The Interoperable Message Passing Interface (IMPI) Extensions to LAM/MPI

Jeffrey M. Squyres, Andrew Lumsdaine

Department of Computer Science and Engineering

University of Notre Dame

William L. George, John G. Hagedorn, Judith E. Devaney

National Institue of Standards and Technology

Overview

- Introduction
- IMPI Overview
- LAM/MPI Overview
- IMPI Implementation in LAM/MPI
- Results
- Conclusions / Future Work

Introduction to IMPI

- Many high quality implementations of MPI are available
  - Both freeware and commercial
  - Freeware implementations tend to concentrate on portability and heterogeneity
  - Commercial implementations focus on tuning latency and bandwidth
- Allows for a high degree of portability between parallel systems

The Problem

- Each implementation of MPI is unique
  - Underlying assumptions and abstractions are different
  - Messaging protocols are custom-written for hardware
- Different MPI implementations cannot interoperate
  - Cannot run a parallel job on multiple machines while utilizing each vendor's highly-tuned MPI

A Solution

- The IMPI Steering Committee was formed to address these issues
- The Committee consisted of vendors who already had high-performance MPI implementations
- Main idea: propose a small set of protocols for starting a multi-implementation MPI job, passing user messages between the implementations, and shutting the job down
- Proposed standard: **http://impi.nist.gov/IMPI/**

LAM's Role in IMPI

- The LAM/MPI Team was asked to join as a non-voting member
- Continues a history of providing a freeware "proof of concept" implementation of proposed standards
- LAM/MPI Team provided both a first implementation of the IMPI protocol, but also an MPI-independent implementation of the IMPI server (described shortly)

Related Work

- PVMPI / MPI Connect: University of Tennessee
  - Use PVM as a bridge between multiple MPI implementations
- Unify: Eng. Research Center / Mississippi State University
  - Allows both PVM and MPI in a single program
- Problems with previous approaches
  - Use of non-MPI functions
  - Subset of MPI-1 (e.g., no **INTERCOMM_MERGE**)
  - Incomplete **MPI_COMM_WORLD**

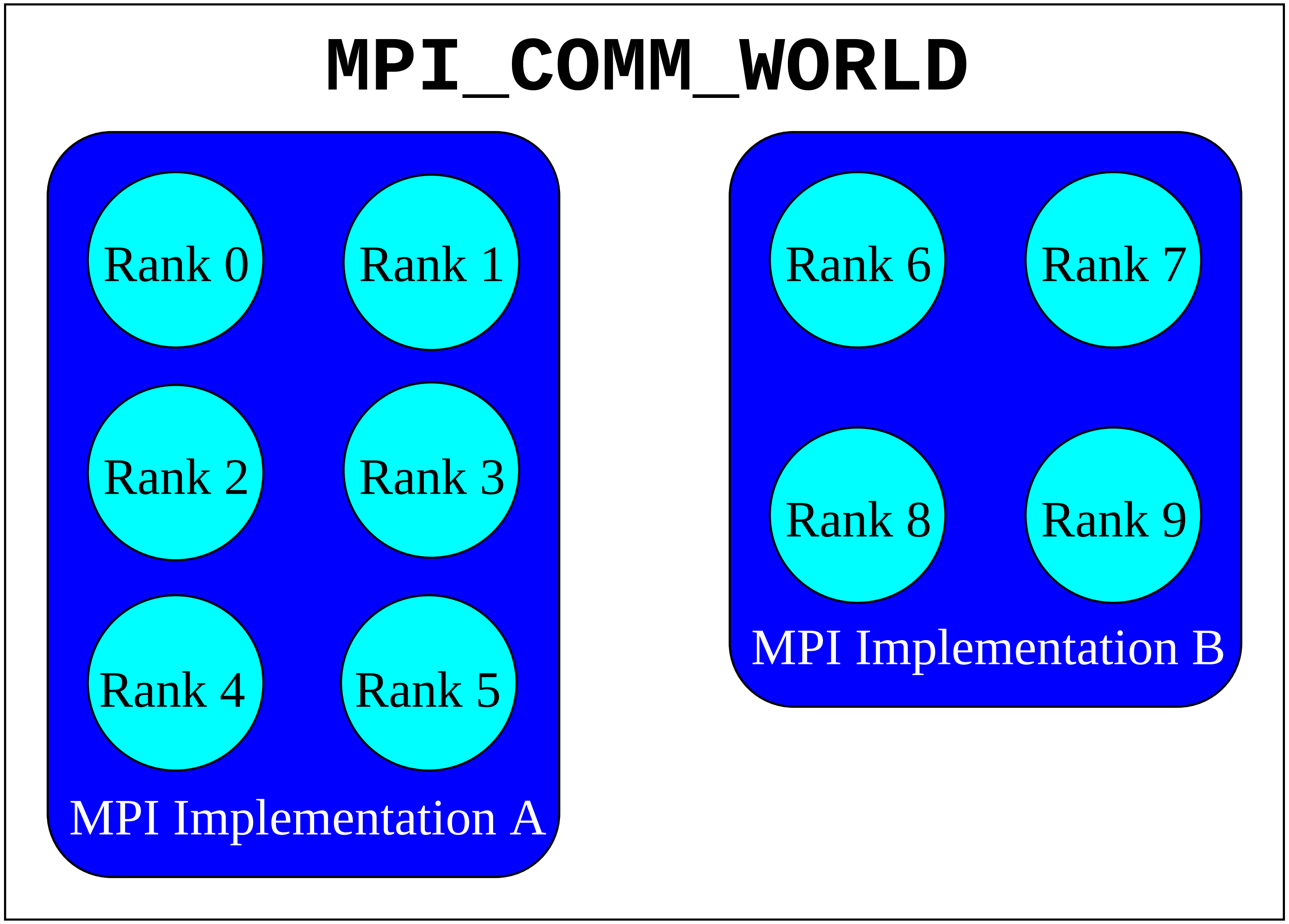
Overview

- Introduction
- IMPI Overview
- LAM/MPI Overview
- IMPI Implementation in LAM/MPI
- Results
- Conclusions / Future Work

IMPI Goals

- User goals
  - Same MPI-1 interface and functionality; any MPI-1 program should function correctly under IMPI.
  - Provide a “complete” **MPI_COMM_WORLD**
- Implementation goals
  - Standard way to start and finish multiple MPI jobs
  - Common data passing protocols between implementations
  - Distributed algorithms for collectives

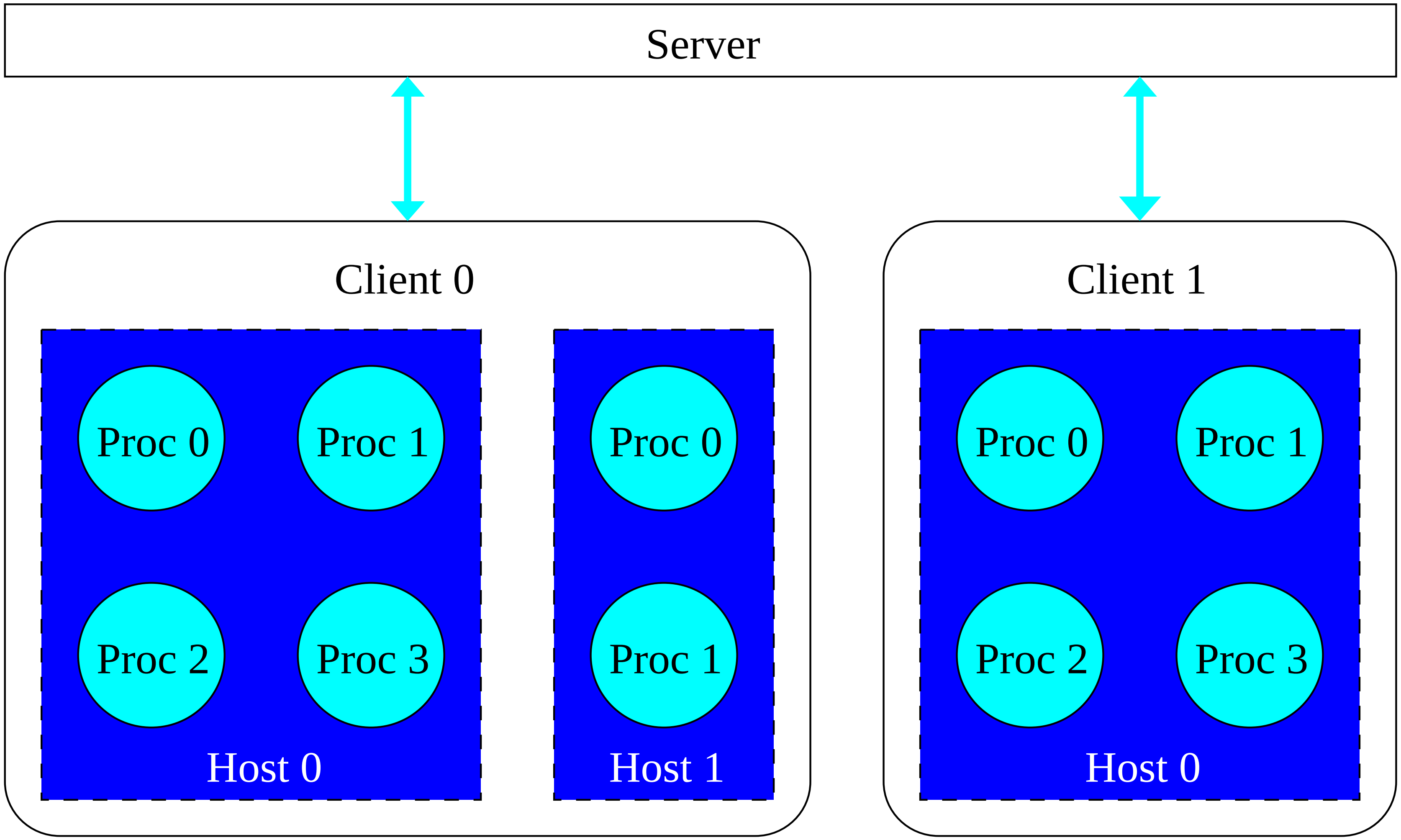
Complete MPI_COMM_WORLD



Terminology

- Four main IMPI entities
  - **Server**: Rendezvous point for starting jobs
  - **Client**: One client per MPI implementation; connects to server to exchange startup/shutdown data
  - **Host**: Subset of MPI ranks within an implementation
  - **Proc**: Individual rank in **MPI_COMM_WORLD**

The Big Picture



Startup Protocol

- A two step process used to launch IMPI jobs:

  1. Launch the server

  2. Launch the individual MPI jobs

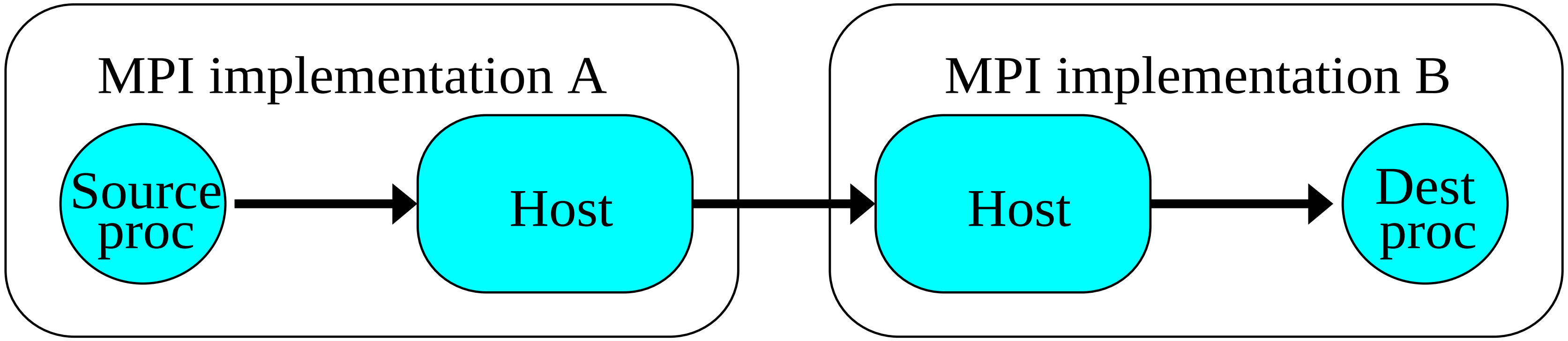
- The clients connect to the server and send startup information

- Server collates all information and re-broadcasts to all clients

- Clients use this data to form a complete **MPI_COMM_WORLD**

Connecting Hosts

- After the clients have received the server data, hosts make a fully-connected mesh of TCP/IP sockets
- User data will travel across these sockets (e.g., **MPI_SEND**)

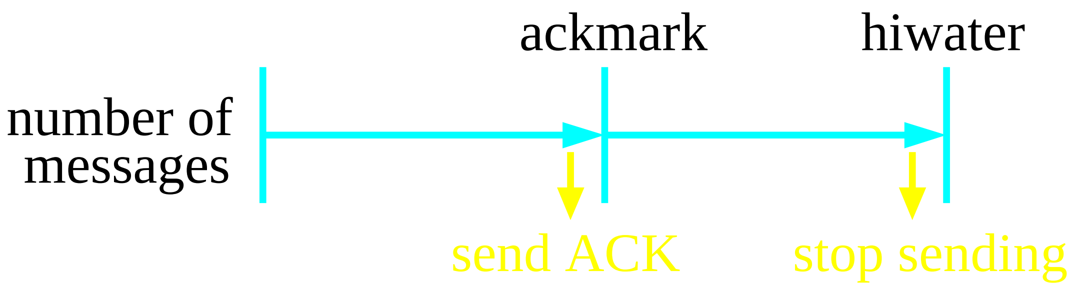


Data Transfer Protocol

- Only messages *between* implementations are regulated in IMPI
- Messages within a single implementation are not standardized
- User data is passed between procs on different implementations via hosts
  - This causes a potential communication bottleneck
  - But IMPI communication is expected to be slow anyway
  - Note that a single implementation may have multiple hosts; those messages are not regulated

Message Packetization

- Messages between hosts are packetized
- Several values are negotiated during startup
  - **maxdatalen**: Maximum length of payload in IMPI packets
  - **ackmark**: Between each host pair, an ACK must be sent for every **ackmark** received packets
  - **hiwater**: Messages can continue to be sent until **hiwater** packets have not been acknowledged

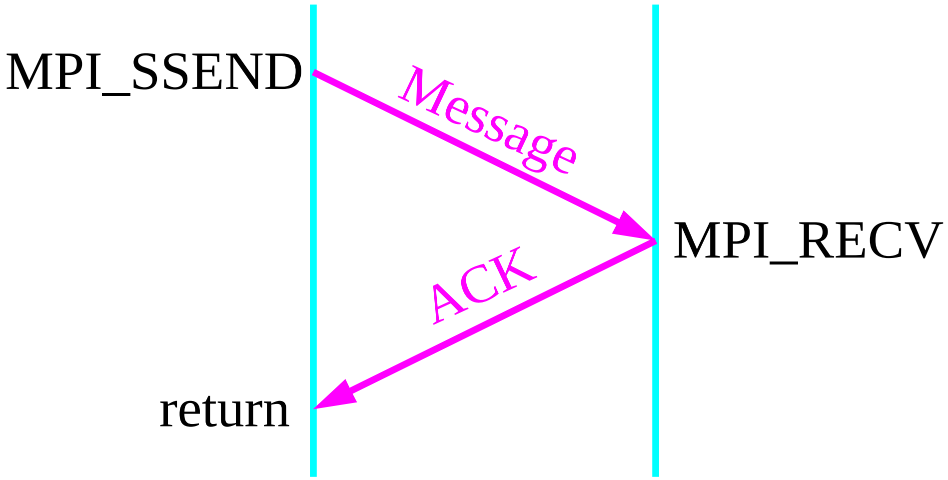


Data Protocols

- Short message protocol
  - Non-synchronous messages $\leq$ **maxdatalen** bytes are sent eagerly in one packet
- Long message protocol
  - Messages $>$ **maxdatalen** bytes are fragmented into packets of **maxdatalen** bytes
  - The first packet is sent eagerly (like short messages)
  - The receiver will send an ACK when it has allocated resources to receive the rest of the message

Synchronous Messages

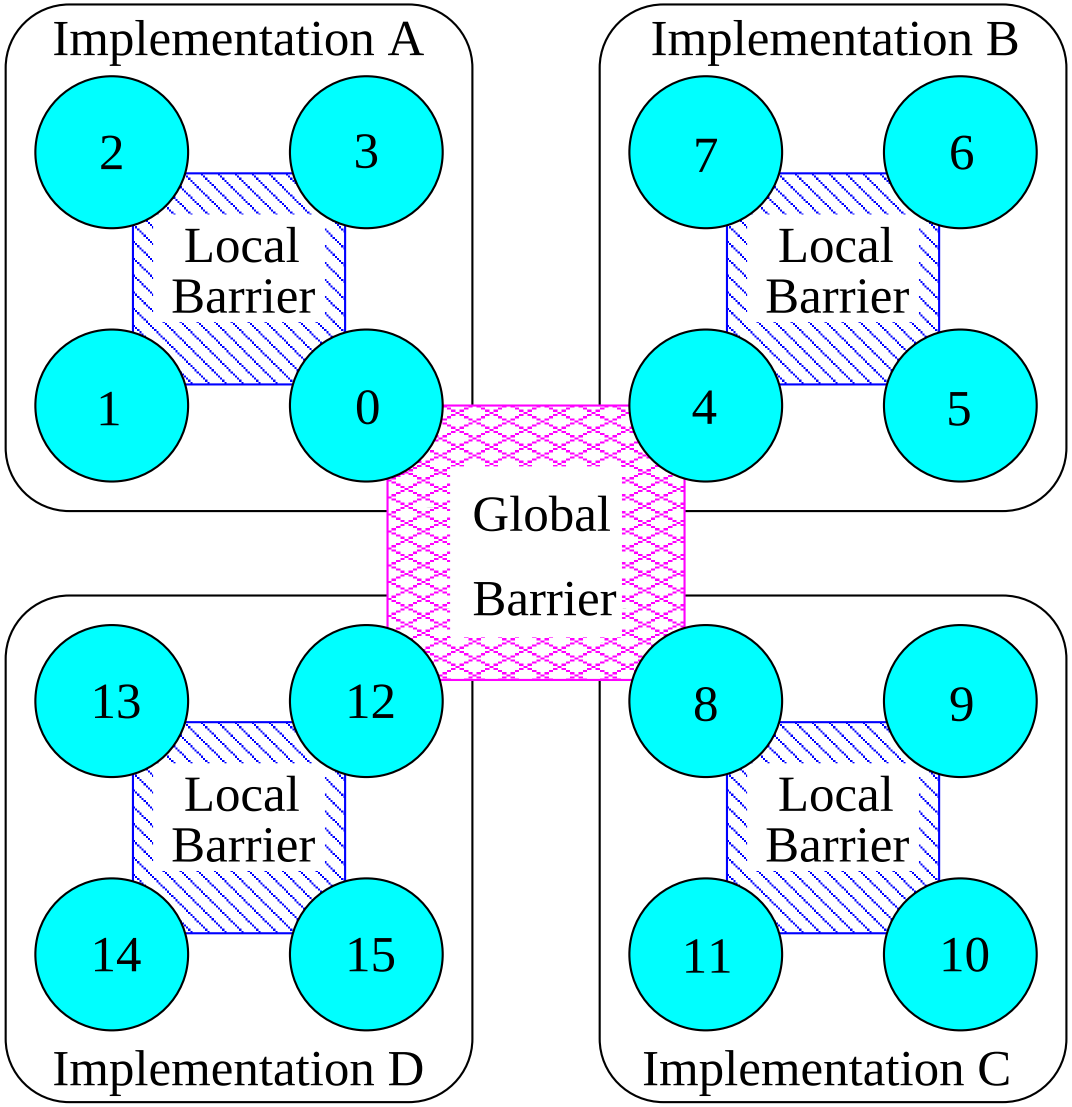
- **MPI_SSEND**: Returns when message has begun to be received
  - Always uses the long message protocol
  - Can use the ACK in the long protocol



Collective Algorithms

- IMPI implementations must share common collective algorithms so that they know their role in the larger computation
- Affects both data-passing collectives (e.g., **MPI_BCAST**) and communicator constructor / destructors (e.g., **MPI_COMM_SPLIT**)
- Pseudocode for all MPI collectives are in the IMPI standard
  - Utilizes very low cross-implementation communication
  - Usually has “local” and “global” phases

MPI_BARRIER Collective Algorithm



NIST Conformance Tester

- NIST has implemented a Java applet to test IMPI implementations
  - Emulates IMPI server, clients, hosts, and procs
  - C source code provided to compile / link against the IMPI implementation being tested
  - Run the resulting program, link up to the Java applet
  - A series of tests can be run from the Java client
- Available on the NIST IMPI web site

Shutdown Protocol

$$\text{proc} \xrightarrow[\text{message}]{\text{shutdown}} \text{host} \xrightarrow[\text{message}]{\text{shutdown}} \text{client} \xrightarrow[\text{message}]{\text{shutdown}} \text{server}$$

- As each proc enters **MPI_FINALIZE**, it sends a message to its host indicating that it is finished
- When a host gets finalize messages from all of its procs, it sends a message to its client
- Similarly, the client sends a message to the server when its hosts are finished
- The server quits when it receives a message from each client

Overview

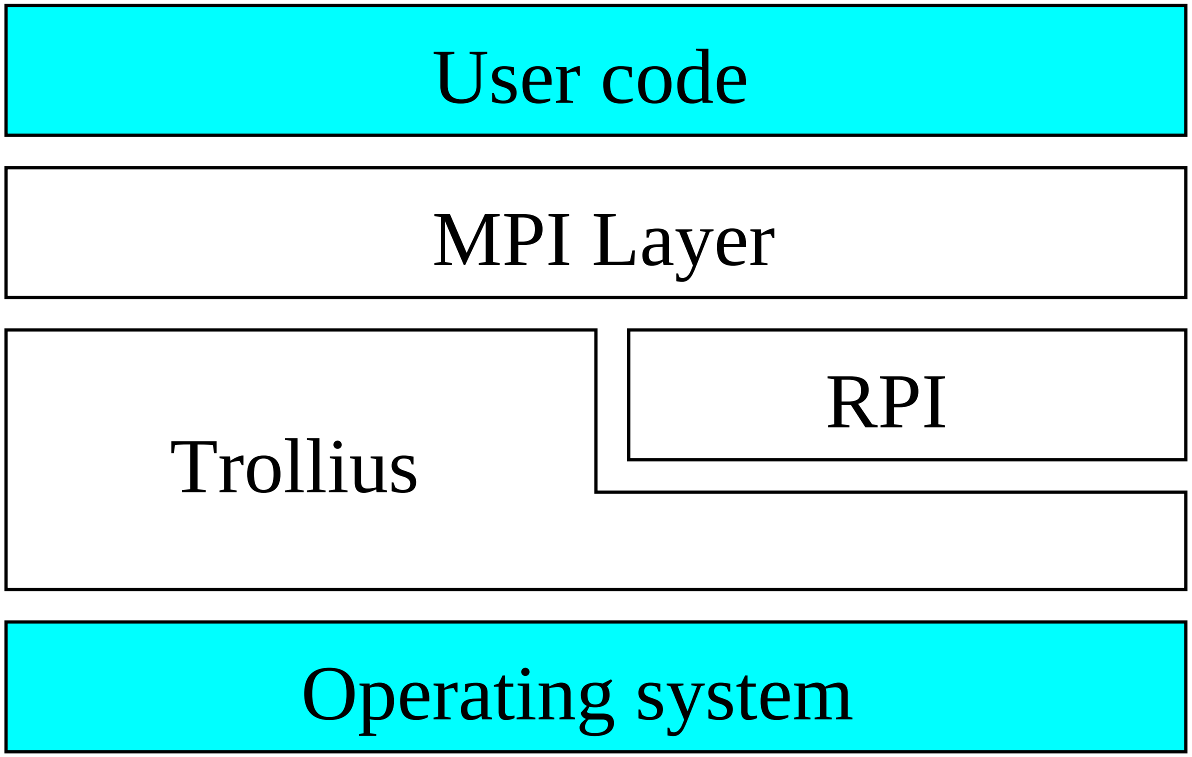
- Introduction
- IMPI Overview
- LAM/MPI Overview
- IMPI Implementation in LAM/MPI
- Results
- Conclusions / Future Work

LAM/MPI Overview

- Multiple original LAM developers were on the IMPI Steering Committee; the design of IMPI is similar to that of LAM/MPI
- Originally written at OSC as part of the Trollius project, now developed and maintained at Notre Dame
  - Full MPI-1.2 implementation, much of MPI-2
  - Multi-protocol shared memory / network protocols
  - Persistent daemon-based run-time environment, used for process control and out-of-band messaging of meta data

Code Structure

- Divided into three main parts: MPI layer, Request Progression Interface (RPI), and the Trollius core



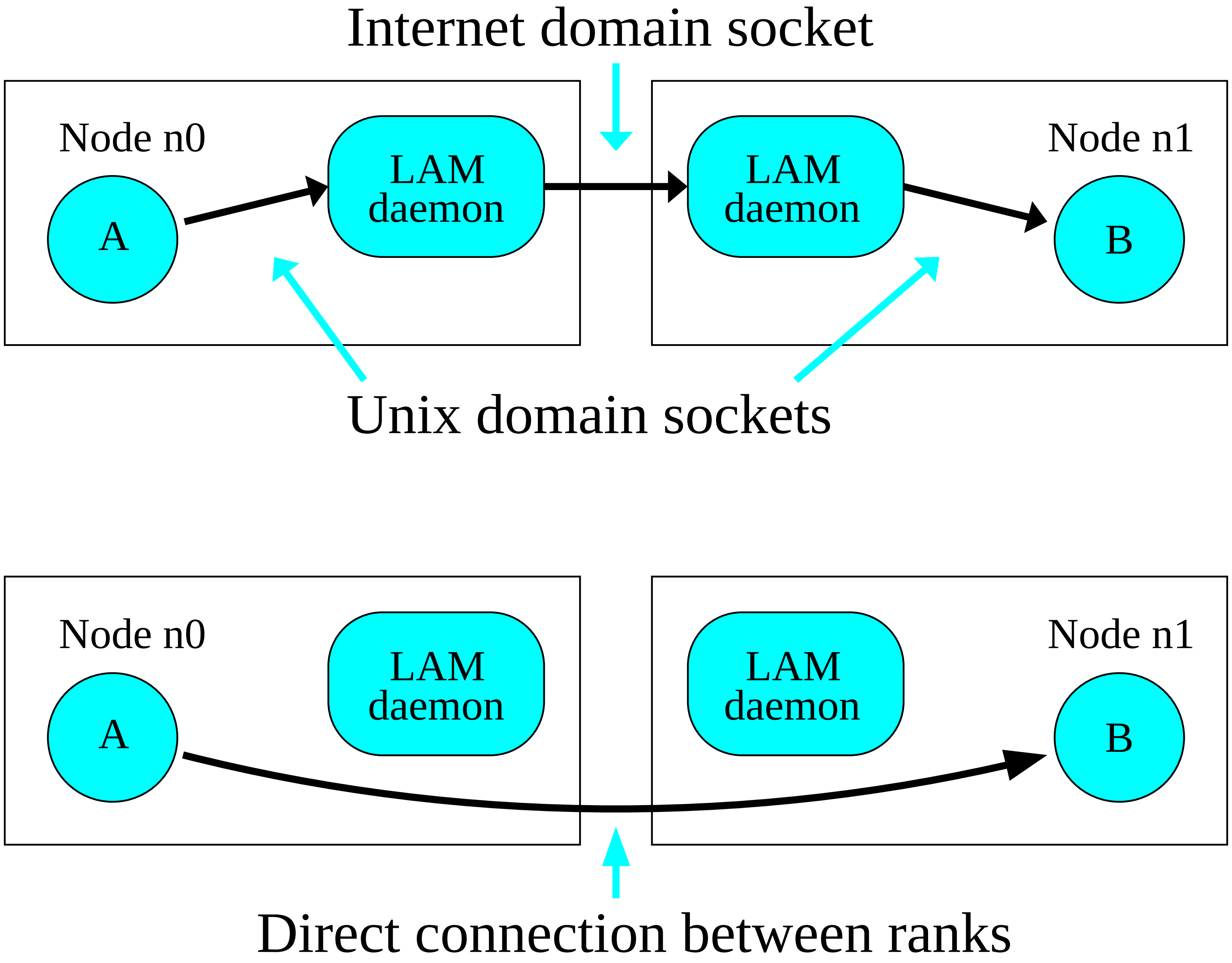
Code Structure

- MPI Layer
  - Every communication is a *request* (i.e., **MPI_Request**)
  - Creates and maintains communication queues of requests
  - e.g, **MPI_SEND** generates a request and places it on the appropriate queue
- Trollius Core
  - Provides a backbone for most services, including the LAM daemons
  - Contains most of the “kitchen sink” functions for LAM/MPI

Request Progression Interface (RPI)

- Responsible for all aspects of communication; the RPI progresses the queues created in the MPI layer
- Rigidly defined layer – has a published API
- Two classifications of RPIs: **lamd** and **c2c**
  - **lamd**: Daemons based – slower, but more monitoring and debugging capabilities available
  - **c2c**: Client-to-client – faster, no extra hops

lamd and c2c RPI Diagrams



c2c RPIs

- LAM currently includes three **c2c** RPIs:

  1. **tcp**: Uses internet domain sockets between all ranks
  2. **sysv**: Same as **tcp**, but uses shared memory to communicate between ranks on the same node; SYSV semaphores used to lock memory
  3. **usysv**: Same as **sysv**, but spin locks used to lock memory

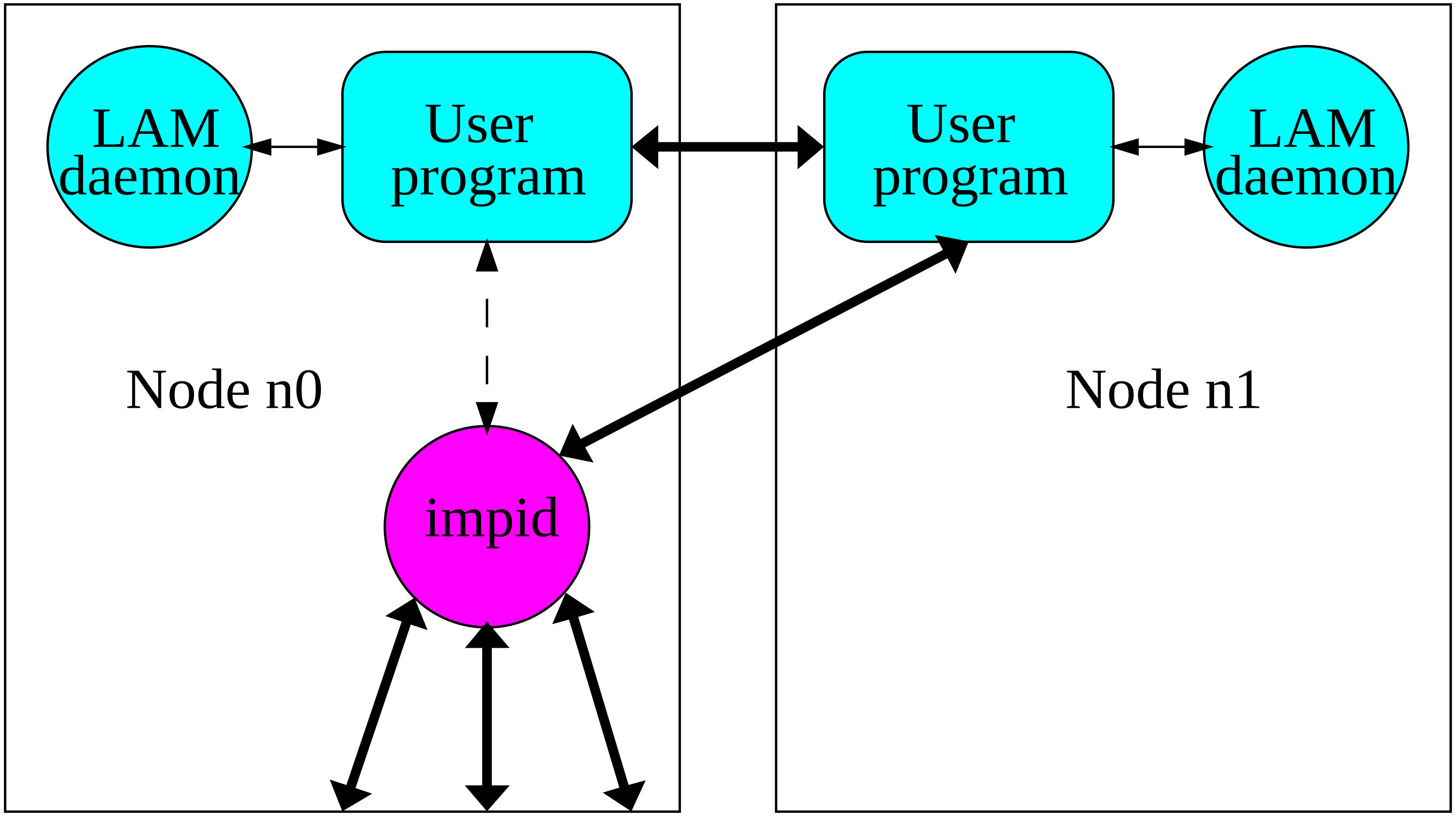
Overview

- Introduction
- IMPI Overview
- LAM/MPI Overview
- IMPI Implementation in LAM/MPI
- Results
- Conclusions / Future Work

Overall Goals

1. The IMPI client and host will be implemented in a separate daemon, (the **impid**)
2. There will only be one **impid** per IMPI job; only one IMPI host will be supported
3. The **impid** must be transparent to the user; it will automatically be started, die gracefully when the program finishes, and be able to be killed by the **lamclean** command on error
4. The **impid** must be able to use any of the four existing RPIs

How the impid Fits In



Connections to other IMPI hosts

IMPI Job Startup

- Start the IMPI server somewhere
- **mpirun -np 4 -client** $<$**server-ip**$>$ $<$**server-rank**$>$ **a.out**
- **impid** is an "almost MPI" process
  - Is started via **MPI_COMM_SPAWN** during **a.out**'s **MPI_INIT**
  - Invokes **MPI_INIT** / **MPI_FINALIZE** itself
  - Uses MPI calls for the majority of message passing between local LAM ranks
  - Receives $<$**server-ip**$>$ and $<$**server-rank**$>$ from **mpirun**

The impid

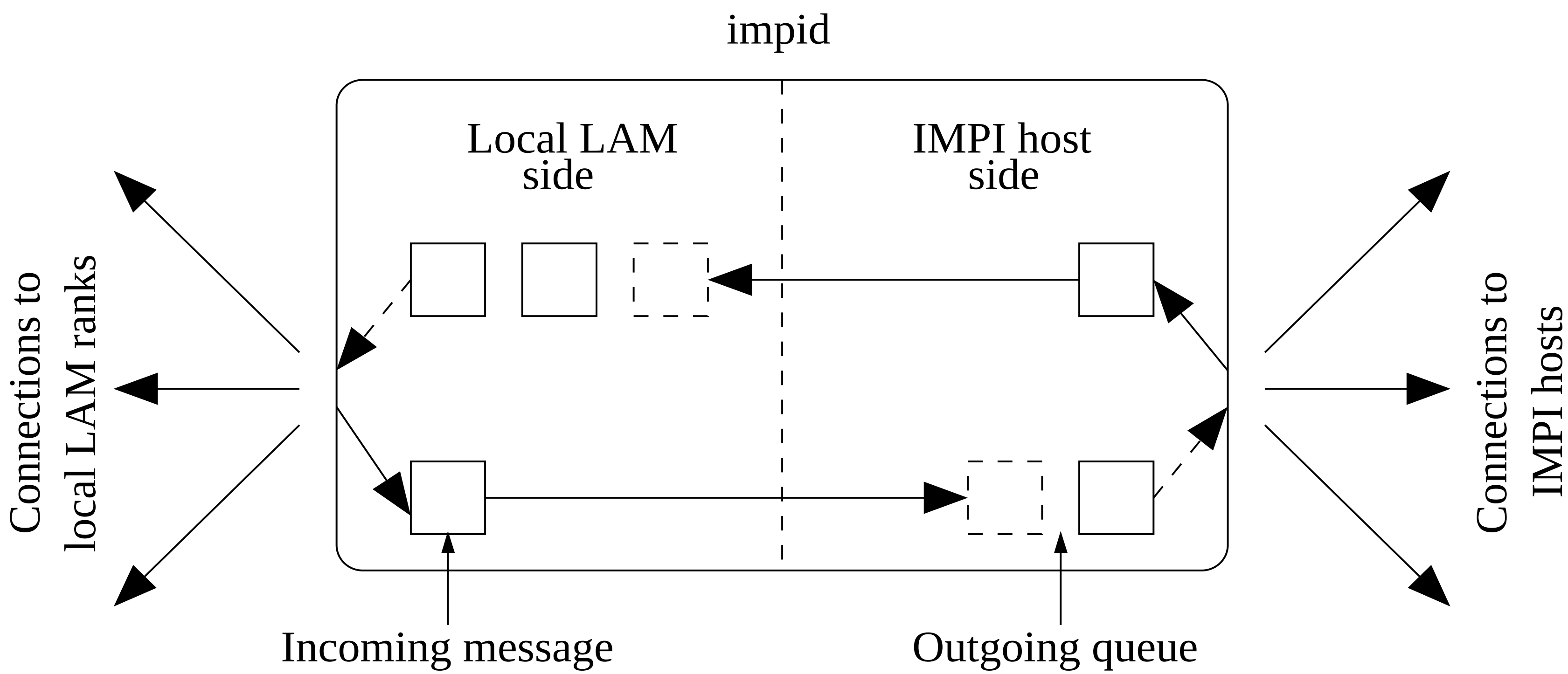
- Provides some degree of asynchronous communication since it is a separate process
  - For example, **MPI_SEND** to a remote proc can return long before the message is actually received
  - Can allow for overlap of communication / computation
  - Especially important over high latency links
- Communicates with local LAM ranks using the current RPI
- Allows for multiple simultaneous IMPI jobs within a single LAM

impid Design

- **impid** is split in half: one side communicates with the local LAM ranks, the other side communicates with the other IMPI hosts
- Could not use threads; spinning necessary
  - Use blocking **poll(2)** system call to check for messages on host side, and **MPI_TESTANY** to check for completion on local LAM side
  - Cannot block while communicating; persistent / non-blocking modes used to communicate with local LAM ranks
- Host side must obey flow control rules

impid Design (continued)

- Messages that enter one side generally exit the other
- Queuing system used on host side because of long message protocol and flow control

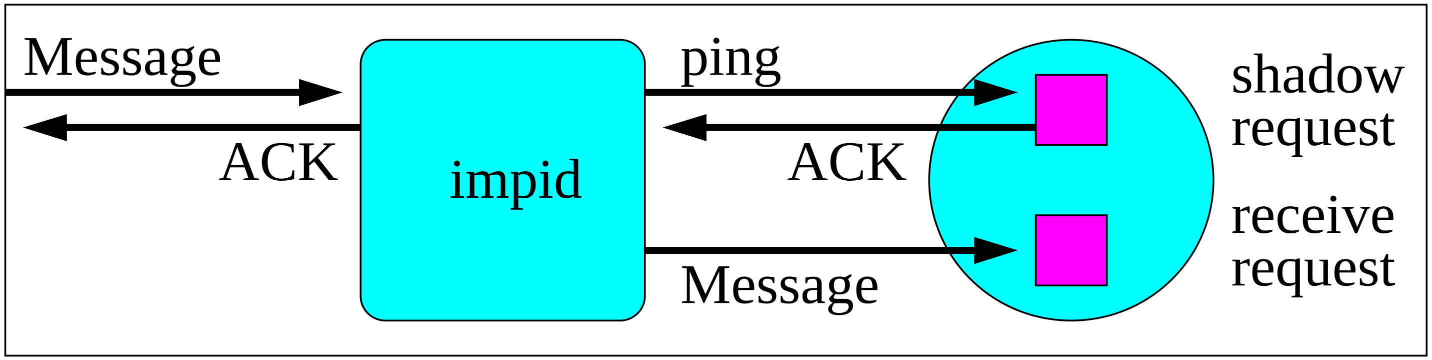


General Operation

- Sending messages
  - If a message is destined for a proc on a remote host, its request is re-routed in the MPI layer to the **impid**
  - The **impid** then sends the message to the remote host according to the long / short protocols
- Receiving messages
  - Similarly, if receiving from a remote proc, MPI layer redirects the request to receive from the **impid**
  - Complications arise for synchronous mode sends

Shadow Requests

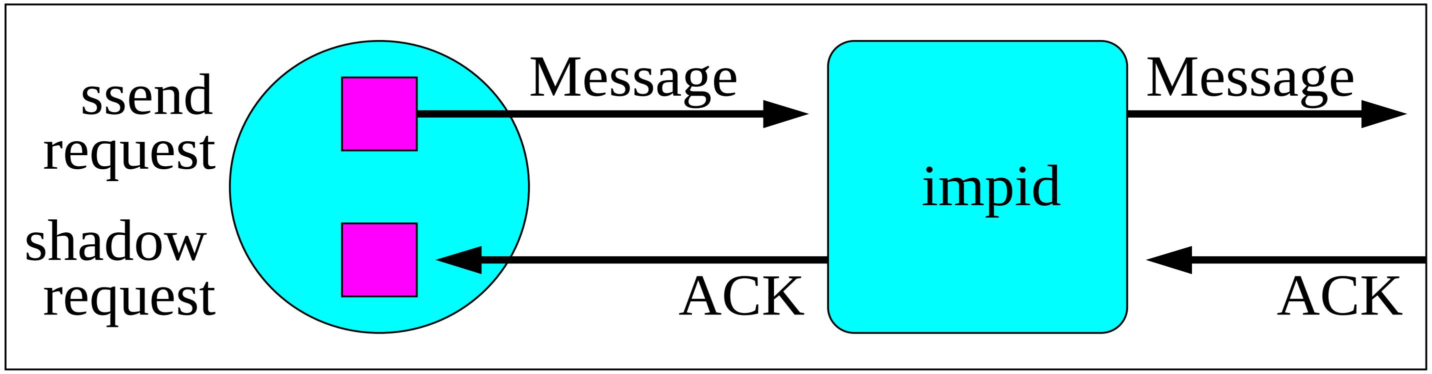
- No way for **MPI_RECV** to know if incoming message will be synchronous before message is received; must send ACK if so
  - Post *two* receives; second receive is a “shadow”



- If message was not synchronous, shadow receive is canceled after message is received

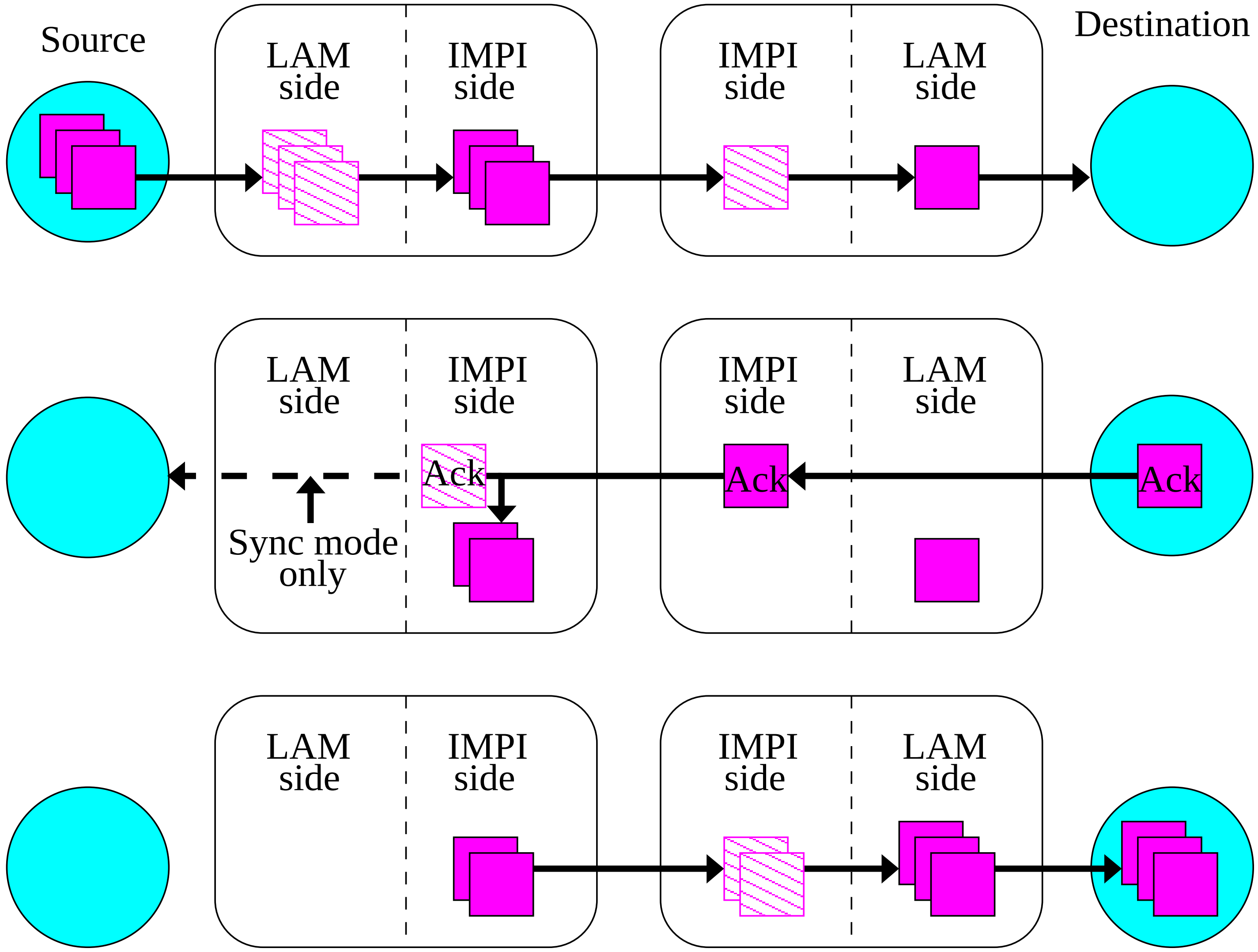
Synchronous Sends

- Similarly, synchronous sends must post a shadow receive



- Shadow requests are new for IMPI – added significant complexity in **MPI_WAIT** and **MPI_TEST**

Example: Sending a Long Message



Limitations of Implementation

- Except for **MPI_BARRIER**, no MPI collectives implemented
  - Data-passing collectives not hard to implement
  - Communicator constructor / destructor collectives will probably require a redesign
- **MPI_CANCEL** is not implemented for sends of messages

Overview

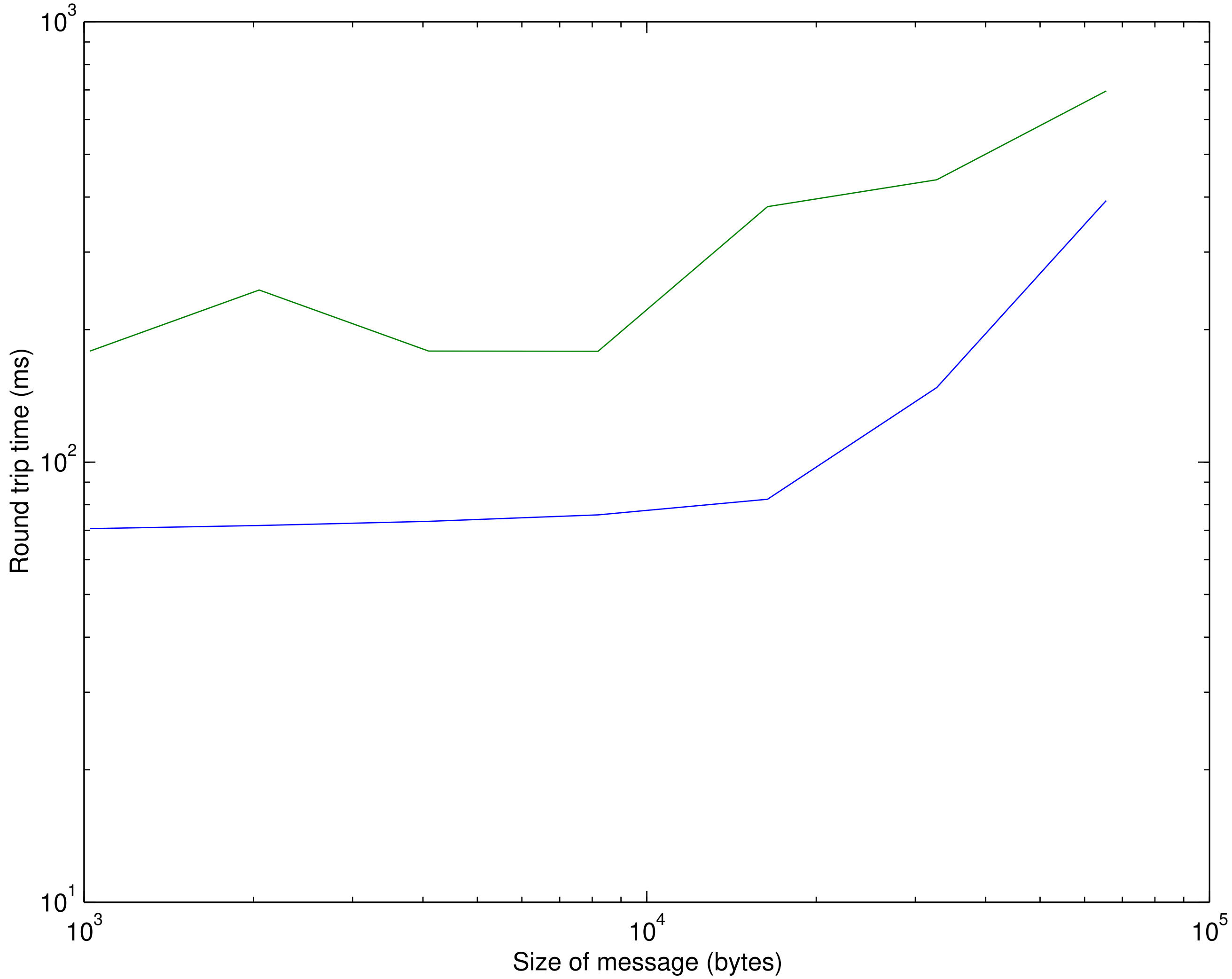
- Introduction
- IMPI Overview
- LAM/MPI Overview
- IMPI Implementation in LAM/MPI
- Results
- Conclusions / Future Work

Experimental Setup

- Two unloaded 400Mhz Pentium-II machines, Linux 2.2.12
  - One located at Notre Dame (Indiana, USA)
  - Other located at Lawrence Berkeley Labs (California, USA)
- Each on local 100Mbps switched networks
- Aggregate bandwidth at time of test was 2.3Mbps
- Standard ping-pong test (messages of 100 bytes to 1Mbyte)
  - **Test one**: two LAMs connected via IMPI
  - **Test two**: one LAM spanning both machines

Round Trip Times (ms) vs. Message Size

Green: times for IMPI (2 LAMs)   Blue: times for one LAM



Is This Bad?

- It's not good, but it's not bad
- IMPI communication is *supposed* to be slow (by design)
- The value is that it works and is not horrendous
  - Can connect highly tuned MPI implementations
  - Well written MPI codes can utilize high-latency links properly, and overlap communication and computation
  - Need at least one vendor-tuned IMPI to see true benefits

Overview

- Introduction
- IMPI Overview
- LAM/MPI Overview
- IMPI Implementation in LAM/MPI
- Results
- Conclusions / Future Work

Conclusions

- IMPI works
- Now we need support for IMPI in other MPI implementations
- Future work for LAM:
  - Finish data-passing and constructor collectives
  - Optimize collectives over distributed links
  - Restructure LAM/MPI for generic multi-protocols
  - Thread support

http://www.mpi.nd.edu/lam/