Expressing Fault Tolerant Algorithms with MPI-2

William D. Gropp
Ewing Lusk
www.mcs.anl.gov/~gropp

Overview

- Myths about MPI and Fault Tolerance
  - ♦ Error handling and reporting
- Goal of Fault Tolerance
  - ♦ Run applications
    - Science simulations are different from real-time controls and from databases
- Checkpointing
  - ♦ The best solution?
- Generalizing transactional semantics
  - ♦ Well-studied for databases
  - ♦ Built around two-party transactions

Myths and Facts

Myth: MPI behavior is defined by its implementations.

Fact: MPI behavior is defined by the Standard Document at http://www.mpi-forum.org

Myth: MPI is not fault tolerant.

Fact: This statement is not well formed. Its truth depends on what it means, and one can't tell from the statement itself. More later.

Myth: All processes of MPI programs exit if any one process crashes.

Fact: Sometimes they do; sometimes they don't; sometimes they should; sometimes they shouldn't. More later.

Myth: Fault tolerance means reliability.

Fact: These are completely different. Again, definitions are required.

More Myths and Facts

Myth: Fault tolerance is independent of performance.

Fact: In general, no. Perhaps for some (weak) aspects, yes. Support for fault tolerance will negatively impact performance.

Myth: Fault tolerance is a property of the MPI standard (which it doesn't have).

Fact: Fault tolerance is not a property of the specification, so it can't not have it. ☺

Myth: Fault tolerance is a property of an MPI implementation (which most don't have).

Fact: Fault tolerance is a property of a program. Some implementations make it easier to write fault-tolerant programs than others do.

Even More Myths and Facts

Myth: Computers with tens or hundreds of thousands of processors will be failing constantly

Fact: The frequency of faults does not scale (simply) with the number of processors. More important are the number of mechanical connections (e.g., pins and cables), non-redundant systems with moving parts (e.g., fans and disks), and high-stress, low margin components (e.g., cheap PC power supplies). And software.

What Does the MPI Standard Say That is Relevant to Fault Tolerance?

- MPI requires reliable* communication. An implementation that permits messages to be corrupted in transit and still delivered to the user is a non-conforming MPI implementation. (Regrettably, not a hypothetical case.)
- MPI allows users to attach error handlers to communicators.
  - MPI_ERRORS_ABORT, the "all-fall-down" error handler, is required to be the default.
    - How often do *you* check the return code from a routine?
  - MPI_ERRORS_RETURN can be used to allow applications (and especially libraries) to handle errors.
  - Users can write and attach their own error handlers on a communicator-by-communicator basis.
    - Modularity!

*guaranteed delivery, for network types

Goals of Fault Tolerance in (many) Scientific Simulations

- The goal of the simulation is to answer a question with the minimum *total* resource.
- A failed simulation is “only” lost resource (compared to a lost bank transaction)

Checkpointing

- Is Checkpointing so bad?
  - ♦ Pros:
    - Does not change user's algorithm
    - Modular; does not impact other components of the application
  - ♦ Cons:
    - Depends on high-performance I/O
    - Requires either user-directed or compiler-assisted checkpointing for efficiency
- How expensive is checkpointing?

Checkpointing

- K_0 – Cost to create and write
- K_1 – Cost to read and restore
- A – Probability of failure
- T_0 – Time between checkpoints
- T – Total time to run, without checkpoints

The Cost of Checkpointing

- If the probability of failure is independent of time and has an exponential PDF, and is small, then an estimate of the total time with failures is
  $E_T=(T/t_0)(K_0+t_0+a(K_1t_0+(1/2)t_0^2))$
  - Tradeoff – frequent checkpoints reduce the about of “lost” compute time but incure greater overhead
- We can optimize for the number of checkpoints by finding the value of t_0 that minimizes this, leading to

Optimized Checkpointing

- $E_T = T (1 + aK_1 + (2aK_0)^{1/2})$
- To minimize this cost, we can
  - ♦ Reduce the probability of failure “a”
    - Various robust communication strategies for internode communication failures
    - Use better hardware and software
  - ♦ Reduce the cost of reading and writing a checkpoint
    - Use parallel I/O and checkpoint only the data needed to restart the computation
    - Use “lazy redundancy” to provide fully overlapped, cost effective fault-tolerance in the I/O system

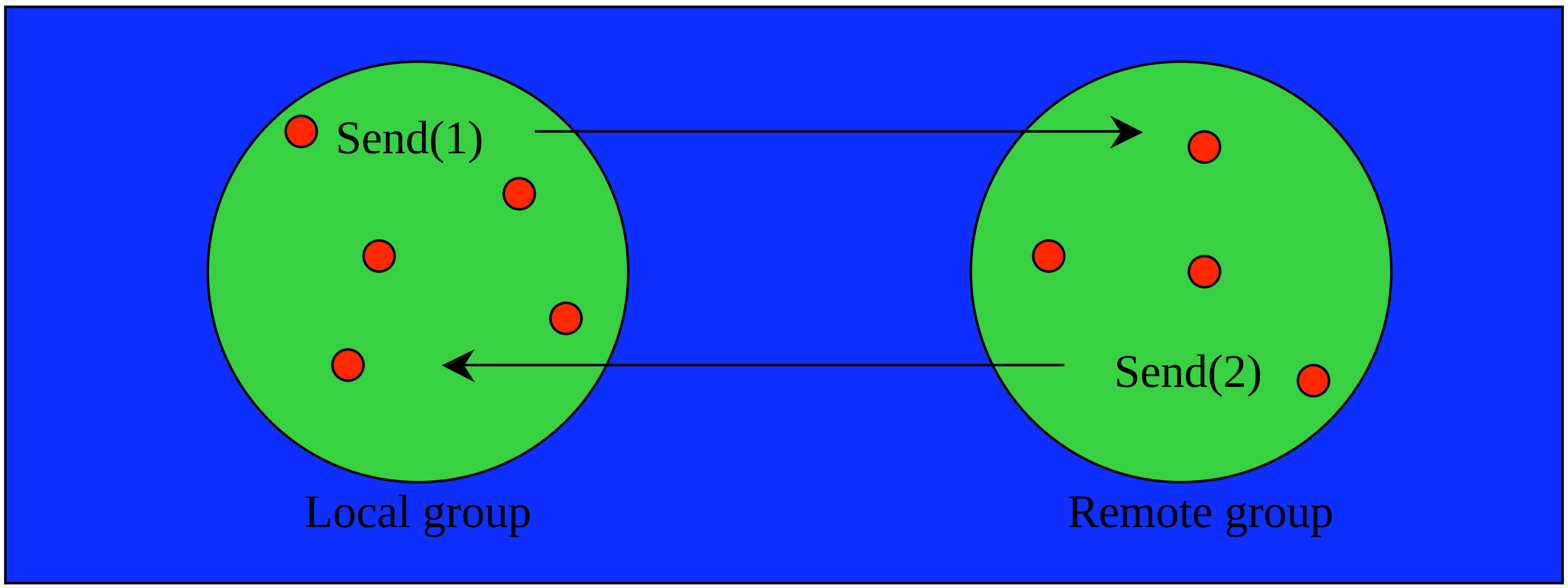
Generalizing Two-Party Operations

- Fault tolerance is well studied and understood in other areas of CS
- One major approach relies on carefully defined operations between two agents.
- In many fault-tolerant scientific applications today, the agents are processes and the communication is handled by a *socket* or a *remote procedure call*
- MPI provides a natural way to generalize this: the *intercommunicator*

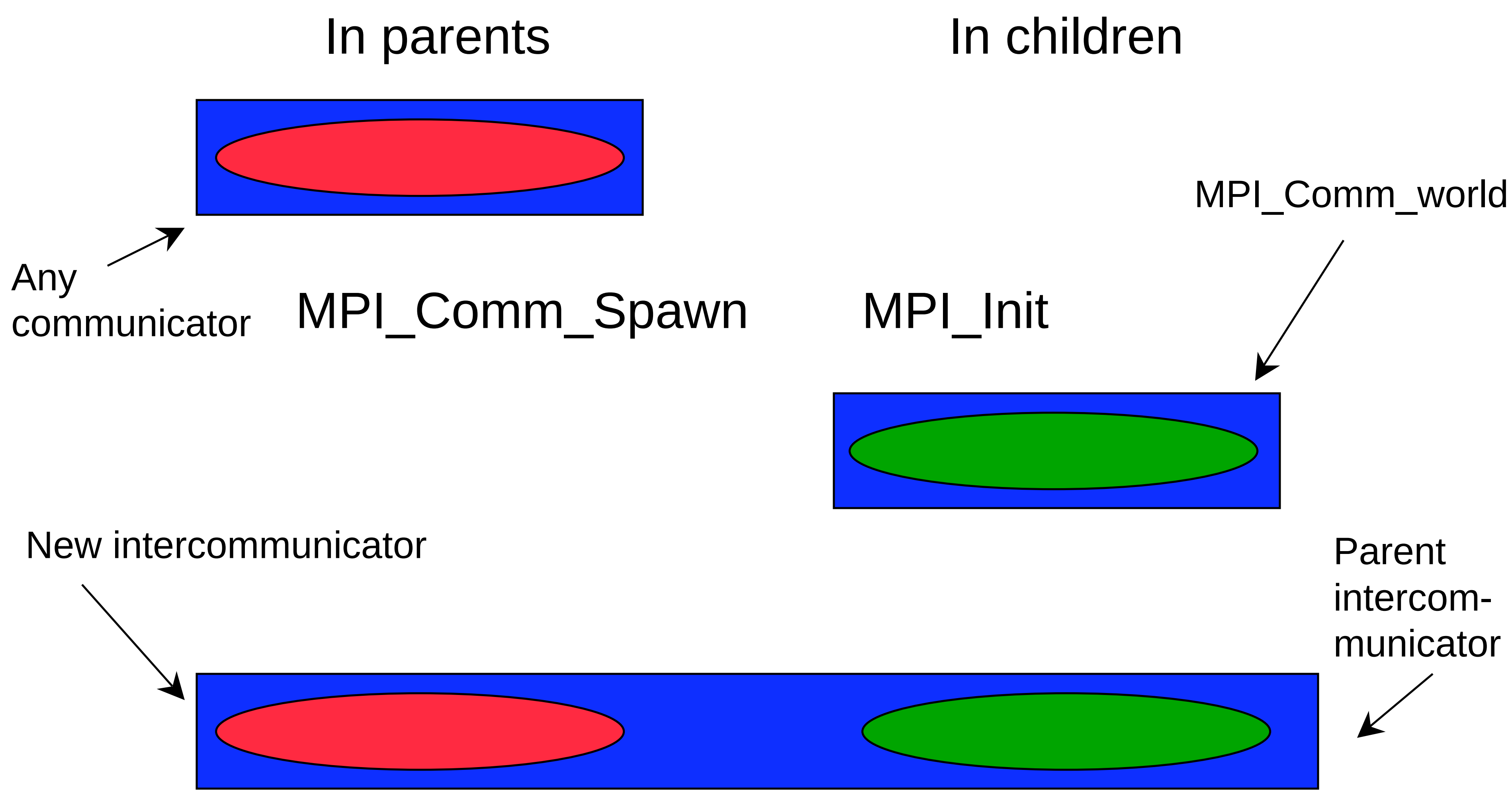
Intercommunicators

- Contain a *local* group and a *remote* group
- Point-to-point communication is between a process in one group and a process in the other.
- Can be merged into a normal (intra) communicator.
- Created by **`MPI_Intercomm_create`** in MPI-1.
- Play a more important role in MPI-2, created in multiple ways.

Intercommunicators



Spawning New Processes



Two Party Operations in MPI

- Generalize the *process* to be an MPI communicator
  - Well defined behavior and semantics, just like a process.
  - Provides more resources (parallelism) without changing application details
- Generalize the communication to operations on an intercommunicator
  - MPI provides both point-to-point and collective
  - Semantics of intercommunicator collectives often appropriate for fault-tolerant apps
    - Implementations can enhance (not change) the semantics to provide additional guarantees

Master/Slave Programs with Intercommunicators

- One type of program easy to make fault-tolerant is the master/slave paradigm (seti@home).
- This is because slaves hold very small amount of state at a time.
- Such an algorithm can be expressed in MPI, using intercommunicators to provide a level of fault-tolerance, if the MPI implementation provides a robust implementation of MPI_ERRRORS_RETURN for intercommmunicators.

A Fault-Tolerant MPI Master/Slave Program

- Master process comes up alone first.
  - Size of MPI_COMM_WORLD = 1
- It creates slaves with MPI_Comm_spawn
  - Gets back an intercommunicator for each one
  - Sets MPI_ERRORS_RETURN on each
- Master communicates with each slave using its particular communicator
  - MPI_Send/Recv to/from rank 0 in remote group
  - Master maintains state information to restart each subproblem in case of failure
- Master may start replacement slave with MPI_Comm_spawn
- Slaves may themselves be parallel
  - Size of MPI_COMM_WORLD > 1 on slaves
  - Allows programmer to control tradeoff between fault tolerance and performance

Extending MPI

- New objects and methods with new syntax and semantics to support the expression of fault-tolerant algorithms in MPI
- Example – The MPI_Process_array object, somewhat like an MPI Communicator (retains idea of context), but
  - ♦ Has dynamic instead of constant size
  - ♦ Rank of process replaced by constant array index
  - ♦ No collective operations for process arrays
    - Full semantics too application specific – this should be left to libraries built on MPI that applications use
  - ♦ New send/receive operations would be defined for processes identified by an index into a process array.
  - ♦ Can have attached error handler
- Might be more convenient than an intercommunicator-based approach for master/slave computations where slaves communicate among themselves.

Conclusion

- Fault tolerance is a property of an algorithm, not a library
  - Management of state is the key
- It is important to be able to express a fault-tolerant parallel algorithm as an MPI program
- Some solutions are already in use
- Implementations can provide more support than they currently do for fault tolerance, without changing the MPI specification
- Additions to the MPI Standard may be needed to extend the class of fault tolerant algorithms that can be expressed conveniently in MPI
- Further research is needed, first in improvements to MPI-2 implementations, and eventually into MPI extensions

Backup

Fault Tolerance in MPI

- Can MPI be fault tolerant?
  - What does that mean?
- Implementation vs. Specification
  - Work to be done on the implementations
  - Work to be done on the algorithms
    - Semantically meaningful and efficient collective operations
  - Use MPI at the correct level
    - Build libraries to encapsulate important programming paradigms
- (Following slides are joint work with Rusty Lusk)

Outline

- Myths about MPI and fault tolerance
- Definitions of fault tolerance
- Relevant parts of the MPI standard
- MPI can support a class of fault-tolerant programs
  - ♦ If implementation provides certain features
  - ♦ Example of fault-tolerant master-slave program in MPI
- Extending the MPI Standard to allow more fault-tolerant programs
  - ♦ Adding new MPI objects and methods
- Disclaimer – These are preliminary thoughts

What is Fault Tolerance Anyway?

- A fault-tolerant program can “survive” (in some sense we need to discuss) a failure of the infrastructure (machine crash, network failure, etc.)
- This is not in general completely attainable. (What if *all* processes crash?)
- How much is recoverable depends on how much *state* the failed component holds at the time of the crash.
  - In many master-slave algorithms a slave holds a small amount of easily recoverable state (the most recent subproblem it received).
  - In most mesh algorithms a process may hold a large amount of difficult-to-recover state (data values for some portion of the grid/matrix).
  - Communication networks hold varying amount of state in communication buffers.

Types of "Survival"

- The MPI library automatically recovers.
- Program is notified of problem and takes corrective action.
- Certain operations, but not all, become invalid.
- Program can be restarted from checkpoint.
- Perhaps combinations of these.

What Does the Standard Say About Errors?

- A set of errors is defined, to be returned by MPI functions if MPI_ERRORS_RETURN is set.
- Implementations are allowed to extend this set.
- It is not required that subsequent operations work after an error is returned. (Or that they fail, either.)
- It may not be possible for an implementation to recover from some kinds of errors even enough to return an error code (and such implementations are conforming).
- Much is left to the implementation; some conforming implementations may return errors in situations where other conforming implementations abort. (See “quality of implementation” issue in the Standard.)
  - Implementations are allowed to trade performance against fault tolerance to meet the needs of their users

Some Approaches to Fault Tolerance in MPI Programs

- Master-slave algorithms using intercommunicators
  - No change to existing MPI semantics
  - MPI intercommunicators generalize the well-understood two party model to groups of processes, allowing either the master or slave to be a parallel program optimized for performance.
- Checkpointing
  - In wide use now
  - Plain vs. fancy
  - MPI-IO can help make it efficient
- Extending MPI with some new objects in order to allow a wider class of fault-tolerant programs.
  - The “pseudo-communicator”
- Another approach: Change semantics of existing MPI functions
  - No longer MPI (semantics, not syntax, defines MPI)

Checkpointing

- Application-driven vs. externally-driven
  - ♦ Application knows when message-passing subsystem is quiescent
  - ♦ Checkpointing every n timesteps allows very long (months) ASCI computations to proceed routinely in face of outages.
  - ♦ Externally driven checkpointing requires much more cooperation from MPI implementation, which may impact performance.
- MPI-IO can help with large, application-driven checkpoints
- “Extreme” checkpointing – MPICH-V (Paris group)
  - ♦ All messages logged
  - ♦ States periodically checkpointed asynchronously
  - ♦ Can restore local state from checkpoint + message log since last checkpoint
  - ♦ Not high-performance
  - ♦ Scalability challenges