Using ULFM for implementing Fault Tolerant Applications

Fault Tolerant MPI Applications with ULFM BoF

Fault Tolerant PDE Applications
Md Mohsin Ali¹
(md.ali@anu.edu.au)
Peter E Strazdins¹

Resilient X10 over MPI
Sara S. Hamouda¹
(sara.salem@anu.edu.au)
Benjamin Herta²
Josh Milthorpe¹,²
David Grove²
Olivier Tardieu²

¹Australian National University
²IBM T. J. Watson Research Center
Application Level Fault Tolerance using the Sparse Grid Combination Technique

Md Mohsin Ali, Peter E Strazdins
Australian National University
Our Success with ULFM MPI

• Provided fault tolerance support for three PDE based applications:
  – GENE Gyrokinetic Plasma Application
  – Taxila Lattice Boltzmann Method Application
  – Solid Fuel Ignition Application (from Petsc Examples)

• Algorithm-Based Fault Tolerance based on the Sparse Grid Combination Technique (SGCT).

• Designed and implemented general recovery routines for any application:
  – Non shrinking recovery (on same or spare nodes)
  – Shrinking recovery (under way ...)

Sparse Grid Combination Technique

- SGCT is a cost-effective method for solving time-evolving PDEs, specially for high dimensionality problems.
Fault Tolerant SGCT

=> Adding some extra small redundancies on SGCT achieves robustness

=> No failure
- redundancies remain unused

=> Failure
- some redundancies are used through alternate combination formula

=> “+” means added, “-” means subtracted, “x” and “o” means unused

no failure
(regular combination)

failure: one grid lost
(alternate combination)
Algorithm: FT-SGCT Application

1. \( W \): global communicator;
2. \( G = \{G_i\} \): set of sub-grids;
3. \( C = \{C_i\} \): set of sub-grid communicators created from \( W \);
4. \( g = \{g_i\} \): set of fields returned from the application computed on \( G \);
5. \( u = \{u_i\} \): corresponding set of sub-grid solutions;
6. \( u_i^c \): combined solution of the SGCT;
7. \( C_i \in C \) do in parallel
   \[ u_i \leftarrow \text{null}; //make runApplication initialize } g_i \]
8. \( \text{for each required combination do} \)
9. \( \text{for each } C_i \in C \) do in parallel
10. \( g_i \leftarrow \text{runApplication}(u_i, G_i, C_i); \)
11. \( u_i \leftarrow g_i; //on their common points \)
12. \( \text{updateBoundary}(u_i, C_i); \)
13. \( \text{reconstructFaultyCommunicator}(W); \)
14. \( u_i^c \leftarrow \text{gather}(u, W); \)
15. \( u \leftarrow \text{scatter}(u_i^c, W); \)


Our ULFM Experience

- **Good points:**
  - Sufficient functions available for different recovery implementations
  - Lots of example implementations and tutorials are available
  - Functions are designed in such a way that finding out application bugs is easy
• **Improvements points:**
  – The Two-Phase Commit agreement algorithm does not scale well on large core counts
    • Log Two-Phase Commit is scalable, but instable
  – Parallel I/O and non-blocking collectives are not supported
  – Performance varies according to the identity of the failed process
  – Bugs, hanging issues
Resilient X10 over ULFM

Sara S. Hamouda\textsuperscript{1}, Benjamin Herta\textsuperscript{2}, Josh Milthorpe\textsuperscript{1,2}, David Grove\textsuperscript{2}, and Olivier Tardieu\textsuperscript{2}

\textsuperscript{1}Australian National University, \textsuperscript{2}IBM T. J. Watson Research Center
Asynchronous Partitioned Global Address Space language

spawn a single task at startup
/starting at Place 0
val wordCount = new AtomicInteger();
val ref = GlobalRef(wordCount);
finish for (p in Place.places()) {
    val files = getFilesForPlace(p);
    at (p) async {
        //create task at place p
        val pCount = countWords(files, "ibm");
        at (refCount.home)
            ref().addAndGet(pCount);
    }
}
Console.OUT.println(wordCount);
Resilient X10

- Resilient **finish** construct:
  - Resilient termination detection algorithm

- Failure detection:
  - Through the transport layer

- Failure propagation:
  - Not required

- Failure notification to the application:
  - Exception

- Application Recovery:
  - Application's responsibility
• Resilient X10 was supported only over sockets.
X10 over MPI: Point to Point

### Initialization

```
MPI_Init_thread(..., ..., MPI_THREAD_MULTIPLE, ...);
MPI_Barrier(...);
```

### Sender

```
send_message(dest, ...):
    MPI_Isend(...) &request);
    pendingSends.add(request);
```

### Receiver

```
check_incoming_messages():
    MPI_Iprobe(MPI_ANY_SOURCE, &arrived, &status);
    if (arrived) {
        MPI_Irecv(..., &request);
        pendingRecieves.add(request);
    }
```

### Sender

```
check_pending_sends():
    for (request in pendingSends) {
        MPI_Test(request, ..., &completed);
        if (completed)
            pendingSends.remove(request);
    }
```

### Receiver

```
check_pending_receives():
    for (request in pendingReceives) {
        MPI_Test(request, ..., &completed);
        if (completed)
            pendingRecieves.remove(request);
    }
```

### Finalize

```
MPI_Barrier(...);
MPI_Finalize();
```
MPI Threading Support Levels

- **MPI_THREAD_SINGLE**
- **MPI_THREAD_FUNNELED**
- **MPI_THREAD_SERIALIZED**
- **MPI_THREAD_MULTIPLE**
• Team APIs

```scala
val team = new Team(places);

finish for (place in places) at (place) async {
  val src = new Rail[Int](SIZE, (i:Long)=> i as Int);
  val dst = new Rail[Int](SIZE);
  team.allreduce(src, 0, dst, 0, SIZE, Team.ADD);
}
```
• Team APIs

```scala
val team = new Team(places);

MPI_Comm_create(MPI_COMM_WORLD, grp, &comm);

finish for (place in places) at (place) async {
  val src = new Rail[Int](SIZE, (i:Long)=> i as Int);
  val dst = new Rail[Int](SIZE);
  team.allreduce(src, 0, dst, 0, SIZE, Team.ADD);
}

MPI_Iallreduce(....);
```
### Initialization

```c
MPI_Init_thread(..., MPI_THREAD_MULTIPLE, ...);
MPI_Comm_set_errhandler(comm, CustomErrorHandler);
MPI_Barrier(...);
```

### Sender

```c
send_message(dest, ...):
if (dest in failed_places) return;

MPI_Isend(..., &request);
pendingSends.add(request);
```

### Receiver

```c
check_incoming_messages():

MPI_Iprobe(MPI_ANY_SOURCE, &arrived, &status);
if (arrived) {
    MPI_Irecv(..., &request);
pendingRecieves.add(request);
}
```

### Check pending sends

```c
check_pending_sends():

for (request in pendingSends) {
    MPI_Test(request, ..., &completed);
    if (completed)
        pendingSends.remove(request);
}
```

### Check pending receives

```c
check_pending_receives():

for (request in pendingReceives) {
    MPI_Test(request, ..., &completed);
    if (completed)
        pendingRecieves.remove(request);
}
```

### Finalize

```c
MPI_Barrier(...);
MPI_Finalize();
```

### CustomErrorHandler

```c
failed_places = x10_get_failed_places(failedGroup);
```
X10 over ULFM

**Initialization**

- `MPI_Init_thread(..., MPI_THREAD_SERIALIZE, ...);
- `MPI_Comm_set_errhandler(comm, CustomErrorHandler);
- `MPI_Barrier(...);

**Sender**

- `send_message(dest, ...):
  - if (dest in failed_places) return;
  - `MPI_Isend(..., &request);
  - `pendingSends.add(request);

**Receiver**

- `check_incoming_messages():
  - `MPI_Iprobe(MPI_ANY_SOURCE, &arrived, &status);
  - if (arrived) {
    - `MPI_Irecv(..., &request);
    - `pendingRecieves.add(request);
  }

**Check pending sends**

- `check_pending_sends():
  - for (request in pendingSends) {
    - `MPI_Test(request, ..., &completed);
    - if (completed) `pendingSends.remove(request);
  }

**Check pending receives**

- `check_pending_receives():
  - for (request in pendingReceives) {
    - `MPI_Test(request, ..., &completed);
    - if (completed) `pendingRecieves.remove(request);
  }

**Finalize**

- `MPI_Barrier(...);
- `MPI_Finalize();

**CustomErrorHandler**

- `OMPI_Comm_failure_ack(*comm);
- `OMPI_Comm_failure_get_acked(*comm, &failedGroup);
- `failed_places = x10_get_failed_places(failedGroup);
• Team APIs

```scala
val team = new Team(places);

MPI_Comm_create(MPI_COMM_WORLD, grp, &comm);

finish for (place in places) at (place) async {
    val src = new Rail[Int](SIZE, (i:Long)=> i as Int);
    val dst = new Rail[Int](SIZE);
    team.allreduce(src, 0, dst, 0, SIZE, Team.ADD);
}

MPI_Iallreduce(....);
```
• Team APIs

```scala
val team = new Team(places);

finish for (place in places) at (place) async {
    val src = new Rail[Int](SIZE, (i:Long)=> i as Int);
    val dst = new Rail[Int](SIZE);
    team.allreduce(src, 0, dst, 0, SIZE, Team.ADD);
}

OMPI_Comm_shrink(MPI_COMM_WORLD, &shrunken);
MPI_Comm_create(shrunken, grp, &comm);

MPI_Iallreduce(....);
```
Team APIs

```scala
val team = new Team(places);

OMPI_Comm_shrink(MPI_COMM_WORLD, &shrunken);
MPI_Comm_create(shrunken, grp, &comm);

finish for (place in places) at (place) async {
  val src = new Rail[Int](SIZE, (i:Long)=> i as Int);
  val dst = new Rail[Int](SIZE);
  team.allreduce(src, 0, dst, 0, SIZE, Team.ADD);
}

MPI_Iallreduce(....);
```

Non blocking collectives are not supported in the current ULFM implementation
Team APIs – Moving to blocking collective

```scala
val team = new Team(places);

OMPI_Comm_shrink(MPI_COMM_WORLD, &shrunken);
MPI_Comm_create(shrunken, grp, &comm);

finish for (place in places) at (place) async {
    val src = new Rail[Int](SIZE, (i:Long)=> i as Int);
    val dst = new Rail[Int](SIZE);
    team.allreduce(src, 0, dst, 0, SIZE, Team.ADD);
}

x10_emu_barrier();

MPI_allreduce(....);
```

Blocking collective
X10 Over ULFM

**LULESH proxy application**

The performance improvement due to using ULFM v1.0 for running the LULESH proxy application, running on 64 processes on 16 nodes with problem size $20^3$ per process. The cluster is an AMD64 Linux cluster, each node having 16G RAM and 2 quad core AMD Opteron 2356 processors.
• Good points:
  – Sufficient functions available for different recovery implementations
  – Lots of example implementations and tutorials are available
  – Functions are designed in such a way that finding out application bugs is easy
  ➢ Flexibility of the minimalistic fault tolerance approach provided by ULFM
  ➢ Prompt support from the ULFM team
• **Improvement points:**
  
  – The Two-Phase Commit agreement algorithm does not scale well on large core counts
    
    • Log Two-Phase Commit is scalable, but instable
  
  – Parallel I/O and non-blocking collectives are not supported
  
  – Performance varies according to the identity of the failed process
  
  – Bugs, hanging issues
  
  ➢ ULFM is based on an old OpenMPI 1.7 version, in which multi-threading is not well tested.
  
  ➢ Portability and continuity concerns
• Resilient X10 applications can now run over ULFM and achieve better performance with the optimized MPI communication routines and the support for high speed network protocols provided by MPI (e.g. Infiniband verbs).

• Try it out!
  – X10 web site: x10-lang.org
  – X10 source code: https://github.com/x10-lang