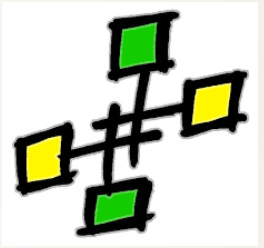
Surviving Errors with OpenSHMEM

Aurelien Bouteiller, George Bosilca, Manjunath Gorentla Venkata

OpenSHMEM Workshop 2016

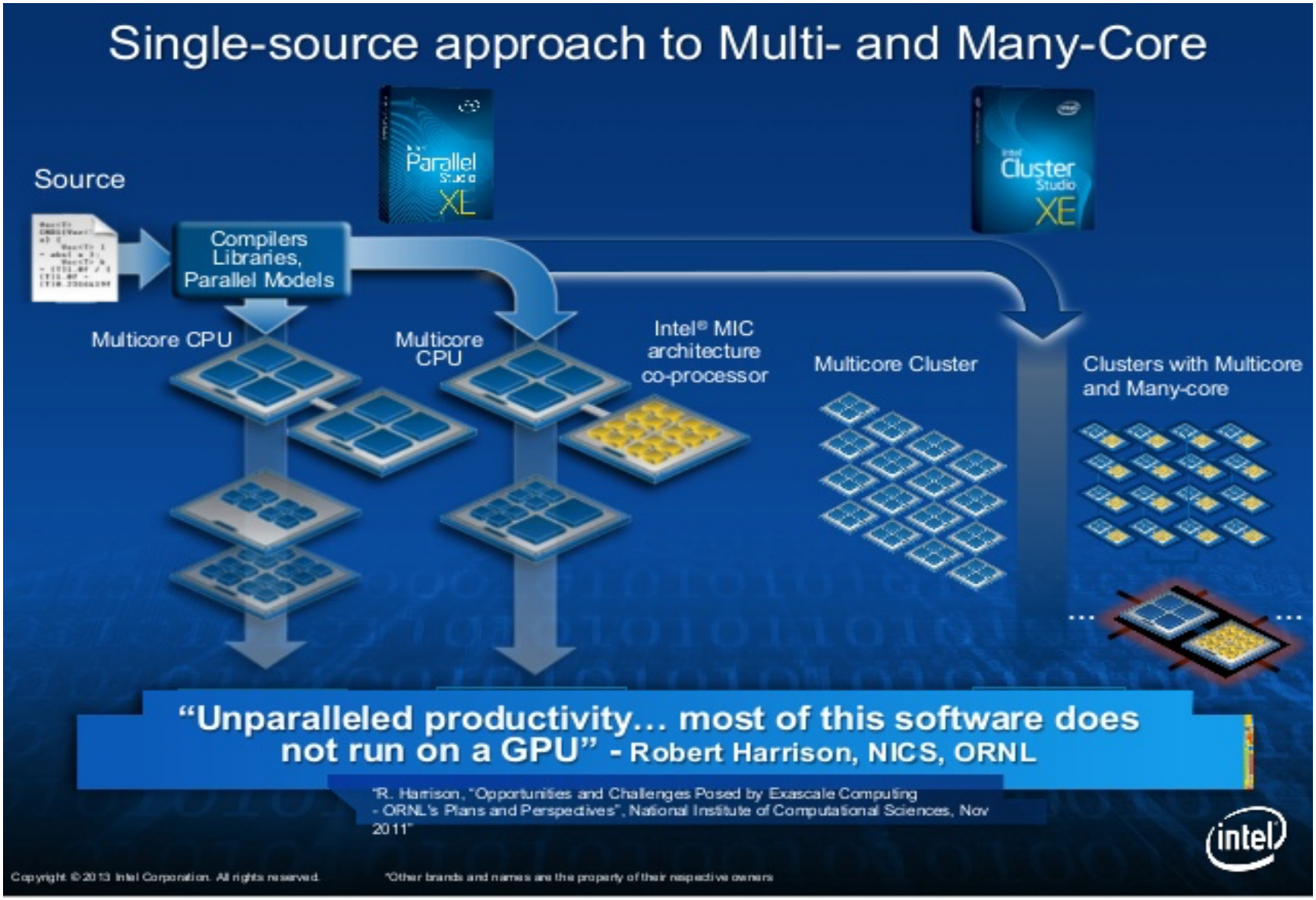
Baltimore, MD



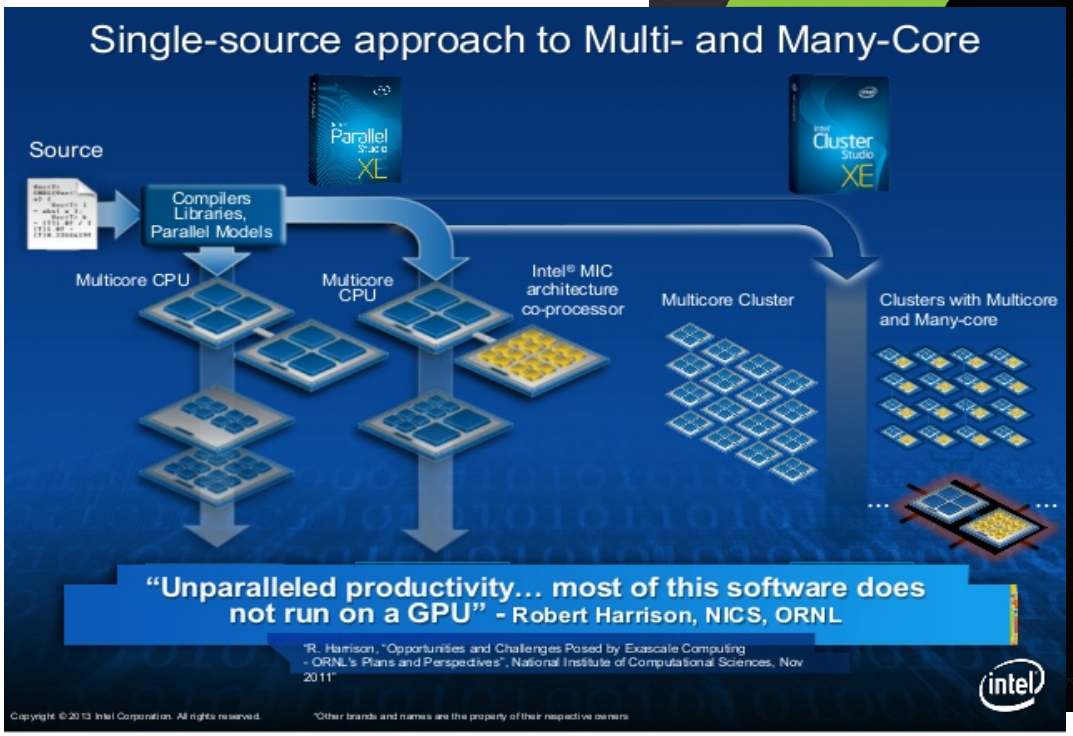
ICL UT
INNOVATIVE COMPUTING LABORATORY
THE UNIVERSITY of TENNESSEE

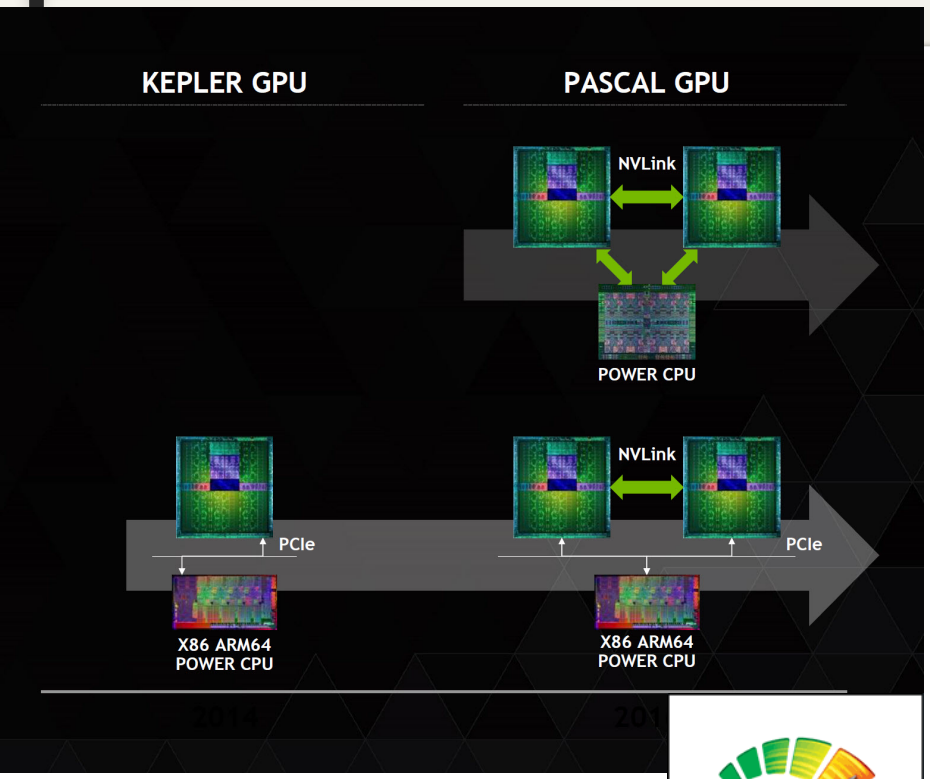
OAK RIDGE National Laboratory

Motivation: a complicated world

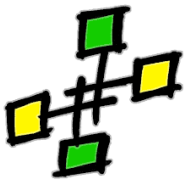


Motivation: a complicated world





- Multiple vectors make **errors** more common
  - Multicore/NUMA
  - Accelerators
  - MPI+OpenMP+OpenSHMEM
  - +CUDA, +OpenCL, +MPSS, +OpenACC



OpenCL

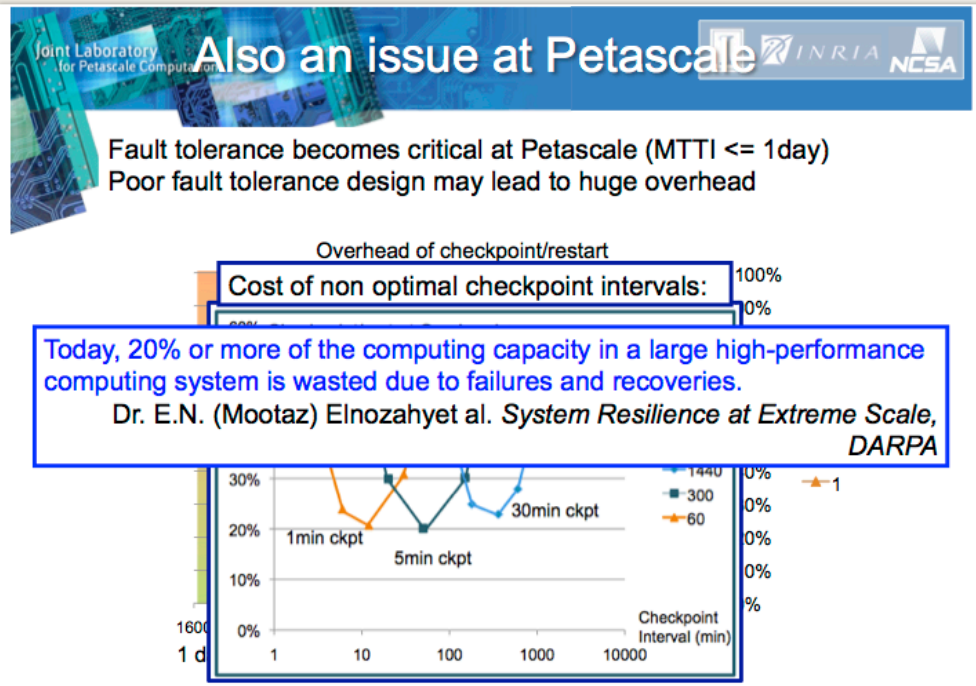
OpenMP

MPI

OpenACC
Directives for Accelerators

Multiple software stacks interoperating:
more user errors, more unexpected resource exhaustions

Motivation: large scale



Slide courtesy of F. Cappello

- Crash **failures**
  - More common with system scale
  - Already harming at Petascale
- Other "communication environments" are getting ready
  - User Level Failure Mitigation in MPI
  - +Fenix to support easy in-place C/R
  - +LFLR to support easy containment domains
  - Resilient X10
  - Resilient Coarrays
  - …
- OpenSHMEM cannot be the weak link in an interoperable application!

Goals

- Failure types:
  - Resource exhaustion
  - Processor Crash
  - Network interface crash
  - ~~Byzantine errors/arbitrary corruptions~~
  - Erroneous system software (debugging/resilience)
  - Erroneous user code (for debugging)
  - Memory corruption resulting in process crash
  - Silent data corruption

At a minimum, we want to return control to the user, so that continuation with an FT communication library (OpenSHMEM or other) is possible.

Always favor error free performance

Error codes

Page 8, Section 6: library constants

C/C++: SHMEM_ERR_ARG *Fortran*: SHMEM_ERR_ARG	An invalid argument value was passed to an operation.
C/C++: SHMEM_ERR_NOMEM *Fortran*: SHMEM_ERR_NOMEM	Not enough memory to perform the operation.
C/C++: SHMEM_ERR_UNREACH *Fortran*: SHMEM_ERR_UNREACH	Impossible to contact a remote PE.

It is expected that we will add more error codes as we want to address more issues

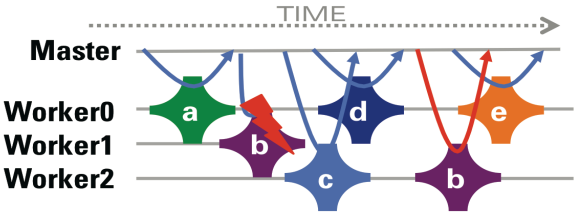
Landscape of FT techniques

Coordinated Checkpoint/Restart, Automatic, Compiler Assisted, User-driven Checkpointing, etc.

In-place restart (i.e., without disposing of non-failed processes) accelerates recovery, permits in-memory checkpoint

Naturally Fault Tolerant Applications, Master-Worker, Domain Decomposition, etc.

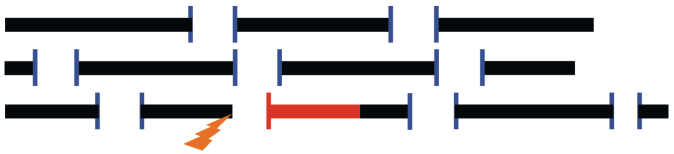
Application continues a simple communication pattern, ignoring failures



OpenSHMEM API for failure reporting, propagation and correction

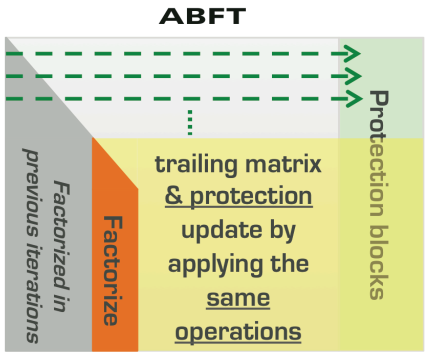
Uncoordinated Checkpoint/Restart, Transactional FT, Migration, Replication, etc.

ULFM makes these approaches portable across MPI implementations



Algorithm Fault Tolerance

ULFM allows for the deployment of ultra-scalable, algorithm specific FT techniques.



What we need is a set of flexible API that enable all these recovery strategies, w/o paying the cost of the most demanding

Content of the interface

- Error Reporting
- Error Propagation
- Error Correction

Error Reporting

- Main objective: prevent infinite blocking/abort on error
- Secondary objective: inform about reason

- Technical mechanism relies on Error Handler registration

Registering an error handler

```
typedef void (*shmem_errhandler_cb_fn)
                (int errcode, void* user_param);

void shmem_errhandler_set(
// IN: the managed error type
   int errcode,
// IN: the error handler function
   shmem_errhandler_cb_fn errh,
// IN: an user parameter to the callback
   void* user_param
);
```

- An user can set a different handler for different error types
- An user can provide an user specific “context” to the error handler callback through “user_param”
  - Useful to track state without global variables

Chaining error handlers

```
void shmem_errhandler_get(
// IN: the query error type
    int errcode,
// OUT: the error handler function
    shmem_errhandler_cb_fn errh,
// OUT: the parameter provided when registering the
callback
    void* user_param
);
```

- An user can get the previously set error handler and user_param
- An user can then set a new handler
- and restore the previous handler when done…
- or call the previous handler from the new handler

Error reporting, designs not retained

- Error codes returned from OpenSHMEM calls
  - Some SHMEM operations already have a return value
  - shmem_fadd, shmem_alloc…
  - Would break backward compatibility
  - Coding style is not elegant

- shmem_errno
  - Muddy semantic in multithreaded programs
  - Per-thread errno requires local thread storage
  - Coding style is not elegant

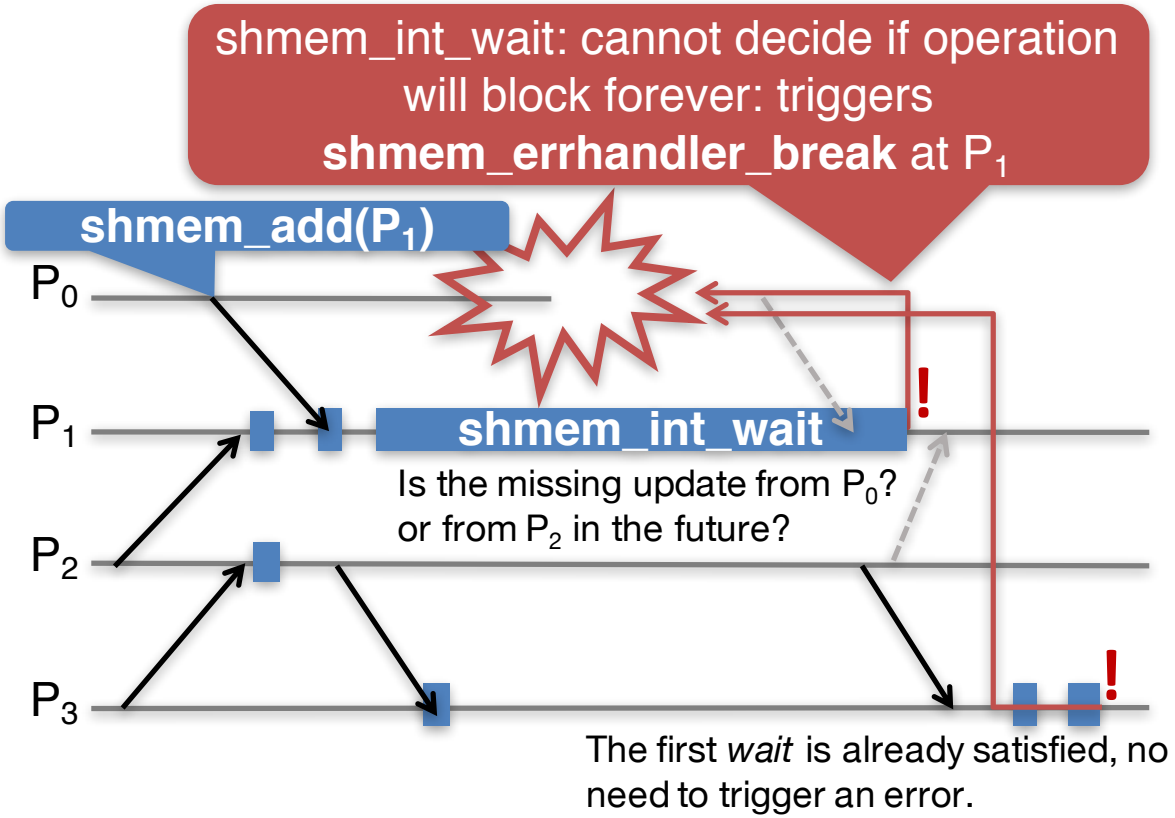
Default Error Handlers

`shmem_errhandler_gexit`	the error handler calls *shmem_global_exit* with the error code as parameter, which effectively terminates the application. This is the default error handler.
`shmem_errhandler_break`	the error handler breaks from blocking OpenSHMEM operations at the PE. It has no effect at other PEs.
`shmem_errhandler_gbreak`	the error handler breaks from blocking OpenSHMEM operations at all PEs.

Table 1. List of predefined error handlers in OpenSHMEM.

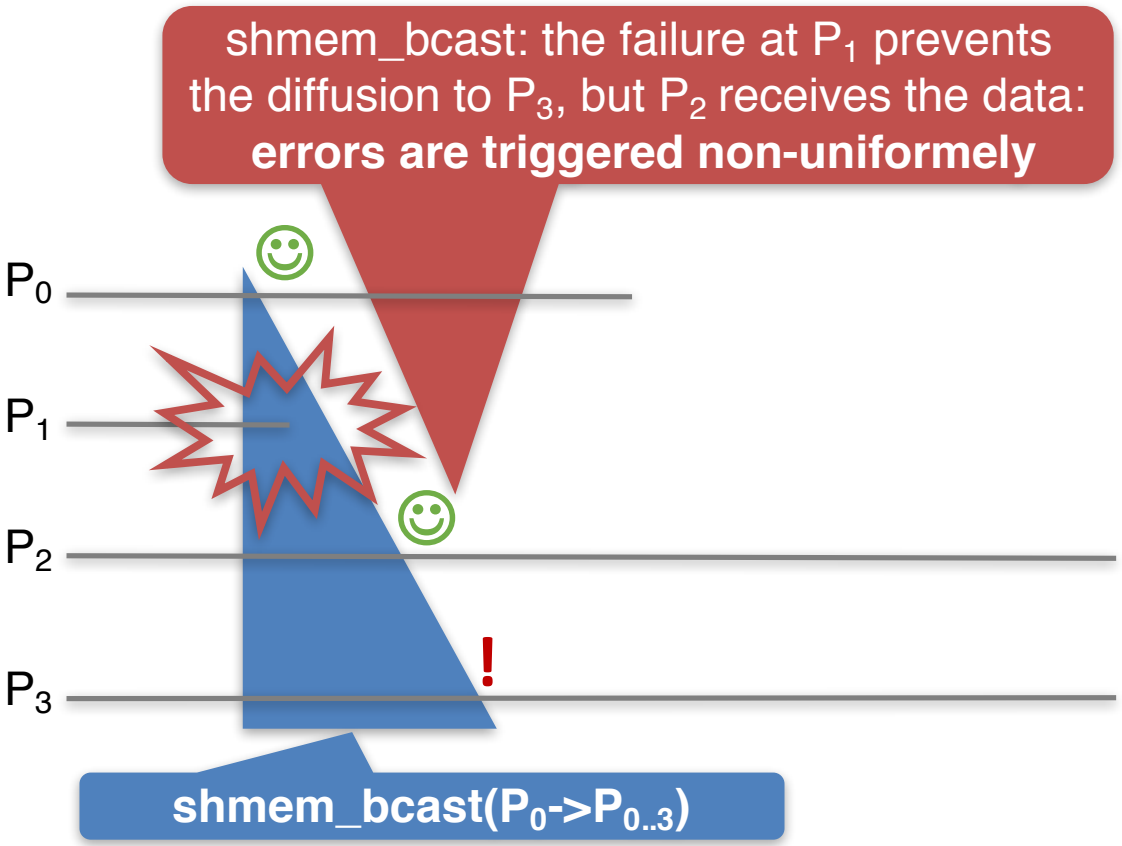
- Handler set by default is “errhandler_gexit”
- When a custom handler returns, it has the same effect as calling “errhandler_break”

Scope of Error Reporting



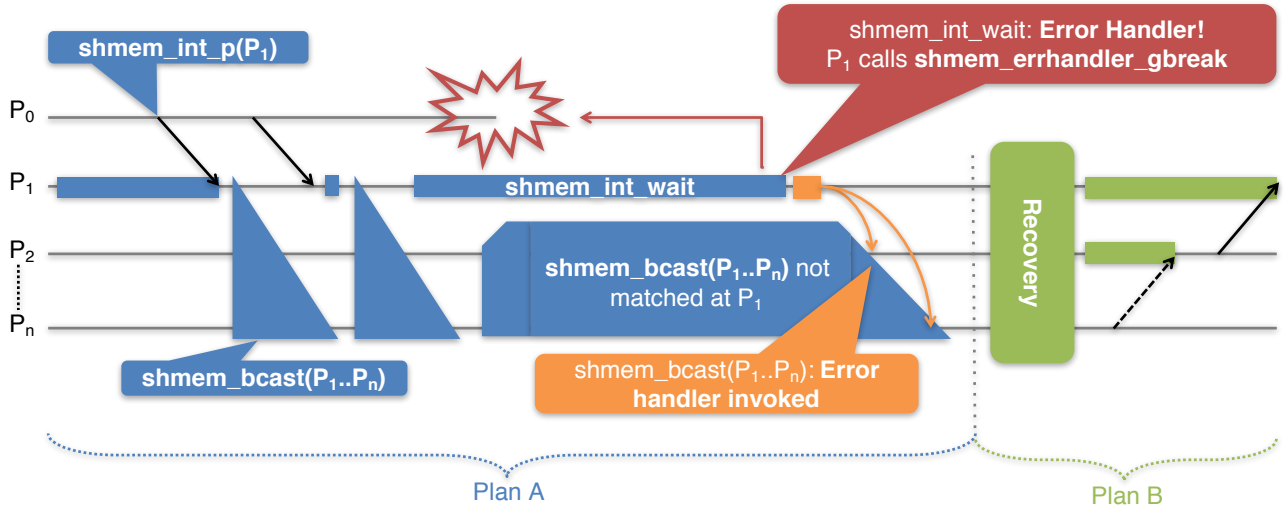
- When a non-local error **must** be reported at a PE?
  - Only when an OpenSHMEM operation may block
  - Relaxed semantic **preserves failure free performance** (no need to check for errors when things are doing fine)

Uniformity of Error Reporting



- It would be nice for collective operations (or even semantic synchronization) report errors uniformely
- But…
- Heavy performance penalty to be expected
- shmem_bcast is not synchronizing, adding uniform error reporting would force synchronizing
- Same for shmem_add
  - Compare to shmem_fadd performance!
- Even in synchronizing operations, errors that happen *during* the operation could be observed differently without adding **a fault tolerant agreement** to the operation

When Error Propagation is Needed




```
1  if(0==rank) {
2    shmem_int_p(&cond, 1, 1);
3    cond++;
4  } else {
5    if(1==rank) shmem_int_wait_until(&cond, comp++);
6    shmem_broadcast32(&comp, &comp, 1, 1, 1, 0, npes-1, psync);
7    /* (dest, src, count, root, PEstart, PEstride, PEsize, psync) */
8  }
```

Fig. 3. The transitive communication pattern *plan A*, from the source code, must be interrupted before the PEs can switch to the recovery communication pattern *plan B*. By calling the `shmem_errhandler_gbreak` error handler, P_1 ensures that all possibly unmatched operations in *plan A*, which could provoke deadlocks, are interrupted.

Error Propagation

`shmem_errhandler_gexit`	the error handler calls *shmem_global_exit* with the error code as parameter, which effectively terminates the application. This is the default error handler.
`shmem_errhandler_break`	the error handler breaks from blocking OpenSHMEM operations at the PE. It has no effect at other PEs.
`shmem_errhandler_gbreak`	the error handler breaks from blocking OpenSHMEM operations at all PEs.

Table 1. List of predefined error handlers in OpenSHMEM.

- "errhandler_gbreak" permits propagating an error at all PEs
- Depending on use-case, end-user can propagate (or not) errors
  - For example, errors on slaves in master-slave workloads do not require propagation
  - Some errors can be repaired locally (typically resource errors)

Propagation: Implementation Challenges

- Progress “gbreak” notifications
  - Can be managed by active messages if hardware support present
  - Asynchronous state machine
  - Can be managed by progress thread (at worse)

- Interrupting blocking calls
  - Interrupting operations is challenging if OpenSHMEM posts blocking transport calls
  - Either have capability to cancel blocked calls (from AM callback or progress thread)
  - Or asynchronous progress engine
  - Or timeout based blocking call
  - Some calls do not need to be interrupted :) (put, for example)

  To preserve performance, gbreak notifications can be checked only *after* unsuccessfully trying to progress normal operations (thanks to no ordering w/application messages)

- UCX-OpenSHMEM reference implementation: all calls asynchronous, and active messages

Post-error Status

```
void shmem_error_query(
// IN: PE whose status to query
   int pe,
// OUT: PE status (0: PE in good condition)
   int* code);
```

- After an error handler is called, OpenSHMEM may not be able to communicate anymore
- Querying for pe==mype tells the application if the current PE can still communicate with OpenSHMEM
- Local operation: does not synchronize the status of other PEs implicitly

Post-error Stabilization

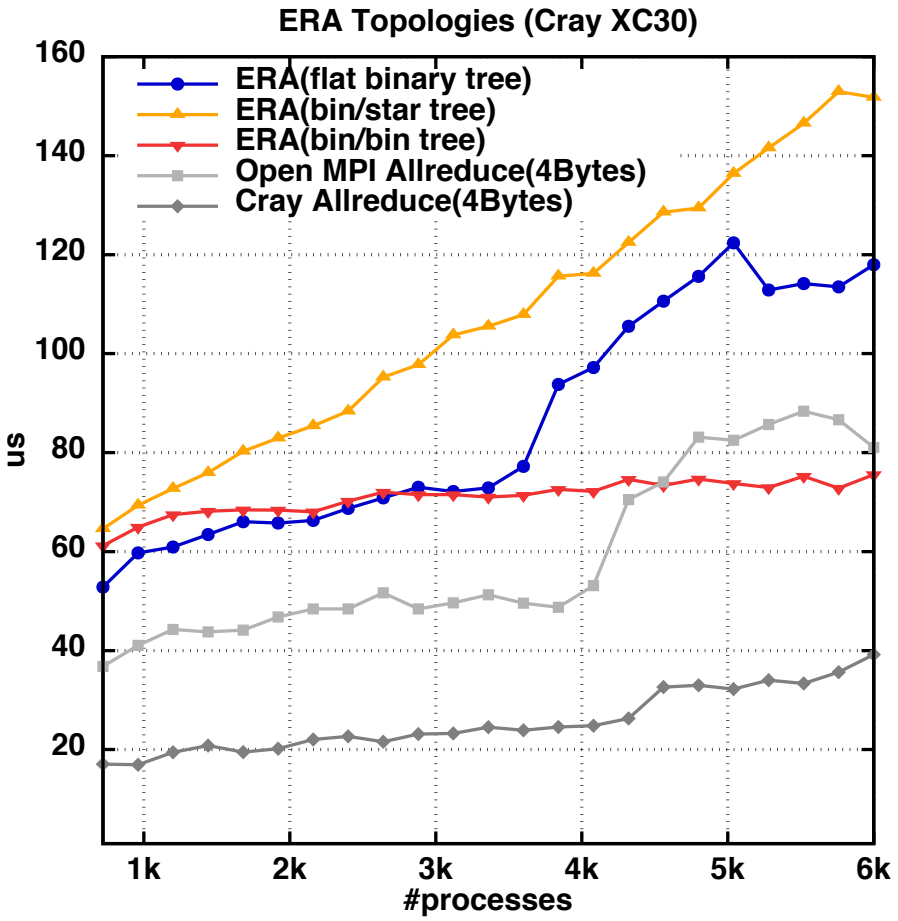
```
void shmem_error_barrier_all(void);
```

collective, synchronizing, fault-tolerant

- How can one resume communicating with OpenSHMEM after an error?
- Problem 1: when have all errors for the current “epoch” been captured?
  - Error_barrier_all garantees that all gbreak notifications issued before the call at any PE are triggered before the call completes at the local PE (gbreak flush)
  - The error handler may be triggered withing error_barrier_all, without interrupting the operation
- Problem 2: how can one synchronize which PE can still communicate?
  - After Error_barrier_all, shmem_error_query is synchronized
- Problem 3: collective communications
  - After Error_barrier_all, collective communication are collective on the participating PEs for which error_query reports a status of 0

Implementation Effort

- In OpenSHMEM-UCX
- “gbreak” based on conversion from ULFM-MPI “Revoke” operation
- “error_barrier_all” based on conversion from ULFM-MPI “Agree” operation
- Algorithms demonstrated scalable
- Former implementation based on BTL bears similarities with UCX



Agreement performance in MPI-ULFM
Compares with 2x Cray Allreduce latency at scale!

Concluding Remarks

- Error handling framework for OpenSHMEM
  - Notification
  - Propagation
  - Correction
- WIP: implentation
- Future works: process crash **replacement** interface

This work has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy.

Participate!

- Discussion on ticket #195 in Redmine
  - http://www.openshmem.org/redmine/issues/195
- Spec 1.3 latexdiff for the proposed changes
  - http://www.openshmem.org/redmine/attachments/download/195/main_spec.pdf
  - changes are located in 2.8, 4.3, error code table in section 6, 8.1.12, 8.1.13, 8.5.1, 8.5.6 (note that the example need to be updated in the too later instances).
- If you prefer a git diff, please look at
  - https://github.com/abouteiller/specification/pull/1

Livelock problem

- `while(shared_var != somevalue){…}`

1. this is an incorrect error managing code.
   `shmem_wait_int(shared_var, somevalue)` is correct.

A special problem w/ fetch&op

```
 \begin{Csynopsis}
-int shmem_int_fadd(int *dest, int value, int pe);
-long shmem_long_fadd(long *dest, long value, int pe);
-long long shmem_longlong_fadd(long long *dest, long long value,
 int pe);
 \end{Csynopsis}

 \begin{Fsynopsis}
-INTEGER pe
-INTEGER*4 SHMEM_INT4_FADD, ires_i4, value_i4
-ires_i4 = SHMEM_INT4_FADD(dest, value_i4, pe)
-INTEGER*8 SHMEM_INT8_FADD, ires_i8, value_i8
-ires_i8 = SHMEM_INT8_FADD(dest, value_i8, pe)
 \end{Fsynopsis}
```

```
 \begin{Csynopsis}
+int shmem_int_fadd(int *dest, int value, int *old, int pe);
+int shmem_long_fadd(long *dest, long value, long *old, int pe);
+int shmem_longlong_fadd(long long *dest, long long value, long
 long *old, int pe);
 \end{Csynopsis}

 \begin{Fsynopsis}
+INTEGER pe, errcode
+INTEGER*4 old_i4, value_i4
+errcode = SHMEM_INT4_FADD(dest, value_i4, old_i4, pe)
+INTEGER*8 old_i8, value_i8
+errcode = SHMEM_INT8_FADD(dest, value_i8, old_i8, pe)
 \end{Fsynopsis}
```

```
+\apiargument{OUT}{old}{The value stored at \VAR{dest} address
 on the remote \ac{PE}
+    prior to the atomic operation. The type of \VAR{old} should
 match that implied in
+    the SYNOPSIS section.}
```

```
    \VAR{value} to \VAR{dest} on \VAR{pe} and return the
previous contents of
-    \VAR{dest} as an atomic operation.
```

```
     \VAR{value} to \VAR{dest} on \VAR{pe} and return the
 previous contents of
+    \VAR{dest} as an atomic operation.
+    The contents that had been at the \VAR{dest} address on the
 remote \ac{PE}
+    prior to the atomic addition operation is available in
 \VAR{old}.
```