H2020 FETHPC-1-2014

An Exascale Programming, Multi-objective Optimisation and Resilience Management Environment Based on Nested Recursive Parallelism
Project Number 671603

D5.5 – Implementation and Evaluation of Application Specific Resilience Techniques (a)

WP5: Cross layer resilience and online analysis for non-functional parameters

Version: 1.0
Author(s): Kiril Dichev (QUB), Herbert Jordan (UIBK), Thomas Heller (FAU), Xavier Aguilar (KTH), Amitabh Trehan (QUB)
Date: 20/01/17

D5.5 – Implementation and Evaluation of Application Specific Resilience Techniques (a)

Due date:	PM16
Submission date:	day/month/year
Project start date:	01/10/2015
Project duration:	36 months
Deliverable lead organization	Queen’s University Belfast
Version:	1.0
Status	Final
Author(s):	Kiril Dichev (QUB), Herbert Jordan (UIBK), Thomas Heller (FAU), Xavier Aguilar (KTH), Amitabh Trehan (QUB)
Reviewer(s)	Roman Iakymchuk (KTH), Stéphane Monté (NUM)

Dissemination level	
PU/PP/RE/CO	*PU*

Acknowledgements

The work presented in this document has been conducted in the context of the EU Horizon 2020. AllScale is a 36-month project that started on October 1st, 2015 and is funded by the European Commission.

The partners in the project are UNIVERSITÄT INNSBRUCK (UBIK), FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN NÜRNBERG (FAU), THE QUEEN'S UNIVERSITY OF BELFAST (QUB), KUNGLIGA TEKNISKA HÖGSKOLAN (KTH), NUMERICAL MECHANICS APPLICATIONS INTERNATIONAL SA (NUMEXA), IBM IRELAND LIMITED (IBM).

The content of this document is the result of extensive discussions within the AllScale Consortium as a whole.

More information

Public AllScale reports and other information pertaining to the project are available through the AllScale public Web site under http://www.allscale.eu.

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	08/01/17	First draft	Kiril Dichev
0.2	10/01/17	Nicer figures, improved simulator section, a section on future work and limitations	Kiril Dichev
0.3	18/01/17	Modifications addressing RI feedback	Kiril Dichev, Roman Iakymchuk
0.4	19/01/17	Review from Numeca, mostly English corrections	Stéphane Monté
1.0	20/01/17	Final version	Kiril Dichev

Table of Contents

Index of Figures

Executive Summary

This deliverable focuses on the ongoing work within Task 5.3 (Resilience Primitives) and Task 5.4 (Development of Application-Specific Resilience Techniques). Task 5.3 well summarizes the central resilience strategy for the AllScale project, which is a "generic task-level checkpoint-restart". In this document, we interchangeably use the terms "task" and "work item". The entire resilience protocol will be based on tasks, so different applications will differ in the different manifestation of task dependencies, task trees, and task execution cost. In this sense, the recovery protocol will be application-specific *only* at runtime.

However, no algorithmic knowledge will be integrated in the resilience protocol. In this sense, there is a digression from some aspects proposed in Task 5.4. Algorithmic approaches to resilience will not be pursued, since application requirements are based on hardware failure recovery (D5.1). For the same reason, self-stabilizing techniques, understood as based on algorithmic properties to recover certain invariants of the application, will not be pursued. On the other hand, our proposed node failure recovery is compatible with self-healing as proposed in parts of T5.4, and in D5.1. Self-healing is understood here in the broad sense of coping with failure in continued execution via checkpoint/restart.

We are currently implementing a discrete-event simulator to benchmark and experiment with generic task-level checkpoint-restart strategies. A simulator allows to prototype a distributed protocol for resilience in shared memory, independent on orthogonal efforts of other packages. It also directly can support the cost model of resilience which is part of T5.3. The simulator is also related (but not identical) to the proposed study of resilience strategies "in an isolated context limited to shared memory systems" of T5.4.

1 Introduction

This deliverable will detail the ongoing work on a generic task-level resilience protocol. In this work, recovery from node failures is central to the protocol, since this was requested by the majority of applications in a questionnaire listed in D5.1. The task-based nature of the protocol is dictated by the overall AllScale architecture (see D2.3) we develop. We will first describe the protocol. Then, we will outline the ongoing work on a prototype and verification of the protocol in a discrete-event simulator.

2 A Task-Based Resilience Protocol

2.1 Guard-Protectee Relationship between Nodes

At the core of the recovery is a guard-protectee relationship established between pairs of nodes in a distributed run. Each node has a protectee node, and each node has a guard node. For 2 nodes *A* and *B*, *protectee(A) = B* is equivalent to *Guard(B) = A*. Of course, a guard and a protectee of a node may differ. One possible way to arrange this is to build a logical ring of all nodes in a run, for example as:
protectee(A) = <right_neighbour of A>;
guard(<right_neighbour of A>) = A;
A visual example of such a relation is Figure 1.

It is the responsibility of the guard to protect its protectee; that is to perform recovery if it gets a signal that its protectee has crashed. This requires checkpoints to be written and read. We detail the checkpoints in following subsection.

2.2 Checkpoints of Closures and Data

Each protectee stores a consistent checkpoint in a way that it is permanently accessible to its guard. This could be for example by requesting the guard node to keep a copy in main memory, or by using disk that is globally accessible (e.g. NFS partition). A checkpoint consists of:

- A **list of the work items** scheduled to run at the point of checkpointing. For each of these work items, following checkpoint is required:
  - **A closure** of the work item (i.e. what the runtime needs to restart it) -- mandatory
  - **Any input data** accessed for read/write operations within the work item – optional (but needed in a vast majority of applications and application stages). This data is application-specific, but may be uniformly accessed via the AllScale Core API (D2.3, Sect. 2.2.3.2).

As an example of each node keeping both scheduled work items, and the additional list of protected work items (scheduled on the protectee), consider Figure 1: Example of running tasks and protected tasks for 3 running nodes.

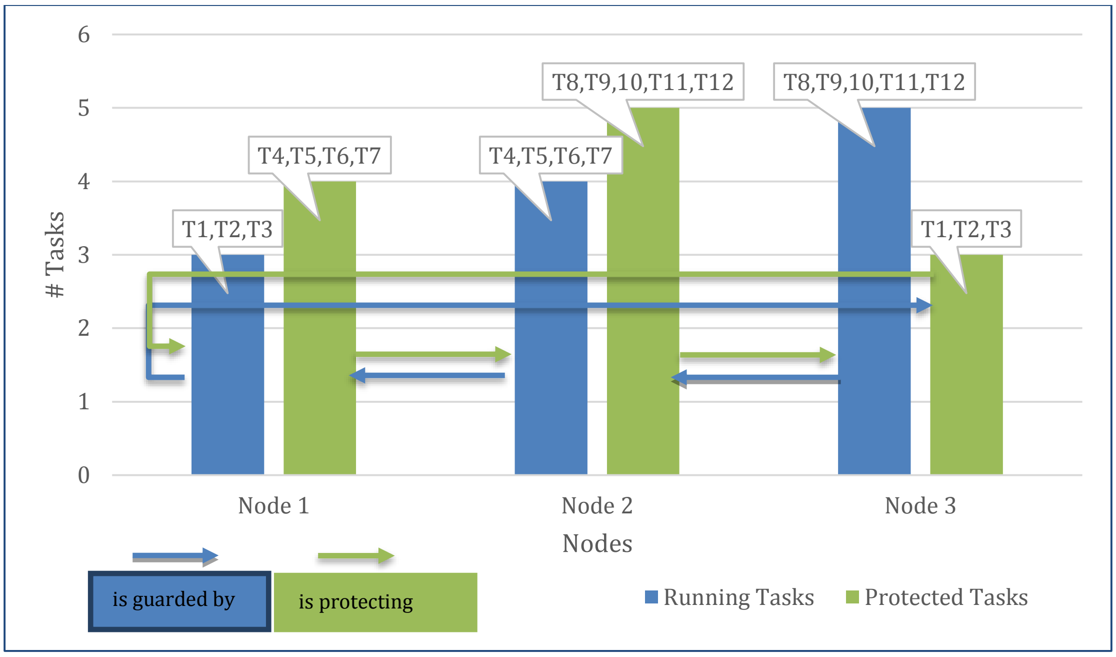

Figure 1: Example of running tasks and protected tasks for 3 running nodes.

Nodes 1, 2 and 3 build a logical ring as defined above, and each node protects its right neighbor by storing a checkpoint of the right-neighbor tasks. The resilience protocol requires the additional protected queues (in green) at each node.

The granularity at which checkpoints are taken is discussed in Sect. 2.5.

2.3 Fault Detection

The fault detection will be part of the monitoring component running. This part of the monitoring component will follow the distributed heartbeat protocol we already detailed in D5.1, Sect. 5.1. The monitoring component runs in distributed fashion on all nodes, and if one monitoring instance detects a node failure, the guard node of the failed node needs to be signaled. Since in D5.1 we proposed a logical ring for the distributed heartbeat protocol, it is very intuitive for this ring to follow the guard-protectee relationship ring. That is, each protectee sends its guard heartbeats in regular intervals to its guard. If a protectee skips a heartbeat, the guard's monitoring component signals it about its protectee failure. The guard then needs to perform recovery, as detailed in following section.

2.4 Recovery

Upon signal of a fault of its protectee, a guard is responsible for the recovery strategy. The recovery consists of:

- Eliminating all orphan work items (that is, work items that are children of work items that died on failed node) on all running nodes (this is a collective operation involving all nodes). If this step is ignored, duplicate tasks (orphan tasks followed by restarted tasks) are very likely to corrupt any global data that is being modified in a non-idempotent way. This is the case for most meaningful applications. We eliminate orphan tasks by:
    - Remove all orphan items scheduled to run on a node,
    - Wait on all orphan items already running on a node. The modifications on global data they make will be reverted when restarting the tasks.

- After waiting on completion of previous collective operation, reschedule list of failed work items on the available nodes. Each work item can be restored through its closure (from checkpoint). Each item that reads input data has the data recovered by resilience component (from checkpoint). The rescheduling from checkpoint also cancels out undesired side effects on global data caused e.g. by orphan tasks.
- Recover the broken guard-protectee relation among the set of running nodes, for example by replacing the failed node with a node from a pool of available nodes (other strategies like shrinking the worker nodes, or restarting the failed node, are also viable). In addition, the protectee's protectee needs to be protected by the new node in order for a node not to lose its failed guard. Also, reset the list of work items on guard node.

We described in depth a recovery strategy in D5.1, Sect. 5.1, and a finite state machine with different states and transitions implementing a recovery in Sect. 5.2. The current design differs from this initial proposal: In D5.1, we assume a more loosely-coupled system, in which the guard node simply restarts a work item that has failed, but the work item goes through a recovery without further synchronization with the guard node. It is now clear that this is not possible – the AllScale Runtime is very tightly coupled, and a work item cannot be recovered in isolation from its parent item. Now, the guard node takes full control over the recovery of work items on a failed protectee, and retains a tight coupling to the recovered items by waiting on and applying its results in its recursion.

Also, it is possible but unrealistic with the current design of the AllScale Runtime to roll back work items. Among others, work items may spawn other work items across different localities, which would make a roll back a global and very challenging operation triggering chains of rollback events across the system. Accordingly, in the current resilience strategy, no work item ever needs to roll back to a previous step of the execution. It is only permitted to restart work items from the current state of a work item.

These and other limitations will be summarized in Sect. 4.

2.5 Granularity of Checkpoints

It is conceivable to implement an extremely fine-grained checkpoint strategy, in which each single work item is checkpointed by the protectee and accessible to the guard node. This granularity is extremely expensive. In contrast, it is desirable to perform checkpointing on a higher level of recursion, applying all the principles described so far without modification.

Consider, for example, the very important stencil code computation (Details in the context of AllScale project can be found in D2.3, Sect. 4). For simplicity, we focus on the field update and the potential to be parallelized, rather than the time loop. Its *pfor* formulation within the time loop is given as:

```
pfor(1, N – 1, [&](int x) {
       B[x] = A[x] + c * (A[x-1] + A[x+1] – 2*A[x]);
},wait_for_neighbors(ref));
```

In AllScale, the *pfor* itself is implemented via the recursive *prec* operator, part of the AllScale Core API (see D2.3). The *prec* operator can be implemented as (see D2.3, Sect. 2.2.3.1 for similar formulation):

```
auto init = prec(
       [&](const range& r){ return r.second - r.first <= 1; },
       [&](const range& r) { for( i = r.first … r.second ) B[x] = A[x] + c *
(A[x-1] + A[x+1] – 2*A[x]);},
       [&](const range& r, const auto& rec) {
              int mid = r.first + (r.second – r. first) / 2;
              return par(
                     rec(range( r.first, mid )),
                     rec(range( mid, r.second )),
                     );
       }
);
```

This implementation of *prec* will recursively break down the computation of the array B into left and right half – note that in the pfor formulation inside a time step, no dependencies exist, so the B[..] = … assignment can be parallelized. For example, consider that 2 nodes run the stencil code, each of them responsible either for the left or right half of the field array (see Figure 2). They further subdivide the array locally. Then it is possible to choose the granularity of checkpointing at a per-node level, as outlined in the figure. The work item of the node 1, computing B[0..3], is checkpointed (closure and input array A[0..3] included) on node 2 (the guard of node 1). The work item of the node 2, computing B[4..7], is checkpointed (closure and input array A[4..7] included) on node 1 (the guard of node 2). None of the finer-grained parallelization steps are checkpointed. If node 1 fails during the calculation of this work item, node 2 as its guard can restart it on any node based on the closure and input data needed for B[0..3] computation. The scenario of node 2 failing is similar, and its guard node 1 can recover the work item responsible for computing B[4..7].

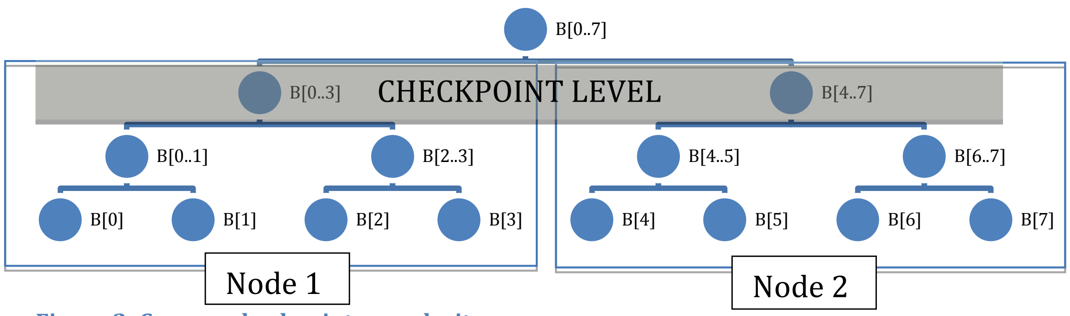

Figure 2: Coarser checkpoint granularity.

This checkpoint strategy is not explicitly coordinated. However, it is implicitly coordinated – note that the user specified *wait_for_neighbors* which enforces elements to wait for their neighbors before proceeding to the next time step.

3 Discrete-Event Simulator for Task Processing and for Resilience Protocol

A discrete-event simulator of the proposed resilience protocol is under development. An important challenge is replicating the realistic simulation of task processing the way the AllScale Runtime would process them. The implementation of a resilience strategy on top of that is described next.

We currently use the valuable help of SimPy (SimPy Documentation, 2017) as a discrete-event simulation framework. It provides a variety of primitives to implement discrete-event simulations of different types.

Our first goal for the simulator is to prototype the resilience protocol and verify its correctness under scenarios like node failures. The second goal is to use the simulator for the cost models of resilience, which are part of T5.3. Finally, all the concepts as implemented in the simulator need to be ported to the AllScale Runtime.

The simulator implements an abstraction of a distributed task-based runtime. The main entities currently implemented in the simulator are:

- **Runtime** - schedules simulator events, which correspond to work events, for execution.
- **Task** – a reference to a work item
- **Work Item** –a work item implementation has a *process* and *split* operations (e.g. a stencil work item or a Fibonacci work item). In the simulator, both of these functions schedule timeout events for the completion of their tasks (no actual computation is done). The *process* operation runs at the termination of recursion, where the *split* operation performs a recursion step.
- **Worker** – the worker is the abstraction for a physical node. It keeps a queue of scheduled/running/complete tasks. It handles the tasks on first-in first-out basis. Whenever the worker's queue is empty, it steals work from other workers. The worker also implements the resilience strategy we have outlined, including a queue of protectee tasks. If its protectee dies, a guard node triggers the recovery strategy.

Because of the importance of the worker logic to the resilience protocol, we include a code snippet that demonstrates the recovery described in Sect. 2.4. This recovery is a method of the **Worker** class:

```
def on_failed_protectee(self):
  # replace dead node with a new node
  # and establish guard-protectee relationship
  dead_worker = self.protectee
  new_worker = Worker(self.runtime, len(self.workers)+1)
  new_worker.guard = self
  new_worker.protectee = self.protecteesProtectee
  new_worker.protecteesProtectee = new_worker.protectee.protectee
  new_worker.protecteesTasks = self.protecteesProtectee.queue
  self.protectee = new_worker
  self.env.process(new_worker.run())

  # extract all top-level failed tasks from protectees tasks
  # their children can be ignored since they will be
  # re-run by their parent tasks
  root_tasks = self.extract_root()

  # delete all scheduled and running tasks
  for w in self.workers:
    w.purge(root_tasks)

  del self.protecteesTasks[:]

  # re-schedule all top-level failed tasks
  for t in root_tasks:
    t.parent.children.remove(t)
    new_worker.schedule(t)

  # inform all workers about the new worker
  for w in self.workers:
    if w !=dead_worker and w != new_worker:
      w.workers.append(new_worker)

  new_worker.workers = self.workers
```

Apart from the ongoing verification and testing of the simulator in itself, and of the resilience protocol implemented in the simulator, the main efforts are directed at:

- Improvements on the realistic tasks processing in the simulation.
- A simulation for checkpointing closures of work items and data (Sect. 2.2)
- Implementation of various checkpoint granularities based on use cases like stencil codes (Sect. 2.5)
- Extensive benchmarking of the total runtime for various scenarios to evaluate the cost of the resilience strategy.

The simulator of the resilience protocol is available under the project repository (access is detailed in D6.8). It is available to project partners; it can be made available to reviewers on demand.

4 Limitations of the Resilience Strategy

The resilience strategy has following limitations:

- The root node of the entire computation, as responsible for the initial tree of work items, is a single point of failure. This issue can be resolved via replication of that particular key element in the computation.
- The communication network may not fail, or else the detection and correction of failures may fail.
- The current protocol assumes that at most one node may fail at a time. This restriction can be removed if we extend the guard-protectee relationship to multiple guards and multiple protectees, and extend the recovery.

- The runtime may not roll back the computation of work items, due to the expensive and collective nature of such operation. The proposed protocol never rolls back work items of running nodes.
- The runtime is tightly coupled between work items, and therefore it is the responsibility of a guard node to recover the protectee task. A protectee cannot recover autonomously.

5 Conclusions and Future Work

The simulator is a support tool in order to prototype and estimate the cost of the resilience strategy, and is in line with T5.3 (developing a cost model) and the first phase of T5.4 (first prototype exploring resilience in shared memory). One difficulty we currently are facing is not the resilience protocol in itself, but the realistic implementation of task processing in a distributed setting, while mimicking the AllScale Runtime.
Once the resilience protocol has been verified and its cost estimated, the most efficient checkpoint/restart strategies have to be implemented, based on the cost model we derived using the simulator (T5.4). On the other hand, the simulator is simply a first step before implementing the resilience protocol as a resilience manager within the AllScale Runtime. This extensive task includes the second phase of T5.4, and T5.5.

6 Bibliography

SimPy Documentation. (2017). Retrieved from https://simpy.readthedocs.io/en/latest/