Message Passing Interface (MPI-1)
(PSC Appendix C, §C.1–C.2.4)
History of MPI

- 1994: **Message Passing Interface** (MPI) became available as a standard interface for parallel programming in C and Fortran 77.
- Designed by a committee called the **MPI Forum** consisting of computer vendors, users, computer scientists.
- Based on **sending and receiving messages** by a pair of processors. One processor sends; the other receives. Both are active in the communication.
- Underlying model: **communicating sequential processes** (CSP) proposed by Tony Hoare in 1978.
- MPI itself is **not a model**. BSP is a model.
- MPI is an **interface** for a communication library, like BSPlib.
Recent history of MPI

- 1997: MPI-2 standard defined. Added functionality:
  - one-sided communications (put, get, sum)
  - dynamic process management
  - parallel input/output
  - languages C++ and Fortran 90

- 2003: first full implementations of MPI-2 arrive, namely MPICH (Argonne National Labs) and LAM/MPI (Indiana University).

- 2004–: Open MPI. Open-source project, merges 3 MPI implementations: LAM/MPI, FT-MPI (University of Tennessee), LA-MPI (Los Alamos National Laboratory).

- Many users still use MPI-1, particularly its latest version MPI-1.3.

Why use MPI?

- It is available on almost every parallel computer, often in an optimised version provided by the vendor. Thus MPI is the most portable communication library.
- Many libraries are available written in MPI, such as the numerical linear algebra library ScaLAPACK.
- You can program in many different ways using MPI, since it is highly flexible.
Why not?

▶ It is **huge**: the full standard has about 300 primitives. The user has to make many choices.
▶ It is **not so easy** to learn. Usually one starts with a small subset of MPI. Full knowledge of the standard is hard to attain.
▶ The one-sided communications of MPI-2 and MPI-3 are rather complicated. If you like one-sided communications you may want to consider BSPlib as an alternative.
Ping pong benchmark

- The cost of communicating a message of length $n$ is

$$T(n) = t_{\text{startup}} + nt_{\text{word}}.$$ 

Here, $t_{\text{startup}}$ is a fixed startup cost and $t_{\text{word}}$ is the additional cost per data word communicated.

- Communication of a message (in its blocking form) synchronises the sender and receiver. This is pairwise synchronisation, not global.

- Parameters $t_{\text{startup}}$ and $t_{\text{word}}$ are usually measured by sending a message from one processor to another and back: ping pong.

- The message length is varied in the ping pong benchmark.

- There is only one ping pong ball on the table.
Send and receive primitives

if (s==2)
    MPI_Send(x,5,MPI_DOUBLE,3,0,MPI_COMM_WORLD);
if (s==3)
    MPI_Recv(y,5,MPI_DOUBLE,2,0,MPI_COMM_WORLD,
             &status);

- Processor $P(2)$ sends 5 doubles to $P(3)$.
- $P(2)$ reads the data from its array $x$. After transmission, $P(3)$ writes these data into its array $y$.
- The integer ‘0’ is a tag for distinguishing between different messages from the same source processor to the same destination processor.
- MPI_Send and MPI_Recv are of fundamental importance in MPI.
Communicator: the whole processor world

```c
if (s==2)
    MPI_Send(x,5,MPI_DOUBLE,3,0,MPI_COMM_WORLD);
if (s==3)
    MPI_Recv(y,5,MPI_DOUBLE,2,0,MPI_COMM_WORLD, &status);
```

- A **communicator** is a subset of processors forming a communication environment with its own processor numbering.
- **MPI_COMM_WORLD** is the communicator consisting of all the processors.
Send/Receive considered harmful

- 1968: Edsger Dijkstra, guru of structured programming, considered the **Go To** statement harmful in sequential programming.
- Go To was widely used in Fortran programming in those days. It caused **spaghetti code**: if you pull something here, something unexpected moves there.
- No one dares to use Go To statements any more.
- Send/Receive in parallel programming has the same dangers, and even more, since several diners eat from the same plate.
- Pull here, pull there, nothing moves: deadlock.
- **Deadlock** may occur if \( P(0) \) wants to send a message to \( P(1) \), and \( P(1) \) to \( P(0) \), and both processors want to send before they receive.
int main(int argc, char **argv){

    int p, s, n;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&s);

    if (s==0){
        printf("Please enter n:\n");
        scanf("%d", &n);
        if (n<0)
            MPI_Abort(MPI_COMM_WORLD,-1);
    }
    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
    ...

Collective communication: broadcast

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(buf, count, datatype, root, communicator);

- Broadcast count data items of a certain datatype from processor root to all others in the communicator, reading from location buf and also writing it there.
- All processors of the communicator participate.
- Extensive set of collective communications available in MPI. Using these reduces the size of program texts.
Inner product program mpiinprod (cont’d)

...  

nl = nloc(p, s, n);

x = vecallocd(nl);

for (i=0; i<nl; i++){
    iglob = i*p+s;
    x[i] = iglob+1;
}

/* global sync for timing */
MPI_Barrier(MPI_COMM_WORLD);

alpha = mpiip(p, s, n, x, x);

MPI_Barrier(MPI_COMM_WORLD);

time1 = MPI_Wtime();

MPI_Finalize();

exit(0);
double mpiip(int p, int s, int n, double *x, double *y) {
    double inprod, alpha;
    int i;

    inprod = 0.0;
    for (i = 0; i < nloc(p, s, n); i++)
        inprod += x[i] * y[i];
    MPI_Allreduce(&inprod, &alpha, 1, MPI_DOUBLE,
                  MPI_SUM, MPI_COMM_WORLD);

    return alpha;
}
Collective communication: reduce

MPI_Allreduce(&inprod, &alpha, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

MPI_Allreduce(sendbuf, recvbuf, count, datatype, operation, communicator);

- The reduction operation by MPI_Allreduce sums the double-precision local inner products inprod, leaving the result alpha on all processors.
- One can also do this for an array instead of a scalar, by changing the parameter 1 to the array size count, or perform other operations, such as taking the maximum, by changing MPI_SUM to MPI_MAX.
Benchmark: which primitive to measure?

- Benchmarking all communication primitives in MPI is a lot of work. This does not appeal to us.
- A typical MPI user would look first if there is a suitable collective-communication primitive that would do the job.
- This would lead to shorter program texts, and is good practice from the BSP point of view as well.
- Therefore, we choose a collective communication as the operation to be benchmarked.
- The BSP superstep, where every processor can communicate in principle with all others, is reflected best by the all-to-all primitives from MPI.
- Using an all-to-all primitive gives the MPI system the best opportunities for optimisation, similar to supersteps in BSPlib programs.
Measure time of MPI_Alltoallv

MPI_Barrier(MPI_COMM_WORLD);
time0= MPI_Wtime();

for (iter=0; iter<NITERS; iter++){
    MPI_Alltoallv(src,Nsend,Offset_send,MPI_DOUBLE,
                   dest,Nrecv,Offset_recv,MPI_DOUBLE,
                   MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
}

time1= MPI_Wtime();
time= time1-time0;
Syntax of MPI\textunderscore Alltoallv

MPI\textunderscore Alltoallv(src, Nsend, Offset\textunderscore send, datatype\textunderscore send, 
dest, Nrecv, Offset\textunderscore recv, datatype\textunderscore recv, 
communicator);

▶ So-called \textbf{vector variant} allows a varying number of data to be 
    sent (or even no data).

▶ The sender reads $N_{\text{send}}[t]$ data from array $\text{src}$ starting at 
   $\text{Offset\textunderscore send}[t]$ for each processor $P(t)$, $0 \leq t < p$, and 
   sends these data.

▶ The receiver receives data from all processors, and stores 
   them in array $\text{dest}$, with $N_{\text{recv}}[t]$ data arriving from 
   processor $P(t)$ at offset $\text{Offset\textunderscore recv}[t]$.

▶ All offsets are measured in units of the data type involved, 
   e.g. MPI\textunderscore DOUBLE. (Not in raw bytes, like in BSPlib).
Initialise $h$-relation

for (i=0; i<h; i++)
    src[i] = (double)i;
if (p==1){
    Nsend[0] = Nrecv[0] = h;
} else {
    for (s1=0; s1<p; s1++)
        Nsend[s1] = h/(p-1);
    for (i=0; i < h%(p-1); i++)
        Nsend[(s+1+i)%p]++; /* one extra */
    Nsend[s] = 0; /* no talking to yourself */

    for (s1=0; s1<p; s1++)
        Nrecv[s1] = h/(p-1);
    for (i=0; i < h%(p-1); i++)
        Nrecv[(s-1-i+p)%p]++;;
    Nrecv[s] = 0;
}
Determine offsets

Offset_send[0] = 0;
Offset_recv[0] = 0;

for(s1=1; s1<p; s1++){
    Offset_send[s1] = Offset_send[s1-1] + Nsend[s1-1];
    Offset_recv[s1] = Offset_recv[s1-1] + Nrecv[s1-1];
}

Messages are stored in order of destination processor. Thus, offsets can be computed by a prefix operation.
LU decomposition function mpilu

```c
void mpilu(int M, int N, int s, int t, int n,
    int *pi, double **a)
{

    MPI_Comm row_comm_s, col_comm_t;

    /* Create a new communicator for
       my processor row and column */
    MPI_Comm_split(MPI_COMM_WORLD,s,t,&row_comm_s);
    MPI_Comm_split(MPI_COMM_WORLD,t,s,&col_comm_t);
    ...

    // 2D numbering directly available in MPI: create a
    // communicator for every processor row and column by splitting
    // the world communicator.
```
Splitting a communicator

MPI_Comm_split(MPI_COMM_WORLD,s,t,&row_comm_s);

- Processors that call MPI_Comm_split with the same value of \( s \) end up in the same communicator, which we call row_comm_s.
- Thus, we obtain \( M \) communicators, each corresponding to a processor row \( P(s, \ast) \).
- Every processor obtains a processor number within its communicator. This number is by increasing value of the third parameter of the primitive, i.e., \( t \).
- Broadcast of pivot value within processor column, i.e., within communicator col_comm_t now becomes:

```c
if (k\%N==t)
    MPI_Bcast(&pivot,1,MPI_DOUBLE,smax,col_comm_t);
```
Swapping the permutation in $P(\ast, 0)$

```c
/* piece of code for k%M != r%M */
if (k%M==s){
    MPI_Send(&pi[k/M],1,MPI_INT,r%M,0,MPI_COMM_WORLD);
    MPI_Recv(&pi[k/M],1,MPI_INT,r%M,0,MPI_COMM_WORLD, &status);
}
if (r%M==s){
    MPI_Recv(&tmp,1,MPI_INT,k%M,0,MPI_COMM_WORLD, &status);
    MPI_Send(&pi[r/M],1,MPI_INT,k%M,0, MPI_COMM_WORLD);
    pi[r/M]= tmp;
}

// Don't change the order of the sends and receives!
(Punishment: deadlock on certain machines.)
```
Sender info must be initialised for FFT

```c
offset= 0;
j0= s%c0; j2= s/c0;
for(j=0; j<npackets; j++)
{
    jglob= j2*c0*np + j*c0 + j0;
destproc= (jglob/(c1*np))*c1 + jglob%c1;
    Nsend[destproc]= 2*size;
    Offset_send[destproc]= offset;
    for(r=0; r<size; r++)
    {
        tmp[offset + 2*r]= x[2*(j+r*ratio)];
        tmp[offset + 2*r+1]= x[2*(j+r*ratio)+1];
    }
    offset += 2*size;
}
```

- `mpifft` is identical to `bspffft`, except for redistribution. Packets are the same.
Receiver info must also be initialised

... 
/* Initialise receiver info */
offset= 0;
j0= s%c1; j2= s/c1;
for(r=0; r<npackets; r++){
   j= r*size;
jglob= j2*c1*np + j*c1 + j0;
srcproc= (jglob/(c0*np))*c0 + jglob%c0;
Nrecv[srcproc]= 2*size;
Offset_recv[srcproc]= offset;
offset += 2*size;
}
MPI_Barrier(MPI_COMM_WORLD); /* for safety */
MPI_Alltoallv(tmp,Nsend,Offset_send,MPI_DOUBLE,
x, Nrecv,Offset_recv,MPI_DOUBLE,
MPI_COMM_WORLD);
Summary

- The Message Passing Interface (MPI) is a highly portable communication library supported by most vendors of parallel computers.
- In MPI, you should try to use collective communications as much as possible. They reduce the size of program texts, and they also create supersteps, thus structuring the program in BSP style.
- MPI rule:
  
  *collective communications may synchronise the processors, but you cannot rely on this.*

So feel free to add global synchronisations where needed.