Distributed-Memory Programming Models II

Stefan Lang

Interdisciplinary Center for Scientific Computing (IWR)
University of Heidelberg
INF 368, Room 532
D-69120 Heidelberg
phone: 06221/54-8264
email: `Stefan.Lang@iwr.uni-heidelberg.de`

WS 15/16

Distributed-Memory Programming Models II

Communication by message passing

- MPI Standard
- Global communication for different topologies
  - Array (1D / 2D / 3D)
  - Hypercube
- Local exchange

MPI: Introduction

The *Message Passing Interface* (MPI) is a portable library of functions for message exchange between processes.

- MPI has been designed 1993/94 by an international gremium.
- Is available on nearly all platforms, including the free implementations OpenMPI, MPICH and LAM.
- Characteristics:
  - Library for binding with C-, C++- and FORTRAN programs (no language extension).
  - Large choice of point-to-point communication functions.
  - Global communication.
  - Data conversion for heterogeneous systems.
  - Creation of partial sets and topologies.
- MPI consists of over 125 functions, that are described on over 800 pages in the standard. Thus we can only discuss a small choice of its functionality.
- MPI-1 has no possibilities for dynamic process generation, this is possible in
  MPI-2, furthermore in-/output.
  MPI-3 is released since 09/2012 with minor extensions.

MPI: Hello World

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

int main (int argc, char *argv[])
{
    int my_rank, P;
    int dest, source;
    int tag=50;
    char message[100];
    MPI_Status status;

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

    if (my_rank!=0)
    {
        sprintf(message,"I am process %d\n",my_rank);
        dest = 0;
        MPI_Send(message,strlen(message)+1,MPI_CHAR,
                 dest,tag,MPI_COMM_WORLD);
    }
    else
    {
        puts("I am process 0\n");
        for (source=1; source<P; source++)
        {
            MPI_Recv(message,100,MPI_CHAR,source,tag,
                     MPI_COMM_WORLD,&status);
            puts(message);
        }
    }
    MPI_Finalize();

    return 0;
}
```

- SPMD style!
- Compilation and startup is done with

  ```
  mpicc -o hello hello.c
  mpirun -machinefile machines -np 8 hello
  ```

- `machines` contains names of the usable machines.

MPI: Blocking Communication I

- MPI supports different variants of blocking and non-blocking communication, guards for the **receive** function, as well as data conversion during communication between machines with distinct data formats.
- The fundamental blocking communication functions are defined by:

```
int MPI_Send(void *message, int count, MPI_Datatype dt,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *message, int count, MPI_Datatype dt,
             int src, int tag, MPI_Comm comm,
             MPI_Status *status);
```

- A message in MPI consists of plain *data* and an envelope (meta information).
- Data is always an array of elementary data types. This enables MPI to handle data conversion.

MPI: Blocking Communication II

- The envelope consists of:
  1. Number of sender,
  2. Number of receiver,
  3. Tag,
  4. and a Communicator.
- Number of sender and receiver is called rank.
- Tag is also an integer number and serves as identificator for different messages between identical communication partners.
- A communicator is defined by a partial set of the processes and a communication context. Messages, that belong to different contexts,do not influence each other, resp. sender and receiver have to use the same communicator.
- Meanwhile we only use the default communicator `MPI_COMM_WORLD` (all started processes).

MPI: Blocking Communication III

- `MPI_Send` is fundamentally blocking, there are however diverse variants:
  - *buffered send* (`B`): If the receiver has still not executed the corresponding **recv** function, the message is buffered on sender side. A „buffered send“ is, while assuming enough buffer space, always immediately finished. In comparison to asynchronous communication can the send buffer `message` be reused immediately.
  - *synchronous send* (`S`): Finishing of synchronous send indicates, that the receiver executes a **recv** function and has started to read the data.
  - *ready send* (`R`): A ready send may only be executed, if the receiver has already executed the corresponding **recv**. Otherwise the call results in an error.
- The according calls are designated `MPI_Bsend`, `MPI_Ssend` and `MPI_Rsend`.
- The `MPI_Send` instruction has either the semantics of `MPI_Bsend` or `MPI_Ssend`, according to implementation specifics. Therefore `MPI_Send` can, but must not block. In every case the send buffer `message` can be reused immediately after finishing.

MPI: Blocking Communication IV

- The instruction `MPI_Recv` is in every case blocking.
- The argument `status` contains source, tag, and error status of the receiving message.
- For the arguments `src` and `tag` can the values `MPI_ANY_SOURCE` resp. `MPI_ANY_TAG` be inserted. Thus `MPI_Recv` contains the functionality of **recv_any**.
- A non-blocking guard function for the receiving of messages is available by means of

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,
               int *flag, MPI_Status *status);
```

.

MPI: Non-blocking and Global Communication I

- For non-blocking communication there are the functions

```
int MPI_ISend(void *buf, int count, MPI_Datatype dt,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *req);
int MPI_IRecv(void *buf, int count, MPI_Datatype dt,
              int src, int tag, MPI_Comm comm,
              MPI_Request *req);
```

  available.
- Via the `MPI_Request` objects it is possible to determine the state of the communication request (corresponds to **msgid** in our pseudo code).
- Herefore exists (beneath other) the function

```
int MPI_Test(MPI_Request *req, int *flag, MPI_Status
```

- The `flag` is set to `true` ($\neq 0$), if the communication denoted by `req` has been finished. In this case `status` contains information about sender, receiver and error status.

  It needs to be considered, that the `MPI_Request` object gets invalid as soon as `MPI_Test` returns with `flag==true`. It may then not be used again.

MPI: Non-blocking and Global Communication II

- For global communication are available (beneath other):

  ```
  int MPI_Barrier(MPI_Comm comm);
  ```

  blocks all processes of a communicator until all are there.

-
  ```
  int MPI_Bcast(void *buf, int count, MPI_Datatype dt,
                int root, MPI_Comm comm);
  ```

  distributes the message in process `root` to all other processes of the communicator.

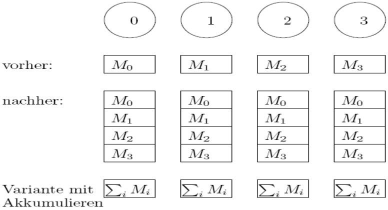
- For the collection of data different operations are present. We describe only one of these:

  ```
  int MPI_Reduce(void *sbuf, void *rbuf, int count, MPI_Datatype
                 MPI_Op op, int root, MPI_Comm comm);
  ```

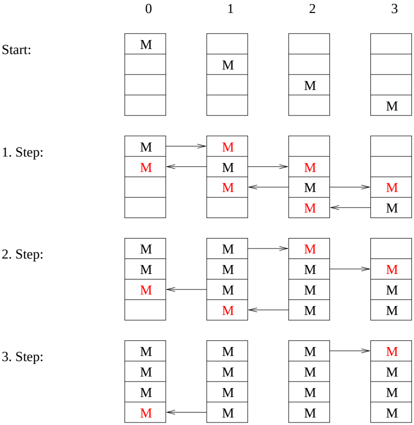
  combines the data in the input buffer `sbuf` of all processes by the associative operator op. The final result is available in the receive buffer `rbuf` of the process `root`. Examples for op are `MPI_SUM`, `MPI_MAX`.

All-to-all: 1D Array, Principle

Each wants to send data to all (variant: accumulate with associative operator):



We skip the ring topology and consider the 1D array at once: Each process sends into both directions.



We use synchronous communication. Decide who sends/receives by black-white coloring:

All-to-all: 1D Array, Code I

Program (All-to-all in 1D array)

```
parallel all-to-all-1D-array
{
    const int $P$;
    process $\Pi$[int $p \in \{0, \ldots, P-1\}$]
    {
        void all_to_all_broadcast(msg $m[P]$)
        {
            int $i$,
                from_left$= p - 1$, from_right$= p + 1$,
                                                        // I receive that
                to_left$= p$, to_right$= p$;            // I send that
            for ($i = 1; i < P; i++$)                   // $P - 1$ steps
            {
                if ($(p\%2) == 1$)                      // black/white coloring
                {
                    if (from_left $\geq 0$) recv($\Pi_{p-1}$, $m$[from_left]);
                    if (to_right $\geq 0$) send($\Pi_{p+1}$, $m$[to_right]);
                    if (from_right $< P$) recv($\Pi_{p+1}$, $m$[from_right]);
                    if (to_left $< P$) send($\Pi_{p-1}$, $m$[to_left]);
                }
                else
                {
                    if (to_right $\geq 0$) send($\Pi_{p+1}$, $m$[to_right]);
                    if (from_left $\geq 0$) recv($\Pi_{p-1}$, $m$[from_left]);
                    if (to_left $< P$) send($\Pi_{p-1}$,$m$[to_left]);
                    if (from_right $< P$) recv($\Pi_{p+1}$, $m$[from_right]);
                }
                ...
```

All-to-all: 1D Array, Code II

Program (All-to-all in 1D array cont.)

```
parallel all-to-all-1D-feld cont.
{
                    . . .

                    from_left−−; to_right−−;
                    from_right++; to_left++;
               }
          }
          . . .
          m[p] =„That is from p!“;
          all_to_all_broadcast(m);
          . . .
     }
}
```

All-to-all: 1D Array, Runtime

- For the runtime analysis consider P odd, $P = 2k + 1$:

$$\underbrace{\Pi_0, \dots, \Pi_{k-1},}_{k}\ \Pi_k,\ \underbrace{\Pi_{k+1}, \dots, \Pi_{2k}}_{k}$$

Process Π_k	receives	k	from left
	sends	$k+1$	to right
	receives	k	from right
	sends	$k+1$	to left.
	$\sum =$	$4k+2$	
		$= 2P$	

- After that Π_k has all messages. Now the message from 0 has to be send to $2k$ and vice versa. This needs again additonal

$$(\underbrace{k}_{\text{Entfernung}} - 1) \cdot \underbrace{2}_{\substack{\text{senden u.}\\ \text{empfangen}}} + \underbrace{1}_{\substack{\text{der Letzte}\\ \text{empfängt nur}}} = 2k - 1 = P - 2$$

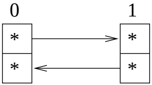
so we have in total

$$T_{all-to-all-array-1d} = (t_s + t_h + t_w \cdot n)(3P - 2)$$

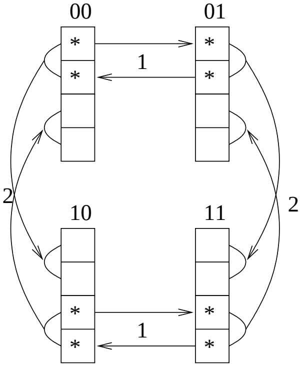
All-to-all: Hypercube

The following algorithm for the hypercube is known as *dimension exchange* and is again derived recursively.

Start with $d = 1$:



With four processes exchange processes 00 and 01 resp. 10 and 11 first their data, then exchange 00 and 10 resp. 01 and 11 each two data



All-to-all: Hypercube

```
void $all\_to\_all\_broadcast$(msg $m[P]$) {
    int $i$, $mask = 2^d - 1$, $q$;
    for ($i = 0$; $i < d$; $i++$) {
        $q = p \oplus 2^i$;
        if ($p < q$) {                                  // who first?
            send($\Pi_q$,$m[p\&mask]$,...,$m[p\&mask + 2^i - 1]$);
            recv($\Pi_q$,$m[q\&mask]$,...,$m[q\&mask + 2^i - 1]$);
        }
        else {
            recv($\Pi_q$,$m[q\&mask]$,...,$m[q\&mask + 2^i - 1]$);
            send($\Pi_q$,$m[p\&mask]$,...,$m[p\&mask + 2^i - 1]$);
        }
        $mask = mask \oplus 2^i$;
    }
}
```

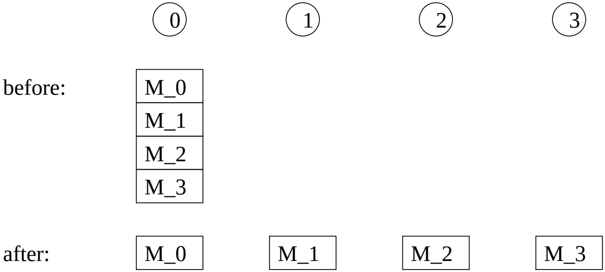
- Runtime analysis:

$$\begin{aligned} T_{all-to-all-bc-hc} &= \underbrace{2}_{\substack{\text{send a.}\\ \text{receive}}} \sum_{i=0}^{\operatorname{ld} P-1} t_s + t_h + t_w \cdot n \cdot 2^i = \\ &= 2 \operatorname{ld} P (t_s + t_h) + 2 t_w n (P - 1). \end{aligned}$$

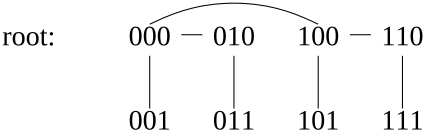
- For large messages the HC has no advantage: Each has to receive *n* words from each, whatever the topology looks like.

One-to-all with indiv. messages: Hypercube, Principle

- Process 0 sends to each a message, but to each a different one!



- Example is the in/output to a *single* file.
- Because of variation purposes we consider the output, this means all-to-one with indidvidual messages.
- We use the well-known hypercube structure:



One-to-all with indiv. messages: Hypercube, Code I

Program (*Collection* of individual messages on the hypercube)

```
parallel all-to-one-personalized
{
      const int $d, P = 2^d$;
     process $\Pi$[int $p \in \{0, \ldots, P-1\}$]{
          void all_to_one_pers(msg m) {
               int mask, i, q, root;
               // determine p's root: How many bits from end are zero?
               $mask = 2^d - 1$;
               for ($i = 0$; $i < d$; $i++$)
               {
                    $mask = mask \oplus 2^i$;
                    if ($p \& mask \neq p$) break;
               } // $p = p_{d-1} \ldots p_{i+1} \underbrace{1}_{\text{set to 0 at last in mask}} \underbrace{0 \ldots 0}_{i-1,\ldots,0}$

               if ($i < d$) $root = p \oplus 2^i$;                    // my root direction

               // own data
               if ($p == 0$) self-processing(m);
               else send(root,m);                                    // pass up

               ...
}
```

One-to-all with indiv. messages: Hypercube, Code II

Program (*Collection* of individual messages on the hypercube cont.)

```
parallel all-to-one-personalized cont.
{
            ...

            // process sub-trees:
            $mask = 2^d - 1$;
            for ($i = 0$; $i < d$; $i++$) {
                $mask = mask \oplus 2^i$; $q = p \oplus 2^i$;
                if ($p \& mask == p$)
                                                  // $p = p_{d-1} \dots p_{i+1} \quad 0 \quad \underbrace{0 \dots 0}_{i-1,\dots,0}$
                                                  // $q = p_{d-1} \dots p_{i+1} \quad 1 \quad \underbrace{0 \dots 0}_{i-1,\dots,0}$
                                                  // $\Rightarrow$ I am root of a HC of dim. $i + 1$!
                    for ($k = 0$; $k < 2^i$; $k++$) {
                        recv($\Pi_q$,m);
                        if ($p == 0$) process(m);
                        else send($\Pi_{root}$,m);
                    }
            }
        }
    }
}
```

One-to-all with indiv. messages: Runtime, Variants

For the *runtime* one has for large (n) messages

$$T_{all-to-one-pers} \geq t_w n(P-1)$$

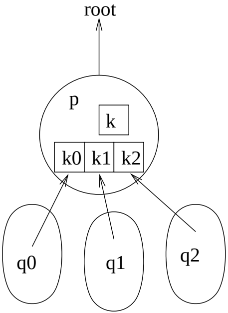
because of the pipelining.

Some variants are worth considering:

- *Individual length of messages:* Here sends one before sending the message itself only the length information (this is practically necessary → MPI).
- *Arbitrary message length* (but only finite intermediate buffer!): subdivide message into packets of fixed length.
- *Sorted output:* Each message M_i (of process i) is associated a sorting key k_i. The messages should be processed by process 0 in increasing order of keys, *without* intermediate buffering of all messages.

One-to-all with indiv. messages: Runtime, Variants

- With *sorted output* one may be inspired by the following idea:



p has three „servants“, q_0, q_1, q_2, that represent complete subtrees.
Each q_i sends its next smallest key to p, that searches the smallest key and then itself passes this key with its already transmitted data further.