Overview

- MPI overview
- MPI process creation
- MPI point-to-point communication
- MPI collective communications
- MPI groups and communicators
- MPI virtual topologies
- MPE and Jumpshot
- Further reading
MPI

- Message Passing Interface (MPI) is a standard with portable implementations
  - MPICH, LAM-MPI, OpenMPI, …
- Hardware platforms:
  - Distributed Memory
  - Shared Memory
    - Particularly SMP / NUMA architectures
  - Hybrid
    - SMP clusters, workstation clusters, and heterogeneous networks
- Parallelism is explicit
  - Programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs
- SPMD model with static process creation
  - MPI-2 allows dynamic process creation: `MPI_Comm_spawn()`
MPI Process Creation

- Static process creation
  - Start a $N$ processes running the same program `prog1`
    ```
    mpirun prog1 -np $N$
    ```
  - but this does not specify where the `prog1` copies run
  - Use batch processing tools, such as SGE, to run MPI programs on a cluster

- MPI-2 supports dynamic process creation:
  ```
  MPI_Comm_spawn()
  ```
MPI SPMD Computational Model

```c
main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    doWork();
    MPI_Finalize();
}
```

All processes execute this work

```c
doWork()
{
    int myrank;
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0)
        printf("I’m the master process");
}
```

All processes belong to the MPI_COMM_WORLD communicator, and each process has a rank from 0 to P-1 in the communicator

Use rank numbers to differentiate work
MPI Point-to-point Send Format

```
MPI_Send(&buf, count, datatype, dest, tag, comm)
```

- **Data parameters**
  - Address of send buffer with data
  - Number of items to send

- **Envelope parameters**
  - Data type of each item: `MPI_Datatype`
  - Rank of destination process: `int`
  - Message tag: `int`
  - Communicator: `MPI_Comm`
MPI Point-to-point Recv Format

```
MPI_Recv(&buf, count, datatype, src, tag, comm, &status)
```

- **Data parameters**
  - Address of buffer to collect data
  - Maximum number of items to receive
  - Data type of each item (MPI_Datatype)

- **Envelope parameters**
  - Rank of source process (int)
  - Message tag (int)
  - Communicator (MPI_Comm)
  - Status after operation (MPI_Status)
Example: Send-Recv Between two Processes

```c
main(int argc, char *argv[])
{
    int myrank;
    int value = 123;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_Rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0)
        MPI_Send(&value, 1, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD);
    else if (myrank == 1)
        MPI_Recv(&value, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    MPI_Finalize();
}
```
## MPI Point-to-point Communication Modes

<table>
<thead>
<tr>
<th>Communication mode</th>
<th>Blocking routines</th>
<th>Nonblocking routines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synchronous</strong></td>
<td>MPI_Ssend</td>
<td>MPI_Issend</td>
</tr>
<tr>
<td><strong>Ready</strong></td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
</tr>
<tr>
<td><strong>Buffered</strong></td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td><strong>Standard</strong></td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td>MPI_Irecv</td>
</tr>
<tr>
<td></td>
<td>MPI_Sendrecv</td>
<td>MPI_Sendrecv_replace</td>
</tr>
</tbody>
</table>

Roughly speaking:  
\[
\text{MPI\_Xsend} = \text{MPI\_I\text{X}send} + \text{MPI\_Wait} \\
\text{MPI\_Recv} = \text{MPI\_I\text{Recv}} + \text{MPI\_Wait}
\]
MPI Synchronous Send

MPI_Ssend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
**MPI Blocking Ready Send**

*Only sends when a receiver was posted, otherwise error*

```
MPI_Rsend( void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )
```
MPI Buffered Send

User-supplied buffer, where only one buffer can be active
Note: buffer size must be the maximum data size + MPI_BSEND_OVERHEAD + 7

MPI_Buffer_attach(void *buf, int size)

MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
MPI Blocking Standard Send
Small Message Size

MPI_SEND
(blocking standard send)
size < threshold

data transfer from source complete

S
R
int. buffer on receiver

MPI_RECV

task continues when data transfer to user's buffer is complete
MPI Blocking Standard Send
Large Message Size

The threshold value (also called the "eager limit") differs between systems.
MPI Nonblocking Standard Send
Small Message Size

\textbf{MPI} \texttt{ISEND} (nonblocking standard send)

size \leq \text{threshold}

\textbf{MPI} \texttt{WAIT}

no delay even though message is not yet in user's buffer on receiving node

\textbf{MPI} \texttt{IRECV}

transfer to buffer on receiving node can be avoided if \texttt{MPI} \texttt{IRECV} posted early enough

\textbf{MPI} \texttt{WAIT}

no delay if \texttt{MPI} \texttt{WAIT} is late enough

\begin{verbatim}
MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

MPI_Wait(MPI_Request *request, MPI_Status *status)
\end{verbatim}
MPI Nonblocking Standard Send
Large Message Size

MPI_ISEND (nonblocking standard send)

size > threshold

MPI_WAIT
data transfer from source complete

task waits

S
R

transfer doesn't begin until word has arrived
that corresponding MPI_Irecv has been posted

MPI_Irecv

MPI_WAIT

no interruption if wait is late enough
## MPI Communication Modes

<table>
<thead>
<tr>
<th>Communication mode</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Synchronous</strong></td>
<td>Safest, and therefore most portable</td>
<td>Can incur substantial synchronization overhead</td>
</tr>
<tr>
<td></td>
<td>SEND/RECV order not critical</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Amount of buffer space irrelevant</td>
<td></td>
</tr>
<tr>
<td><strong>Ready</strong></td>
<td>Lowest total overhead</td>
<td>RECV must precede SEND</td>
</tr>
<tr>
<td></td>
<td>SEND/RECV handshake not required</td>
<td></td>
</tr>
<tr>
<td><strong>Buffered</strong></td>
<td>Decouples SEND from RECV</td>
<td>Additional system overhead incurred by copy to buffer</td>
</tr>
<tr>
<td></td>
<td>No sync overhead on SEND</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Order of SEND/RECV irrelevant</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Programmer can control size of buffer space</td>
<td></td>
</tr>
<tr>
<td><strong>Standard</strong></td>
<td>Good for many cases</td>
<td>May not be suitable for your program</td>
</tr>
</tbody>
</table>
Example 1: Nonblocking Send and Blocking Recv

```c
main(int argc, char *argv[])
{
    int myrank;
    int value = 123;
    MPI_Status status;
    MPI_Request req;

    MPI_Init(&argc, &argv);

    MPI_Comm_Rank(MPI_COMM_WORLD, &myrank);

    if (myrank == 0)
    {
        MPI_Isend(&value, 1, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD, req);
        doWork(); /* do not modify value */
        MPI_Wait(&req, &status);
    }
    else if (myrank == 1)
        MPI_Recv(&value, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    MPI_Finalize();
}
```
Example 2:
Nonblocking Send/Recv

```c
main(int argc, char *argv[])
{
    int myrank;
    int value = 123;
    MPI_Status status;
    MPI_Request req;

    MPI_Init(&argc, &argv);
    MPI_Comm_Rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0)
    {
        MPI_Isend(&value, 1, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD, &req);
        doWork(); /* do not modify value */
        MPI_Wait(&req, &status);
    }
    else if (myrank == 1)
    {
        MPI_Irecv(&value, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &req);
        doWork(); /* do not read or modify value */
        MPI_Wait(&req, &status);
    }
    MPI_Finalize();
}
```
MPI Wait

**MPI_Wait** (MPI_Request *request, MPI_Status *status)
Waits until pending send/recv request is completed, sets **status**

**MPI_Waitall** (int count, MPI_Request *array_of_requests,
                    MPI_Status *array_of_statuses)
Wait for all pending send/recv requests to be completed, sets array of status values

**MPI_Waitany** (int count, MPI_Request *array_of_requests, int *index,
                  MPI_Status *status)
Wait until any of the pending send/recv requests is completed, sets **index** and **status**

**MPI_Waitsome** (int incount, MPI_Request *array_of_requests,
                  int *outcount, int *array_of_indices,
                  MPI_Status *array_of_statuses)
Wait until some of the pending send/recv requests is completed, sets arrays of index and status
MPI Test

**MPI_Test**(MPI_Request *request, int *flag, MPI_Status *status)
Check if pending send/recv request is completed (flag=1), or pending (flag=0), sets status

**MPI_Testall**(int count, MPI_Request *array_of_requests, int *flag, MPI_Status *array_of_statuses)
Check if all pending send/recv requests are completed (flag=1), sets array of status values

**MPI_Testany**(int count, MPI_Request *array_of_requests, int *index, int *flag, MPI_Status *status)
Check if any of the pending send/recv requests is completed (flag=1), sets index and status

**MPI_Testsome**(int incount, MPI_Request *array_of_requests, int *outcount, int* array_of_indices, MPI_Status *array_of_statuses)
Checks if some of the pending send/recv requests is completed, sets arrays of index and status
MPI Combined Sendrecv

**MPI_Sendrecv**
```c
(void *sendbuf, int sendcount, MPI_Datatype sendtype,
  int dest, int sendtag,
  void *recvbuf, int recvcount,
  MPI_Datatype recvtype, int source, int recvtag
  MPI_Comm comm, MPI_Status *status)
```
_Combined send and receive operation (blocking), with disjoint send and receive buffers_

**MPI_Sendrecv_replace**
```c
(void *buf, int count, MPI_Datatype datatype,
  int dest, int sendtag, int source, int recvtag,
  MPI_Comm comm, MPI_Status *status)
```
_Combined send and receive operation (blocking), with shared send and receive buffers_
Example 3: Combined Sendrecv

```c
main(int argc, char *argv[])
{
    int myrank, hisrank;
    float value;
    MPI_Status status;

    MPI_Init(&argc, &argv);

    MPI_Comm_Rank(MPI_COMM_WORLD, &myrank);
    hisrank = 1 - myrank;
    if (myrank == 0)
        value = 3.14;
    else
        value = 1.41;

    MPI_Sendrecv_replace(&value, 1, MPI_FLOAT, hisrank, MPI_ANY_TAG,
                         hisrank, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

    printf("Process %d got value %g\n", myrank, value);

    MPI_Finalize();
}
```
MPI Basic Datatypes

MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED

signed char
signed short int
signed int
signed long int
unsigned char
unsigned short int
unsigned int
unsigned long int
float
double
long double
MPI Collective Communications

- Coordinated communication within a group of processes identified by an MPI communicator
- Collective communication routines block until locally complete
- Amount of data sent must exactly match amount of data specified by receiver
- No message tags are needed
MPI Collective Communications Performance Considerations

- Communications are hidden from user
  - Communication patterns depend on implementation and platform on which MPI runs
  - In some cases, the root process originates or receives all data
  - Performance depends on implementation of MPI

- Communications may, or may not, be synchronized (implementation dependent)
  - Not always the best choice to use collective communication
  - There may be forced synchronization, which can be avoided with nonblocking point-to-point communications
MPI Collective Communications Classes

Three classes:

1. Synchronization
   - Barrier synchronization

2. Data movement
   - Broadcast
   - Scatter
   - Gather
   - All-to-all

3. Global computation
   - Reduce
   - Scan
MPI Barrier

MPI_Barrier(MPI_Comm comm)

A node invoking the barrier routine will be blocked until all the nodes within the group (communicator) have invoked it.
MPI Broadcast

**MPI_Broadcast** (void* buffer,
int count,
MPI_Datatype datatype,
int root,
MPI_Comm comm)

*Simple broadcast implementation: root sends data to all processes, which is efficient on a bus-based parallel machine*

*More efficient on a network: broadcast as a tree operation*

**Step 0**
**Step 1**
**Step 2**
**Step 3**

Log₂(P) steps

Total amount of data transferred: N (P-1)
MPI Broadcast Example

```
float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];
...
root = 0;
MPI_Broadcast(b, N, MPI_Float, root, MPI_COMM_WORLD);
...```
MPI Scatter

\texttt{MPI\_Scatter}(void *sbuf,
int scount,
\texttt{MPI\_Datatype} stype,
void *rbuf,
int rcount,
\texttt{MPI\_Datatype} rtype,
int root,
\texttt{MPI\_Comm} comm)
MPI Scatter

\texttt{MPI\textunderscore Scatter}(void *sbuf,
int scount,
MPI\_Datatype stype,
void *rbuf,
int rcount,
MPI\_Datatype rtype,
int root,
MPI\_Comm comm)

\begin{itemize}
\item \textbf{Step 0}
\item \textbf{Step 1}
\item \textbf{Step 2}
\item \textbf{Step 3}
\end{itemize}

\begin{align*}
\text{Total amount of data transferred:} & \quad N P \log_2(P)/2
\end{align*}
float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];
...
root = 0;
...
MPI_Scatter(A, N/P*N, MPI_Float, Ap, N/P*N, MPI_Float, root, MPI_COMM_WORLD);
MPI Gather

\texttt{MPI\_Gather}(\texttt{void *sbuf,} \\
\hspace{1cm} \texttt{int scount,} \\
\hspace{1cm} \texttt{MPI\_Datatype stype,} \\
\hspace{1cm} \texttt{void *rbuf,} \\
\hspace{1cm} \texttt{int rcount,} \\
\hspace{1cm} \texttt{MPI\_Datatype rtype,} \\
\hspace{1cm} \texttt{int root,} \\
\hspace{1cm} \texttt{MPI\_Comm comm})
MPI Gather Example

float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];

...  
  for (i = 1; i < N/P; i++)
  {
    cp[i] = 0;
    for (k = 0; k < N; k++)
      cp[i] = cp[i] + Ap[i][k] * b[k];
  }
  MPI_Gather(cp, N/P, MPI_Float, c, N/P, MPI_Float, root, MPI_COMM_WORLD);
MPI AllGather

MPI_AllGather(void *sbuf,
              int scount,
              MPI_Datatype stype,
              void *rbuf,
              int rcount,
              MPI_Datatype rtype,
              MPI_Comm comm)
float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];
...
for (i = 1; i < N/P; i++)
{
    cp[i] = 0;
    for (k = 0; k < N; k++)
        cp[i] = cp[i] + Ap[i][k] * b[k];
}
MPI_AllGather(cp, N/P, MPI_Float, c, N/P, MPI_Float, MPI_COMM_WORLD);
MPI Scatterv and (All)Gatherv

**MPI_Scatterv**

```c
MPI_Scatterv(void *sbuf,
             int *scounts,
             int *displs,
             MPI_Datatype stype,
             void *rbuf,
             int rcount,
             MPI_Datatype rtype,
             int root,
             MPI_Comm comm)
```

**MPI_Gatherv**

```c
MPI_Gatherv(void *sbuf,
            int scount,
            MPI_Datatype stype,
            void *rbuf,
            int *rcounts,
            int *displs,
            MPI_Datatype rtype,
            int root,
            MPI_Comm comm)
```

*Note: counts are same in this example*
MPI All to All

MPI_Alltoall(void *sbuf,
    int scount,
    MPI_Datatype stype,
    void *rbuf,
    int rcount,
    MPI_Datatype rtype,
    MPI_Comm comm)

Global transpose: the jth block from processor i is received by processor j and stored in ith block
MPI Reduce

```c
MPI_Reduce(void *sbuf,
           void *rbuf,
           int count,
           MPI_Datatype stype,
           MPI_Op op,
           int root,
           MPI_Comm comm)
```
MPI Reduce Example

float abcd[4], sum[4];

... 

MPI_Reduce(abcd, sum, 4, MPI_Float, root, MPI_SUM, MPI_COMM_WORLD);
MPI AllReduce

MPI_AllReduce(void *sbuf,
              void *rbuf,
              int count,
              MPI_Datatype stype,
              MPI_Op op,
              MPI_Comm comm)
MPI AllReduce Example

```c
float abcd[4], sum[4];

...  
MPI_AllReduce(abcd, sum, 4, MPI_Float, MPI_SUM, 
    MPI_COMM_WORLD);
```
MPI Reduce_scatter

MPI_Reduce_scatter(void *sbuf,
void *rbuf,
int *rcounts,
MPI_Datatype stype,
MPI_Op op,
MPI_Comm comm)

Same as Reduce followed by Scatter

Note: rcounts = number of elements received, which is >1 when N>P
MPI Scan

\[
\text{MPI\_Scan} (\text{void} *\text{sbuf}, \\
\text{void} *\text{rbuf}, \\
\text{int} \text{ count}, \\
\text{MPI\_Datatype} \text{ stype}, \\
\text{MPI\_Op} \text{ op}, \\
\text{MPI\_Comm} \text{ comm})
\]
## MPI Reduce and Scan Reduction Operators

<table>
<thead>
<tr>
<th>MPI_OP</th>
<th>Operation</th>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum</td>
<td>integer, float</td>
<td>integer, real, complex</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum</td>
<td>integer, float</td>
<td>integer, real, complex</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
<td>integer, float</td>
<td>integer, real, complex</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
<td>integer, float</td>
<td>integer, real, complex</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
<td>integer</td>
<td>logical</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bit-wise and</td>
<td>integer, MPI_BYTE</td>
<td>integer, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>logical or</td>
<td>integer</td>
<td>logical</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>bit-wise or</td>
<td>integer, MPI_BYTE</td>
<td>integer, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>logical xor</td>
<td>integer</td>
<td>logical</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>bit-wise xor</td>
<td>integer, MPI_BYTE</td>
<td>integer, MPI_BYTE</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max val and loc</td>
<td>float, double, long double</td>
<td>real, complex, double precision</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min val and loc</td>
<td>float, double, long double</td>
<td>real, complex, double precision</td>
</tr>
</tbody>
</table>
MPI Groups and Communicators

- A group is an ordered set of processes
  - Each process in a group is associated with a unique integer rank between 0 and $P-1$, with $P$ the number of processes in the group

- A communicator encompasses a group of processes that may communicate with each other
  - Communicators can be created for specific groups
  - Processes may be in more than one group/communicator

- Groups/communicators are dynamic and can be setup and removed at any time

- From the programmer's perspective, a group and a communicator are the same
MPI Groups and Communicators

Image source: Lawrence Livermore National Labs
MPI Group Operations

**MPI_Comm_group**
returns the group associated with a communicator

**MPI_Group_union**
creates a group by combining two groups

**MPI_Group_intersection**
creates a group from the intersection of two groups

**MPI_Group_difference**
creates a group from the difference between two groups

**MPI_Group_incl**
creates a group from listed members of an existing group

**MPI_Group_excl**
creates a group excluding listed members of an existing group

**MPI_Group_range_incl**
creates a group according to first rank, stride, last rank

**MPI_Group_range_excl**
creates a group by deleting according to first rank, stride, last rank

**MPI_Group_free**
marks a group for deallocation
 MPI Communicator Operations

- **MPI_Comm_size**
  returns number of processes in communicator's group
- **MPI_Comm_rank**
  returns rank of calling process in communicator's group
- **MPI_Comm_compare**
  compares two communicators
- **MPI_Comm_dup**
  duplicates a communicator
- **MPI_Comm_create**
  creates a new communicator for a group
- **MPI_Comm_split**
  splits a communicator into multiple, non-overlapping communicators
- **MPI_Comm_free**
  marks a communicator for deallocation
MPI Virtual Topologies

- A *virtual topology* can be associated with a communicator
- Two types of topologies supported by MPI
  - Cartesian (grid)
  - Graph
- Increases efficiency of communications
  - Some MPI implementations may use the physical characteristics of a given parallel machine
  - Introduces locality: low communication overhead with nodes that are “near” (few message hops), while distant nodes impose communication penalties (many hops)
MPE and Jumpshot

MPE_Log_get_state_eventIDs(int *startID, int *finalID)
Get a pair of event numbers to describe a state (see next)

MPE_Describe_state(int startID, int finalID, const char *name
                  const char *color)
Describe the state with a name and color (e.g. “red”, “blue”, “green”)]

MPE_Log_get_solo_eventID(int *eventID)
Get an event number to describe an event (see next)

MPE_Describe_event(int eventID, const char *name, const char *color)
Describe the event with a name and color (e.g. “red”, “blue”, “green”)

MPE_Log_event(int event, int data, const char *bytebuf)
Record event in the log, data is unused and bytebuf carries informational data are should be NULL.
Use two calls, one with the startID and the other with the finalID to log the time of a state.
MPE and Jumpshot

int event1, statels, state1f;
int myrank;

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPE_Log_get_solo_eventIDs(&event1);
MPE_Log_get_state_eventIDs(&statels, &state1f);

if (myrank == 0)
{  MPE_Describe_event(event1, "Start", "green");
    MPE_Describe_state(statels, state1f, "Computing", "red");
}

MPE_Log_event(event1, 0, NULL);
MPI_Bcast(...);
...
MPE_Log_event(statels, 0, NULL);
doComputation();
MPE_Log_event(state1f, 0, NULL);
MPI_Barrier();
MPI+MPE Compilation, Linking, and Run

- Compilation with MPE requires `mpe` and `lmpe` libs:
  - `mpicc -o myprog myprog.c -lmpe -llmpe`
  - Note: without `-llmpe` you must init and finalize the logs
- Run directly…
  - `mpirun -np 4 ./myprog`
- … or in batch mode (e.g. with SGE)
  - `qsub runmyprog.sh`
- This generates the log file (typically in home dir)
  - `myprog.clog2`
- Display log file with
  - `jumpshot`

```bash
#!/bin/bash
# All environment variables active within the qsub
# utility to be exported to the context of the job:
#$ -V
# use openmpi with 4 nodes
#$ -pe openmpi_4 4
# use current directory
#$ -cwd
mpirun -np $NSLOTS ./myprog
```
Jumpshot generates a space-time diagram from a log file
- Supports clog2 (through conversion) and newer slog2 formats
- Shows user-defined states (start-end) and single point events
  - www.cs.fsu.edu/~engelen/courses/HPC/cpiolog.c
  - www.cs.fsu.edu/~engelen/courses/HPC/fpiolog.f
Further Reading

- [PP2] pages 52-61
- Optional: Lawrence Livermore National Laboratories MPI Tutorial
  http://www.llnl.gov/computing/tutorials/mpi/