Fault Tolerance in AMT Programming Models & Runtimes

Sandia National Labs
Hemanth Kolla,
Keita Teranishi,
Nicole Slattengren,
Matthew Whitlock,
Jackson Mayo

Georgia Tech
Sriraj Paul,
Akihiro Hayashi,
Vivek Sarkar

Platform for Advanced Scientific Computing (PASC)
July 02^{nd}-04^{th}, 2018, Basel
Motivation & Background

- Substantial progress in resilience and asynchronous many-task (AMT) programming models, separately.

- AMT offer:
  - more flexible and efficient failure mitigation compared to conventional (e.g. checkpoint) strategies.
  - ability to quantify the effects of failures and benefits of various resilience strategies.

- Complex tradeoffs of multiple AMT resilience techniques with dynamic failure behavior need to be understood/documentd.

- Need ability to extrapolate tradeoffs to extreme(exa)-scale.
Objectives

- An analysis of the scalability, performance and costs for multiple AMT resilience options.
- Prototype implementation of resilience schemes in actual asynchronous many-task programming model:
  - task replication.
  - task replay.
  - algorithm-based fault tolerance.
  - task-level checkpointing.
- An analysis of accuracy-cost tradeoffs of application-specific failure detection and mitigation schemes.
- Use representative mini-apps as basis for study.
Current Scope: On-Node AMT

- MPI+(on-node) AMT an anticipated programming model for future complex node architecture.

- First comprehensive study with on-node AMT. Extend concepts to distributed AMT in future.

- Analyzing Failure/Error mitigation by On Node AMT is essential:
  - Hard failure of cores and accelerators, silent errors, performance degradation.
  - Failure can be manifested as task failure: non-finishing tasks, data corruption or very slow execution.

- We still need better understanding of failure-free AMT as a baseline, production-ready distributed AMT is scant.
Is Analytical Modeling Tractable?

“Series-Parallel” graphs are analysable.

- Decompose a DAG into series/parallel components [1].
- Start analysis from bottom-up.
- Powerful abstraction for graph analysis.

Is Analytical Modeling Tractable?

Optimal algorithms for “series-parallel” decomposition [2].

Is Analytical Modeling Tractable? (hint: NO)

Decomposition not possible if forbidden patterns exist [3].

Task-DAG for 1D-stencil

Alternate Solution: Task-DAG Simulator

- Task-DAG simulator to hypothesize the behavior of resilient AMT under numerous system and runtime situations.
- Developed a tool to traverse task-dags on multicore multi threaded environment:
  - C++ code using Boost Graph Library.
  - Parametrize various aspects of runtime/hardware: # threads, scheduling algorithm, memory hierarchy, task execution time, resilience strategy.
- Replay by subgraph to support failure containment in multiple granularities
- Tune and validate with actual AMT experiments, extrapolate to different scales/parameters.
The graph generated/traversed is app-specific (e.g. 1D-stencil).

Simulator itself is serial (parallelizing is a research topic unto itself).
Sample (hypothetical) Simulation

Simulation of 1D-stencil with 0.1% failure, task replay.

- Simple heuristics for scheduling (FIFO).
- No over-decomposition penalty, small variability in task exec times
- Take away: more the over-decomposition the better.
Resilient AMT Prototype

- **Aim:** implement general building blocks for resilience in an actual AMT.

- **Three Major “Task” interfaces for:**
  - Task Replication
  - Task Replay
  - Algorithm-based Fault Tolerance (ABFT) Tasks

- **Resilient data abstraction**
  - Reference counting for replica management
  - Abstraction of replicated data dependencies
  - Data Persistence (ongoing)

- **AMT of choice:** Habanero-C++.
Habanero-C++ Overview

- Project led by Vivek Sarkar (GaTech/Rice U)
- Library-based tasking runtime and API
  - Semantically derived from X10
- Focused on: lightweight, minimal overheads; flexible synchronization; locality control; composability with other libraries;
- Simplified deployment: no custom compiler, entirely library-based, only requires C++11 compliant compiler
- Uses runtime-managed call stacks to avoid blocking
- [https://github.com/habanero-rice/hclib](https://github.com/habanero-rice/hclib)
# Habanero-C++ Overview

## HClib constructs

<table>
<thead>
<tr>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous task creation</td>
<td><code>async(() -&gt; { S1; });</code></td>
</tr>
<tr>
<td>Bulk task synchronization</td>
<td><code>finish(() -&gt; { async(() -&gt; { S1; async(() -&gt; S2;); }); });</code></td>
</tr>
<tr>
<td>Futures and promises</td>
<td>`async(() -&gt; { prom-&gt;put(42); });</td>
</tr>
<tr>
<td></td>
<td><code>async(() -&gt; { prom-&gt;get_future()-&gt;wait(); });</code></td>
</tr>
<tr>
<td></td>
<td><code>async_wait(() -&gt; {...}, prom-&gt;get_future());</code></td>
</tr>
<tr>
<td>Bulk task creation</td>
<td><code>forall(loop, (i, j, k) -&gt; { S3; });</code></td>
</tr>
<tr>
<td>Places for locality control</td>
<td><code>async_at(pl, () -&gt; { S4; });</code></td>
</tr>
</tbody>
</table>
Habanero-C++ Overview

- Express data dependencies using **promises** and **futures**.
  - **hclib::promise**
    - Store a value using single assignment semantics: `promise.put(value)`
  - **hclib::future**
    - Retrieve the value stored in a promise: `value = future.get()`
    - Can be used as dependency for tasks
- Relation between future and promise
  - `future = promise.get_future()`
  - If accessed from different threads `put()` and `get()` are synchronized, thus enabling a way for synchronization.
HClib extension: (1) Reference Counting

- Current implementation leaves it to user to manage dynamic allocated memory (no automatic garbage collection).

- Reference counting semantics:
  - Provide a way to perform garbage collection based on the use of future as task dependency
  - Allows transparent handling of data access by replay/replicated tasks.

- Implementation extends promise to have a reference count
  - Count set during object construction
  - Count decreased using release() method

- Extend async-await to perform automatic reference counting
  - Reference count is decreased each time a future associated with the promise is used as dependency
HClib extension: (2) Task Replication

- diamond::async_wait_check<N>( lambda, hclib::promise<int> out, hclib_future_t *f1, .., hclib_future_t *f4);

- N-plicates the task and checks for equality of put() operations at the end of the task
- If error checking succeeds (majority voting), actual puts are done
- If error checking fails, puts are ignored and the error is reported using an output promise
- Duplicate (N=2) – Create two tasks and check for error in puts
  - If error checking fails, a third task is created
- Triplicate and more (N=3 ore more) – Create three tasks and check for error in puts
  - Two out of three outputs should match for success
HCLib extension: (3) Task Replay

```cpp
replay::async_wait_check<N>( lambda, hclib::promise<int> out, std::function<int(void*)> error_check_fn, void * params, hclib_future_t *f1, .. , hclib_future_t *f4);
```

- Dynamic response to failure
- Executes the task and checks for error using the error checking function
- `error_check_fn(params)` returns true if there is no error
- The task is executed \textbf{N} times at most if there is any error
  - If error checking fails, puts are ignored and the error is reported using an output promise
HClCompare extension: (4) ABFT Task

\[
\text{abft::async\_await\_check ( lambda, hclib::promise<int> \text{ out, std::function<int(void*)> error\_check\_fn, void * params, hclib\_future\_t *f1, .. , hclib\_future\_t *f4, ABFT\_lambda);}\]

- Executes the task and checks for error using the error checking function
- \text{error\_check\_fn(params)} returns true if there is no error
- If there is error then \text{ABFT\_lambda} is executed and checked for error again at its end
  - If error checking fails, puts are ignored and the error is reported using an output promise
HClib Task Scheduling

- **Worker**: a thread that executes tasks (pthread)
  - Fixed number of workers is created at program start (thread pool).

- **Distributed dequeue**:
  - Each worker has a dequeue.
  - A new task is pushed (to tail), stolen from head.

```c
/* pseudo code*/
do {
  task = pop();
  if (!task) {
    task = steal();
  }
  if (task) exec(task);
} while(1);
```
HClib Hierarchical Work Stealing

- **Place**: a virtual partition of workers
  - Can be specified to reflect memory hierarchy.

- **Work stealing gradually up the hierarchy of places**
  - When idle, first look for work in current place worker dequeues.
  - If unsuccessful, look at place one level above....

![Diagram showing the hierarchical work stealing process](image)

Place: a virtual partition of workers. Can be specified to reflect memory hierarchy.

Work stealing gradually up the hierarchy of places. When idle, first look for work in current place worker dequeues. If unsuccessful, look at place one level above....
HClib Resilience Overheads

- Overhead only for the API i.e. no actual failures.

- 1D stencil:
  - 8000 pts per task
  - 64*32000 tasks

- Intel Xeon X5660 Westmere
  - 2 sockets
  - 6 cores per socket
  - 1 thread per core

- 12 HClib workers (1 per thread).
HClib Resilience Overheads

(Execution time) × (Number of Workers)

- basic
- ref-ct
- replication
- replay

+1.6%
+116%
+9.5%
Results

- Suite of experiments to study:
  - Habanero’s performance for imbalanced applications (no failure).
  - Habanero’s performance with resilience mechanisms active.
  - Simulator runs to fine tune parameters (represent Habanero performance).

- Two representative mini apps:
  - 1D stencil: spatial grid decomposed into tiles, fixed number of iterations
    - balanced: uniform tile size, imbalanced: non-uniform tile sizes
  - 3D stencil:
    - more dependencies per task (7 vs 3 compared to 1D), more compute intensity, non-contiguous memory, error-delay propagation pattern is different.

- Unstructured mesh Sparse-Matrix Vector kernel:
  - inherently imbalanced, problem dependent
**1D stencil: Balanced vs Imbalanced**

Balanced workload

Balanced workloads min=avg=max=5 over 512 tiles
Vary extent that tasks are pinned to threads

<table>
<thead>
<tr>
<th>Workers</th>
<th>Walltime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0.5</td>
</tr>
<tr>
<td>16</td>
<td>0.25</td>
</tr>
</tbody>
</table>

- Never
- Init only
- Init and all iters

Imbalanced workload

Irregular workloads min=4, avg=5, max=100 over 512 tiles
Vary extent that tasks are pinned to threads

<table>
<thead>
<tr>
<th>Workers</th>
<th>Walltime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0.5</td>
</tr>
<tr>
<td>16</td>
<td>0.25</td>
</tr>
</tbody>
</table>

- Never
- Init only
- Init and all iters

**Even simple scheduler + work-stealing hides 25x imbalance.**
Unstructured Sparse Mat-Vec

- Sparse Matrix for sample structural mech problem (real, symmetric)
- TAMU Sparse Matrix collection
  - Formerly Univ. Florida coll.
  - https://sparse.tamu.edu
  - Crank Shaft model (crankseg_1)
- Problem params:
  - nRows=nCols = 52804
  - num_non_zeros = 10614210
- 1D decomposition of Matrix/Vector
  - 128 tiles.
  - 1 task per tile
1D/3D stencil with failures

- Actual experiments with Habanero implementation of stencil.
- Corresponding simulator runs.
- Simulator params tuned from baseline (failure-free) runs.
- Various failure rates
  - Range from 0.1%-10%.
  - Emulate identical tasks to fail in both.
  - Task replay invoked to recover.
  - Error check based on checksum.

- Cost of recovery marginal, proportional to failure rate.
- Actual Habanero experiments for 3D better than linear.
1D/3D stencil with failures

No noticeable perturbation to progress on each work unit (tile).
Conclusions & Future Work

- AMT offers good prospects for handling failures.
- Even simple scheduling and work-stealing methods allow AMT to balance out extra work to recover from faults.
- Demonstrated benefits of AMT using:
  - A Task-DAG simulator that can effectively parametrize AMT stack.
  - Implementations in an actual AMT runtime – Habanero C++.
- Failure recovery mechanisms are not a silver bullet:
  - Cost of task replay depends on cost of error checking vs cost of replaying.
  - Cost of task replication depends on cost of each work unit, data replication.
- Extensions to distributed AMT (HPX, Habanero + MPI).
Backup
Tiled Sparse Matrix-Vector Multiply: PULL Model

- No duplicate copies of the vector, each vec tile accessed by every neighbour tile that needs it (reference counted).
- Extremely cache unfriendly since one promise might be accessed by multiple async tasks within the same iteration, overhead of reference counting??
- Each vector tile creates copies of subsets needed by other tiles. Should decrease cache contention.
- Duplication of the vector increases with number of tiles. Many more promises, and no reduction in reference count penalty.
Performance Work Stealing VS No Work Stealing

16 tasks per worker
4 tasks per worker
1 task per worker (No work stealing)

- 1D stencil code with over-decomposition
- HCLIB like Decentralized LIFO Scheduler with Work Stealing
- Task Replay for failure
- Run 1000 time steps.
- Each line indicate the elapsed time unit for every 20 time step.
Switching the scheduling to FIFO change the progress of tasks.

For ODF=16, a large variation of the progress pattern.

- Small variation in the execution time to finish all the tasks.
- Need to integrate MPI communication to tasks on both sides.