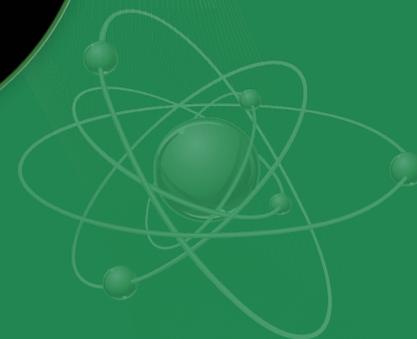
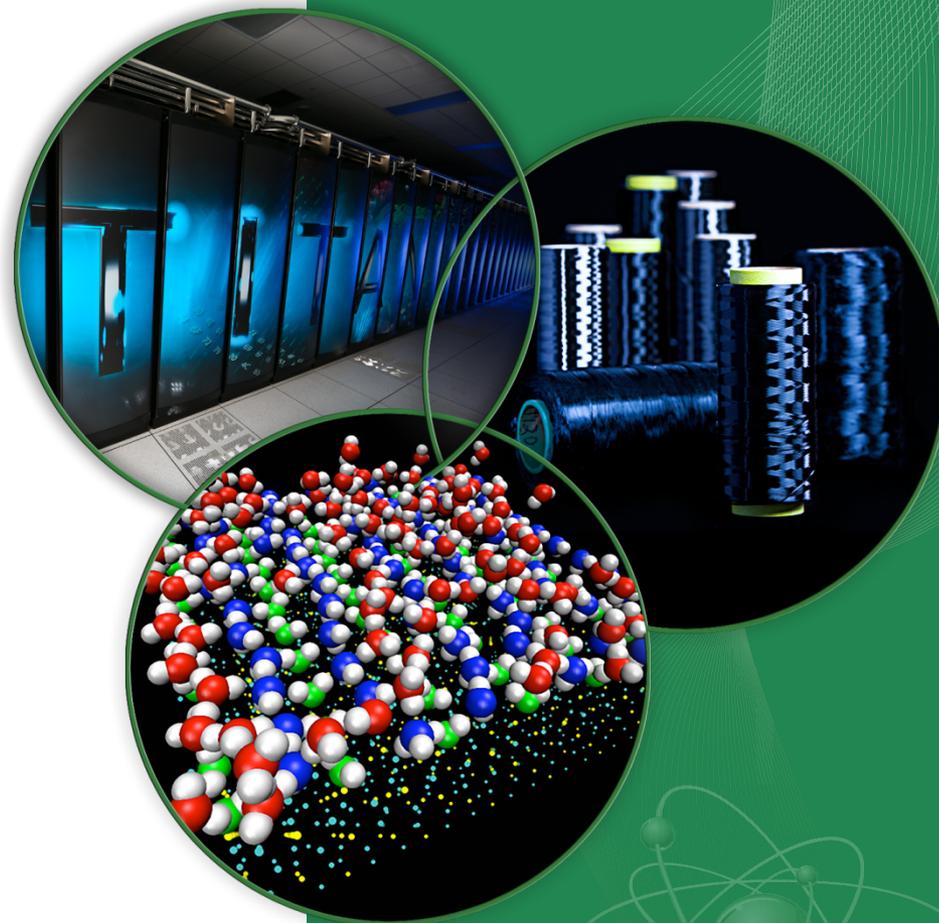


OpenSHMEM 1.3

OpenSHMEM Specification Face-to- face community discussion

OpenSHMEM Workshop 2015

August 6th , 2015



Welcome

Steve Poole

OpenSHMEM Workshop 2015 – Face-to-face Specification discussion

Progress update for 1.3 Specification

- Tentative Schedule
 - Features freeze - August
 - Specification draft - September
 - Final draft – October
 - Ratification – November
 - Release 1.3 – SC 2015

#160 – Deprecated cache management

- Formally deprecate all cache management routines
 - Not used on current architectures
 - If they become necessary in the future, we would rather redesign from scratch than patch up the old ones
- July telecon: unanimous vote to proceed
- Working draft is present in the spec (posted to Redmine)

#161 – C/C++ library constants

- Deprecate leading underscore for C/C++ SHMEM constants
- Also specify that these are compile-time constants
 - Avoids possible namespace issues
- July telecon: unanimous vote to proceed
- Working draft is present in the spec (posted to Redmine)

#166 – Symmetric heap text

- Remove text that specifies symmetric objects have the “same offset” from an arbitrary global address on each PE
- July telecon: unanimous vote to proceed
- Working draft is present in the spec (posted to Redmine)

#167 – Non-character types

- Remove text from data type descriptions that specifies “non-character” types and explicitly require proper alignment
- Other issues (e.g. aliasing) and further improvements can be made by adding const, restrict, volatile throughout the spec
 - Separated into new ticket #181
- July telecon: unanimous vote to proceed
- Working draft is present in the spec (posted to Redmine)

#171 – PEs as threads

- Remove text specifying that PEs may be implemented as OS threads, and leave unspecified to allow for future flexibility
- July telecon: unanimous vote to proceed
- Working draft is present in the spec (posted to Redmine)

#124 – Fence/quiet with PE argument

- Proposal to have a fence/quiet that take a PE argument, rather than affecting all PEs
 - Need motivating cases
 - Unknown performance implications
- July telecon: 2-yes, 2-yes (fence only), 1-no, 3-abstain
- Intel is leading this ticket and further investigations

#165 – Atomicity vs. datatype

- Insert underlined text into "These routines guarantee that accesses by OpenSHMEM's atomic operations with the same datatype will be exclusive".
- Mixing atomics on *signed* integers is dangerous
 - OpenSHMEM currently only has atomics on signed types
 - If we want this capability, should define atomics for unsigned types and define compatibility classes
- July telecon: 4-yes, 2-no, 2-abstain
- Intel is leading this ticket

#181 – const/restrict/volatile

- Proposal to add “const,” “restrict,” and “volatile” keywords to C API everywhere that is appropriate
- July telecon: Not discussed in detail

#182/183 – `alltoall()` / `alltoallv()`

- Current proposal follows the MPI / Cray SHMEM interface
- There may be some SHMEM-specific issues due to memory model, etc.

- July telecon: not discussed in detail

#113: Non-blocking RMA Operations

#113: Non-blocking RMA Operations

- Non-blocking Put (Get) operation starts the Put (Get) operation, and is completed by another operation.
- Explicit Non-blocking Operation: Every operation has a handle to identify the operation
- Implicit Non-blocking Operation: The operation is not identified by a handle by the user.

Proposal : Non-blocking RMA Operations

- Only supports implicit non-blocking operations
- Example API:
 - `void shmem_TYPE_put_nbi(int *dest, const int *source, size_t nelems, int pe);`
 - `void shmem_TYPE_get_nbi(int *dest, const int *source, size_t nelems, int pe);`

Non-blocking Operations : Ordering and Completion

- Completion:
- `shmem_quiet` completes all non-blocking implicit operations
- `shmem_barrier/all` completes all non-blocking RMA operations
- `shmem_finalize` completes all non-blocking RMA operations
- Ordering:
- `shmem_fence` does not order the non-blocking `shmem_put_nbi` operations

The following example uses `shmem_logical_iget` in a Fortran program.

```

PROGRAM STRIDELOGICAL
INCLUDE "shmem.fh"

LOGICAL SOURCE(10), DEST(5)
SAVE SOURCE ! SAVE MAKES IT REMOTELY ACCESSIBLE
DATA SOURCE /.T.,.F.,.T.,.F.,.T.,.F.,.T.,.F.,.T.,.F./
DATA DEST / 5*.F. /
CALL SHMEM_INIT()
IF (SHMEM_MY_PE() .EQ. 0) THEN
  CALL SHMEM_LOGICAL_IGET(DEST, SOURCE, 1, 2, 5, 1)
  PRINT*, 'DEST AFTER SHMEM_LOGICAL_IGET:', DEST
ENDIF
CALL SHMEM_BARRIER_ALL

```

8.4 Nonblocking Remote Memory Access Routines

8.4.1 SHMEM_PUT_NBI

The nonblocking put routines provide a method for copying data from a contiguous local data object to a data object on a specified PE.

SYNOPSIS

C/C++:

```

void shmem_double_put_nbi(double *dest, const double *source, size_t nelems, int pe);
void shmem_float_put_nbi(float *dest, const float *source, size_t nelems, int pe);
void shmem_int_put_nbi(int *dest, const int *source, size_t nelems, int pe);
void shmem_long_put_nbi(long *dest, const long *source, size_t nelems, int pe);
void shmem_longdouble_put_nbi(long double *dest, const long double *source, size_t nelems,
int pe);
void shmem_longlong_put_nbi(long long *dest, const long long *source, size_t nelems, int pe);
void shmem_put32_nbi(void *dest, const void *source, size_t nelems, int pe);
void shmem_put64_nbi(void *dest, const void *source, size_t nelems, int pe);
void shmem_put128_nbi(void *dest, const void *source, size_t nelems, int pe);
void shmem_putmem_nbi(void *dest, const void *source, size_t nelems, int pe);
void shmem_short_put_nbi(short *dest, const short *source, size_t nelems, int pe);

```

FORTRAN:

```

CALL SHMEM_CHARACTER_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_COMPLEX_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_DOUBLE_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_INTEGER_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_LOGICAL_PUT_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT4_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT8_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT32_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT64_NBI(dest, source, nelems, pe)
CALL SHMEM_PUT128_NBI(dest, source, nelems, pe)
CALL SHMEM_PUTMEM_NBI(dest, source, nelems, pe)
CALL SHMEM_REAL_PUT_NBI(dest, source, nelems, pe)

```

DESCRIPTION

Arguments

IN

dest

Data object to be updated on the remote PE. This data object must be remotely accessible.

Teams #179

- Proposed API

- SHMEMX_WORLD_TEAM, SHMEMX_NULL_TEAM
- `shmemx_create_strided_team(long PE_start, long PE_stride, long PE_size, shmemx_team parent, shmemx_team *subteam);`
- `shmemx_team_2d_split(long xrange, long yrange, shmemx_team parent, shmemx_team *xaxis, shmemx_team *yaxis);`
- `shmemx_team_3d_split(long xrange, long yrange, long zrange, shmemx_team parent, shmemx_team *xaxis, shmemx_team *yaxis, shmemx_team *zaxis);`
- `shmemx_team_axial_split(long dimensions, long *range, shmemx_team parent, shmemx_team **axes);`
- `shmemx_team_translate_index(int PE, shmemx_team src, shmemx_team dst);`
- `shmemx_team_size(shmemx_team team);`
- `shmemx_destroy_team(shmemx_team team);`

Teams – axial split

```
1 shmemx_team blue_team , yellow_team ;  
2 shmemx_team_2d_split (4 , 4 ,  
    SHMEMX_WORLD_TEAM, &blue_team ,  
    &yellow_team ) ;
```

Listing 2: 2-D Axial Split

	t5	t6	t7	t8
t1	0	1	2	3
t2	4	5	6	7
t3	8	9	10	11
t4	12	13	14	15

Figure 3: Visual Representation of 2-D Axial Split in Listing 2.

Open Questions

- Passing teams to RMA/AMO operations:
 - `shmemx_team_translate_index()`
 - API extension with C11 generics
- Passing teams to Collectives
 - API extension with C11 generics
 - Team can be associated with a negative integer value (ID) and then the ID can be passed as an argument instead of “int PE_start”
- Do we need more “complex” teams ?
- Removing pSync ?
 - It means that `shmemx_create_strided_team` has to be collective operation that coordinates the team creation across PEs
 - Today team’s creation is a local operation (a.k.a very fast)
 - But, Psync allocation is a collective operation (`shmalloc`) unless it is global

Proposal for the path forward

- Coordination of the efforts between ORNL, Cray, and Intel proposals
- Split the team feature to a phases:
 - Phase-1: Adding support for Team management operations and re-indexing
 - Support only for one-sided operations
 - Phase-2: Adding support for collective operations
 - Using C11 to extend the list of parameters
 - Removing pwork/psync
 - Phase-3: Teams extension
 - Intel's federation
 - Memory “spaces” ?
- Coordination with other proposals
 - Thread context ?

Lock + Teams

- The question was raised in #115
- Do we care ?

Atomics (AMO) Extension (#172)

- Bitwise AMOs (borrowed from OpenSHMEM RF+IBM Zurich)
 - void shmem_int_xor (int *target, int value, int pe);
 - void shmem_long_xor (long *target, long value, int pe);
 - void shmem_longlong_xor (long long *target, long long value, int pe);
- Masked AMOs (borrowed from Quadrics SHMEM)
 - int shmem_int_mswap(int *target, int mask, int value, int pe);
 - long shmem_long_mswap(long *target, long mask, long value, int pe);
 - long shmem_longlong_mswap(long *target, longlong mask, long value, int pe);
- Main Question – do we have these capabilities in hardware

Simplifying the OpenSHMEM API via C11 Generic Selection

Background

- The OpenSHMEM (and historical SHMEM) RMA and AMO interfaces are verbose and type-specific due to C's lack of function overloading.
- C11 adds generic selection, which enables type-generic interfaces (much like C99's type-generic math functions)

Proposal

- // RMA Interface
- void shmem_put(TYPE *dst, const TYPE *src, size_t len, int pe);
- void shmem_get(TYPE *dst, const TYPE *src, size_t len, int pe);
- void shmem_iput(TYPE *dst, const TYPE *src, ptrdiff_t dst, ptrdiff_t sst size_t nelems, int pe);
- void shmem_iget(TYPE *dst, const TYPE *src, ptrdiff_t dst, ptrdiff_t sst size_t nelems, int pe);
- void shmem_p(TYPE *dst, TYPE val, int pe); TYPE
- shmem_g(TYPE *dst, int pe);

Proposal

// AMO Interface

- void shmem_add(TYPE *dst, TYPE val, int pe); TYPE
- shmem_fadd(TYPE *dst, TYPE val, int pe);
- void shmem_inc(TYPE *dst, int pe);
- TYPE shmem_finc(TYPE *dst, int pe);
- TYPE shmem_swap(TYPE *dst, TYPE val, int pe);
- TYPE shmem_cswap(TYPE *dst, TYPE cond, TYPE val, int pe);

Notes

- Generic selection in C11 is sufficiently robust as to support the typed RMA interfaces. (All AMO interfaces are typed.)
- The non-typed `shmem_(put|get)(32|64)` interfaces are used to support block-based put and get operations on `uint32_t` and `uint64_t` data types. N.B., `int32_t` and `int64_t` are not required as they are equivalent types in C to `int` and `long`.
- As implemented, generic selection does not support RMA operations on user-defined data types; e.g., `my_struct_t`. Developers would need to call the appropriate `shmem_put(32|64|128|mem)` operation.
- No proposed Fortran changes.

Limitations

- Requires compiler support for C11 generics
 - Available in GCC 4.9+, Clang 3.0+, PGI C/C++ 2015
 - Not currently supported by Intel C/C++ compiler
- However, the non-generic interface is still available to those without modern compilers

Generics and API Philosophy

- As proposed here, C11 Generics are a convenient, yet relatively modest change to the API.
- However, their use can be expanded to maintain a clean, simple OpenSHMEM API in the face of many extension proposals.
- C11 Generics can be used to overload:
 - `shmem_broadcast(32|64)` for both the existing API and (an API similar to) the Teams proposal
 - `shmem_put(...)` for this proposed generic API and the communication contexts proposal

Generics and API Philosophy

- Extensive use of C11 generics can help maintain a simple, user-facing API that provides, but designing future APIs should consider the impact on generic selection.
- Specifically:
 - Generic selection favors (or requires) generic selection on left-most function arguments for differentiating functions with signatures of different argument lengths.
 - For example, this favors the `shmem_team_t` and `shmem_ctx_t` arguments as the first argument to the function.

References

- OpenSHMEMRedmine
 - Issue #178 (Generic RMA and AMO):
<http://bongo.cs.uh.edu/redmine/issues/178>
 - Issue #177 (Communication Contexts):
<http://bongo.cs.uh.edu/redmine/issues/177#note-3>
 - Issue #179 (Teams API):
<http://bongo.cs.uh.edu/redmine/issues/179#note-2>
- (Incomplete) prototype implementation as shmex_* extensions:
<https://github.com/nspark/shmex/blob/master/include/shmex.h>

Future Extensions: Fault Tolerance

- Aurelien from UTK

Future Extensions

- Signaling Put (Cray)
- Thread Model (Cray)
- Locality (Cray)
- Teams (ORNL/Cray)
- Communication Contexts (Intel)
- PE locks within subgroups (ORNL ?)
- Nonblocking Quiet/Fence
- shmem_iputmem/shmem_igetmem (UH)
- Bitwise atomic operations (ORNL)

Acknowledgements



This work was supported by the United States Department of Defense & used resources at Oak Ridge National Laboratory.

Questions?



 **OAK
RIDGE**
National Laboratory

#124: Fence/Quiet With PE Argument

Issue: Fence/Quiet affect all peers

Summary: This ticket proposes that we consider adding fence and quiet functions that take a PE argument. These functions would guarantee ordering and remote completion with respect to the PE argument.

- `void shmem_fence_pe(int pe);`
- `void shmem_quiet_pe(int pe);`

Pro: Improves the model (code readability), could improve performance (affect some, but not all traffic)

Con: Need motivating cases of where this would improve performance before integrating the change

#165: Atomicity Versus Op and Datatype

Issue: Atomicity requirements are too loose and may impact performance

Summary: Presently, all SHMEM atomic operations are atomic with respect to all other atomic operations, regardless of operation performed and datatype. This requires that all atomics be implemented through the same mechanism (i.e. HW offload or active message).

Proposed Change: Consider restricting this to provide more flexibility to implementors (and consequently, better performance to users).

- Sec 4.2: Insert underlined text into "These routines guarantee that accesses by OpenSHMEM's atomic operations with the same datatype will be exclusive".

History: This proposal has already had some discussion on #164

- Summary: mixing atomics on *signed* integers of different lengths is dangerous, but does anyone do this?
 - If we want this functionality, we need to do it right (with unsigned types) rather than as a hack with ugly semantics.
 - In this case, we would add "with compatible datatypes" instead of "same" and define compatibility classes

Fault Tolerance API for OpenSHMEM

Aurelien Bouteiller
OpenSHMEM Workshop,
Annapolis, MD, 2015 August 6



Situation Today



Fault tolerance becomes critical at Petascale (MTTI \leq 1day)
Poor fault tolerance design may lead to huge overhead

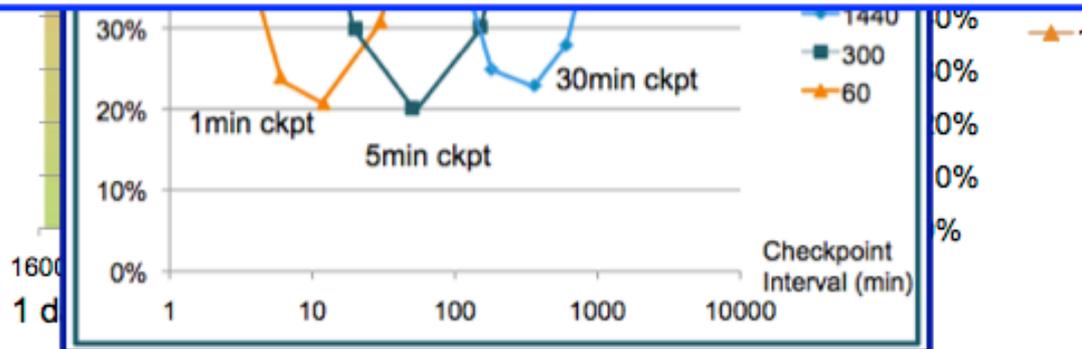
Overhead of checkpoint/restart

Cost of non optimal checkpoint intervals:

100%
0%

Today, 20% or more of the computing capacity in a large high-performance computing system is wasted due to failures and recoveries.

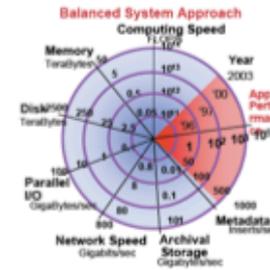
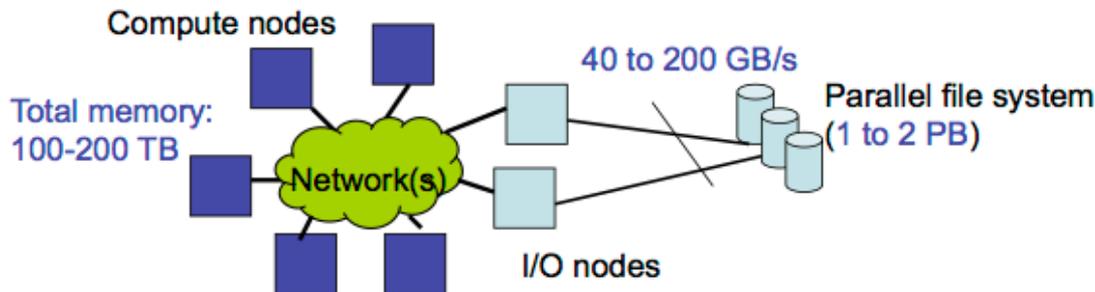
Dr. E.N. (Mootaz) Elnozahy et al. *System Resilience at Extreme Scale*, DARPA



Checkpoint Restart

Classic approach for FT: Checkpoint-Restart

Typical “Balanced Architecture” for PetaScale Computers



TACC RoadRunner



LLNL BG/L

➔ Without optimization, Checkpoint-Restart needs about 1h! (~30 minutes each)

Systems	Perf.	Ckpt time	Source
RoadRunner	1PF	~20 min.	Panasas
LLNL BG/L	500 TF	>20 min.	LLNL
LLNL Zeus	11TF	26 min.	LLNL
YYY BG/P	100 TF	~30 min.	YYY

The FT methods landscape

Checkpointing
& Restart (C/R)

Diskless
Checkpointing

Algorithm Based
Fault Tolerance
(ABFT)

Overhead

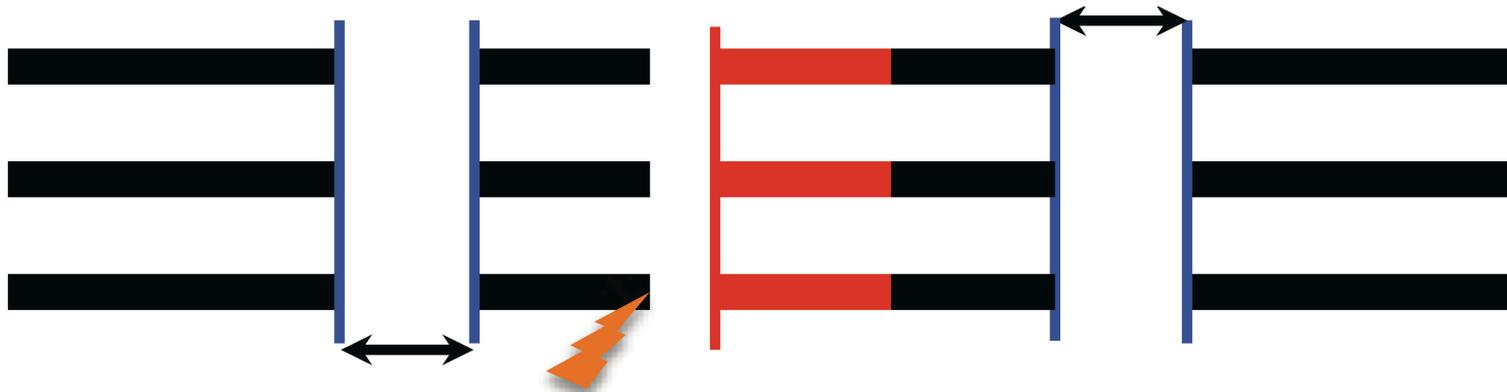
Small

Application Specificity

Small

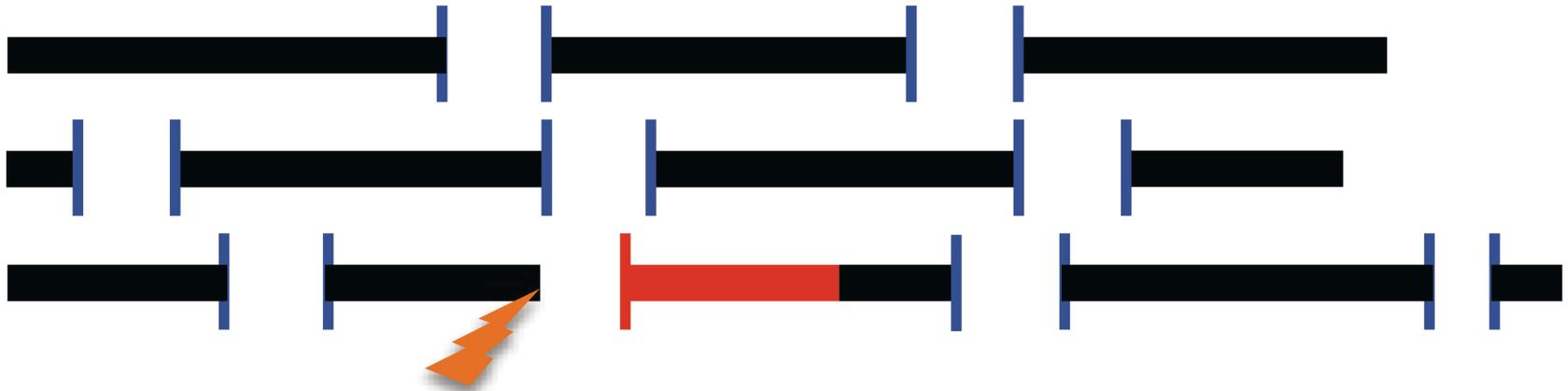
Backward recovery: C/R

Coordinated checkpoint (possibly with incremental checkpoints)



- Coordinated checkpoint is the workhorse of FT today
 - I/O intensive, significant failure free overhead ☹️
 - Full rollback (1 fails, all rollback) ☹️
 - Can be deployed w/o MPI support 😊
- **We need to enable: deployment of in-memory, Buddy-checkpoints, Diskless checkpoint**
 - Checkpoints stored on other compute nodes
 - No I/O activity (or greatly reduced), full network bandwidth
 - Potential for a large reduction in failure free overhead, better restart speed

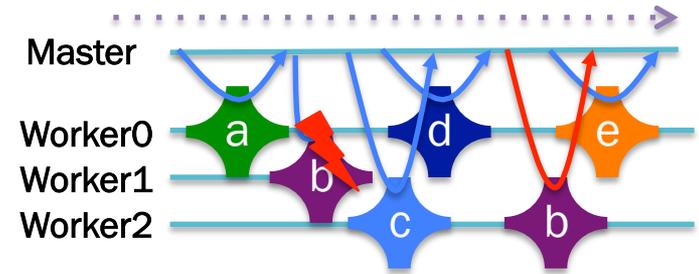
Uncoordinated C/R



- Checkpoints taken independently
- Based on variants of Message Logging
- 1 fails, 1 rollback
- Can be implemented w/o a standardized user API
- Benefit from FT support in spec: **implementation becomes portable across multiple Oshmem libraries**

Forward Recovery

- Forward Recovery: Any technique that permit the application to continue without rollback
 - Master-Worker with simple resubmission
 - Iterative methods, Naturally fault tolerant algorithms
 - Algorithm Based Fault Tolerance
 - Replication (the only system level Forward Recovery)
- No checkpoint I/O overhead
- No rollback, no loss of completed work
- May require (sometime expensive, like replicates) protection/recovery operations, but still generally more scalable than checkpoint 😊
- Often requires in-depths algorithm rewrite (in contrast to automatic system based C/R) ☹️



With ULFM FT-MPI:
PDE, domain decomposition

Applications

HemeLB

Lattice Boltzmann Flow Solver
University College London

Processor fails

- Re-initialize substitute processor with average mass flow, velocity from neighbors

passable error in domain size and magnitude if real solution sufficiently smooth

4/11/2013 Fault Tolerance in MPI | EASC 2013 | sochi@cray.com

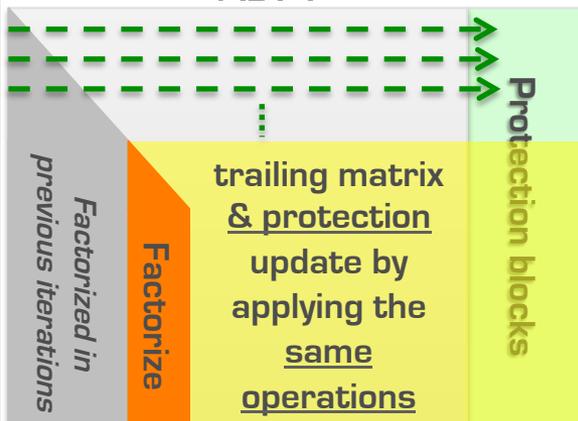
CREST

Application specific forward recovery

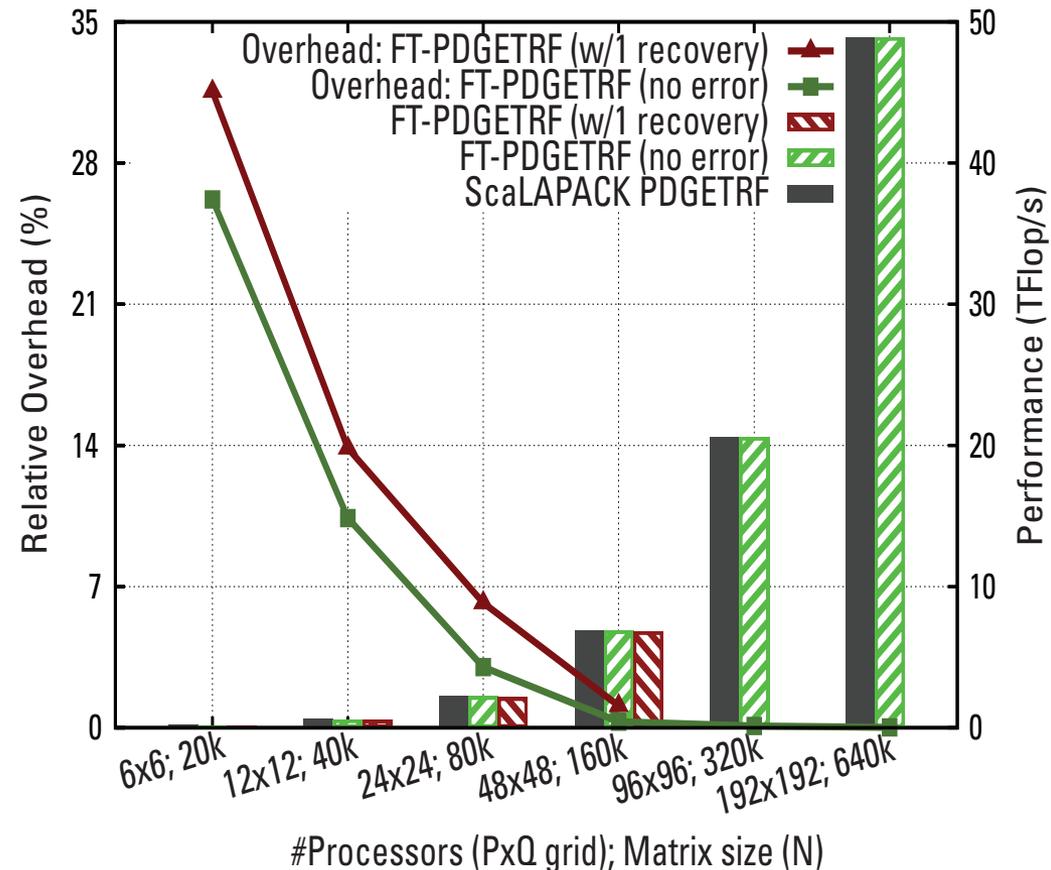
- Algorithm specific FT methods

- Not General, but...
- Very scalable, low overhead 😊
- Can't be deployed w/o FT-communication library*

ABFT



With ULFM FT-MPI



An API for diverse FT approaches

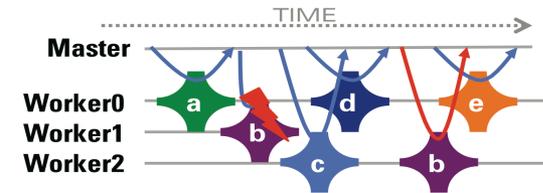
Coordinated Checkpoint/Restart, Automatic, Compiler Assisted, User-driven Checkpointing, etc.

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



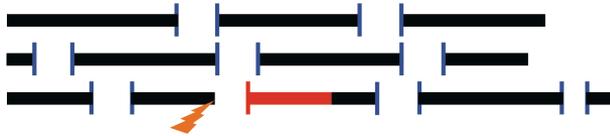
Naturally Fault Tolerant Applications, Master-Worker, Domain Decomposition, etc.

Application continues a simple communication pattern, ignoring failures



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

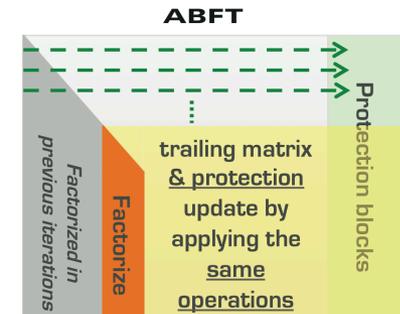
ULFM makes these approaches portable across MPI implementations



OpenSHMEM API for failure reporting, propagation and correction

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

FT MPI vs FT Oshmem

- You've been there, done that in MPI, so...
problem solved?
 - MPI already has a defined error reporting framework (error handlers)
 - MPI has well defined scopes (communicators, etc)
 - MPI has no progress guarantee
 - MPI has a lot of 2-sided operations
 - MPI 2-sided FT spec is complete, 1-sided spec is still cooking
 - MPI 1-sided synchronizations are different from oshmem synchronizations, uniformity with 2-sided also a factor, "MPIness" of proposed concepts important
- We believe the FT spec for Oshmem should look significantly different than for MPI

Error Reporting

- New concept: “crash_handler”
- `shmem_register_crash_handler (shmem_crash_handler_fn_t hdlr)`
 - Register a crash handler to be invoked asynchronously when shmem detects that a process (any process) has crashed.
- `shmem_crash_handler_fn_t (int npes, int* pes)`
 - Invoked only once with a particular process in the array “pes”
 - Async invocation, at any time (like a signal)
 - Invocation is not ordered with synchronizations across Pes: PE0: crash_handler before barrier_all(), PE1: crash_handler after barrier_all, this is legal

Regaining control

- Worst situation is deadlock/livelock from waiting for dead processes to “do something”
- Default crash handlers can return control:
 - `shmem_crash_handler_abort(int errcode)`: aborts the execution at all PEs
 - `shmem_crash_handler_break(int errcode)`: interrupts all blocking `shmem` calls. Writes "errcode" into the global variable `shmem_error`. When an operation is interrupted, the target buffers are undefined, and flushing/quieting semantics are not guaranteed.
 - `shmem_crash_handler_report(int errcode)`: write into the global variable `shmem_error`, does not interrupt any operations (application may deadlock if the user code posts blocking `shmem` operations, so users are responsible for changing locally the target values for variables appearing in `shmem_wait_int` and similar).
 - **User provided handlers**: users can provide their own failure handler, the failure handler must be reentrant. The failure handler can call one of the default failure handlers, as the last instruction of the handler.

Livelock problem

- `while (shared_var != somevalue) {...}`
- **Two approaches are possible:**
 1. this is incorrect code. `shmem_wait_int(shared_var, somevalue)` is correct.
 2. a failure handler may be called at any moment, and `while (shared_var != somevalue) {if (shmem_error) break }` is then correct.

Knowing who is still alive

- Local semantic (may return true at PE1 and false at PE2)
- `bool shmем_pe_accessible(pe)`: added semantic: returns false after the `crash_handler` has been invoked for 'pe'. The crash handler may be invoked from inside `shmем_pe_accessible`.
- Or `bool shmем_pe_crashed(pe)`: same as above, does not overload accessibility (it was accessible, it remains accessible but crashed).

Validating progress/reconciliation

- `shmem_ft_barrier_all()`: Same semantic as `barrier_all`. In addition
 - all processes are guaranteed to have invoked the same set of crash handlers.
 - Crash handlers cannot interrupt `shmem_ft_barrier_all`, which continues to have flushing semantics.
- FT Barrier
 - so one can change algorithmic phase knowing previous phase succeeded or not)
- Reconciliate disparate view of failures across Pes
 - All Pes know the same failure set, can undergo some collective decision about what to do next.

Repairing: tentative ideas

- `shmem_ftshrink()`
 - collective, compacts the pe "world" by excluding all dead pes (the list of dead pes is agreed upon during the operation).
 - The value returned by `shmem_my_pe` and `num_pes` change.
 - Requires users to *rebalance the work/recompute communication structure* etc. ☹️
- `shmem_ftblank()`
 - collective, agrees on a list of dead pes. Operations targeting these pes are NOPs.
 - *may have a severe performance impact on collective operations* (even w/o failures).
- `shmem_ftreplace()`
 - collective, agrees on a list of dead pes. Replacement PEs are spawned to replace the dead.
 - All surviving Pes keep their former rank.
 - New spawned PEs receive control from `shmem_init()`.
 - Additional interface `shmem_pe_my_incarnation()` returns a count of how many times that PE has been spawned (so 1 means no failures).
 - Issue: *how are symmetric memory segments recreated/accessed from restarted Pes?*
Stringent issue is when some segment has been free of realloc since its creation.

Concluding remarks

- Nothing carved in stone, at this point
 - Would like to receive input and participations from other interested parties!
 - In particular C/R framework designers
- How do we interoperate with ongoing efforts (teams, thread safety, for example)?
- Looking for community input!