# Design patterns for error handling in C++ programs using parallel algorithms and executors


Mark Hoemmen*

mhoemmen@stellarscience.com


CppCon 2020

* hoʊ’mən, or hœm’mən; he/him

# Who am I?

- > 10 years post-PhD experience writing parallel C++ for science and engineering
- Background: Parallel algorithms for big linear algebra problems
- 1st WG21: Nov 2017
- Started new job at Stellar Science in March

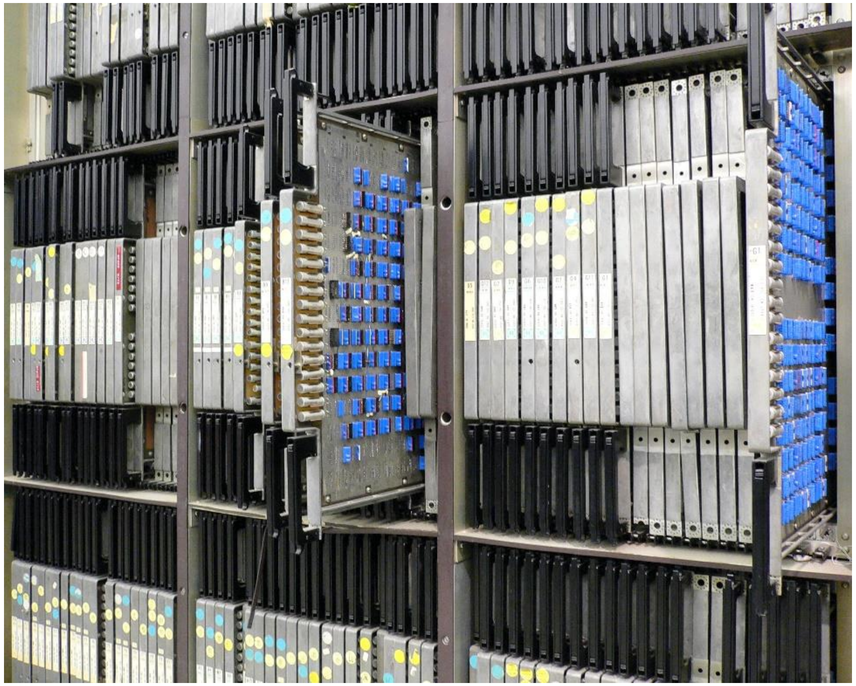
Eschew raw pointers

# Outline

- Parallelism makes error handling harder…
- …C++ parallel algorithms and tasks specifically
- Message Passing Interface (MPI): 3 decades of distributed-memory parallel programming
- MPI teaches design patterns to detect and handle recoverable errors

# What is "parallel"?

- **Use multiple hardware resources**
  - Nodes, cores, SIMD, …
- **To accomplish >1 work item at the same time**
- **To improve performance**
  - Latency,
  - Throughput, or
  - Responsiveness



ILLIAC IV: the 1st massively parallel computer (image: Steve Jurvetson)

# Parallel hinders error handling

- **… because parallelism <u>relaxes execution order</u>**
  - Deliberately, to improve performance
- **Errors <u>interrupt</u> execution; handling <u>constrains</u> order**
- **Errors could lead to deadlock (waiting forever)**
  - e.g., 1 worker drops out before collective synchronization
- **Correct handling requires <u>communication</u>**
  - Data movement, or synchronization (same thing)
  - Stop other workers from waiting forever
  - Propagate and combine error info from workers

# No zero-overhead solution

- Error handling requires communication
- Communication is expensive
- Making C++ “do it for you” won’t be free
- ➔ If you want a zero-overhead solution, …
- … you, the coder, will need to handle errors

 ¥ € $

(there is no free lunch)

# Parallelism in Standard C++

- **Parallel (C++17) algorithms**
  - `for_each`, `reduce`, `transform`, `sort`, …
  - `ExecutionPolicy`: Permitted changes in execution order
  - Throw in loop body ➔ `terminate` (*)
- **Asynchronous tasks (C++11 `async`)**
  - Uncaught exception in task gets captured
  - Waiting on result throws passed-along exception
- **P0443 executors + P1897 asynchronous algorithms**
  - Separate path for handling ancestor task's uncaught exception
  - `when_all`: Express dependency on >1 tasks
  - If >1 parent throws, `when_all` captures <u>any 1</u> exception

(*) for all policies currently in the Standard

# Exceptions cause trouble

- Parallel (C++17) algorithms
  - `for_each`, `reduce`, `transform`, `sort`, …
  - `ExecutionPolicy`: Permitted changes in execution order
  - Throw in loop body ➔ `terminate`
- Asynchronous tasks (C++11 `async`)
  - Uncaught exception in task gets captured
  - Waiting on result throws passed-along exception
- P0443 executors + P1897 asynchronous algorithms
  - Separate path for handling ancestor task's uncaught exception
  - `when_all`: Express dependency on >1 tasks
  - If >1 parent throws, `when_all` captures <u>any 1</u> exception

# Exceptions cause trouble

- **Parallel (C++17) algorithms**
  - `for_each`, `reduce`, `transform`, `sort`, …
  - `ExecutionPolicy`: Permitted changes in execution order
  - Throw in loop body ➔ `terminate`
- **Asynchronous tasks (C++11 `async`)**
  - Uncaught exception in task gets captured
  - Waiting on result throws passed-along exception
- **P0443 executors + P1897 asynchronous algorithms**
  - Separate path for handling ancestor task's uncaught exception
  - `when_all`: Express dependency on >1 tasks
  - If >1 parent throws, `when_all` captures any 1 exception

Exceptions are for recoverable errors

# Code characteristics leading to recoverable errors

Outside parallel code

| Status of exceptions | Typical code activities |
|---|---|
| May break user interface if not carefully handled | User interface, app driver |
| | Launches &/or drives parallel algorithms |
| Non-bug exceptions & other recoverable errors are more likely | Call 3rd-party libraries; do I/O; complicated &/or speculative computations |
| Exceptions usually indicate bugs | Tight, optimized loops |
| Not allowed, even if caught | Explicit SIMD; C++ dialects |

Inside parallel code

Granularity (complexity of parallel “loop body” / task)

# Example: Domain decomposition

- **Solve big linear system of equations Ax=b**
  - Decompose into many small systems (“subdomains”)
  - Solve small systems <u>independently</u> and combine results
  - Approximation → repeat to convergence
- **Domain decomposition might fail**
  - Entire problem has no solution (mathematically), or
  - Too many iterations to get an accurate solution, or
  - Some small systems may not have a solution
- **Fall-back algorithms take more memory and time**

# Parallel domain decomposition

## Previous non-parallel code

```
try {
  preSolveWork();
  for(auto&& subdomain: subdoms) {
    solve(subdomain);
  }
  postSolveWork();
} catch(/* what solve throws */) {
  useSlowerSolver = true;
}
```

## Parallelization attempt

```
FixedMemoryPool pool(initSize);
try {
  preSolveWork();
  for_each(execution::par_unseq,
    begin(subdoms), end(subdoms),
    [&] (auto&& subdomain) {
      solve(subdomain, pool);
    });
  postSolveWork();
} catch(/* what solve throws */) {
  useSlowerSolver = true;
}
```

# Parallel domain decomposition

## Previous non-parallel code

```
try {
  preSolveWork();
  for(auto&& subdomain: subdoms) {
    solve(subdomain);
  }
  postSolveWork();
} catch(/* what solve throws */) {
  useSlowerSolver = true;
}
```

- Fixed-size memory pool to speed up allocation in parallel loop
- Replace for loop with C++17 parallel algorithm

## Parallelization attempt

```
FixedMemoryPool pool(initSize);
try {
  preSolveWork();
  for_each(execution::par_unseq,
    begin(subdoms), end(subdoms),
    [&] (auto&& subdomain) {
      solve(subdomain, pool);
    });
  postSolveWork();
} catch(/* what solve throws */) {
  useSlowerSolver = true;
}
```

# Parallel domain decomposition

## Must distinguish 2 errors

- **Memory pool too small?**
  - Count total needed space
  - Reallocate pool
  - Retry domain decomposition
- **Any subdomain solve fails?**
  - Fall back to slower solver
- **Throw ➔ `terminate`**
  - Can't distinguish by `catch`ing different exception types

## Parallelization attempt

```
FixedMemoryPool pool(initSize);
try {
  preSolveWork();
  for_each(execution::par_unseq,
    begin(subdoms), end(subdoms),
    [&] (auto&& subdomain) {
      solve(subdomain, pool);
    });
  postSolveWork();
} catch(/* what solve throws */) {
  useSlowerSolver = true;
}
```

# Parallel domain decomposition

## Must distinguish 2 errors

- **Memory pool too small?**
  - Count total needed space
  - Reallocate pool
  - Retry domain decomposition
- **Subdomain solve failed?**
  - Fall back to slower solver
- **Throw → `terminate`**
  - Can't distinguish by `catch`ing different exception types

## Parallelization attempt

```
FixedMemoryPool pool(initSize);
try {
  preSolveWork();
  for_each(execution::par_unseq,
    begin(subdoms), end(subdoms),
    [&] (auto&& subdomain) {
      solve(subdomain, pool);
    });
  postSolveWork();
} catch(/* what solve throws */) {
  useSlowerSolver = true;
}
```

What do we do?!?

stellar science

# <u>M</u>essage <u>P</u>assing <u>I</u>nterface

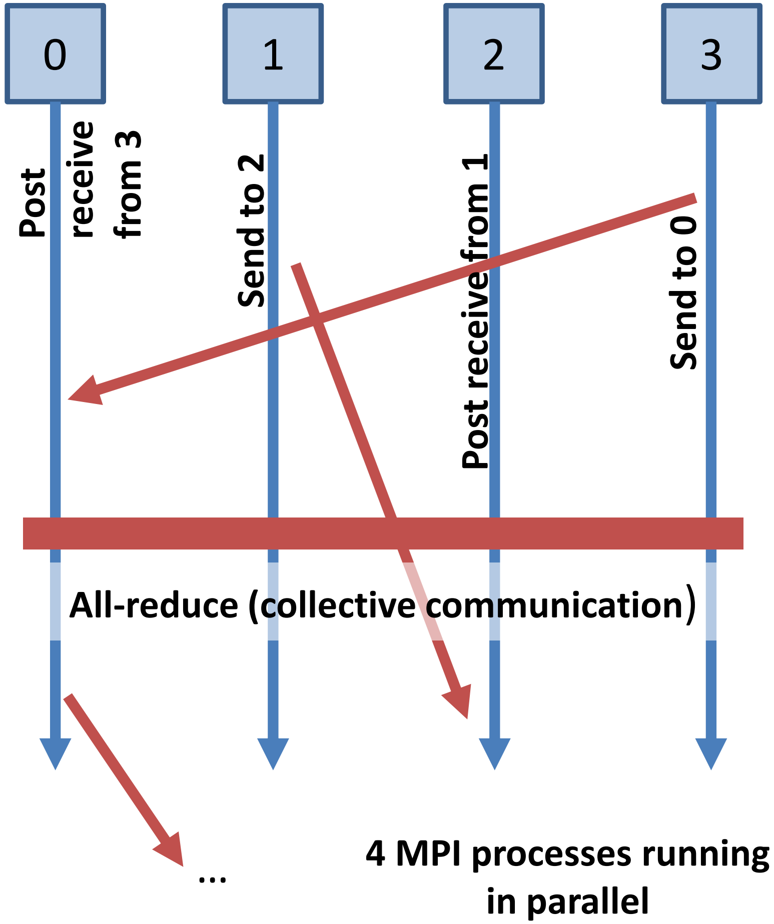
- **C and Fortran interface**
  - For writing distributed-memory parallel code
- **Standard for 3 decades**
  - Unified divergent interfaces circa 1991
  - MPI 1.0 released 1994
  - 3.1 in 2015; 4.0 pending
- **Millions-way parallelism**
- **Stable: ‘90s code works**

MPI

- **Modest hardware requirements**
- **Cooperates with other programming models**
  - e.g., for GPUs

# MPI's parallelism model

- **P "processes"**
  - Fixed location & number
  - Like parallel `for_each` over 0, 1, …, P-1
- **Do NOT share memory**
- **Communicate explicitly, mainly by <u>messages</u>**
  - Explicit function calls
  - "2-sided": sender & receiver must participate
  - Point to point or collectives



# MPI hostile to error handling

- **…a bit more so than C++ parallel algorithms**
- **MPI deliberately punted on error handling**
  - Compared to early ‘90s competitor PVM
  - Goal: Run faster on wider range of systems
- **MPI equivalent of `terminate`: only best effort**
  - `MPI_Abort` on 1 process (hopefully) stops all processes
- **C++ exceptions: Undefined behavior if uncaught**
  - 1 process’ uncaught exception often causes deadlock
  - Can set `terminate_handler` that calls `MPI_Abort`

# MPI's error handling lessons

- **Turn exceptions into values**
  - before they risk breaking control flow or being uncaught
- **Turn fail-able "parallel for" into reduction**
  - Reduce on "did everybody succeed?"
  - Collect info for recovery and/or reporting
- **Prevent synchronization-related deadlock**
  - If you must synchronize…
  - … use it as opportunity to communicate error state
- **Exploit out-of-band error reporting**

# Turn exceptions to values, and reduce over work items

- **C++17 parallel algorithms**
  - Can't get uncaught exceptions back to caller, so:
  - Turn them to values (e.g., error code) via try-catch
  - Reduce over whether all work items succeeded
  - e.g., change `for_each` to `reduce`
  - See later for "not just a parallel for" algorithms like `sort`
- **P0443 tasks with P1897 `when_all`**
  - `when_all` drops all but 1 exception; if you need more:
  - Turn them to values via e.g., `let_error`
  - Next task "reduces" over values (like above)

# Parallel domain decomposition

## Reduce over error info

- **Each solve returns `Result`**
  - `status`: Why failed (bit field)
  - `bytesNeeded` (from pool)
- **Reduce over solve results**
  - If memory pool too small, this gives us required pool size

## Parallelization

```
FixedMemoryPool pool(initSize);
preSolveWork();
Result result = transform_reduce(
  execution::par_unseq,
  begin(subdoms), end(subdoms),
  [] (Result x, Result y) {
    return Result{x.status & y.status,
      x.bytesNeeded + y.bytesNeeded}; },
  [&] (auto&& subd) {
    return trySolve(subd, pool);
  });
postSolveWork();
if(result.status & SOLVE_FAILED) {
  useSlowerSolver = true;
} else if(result.status & OUT_OF_MEM) {
  pool.resize(result.bytesNeeded);
  retry = true;
}
```

# Parallel domain decomposition

## Reduce over error info

- **Each solve returns `Result`**
  - `status`: Why failed (bit field)
  - `bytesNeeded` (from pool)
- **Reduce over solve results**
  - If memory pool too small, this gives us required pool size

## Parallelization

```
FixedMemoryPool pool(initSize);
preSolveWork();
Result result = transform_reduce(
  execution::par_unseq,
  begin(subdoms), end(subdoms),
  [] (Result x, Result y) {
    return Result{x.status & y.status,
      x.bytesNeeded + y.bytesNeeded}; },
  [&] (auto&& subd) {
    return trySolve(subd, pool);
  });
postSolveWork();
if(result.status & SOLVE_FAILED) {
  useSlowerSolver = true;
} else if(result.status & OUT_OF_MEM) {
  pool.resize(result.bytesNeeded);
  retry = true;
}
```

# Parallel domain decomposition

## Reduce over error info

- **Each solve returns `Result`**
  - `status`: Why failed (bit field)
  - `bytesNeeded` (from pool)
- **Reduce over solve results**
  - If memory pool too small, this gives us required pool size

## Parallelization

```
FixedMemoryPool pool(initSize);
preSolveWork();
Result result = transform_reduce(
  execution::par_unseq,
  begin(subdoms), end(subdoms),
  [] (Result x, Result y) {
    return Result{x.status & y.status,
      x.bytesNeeded + y.bytesNeeded}; },
  [&] (auto&& subd) {
    return trySolve(subd, pool);
  });
postSolveWork();
if(result.status & SOLVE_FAILED) {
  useSlowerSolver = true;
} else if(result.status & OUT_OF_MEM) {
  pool.resize(result.bytesNeeded);
  retry = true;
}
```

# Parallel domain decomposition

## Reduce over error info

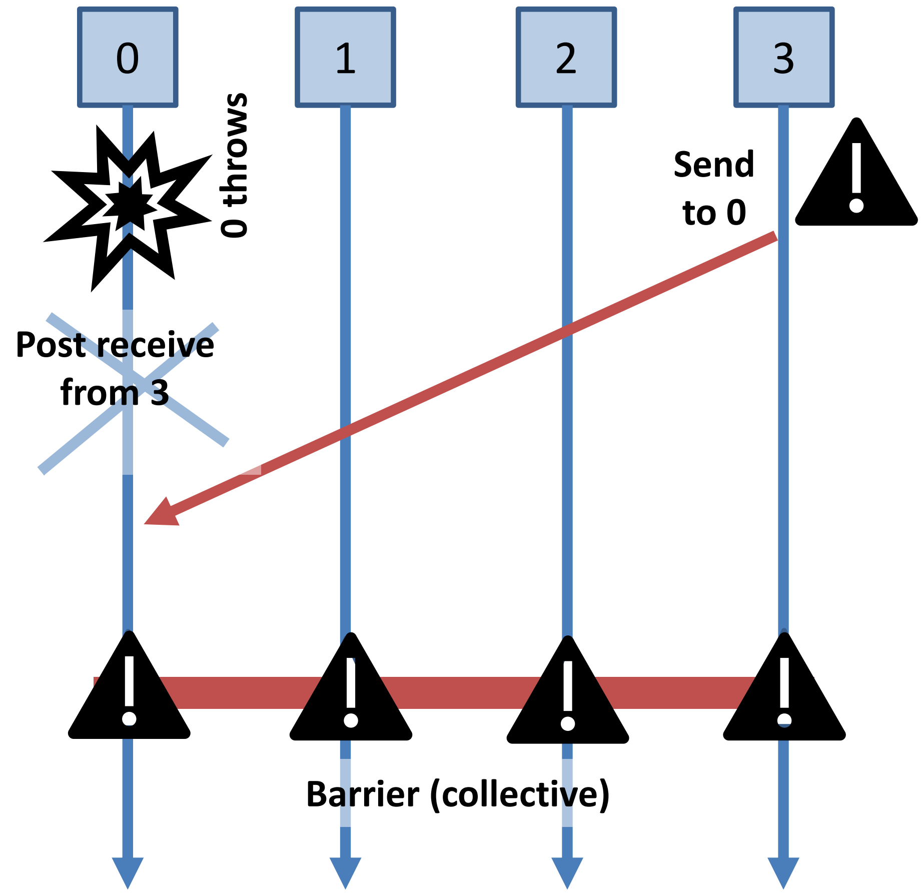
- **Each solve returns `Result`**
  - `status`: Why failed (bit field)
  - `bytesNeeded` (from pool)
- **Reduce over solve results**
  - If memory pool too small, this gives us required pool size

## Parallelization

```
FixedMemoryPool pool(initSize);
preSolveWork();
Result result = transform_reduce(
  execution::par_unseq,
  begin(subdoms), end(subdoms),
  [] (Result x, Result y) {
    return Result{x.status & y.status,
      x.bytesNeeded + y.bytesNeeded}; },
  [&] (auto&& subd) {
    return trySolve(subd, pool);
  });
postSolveWork();
if(result.status & SOLVE_FAILED) {
  useSlowerSolver = true;
} else if(result.status & OUT_OF_MEM) {
  pool.resize(result.bytesNeeded);
  retry = true;
}
```

# Deadlock is easy in MPI

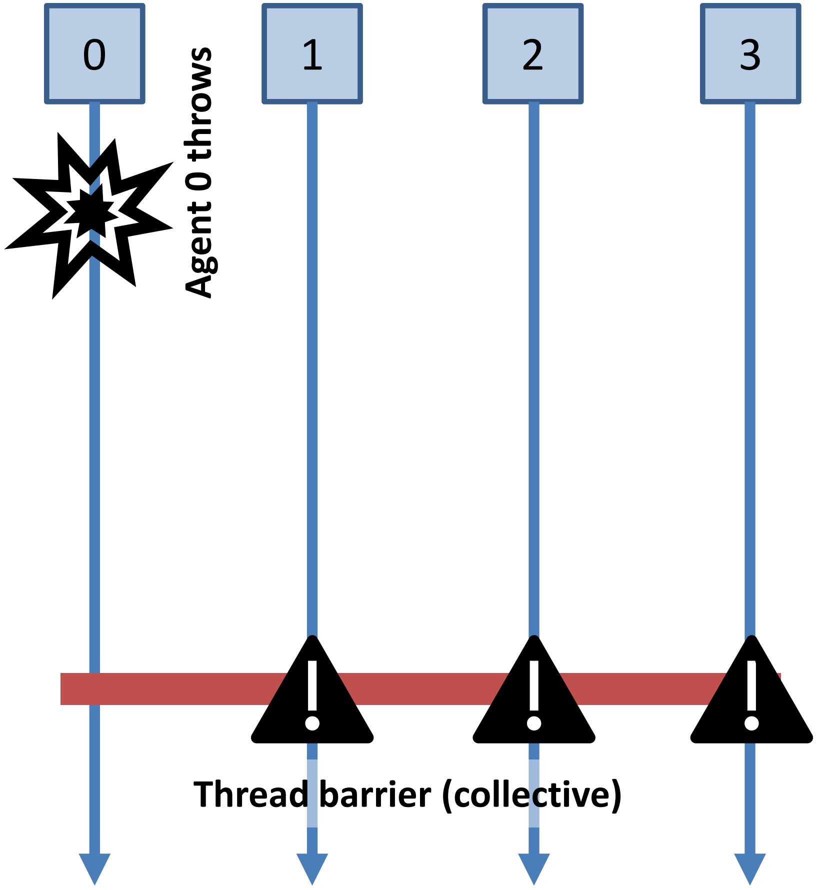
- Entire program is a “parallel for_each”
- Processes sometimes must communicate
- Each involved process must participate
- Else other processes may wait forever




Where a process may wait forever

# Deadlock in parallel C++: Still possible

- **e.g., “thread barrier”**
  - All threads must reach it before any may proceed
  - e.g., `std::latch`
  - Can implement with atomic counter
- **Use: Shared resource**
  - Make sure all agents have stopped using it, so coordinator can release




⚠ Where an agent of execution may wait forever

# Avoid deadlock by “going through the motions”

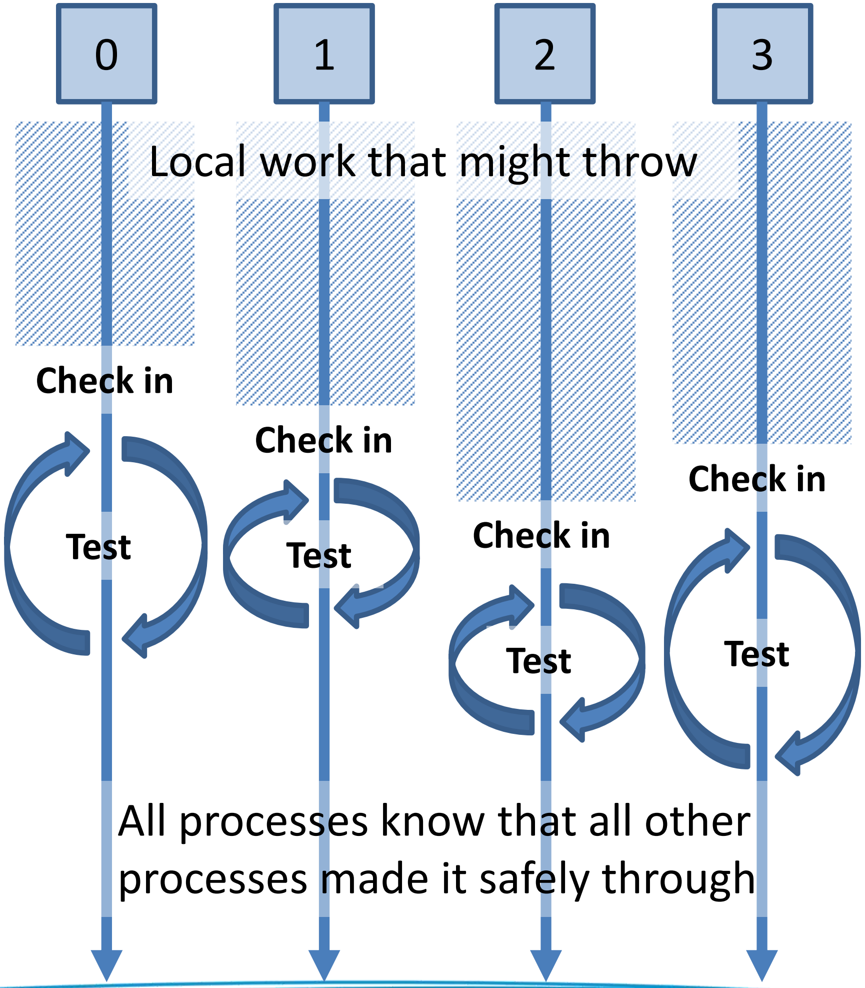
- **Think of parallel loop body as a sequence of “local” blocks, punctuated by synchronization**
  – Always participate in synchronization
  – Give each block a “bypass”: if error, do nothing harmlessly and pass along the running error state
  – Compare to P0443 “error channel”
- **Synchronization: opportunity for error reporting**
  – For thread barrier implemented as atomic counter:
  – Agent with error adds large number instead of one
  – (final count > number of agents) ➔ error

# Out-of-band error reporting

- **Asynchronous with respect to what you're using**
- **MPI: nonblocking barrier `MPI_Ibarrier`**
  - Collectives (barriers, reductions, …) are not ordered with respect to other kinds of communication
  - Nonblocking != "makes progress in the background"
    - Communication need not happen until you wait or test
    - Analogous to `async(f, ...)`
    - But you can poll (test in a loop) to "force progress"
- **Use `MPI_Ibarrier` to test if a process threw**
  - Or dropped out

# MPI: “Exception barrier”

- **Run “risky” local work**
  - Might throw
- **“Check in” when done**
  - Call `MPI_Ibarrier`
- **Spin on `MPI_Test`**
  - With a timeout
  - Can do other work speculatively while spinning
- Timeout ➔ call `MPI_Abort`
- Success ➔ safely through



# Out-of-band in C++: Atomics

- **(Lock-free) atomic updates don't block**
  - ➔ can use them anywhere in parallel algorithms or tasks
  - e.g., write error info somewhere for later use
- **Use in `sort` or other non-reduction algorithms**
- **Use if you don't want to pay for reduction**
  - Not a zero-overhead abstraction if errors are rare
  - Atomic updates may hinder compiler optimizations
  - But that only matters if loop is "too optimized for recoverable exceptions" anyway

# Summary

- **C++ parallel algorithms and asynchronous tasks make error handling harder**
- **Use design patterns developed for MPI**
  - Turn exceptions into values; reduce over error info
  - Avoid deadlock due to synchronization
  - Exploit out-of-band communication for error reporting
- **Thanks to Stellar Science!**

# Summary

- **C++ parallel algorithms and asynchronous tasks make error handling harder**
- **Use design patterns developed for MPI**
  - Turn exceptions into values; reduce over error info
  - Avoid deadlock due to synchronization
  - Exploit out-of-band communication for error reporting
- **Thanks to Stellar Science!**

Hiring modern C++ software engineers!

hmeep hmeep