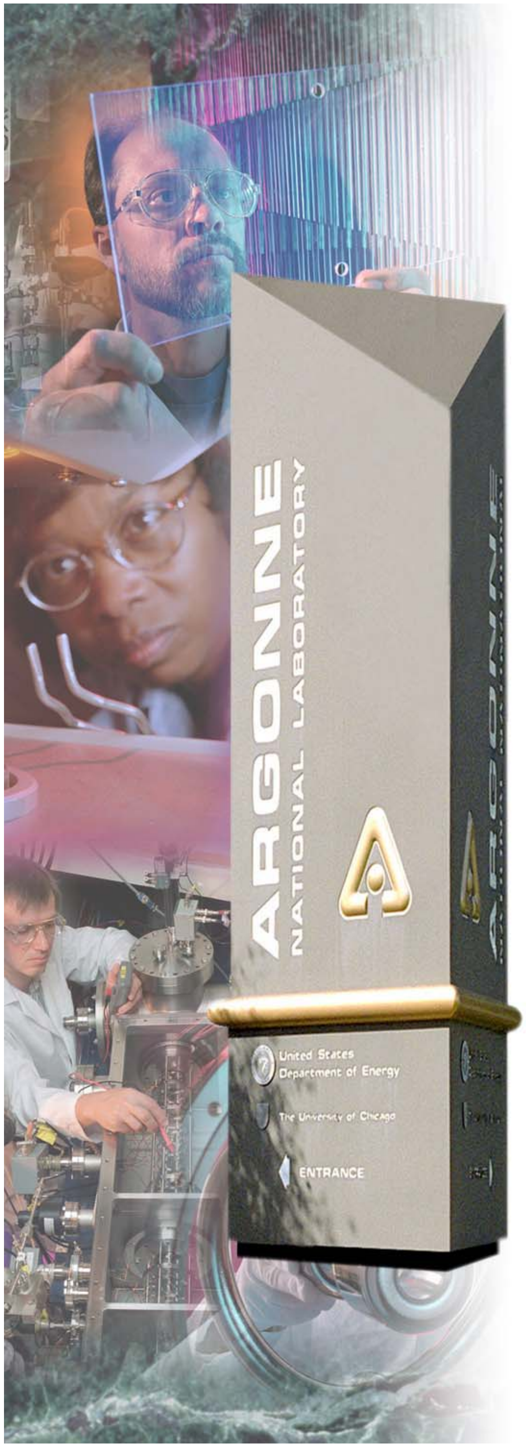


Towards a Productive MPI Environment

William Gropp
www.mcs.anl.gov/~gropp

Argonne National Laboratory

Office of Science
U.S. Department of Energy

A U.S. Department of Energy
Office of Science Laboratory
Operated by The University of Chicago

Outline

- **Building, testing, distributing MPI-based applications**
  - MPI API vs. MPI ABI
  - Partial Steps
- **Enhancing and customizing the MPI environment**
  - MPICH2 components
- **Improving the programmability of MPI**
  - Enhanced error detection, reporting
  - Exploiting the Profiling interface
  - Introducing higher-level abstractions
    - *Higher Level Libraries*
    - *Source-to-source transformations*

Working with Multiple MPI Implementations

- **MPI ABI Revisited**
  - History:
    - *Building Library Components That Can Use Any MPI Implementation at Euro PVMMPI 2002*
    - *Greg Lindahl's The Case for an MPI ABI*
    - *Subsequent comments on the Beowulf list and elsewhere*
  - Obvious Issues
    - *Mpi.h contents*
    - *Library linkage*
    - *Non-opaque objects*
  - Less Obvious Issues
    - *Process Startup*
    - *Shared libraries*
    - *Scalability*

The Problem

- **Libraries and ISVs want to use MPI**
  - Which MPI? MPICH? OpenMPI? LAM/MPI? Vendor MPI? MPICH-G2? <your-favorite-MPI-here>?
  - Could build under all versions
    - *Must install and test each version*
  - Most libraries distributed as object files are built for a single MPI
- **Applications want to use libraries**
  - What if the libraries need different MPI implementations?

Building a Generic mpi.h

- **To create a common mpi.h, the following parts of the MPI definition must be addressed:**
  - Compile-time values
    - *E.g., MPI_ERR_TRUNCATE, MPI_ANY_SOURCE*
  - Compile-time values used in declarations
    - *E.g., MPI_MAX_ERROR_STRING*
  - Init-time constants
    - *E.g., MPI_INT, MPI_COMM_WORLD*
  - Opaque objects
    - *E.g., MPI_Request, MPI_Comm*
  - Defined Pointers
    - *E.g., MPI_BOTTOM, MPI_STATUS_IGNORE*
  - Defined Objects
    - *E.g., MPI_Status*
- **For systems with sizeof(int) == sizeof(void*), most of these can be handled by carefully making values extern ints rather then #define or enums. The exception is MPI_Status:**

Defined Objects

- **MPI_Status**
  - Defined as a struct, but not all fields (and hence size) nor the placement of the fields defined
- **Replace interface with access methods (close to the C++ interface)**
  - One possible approach: define an API for handling arrays of status (needed by Wait/Test some/all)
  - int GMPI_Status_get_tag(MPI_Status *s, int idx)
    MPI_Status *GMPI_Status_create(int n)
    void GMPI_Status_free(MPI_Status *p)
  - This API permits macro implementation for specific MPI implementations, e.g.,
    - *#define GMPI_Status_get_tag(s, idx) s[idx].MPI_TAG*

Using Generic MPI

- **How easy is it to use a generic MPI based on these ideas?**

```
# Independent of MPI implementation (generic mpi.h in
# /usr/local/gmpi)
%   cc -c myprog.c -I/usr/local/gmpi/include
%   cc -c mylib.c -I/usr/local/gmpi/include
%   ar cr libmymlib.a mylib.o
%   ranlib libmylib.a
# For MPICH
%   /usr/local/mpich/bin/mpicc -o myprog myprog.o -lmylib \
       -L/usr/local/gmpi/lib -lgmpitompich
# For LAM/MPI
%   /usr/local/lammpi/bin/mpicc -o myprog myprog.o -lmylib \
       -L/usr/local/gmpi/lib -lgmpitolam
```

Compile with gmpi

Link with specific MPI implementation

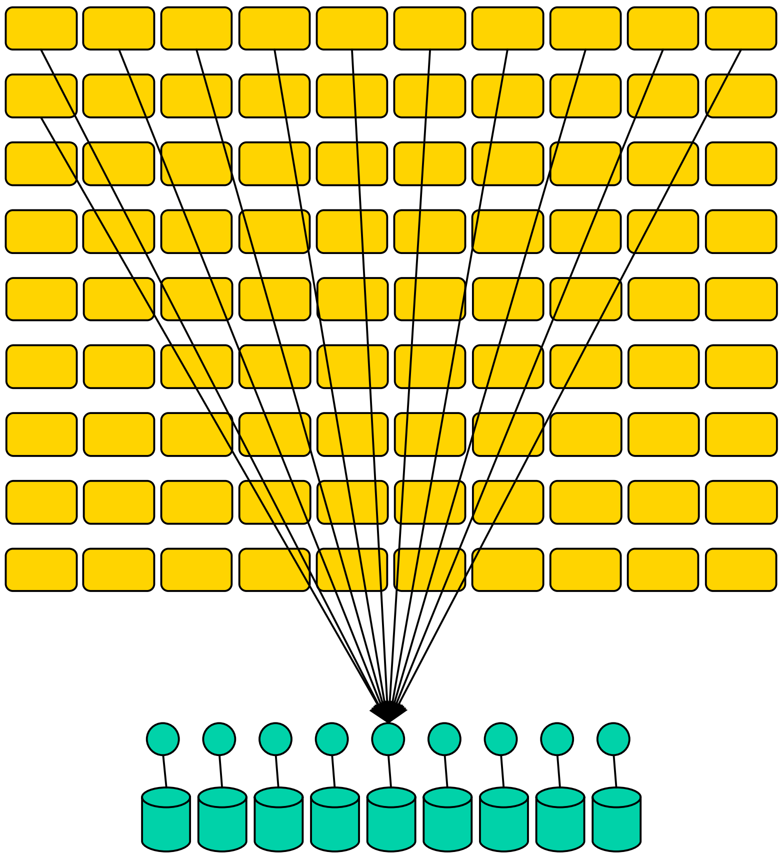
Handling 64-Bit Systems

- **64bit systems**
  - Ints usually 32 bits, pointers 64 bits
  - Handles are no longer the same length in all implementations
- **Solutions:**
  - Separate based on handle length
    - *Reduces overall number of versions*
    - *gmpi32.h and gmpi64.h ?*
  - Use methods to create and delete handles
    - *Forces more significant changes to existing C programs*
    - *A generic C++ binding could handle this*

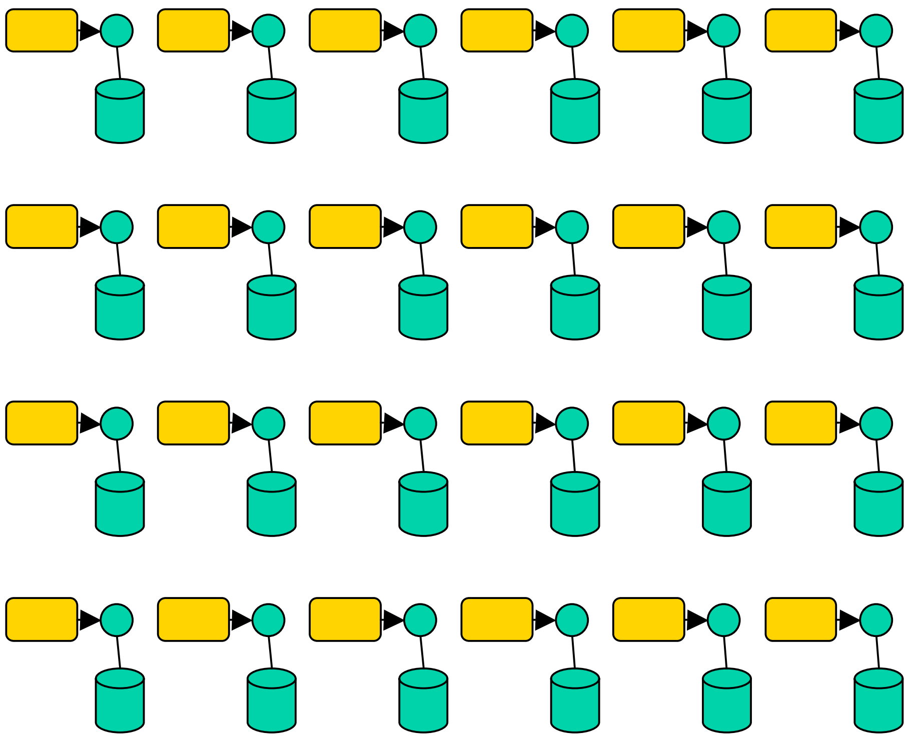
Why Wasn't this enough?

- **Construction of shim programs and header files (e.g., to replace #define MPI_INT … with const int MPI_INT=(int)…)**
  - Partially automated as part of 2002 paper, but process is fragile and requires manual inspection
- **Changes MPI**
  - Programs must be rewritten to handle MPI_Status
- **Greg Lindahl pointed out missing features in model**
  - Does not address starting and running MPI jobs
  - Many libraries and applications wish to use shared libraries instead of static libraries
    - *Real potential for problems with mismatched shared libraries. This problem is so common that it is called "DLL Hell". Most (all?) suggestions to date are very fragile*
    - *One piece of the solution may be "collective system calls", part of one of the DOE FastOS projects*
  - The 64-bit "problem" isn't going to go away
- **Let's look at starting and running MPI jobs**
  - Beginning with MPI_Init…

DLL Hell Illustrated



Common Shared Library
System suffers a "system call storm"



Distributed Shared Library
(All of these are identical, right?)

MPI Process Startup

- **MPI-2 specified mpiexec**
  - Scripts can now use mpiexec –n 64 a.out
- **Some features still missing, as Lindahl points out**
  - Standard I/O: Mpiexec a.out < foo >bar
  - But the same problem exists with queuing systems
    - *Try qsub a.out < foo > bar*
  - Command line arguments, environment variables are not guaranteed
- **Some things undefined**
  - Process state before MPI_Init or after MPI_Finalize
    - *How many processes? Values of environment variables?*
- **But a major problem is that mpiexec and a particular MPI implementation (and even choice of communication device) have been closely coupled**

Process Manager Interface

- **The process manager and the interface between the process manager and the MPI job can be a separately standardized component**
- **In standardizing the functions and the interface, scalability is a key issue.**
  - Starting with the "BNR" interface in MPICH-1, MPICH2 uses a *scalable* process management interface (PMI) defined by:
    - *An Applications Programmer Interface (API) (set of routines called by MPICH2)*
    - *A wire protocol for a particular implementation of the API*
    - *All process management functions (startup, spawn, connect) are handled through this interface*
  - Note that the interface is *scalable*. It is easy to make mistakes here.
- **In MPICH2, a single executable may be run with different process managers**
  - Configure –with-pm=mpd:gforker … ; make ; make install
  - mpicc –o myprog myprog.c
  - mpiexec –n 10 myprog
  - mpiexec.gforker –n 10 myprog

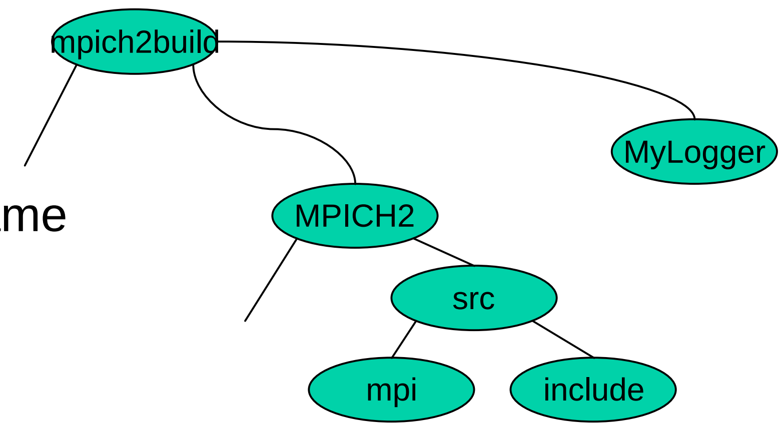
Customizing the MPI implementation

- **Well-defined component interfaces provide a good way to customize MPI implementations**
  - Process management interface makes it easy to connect to other process management styles
    - *I'm looking for people interested in adding new mpiexec implementations, including bproc and remote shell (ssh) versions*
- **Other interfaces**
  - Performance information
  - *MPICH2 provides configure-time hook with*

    Configure –with-logging=/abspathname

    …

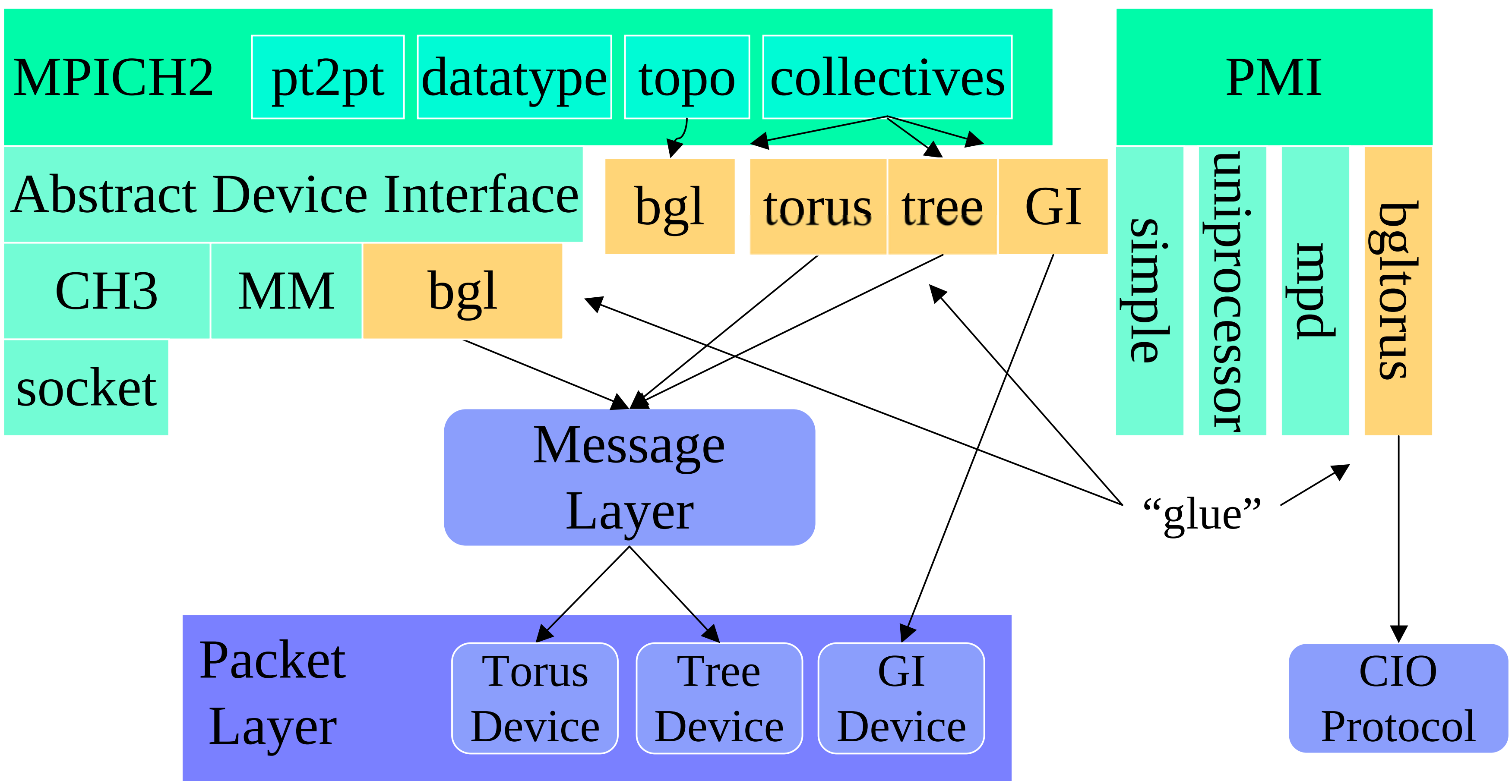
    where /abspathname is a directory containing an implementation of the MPICH2 logging interface and Implementation of MPI operations



Implementation of MPI Operations

- **Collectives**
  - Since early in MPICH1, MPICH1 offered an interface allowing replacement of each collective operation on a per-communicator basis
    - *Based on code provided by Jim Cownie for the Meiko*
  - MPICH2 redesigned this interface to minimize code footprint:
    - *Each collective defines a general yet high-quality implementation of the collective*
    - *Each communicator maintains a pointer to a table of function for collectives*
      - A null pointer for this table => use default
      - A null pointer for this function in table => use default
    - *Allows customization based on communicator (Meiko use comm world and dups of comm world), including application-specific (e.g., restricted implementations in communicators used within a library)*
- **Topology**
  - Similar approach used to interface with information about process layout
- **Both of these are exploited by the IBM BG/L implementation of MPI**

IBM BlueGene/L MPI Software Architecture



(slide based on one provided by IBM)

State of MPICH2

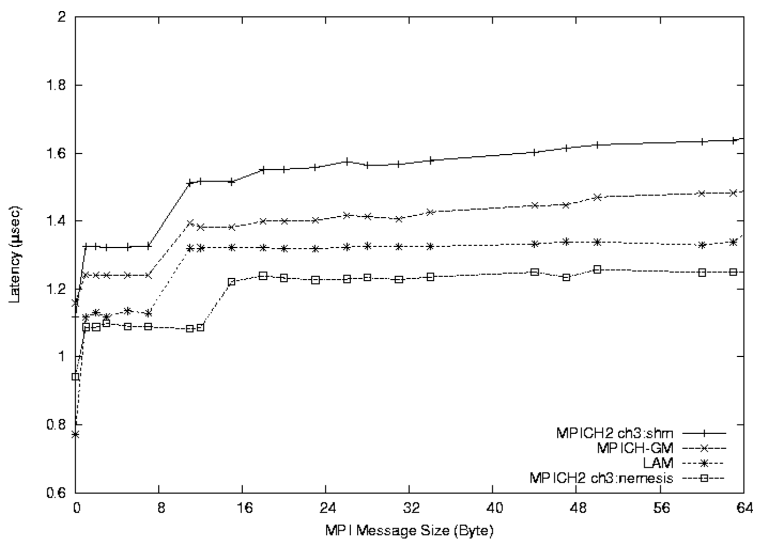
- **All new (from scratch) implementation of MPI-2 (and MPI-1)**
  - Not encumbered by limitations of old MPICH1 code
- **Version 1.0 of MPICH2 released at SC2004**
- **Current version 1.0.2p1**
- **Supports all of MPI-2 except** `external32` **data representation**
- **Includes beta-level support for MPI_THREAD_MULTIPLE**
- **Next release before SC2005**
- **Robust implementations for TCP and shared memory**
- **Experimental implementations for InfiniBand and GASNet**
- **Basis for many implementations, including**
  - IBM BG/L, Cray XT3, Intel, Microsoft, Myricom,

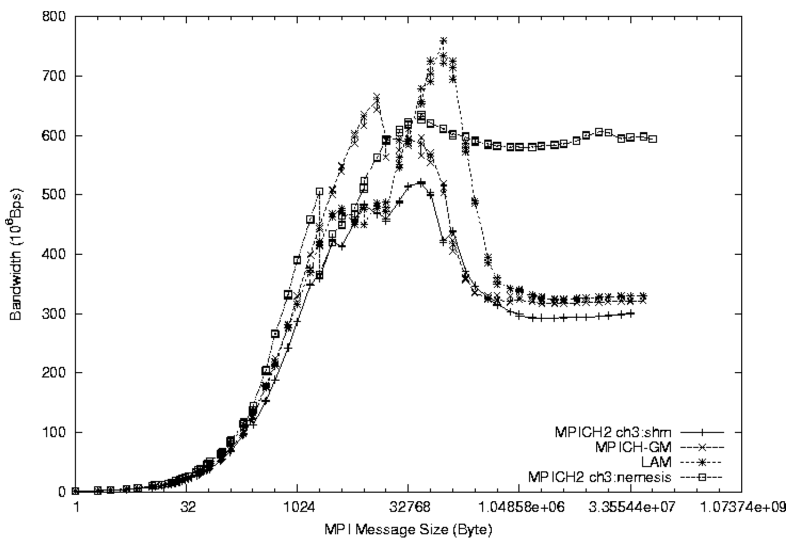
Plans for the Next Year

- **Full MPI-2 compliance**
  - Add external32 data representation
- **Thread safety**
  - Thread *safety* is relatively easy; safety *and performance* is not
  - Explore how to do this efficiently with fine-grained locks, rather than locking the entire progress engine on entry
- **Collective communication**
  - Currently optimized for flat network topologies
  - Recent work this summer looked at multiple concurrent communication channels (available on IBM BG/L)
  - Optimize for hierarchical network topologies, such as clusters of SMPs and the TeraGrid
- **One-sided communication**
  - Synchronization functions already optimized, but data transfer uses two-sided semantics at lowest levels
  - Extend low-level APIs and implementation to allow true RDMA
- **Replacement Basic Communication Device**

New Communication Core

- **Provide an infrastructure to answer basic questions about scaling MPI implementations**
  - What is the overhead of MPI?
    - *Typically, one measures some MPI implementation, then claims that is the overhead of MPI; confuses an implementation with a specification*
- **Our goal: Develop a fast, well-instrumented and analyzed communication core**
  - Answer questions about overhead, cost of MPI
    - *E.g., ~480 ns of latency below is mandatory cache miss cost*
  - Provide higher-performance, lower-latency open MPI





New MPICH2 Communication Device

- **Current work is developing a “channel” for the ch3 device**
- **Key Features**
  - Shared memory is a special-case method
    - *Lock-free queues*
      - Low latency
      - Extremely scalable
  - Multi-method
    - *New networks are easy to add*
    - *4 required functions*
      - init, finalize, send, poll
    - *Optional functions for RMA and collectives for enhanced performance*
      - Follows standard MPICH approach that allows easier initial ports, followed by performance tuning (the ch3 device fell off the true path for a while ☺)
- **See the *Designing a Common Communication System* on Wednesday for more on high-performance communication device issues**

Lock-Free Queues

- **Low latency**
  - No locks
    - *Uses compare-and-swap and swap atomic instructions*
  - Simple implementation
    - *Enqueue: 6 instructions, 1 L2 cache miss*
    - *Dequeue: 11 instructions, 1-2 L2 cache misses*
  - Progress engine has only one queue to poll
- **Extremely scalable**
  - Each process needs two queues regardless of the number of processes
    - *Recv queue*
    - *Free queue*
  - Progress engine has only one queue to poll
- **Same queue mechanism is used for networks**
  - Messages received from networks are enqueued on the recv queue

Improvements to MPICH2 I/O

- **MPI-IO Enhancements in ROMIO**
  - MPI-2 one-sided (RMA) operations allow us to operate on remote memory regions without remote process intervention
  - Atomic mode and shared file pointers can be implemented using MPI-2 capabilities
  - Talks on both Tuesday (4B, 5B)
- **MPI-IO Interface Extensions**
  - Extensions are needed for name space traversal
    - *Equivalent to readdir in POSIX*
  - Opportunity to think about forthcoming storage name space organizations (e.g., database-like, others)

Improving the MPI Development Environment

- **Implementations should have robust, complete error reporting**
- **Errors should be *instance specific***
  - Which would you rather have:
    - *Invalid rank*
    - *Invalid rank of 5, must be between 0 and 4*
  - (You probably want a traceback too — a standard ABI for acquiring a traceback would be a tremendous asset for any OS or language)
  - MPICH2 exploits the difference between an error *code* and an error class
    - *Each error code includes a reference to the error class and a string that contains the instance-specific data. A hash is used to address issues of limited storage for errors and "stale" error codes*
    - *Never worse than an error class*
    - *It's a good thing that the error codes were not fixed by the MPI Forum.*
- **Missplaced objects (e.g., a tag value where a communicator is expected) should be detected**
- **For development, an implementation should pass at least the local error detection tests in the Intel MPI-1 test suite**
- **Non-local tests (e.g., send/receive types and consistency of parameters to collective calls) are harder**

Exploiting the Profiling Interface

- **All MPI routines may be accessed through MPI_Xxx or PMPI_Xxx**
- **Allows customized development and debugging modifications**
- **Simple example: Write an MPI_Send that calls PMPI_Issend/PMPI_Test to check for dependencies on message buffering**
- **Many performance debugging tools, for example**
  - MPE tools within MPICH and MPICH2
  - FPMPI (summary tool) www.mcs.anl.gov/fpmpi
- **Correctness debugging tools**
  - E.g., detect errors in arguments to collective operations (4B)
- **Another place to simplify life for users**
  - MPICH2 provides –profile=name argument for complication scripts
  - If libname.a exists, use that (in the correct place in the link order)
  - If name.conf exists, read that for more complex linking instructions
  - Environment variables allow specification of profiling without changing existing build or make scripts
- **All of these do require a detailed understanding of the MPI standard**

Improving Parallel Programming

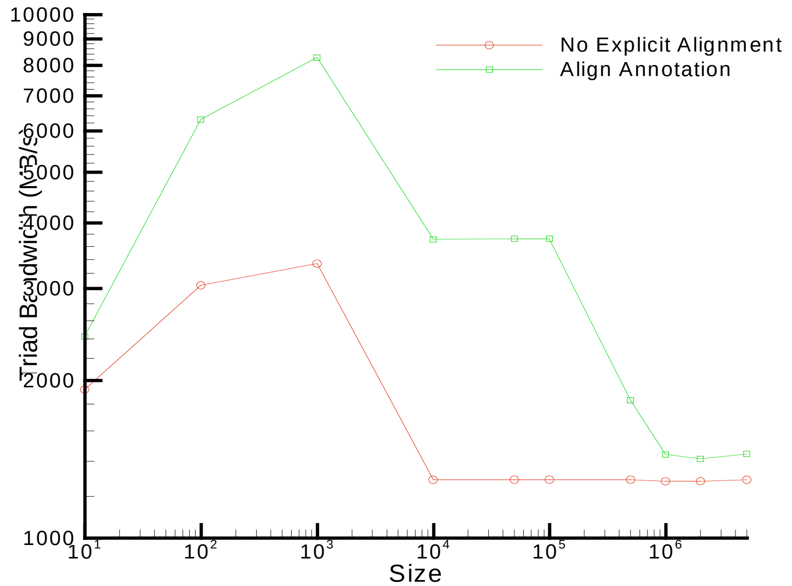
- **How can we make the programming of real applications easier?**
- **Problems with the Shared Memory Model**
  - Performance costs
    - *False sharing, ensuring atomic updates, scalability, dependence on the compiler to recognize and optimize collective operations*
  - "Action at a distance"
  - Loss of determinism
  - Performance goals may still require user-managed data decomposition
- **Problems with the Message-Passing Model**
  - Performance costs of a library (no compile-time optimizations)
    - *Latency costs force larger "grain size", exacerbating the decomposition problem*
  - "Action at a distance"
    - *Matching sends and receives*
    - *Remote memory access*
  - User's responsibility for data decomposition

Program Annotation Tools

- **Use annotations to augment existing languages**
  - Not a new approach; used in HPF, OpenMP, others
  - Aspect-oriented programming another example
  - Some applications already use this approach for performance portability
    - *WRF weather code*
- **Annotations do have limitations**
  - Fits best when most of the code is independent of the parts affected by the annotations
  - Limits optimizations that are available to approaches that augment the language (e.g., telescoping languages)
- **We are looking at a standard framework for annotating source code that can invoke “third party” transformation tools**
  - Creates an “annotation ecosystem” to spur evolution of improved tools
  - Provides a uniform approach for applications

Annotation Example on BG/L

- **Use of second FPU requires that data be aligned on 16-byte boundary**
- **Source code requires non-portable pseudo-functions (__alignx(16,var))**
- **By using simple, comment-based annotations, speeds up triad by 2x while maintaining portability and correctness**



Annotations example: stream triad.c

```
void triad(double *a, double *b, d
{
  int i;
  double ss = 1.2;
  /* --Align;;var:a,b,c;; */
  for (i=0; i<n; i++)
    a[i] = b[i] + ss*c[i];
 /* --end Align */
}
```

```
void triad(double *a, double *b, double *c, int n)
{
#pragma disjoint (*c,*a,*b)
  int i;
  double ss = 1.2;
  /* --Align;;var:a,b,c;; */
 if ( ((int)(a) | (int)(b) | (int)(c)) & 0xf == 0) {
   __align(16,a)
   __alignx(16,a);
   __alignx(16,b);
   __alignx(16,c);
   for (_i=0; _i<=n; _i++) {
       a[_i]=b[_i]+ss*c[_i];
    }
 }
 else {
    for (_i=0;_i<=n;_i++) {
        a[_i]=b[_i]+ss*c[_i];
 }
  /* --end Align */
}
```

Simple annotation example: stream triad.c

Size	No Annotations (MB/s)	Annotations (MB/s)
10	1920.00	2424.24
100	3037.97	6299.21
1000	3341.22	8275.86
10000	1290.81	3717.88
50000	1291.52	3725.48
100000	1291.77	3727.21
500000	1291.81	1830.89
1000000	1282.12	1442.17
2000000	1282.92	1415.52
5000000	1290.81	1446.48

>2X

Alternative example: A Regular Mesh Sweep

- **C$AAS Declare Mesh(nx,ny); stencil width 1; a**
  double precision a(nx,ny)
  C$AAE
  …
  C$AA Init a
  …
  C$AAS LoopOver a
  do i=1, nx
  do j=1, ny
  a(i,j) = a(i-1,j-1) + ….
  enddo
  enddo
  C$AAE LoopOver nolast

Require user provide information on halo (easy for users, hard for compiler)

Regular mesh, distributed across all processes

Hook for initialization

Usual grid sweep, written in "global" coordinates

Generated (Readable!) Code

```
C$AAS Declare Mesh(nx,ny); stencil width 1; a; md5=0xccde2
double precision locala(0:lnx+1,0:lny+1)
C$AAE
…
C$AAS Init a; md5=00
call AAMeshInit(locala,nx,ny,lnx,lny)
C$AAE
…
C$AAS LoopOver a; md5=0xcfd234
call AAMeshExchange(locala,lnx,lny)
do i=1,lnx
   do j=1,lny
      locala(i,j) = locala(i-1,j-1)+…
   enddo
enddo
C$AAE LoopOver nolast
```

Or allocate dynamically

Detect user changes to block

Or explicit MPI-1 or MPI-2 calls

Or split into Morton ordered loops

A Real Example

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

int main()
{
   int i;
   /* --DA1d-declare
     var:mesh;;type:double;;halo:1;; */
   double mesh[100];
   /* --end DA1d-declare */

   /* --DA1d-alloc var:mesh;;gsize:100;; */
   /* --end DA1d-alloc */

   /* --DA1d-sweep var:mesh;;block:mesh[@] = sin
   for (i=0; i<100; i++) {
                     mesh[i] = sin( i / 100.0 );
   }
   /* --end DA1d-sweep */

   /* --DA1d-sweep var:mesh;;block:<<within>>;;inc
   for (i=1; i<99; i++) {
                     mesh[i] = 0.5 * (mesh[i-1] + mesh[i
   }
   /* --end DA1d-sweep */

   /* --DA1d-serialize var:mesh;;block:<<within>>;;
   for (i=0; i<100; i++) {
     printf( "mesh[%d] %f\n", i, mesh[i] );
   }
   /* --end DA1d-serialize */
}
```

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

int main()
{
   int i;
   /* --DA1d-declare      var:mesh;;type:double;;halo:1;; */
   double* _lmesh=0;
   int _lsizemesh = 0, _gsizemesh = 0, _gleftmesh;
   int _crankmesh = -1, _csizemesh = 0;
   /* --end DA1d-declare */

   /* --DA1d-alloc var:mesh;;gsize:100;; */
   MPI_Comm_rank( MPI_COMM_WORLD, &_crankmesh );
   MPI_Comm_size( MPI_COMM_WORLD, &_csizemesh );
   _lsizemesh = 100/_csizemesh + 2 * 1;
   _gleftmesh = _crankmesh * _lsizemesh;
   _lmesh = (double *)malloc( sizeof(double) * _lsizemesh );
   /* --end DA1d-alloc */

   /* --DA1d-sweep var:mesh;;block:mesh[@] = sin( @g/100.0 );; */
   for (i=0; i<=_lsizemesh; i++) _lmesh[i] = sin( (i+_gleftmesh)/100.0 );
   /* --end DA1d-sweep */

   /* --DA1d-sweep var:mesh;;block:<<within>>;;index:i;; */
   for (i=0; i<=_lsizemesh; i++)  {
                     _lmesh[i] = 0.5 * (_lmesh[i-1] + _lmesh[i+1]);
   }
   /* --end DA1d-sweep */

   /* --DA1d-serialize var:mesh;;block:<<within>>;; */
   if (_crankmesh > 0) {
      MPI_Recv(MPI_BOTTOM,0,MPI_BYTE,_crankmesh-1, 5678,MPI_COMM_WORLD,
MPI_STATUS_IGNORE);}
   for (i=0; i<=_lsizemesh; i++)  {
     printf( "mesh[%d] %f\n", i+_gleftmesh, _lmesh[i] );
   }
   if (_crankmesh+1 < _csizemesh) {
     MPI_Send(MPI_BOTTOM,0,MPI_BYTE,_crankmesh+1,5678,
MPI_COMM_WORLD);}
   /* --end DA1d-serialize */
```

Conclusions

- **MPI has served us well, but**
  - Need to address API/ABI issues in MPI-3
  - Scalability and performance are still two of the great strengths of MPI
- **Some issues can be addressed by embracing components**
  - Standardized components are easiest
  - (Almost) any component allows a “shim” implementation
- **MPICH2 continues to explore the implementation space**
  - Long history of components, focus on development aids
- **Finally, MPI often called the “assembly language of parallel programming”. Given a portable, high-performance assembly language, where are the high-level languages?**
  - Annotations provide on easy, application or domain-specific path