

# Minimizing MPI Resource Contention in Multithreaded Multicore Environments

**Dave Goodell**,<sup>1</sup> Pavan Balaji,<sup>1</sup> Darius Buntinas,<sup>1</sup>  
Gábor Dózsa,<sup>2</sup> William Gropp,<sup>3</sup> Sameer Kumar,<sup>2</sup>  
Bronis R. de Supinski,<sup>4</sup> Rajeev Thakur,<sup>1</sup>

`goodell@mcs.anl.gov`

ANL,<sup>1</sup> IBM,<sup>2</sup> UIUC/NCSA,<sup>3</sup> LLNL<sup>4</sup>

September 21, 2010

# Overview

## MPI Background

- MPI Objects

- MPI & Threads

## Naïve Reference Counting

- Basic Approach

- An Improvement

## Hybrid Garbage Collection

- Algorithm

- Analysis

## Results

- Benchmark and Platform

- The Numbers



# MPI Objects

- Most MPI objects are **opaque objects**
- Created, manipulated, and destroyed via **handles** and functions
- Object handle examples: `MPI_Request`, `MPI_Datatype`, `MPI_Comm`
- MPI types such as `MPI_Status` are *not* opaque (direct access to `status.MPI_ERROR` is valid)
- In this talk, **object** always means an opaque object



# The *Premature Release* Problem

## Example

```
MPI_Datatype tv;
```

```
MPI_Type_vector(..., &tv);
```

```
MPI_Type_commit(&tv);
```

```
MPI_Type_free(&tv);
```

# The *Premature Release* Problem

## Example

```
MPI_Datatype tv;  
MPI_Comm comm;  
  
MPI_Comm_dup(MPI_COMM_WORLD, &comm);  
MPI_Type_vector(..., &tv);  
MPI_Type_commit(&tv);  
  
MPI_Comm_free(&comm);  
MPI_Type_free(&tv);
```

# The *Premature Release* Problem

## Example

```
MPI_Datatype tv;  
MPI_Comm comm;  
MPI_Request req;  
MPI_Comm_dup(MPI_COMM_WORLD, &comm);  
MPI_Type_vector(..., &tv);  
MPI_Type_commit(&tv);  
MPI_Irecv(buf, 1, tv, 0, 1, comm, req);  
MPI_Comm_free(&comm);  
MPI_Type_free(&tv);  
... arbitrarily long computation ...  
MPI_Wait(&req);
```

This is a premature release. `comm` and `tv` are still in use at user-release time



# User Convenience, Implementer Pain

- Supporting the “simple” case is trivial:
  - `MPI_Type_vector`  $\mapsto$  `malloc`
  - `MPI_Type_free`  $\mapsto$  `free`
- The more complicated premature release case requires more effort, typically reference counting.



## Terminology Note

- To minimize confusion, let us refer to functions like `MPI_Type_free` as **user-release functions** and their invocation as **user-releases**.
- **ref** means “reference”



# MPI Reference Counting Semantics

- MPI objects must stay alive as long as logical references to them exist. Usually corresponds to a pointer under the hood.
- Objects are born with only the user's ref.
- The user can release that ref with a user-release (e.g. `MPI_Comm_free`)
- MPI operations logically using an object may acquire a reference to that object, which is then released when finished.
- An MPI object is no longer in use and eligible for destruction when there are no more references to the object.



# MPICH2 Objects

- All MPICH2 objects are allocated by a custom allocator (not directly by `malloc/free`).
- All objects have a common set of header fields.
- We place an atomically-accessible, **reference count** (“`refcount`”) integer field here.
- This field is initialized to 1 on object allocation.

# The Naïve Algorithm

( $A$ ,  $B$ , and  $C$  are opaque MPI objects)

1. If  $A$  adds a ref to  $B$ , atomically increment  $B$ 's reference count.
2. If ownership of a ref to  $B$  changes hands from  $A$  to  $C$ , don't change  $B$ 's reference count.
3. If  $A$  releases a ref to  $B$ , atomically decrement and test  $B$ 's reference count against zero. If zero, deallocate the object.



# Reference Counting Example

## Example

refcount		
tv	comm	
-	-	MPI_Datatype tv;
-	-	MPI_Comm comm;
-	-	MPI_Request req;
-	1	MPI_Comm_dup(MPI_COMM_WORLD, &comm);
1	1	MPI_Type_vector(..., &tv);
1	1	MPI_Type_commit(&tv);
2	2	MPI_Irecv(buf, 1, tv, 0, 1, comm, req);
2	1	MPI_Comm_free(&comm);
1	1	MPI_Type_free(&tv);
1	1	... arbitrarily long computation ...
0	0	MPI_Wait(&req);



## Downsides

### Example

```
MPI_Request req[NUM_RECV];  
for (i = 0; i < NUM_RECV; ++i)  
    MPI_Irecv(..., &req[i]); // ATOMIC{++(c->ref_cnt)}  
MPI_Waitall(req); // for NUM_RECV: ATOMIC{--(c->ref_cnt)}
```

- Different threads running on different cores/processors will fight over the cache line containing the ref count for the communicator and datatype.
- Even the waitall will result in NUM\_RECV atomic decrements for each shared objects.

# An Improvement

- Many codes (and benchmarks) don't use user-derived objects.
- Predefined objects (`MPI_COMM_WORLD`, `MPI_INT`, etc) are not explicitly created in the usual fashion.
- Their lifetimes are bounded by `MPI_Init` and `MPI_Finalize` and **cannot be freed**.



# An Improvement

- Many codes (and benchmarks) don't use user-derived objects.
- Predefined objects (`MPI_COMM_WORLD`, `MPI_INT`, etc) are not explicitly created in the usual fashion.
- Their lifetimes are bounded by `MPI_Init` and `MPI_Finalize` and cannot be freed.
- *Upshot*: simply don't maintain reference counts for predefined objects.



## An Improvement

- Many codes (and benchmarks) don't use user-derived objects.
- Predefined objects (`MPI_COMM_WORLD`, `MPI_INT`, etc) are not explicitly created in the usual fashion.
- Their lifetimes are bounded by `MPI_Init` and `MPI_Finalize` and cannot be freed.
- *Upshot*: simply don't maintain reference counts for predefined objects.
- Easy to implement in MPICH2; completely removes contention in the critical path.
- Doesn't help us at all for user-derived...



# One Man's Trash...

- Problem: MPI\_Comm and MPI\_Datatype refcount contention (possibly others too, MPI\_Win)
- Communicators/datatypes/etc are usually long(ish) lived.
- MPI\_Requests are frequently created and destroyed.
- Suggests a garbage collection approach to manage communicators, etc.



# Definitions

**GCMO** Garbage Collection Managed Object. These are long-lived, contended objects: communicators, datatypes, etc.

**Transient** Short-lived, rarely contended objects: requests

$G_\ell$  The set of live GCMOs, must not be deallocated

$G_e$  The set of GCMOs eligible for deallocation

$T$  The set of transient objects



## High Level Approach

- Disable reference counting on GCMO objects due to transient objects. Other refcounts remain!
- Add a live/not-live boolean in the header of all GCMOs.
- Maintain  $T$ ,  $G_\ell$ , and  $G_e$  somehow (we used lists)
- At creation, GCMOs are added to  $G_\ell$ . Refcount starts at 2 (user ref and garbage collector ref).
- When a GCMO's refcount drops to 1, move it to  $G_e$ .
- Periodically run a garbage collection cycle (next slide).



# Garbage Collection Cycle

1. lock the allocator if not already locked
2. *Reset*: Mark every  $g \in G_e$  not-live.
3. *Mark*: For each  $t \in T$ , mark any referenced GCMOs (eligible or not) as live.
4. *Sweep*: For each  $g \in G_e$ , deallocate if  $g$  is still marked not-live.
5. unlock the allocator if we locked it in step 1



## Garbage Collection Example

refcount		
tv	comm	
-	-	MPI_Datatype tv;
-	-	MPI_Comm comm;
-	-	MPI_Request req;
-	2	MPI_Comm_dup(MPI_COMM_WORLD, &comm);
2	2	MPI_Type_vector(..., &tv);
2	2	MPI_Type_commit(&tv);
2	2	MPI_Irecv(buf, 1, tv, 0, 1, comm, req);
2	1	MPI_Comm_free(&comm);
1	1	MPI_Type_free(&tv);
1	1	... arbitrarily long computation ...
1	1	MPI_Wait(&req);
0	0	// something triggers GC cycle



# Analysis

- When  $|G_e| > 0$ , collection cycle cost bound, fixed # GCMO refs per transient object:  $\mathcal{O}(|G_e| + |T|)$
- When  $|G_e| > 0$ , cycle cost bound, variable # GCMO refs per transient object:  $\mathcal{O}(|G_e| + r_{\text{avg}}|T|)$
- $|G_\ell|$  is not present in bound  $\implies$  GC performance penalty only for “prematurely” freed GCMOs and outstanding requests.

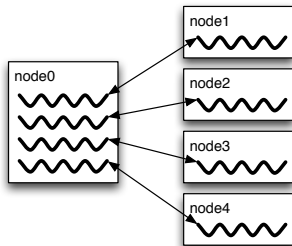


## When to Collect?

- `MPI_Finalize`, obviously
- Collection at new GCMO allocation time makes sense.
- Flexible here: could be probabilistic, could be a function of memory pressure, could be a timer.
- GCMO creation is not usually expected to be lightning fast, won't be in most inner loops.
- We already hold the allocator's lock.
- GCMO user-release time is an option, but makes less sense.



# Benchmark



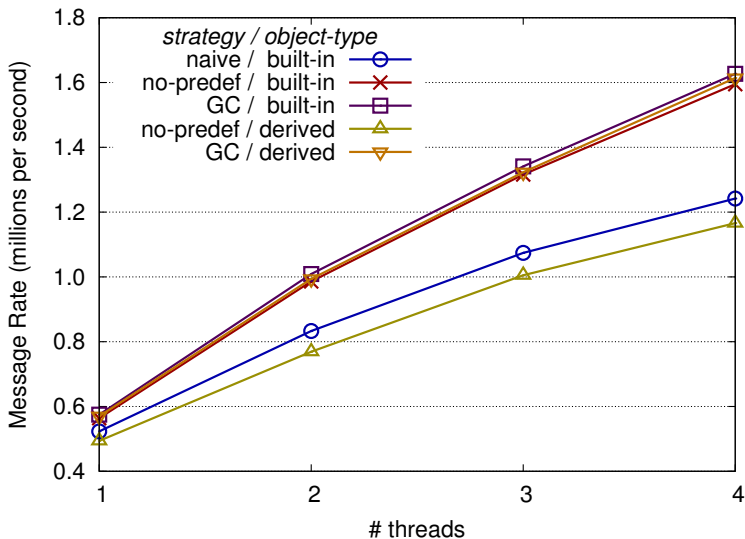
- `MPI_THREAD_MULTIPLE` benchmarks and applications are rare/nonexistent.
- We wrote a benchmark based on the Sequoia Message Rate Benchmark (SQMR).
- Each iteration posts 12 nonblocking sends and 12 nonblocking receives, then calls `MPI_Waitall`.
- 10 warm-up iterations, then time 10,000 iterations, report average time per message.
- All are 0-byte messages.

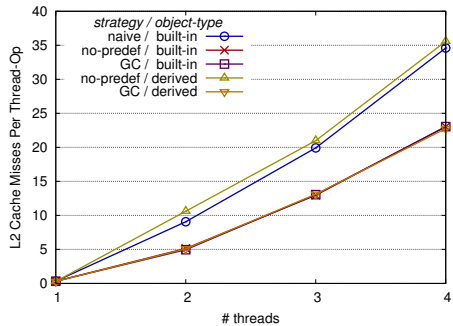
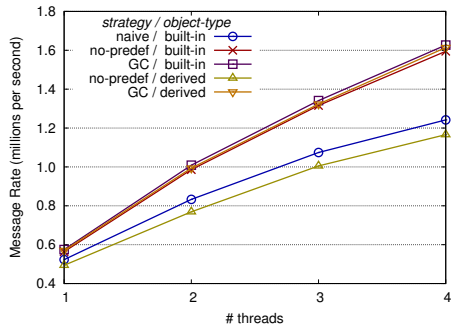
# Test Platform

- ALCF's Surveyor Blue Gene/P system.
- 4 – 850 MHz PowerPC cores
- 6 bidirectional network links per node, arranged in a 3-D torus
- multicore, but unimpressively so
- network-level parallelism is the key here, a serialized network makes this work pointless



## Message Rate Results — Absolute





# Summary

- MPI specifies clear semantics for opaque object lifetimes that map trivially to reference counting.
- Reference counting with multithreading is usually expensive due to cache line contention.
- Suppressing refcounts for predefined objects (`MPI_COMM_WORLD`) is cheap and safe. Doesn't help user-defined objects.
- Hybrid refcount+GC can pull the performance bottleneck out of the critical path.
- Hybrid scheme is fairly easy to retrofit into an existing refcount mechanism.

Questions?

Questions?

