Debug MPI Applications

Ruini Xue

School of Computer Science and Engineering

University of Electronic Science and Technology of China

2016

Outline

- Background
- Subgroup Reproducible Replay
- Challenges
- Evaluation
- Conclusion

Background

Difficulties of Debugging MPI

- Debugging parallel codes can be incredibly difficult, particularly as codes scale upwards.
  - Large number of processes
  - Distributed over different nodes
  - Long running time
  - Complicated communication pattern
  - All bugs in *sequential* programs
  - And new bugs due to *parallelism*
    - *Non-determinism, dead-lock…*

- **MPI debugging is a hard experience!**

Related Work

- Static Checking
- Runtime Detection
- Postmortem Trace Analysis
- Model Checking
- Data Mining
- Replay-based Methods
  - Data-replay: Huge Log size
  - Order-replay: Huge replay resource
- …
- **Debugging with replay is the future.** (The Parallel Computing Landscape: A view From Berkeley 2.0)

Static Checking

- Compile-time source checking
  - Limited to semantic errors
  - E.g., MPI-CHECK(CCPE'03)

Runtime detection

- Cyclic debugging
  - ☹ Scalability, reproducibility, portability
  - E.g., *printf*, Totalview, DDT, PGDBG, Net-dbx
- PMPI Intercepting
  - ☹ only MPI APIs and predefined errors
  - E.g., Umpire(SC'00), MARMOT(Parco'03), Retrospect(EuroPVM/MPI'07)
- Debug version of MPI Library
  - ☹ Transparency, reproducibility
  - E.g., NEC mpi, MPICH,

Postmortem trace analysis

- Random message payload on non-deterministic point
  - E.g., HASE'05, IPDPS'07
- Check the trace for predefined errors
  - E.g., Intel Message Checker(SE-HPCS'05)

Replay-based Methods

- Data-driven replay
  - No code modification
    - PDT(TechRept'95):PMPI
    - Dieter(EuroPVM/MPI'01):MPICH-1.2.1
    - Retrospect (EuroPVM/MPI'07): OpenMPI core
  - MPI prototypes changed
    - Christian(PhDThesis'00): wrap MPI calls
- Order-driven replay
  - Happen-before relationship
  - NOPE (ACPC'99): only p2p APIs

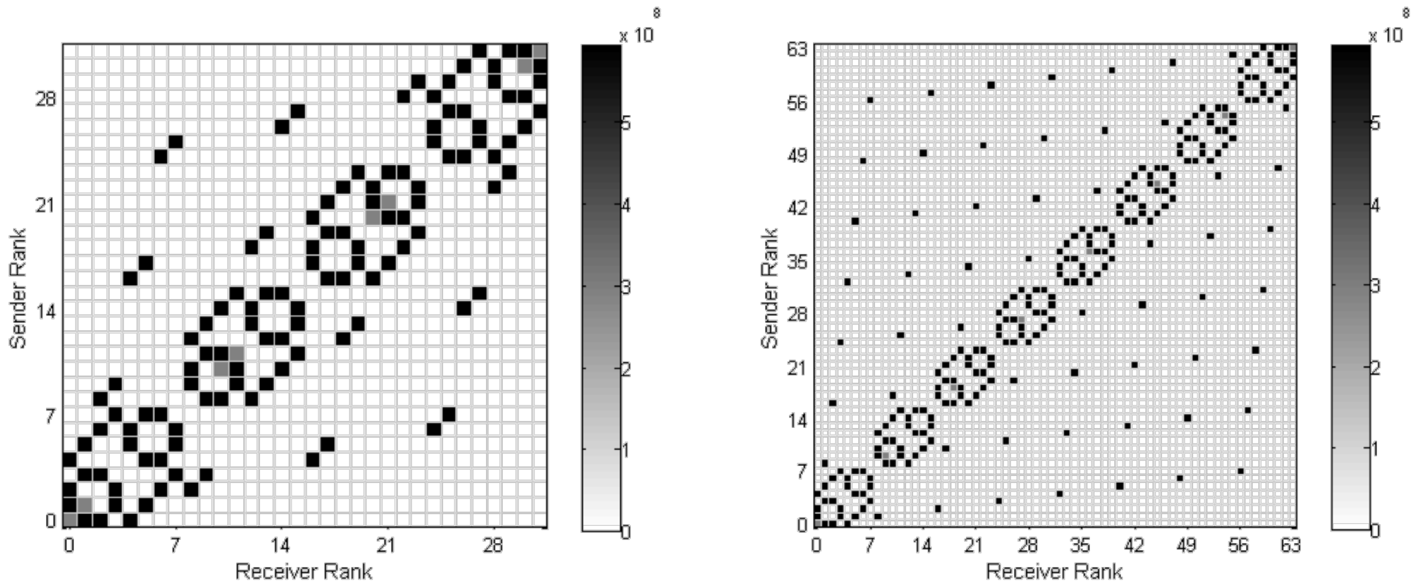
Miscellaneous

- Model-Checking
  - Spin-MPI (VMCAI'07, EuroPVM/MPI'07)

Subgroup Reproducible Replay

Inspiration: Communication Locality

- NPB CG.C.32 & CG.C.64



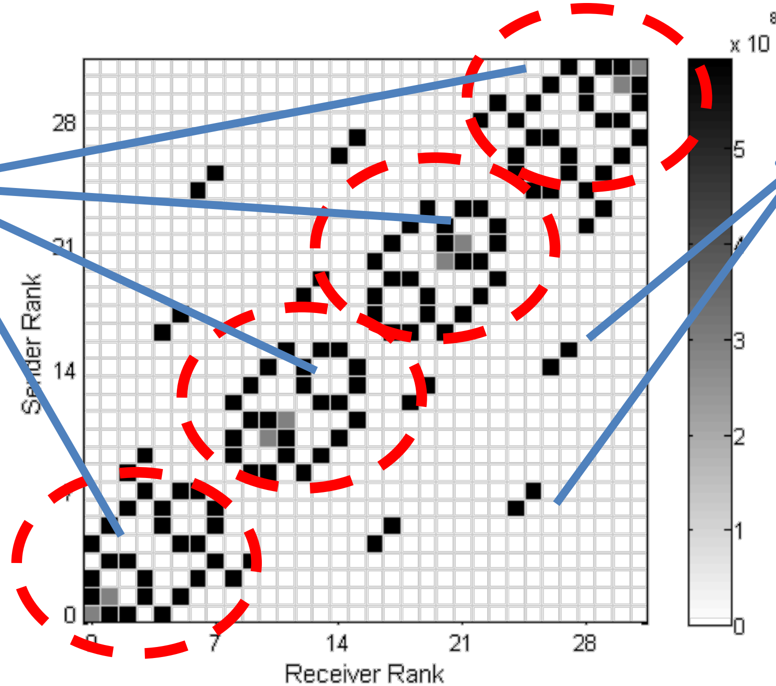
- Intra-group communication: ≈77%

Observation: Communication Locality

77% traffic

Order Replay

Small log size



CG.C.32

23% traffic

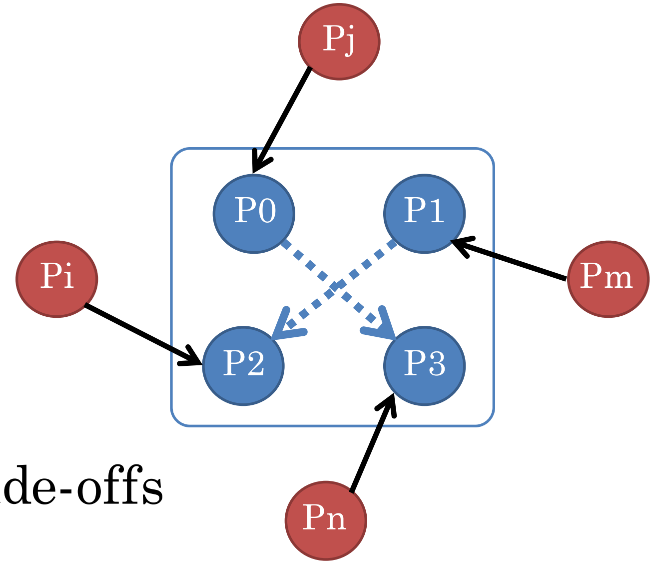
Data Replay

Can replay individual groups

Process grouping can be exploited for **a good balance** between record and replay cost (log size vs replay capability with limited resource)

Subgroup Reproducible Replay

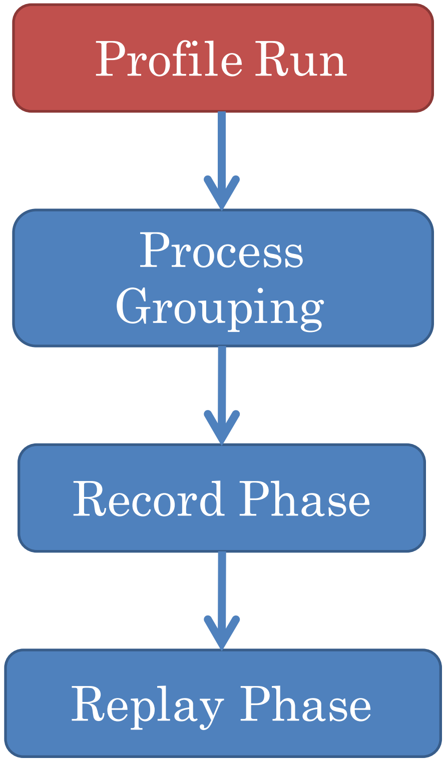
- Combine data- and order-replay
  - Data-replay: inter-group communication
  - Order-replay: intra-group communication



- Generalization of them
  - A balance for record/replay trade-offs

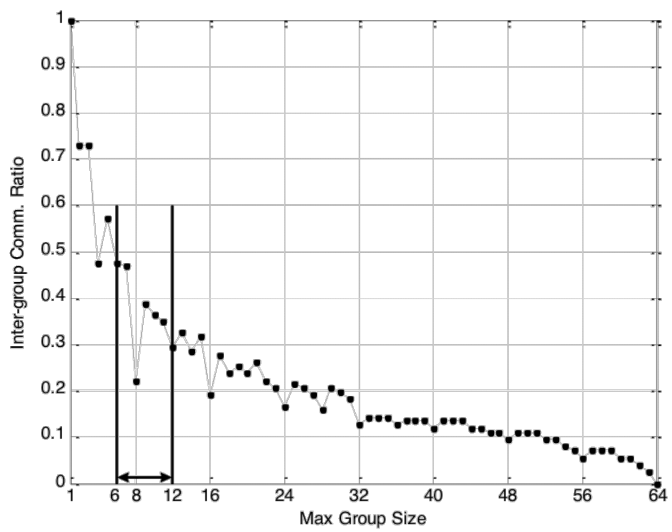
Profile Run

- Collect communication trace (optional)
  - Profiling
- Expert Knowledge

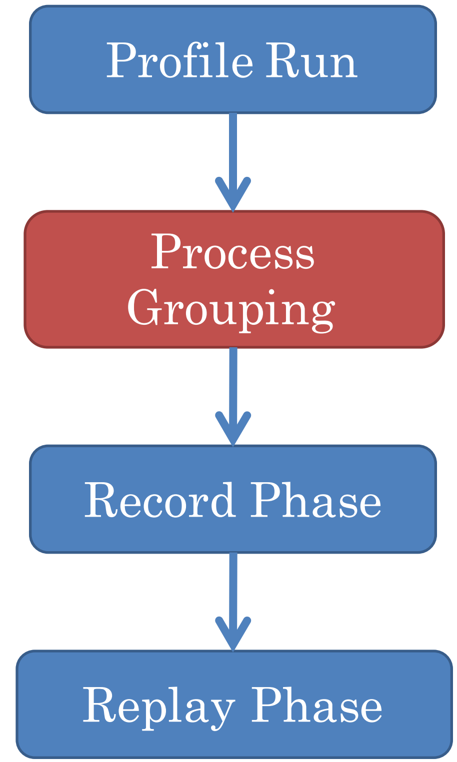


Process Grouping

- Communication Graph Partition
  - Exploit comm. Locality
    - Process --- vertex
    - Comm. Volume --- edge weight



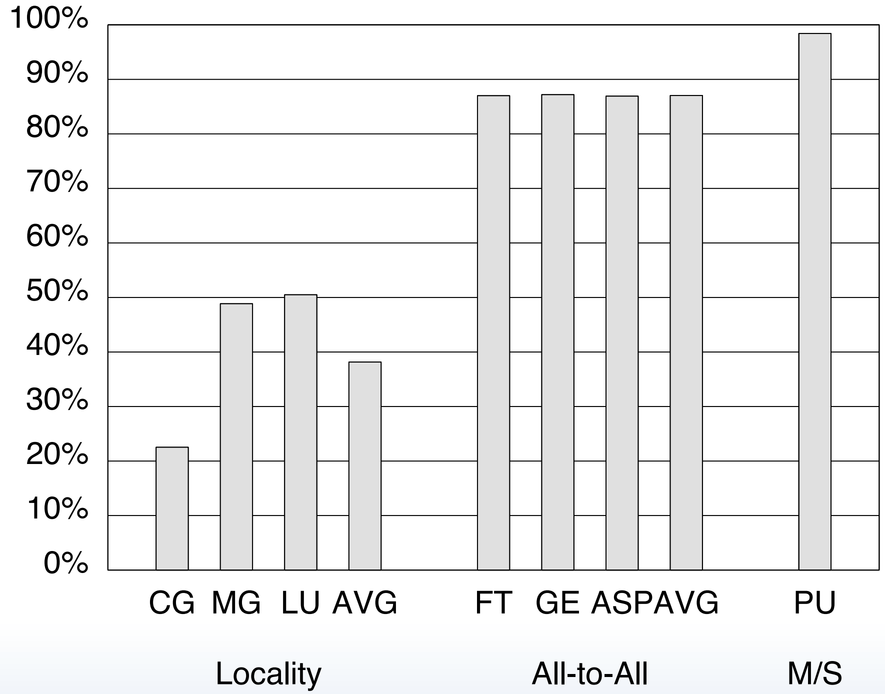
- Assign as your will

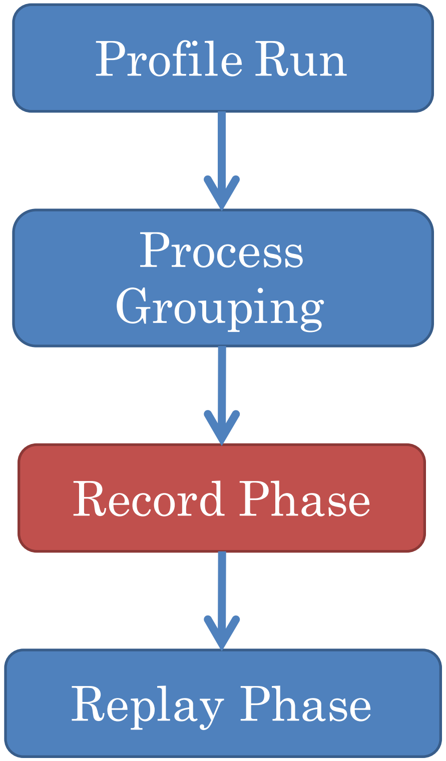


Record Phase

- Record different groups independently

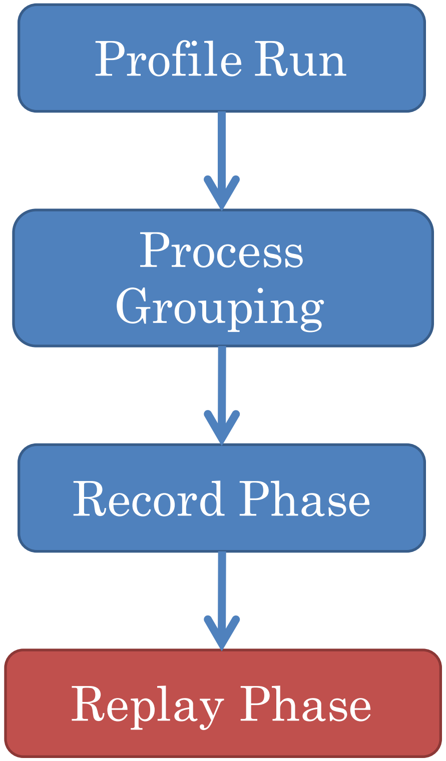
- Log size decreases dramatically.





Replay

- Replay processes of one group altogether
- Feed-back
  - Inter-group comm. & OS calls
- Reproduce
  - Intra-group comm.
  - *faked* messages



So far, so easy ☺

Designs are cheap, but implementations are expensive.

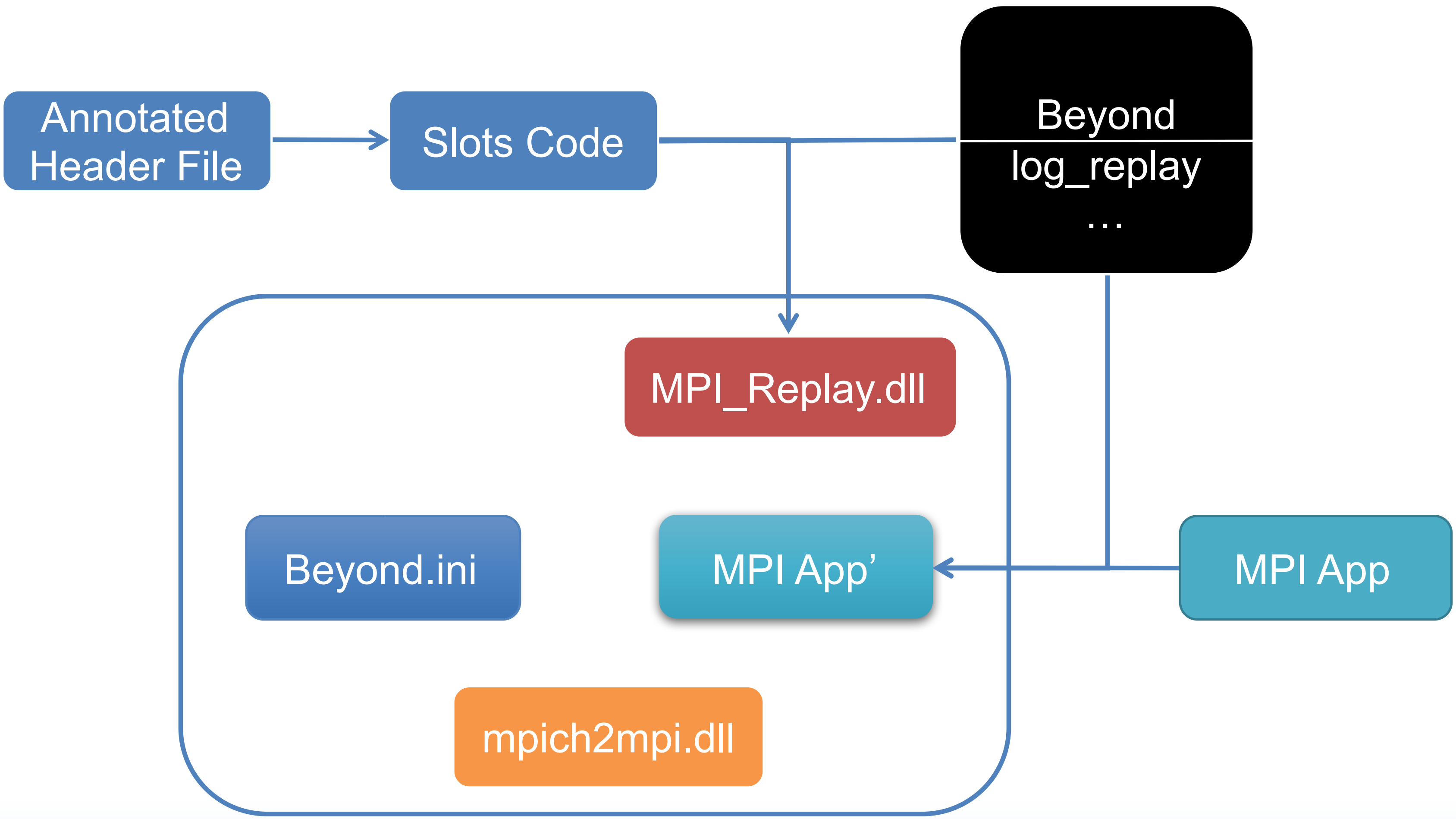
—— *How (and How Not) to Write a Good Systems Paper, 1983*

Challenges

Basic Challenges

- Interposition
  - Dynamic instrumentation
- Wrapper codes
  - Automatic generate = Annotation + Template
  - 18,000 C++ codes for MPI calls
- Faked replay
  - Replayer: a minimal MPI run-time
- MPI Standard
  - Dive in to details, too many tricks

How does it work?



Annotation

```
int
MPI_Init (
    [in] int *argc,
    [in] char ***argv
    );
```

```
int
MPI_Get_processor_name (
    [out, ecap(*len), esize(*len+1), force] char *name,
    [out] int *len
    );
```

```
int
MPI_Recv (
    [out, bsize("get_MPI_general_buf_size(datatype, count)"), force] void* buf,
    [in] int count,
    [in] MPI_Datatype datatype,
    [in] int src,
    [in] int tag,
    [in] MPI_Comm comm,
    [out, opt(MPI_STATUS_IGNORE)] MPI_Status* status
    );
```

Generated Record/Replay Codes (1)

```
/* int[cc(__cdecl), module(mpich2mpi)]
MPI_Init (
    [in] int * argc,
    [in] char * * * argv);*/

BEGIN_SIGEX_MIXIN(MPI_Init, hw_log_MPI_Init)
        START_PROFILE_WITH_NAME("hw_log_MPI_Init")
        g_LogProducer.set_identifier(g_papi_info_MPI_Init->api_sig);
        g_LogProducer << sx_last_ret;
        g_LogProducer << log::end;
END_SIGEX_MIXIN

BEGIN_SIGEX_SLOT(MPI_Init, hw_replay_MPI_Init)
        START_PROFILE_WITH_NAME("hw_replay_MPI_Init")
        g_LogConsumer.chk_identifier(g_papi_info_MPI_Init->api_sig);
        int sx_last_ret;
        g_LogConsumer >> sx_last_ret;
        g_LogConsumer >> log::end;
        return sx_last_ret;
END_SIGEX_SLOT
```

Generated Record/Replay Codes (2)

```
/*int[cc(__cdecl), module(mpich2mpi)]
MPI_Comm_size ( [in] MPI_Comm comm,    [out] int * size );*/

BEGIN_SIGEX_MIXIN(MPI_Comm_size, hw_log_MPI_Comm_size)
        START_PROFILE_WITH_NAME("hw_log_MPI_Comm_size")
        g_LogProducer.set_identifier(g_papi_info_MPI_Comm_size->api_sig);
        g_LogProducer << sx_last_ret;
        g_LogProducer << (uint32)sizeof(deref<int *>::type);
        g_LogProducer.write((LPVOID)size, (uint32)sizeof(deref<int *>::type));
        g_LogProducer << log::end;
END_SIGEX_MIXIN

BEGIN_SIGEX_SLOT(MPI_Comm_size, hw_replay_MPI_Comm_size)
        START_PROFILE_WITH_NAME("hw_replay_MPI_Comm_size")
        g_LogConsumer.chk_identifier(g_papi_info_MPI_Comm_size->api_sig);
        int sx_last_ret;
        g_LogConsumer >> sx_last_ret;
        uint32 nParamBufLength;
        g_LogConsumer >> nParamBufLength;
        g_LogConsumer.read((LPVOID)size, nParamBufLength);
        g_LogConsumer >> log::end;
        return sx_last_ret;
END_SIGEX_SLOT
```

More Challenges: Non-determinisms

- System calls
  - `gettimeofday, random, socket...`
  - data-replay works
- MPI calls
  - Inter-group Messages
  - Wildcard Receives
  - Waits, Test and Probes
  - Collective Operations
- How to record and replay them faithfully?

Inter-group Messages

- Are they non-deterministic?
- Record
  - Ignore out messages
  - Record in messages
    - include the order, if necessary
- Replay
  - Ignore out messages
  - Feed-back in messages from log

Wildcard Receives

- MPI_ANY_SOURCE, MPI_ANY_TAG
  - Record the real values, replace them during replay
- MPI_STATUS[ES]_IGNORE
  - Replace the parameter with a new allocated memory
  - Record the status data
    - Real values for `source` and `tag`
- Non-blocking Receives
  - Same solution but in MPI_Wait/MPI_Test

Waits, Tests, and Probes

- MPI_*some, *_*any (* = Wait | Test)
- Record
  - Map table: request <-> buffer info
  - Record returned request, and corresponding buffer
- Replay
  - Map table too
  - Feed-back (or receive) buffer
- Nothing special for Probes
  - following operation does the real work

Collective Operations

- How to replay if some processes are out of the group?
  - Record phase: save the membership
  - Replay phase: emulated with p2p communication
  - MPI_Bcast() example

```
/* MPI_Bcast() replay code */
load MPI_Bcast rank_list from log
if (I am root) { /* for data sender */
  foreach rank in rank_list:
    if (rank is in replay group)
      send message to rank
} else { /* for data receiver */
  if (root is in group)
    recv message from root
  else
    load message from log
}
```

Even More

- MPI_BOTTOM
  - Seldom used in real applications!
- MPI_IN_PLACE
  - Optimization for some collective operations (MPI_Scatter)
  - Just mark as: opt(MPI_IN_PLACE)
- MPI_Start()
  - Why no MPI_Stop()?
  - We can cleanup the rubbish, but not in a decent manner.
- MPI_Alloc_mem()/MPI_Free_mem()
  - The MPI forum committee is, eh…, dreaming.
- MPI_Cancel()
  - Actually as normal

1. What's the length of the buffer?

- MPI_Recv(buf, count, MPI_FLOAT, src, tag, &req)
  - count * sizeof(MPI_FLOAT) =? count * sizeof(FLOAT)

```
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status )

typedef int MPI_Datatype
```

- sizeof() sucks for even built-in/primitive datatypes!

Even worse

- Derived datatype
  - Typemap = {($type_0$, $disp_0$),…, ($type_{n-1}$, $disp_{n-1}$)}
  - Extent(Typemap) = ub(Typemap) - lb(Typemap)
  - Size: remove the spaces in extent
  - Built-in datatypes are indeed derived datatypes!

```
#define MPI_INT ((MPI_Datatype)6)

MPI_INT = {(int, 0)}
```

Why extent and size?

- Insights on MPI_Send and MPI_Recv
  - sendbuf, recvbuf = count * extent(datatype) + lb(datatype)
  - network data bytes = count * size(datatype)
  - straightforward optimization

- *How do we know the upper bound of a type?*
  - int MPI_Type_get_extent(MPI_Datatype dt, int* lb, int* ext)

Determine the buffer size

- Solution
  - save count * extent(datatype) + lb(datatype)
  - usually: lb = 0

```
int
MPI_Recv (
   [out, bsize("get_MPI_buf_size(datatype, count)"), force] void* buf,
   ……
   [out, opt(MPI_STATUS_IGNORE)] MPI_Status* status
   );
```

2. What does the returned length mean?

- The length of the '*out*' buffer is indicated by another '*out*' parameter

```
int  MPI_Get_processor_name (char *name, int *len);
```

- Old method:

```
int
MPI_Get_processor_name (
    [out, ecap(*len), esize(*len + 1)] char *name,
    [out] int *len
    );
```

- Error !

Why is it wrong?

- log slot

```
BEGIN_SIGEX_MIXIN(MPI_Get_processor_name, hw_log_MPI_Get_processor_name)
  START_PROFILE_WITH_NAME("hw_log_MPI_Get_processor_name")
  g_LogProducer.set_identifier(g_papi_info_MPI_Get_processor_name->api_sig);
  g_LogProducer << sx_last_ret;
  if ((*len + 1)*sizeof(deref<char *>::type) > 0) {
    g_LogProducer << (uint32)(* len + 1)*sizeof(deref<char *>::type);
    g_LogProducer.write((LPVOID)name,
                        (uint32)(* len + 1)*sizeof(deref<char *>::type));
  }
  g_LogProducer << (uint32)sizeof(deref<int *>::type);
  g_LogProducer.write((LPVOID)len, (uint32)sizeof(deref<int *>::type));
  g_LogProducer << log::end;
END_SIGEX_MIXIN
```

Why is it wrong?

- replay slot

```
BEGIN_SIGEX_SLOT(MPI_Get_processor_name, hw_replay_MPI_Get_processor_name)
  START_PROFILE_WITH_NAME("hw_replay_MPI_Get_processor_name")
  g_LogConsumer.chk_identifier(g_papi_info_MPI_Get_processor_name->api_sig);
  int sx_last_ret;
  g_LogConsumer >> sx_last_ret;
  uint32 nParamBufLength;
  if ((*len + 1)*sizeof(deref<char *>::type) > 0) {
    g_LogConsumer >> nParamBufLength;
    g_LogConsumer.read((LPVOID)name, nParamBufLength);
  }
  g_LogConsumer >> nParamBufLength;
  g_LogConsumer.read((LPVOID)len, nParamBufLength);
  g_LogConsumer >> log::end;
  return sx_last_ret;
END_SIGEX_SLOT
```

Determine the buffer size

- Solution
  - a new tag: `force`
    - `always save it on log, and no test on replay`

```
int
MPI_Get_processor_name (
    [out, ecap(*len), esize(*len + 1), force] char *name,
    [out] int *len
    );
```

  - **`also necessary for MPI_Type_get_extent()`**

New Log slot

```
BEGIN_SIGEX_MIXIN(MPI_Get_processor_name, hw_log_MPI_Get_processor_name)
   START_PROFILE_WITH_NAME("hw_log_MPI_Get_processor_name")
   g_LogProducer.set_identifier(g_papi_info_MPI_Get_processor_name->api_sig);
   g_LogProducer << sx_last_ret;
   g_LogProducer << (uint32)(* len + 1)*sizeof(deref<char *>::type);
   g_LogProducer.write((LPVOID)name,
                       (uint32)(* len + 1)*sizeof(deref<char *>::type));
   g_LogProducer << (uint32)sizeof(deref<int *>::type);
   g_LogProducer.write((LPVOID)len, (uint32)sizeof(deref<int *>::type));
   g_LogProducer << log::end;
END_SIGEX_MIXIN
```

New Replay slot

```
BEGIN_SIGEX_SLOT(MPI_Get_processor_name, hw_replay_MPI_Get_processor_name)
  START_PROFILE_WITH_NAME("hw_replay_MPI_Get_processor_name")
  g_LogConsumer.chk_identifier(g_papi_info_MPI_Get_processor_name->api_sig);
  int sx_last_ret;
  g_LogConsumer >> sx_last_ret;
  uint32 nParamBufLength;
  g_LogConsumer >> nParamBufLength;
  g_LogConsumer.read((LPVOID)name, nParamBufLength);
  g_LogConsumer >> nParamBufLength;
  g_LogConsumer.read((LPVOID)len, nParamBufLength);
  g_LogConsumer >> log::end;
  return sx_last_ret;
END_SIGEX_SLOT
```

3. What is expected for '*opt*'?

- *If the arg might be NULL, set its flag to 'opt'*
- What about other special values but not NULL?
  - MPI_STATUS_IGNORE
  - MPI_STATUSES_IGNORE

```
MPI_Wait(&req, MPI_STATUS_IGNORE)
MPI_Waitall(cnt, &reqs, MPI_STATUSES_IGNORE)

#define MPI_STATUS_IGNORE (MPI_Status *)1   //!= NULL
#define MPI_STATUSES_IGNORE (MPI_Status *)1 //!= NULL
```

  - *NULL is not enough!*

Extends opt

- Solution
  - provide test value for ‘*opt*’
    - `[out, opt] arg`
      `if (arg != NULL)`
    - `[out opt(MY_VALUE)] arg`
      `if (arg != MY_VALUE)`

```
int
MPI_Wait (
    [in] MPI_Request *request,
    [out, opt(MPI_STATUS_IGNORE)] MPI_Status *status
    );
```

4. What do non-blocking operations imply?

- Return immediately, and test later before using
- Example

```
MPI_ISend(buf, cnt, dt, srg, tag,  comm, &request);
// do something
MPI_Wait(&request, &status);
```

```
MPI_IRecv(buf, cnt, dt, srg, tag,  comm, &request);
// do something
MPI_Wait(&request, &status);
```

- The buffer is NOT ready when it returns.

What do non-blocking operations imply?

- Similar to *asio*
- All non-blocking APIs post `MPI_Request` objects
- MPI_Wait() tests the request in blocking manner
  - MPI_Wait/MPI_Waitall/MPI_Waitany/MPI_Waitsome
- They set requests to MPI_REQUEST_NULL after returning
  - Take care of the pitfalls!
- *When and how to log the buffer?*
  - MPI_Wait() and MPI_Test() family APIs

MPI_Request

- MPI opaque object

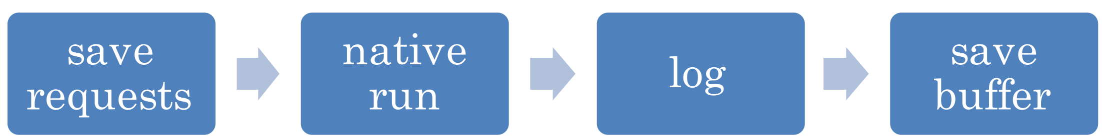
```
typedef int MPI_Request
```

- Track the buffer info attached to the request
- Useful in receives only
  - non-blocking send is nothing different with blocking send
  - Harmless even if the send fails
- Should we know the request type?
  - Sure, MPI_Wait does NOT set it to MPI_REQUEST_NULL always. ([persistent] send/recv)

Additional Slots

- Solution
  - Log
    - save buffer info on MPI_Irecv() and MPI_Recv_init()

    - save requests in MPI_Wait() family APIs
    - log the buffer after MPI_Wait() family APIs



  - Replay
    - back fill the buffer from the log as normal

5. What's special for collective operations?

- Who are they?
  - MPI_Bcast
  - MPI_Gather(v)/MPI_Allgather(v)
  - MPI_Scatter(v)
  - MPI_Reduce/MPI_Allreduce/MPI_Reduce_Scatter
  - MPI_Alltoall(v)
  - MPI_Scan
- Blocking
  - no non-blocking counterparts

What's special for collective operations?

- For the same arg, different meanings for different processes!
- e.g. MPI_Reduce()
  - recvbuf is only significant for the root!

```
int MPI_Reduce(void *sendbuf,  void *recvbuf, int count,
                       MPI_Datatype datatype,  MPI_Op op,
                       int root, MPI_Comm comm );
```

- Solution
  - always mark 'out', but different 'bsize'

What's special for collective operations?

- How to replay if some processes are out of the group?
  - Record phase: save the membership
  - Replay phase: emulated with p2p communication
  - MPI_Bcast() example

```
/* MPI_Bcast() replay code */
load MPI_Bcast rank_list from log
if (I am root) { /* for data sender */
  foreach rank in rank_list:
    if (rank is in replay group)
      send message to rank
} else { /* for data receiver */
  if (root is in group)
    recv message from root
  else
    load message from log
}
```

6. Where to pack the data?

- MPI_Pack/MPI_Unpack

```
int MPI_Pack(
  [in] void *inbuf,
  [in] int incount,
  [in] MPI_Datatype datatype,
  [out, bsize("get_pack_size()"), force] void *outbuf,
  [in] int outcount,
  [inout] int *position,
  [in] MPI_Comm comm
);
```

- Solution
  - Additional slots:

save in position → native run → log

7. What about MPI_Cancel()?

- MPI_Cancel() just posts a cancel request!
- MPI_Wait() to make sure whether the request is cancelled or not internally by set the **cancelled** field in **status**.
- MPI_Test_cancelled() simply tests the **cancelled** field in **status**.

```
MPI_Cancel(&request);
MPI_Wait(&request, &status);
MPI_Test_cancelled(&status, &flag);
```

Remove the cancelled requests

- Solution
  - Filter the cancelled requests in MPI_Wait()/MPI_Test() family APIs when saving the non-blocking buffers according to the status.
  - Actually, this is an optimization more than a feature. Since there is no harm if we always save the cancelled operations.

8. Some other evils

- MPI_BOTTOM
  - No one would use this bullshit API in real applications!
- MPI_IN_PLACE
  - Optimization for some collective operations (MPI_Scatter)
  - Just mark as: opt(MPI_IN_PLACE)
- MPI_Start()
  - Why no MPI_Stop()?
  - We can cleanup the rubbish, but not in a decent manner.
- MPI_Alloc_mem()/MPI_Free_mem()
  - The MPI forum committee is, eh…, dreaming.

9. I/O Redirection

- All STD IO HANDLES are redirected to socks
  - printf → NtWriteFile
- Replay
  - printf→NtWriteConsoleA
- Solution
  - Replayer helps
  - anonymous pipes to take over the STD HANDLES

Evaluation

Setup

- 8 nodes of 2×4-Core Intel Xeon 2.33 GHz CPUs
- 8GB RAM
- 140GB Disk
- Windows Server 2003 Enterprise Edition SP1
- MPICH2-1.0.7
- 1Gbps Ethernet LAN

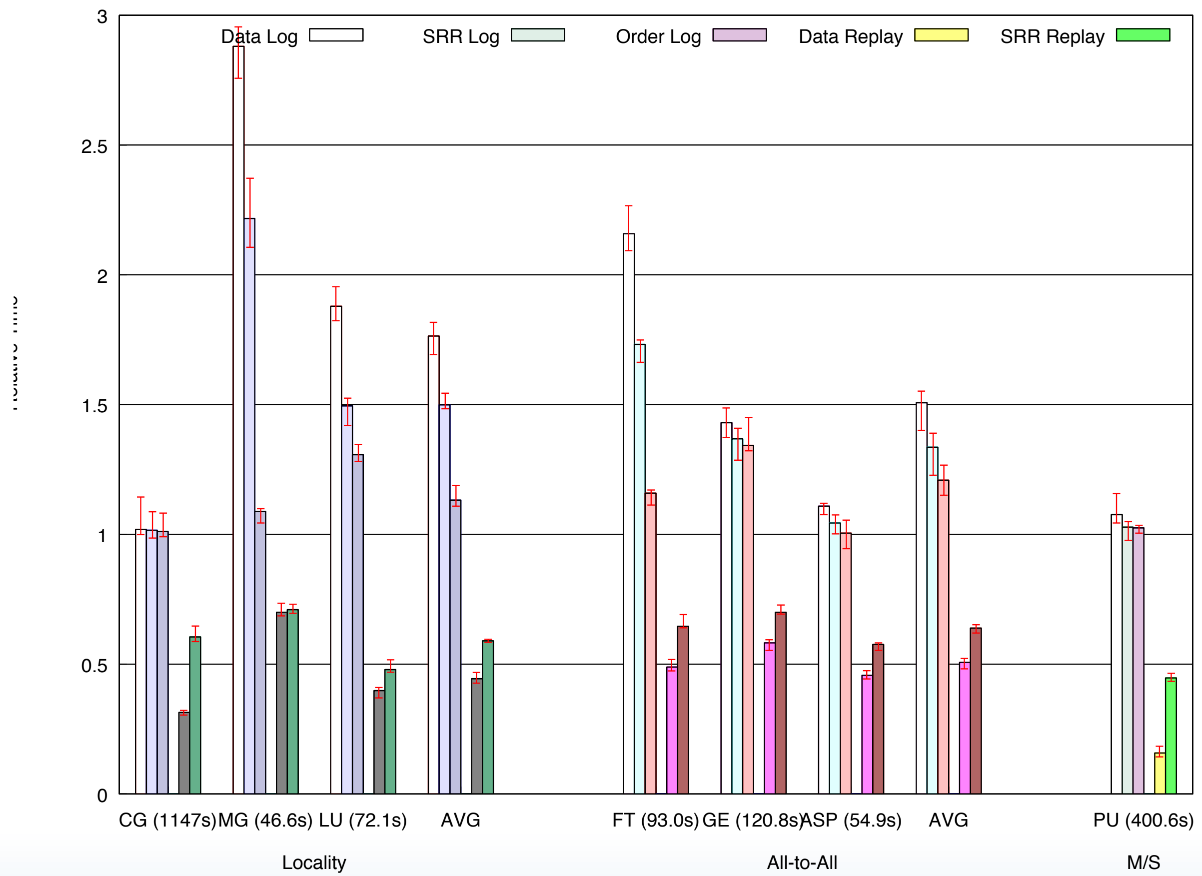
Applications

- Three kinds of communication patterns
- Different kinds of non-determinisms

	Communication Pattern						
	Locality			All-to-All			M/S
Operations	CG	MG	LU	FT	GE	ASP	PU
Non-determ. MPI		√	√		√	√	√
Non-determ. Sys	√	√	√	√	√		
Coll. Operations	√	√	√	√	√	√	

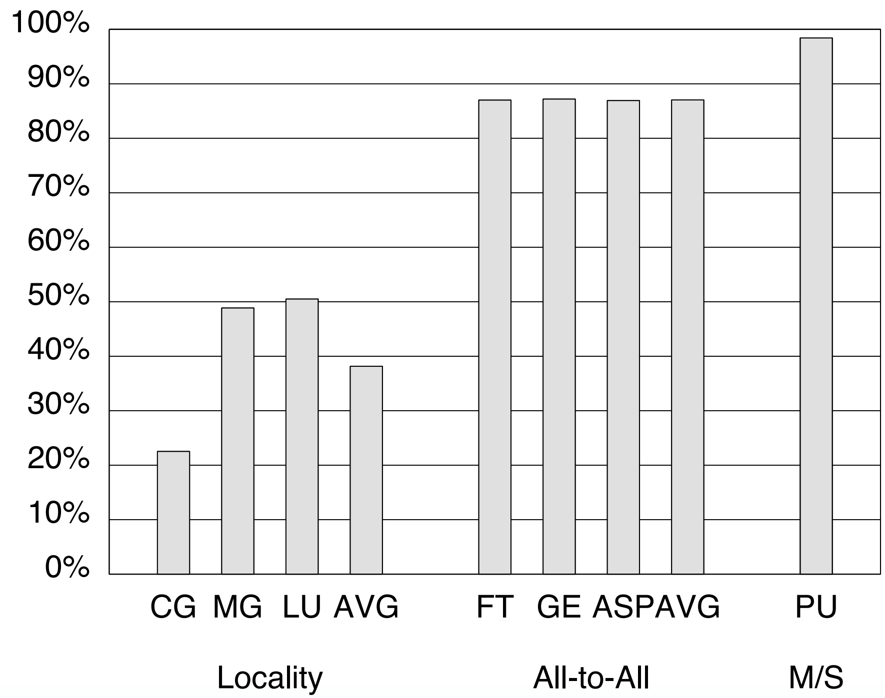
Execution Time

- Record Overhead
- Replay Overhead



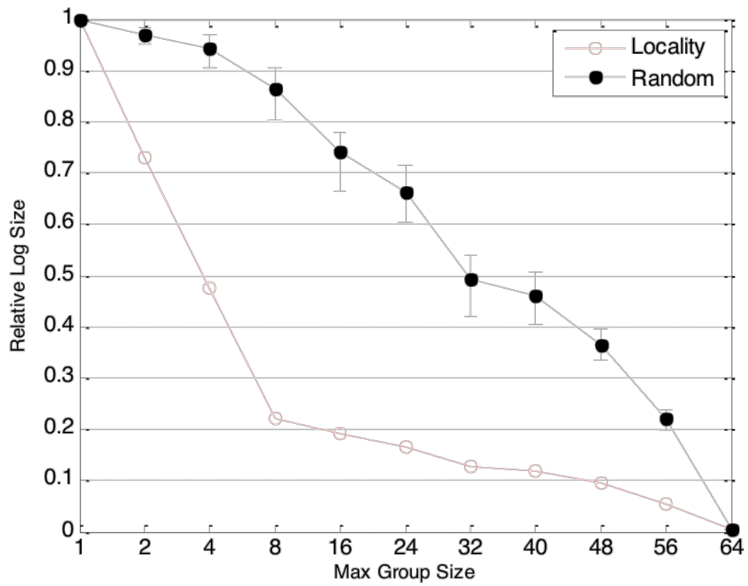
Log Size

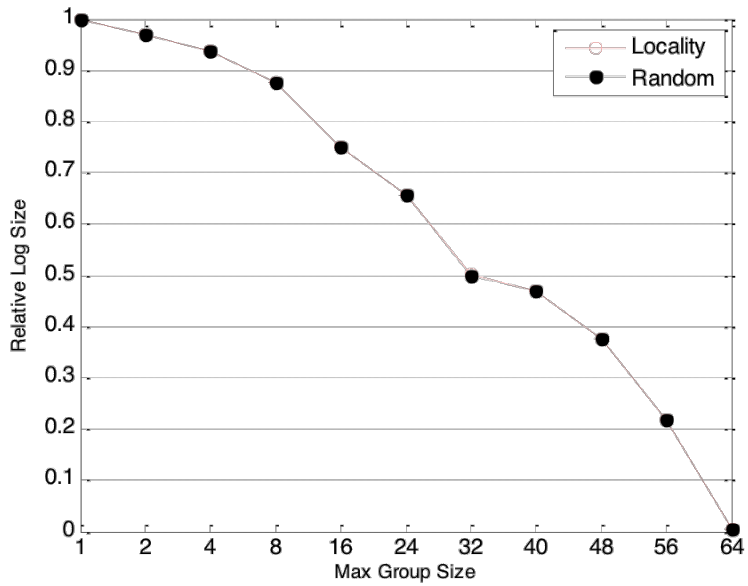
- Locality: 38%
- All-to-All: 87%
- M/S: 98% (1%)



Group-size and Membership

- Informed vs. Random
- CG & FT (CLASS=C, NPROCS=64)





Conclusion

- Subgroup Reproducible Replay
- A flexible balance between data- and order-replay
- Exploit communication locality
- Handles MPI and OS non-determinisms