A crash course on MPI

Performance of laptop is only 12-14 years behind … which is about the time it takes to complete PhD ….
Parallel computing: a few things everyone should be familiar with...

- **Amdahl’s law:**
  \[ S_P = \frac{1}{\xi + (1-\xi) \frac{1}{P}} < \frac{T_1}{T_P} \]

- **Parallel efficiency**
  \[ \eta_P = \frac{P-1}{P \log_2 P} \]

- **Communication cost**
  \[ C = L + \beta l \]

  - \( L \) – latency, \( l \) – message size,
  - \( \beta^{-1} \) – bandwidth
Message Passing Interface - MPI: what do we need it for?
The minimal MPI subset

1. MPI_Init()
2. MPI_Finalize()
3. MPI_Comm_size()
4. MPI_Comm_rank()
5. MPI_Send()
6. MPI_Recv()

```c
#include <stdio.h>
#include <mpi.h>

int main (argc, *argv[ ]){
    int rank, size;
    MPI_Init (&argc, &argv);
    /* starts MPI */
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    /* get current process id */
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    /* get number of processes */

    printf( "Hello world from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```
MPI Communications

• **Point-to-point communications**
  – Involves a sender and a receiver
  – Only the two processors participate in communication

• **Collective communications**
  – All processors within a communicator participate in communication (by calling same routine, may pass different arguments);
  – Barrier, reduction operations, gather, scatter…
int **MPI_Send**  
    void *buf,  /* initial address of send buffer */  
    int count,  /* number of elements in send buffer (nonnegative integer) */  
    MPI_Datatype datatype,  /* datatype of each send buffer element */  
    int dest,  /* rank of destination (integer) */  
    int tag,  /* message tag (integer) */  
    MPI_Comm comm /* communicator */  
    );

int **MPI_Recv**  
    void *buf,  /* initial address of receive buffer */  
    int count,  /* number of elements in receive buffer (nonnegative integer) */  
    MPI_Datatype datatype,  /* datatype of each receive buffer element */  
    int dest,  /* rank of source (integer) */  
    int tag,  /* message tag (integer) */  
    MPI_Comm comm,  /* communicator */  
    MPI_Status *status /* status object */  
    );
Deadlock

MPI_Comm_rank(MPI_COMM_WORLD,&rank);
If(rank==0)
{
    MPI_Recv(buf1,count,MPI_DOUBLE,1,tag,comm);
    MPI_Send(buf2,count,MPI_DOUBLE,1,tag,comm);
}
else if (rank==1)
{
    MPI_Send(buf2,count,MPI_DOUBLE,0,tag,comm);
    MPI_Recv(buf1,count,MPI_DOUBLE,0,tag,comm);
}

MPI_Comm_rank(MPI_COMM_WORLD,&rank);
If(rank==0)
{
    MPI_Recv(buf1,count,MPI_DOUBLE,1,tag,comm);
    MPI_Send(buf2,count,MPI_DOUBLE,1,tag,comm);
}
else if (rank==1)
{
    MPI_Send(buf2,count,MPI_DOUBLE,0,tag,comm);
    MPI_Recv(buf1,count,MPI_DOUBLE,0,tag,comm);
int MPI_Sendrecv(
    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
);

Blocking point-to-point communication
```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int my_rank, ncpus;
    int left_neighbor, right_neighbor;
    int data_received;
    int send_tag = 101, recv_tag=101;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &ncpus);

    left_neighbor = (my_rank-1 + ncpus)%ncpus;
    right_neighbor = (my_rank+1)%ncpus;

    MPI_Sendrecv(&my_rank, 1, MPI_INT, left_neighbor, send_tag,
                   &data_received, 1, MPI_INT, right_neighbor, recv_tag,
                   MPI_COMM_WORLD, &status);

    printf("P%d received from right neighbor: P%d\n", my_rank, data_received);

    // clean up
    MPI_Finalize();
    return 0;
}
```

Output:
P3 received from right neighbor: P0
P2 received from right neighbor: P3
P0 received from right neighbor: P1
P1 received from right neighbor: P2
Non-blocking point-to-point communication

```c
int MPI_Isend(
    void *buf,       /* initial address of send buffer */
    int count,       /* number of elements in send buffer (nonnegative integer) */
    MPI_Datatype datatype, /* datatype of each send buffer element */
    int dest,        /* rank of destination (integer) */
    int tag,         /* message tag (integer) */
    MPI_Comm comm,   /* communicator */
    MPI_Request *request /* communication request */
);

int MPI_Irecv(
    void *buf,       /* initial address of receive buffer */
    int count,       /* number of elements in receive buffer (nonnegative integer) */
    MPI_Datatype datatype, /* datatype of each receive buffer element */
    int dest,        /* rank of source (integer) */
    int tag,         /* message tag (integer) */
    MPI_Comm comm,   /* communicator */
    MPI_Request *request /* communication request */
);
```
What should we use?

MPI_Send + MPI_Recv
MPI_Send + MPI_Irecv
MPI_Isend + MPI_Recv
MPI_Isend + MPI_Irecv
MPI_Sendrecv
MPI_Alltoall
Collective communication

```
int MPI_Reduce(
    void *sendbuf,       
    void *recvbuf,       
    int count,           
    MPI_Datatype datatype,
    MPI_Op op,           
    int root,            
    MPI_Comm comm)
```

<table>
<thead>
<tr>
<th>MPI function</th>
<th>Math Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>maximum</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>minimum</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>maximum and location of maximum</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>minimum and location of minimum</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>logical or</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>logical exclusive or</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bitwise and</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>bitwise or</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>bitwise exclusive or</td>
</tr>
</tbody>
</table>

Implemented in integration, dot products, finding maxima or minima ….
Collective communication

```c
int MPI_Gather (  
    void *sendbuf,  
    int sendcnt,  
    MPI_Datatype sendtype,  
    void *recvbuf,  
    int recvcount,  
    MPI_Datatype recvtype,  
    int root,  
    MPI_Comm comm  
);
```
Collective communication

```
int MPI_Scatter (  
    void *sendbuf,  
    int sendcnt,  
    MPI_Datatype sendtype,  
    void *recvbuf,  
    int recvcount,  
    MPI_Datatype recvtype,  
    int root,  
    MPI_Comm comm  
);  
```
Collective communication

To operate on messages of unequal length:
• MPI_Scatterv
• MPI_Gatherv

To obtain results on all processors:
• MPI_Allreduce
• MPI_Allgather (v)

To Send data from all to all processes
• MPI_Alltoall(v)

To broadcast message
• MPI_Bcast

To synchronize between processors
• MPI_Barrier
Good programming

- **Reliability** – The code does not have errors and can be trusted to compute what it is supposed to compute.

- **Robustness**, which is closely related to reliability – The code has a wide range of applicability as well as the ability to detect bad data, “singular” or other problems that it can not be expected to handle, and other abnormal situations, and deal with them in a way that is satisfactory to user.

- **Portability** – The code can be transferred from one computer to another with a minimum effort and without losing reliability. Usually this means that the code has been written in a general high-level language like FORTRAN (C++) and uses no “tricks” that are dependent on the characteristic of a particular computer. Any machine characteristics that must be used are clearly delineated.

- **Maintainability** – Any code will necessary need to be changed from time to time, either to make corrections or to add enhancements, and this should be possible with minimum effort.

Code optimization through code profiling

Difference: 40 sec → 25 sec
Debugging parallel code

DDT

Totalview
Performance analysis tools

pgprof

vampire

crayPAT

100.0% | 100.0% | 512 | Total

------------------------------------
| 59.8% | 59.8% | 306 | stepfx_
| 17.6% | 77.3% | 90  | getrusage
| 8.0%  | 85.4% | 41  | stepfy_
| 6.2%  | 91.6% | 32  | integr_
| 2.0%  | 93.6% | 10  | gradco_
| 1.0%  | 94.5% | 5   | __write
| 0.8%  | 95.3% | 4   | filerx_ |

IMP
How to learn MPI programming?

Just do it!