</td>
<td>MPI_UNSIGNED_SHORT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_SHORT_UNSIGNED_SHORT</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>MPI_INT</td>
<td>4</td>
</tr>
<tr>
<td>unsigned int</td>
<td>MPI_UNSIGNED</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_UNSIGNED_INT</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>MPI_LONG</td>
<td>8</td>
</tr>
<tr>
<td>unsigned long</td>
<td>MPI_UNSIGNED_LONG</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_UNSIGNED_LONG</td>
<td></td>
</tr>
<tr>
<td>long long</td>
<td>MPI_LONG_LONG</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>MPI_LONG_LONG_UNSIGNED</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>MPI_FLOAT</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>MPI_DOUBLE</td>
<td>8</td>
</tr>
</tbody>
</table>
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int my_rank;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    if (my_rank == 0) {
        // rank 0: receive a single integer from any source
        int data = -1;
        MPI_Recv(&data, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                  MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Received data in rank %d: %d\n", my_rank, data);
    } else {
        // other processes: send our rank to rank 0
        MPI_Send(&my_rank, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
Blocking and safety

- Exact blocking behavior is implementation-dependent
  - `MPI_Send` may block until the message is sent
    - Sometimes depends on the size of the message
    - `MPI_Ssend` will always block until the message is received
  - `MPI_Recv` will always block until the message is received
- A program is unsafe if it relies on MPI-provided buffering
  - You can use `MPI_Ssend` to check your code (forces blocking)
  - Use `MPI_SendRecv` if both sending and receiving in a cycle
    - Or use `MPI_Isend` / `MPI_Recv` pairs

```c
int MPI_Sendrecv (void *send_buf, int send_count, MPI_Datatype send_dtype, int dest, int send_tag
void *recv_buf, int recv_count, MPI_Datatype recv_dtype, int src, int recv_tag,
MPI_Comm comm, MPI_Status *status)
```
Non-blocking send/receive

- Some operations are guaranteed not to block
  - Point-to-point: MPI_Isend and MPI_Irecv
  - Includes some collectives (in MPI-3)
- These operations merely “request” some communication
  - MPI_Request variables can be used to track these requests
  - MPI_Wait blocks until an operation has finished
  - MPI_Test sets a flag if the operation has finished

```c
int MPI_Isend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv (void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *request, MPI_Status *status)
int MPI_Wait (MPI_Request *request, MPI_Status *status)
int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
```
Issues with point-to-point

- No global message order guarantees
  - Between any send/recv pair, messages are nonovertaking
    - If $p_1$ sends $m_1$ then $m_2$ to $p_2$, then $p_2$ must receive $m_1$ first
  - No guarantees about global ordering
  - Communication between all processes can be tricky
- Rank 0 must read input, distribute data, and collect results
  - Using point-to-point operations does not scale well
  - Need a more efficient method

- Collective operations provide correct and efficient built-in all-process communication
Tree-structured communication

**Broadcast**

```c
int MPI_Bcast (void  *buf,
               MPI_Datatype dtype,
               int    count,
               int    root,
               MPI_Comm comm)
```

**Reduction**

```c
int MPI_Reduce (void  *send_buf,
                void  *recv_buf,
                int    count,
                MPI_Datatype dtype,
                MPI_Op  op,
                int    root,
                MPI_Comm comm)
```
Tree-structured communication

**Broadcast**

```c
int MPI_Bcast (void *buf, MPI_Datatype dtype, int count, int root, MPI_Comm comm)
```

**Reduction**

```c
int MPI_Reduce (void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)
```

`can't be aliases`
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int my_rank;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

    // send rank id from rank 0 to all processes
    int data = my_rank;
    MPI_Bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Received data in rank %d: %d\n", my_rank, data);

    MPI_Finalize();
    return 0;
}
Collective reductions

- Reduction operations
  - `MPI_SUM`, `MPI_PROD`, `MPI_MIN`, `MPI_MAX`
- Collective operations are matched based on ordering
  - Not on source / dest or tag
  - Try to keep code paths as simple as possible

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a = 1; c = 2</td>
<td>a = 1; c = 2</td>
<td>a = 1; c = 2</td>
</tr>
<tr>
<td>1</td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI_Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI_Reduce(&amp;c, &amp;d, ...)</code></td>
</tr>
</tbody>
</table>

**NOTE**: Reductions with count > 1 operate on a per-element basis
MPI_Allreduce

- Combination of MPI_Reduce and MPI_Broadcast
  - More efficient “butterfly” communication pattern
Data distribution

- **MPI_Scatter** and **MPI_Gather**
  - **MPI_Allgather** (gather + broadcast)
  - Provides efficient data movement in common patterns
  - Send and receive buffers must be different (or use **MPI_IN_PLACE**)

- **Partitioning**: **block** vs. **cyclic**
  - Usually application-dependent (locality and task size)
  - Block is the default; use **MPI_Type_vector** for cyclic or block-cyclic

<table>
<thead>
<tr>
<th>Process</th>
<th>Block</th>
<th>Components</th>
<th>Block-cyclic Blocksize = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1 2 3</td>
<td>0 1 6 7</td>
</tr>
<tr>
<td>1</td>
<td>4 5</td>
<td>6 7</td>
<td>2 3 8 9</td>
</tr>
<tr>
<td>2</td>
<td>8 9</td>
<td>10 11</td>
<td>4 5 10 11</td>
</tr>
</tbody>
</table>
int main(int argc, char *argv[]) {
    int my_rank, num_ranks;
    int data[MAX_SIZE];

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_ranks);

    // initialize 'data' to dummy values
    for (int i = 0; i < num_ranks; i++) {
        data[i] = -1;
    }

    // send rank id from every process to rank 0
    MPI_Gather(&my_rank, 1, MPI_INT,
               data, 1, MPI_INT, 0, MPI_COMM_WORLD);

    // print 'data' at rank 0
    if (my_rank == 0) {
        printf("Received data in rank %d: ", my_rank);
        for (int i = 0; i < num_ranks; i++) {
            printf("%d ", data[i]);
        }
        printf("\n");
    }

    MPI_Finalize();
    return 0;
}
MPI Gather Example

```c
int src = my_rank;
int dst[num_ranks]; // assume num_ranks == 3
MPI_Gather(&src, 1, MPI_INT,
    dst, 1, MPI_INT, 0, MPI_COMM_WORLD);
not 3!
```

Before MPI_Gather

<table>
<thead>
<tr>
<th>Rank 0</th>
<th>Rank 1</th>
<th>Rank 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>0</td>
<td>src</td>
</tr>
<tr>
<td>dst</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After MPI_Gather

<table>
<thead>
<tr>
<th>Rank 0</th>
<th>Rank 1</th>
<th>Rank 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>src</td>
<td>0</td>
<td>src</td>
</tr>
<tr>
<td>dst</td>
<td>0 1 2</td>
<td></td>
</tr>
</tbody>
</table>
int src[2] = { my_rank, my_rank+1 };  
int dst[num_ranks*2];    // assume num_ranks == 3  
MPI_Gather(src, 2, MPI_INT,  
dst, 2, MPI_INT, 0, MPI_COMM_WORLD);  

not 6!
MPI collective summary

- **MPI_Bcast()**  Broadcast (one to all)
- **MPI_Reduce()**  Reduction (all to one)
- **MPI_Allreduce()**  Reduction (all to all)

- **MPI_Scatter()**  Distribute data (one to all)
- **MPI_Gather()**  Collect data (all to one)
- **MPI_Alltoall()**  Distribute data (all to all)
- **MPI_Allgather()**  Collect data (all to all)

(These four include “*v*” variants for variable-sized data)
MPI reference (PDF on website)

General

```c
int MPI_Init (int *argc, char ***argv)
int MPI_Finalize ()
int MPI_BARRIER (MPI_Comm comm)
double MPI_Wtime ()
```

```c
struct MPI_STATUS {
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
};
```

```c
int MPI_Comm_size (MPI_Comm comm, int *size)
int MPI_Comm_rank (MPI_Comm comm, int *rank)
Default communicator: MPI_COMM_WORLD
```

Point-to-point Operations

```c
int MPI_Send (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
int MPI_Ssend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm)
int MPI_Recv (void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Status *status)
(int maximum count)                     (MPI_ANY_SOURCE / MPI_ANY_TAG)                    (MPI_STATUS_IGNORE)
int MPI_Ssendrecv (void *send_buf, int send_count, MPI_Datatype send_dtype, int dest, int send_tag void *recv_buf, int recv_count, MPI_Datatype recv_dtype, int src, int recv_tag, MPI_Comm comm, MPI_Status *status)
int MPI_Isend (void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv (void *buf, int count, MPI_Datatype dtype, int src, int tag, MPI_Comm comm, MPI_Request *request, MPI_Status *status)
int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Wait (MPI_Request *request, MPI_Status *status)
int MPI_Get_count (MPI_Status *status, MPI_Datatype dtype, int *count)
```

Collective Operations

```c
int MPI_Bcast (void *buf, int count, MPI_Datatype dtype, int root, MPI_Comm comm)
int MPI_Reduce (void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op op, int root, MPI_Comm comm)
int MPI_Allreduce (void *send_buf, void *recv_buf, int count, MPI_Datatype dtype, MPI_Op op, MPI_Comm comm)
int MPI_Scatter (void *send_buf, void *recv_buf, int send_count, MPI_Datatype send_dtype, int recv_count, MPI_Datatype recv_dtype, int root, MPI_Comm comm)
int MPI_Gather (void *send_buf, void *recv_buf, int send_count, MPI_Datatype send_dtype, int recv_count, MPI_Datatype recv_dtype, int root, MPI_Comm comm)
int MPI_Allgather (void *send_buf, void *recv_buf, int send_count, MPI_Datatype send_dtype, int recv_count, MPI_Datatype recv_dtype, MPI_Comm comm)
int MPI_Alltoall (void *send_buf, void *recv_buf, int send_count, MPI_Datatype send_dtype, int recv_count, MPI_Datatype recv_dtype, MPI_Comm comm)
```
More collectives

- \textbf{MPI\_Reduce\_scatter}
  - Reduce on a vector, then distribute result

\begin{verbatim}
recvcnt = 1;
MPI\_Reduce\_scatter(sendbuf, recvbuf, recvcount,
                     MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD);
\end{verbatim}

\begin{tabular}{cccc}
  task0 & task1 & task2 & task3 \\
  \hline
  1 & 1 & 1 & 1 \\
  2 & 2 & 2 & 2 \\
  3 & 3 & 3 & 3 \\
  4 & 4 & 4 & 4 \\
\end{tabular}

\begin{tabular}{cccc}
  4 & 8 & 12 & 16 \\
\end{tabular}
More collectives

- **MPI_Scan**
  - Compute partial reductions

```c
count = 1;
MPI_Scan(sendbuf, recvbuf, count, MPI_INT,
          MPI_SUM, MPI_COMM_WORLD);
```

```
task0  task1  task2  task3
1      2      3      4
|     |     |     | sendbuf (before) |
1      3      6      10
|     |     |     | recvbuf (after) |
```
MPI datatypes

• MPI provides basic datatypes
  – MPI_INT, MPI_LONG, MPI_CHAR, etc.
• MPI also provides ways to create new datatypes
  – MPI_Type_contiguous: simple arrays
    int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
  – MPI_Type_vector: blocked and strided arrays
    • Useful for cyclic or block-cyclic data distributions
      int MPI_Type_vector(int count, int blocklength, int stride,
                           MPI_Datatype oldtype, MPI_Datatype *newtype)
  – Derived datatypes: records
  – New datatypes must be committed before they are used
    int MPI_Type_commit(MPI_Datatype *datatype)
**Derived datatypes**

- Goal: Pack related data together to reduce total messages
  - Very similar to C structs, but more detailed
  - Allows MPI to optimize internal representations

```c
MPI_Type_create_struct(5, array_of_block_lengths, array_of_displacements, array_of_types, &new_type)
```

- `array_of_block_lengths = (1, 2, 2, 1, 1)`
- `array_of_displacements = (0, 4, 8, 16, 20)`
- `array_of_types = (MPI_LB, MPI_CHAR, MPI_FLOAT, MPI_SHORT, MPI_UB)`
Virtual topologies

- It is often convenient for MPI to be aware of data decomposition details
- MPI provides built-in Cartesian system support.
  - MPI_Dims_create()
  - MPI_Cart_create()
  - MPI_Cart_get()
  - MPI_Cart_coords()
  - MPI_Cart_shift()

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(0,0)</td>
<td>(0,1)</td>
<td>(0,2)</td>
<td>(0,3)</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>(1,0)</td>
<td>(1,1)</td>
<td>(1,2)</td>
<td>(1,3)</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>(2,0)</td>
<td>(2,1)</td>
<td>(2,2)</td>
<td>(2,3)</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>(3,0)</td>
<td>(3,1)</td>
<td>(3,2)</td>
<td>(3,3)</td>
</tr>
</tbody>
</table>
Parallel file I/O (MPI-2)

- MPI provides a parallel file I/O interface
  - Uses derived data types to create per-process views of a file on disk
  - MPI_File_open()
  - MPI_File_set_view()
  - MPI_File_read_at()
  - MPI_File_read()
  - MPI_File_read_shared()
  - MPI_File_write_at()
  - MPI_File_write()
  - MPI_File_write_shared()
  - MPI_File_close()

Figure 13.2: Partitioning a file among parallel processes
One-sided communication (MPI-2)

- MPI provides remote memory access (RMA)
  - This allows programmers to take advantage of hardware-specific direct memory access features like DMA
  - MPI_Win_create()
  - MPI_Win_allocate()
  - MPI_Put()
  - MPI_Get()
  - MPI_Accumulate()
  - MPI_Win_free()
Non-blocking collectives (MPI-3)

- MPI now provides non-blocking forms of major collective operations
- Like `MPI_Irecv()`, these calls begin the communication and should be concluded with a call to `MPI_Wait()`
  - `MPI_Ibarrier()`
  - `MPI_Ibcast()`
  - `MPI_Igather()`
  - `MPI_Iscatter()`
  - `MPI_Iallgather()`
  - `MPI_Ialltoall()`
  - `MPI_Ireduce()`
  - `MPI_Iallreduce()`
  - `MPI_Ireduce_scatter()`
  - `MPI_Iscan()`
Why MPI_Ibarrier?

Why would you want a non-blocking barrier?

```
work1();
MPI_Barrier(MPI_COMM_WORLD);
work2();           // independent
work3();           // dependent on work1()

work1();
MPI_Request rq;
MPI_Ibarrier(MPI_COMM_WORLD, &rq);
work2();           // independent
MPI_Wait(&rq, MPI_STATUS_IGNORE);
work3();           // dependent on work1()
```

Version 1 vs. Version 2
Tools interface (MPI-3)

• MPI now provides a way to tweak parameters and access monitoring information in a cross-platform manner

• Control variables (cvar)
  – Startup options
  – Buffer sizes

• Performance variables (pvar)
  – Packets sent
  – Time spent blocking
  – Memory allocated
Distributed memory summary

- Distributed systems can scale massively
  - Hundreds or thousands of nodes, petabytes of memory
  - Millions/billions of cores, petaflops of computation capacity
- They also have significant issues
  - Non-uniform memory access (NUMA) costs
  - Requires explicit data movement between nodes
  - More difficult debugging and optimization
- Core design tradeoff: data distribution
  - How to partition, and what to send where (duplication?)
  - Goal: minimize data movement
  - Paradigm: computation is “free” but communication is not