As Much Resilience As You Want: Application-tailored Resilience in Legion

***Abstract*—Resilient is an important aspect of scaling applications on large clusters. As we approach parallelism profiles of several millions and long running applications, we need to ensure that ineffect "we reach the end". Defining the policy interaction with the programming model features necessitates that we revisit the memory concistency model. Recoverability, a critical step in resilience, opens the door to optimizations such as speculation. We also evaluate this. . . .**

***Index Terms*—resilience, legion**

I. Introduction

A failure during a large-scale execution of any application on an extreme-scale system leads to loss of time and money, and can cause a nightmare to any application developer. While failures could be due to application bugs, failures due to errors not detectable by error correcting code (ECC) are on the rise. For example, Geist states that such failures are a common occurrence on Oak Ridge National Laboratories leadership class machines [6], and Schroeder and Gibson state that a large number of CPU and memory failures were from parity errors after tracking a five-year log of hardware replacements for a 765 node high-performance computing cluster [3]. Echoing the sentiment of Snir et al. [7], we believe it is critical to overcome the effect of such failures through a productive and performant resilience interface in the parallel programming model so that a developer can make avail the performance on extreme- and upcoming exa-scale parallel systems.

The traditional solutions to address failures during large-scale parallel executions includes the application developer: (a) manually takes periodic checkpoints, and manually controls the rollback and restart from specific checkpoints [**?**], (b) employs a semi-automatic approach, where the developer employs programming constructs that have well defined semantics in the event of failures, e.g, `finish` blocks in X10 [**?**]. Here, the parallel programming model runtime automatically corrects itself to handle further execution in the event of a failure but limited to the beginning of these specific programming constructs. (c) employs a sophistica

In x10, use programming language constructs such as resilient finish blocks to capture their completing along with useful semnatic guarantees.

Autocorrecting, where the programmer labels computations which form.

We believe these solutions are good, but not flexible to demand the complexities of current extreme-scale and upcoming exa-scale systems. Its is not at one point, every mapping needs to make a decision, support for checking whther there is a tradeoff between recomputing the dag vs .. this has to be dynamic. We also believe that these decision should be in a deferred execution state, not the whole computation stopping.

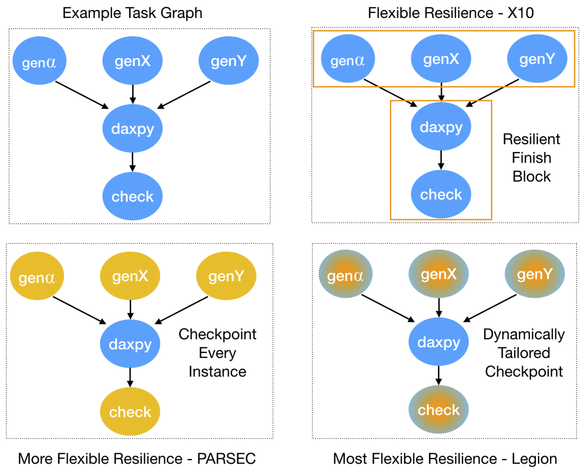
Example Task Graph
genα
genX
genY
daxpy
check
Flexible Resilience - X10
genα
genX
genY
daxpy
check
Resilient Finish Block
genα
genX
genY
daxpy
check
Checkpoint Every Instance
More Flexible Resilience - PARSEC
genα
genX
genY
daxpy
check
Dynamically Tailored Checkpoint
Most Flexible Resilience - Legion

Fig. 1. A spectrum of resilience mechanism supported by different state-of-the-art parallel runtimes along with the proposed resilience strategy for Legion.

To drive this point, consider the figure . A simple computation task graph, X10 would allow this, PARSEC would allow this, Charm++ would focus on the stores of alpha, x and y. However, we believe that this solution. soemtimes you have a resilient store where you map alpha, x but sometimes not. Think here.

The right vehicle to demonstrate such a dynamic decision for checkpoints, along with a deferred execution model is the Legion programming model. The strengths of this include a decoupled scheduling, mapping, and execution analysis stages that drives this solution.

Our contributions are as follows:

- the most flexible resilience mechanism
- dynamic decisions for checkpointing
- auto rollback and recovery
- garbage collector that will also collect previous checkpoints based on a novel post-dominator algorithm.
- a discussion of the semantics of regions when there is rollback

The rest of the paper is organized as follows:

II. Overview of Resilience in Legion

III. Interaction of Resilience Policy with Legion Features

which are the legion features of importance ?

put a figure of interaction.

what is the memory consistency angle here ?

what does it mean to advance the commit wavefront

Resilience is a tangling of the lifetime of a task and a region snapshot

1) when can we advance a commit wavefront ? 2) whats the lifetime of a region-instance snapshot ?

on this front, we see it as a two-step process a) define a consistent cut of tasks problem, b) commit any task strictly behind this wavefront c) garbage collect any snapshot that serves as input to any task that is already committed

consistent cut of tasks that can be part of the commit wavefront: a set of tasks whose inputs are need_preserve'ed, or they are strictly post-dominated by tasks whose inputs are need_preserved.

3) what about copy/index/tasks ? copy local to local follows the above semantics copy local to remote get committed immediately after successful execution. index launch tasks, actually feel need not be in the task graph, unless virtual mapping is used. they can be garbage collected immediately after all child tasks are included in the dependence analysis wavefront - I am thinking we will never have a case where are relaunching the index launch, since hte child tasks either are part of the committed wavefront or are not (pending a discussion on phase barriers)

What to do about must_epoch tasks with phase_barrier inside them ?

answer: restartable phase_barrier with generation commit callback

IV. Implementation of Resilience

A. `need_preserve` *Implementation*

use current_versions inside legion_views.h TODO TODO

tagging region instances tagging tasks

-the mapper marks an instance as persistent, i.e., vector¡vector¡bool¿¿ persistent -in finalize_map_task, the singleTask sees this and notes it down in the Individual_task's persistent_tasks list -when trigger_complete is called on the task,

1) inside line 5560, invalidate_region_tree_contexts, inside which we have runtime-¿forest-¿invalidate_versions, we do not do on region[idx].region. we also do not do the instance_top_views

2) we retain the mark on the task as allowed_for_gc.

3) we go to the incoming of this task, and just like verified_regions[true], we mark outgoing_edge_dominated[true] if all the outgoing of a task is marked as true, then we change allowed_for_gc = true.

Discussion Points for today ———————— 1) allow persistence on a subset of mapped instances in map_task ? Pro: flexibility, Con: if a checkpoint is to be considered useful for a restart, not having the full set of inputs checkpointed seems contradictory. 2) verify_regions tracks op dependencies after physical dependence analysis, correct ? 3) a discussion on persistence inside map_task call

discussed design:

1) build a function set_hardened_instnace(instance,task) along the lines of the set_gc_priority(instance, never, task) inside the mapper call 2) inside that mark a task's incoming edges as saying that it leads to a hardened instance(task) 3) the garbage collector will basically collect a task whose outgoing edges are all marked as verified/hardened. 4) if a task is gc'ed, then it marks all its incoming edges as leads to a hardened instance. 5) steps 1-4 will be based on set_garbage_collection_priority and how verified_regions are set. We will be adding a new list to each task, similar to verified_regions, that will represent edges_ending_in_hardened_regions.

quash_operation: 847 line in realm/proc_impl.cc tells us that we should set the poisoned to true and ensure that the start_event.has_triggered_faultaware(poisoned) returns true

what about two sibling tasks that map the same instance, and one hardens it, the other does not - alex raised this, we discussed that trigger recover would check - but the way we are doing it now, we are doing unverfied_regions.erase based on even on guy below us who will call harden on a region instance - so the parent would have gone away, but the child B who did not call harden ont he physical region should be able to restart since the instance will be there how to show thsi is working, if the application is too fast, then delete meta task on the logical region is called which deletes the region irrespective of whether its GC_NEVER_PRIORITY

REACTIVE faulty task catch it poison propagation identify recovery point trigger_recover

SEMANTICS regions and instances and consistency

PROACTIVE mark instances as harden

DIFFERENCE WITH PARSEC: label a task with snapshot, i.e., all or none policy whereas we allow selective instances to be marked as harden. - what is an example app that is benefited by this ?

1) Interaction of need_preserve with the commit wavefront:

2) Obtaining dependence graph in the mapper before calling need_preserve:

B. *ProfilingResponse callback used*

while using the profiling measurement reporting infrastructure as-is causes overhead, this is because of the invoking of a mapper profiling response function. We do not need that for resilience, so the resilience callback would short circuit it.

task launch -¿ porfiling reported event is there -¿ when it is triggered -¿ `singletask::profiling_response` is invoked -¿ handle things —— in here, there is no need to call the mapper side yet. So, avoid one overhead there, maybe this was never called and so , this is not an optimization. ok, back to square one.

V. Resilience Application: Speculation

there are three different wavefronts in Legion, we can speculate on any of them.

From a speculation perspective, are they different ? Are we novel, since we have these three different wavefronts ? Can we navigate through this, like the blanks

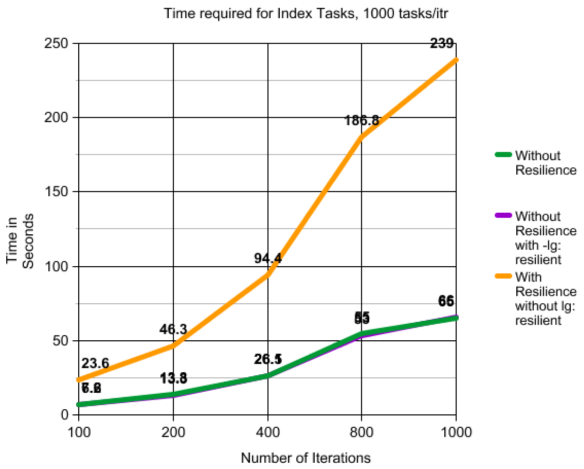

Fig. 2. Total time taken by 02_index_tasks. Time in Seconds, 1000 tasks/index launch

1) execution wavefront what does it mean to speculate here, does the other steps have to be complete before we do this.

2) mapping wavefront

3) dependence analysis wavefront

– see mike's 6 wavefront answer.

There are three different wavefronts, there could

the 6 wavefronts, where does speculation plays a role mike - speculation is more about tracking the resolution wavefront while resilience is more about tracking the commit wavefront, but the when things go bad, then i think the machinery to restart the mapping and execution wavefronts should be the same

VI. Experiments

A. Index Tasks

a) Growth of memory as execution proceeds:

b) Performance overhead as execution proceeds:

NumIter	Native	Resl	lg:resl
128	23	16	21
256	20	22	28
512			

NumIter	Native	lg:resl	Resl
100	7.2	6.6	23.6
200	13.8	13.3	46.3
400	26.1	26.5	94.4
800	55	53	186.8
1000	65	66	239.9

NumIter	Native	Resilience
100	1.12	2.43
200	1.56	4.39
400	3.00	9.45
800	5.58	19.09
1000	7.34	23.2

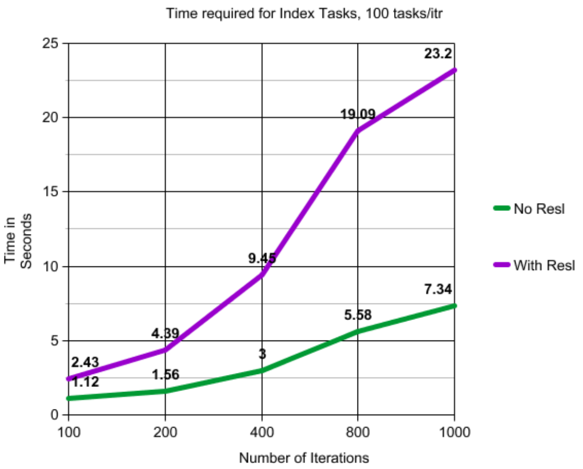

Fig. 3. Total time taken by 02_index_tasks. Time in Seconds, 100 tasks/index launch

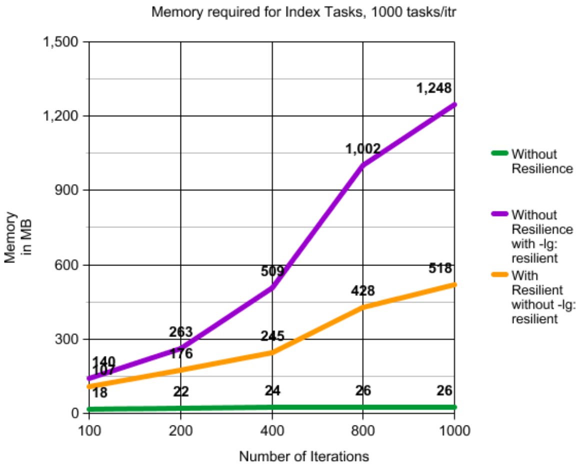

Fig. 4. Total Memory footprint by 02_index_tasks. Memory in MB, 1000 tasks/index launch.

NumIter	Native	lg:resl	Resl
100	18	140	107
200	22	263	176
400	24	509	245
800	26	1002	428
1000	26	1248	518

B. Stencil

C. local recover vs global recovery

D. compute/comm vs no-failure/single failure/multi-failure

E. S3D, Pennant, Stencil, Circuit

F. Some Interesting Task Graphs for Recovery

G. bigger examples

pennant, stencil, miniAero (better understood) /Circuit, -limit the failure to the before

VII. Adaptive Resilience

VIII. Resilience Policy Implemented via Mapper: Example 1 Generic

IX. Resilience Policy Implemented via Mapper: Example 2 UT Austin

X. Conclusion

Acknowledgment

References

[1] S. Treichler, M. Bauer, and A. Aiken. Language support for dynamic, hierarchical data partitioning. In Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2013.

[2] D. Grove et al. Failure Recovery in Resilient X10. IBM Research Report, RC25660 (WAT1707-028) July 21, 2017.

[3] B. Schroeder and G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you. In Proceedings of the 5th USENIX conference on file and storage technologies (FAST), pp. 116, 2007.

[4] Acun et al. Power, Reliability, and Performance: One System to Rule them All In Computer Issue No. 10 - Oct. 2016 vol. 49. ISSN: 0018-9162. pp: 30-37.

[5] C. Cao, G. Bosilca, T. Herault, and J. Dongarra. Design for a soft error resilient dynamic task-based runtime. In 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS), May 2015.

[6] A. Geist, What is the monster in the closet? Invited Talk at Workshop on Architectures I: Exascale and Beyond: Gaps in Research, Gaps in our Thinking, aug 2011.

[7] M. Snir et al. Addressing failures in exascale computing. In The International Journal of High Performance Computing Applications 2014, Vol. 28(2) 129173.