Distributed-Memory Programming Models I

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 14/15

Distributed-Memory Programming Models I

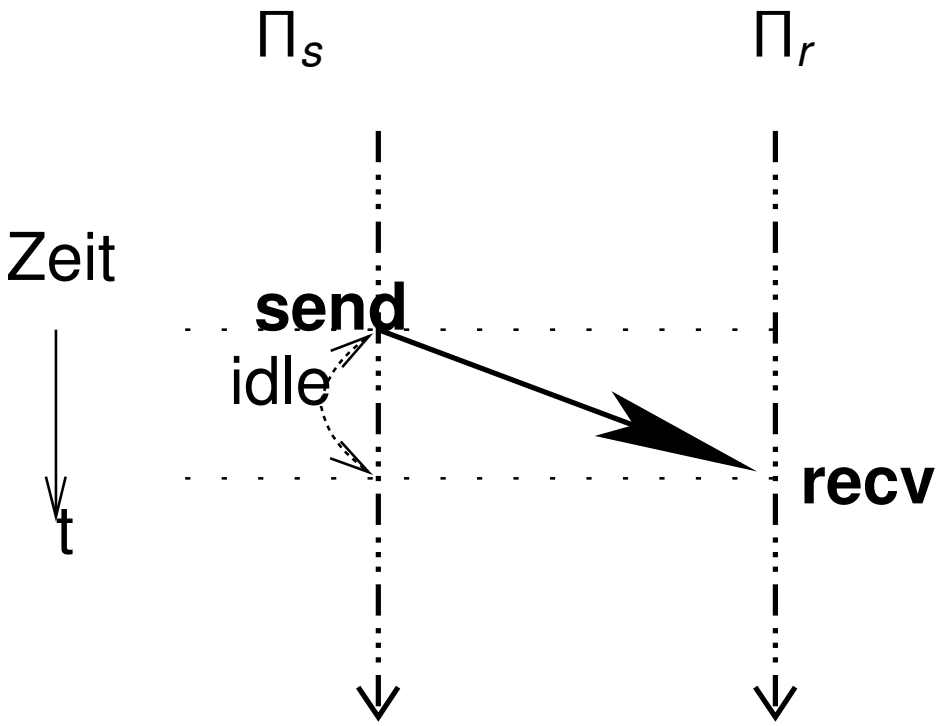
Communication via message passing

- Synchronous message passing
- Asynchronous message passing
- Global communication with
  - Store-and-Forward or
  - Cut-Through routing
- Global communication using different topologies
  - Ring
  - Array (2D / 3D)
  - Hypercube

Synchronous Message Passing I

- For the passing of messages we need at least two functions:
  - **send**: Transmits a memory area from the address space of the source process into the network with specification of the receiver.
  - **recv**: Receives a memory area from the network and writes it to the address space of the destination process.
- We distinguish:
  - Point in time at which the communication function is finished.
  - Point in time at which the communication has really taken place.
- At synchronous communication these points in time are identical,
  - **send** blocks until the receiver has accepted the message.
  - **recv** blocks until the message has arrived.
- Syntax in our programming language:
  - **send**($dest - process, expr_1, \ldots, expr_n$)
  - **recv**($src - process$, $var_1, \ldots, var_n$)

Synchronous Message Passing II



(a)

Π_1: **recv**(Π_3,i)
send(Π_2,i)

Π_2: **recv**(Π_1,i)
send(Π_3,i)

Π_3: **recv**(Π_2,i)
send(Π_1,i)

(b)

(a) Synchronisation of two processes by a pair of **send**/**recv** ops

(b) Example for a deadlock

Synchronous Message Passing III

There are a series of implementation possibilities.

- Senderinitiated, *three-way handshake*:
  - Source Q sends *ready-to-send* to target Z.
  - Target sends *ready-to-receive* if **recv** has been executed.
  - Source transmits message (variable length, single copy).
- Receiver initiated, *two-phase protocol*:
  - Z sends *ready-to-receive* to Q if **recv** has been executed.
  - Q transmits message (variable length, single copy).
- Buffered Send
  - Q transmits message at once, target has eventually to buffer it.
  - Here arises the problem of finite memory space!

Synchronous Message Passing IV

- Synchronous **send**/**recv** is not enough to solve all communication tasks!
- Example: In the producer-consumer-problem the bufferr is realized as an individual process. In this case the process cannot know with which of the producers or consumers it has to communicate next. In consequence a blocking **send** can result in a deadlock.
- Solution: Introduction of additional *guard functions*, that check whether a **send** or **recv** would result in a deadlock:
  - **int sprobe**(*dest* − *process*)
  - **int rprobe**(*src* − *process*).

  sprobe returns 1 if the receiver process is ready to receive, this means a **send** will not block:
  - **if** (**sprobe**(Π_d)) **send**(Π_d,. . .);

  Analogous for **rprobe**.
- Guard functions never block!

Synchronous Message Passing V

- Just one of both functions is needed.
  - **rprobe** is easy to integrate into the sender-initiated protocol.
  - **sprobe** is easy to integrate into the receiver-initiated protocol.
- An instruction with similar effect as **rprobe** is:
  - **recv_any**(who,var_1,...,var_n).

  It allows the receiving from an *arbitrary* process, which ID is stored in the variable *who*.
- **recv_any** is implemented simplest with sender-initiated protocol.

Asynchronous Message Passing I

- Instructions for asynchronous message passing:
  - **asend**($dest-process, expr_1, \ldots, expr_n$)
  - **arecv**($src-process, var_1, \ldots, var_n$)
- Here the return of the communication function does *not* indicate, that the communication has actually taken place. This has to be queried with additional functions.
- We imagine, that a request is passed to the system to execute the corresponding communication, as soon as it is possible. The calculating process can meanwhile do other things (*communication hiding*).
- Syntax:
  - **msgid asend**($dest-process, var_1, \ldots, var_n$)
  - **msgid arecv**($src-process, var_1, \ldots, var_n$)

  these do never block! **msgid** is a bill for the communication request.

Asynchronous Message Passing II

- Caution: The variable $var_1, \ldots, var_n$ may not be modified anymore when the communication instruction has been initiated!
- This means that the program has to manage the memory space for the communication variable by itself. Alternative would be the buffered send, which is connected with subtilities and need of double-copying.
- Finally one has to test whether the communication has already taken place (this means the request is processed):
  - **int** success(**msgid** *m*)
- Thereafter the communication variables may be modified, the bill is then invalidated.

Synchonous/Asynchronous Message Passing

- Synchronous and asynchronous operations may be mixed. This is specified in the MPI standard.
- Up-to-now all operations have been *without connection*.
- Alternatively there exist channel oriented communication operations (or virtual channels):

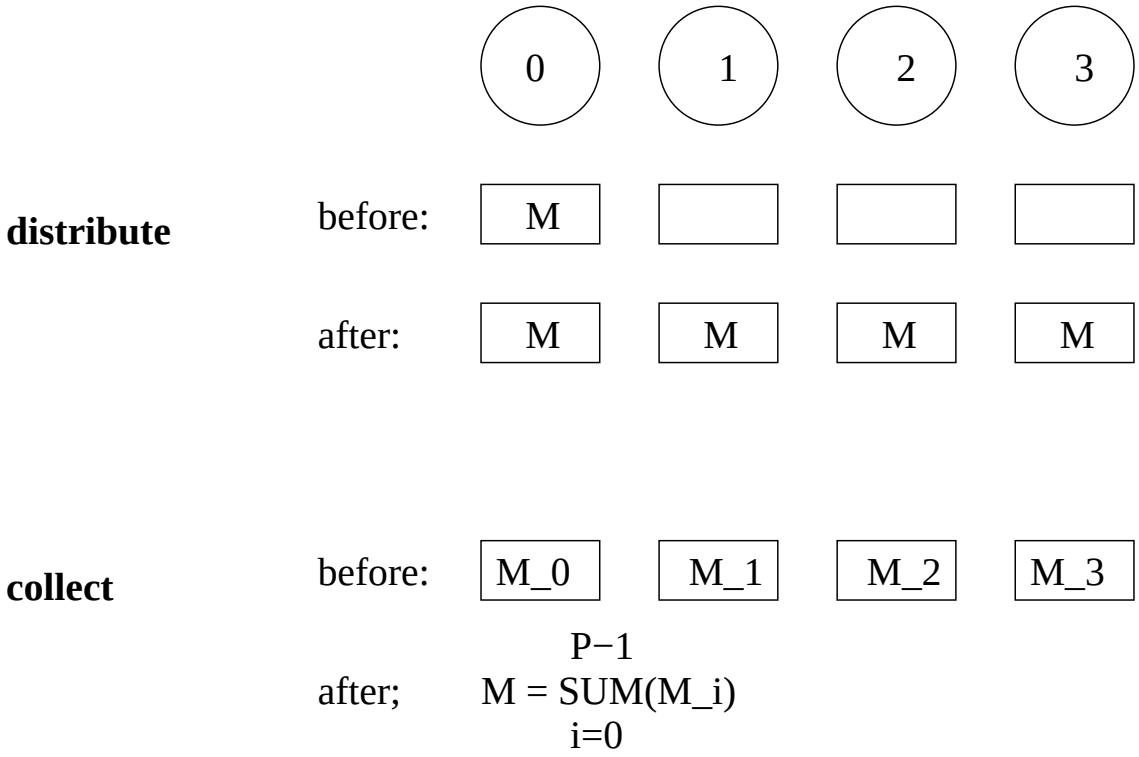
  - Before first send/receive to/from a process a connection has to be established by **connect**.
  - **send**/**recv** operations are assigned a channel instead of a process id as address.
  - Several processes can send on a channel but only one can receive.
    - **send**(*channel*,$expr_1$,...,$expr_n$)
    - **recv**(*channel*,var_1,...,var_n).
- We will use no channel-oriented functions.

Global Communication

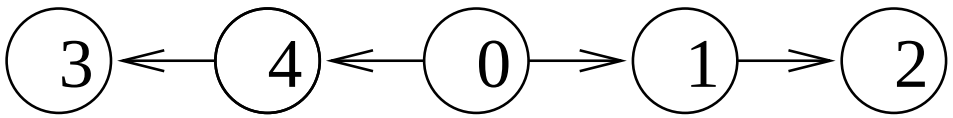
- A process wants to send *identical* data to all other processes
- *one-to-all broadcast*
- dual operation is the collection of individual results on a single process, e.g. sum generation (all associative operators are possible)
- We consider distribution across different topologies and calculate the time demand for store & forward as well as cut-through routing
- Algorithms for the collection result from reversing the sequence and direction of the communications
- The following cases are discussed in detail:
  - One-to-all
  - All-to-one
  - One-to-all with individual messages
  - All-to-all with individual messages

One-to-all: Ring

A process wants to send *identical* data to all other processes:



Here: Communication in ring topology with store & forward:



One-to-all: Ring

Program (One-to-all in the ring)

```
parallel one-to-all-ring
{
    const int P;
    process Π[int p ∈ {0, ..., P − 1}]{
        void one_to_all_broadcast(msg *mptr) {
            // receive messages
            if (p > 0  ∧  p ≤ P/2)        recv(Π_{p−1}, *mptr);
            if (p > P/2)                  recv(Π_{(p+1)%P}, *mptr);
            // pass messages to successor
            if (p ≤ P/2 − 1)              send(Π_{p+1}, *mptr);
            if (p > P/2 + 1 ∨ p == 0)     send(Π_{(p+P−1)%P}, *mptr);
        }
        ...;
        m=...;
        one_to_all_broadcast(&m);
    }
}
```

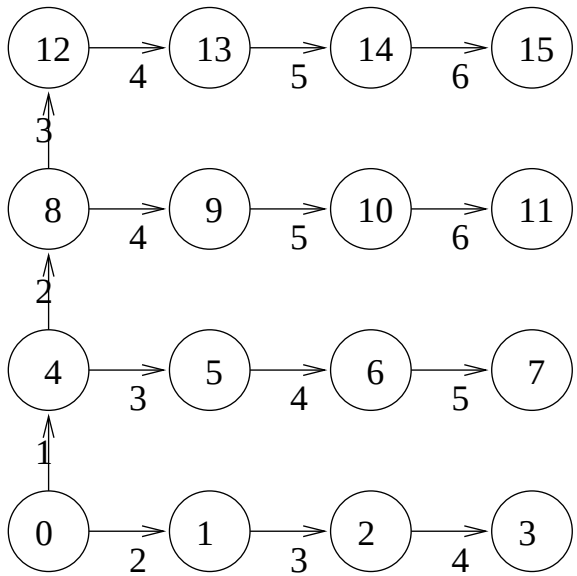
The time consumption for the operation is (nearest-neighbor communication!):

$$T_{one-to-all-ring} = (t_s + t_h + t_w \cdot n) \left\lceil \frac{P}{2} \right\rceil ,$$

where $n = |*mptr|$ denotes the length of the message.

One-to-all: Array

Now we assume a 2D array structure for communication. The messages are routed along the following pathes:



Look at the two-dimensional process index:

One-to-all: Array

Program (One to all on the array)

```
parallel one-to-all-array
{
    int P, Q;                                   // Array size in x and y direction
    process Π[int[2] (p, q) ∈ {0, ..., P − 1} × {0, ..., Q − 1}] {
        void one_to_all_broadcast(msg *mptr) {
            if (p == 0)
            {                                   // first column
                if (q > 0)                      recv(Π(p,q−1),*mptr);
                if (q < Q − 1)                  send(Π(p,q+1),*mptr);
            }
            else                        recv(Π(p−1,q),*mptr);
            if (p < P − 1)              send(Π(p+1,q),*mptr);
        }

        msg m=...;
        one_to_all_broadcast(&m);
    }
}
```

The execution time for $P = 0$ is in a 2d array

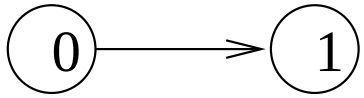
$$T_{one-to-all-array2D} = 2(t_s + t_h + t_w \cdot n)(\sqrt{P} - 1)$$

and in a 3d array

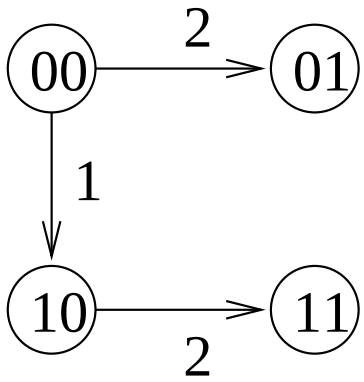
$$T_{one-to-all-array3D} = 3(t_s + t_h + t_w \cdot n)(P^{1/3} - 1).$$

One-to-all: Hypercube

- We advance in a recursive manner. For a hypercube of dimension $d = 1$ the problem can be solved in a trivial way:



- In a hypercube of dimension $d = 2$ node 0 sends first to node 2 and the problem is reduced to 2 hypercubes of dimension 1:



- In general for step $k = 0, \ldots d - 1$ the processes

$$\text{send each a message to } \underbrace{p_{d-1} \ldots p_{d-k}}_{k \text{ dimens.}} \; 1 \; \underbrace{0 \ldots 0}_{d-k-1 \text{ dimens.}}$$

$$\underbrace{p_{d-1} \ldots p_{d-k}}_{k \text{ dimens.}} \; 0 \; \underbrace{0 \ldots 0}_{d-k-1 \text{ dimens.}}$$

One-to-all: Hypercube

Program (One to all in the hypercube)

```
parallel one-to-all-hypercube
{
    int d, P = 2^d;
    process Π[int p ∈ {0, ..., P − 1}]{
        void one_to_all_broadcast(msg *mptr) {
            int i, mask = 2^d − 1;
            for (i = d − 1; i ≥ 0; i − −){
                mask = mask ⊕ 2^i;
                if (p&mask == 0) {
                    if (p&2^i == 0)                    //the last i bits are 0
                        send(Π_{p⊕2^i}, *mptr);
                    else
                        recv(Π_{p⊕2^i}, *mptr);
                }
            }
        }

        msg m = „bla“; one_to_all_broadcast(&m);
    }
}
```

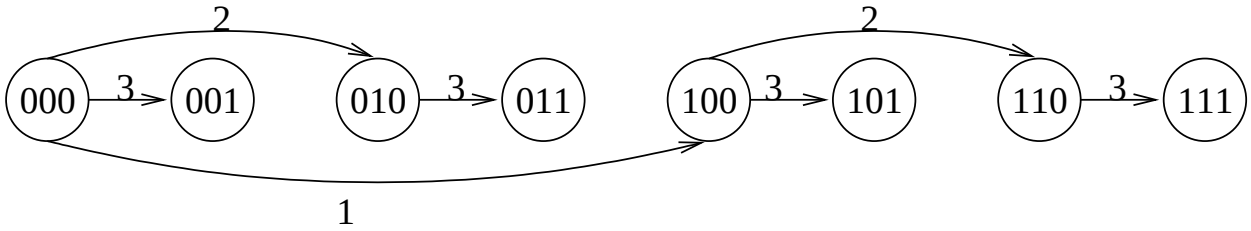
The time consumption is

$$T_{one-to-all-HC} = (t_s + t_h + t_w \cdot n) \operatorname{ld} P$$

Arbitrary source $src \in \{0, \ldots, P-1\}$: Substitute each p by $(p \oplus src)$.

One-to-all: Ring and array with cut-through routing

- If the hypercube algorithm is mapped on the ring, we obtain the following communication structure:

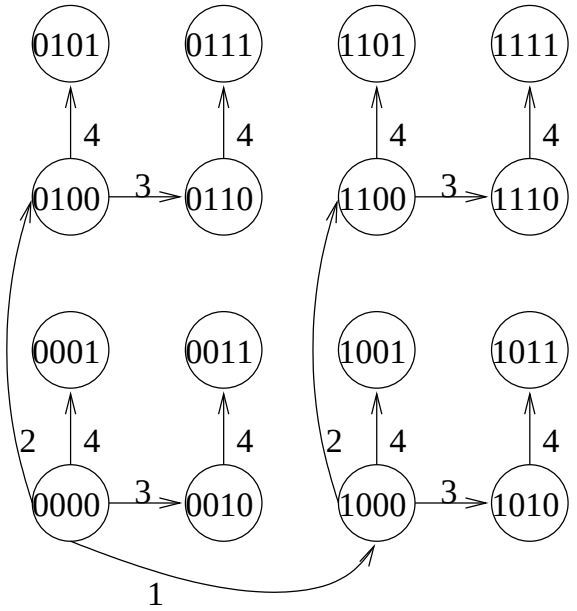


- There are no wires used twice, therefore we obtain by cut-through routing:

$$
\begin{aligned}
T_{one-to-all-ring-ct} &= \sum_{i=0}^{\mathrm{ld}\,P-1} (t_s + t_w \cdot n + t_h \cdot 2^i) \\
&= (t_s + t_w \cdot n)\,\mathrm{ld}\,P + t_h(P-1)
\end{aligned}
$$

One-to-all: Ring and Array with Cut-Through Routing

When using an array structure one obtains the following communication structure:



Again there are no wire collisions and we obtain:

$$T_{one-to-all-field-ct} = \underbrace{2}_{\substack{\text{jede} \\ \text{Entfernung 2} \\ \text{mal}}} \sum_{i=0}^{\frac{\text{ld } P}{2}-1} (t_s + t_w \cdot n + t_h \cdot 2^i)$$

One-to-all: Ring and Array with Cut-Through Routing

$$
\begin{aligned}
T_{one-to-all-field-ct} &= (t_s + t_w \cdot n) 2 \frac{\mathrm{ld}\, P}{2} + t_h \cdot 2 \underbrace{\sum_{i=0}^{\frac{\mathrm{ld}\, P}{2}-1} 2^i}_{=\underbrace{2^{\frac{\mathrm{ld}\, P}{2}}}_{=\sqrt{P}}-1} \\
&= \mathrm{ld}\, P(t_s + t_w \cdot n) + t_h \cdot 2(\sqrt{P} - 1)
\end{aligned}
$$

Especially with the array topology the term covering t_h is neglible and we obtain the hypercube performance also for less rich topologies! Also for $P = 1024 = 32 \times 32$ the time is not determined by t_h, thus because of cut-through routing no physical hypercube structures are necessary anymore.