The Message Passing Interface (MPI)

- MPI is a message passing library standard which can be used in conjunction with conventional programming languages such as C, C++ or Fortran.
- MPI is based on the point-to-point Send and Receive operations between a specified sending process and a specified receiving process.
- Collective communication functions such as MPI_Bcast (broadcast), MPI_Scatter (one-to-all personalized broadcast), MPI_Gather (all-to-one receiving), MPI_Allgather (all-to-all broadcast) or MPI_Alltoall (All-to-all personalized broadcast) are built from the communication primitives Send and Receive.
- Other collective communication functions such as MPI_Reduce, MPI_Allreduce or MPI_Scan also manipulate messages.

The Send Operation

The sending process requests permission from the receiver to send. Before or in the mean time it may copy its message from the send buffer into a system buffer.

- If the recipient has replied, the message is copied from its respective buffer into the communication buffer (such as the TCP/IP buffer) and the “bits flow into the cable” without any further processor intervention.
- As long as the recipient has not replied, the send buffer has to remain unchanged or the sending process may copy the send buffer into a system buffer.
- MPI provides versions of Send which either allow communication to continue as a background process or it may force the sender to wait in order to synchronize communication.

Blocking Send

A version of Send is blocking, if its completion depends on events such as successful message delivery or message buffering.

- Non-blocking communication helps to mask the communication overhead.
  - MPI_Isend posts the request to send immediately and the sender can resume work. (Non-blocking, send buffer should *not* be reused.)
- However unsafe buffer access has to be avoided and additional code has to check the status of the communication process running in the background.
  - MPI_Send copies short messages into a system buffer. For a long message the sender has to wait until the message is successfully delivered. (Blocking, send buffer can be reused.)
  - MPI_Ssend terminates only if the send buffer is emptied and the receiver has begun reception. At this time sender and receiver have synchronized. (Blocking, send buffer can be reused.)

A Communicator

- MPI supports a variety of collective communication functions, in which a group of processes cooperates to distribute or gather a set of values.
- The involved processes as well as their attributes form a communicator (or communication pattern):
  - One such attribute is the topology of the communicator (mesh topologies or general graph topologies).
  - Processes receive coordinates and can be addressed by these coordinates.

Collective Communication Functions

Assume that p processes participate.

- MPI_Bcast (one-to-all broadcast): a specified value is to be sent to all processes of the communicator.
- MPI_Scatter (one-to-all personalized broadcast): a root process sends messages $M_1, \ldots, M_p$ with process i receiving M_i.
- MPI_Gather is the counterpart of MPI_Scatter. : The ith process sends a message M_i to a specified root process.
- MPI_Allgather(all-to-all broadcast): each process i specifies a message M_i. After completion each process of the communicator has to know all messages $M_1, \ldots, M_p$.
- In MPI_Alltoall (all-to-all personalized broadcast): each process i specifies messages M_j^i that it wants to send to process j. After completion process j has to know all messages $M_j^1, \ldots, M_j^p$.

All-to-all Personalized Broadcast: An Example

- A $p \times p$ matrix A is given.
- Initially process i stores the ith row and is supposed to finally store the ith column.
  Thus we want to transpose A.
- All we have to do is to implement an all-to-all personalized broadcast in which process i sends $M_j^i = A[i,j]$ to process j.

MPI_Reduce, MPI_Allreduce and MPI_Scan

- In MPI_Reduce messages $M_1, \ldots, M_p$, an associative operation $*$ and a root process is given. The result $M_1 * \cdots * M_p$ has to be assigned to the root process.
  - One can choose for instance from the following list of operations: maximum, minimum, sum, product, and, or, xor, bitwise and, bitwise or, bitwise xor.
- MPI_Allreduce works as MPI_Reduce, but the result is distributed to all processes of the communicator.
- MPI_Scan is the prefix version of MPI_Reduce: process i has to receive the "sum" $M_1 * \cdots * M_i$.

Analyzing an MPI Program

The cost of communicating by far exceeds the cost of local computing.

> Which characteristics of a parallel machine are of interest when evaluating a parallel algorithm?

- Hopefully few parameters suffice to predict the performance on a large variety of different platforms.
  - Latency (the time from the start of a transmission to the end of the reception for a short message) and the processor count are certainly fundamental parameters.
  - The per-processor communication bandwidth is relevant as well as the time required to send long messages.
  - We also should worry about the overhead when sending a message.
- Measure all parameters as multiples of the processor cycle.

The LogGP Model

- L denotes the latency.
- o denotes the message overhead, namely the time spent for
  - supplying header information,
  - copying a message into the communication buffer and
  - performing the sender-receiver handshake.
- the gap parameter g is the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor for messages of standard length w. $\frac{1}{g}$ is the per-processor communication bandwidth.
- G is the time per byte gap for long messages. $\frac{1}{G}$ is the per-processor communication bandwidth for long messages.
- P is the number of processors.

The Cost of Communicating

The latency of a link is defined as the time from the start of a transmission to the end of the reception for a short message.

- Fast Ethernet or Gigabit Ethernet have latencies of 100 μs. The latest generations of Myrinet and InfiniBand have latencies of as low as 2μs and 1.32 μs respectively.
- Still a simple compute step is by a factor of a few thousands faster than a simple communication step.
- Bandwidth is considerable,
  - Fast Ethernet: 100 Mbit/sec,
  - Gigabit Ethernet: 1 Gbit/sec,
  - Myrinet: 1,92 Gbit/sec,
  - InfiniBand: up to 10 Gbit/sec,
- however long message streams are transported only with interruptions.
- The good news: latency and bandwidth continue to improve.

Typical Parameter Values

- The current Myrinet implementation of the CSC cluster has a bandwidth of 1,92 Gbit/sec and a latency of about 7μs.
- Gigabit Ethernet has a bandwith of 1 Gbit/sec and a latency of about 100 μs.
- The standard message length *w* is 16 KByte.
- The gap parameter:
  - for Myrinet $g = \frac{16KByte}{1.92Gbit} = \frac{128Kbit}{1.92Gbit} \approx 66 \cdot 10^{-6}$. Hence $g \approx 66\mu s$.
  - for Gigabit Ethernet $g = \frac{128Kbit}{1Gbit} \approx 128 \cdot 10^{-6}$. Hence $g \approx 128\mu s$.
- Experiments show $o \approx 70\mu s$ as an approximation for MPI_Ssend on the Myrinet. Gap and overhead almost coincide.

Message Delivery Time

- The time for delivering a short message is estimated as $o + L + o$: add overheads for sending and receiving as well as the latency.
  - The sending process is occupied only for time o.
    It is reasonable to differentiate overhead and latency.
  - The estimate assumes congestion-free routing.
- The time $T_{\text{Send}}(n)$ for delivering a (long) message of length n without support for long messages:
  - Break up the message into $\lceil n/w \rceil$ messages of length w.
  - Use the gap g for performing overhead tasks: we may inject new messages after $\max\{o, g\}$ steps.
  - $T_{\text{Send}}(n) = o + (\lceil \frac{n}{w} - 1 \rceil) \cdot \max\{o, g\} + L + o = O(n)$.
  - The sending process is occupied for $o + (\lceil \frac{n}{w} - 1 \rceil) \cdot o$ cycles.
- With support for long messages:
  $T_{\text{Send}}(n) = o + (n - 1) \cdot G + L + o = O(n)$:
  - The first byte goes after o steps “into the wire” and
  - subsequent bytes follow in intervals of length G.
  - The last byte exits the wire at time $o + (n - 1) \cdot G + L$.
  - The sending process is busy only at the very beginning.

Implementing MPI_Bcast

Process r broadcasts a message M of standard length w.

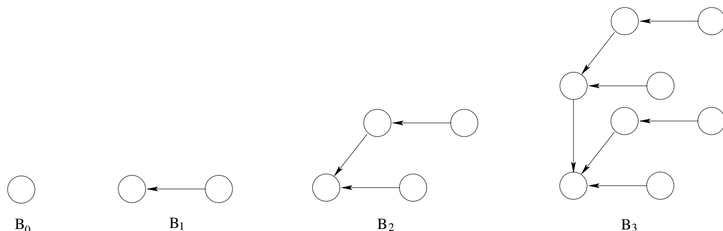
- r sends M to process s.
- r and s continue to broadcast M recursively:
  - r has to wait for $\max\{o, g\}$ cycles, whereas
  - s has to wait for $o + L + o$ cycles.
- If r and s continue sending M recursively to all p processes, then

$$T_{\text{Bcast}} \leq \lceil \log_2 p \rceil \cdot (o + L + o).$$

Here we assume $\max\{o, g\} \leq o + L + o$.

Binomial Trees

Which communication pattern is used, if *r* and *s* proceed recursively?



- The binomial tree B_{k+1}: Take two copies of B_k and make the root s of the second copy a child of the root r of the first copy.
- r may send its second message before s sends its first message:
  - Use a tree with a higher fanout for the root r.
  - The choice of the new fanout depends on L, o and g.
  - Determine the new tree via dynamic programming.

MPI_Scatter: One-To-All Personalized Broadcast

Process r sends a message M_i of standard length to process i.

- We assume support for long messages. Use binomial trees.
- r sends the concatenated message $M_{p/2+1} \cdots M_p$ to process s.
- Both processes continue recursively: subsequent processes break up the concatenation and propagate subsequences.
- Communication time $T_{\text{Scatter}}(n)$, if all messages have length n:
  - $o + (\frac{p}{2} \cdot n - 1) \cdot G + L + o$ cycles in the first round.
  - With an inductive argument:

$$
\begin{aligned}
T_{\text{Scatter}}(n) &\leq \sum_{k=1}^{\lceil \log_2 p \rceil} [o + (\frac{p}{2^k} \cdot n - 1) \cdot G + L + o] \\
&\leq \lceil \log_2 p \rceil \cdot (o + L + o) + p \cdot n \cdot G = O(p \cdot n)
\end{aligned}
$$

- Again, a higher fanout for r helps.
- MPI_Gather is implemented analogously.

MPI_Allgather: A Linear Array Implementation

Each process i sends its message M_i (of standard length w) to all other processes.

- The linear array implementation: Pump all messages through the network via pipelining:
  - process i sends M_i to process $i+1$.
  - process i receives message M_{i-1} after $o+L+o$ cycles.
  - It may forward M_{i-1} to process $i+1$ immediately afterwards.
  - for messages of standard length

$$T_{\text{Allgather},1} \leq (o+L+o) \cdot (p-1),$$

provided $g \leq o+L+o$.

- Assume support for long messages. What happens, if we combine individual messages?

MPI_Allgather: A Hypercube Implementation

- Apply recursive doubling for the hypercube of dimension $\log_2 p$:
  - Process $b = b_1 b_2 b'$ sends its message M_b to neighbor $\overline{b_1} b_2 b'$, receives message $M_{\overline{b_1} b_2 b'}$ in return and computes the concatenation $M_{0 u_2 u'} \circ M_{1 u_2 u'}$.
  - Repeat procedure for neighbor $b_1 \overline{b_2} b'$ and afterwards b has $M_{00b'} \circ M_{10b'} \circ M_{01b'} \circ M_{11b'}$.
  - if all messages have length n,

$$
\begin{aligned}
T_{\text{Allgather},2} &\leq \sum_{k=1}^{\lceil \log_2 p \rceil} [o + (\frac{p}{2^k} \cdot n - 1) \cdot G + L + o] \\
&\leq \lceil \log_2 p \rceil \cdot (o + L + o) + p \cdot n \cdot G = O(p \cdot n),
\end{aligned}
$$

- In comparison with the linear array: $(o + L + o)$ has weight $\lceil \log_2 p \rceil$ instead of $p - 1$.

Broadcasting a Long Message

- To broadcast a short message MPI uses variants of binomial trees.
- To broadcast a long message M, assuming support for long messages,
  - MPI first uses Scatter to break up M into shorter pieces and then
  - applies Allgather to put the pieces back together.
  - Why?

MPI_Alltoall

Each process i sends messages M_j^i to process j.

- Use the $\log_2 p$-dimensional hypercube as communication pattern.
- There is a total of $p - 1$ phases.
  - In phase $b \in \{0,1\}^{\log_2 p}$ with $b \neq 0$, process u sends its message $M_{u \oplus b}^u$ to process $u \oplus b$.
  - There are edge-disjoint paths $u \rightarrow u \oplus b$ in the d-dimensional hypercube for each $b \in \{0,1\}^d$. Congestion-free routing on the hypercube is possible.
- If all messages have length w,

$$T_{\text{Alltoall}} = (o + L + o) \cdot (p - 1) = T_{\text{Allgather},1}.$$

MPI_Reduce, MPI_Allreduce and MPI_Scan

- MPI_Reduce computes a “sum” and assigns it to a distinguished process: use a binomial tree.
- MPI_Allreduce assigns the sum to all processes: run MPI_Reduce and finish up with MPI_Bcast.
- MPI_Scan computes the prefix “sum” and assigns it to a distinguished process: implement the prefix algorithm on binomial trees.
- Performance of MPI_Allreduce and MPI_Scan roughly double the broadcast time.

Comparing Parallel and Sequential Algorithms

Assume that a parallel algorithm $\mathcal{P}$ solves an algorithmic problem $\mathcal{A}$. When should we be satisfied with its performance?

- Assume that $\mathcal{P}$ uses p processors and runs in time $t_{\mathcal{P}}(n)$ for inputs of length n.
  - We can simulate $\mathcal{P}$ sequentially in time $O(p)$ per step of $\mathcal{P}$.
  - The straightforward sequential simulation runs in time $O(p \cdot t_{\mathcal{P}}(n))$, provided the sequential computer has sufficient main memory.

$\text{work}_{\mathcal{P}}(n) = p \cdot t_{\mathcal{P}}(n)$ is the work of $\mathcal{P}$ on inputs of size n.

- $\text{work}_{\mathcal{P}}(n)$ should not be much larger than the running time of a good sequential algorithm.
- Our goal is to find a good parallelization of a good sequential algorithm for $\mathcal{A}$.

Speedup and Efficiency

- Assume that $\mathcal{S}$ is a sequential algorithm for $\mathcal{A}$.
- Let $\mathcal{P}$ be a parallelization of $\mathcal{S}$.

- $S_{\mathcal{P}}(n) = \frac{t_{\mathcal{S}}(n)}{t_{\mathcal{P}}(n)}$ is the **speedup** of $\mathcal{P}$: the speedup is asymptotically bounded by p.
- $E_{\mathcal{P}}(n) = \frac{t_{\mathcal{S}}(n)}{\text{work}_{\mathcal{P}}(n)} = \frac{S_{\mathcal{P}}(n)}{p}$ is the **efficiency** of $\mathcal{P}$: the efficiency is asymptotically at most one.

Scaling Down

A parallel algorithm $\mathcal{P}$ uses p processors. Can we come up with an equivalent parallel algorithm $\mathcal{Q}$ for q ($q < p$) processors, which is as efficient as $\mathcal{P}$?

- The scheduling problem:
  - Assume that $\mathcal{P}$ performs op_i operations in step i.
  - Assign these op_i operations in real time to $q < p$ processors.
- If the scheduling problem is solvable in real time, then step i of $\mathcal{P}$ can be simulated by $\lceil \frac{\text{op}_i(n)}{q} \rceil$ steps of $\mathcal{Q}$ and

$$t_{\mathcal{Q}}(n) = \sum_{i=1}^{t_{\mathcal{P}}(n)} \left\lceil \frac{\text{op}_i(n)}{q} \right\rceil \leq \sum_{i=1}^{t_{\mathcal{P}}(n)} \left(\frac{\text{op}_i(n)}{q} + 1 \right) \leq \frac{\text{work}_{\mathcal{P}}(n)}{q} + t_{\mathcal{P}}(n).$$

- Efficiency is almost the same, since

$$\frac{\text{work}_{\mathcal{P}}(n)}{\text{work}_{\mathcal{Q}}(n)} = \frac{\text{work}_{\mathcal{P}}(n)}{\text{work}_{\mathcal{P}}(n) + q \cdot t_{\mathcal{P}}(n)} = \frac{1}{1 + \frac{q \cdot t_{\mathcal{P}}(n)}{\text{work}_{\mathcal{P}}(n)}} = \frac{1}{1 + q/p}.$$

Rules of Thumb

- If we keep input size fixed:
  - We have just observed, that efficiency „tends to increase“, if we reduce the number of processors.
  - Because of that, efficiency “tends to decrease”, if we increase the number of processors.
- What happens, if we increase input size from n to $N > n$, but keep the number of processors fixed?
  - The sequential running time $t_{\mathcal{S}}(n)$ “tends to grow faster” than the parallel running time.
  - Hence efficiency “tends to grow” when increasing input size

$$\frac{E_{\mathcal{P}}(N)}{E_{\mathcal{P}}(n)} = \frac{t_{\mathcal{S}}(N)}{p \cdot t_{\mathcal{P}}(N)} \Big/ \frac{t_{\mathcal{S}}(n)}{p \cdot t_{\mathcal{P}}(n)} = \frac{t_{\mathcal{S}}(N)}{t_{\mathcal{S}}(n)} \Big/ \frac{t_{\mathcal{P}}(N)}{t_{\mathcal{P}}(n)}.$$

Isoefficiency

A good parallel algorithm $\mathcal{P}$ should reach large efficiency for small input sizes.

- The isoefficiency function f_E with respect to E is the smallest input size $f_E(p)$ with $E_{\mathcal{P}}(n) \geq E$ whenever $n \geq f_E(p)$.
- The slower f_E grows the better.
- The prefix problem:
  - Our solution $\mathcal{P}$ runs in time $t_{\mathcal{P}}(n) = O(\frac{n}{p} + \log_2 p)$ for p processors.
  - Hence $\text{work}_{\mathcal{P}}(n) = O(p \cdot (\frac{n}{p} + \log_2 p)) = O(n + p \log_2 p)$ and
  - $E_{\mathcal{P}}(n) = O(\frac{n}{n+p \log_2 p})$.
  - $f_E(p) = \Omega(p \cdot \log_2 p)$ is the isoefficiency for $E = \Theta(1)$.
- The odd-even transposition sort runs in time $\Theta(\frac{n}{p} \log_2 \frac{n}{p} + n)$.
  - Hence $\text{work}_{\mathcal{P}}(n) = O(p \cdot (\frac{n}{p} \log_2 \frac{n}{p} + n)) = O(n \log_2 \frac{n}{p} + p \cdot n)$.
  - $E_{\mathcal{P}}(n) = \frac{n \cdot \log_2 n}{n \log_2 \frac{n}{p} + p \cdot n}$ and efficiency is constant iff $p = O(\log_2 n)$.
  - For $E = \Theta(1)$, we obtain $f_E(p) = 2^{\Theta(p)}$ as isoefficiency function.

More Rules of Thumb

- Design a parallel algorithm with large efficiency, but slow growing isoefficiency.
- Breaking up the algorithmic problem:
  - Partition the algorithmic problem into as many primitive tasks as possible.
  - Locality Preserving Mapping: Assign tasks to processors such that communication is minimized.
- Try to “hide” communication with local computation whenever possible: keep the processor busy even when communicating.
  - Computation should dominate over communication.