Tutorial

Sorting Out Communication Libraries: A Comparison of NX, CMMD, PVM and MPI

William Saphir

NAS Scientific Computing Branch
NASA Ames Research Center
Computer Sciences Corporation
M/S 258-6 Moffett Field, CA 94035
415-604-4427 Fax: 415-604-4377
wcs@nas.nasa.gov

Topics

1. **Categorization of applications and hardware**

2. **Overview of libraries**

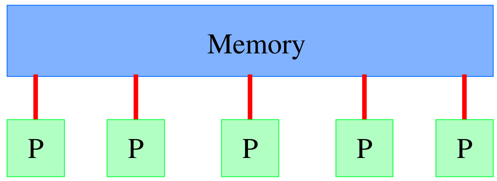
3. **Detailed analysis and comparison by functionality**

    - **Point-To-Point Communication**
        - **Theory (semantics)**
        - **Practice (optimization, performance)**
    - Collective Communication, Groups and Topologies
    - Contexts
    - Process Management
    - I/O

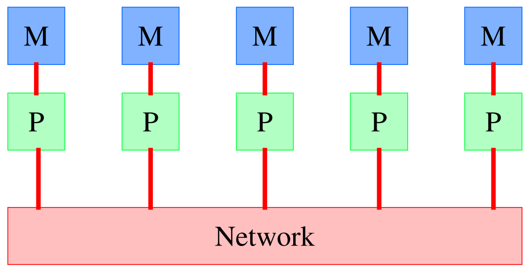
4. **Porting and portability**

Distributed Memory Parallel Computers

Shared memory multiprocessor:



Distributed memory multiprocessor:



Distributed Memory Parallel Programming

Two broad classes of parallel programming

- **Single thread of control (data parallel, e.g. HPF)**
  - Parallelism is implicit
  - Compiler handles communication and data distribution
  - User supplies directives to assist compiler
- **Multiple independent threads of control**
  - Parallelism is explicit
  - User manages communication
  - User manages data layout

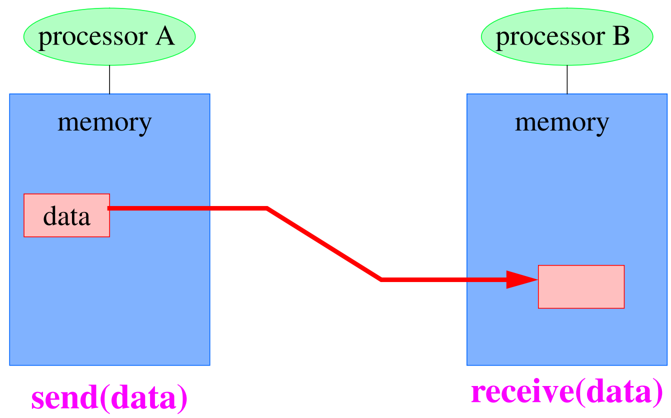
Explicit data transfer

- **Message passing**: cooperative send/receive
- Virtual shared memory
- Remote get/put
- Active messages

Message Passing Basics

- Distributed memory tasks have access only to local data
- Separate program runs on each node

Send data from task A to task B:



Categorization of Applications

The world is easier to understand if we make sweeping categorizations

Type A

- Fine-grained
- Communication bound
- Tightly synchronized
- Rigid (fixed number of processors) and often SPMD

Examples: NAS Parallel Benchmarks, linear algebra

Type B

- Coarse-grained
- Little communication
- Loosely synchronized
- Flexible and often MPMD

Examples: Grid-by-grid parallelism, Monte-Carlo

Categorization of Machines

Type A **(MPPs - "Moderately Parallel Processors")**

- Fast network (high bandwidth, low latency)
- Synchronized scheduling
- Fixed size, dedicated use

Example: Paragon, CM-5, T3D, SP2 (sometimes)

Type B **("Clusters")**

- Slow network (often through IP)
- Standard Unix scheduling
- Dynamic size, shared use

Example: Ethernet-connected network of workstations, SP2 (sometimes)

Application-Machine Compatibility

Observations in a type A/type B universe

- Type A applications are well-suited for type-A computers
- Type B applications are well-suited for type-B computers
- Type A applications run poorly (bad speedup) on type B computers
- Type B applications waste resources when run on type-A computers

Type A applications

- Optimization of communication very important
- *Generalized data exchange* is the right communication to study

Type B applications

- Message passing performance not as important
- Many ways to make use of flexibility

Overview of communication libraries

NX
- Runs on Intel supercomputers (iPSC/860, Delta, Paragon)
- Proprietary

CMMD
- Runs on the Thinking Machines CM-5 (R.I.P.)
- Proprietary

PVM (*Parallel Virtual Machine*)
- Portable: NOWs, MPPs, SMPs
- Public and proprietary implementations available

MPI (*Message Passing Interface*)
- Portable: NOWs, MPPs, SMPs
- Public and proprietary implementations available

More on PVM

PVM is an ongoing research project based at ORNL and UTK

History
- Development of Version 1.0 began March1989
- Version 2.0 written Spring, 1991: rewrite
- Version 2.4 released late1992: vsnd(), vrcv(), other improvements
- Version 3.0 released Spring, 1993: complete redesign
- Version 3.3 released May, 1994: psend/recv, reduce, InPlace, etc.

Current status
- Public domain available from ORNL/UT
- Proprietary versions available from
  - IBM (SP2), Cray (T3D), Convex, others

PVM Strengths/Weaknesses

Strengths

- Machine/task configuration/control
- Support for heterogeneous computing
- Wide use

Weaknesses

- Performance
- Communication functionality
- Stability/standardization
- Support for I/O

More on MPI

History

- MPI Forum: government, industry and academia.
- Formal process began November 1992
- Draft presented at Supercomputing 1993
- Final standard published May 1994

Current status

- Public domain versions available from ANL/MSU (MPICH), OSC (LAM), Edinburgh (CHIMP), MSU (Unify)
- Proprietary versions coming from IBM (SP2), Meiko (CS-2), Fujitsu (AP-1000)
- Interest expressed by CRI (T3D), Convex, Intel SSD

MPI Strengths/Weaknesses

Strengths

- Addresses library development issues
- Has broad scope
- Doesn't impede high performance
- Is a standard
- Becoming widely used

Weaknesses

- CISC-like approach to message passing
- Support for dynamic processes/machines
- Support for I/O

Other Libraries

There are many other message passing libraries, including

APPL	Developed at NASA Lewis
Chameleon	Lightweight portable library from ANL
CHIMP	Developed in England
Express	Proprietary (ParaSoft), portable
LAM	A version of MPI is written on top of this
MPL (EUI-H)	Native message passing library on the IBM SP2 (SP1)
P4	Developed at ANL, successor to PARMACS
PARMACS	Developed at ANL, used widely in Europe
PICL	Emphasis on tracing
TCGMSG	Modelled on PARMACS
Zipcode	Developed at Livermore

Conclusions

	Point-to-Point	Collective	Groups	Context	Process Management	I/O	Performance
CMMD	yes*	yes*	no*	no*	yes*	yes*	yes*
NX	yes*	yes*	no*	no*	yes*	yes*	yes*
PVM	yes*	yes*	yes*	no*	yes*	no*	no*
MPI	yes*	yes*	yes*	yes*	no*	no*	yes*

* **There's more to this than yes/no answers**

Point-To-Point Communication

Why go into great detail on the semantics of point-to-point routines?

- Point-to-point is the core of a communication library
- Documentation is terrible
- Detailed understanding necessary for optimization
- MPI point-to-point is complex, subtle

Note: Some applications use no point-to-point communication! (FFTs)

Semantics vs. Implementation

Distinguish between *semantics* and *implementation* of a routine.

Semantics

What you have to know about a routine in order to use it correctly.

Implementation

Low-level details of how a library routine is constructed in order to implement a certain semantics.

Ideal world: only semantics important
Real world: implementation may be important for performance

(sorry, English majors)

Point-to-Point Communication

All 4 libraries have syntax equivalent to:

```
send(message_buffer, buffer_size, label, destination)

receive(message_buffer, buffer_size, label, source)
```

Differences in semantics:

- Is a **`send`** guaranteed to return whether or not there is a **`receive`**?
- Will a **`send`** return *only* if there is a matching **`receive`**?
- Can buffer be modified immediately after message is sent?

Differences in implementation:

- When are messages buffered?
- Is data transferred by the network independently of the processor?

Terminology

There is no agreement on terminology.

Example: **`pvm_send()`** (PVM) and **`csend()`** (NX) have almost exactly the same semantics but the documentation says differently.

`pvm_send()`

"**The pvm_send routine is asynchronous.** Computation on the sending processor resumes as soon as the message is safely on its way to the receiving processor. ***This is in contrast to synchronous communication, during which computation on the sending processor halts until the matching receive is executed by the receiving processor."***

`csend()`

"**This is a synchronous system call.** The calling process waits (blocks) until the send completes. ***Completion of the send does not mean that the message was received, only that the message was sent and the send buffer can be reused."***

Terminology: My Definitions

Blocking

A routine is *blocking* if its completion (return of control to the calling routine) *may* depend on an external event (an event that is outside the control of the routine itself.) Example: a send is blocking if it does not return until there is a matching receive.

Asynchronous

A routine is *asynchronous* if it initiates an operation that happens logically outside the flow of control of the calling process. The important practical distinction is whether the program may be required to check for completion of the operation before proceeding. "Asynchronous" therefore refers to semantics, not implementation.

Non-blocking

A routine is *non-blocking* if it is guaranteed to complete regardless of external events. Example: a send is non-blocking if it is guaranteed to return whether or not there is a matching receive.

Synchronous

A routine is *synchronous* if its operation happens within the flow of control of the calling process. Opposite of asynchronous.

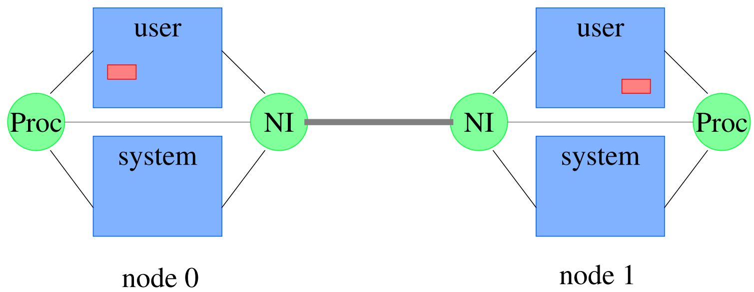
Synchronizing

A routine is synchronizing if it causes two separate processes to become synchronized.

Point-to-Point Communication - picture

To discuss semantics and implementation

- Separate network from processor
- Separate user memory from system memory



CMMD: point-to-point overview

CMMD is

- **Straightforward**
- **Clearly documented**
- **Well-designed**

CMMD provides 3 routines for sending data:

- `CMMD_send_block`
- `CMMD_send_noblock`
- `CMMD_send_async`

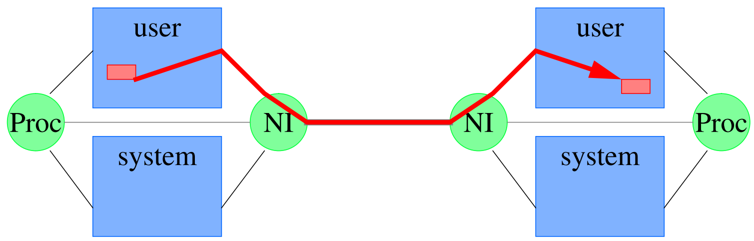
CMMD provides 2 routines for receiving data:

- `CMMD_receive_block`
- `CMMD_receive_async`

CMMD: `CMMD_send_block()`

Semantics: blocking, synchronous, synchronizing

- Waits for receive to be posted
- Returns after message has been received
- Buffer may be reused on completion



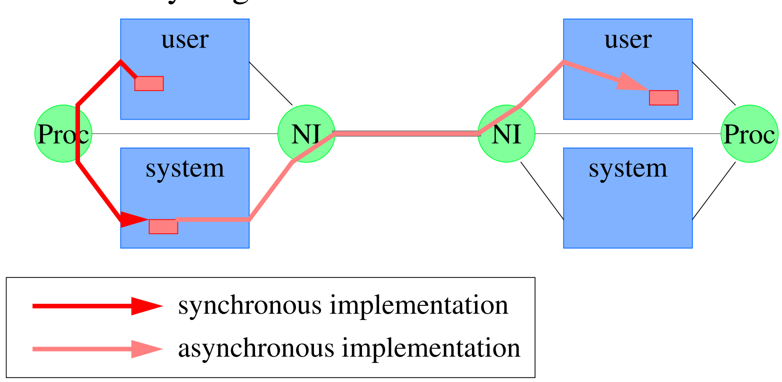
Implementation

- Sends a request-to-send (RTS) to destination node
- Waits for reply
- Sends message

CMMD: `CMMD_send_noblock()`

Semantics: non-blocking, synchronous, not synchronizing

- Returns whether or not a receive has been posted (does not block)
- Safe to modify original data after return



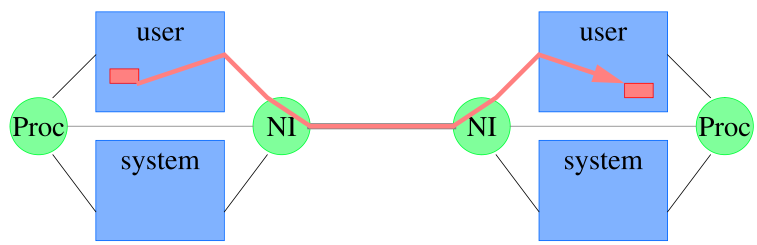
Implementation

- Sends data immediately if receive has been posted
- Otherwise, copies data into local system buffer
- Tells network to deliver asynchronously and returns

CMMD: `CMMD_send_async()`

Semantics: non-blocking, asynchronous

- Returns whether or not a receive has been posted (does not block)
- Not safe to modify original data after return (asynchronous)
- Must use **`CMMD_msg_wait()`** before modifying data
-



Implementation

- Tells network to send data when a receive is posted
- Data is not sent until receive is posted

CMMD Receive

1. **`CMMD_receive_block()`**

   - Synchronous, blocking
   - Does not return until message is safely in user buffer

2. **`CMMD_receive_async()`**

   - Asynchronous, non-blocking
   - Tells system where to place incoming message and returns
   - Message is delivered into buffer while program is executing (asynchronously)
   - Need to call **`CMMD_msg_wait()`** before using buffer

CMMD: optimization

One of the most common data communication patterns in a type A program is **generalized data exchange**:

- Each node executes the same code
- Each node sends data and receives data “at the same time”
- Each node knows where and when to expect data to arrive

Two rules of optimization for CMMD:

- Avoid buffering
- Try to overlap computation with communication

CMMD: how to exchange data (1)

Examine four different methods to exchange data

Method 1
Results in deadlock

```
CMMD_send_block(buffer1, node1)
CMMD_receive_block(buffer2, node2)
```

Method 2
Does not result in deadlock but buffers data unnecessarily

```
CMMD_send_noblock(buffer1, node1)
CMMD_receive_block(buffer2, node2)
```

(more about buffering later)

CMMD: how to exchange data (2)

Method 3

To avoid buffering, use

```
mcb1 = CMMD_send_async(buffer1, node1)
call CMMD_receive_block(buffer2, node2)
```

Method 4

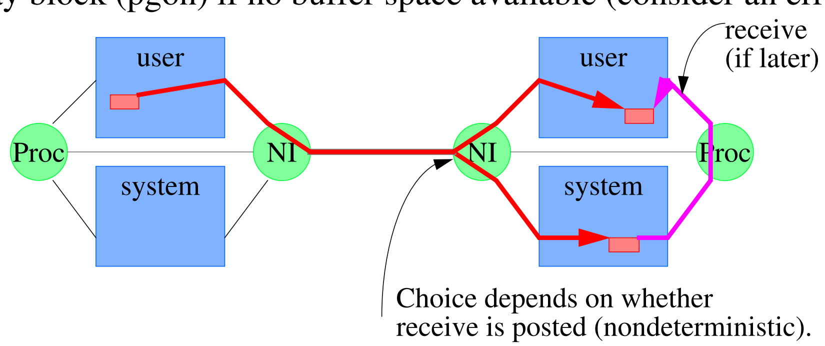
To avoid buffering and overlap computation with communication

```
mcb1 = CMMD_send_async(buffer1, node1)
mcb2 = CMMD_receive_async(buffer2, node2)
... do some calculations ...
CMMD_msg_wait(mcb1)
CMMD_msg_wait(mcb2)
```


NX: `csend()`

Semantics: non-blocking (by practice), synchronous

- Returns whether or not a matching receive has been posted
- Safe to modify original data immediately
- May block (pgon) if no buffer space available (consider an error)



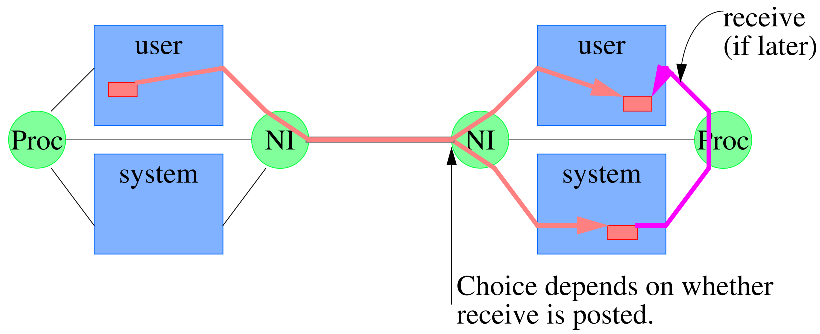
Implementation

- If a receive has been posted, it delivers the message directly.
- If not, it buffers the message in system space on destination node
- Does not return until message has been transferred

NX: Asynchronous Send `isend()`

Semantics: non-blocking, asynchronous

- Returns whether or not a matching receive has been posted
- Not safe to modify original data immediately (use **`msgwait()`**)



Implementation

- If a receive has been posted, it delivers the message directly
- If not, it buffers the message in system space on destination node
- Returns "immediately" before message has been transferred

NX Semantics: `crecv()`

Semantics: blocking, synchronous

- Blocks for message to arrive
- Safe to use data on return

Implementation.

- If a matching message is buffered copies message into user space and returns
- Posts receive for data
- Copies any buffered messages for other posted **`irecv()`**s
- Waits for data to arrive
- Does not return until message has been transferred

NX Semantics: `irecv()`

Semantics: non-blocking, asynchronous

- Does not block waiting for message to arrive
- Cannot use data before checking for completion with msgwait()

Implementation

- If a matching message has arrived (and is buffered)
  - On Paragon/OSF does nothing
  - On iPSC/860 copies message to user buffer (!)
  - On Paragon/SUNMOS copies message to user buffer (!)
- Posts receive and returns

NX: how to exchange data

Works, but **messages are always buffered**:

```
csend(buffer1, node1)
crecv(buffer2)
```

Seems OK but **may result in buffered messages:**

```
mid = irecv(buffer1, node1)
csend(buffer2, node2)
msgwait(mid)
```

Best:

```
mid = irecv(buffer1, node1)
.synchronize (zero-length messages or gsync())
mid2 = isend(buffer2, node2)
... calculate ...
msgwait(mid2)
msgwait(mid)
```

PVM: pack/send

Semantics: non-blocking (depending on whom you ask), synchronous

- Pack message into system buffer (**`pvm_pk*`**)
- Send message (**`pvm_send()`**)
- **`pvm_send()`** returns whether or not there is a matching receive *

Packing: 3 choices

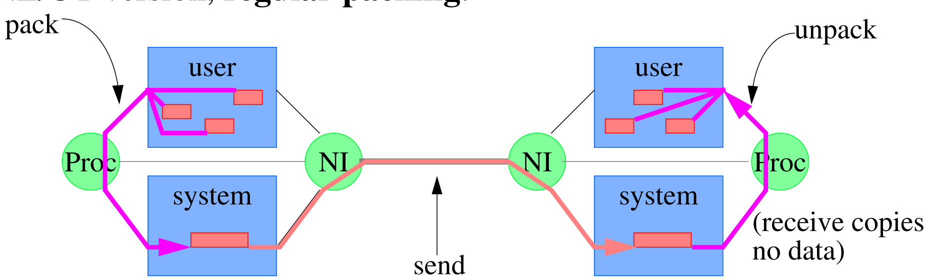
- **Default** packing copies data into a buffer and converts data. (Conversion is currently quite expensive)
- **Raw** packing copies but provides no conversion
- **InPlace** packing does not copy but is restricted to contiguous arrays

Other routines:

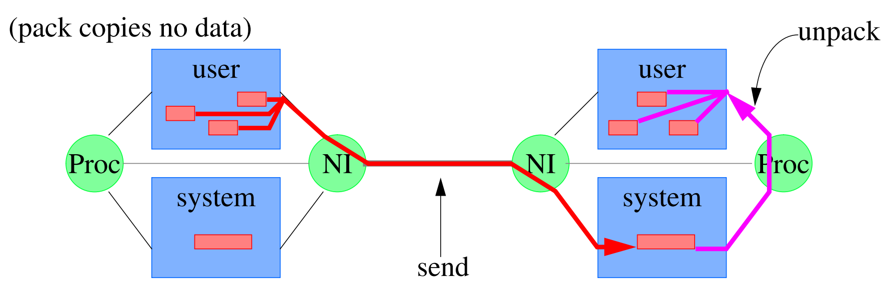
- **`pvm_psend()`** packs (InPlace) and sends in one call
- **`pvm_precv()`** receives and unpacks in one call

PVM pack/send: implementation

ORNL/UT version, **regular packing**:

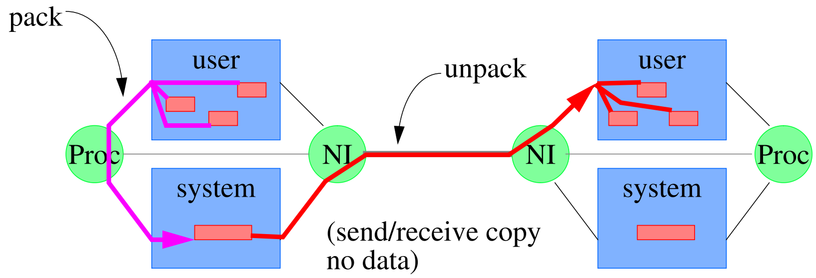


ORNL/UT version, **InPlace packing**

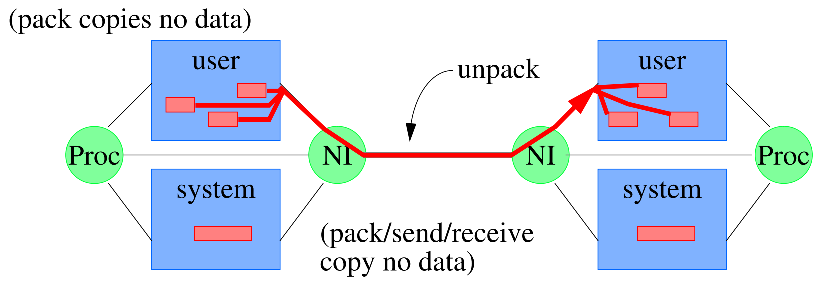


PVM pack/send: implementation

T3D version, **regular packing**:



T3D version, **InPlace packing** (**`pvm_send` is asynchronous!**)



PVM: receive

`pvm_recv()` (regular receive)
- Blocks until a matching message arrives
- Does not copy/transfer any data
- **blocking, synchronous**

`pvm_nrecv()` (nonblocking receive)
- Returns immediately whether or not message has arrived
- Does not post a receive as **`irecv()`** does
- **non-blocking, synchronous**

`pvm_trecv()` (timeout receive)
- Like **`pvm_recv()`**, but if message doesn't arrive within specified period of time, returns.
- Does not post a receive if message hasn't arrived
- **non-blocking, synchronous**

PVM: how to exchange data

There's only one choice:

```
pvm_pack(buffer1)
pvm_send(buffer1, node1)
pvm_recv(node2)
pvm_unpack(buffer2)
```

[This won't block in any current version of PVM, but nothing in the "spec" says it will never deadlock.]

Features:

- Data is always buffered twice, or once with InPlace packing
- Overlap of communication/computation possible except with InPlace packing

MPI: Overview

To send data:

- **MPI_Send()** **MPI_Isend()**
- **MPI_Ssend()** **MPI_Issend()**
- **MPI_Bsend()** **MPI_Ibsend()**
- **MPI_Rsend()** **MPI_Irsend()**

To receive data:

- **MPI_Recv()** **MPI_Irecv()**

Also:

- **MPI_Pack()** **MPI_Unpack()**

MPI: `MPI_Send()`

Semantics: blocking, synchronous

- May block until message is received (depends on implementation) Correct programs work even if **`MPI_Send()`** always blocks.
- Safe to reuse buffer when **`MPI_Send()`** returns

Implementation

- May or may not buffer messages (at source and/or destination).
- Size of buffer unspecified. Implementors “strongly encouraged” to provide buffering (a mistake, IMHO)

MPI: `MPI_Isend()`

Semantics: non-blocking, asynchronous

- Does not block for receive. Returns “immediately”
- Check for completion with **`MPI_Wait()`** before reusing buffer
- **`MPI_Wait()`** returns when message has been safely sent, not when it has been received.

Implementation

- May or may not buffer (no need to, but NX **`isend()`** buffers)

MPI_Recv(), MPI_Irecv()

These have the same semantics as NX **`crecv()`** and **`irecv()`**

`MPI_Recv()`: blocking, synchronous

`MPI_Irecv()`: non-blocking, asynchronous

MPI: other modes

Ready **`MPI_Rsend()`: non-blocking, non-useful (IMHO)**

- Receive must be posted before message arrives, otherwise results are undefined (like force types in NX)
- Often requires synchronization to work
- Performance benefits unclear if **`MPI_Send()`** well-implemented

Synchronous **`MPI_Ssend()`:blocking, synchronous, synchronizing**

- Does not return until message is actually received
- Always synchronizes so performance on type A machines may be poor, hence performance is not portable.
- Performance benefits rare if **`MPI_Send()`** well-implemented

Buffered **`MPI_Bsend()`: non-blocking, synchronous**

- Explicitly buffers messages on sending side
- Uses a user-supplied buffer (ugh!)
- Necessary only if buffer must be reused

A good implementation of MPI_Send()

MPI_Send() can be implemented efficiently on all platforms while maintaining the same semantics.

Type A computer

- Eager protocol for small messages — send message immediately, buffer if no receive posted. Reduces latency, buffering unimportant
- Rendezvous protocol for large messages rendezvous protocol — wait for matching receive then send. Avoids buffering
- Advantage: performance benefits of both Ready and Synchronous mode (see paper on IBM implementation for SP-1)

Type B computer:

- All messages: buffer on sending side, send asynchronously (but synchronously from point of view of user process!)
- Advantage: no performance penalty from synchronization, little penalty from buffering with a slow network

MPI: Recommendations

1. Use **MPI_Send()**, **MPI_Recv()**, **MPI_Isend()**, **MPI_Irecv()**. They can be implemented to be efficient on any platform.

2. Do not use **MPI_Ssend()** unless you have a very good reason. It will probably provide no performance improvement.

3. Do not use **MPI_Rsend()**. There are currently no architectures for which this is more efficient than **MPI_Send()**. Using it correctly usually requires explicit synchronization, eliminating performance benefit.

4. If you must have buffering, use **MPI_Bsend()**. It is difficult to use. Many applications can be restructured to avoid it. MPI: how to exchange data

MPI: How to Exchange Data

Simple and portable way to exchange messages in MPI:

```
sid = MPI_Isend(buffer1, node1)
rid = MPI_Irecv(buffer2, node2)
... computation ...
call MPI_Wait(rid)
call MPI_Wait(sid)
```

For maximum performance on all platforms, this might be better:

```
ids(1) = MPI_Isend(buffer1, node1)
ids(2) = MPI_Irecv(buffer2, node2)
... computation ...
call MPI_Waitall(2, ids)
```


Point-to-point: other features

Tags

- All libraries include a integer message label (tag) as part a **`send`**. (note: MPI and NX restrict the range)
- NX "force type" tags do nothing special on the Paragon

Receipt selectivity

- All libraries allow receipt selectivity by tag
- MPI, CMMD, PVM allow receipt selectivity by source
- All libraries allow wildcards

Data types

- MPI and PVM send typed data,
  - allows heterogeneous operation
  - allows type checking for C programs

Point-to-point: other features (2)

User defined data types

- MPI allows arbitrarily complicated user-defined data types
  - Allows communication hardware to do gather/scatter.
  - Whether hardware-assisted scatter/gather can be done efficiently in practice has not been demonstrated.
  - Pack buffers with MPI or user program? Open question
- PVM can pack strided arrays, but syntax prevents efficiency gain

Message handlers

- CMMD and NX allow you to specify handlers that are called when an asynchronous send or receive completes
- Interrupt handling is hard to do portably

Point-to-Point: Semantics Summary

Routine	Blocking	Synchro-nous	Notes
CMMD_send_block	yes	yes	synchronizing
CMMD_send_noblock	no	yes	
CMMD_send_async	no	no	
CMMD_receive_block	yes	yes	
CMMD_receive_async	no	no	
csend	no	yes	may block if out of buffer space
isend	no	no	buffers unnecessarily
crecv	yes	yes	
irecv	no	no	
pvm_send	no	yes	may block under unusual circumstances
pvm_recv	yes	yes	
pvm_nrecv	no	yes	
pvm_trecv	no	yes	timeout

Point-to-Point: Semantics Summary (2)

Routine	Blocking	Synchro-nous	Notes
MPI_Send	yes	yes	implementation may or may not block
MPI_Isend	no	no	
MPI_Ssend	yes	yes	synchronizes
MPI_Isend	yes	yes	synchronizes, sort of
MPI_Bsend	no	yes	
MPI_Ibsend	no	yes(*)	strange
MPI_Rsend	no	yes	receive must be posted - dubious value
MPI_Irsend	no	no	receive must be posted - dubious value
MPI_Recv	yes	yes	
MPI_Irecv	no	no	

Point-to-point feature summary

Table 1: Point to point library features

Feature	CMMD	NX	PVM	MPI
Ability to overlap computation/ communication	yes	yes	no*	yes
Typed data	no	no	yes	yes
User-defined data types	no	no	no	yes
Possibility of hardware scatter/ gather	no	no	no* (current)	yes
Tag selectivity	yes	yes	yes	yes
Source selectivity	yes	no	yes	yes
Sendrecv routine	yes	yes (bad)	no	yes
Interrupt handlers for send/recv	yes	yes	no	no
Other models	channels	no	no	1/2 channels

Point-to-point: performance

How do you measure point-to-point communication performance?

Good Answer:	Run your application
Bad Answer:	Use vendor-supplied performance figures
Usual Answer:	Measure "important" numbers, generalize

Useful/important numbers:

- **Latency**
- **Bandwidth**
- **Overhead**
- **Buffer copy speed** (if messages are buffered)

Scalability is important but usually a function of network, not library, for point-to-point.

- Aggregate bandwidth
- Contention

Point-to-point performance definitions

Latency and Bandwidth (my definition)

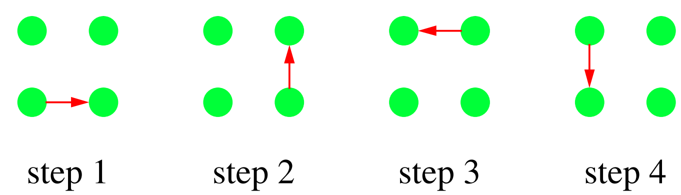
```
transfer_time = latency + message_size/bandwidth
```

Overhead (my definition, related to definition of D. Culler)

Overhead is the amount of communication time you can't hide by using asynchronous communication.

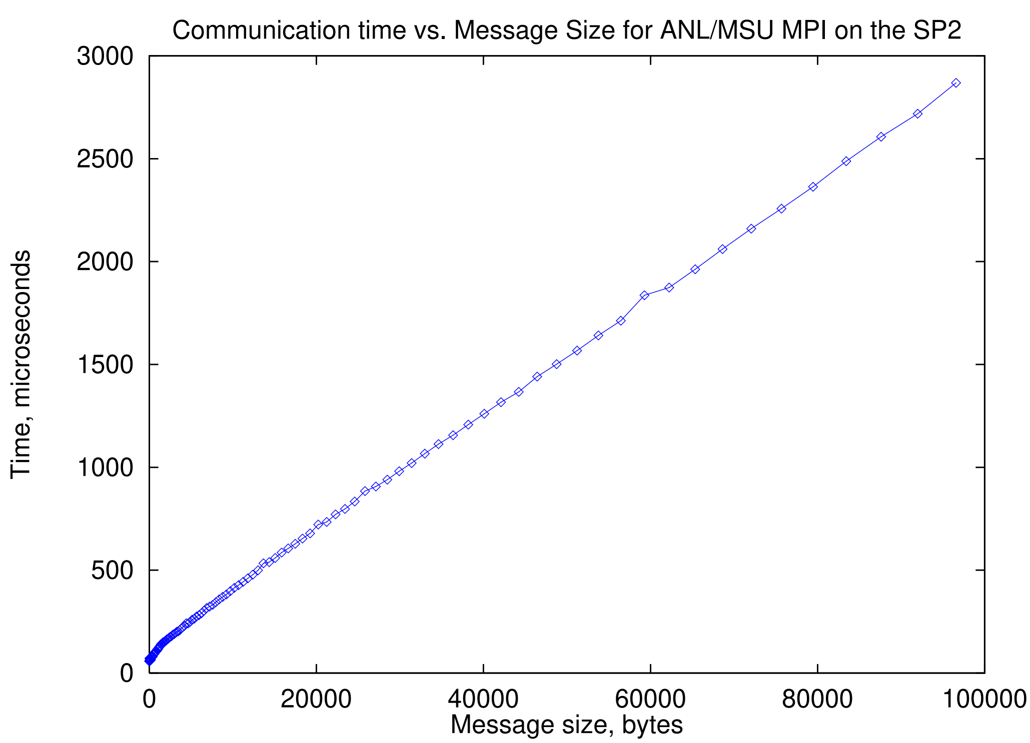
How to measure latency/bandwidth

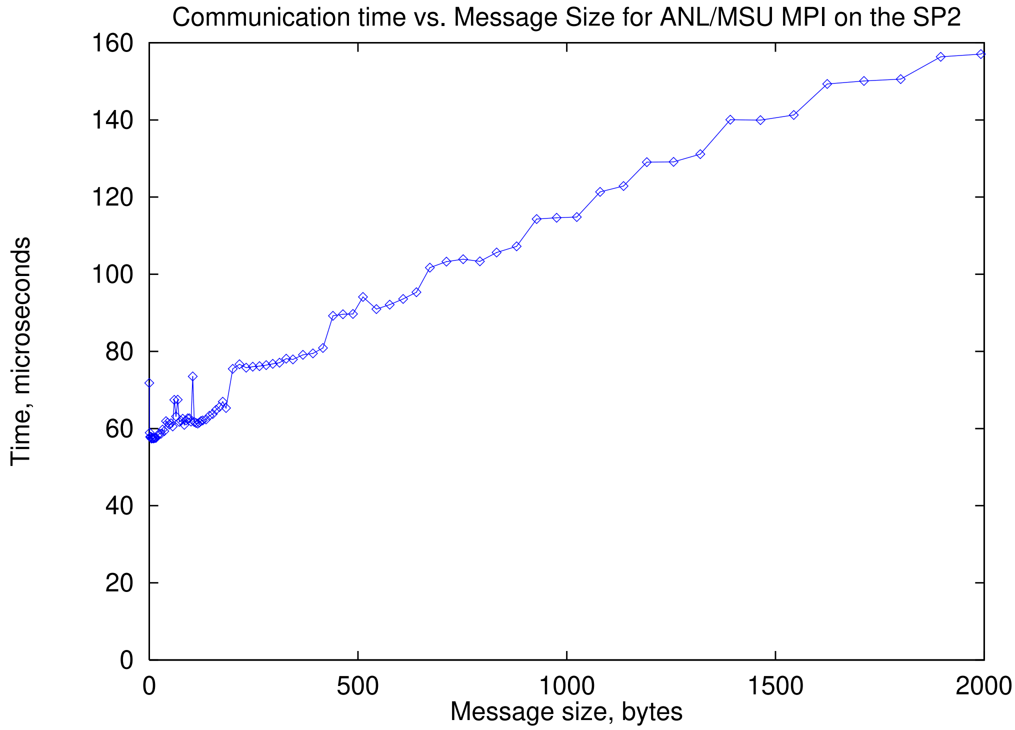
There are lots of ways to measure this.
Token-ring benchmark:

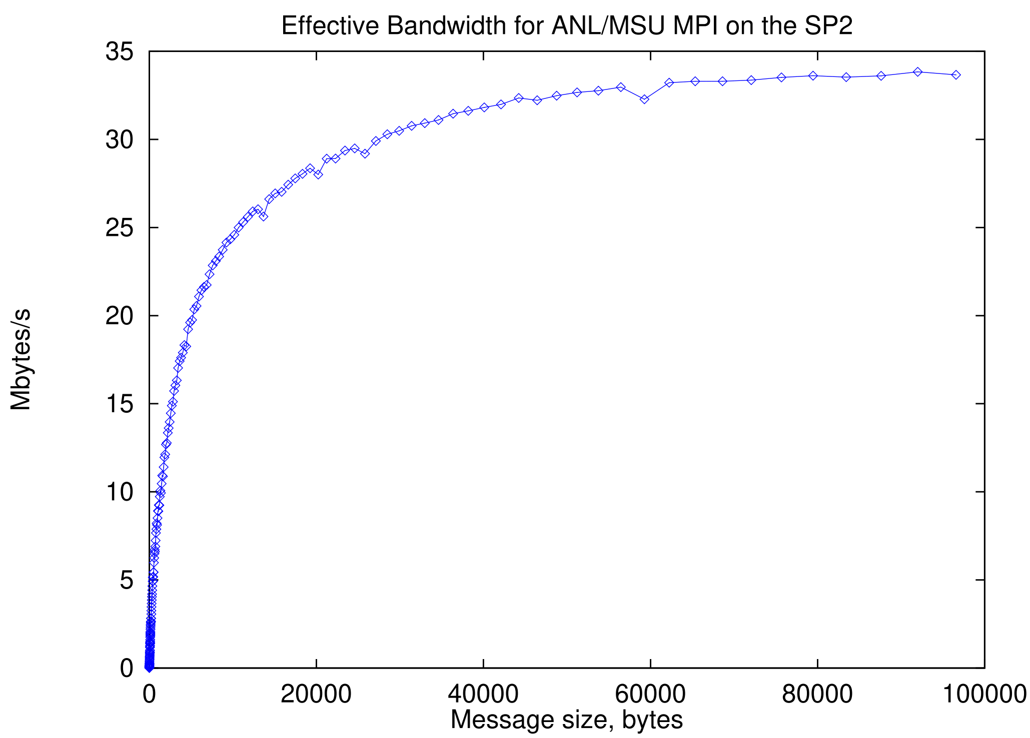


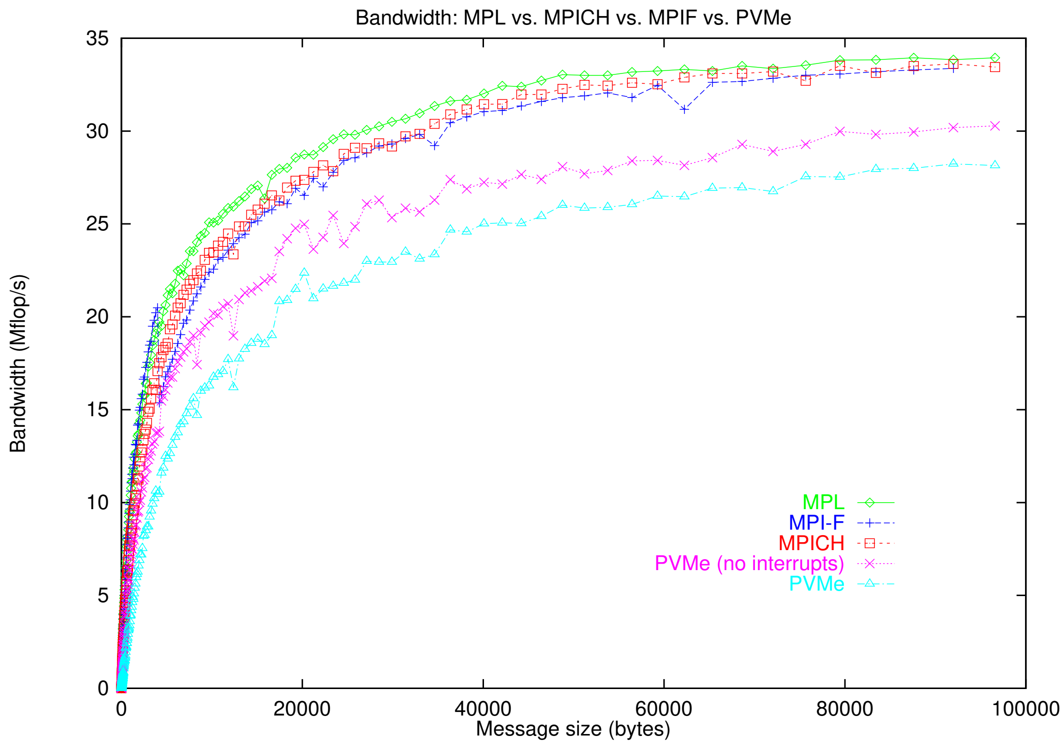
Comments

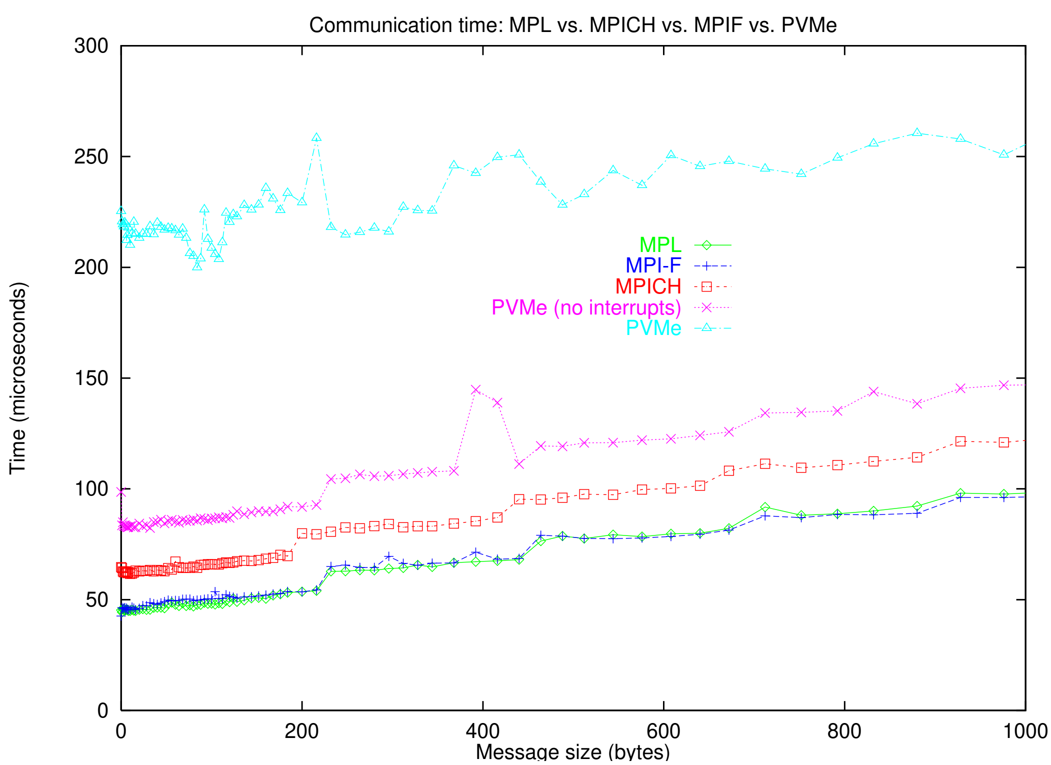
- Amortize startup costs by looping many times
- Ping-pong is a special case
- Choice of measuring asynchronous calls (interrupt) or synchronous calls (poll). Following examples use synchronous calls.
- **Does not measure overhead or buffer copy speed (no buffering)**




Communication time vs. Message Size for ANL/MSU MPI on the SP2
Time, microseconds
160
140
120
100
80
60
40
20
0
0
500
1000
1500
2000
Message size, bytes


Effective Bandwidth for ANL/MSU MPI on the SP2
Mbytes/s
35
30
25
20
15
10
5
0
0
20000
40000
60000
80000
100000
Message size, bytes


Bandwidth: MPL vs. MPICH vs. MPIF vs. PVMe
Bandwidth (Mflop/s)
Message size (bytes)
0
5
10
15
20
25
30
35
0
20000
40000
60000
80000
100000
MPL
MPI-F
MPICH
PVMe (no interrupts)
PVMe


Communication time: MPL vs. MPICH vs. MPIF vs. PVMe
Time (microseconds)
Message size (bytes)
0
50
100
150
200
250
300
0
200
400
600
800
1000
MPL
MPI-F
MPICH
PVMe (no interrupts)
PVMe

Performance summary

	Latency (us)	Bandwidth (MB/s)
CMMD 3.2 (CM-5)	75	8.2
NX/Paragon (R1.2)	167	56
MPICH/Paragon (R1.2)	210	21
PVM /Paragon (PVM 3.2, R1.2, Raw)	540	7.7
PVM/Paragon (PVM 3.2, R1.2, Default)	540	0.8 (!)
NX /Paragon R1.2 wcp	58	91
MPL/SP2	46	34
MPIF/SP2	46	34
MPICH/SP2	58	33
PVMe/SP2 (InPlace packing)	225	28
PVMe/SP2 (InPlace packing, no interrupts - may block)	83	31
PVM 3.2/ethernet/SGI R3000	~8000	~0.3
PVM 3.2/FDDI/SGI R3000	~5000	~0.8

* wcp = with message co-processor

Optimization revisited

There are many things you can do to optimize communication at a low level. Here are several of them:

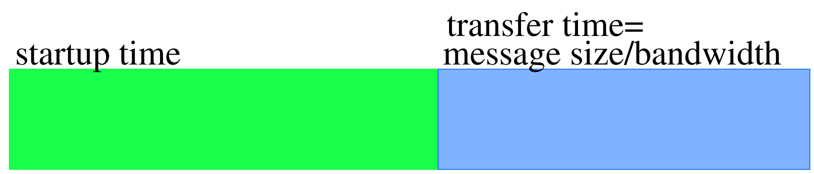
- **Overlap communication with computation**
- **Avoid unnecessary synchronization**
- **Avoid buffering**
- **Beware of interrupts**
- **Align data**

Message transfer time in more detail

Data transfer time expressed in terms of

- **startup time** (latency)
- **bandwidth**

transfer time = startup time + message_size/bandwidth

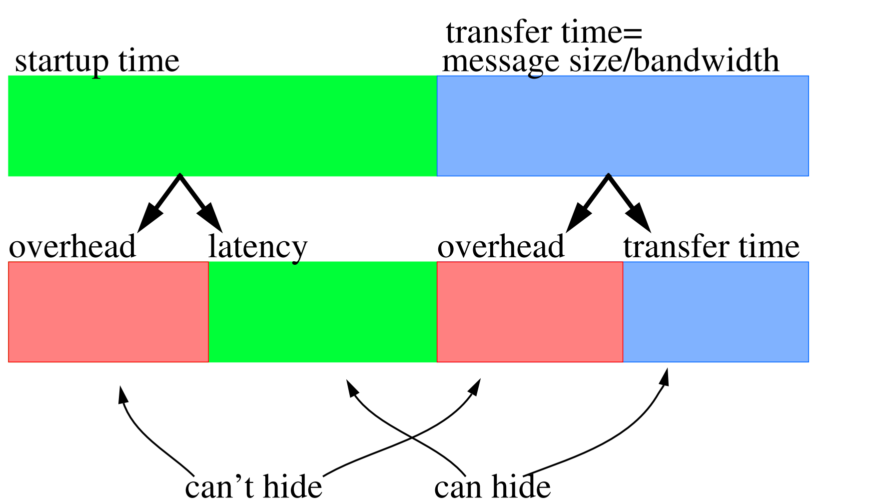


- Bars indicate elapsed time on sender/receiver combined

Communication overhead

How much of your communication time can you hide by using asynchronous communication (no buffering)?

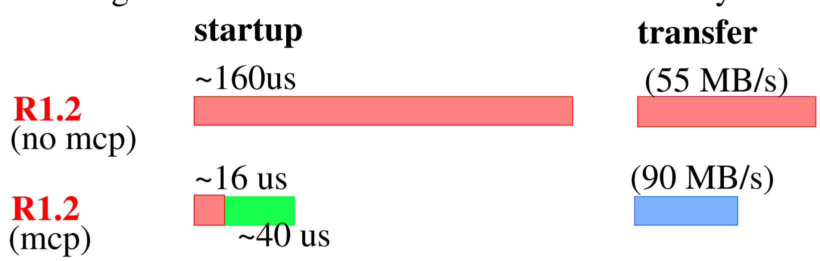
Depends on hardware



Overlap: Theory vs. Practice

- Ability to overlap requires data transfer independent from processor
- Many MPPs can't overlap computation with communication even though they have asynchronous communication routines

Intel Paragon is one of few machines that can actually overlap



(data comes from timing `isend()` and shows lower bound on overhead)

Machines that can't: CM-5, SP2, iPSC/860, T3D
Machines that can: Paragon (with coprocessor), some clusters (partially)

Synchronization

Q: Why use asynchronous semantics if you can't actually overlap computation and communication?

A1: Networks are becoming smarter. Future machines (and Paragon) may be able to overlap

A2: To avoid synchronization

- A processor that is waiting for data is not performing FLOPS.
- If the data is already there, processor doesn't have to wait.
- For type A applications, small fluctuating load imbalance (e.g., due to competing Unix daemons) can kill performance because of synchronization delays.

Interrupts

How is data transferred with asynchronous semantics but without overlap?

- Sending processor may have to participate in part of transfer
- Receiving processor may have to participate "synchronously" when waiting for message to complete.
- Receiving processor may be interrupted to receive data

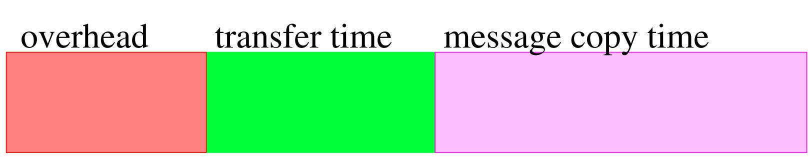
So what?

- Normally not a big problem
- In **fine-grained software pipelines**, (e.g. in ADI algorithms) interrupts may kill performance by causing load imbalance
- Rob Van der Wijngaart, Sekhar Sarukkai, Pankaj Mehra, "*Analysis and Optimization of Software Pipeline Performance on MIMD Parallel Computers*", submitted to IEEE Journal of Parallel and Distributed Computing

Buffering

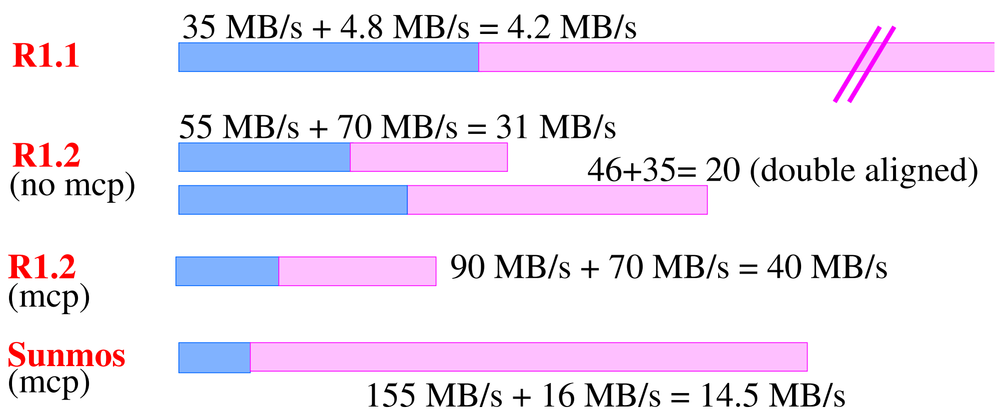
- When a message is buffered (by communication library), it is copied from system space to user space by the processor
- Buffering reduces effective bandwidth
- Buffering can't be overlapped with computation
- Importance of buffering depends on memory bandwidth
- Buffering may occur in user program (e.g. to pack strided arrays) or in message passing library.

Transfer time (bandwidth effects only — ignore latency)



Buffer copy time - Paragon

- Copying from system space to user space is done by the processor and can be quite slow (in OSF, done by the **`bcopy()`** routine)



Note: Horizontal axis is TIME — MB/s shown for convenience

Optimization: Summary of libraries

CMMD
- No overlap possible
- Buffering easy to avoid
- Interrupts not a problem

NX
- Overlap possible on the Paragon
- Buffering nondeterministic, slow, difficult to avoid
- Interrupts can be a problem without coprocessor

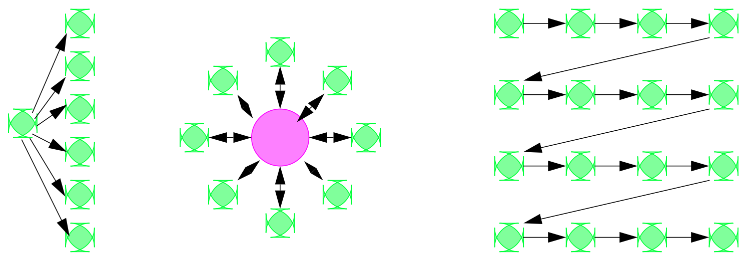
PVM
- Overlap possible in principle, but only with double buffering
- Buffering unavoidable, costly (can avoid some packing InPlace)
- Interrupts may be a problem

MPI
- Overlap possible in principle
- Buffering (all types!) can be avoided
- If interrupts are a problem, can be avoided by synchronous mode

Collective Communication

Collective communication is communication among a group of processes:

- Broadcast
- Synchronization (barrier)
- Global operations (reductions)
- Scatter/gather
- Parallel prefix (scan)



Collective communication (2)

Semantics/Implementation

- All collective operations defined by NX, MPI, CMMD, PVM are blocking, synchronous and synchronizing.
- Collective operations may be implemented with point-to-point
- Collective operations may be implemented in hardware

Performance

- Hardware operations may go extremely fast: CM-5/CMMD
- Simple operations (broadcast, global sum) should scale as ln N if implemented with point-to-point (if network scales)
- Complex operations (all-to-all) should scale as N (if network scales well)

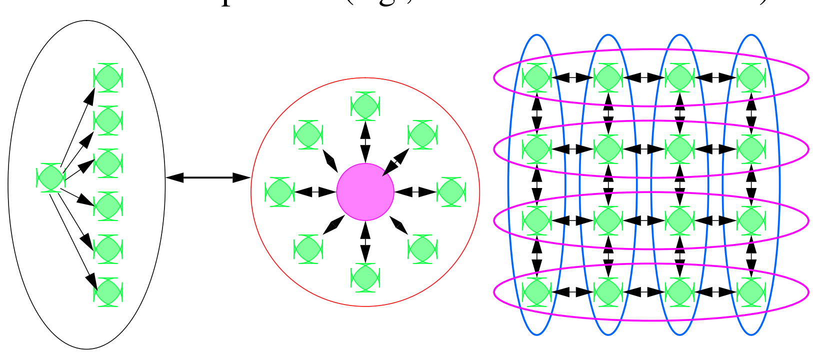
Collective Communication: Summary

	bcast	barrier (sync)	reduce	user reduce	scan	scatter	gather	all-to-all
CMMD	yes (fast)	yes	all-all (fast)	no	yes (fast)	no	all-all	no
NX	yes	yes	all-all	yes	no	no	all-all	no
PVM	yes (slow)	yes (slow)	yes (slow)	no	no	no	no	no
MPI	yes	yes	all-all all-one	yes	yes	yes	all-all, one-all	yes

- PVM operations are nonscalable and slow
- Most CMMD operations are implemented in hardware and have very low latency (independent of application size).
- NX operations are implemented using point-to-point messages and have latency that grows with application size O(ln N).
- MPI defines many operations, but performance is implementation-dependent

Support for Groups

A group is a set of tasks.
- Groups can be used to organize an application
- Collective operations can be defined over a group.
- Multiple groups are convenient for certain applications:
  - Interdisciplinary applications (e.g., fluid-structures)
  - Linear algebra: rows/columns of a matrix
  - Domain decomposition (e.g., multi-zonal calculations)



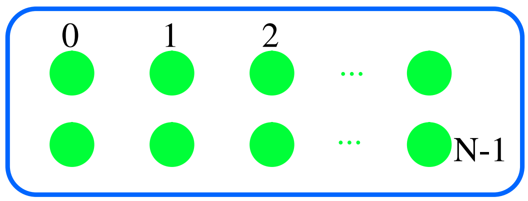
Support for Groups: NX and CMMD

CMMD

- Provides support for initial group only (0 ... N-1)

NX

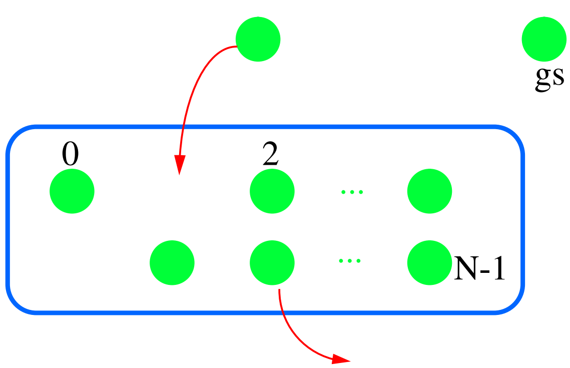
- Provides support for initial group only
- Support for multiple groups (for interdisciplinary apps) available through intercube library (iPSC/860) and map library (Paragon) from NAS.



Support for Groups: PVM

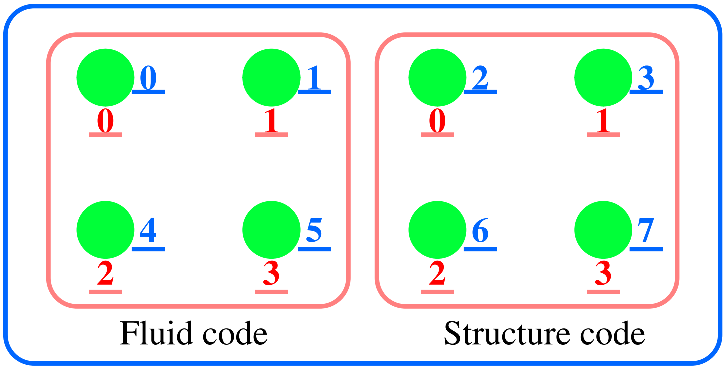
UT/ORNL implementation, version 3.3

- Groups are dynamic - tasks may enter and leave at any time
- Requires group server to resolve race conditions
- Collective operations go through group server.
  This forces extreme inefficiency.
- No support for “original group” 0...N-1. (often inconvenient)



Support for Groups: MPI

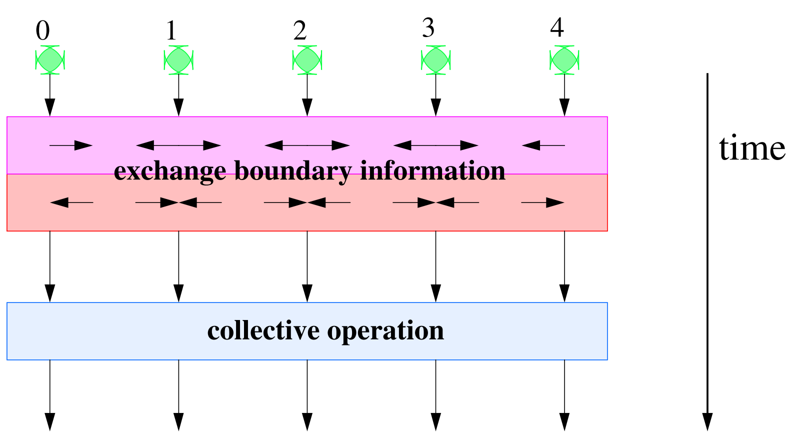
- Provides extensive support for groups, including overlapping groups (overlap possible in PVM but not as useful)
- Provides extensive support for virtual topologies on groups
- Does not specify how to launch several SPMD applications simultaneously. Initialization of tasks not defined in the standard



Why is context necessary?

Hypothetical SPMD program:

- exchange boundary information
- compute
- perform collective operation (e.g., calculate total energy)



Context: Example

```
... calculate ...
send(task_on_right, tag);
send(task_on_left, another_tag);
do twice
    recv(anytask, anytag);
    ... process data ...
end do
... more calculations ...
collective_operation();
```

Possible scenario:

- task 3 waits for information from tasks 2 and 4
- task 4 delayed for some reason, so task 3 waits for info from task 4
- meanwhile, task 1 has exchanged boundary info with 0 and 2
- task 1 starts global reduction, in which it sends a message to task 3
- task 3 receives data from task 1 instead of from task 4

Context

Comments on example:

- Cannot be fixed (even with synchronization)
- Intermittent errors (timing dependent)
- Program appears to be correct.
- Demonstrates one lossage mode of many (big subject)

Possible solution: add a second tag, called a **context**.

```
send(destination, tag1, tag2)
```

wildcardable

not wildcardable (context)

```
recieve(destination, tag1, tag2)
```

Context

CMMD
- Has enough support to protect point-to-point from global operations

NX
- Has enough support to protect point-to-point from global operations

PVM
- No support (except in *Unify*)

MPI
- Extensive support for context
- Group and context merged into a "communicator" - overlapping groups work correctly.
- Additional support for caching information with communicators, generating new communicators, etc., to facilitate library development.

Process/Machine Management

Machine configuration
- NX, CMMD, MPI support static machine only
- PVM supports dynamic host add/delete

Static Processes
- CMMD supports single binary only (+ host/node)
- NX, MPI, PVM support multiple binaries (MPI does not specify how)

Dynamic Processes
- NX, CMMD, MPI support only static processes
- PVM supports adding/deleting processes at any time (this can be intentional, or to support fault tolerance)

PVM Process/Machine Management

PVM supports (through library routines and through “console”)

- Spawning tasks (including heterogeneous environments)
- Killing tasks (and sending arbitrary signals)
- Automatic detection/notification of task exit
- Automatic detection/notification of machine configuration change
- Adding hosts to virtual machine
- Deleting hosts from virtual machine
- Examining machine or task configuration
- Obtaining resources through a resource manager (e.g., Condor)

Comments:

- PVM is the only library that supports any of these
- Supports writing fault-tolerant programs
- Functionality is critical to certain (usually type B) applications.
- Flexibility can get in the way when trying to do something simple

I/O

Distinguish between:

1. **Library**
    - Support for **cooperative parallel I/O** (all nodes read from/write to same file).

2. **Hardware**
    - Fast I/O (fast for one node, fast for many nodes)
    - Scalable I/O (fast for one node, N times as fast for N nodes)
      Must be compatible with cooperative I/O.

	Support for Cooperative I/O	Support for Fast I/O	Support for Scalable I/O
CMMD	yes	yes (SDA)	yes (SDA)
NX	yes	yes (PFS, CFS)	yes (PFS,CFS)
PVM	no	depends on hardware	only for independent I/O
MPI	no	depends on hardware	only for independent I/O MPI-IO in progress

Cooperative I/O Motivation

Two reasons for cooperative I/O

- convenient semantics
- performance

How should this program work?

```
program hello
integer n
... startup code to determine self address...

write(6, *) 'hello from processor ', n

end
```


Cooperative I/O (2)

Option 1. All processors respond. **No particular order**

```
hello from processor 0
hello from processor 29
hello from processor 8
hello from processor 4
hello from processor 6
...
```

Option 2. **All processors respond sequentially**

```
hello from processor 0
hello from processor 1
hello from processor 2
hello from processor 3
hello from processor 4
...
```

Option 3. **Jumbled output**

```
helhello fromh prohelcelo fror 0 f
rohe
...
```

[And what if this is data, not output to the terminal?]

Cooperative I/O (3)

How should this program work?

```
program hello
integer N, input
... startup code to determine node self address ...

read(5, *) input

write(6, *) 'got ', input, ' on processor ', N

end
```


Cooperative I/O - NX and CMMD

NX and CMMD provide similar support for cooperative I/O

Global file pointers: only one file pointer, shared by all processes

1. **Each node has independent access**
   - CMMD_independent (or CMMD_local)
   - M_UNIX
2. **Each node appends to file on first-come, first-served**
   - CMMD_independent with append mode on
   - M_LOG
3. **All operations synchronize; reads/writes performed in node order**
   - CMMD_sync_seq
   - M_SYNC (or M_RECORD, for same size on all nodes)
4. **All operations synchronize; same read/write on each node**
   - CMMD_sync_bc
   - M_GLOBAL

Cooperative I/O - PVM and MPI

There are projects to add cooperative I/O to both PVM and MPI

PVM

- PIOUS - developed by Sunderam and Moyer, Emory Univ.
- Currently in Beta test
- Papers at

`ftp://ftp.mathcs.emory.edu:pub/vss/piousifip94.ps`
`ftp://ftp.mathcs.emory.edu:pub/vss/piousshpcc94.ps`

MPI

- BOF at SC '94 7:30 PM Wednesday (after MPI BOF)
- Groups at NAS (NASA Ames) and IBM working on MPI-I/O
- Implementation in 1Q95
- Send mail to *mpi-io@nas.nasa.gov* for more info

Porting

With a few important exceptions, porting is easy.
The issues are usually maintenance and node-level optimization.

Porting pitfalls

1. **Target library has significantly less functionality than original library**
   - Groups in CMMD, NX
   - Lack of process control in CMMD, NX, MPI
   - Certain group operations in NX, PVM
   - I/O in PVM, MPI

2. **Target library has poor or unusual performance characteristics**
   - PVM point-to-point performance (including lack of ability to overlap computation and communication)
   - PVM group operation performance
   - NX nondeterministic buffering

Porting (2)

Porting pitfalls, continued.

3. **Original library has limited functionality**
   - Application may require restructuring to make best use of new features (e.g., collective operations, point-to-point functionality)

4. **There are hardware-specific optimizations in the original code**
   - Gray codes for hypercube-based algorithms
   - Synchronization to get bidirectional transmission (iPSC/860)

Summary: NX

NX is a dinosaur.
NX is not elegant or well-documented, and has some implementation problems (buffering).

BUT you probably want to use it on the Paragon:

- It's the fastest library (by far, currently) available on the Paragon
- It's the only library that supports Parallel I/O on the Paragon
- It overlaps communication and computation if you have a Paragon with coprocessors.

Use NX if you're going to run on a Paragon and nothing else.

Summary: CMMD

CMMD is moot.
CMMD is elegant, well-documented, well-designed, well-implemented.

BUT:

- You have to use nodal CM Fortran to get performance from CM5 nodes (f77 can't use the vector units).
- Future is uncertain

If you're using a CM-5 program in CM Fortran (global, without CMMD).

Summary: PVM

Major strength: support for dynamic parallelism
Major weaknesses: functionality, performance, robustness

- Appropriate for type B problems
  - only library to support dynamicism

- Not as appropriate for type A problems
  - Poor performance
  - Missing functionality

PVM is almost more of a distributed operating system than a message passing library. Its functionality is essential and will hopefully be integrated into MPI.

Summary: MPI

- Large step forward in support for library development.
  - contexts
  - groups
- Other new functionality
  - thread safety
  - user-defined data types
  - gather/scatter on send/receive
  - virtual topologies
- Can handle anything the other libraries can, with the exception of I/O and dynamicism. MPI-2 will probably add these.
- Offers portability and performance for Type A problems.

MPI isn't perfect but it's good enough and it's a standard. Hopefully, MPI is the future. Don't be intimidated by the MPI document.

PVM: Where to get more information

- **WWW home page:**
  - **`http://hebe.nectar.cs.cmu.edu:8001/`**
- **Book**
  - *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Network Parallel Computing* by Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. Published by MIT press.
  - Available by anonymous ftp from **`netlib2.cs.utk.edu`** in **`pvm3/book/pvm-book.ps`**
  - More information at **`http://www-mitpress.mit.edu/`**
- **Newsgroup:** **`comp.parallel.pvm`**

MPI: Where to get more information

- **WWW home pages**
  - `http://www.erc.msstate.edu/mpi`
  - `http://www.mcs.anl.gov/mpi/index.html`

- **Book**
  - *Using MPI*, by William Gropp, Ewing Lusk, and Anthony Skjellum, published by MIT Press ISBN 0-262-57104-8.
  - More information at `http://www-mitpress.mit.edu/`

- **Newsgroup:** `comp.parallel.mpi`