Message-Coupled Systems

October 4, 2005

Ralf-Peter Mundani
Department of Computer Science – Chair V
Technische Universität München, Germany
5.1. The Message Passing Paradigm

- very general principle, applicable to nearly all types of parallel architectures (message-coupled and memory-coupled)
- standard programming paradigm or model for message-coupled systems:
  - message-coupled multiprocessors
  - clusters of PCs or work stations (homogeneous, dedicated use, high-speed network)
  - networks of PCs or work stations (heterogeneous, non-dedicated use, standard (fast) ethernet network)
- several concrete programming environments:
  - machine-dependent: MPL (IBM), CS (Meiko), PSE (nCUBE), ...
  - machine-independent: EXPRESS, P4, PARMACS, PVM, ...
- machine-independent standards:
  - originally PVM, but today dominated by
  - MPI, with efficient machine-dependent implementations
The Underlying Principle

- A parallel program consists of \( p \) processes with different address spaces.
- Communication takes place via the explicit exchange of data or messages (realized via system calls like `send(...)` or `receive(...)` and others), only.
- Hence, communication is realized with the help of communication libraries:
  - to be designed without dependencies of the hardware or of the programming language of the sequential source
  - to be made available for standard imperative programming languages like C, C++, or Fortran
  - to be made available for multiprocessors and standard monoprocessors (for clusters!)
  - to be linked to the source code during compilation
  - advantage: take existing code as the starting point for the parallelization and add suitable procedure calls
- message consists of
  - header: target ID, message information (type, length, ...)
  - body: the data to be provided
- need for a buffer mechanism: what to do if the receiver is not (yet) ready to receive?
The User’s View

- library functions as the only interface to the communication system!
Message Buffers

• typically (but not necessarily) connected parts of memory
  – homogeneous architectures (all processors of the same type): buffer as a sequence of bytes, without any type information
  – heterogeneous architectures (different types of processors): type information necessary for format conversion by message passing library

• definition and allocation of message buffers:
  – send buffer: generally done by application program
  – receive buffer: either automatically by message passing library or manually by application program (eventually with check whether buffer length is sufficient)
Elementary Communication Operations (point-to-point)

- **send** – required information:
  - receiver *(who shall get the message?)*
  - send buffer *(where is the data provided?)*
  - type of message *(what kind of information is sent?)*
  - communication context *(context within which the message may be sent and received – see next slide)*

- a send command can be
  - **blocking**: continuation possible after passing to communication system has been completed (buffer can be re-used)
  - **non-blocking**: immediate continuation possible (check buffer whether message has been sent and buffer can be re-used)

- **receive** – required information:
  - sender *(wild cards are possible, i.e. receive from any process)*
  - receive buffer *(where is the incoming message to be put?)*
  - type of message *(wild cards are possible, again)*
  - communication context *(no wild cards possible, always one unique context)*

- a receive command can be
  - **blocking**: continuation only after a suitable message has been received
  - **non-blocking**: immediate continuation – either with a successful receive result or with a failure and further checks whether a suitable message has arrived
Communication Context

- scenario:
  - three processes, and all of them call a subroutine B from a library
  - inter-process communication within the subroutines
  - communication context shall ensure this restriction to the subroutines
  - compare correct order (below) and error case (next slide)
Communication Context (cont’d)

- **scenario:**
  - three processes, and all of them call a subroutine B from a library
  - inter-process communication within the subroutines
  - communication context shall ensure this restriction to the subroutines
  - compare correct order (previous slide) and error case (below)
Why Buffers?

• look at the following program:

P1:  compute something
     store result in SBUF
     SendBlocking(P2,SBUF)
     ReceiveBlocking(P2,RBUF)
     read data in RBUF
     process RBUF

P2:  compute something
     store result in SBUF
     SendBlocking(P1,SBUF)
     ReceiveBlocking(P1,RBUF)
     read data in RBUF
     process RBUF

• does this work?
  – YES, if the communication system buffers internally
  – NO, if the communication system does not use buffers (deadlock!)
    hence: avoid this with non-blocking send operations or with an atomic sendreceive operation

• typical buffering options:
  – nothing specified: buffering possible, but not mandatory (standard; users must not rely on buffering)
  – guaranteed buffering: problems if there is not enough memory
  – no buffering: efficient, if buffering is not necessary (due to the algorithm, for example)
Keeping the Order

- problem: there is no global time in a distributed system
- consequence: there may be wrong send-receive assignments due to a changed order of occurrence
  - typically no problem for only one channel P1 ↔ P2
  - may be a problem if more processes communicate and if sender is specified via a wild card:

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>send m1 to P3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>receive buf1 from any task</td>
<td>receive buf2 from any task</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>send m1 to P3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>receive buf1 from any task</td>
<td>receive buf2 from any task</td>
</tr>
</tbody>
</table>
```

OR

```
<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>send m1 to P3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>receive buf2 from any task</td>
</tr>
</tbody>
</table>

```
Collective Communication

- Many applications require not only a point-to-point communication, but also collective communication operations.
- **broadcast:**
  - meaning: send the message to all participating processes
  - example: the first process that finds the solution in a competition informs everyone to stop

![Diagram showing collective communication]

- **multicast:**
  - meaning: send the message to a subset of participating processes
  - example: Gauß-Seidel iteration, one processor per node: the update is communicated to the neighbours
Collective Communication (cont’d)

• gather:
  – meaning: collect information from all participating processes
  – example: each process computes some part of the solution, which shall now be assembled by one process
Collective Communication (cont’d)

• **gather-to-all:**
  - **meaning:** like gather, but all participating processes assemble the collected information
  - **example:** as before, but now all processes need the complete solution for their continuation

![Diagram showing gather-to-all communication in Message-Coupled Systems](image-url)
Collective Communication (cont’d)

• **scatter:**
  
  – meaning: distribute your data among the processes
  – example: two vectors are distributed in order to prepare a parallel computation of their scalar product

```
P_0  |  P_0

P_1  |  P_1

P_2  |  P_2

P_3  |  P_3

send buffers

receive buffers

Scatter
```

\[ P_0 \quad | \quad \quad | \quad P_0 \]
\[ P_1 \quad | \quad \quad | \quad P_1 \]
\[ P_2 \quad | \quad \quad | \quad P_2 \]
\[ P_3 \quad | \quad \quad | \quad P_3 \]

The Message Passing...
Collective Communication (cont’d)

• all-to-all gather/scatter:
  – meaning: data of all processes are distributed among all processes
Collective Communication (cont’d)

- **reduce:**
  - meaning: information of all processes is used to provide a condensed result by/for one process
  - example: calculation of the global minimum of the variables kept by all processes, calculation of a global sum, etc.
Collective Communication (cont’d)

• all-reduce:
  – meaning: like reduce, but condensed result is available for all processes
  – example: suppose the result is needed for the control of each process’ continuation

```
\begin{figure}
\centering
\includegraphics[width=\textwidth]{all_reduce_diagram}
\caption{Diagram illustrating the all-reduce communication pattern.}
\end{figure}
```
Message Types – Two Main Classes

- **data messages:**
  - **meaning:** data are exchanged in order to provide other processes’ input for further computations
  - **example:** interface values in a domain-decomposition parallelization of a PDE solution

- **control messages:**
  - **meaning:** data are exchanged in order to control the other processes’ continuation
  - **examples:** a global error estimator indicates whether a finite element mesh should be refined or not; a flag determines what to do next

```c
... receive(anynstask,anynotype,RBUF)
snd = \langle sender of the received message \rangle
type = \langle type of the received message \rangle
if (type==DATA) then process data from snd
elsef (type==ERROR_MESSAGE) then process error report from snd
else ...
...
```

- in both cases, additional information concerning the format is necessary in general (to be provided with the message type)
Efficiency

• avoid short messages: latency reduces the effective bandwidth

\[ t_{\text{total}} = t_{\text{setup}} + \frac{n}{B} \quad \text{(length } n, \text{ bandwidth } B) \]

\[ B_{\text{eff}} = \frac{n}{t_{\text{total}}} \]

• computation should dominate communication!

• typical conflict for numerical simulations:
  – overall runtime suggests large numbers \( p \) of processes
  – communication-computation ratio and message size suggest small \( p \)

• try to find (machine- and problem-dependent) optimum number of processes
• try to avoid communication points at all, prefer redundant computations (if inevitable)
Starting and Ending a Parallel Application

• here are often the biggest differences between the various communication libraries
• the common thing: some kind of a prologue and of an epilogue
• some possibilities how to start:
  – allocation of participating processors with a command;
    on all processors, the same executable program is loaded;
    ⇒ standard proceeding for MPP
  – definition of available processors in a file;
    start a master process;
    master process starts the other processes dynamically;
    ⇒ the way PVM does it
  – specification of processors and processes in a file;
    ⇒ in case of a static process model
• some possibilities how to stop:
  – automatic stop when all processes have stopped
  – explicit halt of the parallel application