H2020 FETHPC-1-2014

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

D5.7 – Resilience Manager

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

Version: 1.0
Author(s): Kiril Dichev (QUB), Charles Gillan (QUB)
Date: 29/03/18

Due date:	PM30
Submission date:	day/month/year
Project start date:	01/10/2015
Project duration:	36 months
Deliverable lead organization	QUB
Version:	1.0
Status	Draft
Author(s):	Kiril Dichev (QUB)
Reviewer(s)	Stephane Monte (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	12/03/18	Frist draft	Kiril Dichev
0.2	15/03/18	Second draft, with plots clarifying guard-protectee ring, and the now different backup scheme	Kiril Dichev
0.3	27/03/18	Integrate Stephane's review	Kiril Dichev, Stephane Monte
1.0	29/03/18	Final version	Charles Gillan

Table of Contents

Index of Figures

Executive Summary

This deliverable describes the technical work performed as part of task T5.5 (Resilience Manager). T5.5 is a continuation of the efforts of T5.3 on Resilience Primitives, and of T3.6 on Resilience and Monitoring support. Also, T5.5 runs largely in parallel with T5.4, which focuses on Application-Specific Resilience Techniques.

It is important to note that the EC reviewers requested a clarification of the plan to focus on general recovery techniques, and not on some application-specific resilience techniques, such as application invariants, in previous tasks such as T5.4. This clarification is detailed, and was accepted, in a revised version of deliverable D5.5, submitted to the EC in June 2017. The focus on recovery from hardware failures, with closer interaction to the runtime, and less application-specific aspects, was motivated with a general preference to have this type of recovery from application developers. We should clarify that in T5.5, we also do not pursue algorithmic and application-specific resilience strategies and implement generic recovery strategies. This clarification is necessary, since T5.5, similarly to T5.4, lists both application-specific and generic approaches. The current deliverable focuses on the implementation of the resilience manager, as continuation of the proposed designs in D5.5, to target recovery from hardware failures, in a generic way. It also details some of the challenges we faced, and some required improvements in the protocol.

1 Introduction

The resilience manager needs to implement both the detection and recovery strategies, following the protocols detailed in D5.5. Extensive work was done to implement and test the recovery strategy in the resilience manager. This included a shared-memory prototype implementation in C++ in July 2017 in a 4-day face-to-face meeting in Innsbruck, and a first distributed-memory prototype of the resilience manager in an extended 2-week face-to-face meeting in Erlangen in August 2017. This is detailed in following sections.

2 Failure Detector

The failure detector is based on a distributed heartbeat protocol, and its design was outlined in the early phase of the project, in D5.1, Sect. 5. In addition, we should note that the MPI community has implemented a very similar protocol for failure detection, published in (Bosilca, 2017).

We implemented the failure detector as proposed, arranging all application processes in a logical ring (see Figure 1), in which every process has a guard and a protectee process. **Note that the guard-protectee ring is statically determined at initialization, and only changes if a failure happens**. In

particular, it is independent of the scheduler, and the application being scheduled.

We used application-level threads, which made the implementation easier. However, this decision does have some complications. It is possible for application threads to run on all cores for a period of time, which will delay the heartbeat signal. We observed this issue for coarse-grained work. As a solution, we allow the failure detection to be configurable, allowing to wait for a heartbeat for an extended period of time. Configuring an extended period before reporting a failure is essential when large chunks of coarse-grained work occupy all cores for a long time. Otherwise the failure detection reports a false positive, which has an adverse effect on the application run. Currently, the timeout period before signaling a failure is 10 seconds, but it is configurable, and sometimes we modify it to reduce the risk of false positives (particularly for very compute intensive applications). A heartbeat is periodically scheduled every 1 second from a process to its guard process.

The detector is part of the resilience manager at the moment (and not of the monitoring component) for simplicity, even though logically it is a monitoring functionality extension. It is part of the current AllScale runtime prototype (AllScale, 2018). The monitoring component currently provides entirely non-functional parameters, i.e. performance-related data.

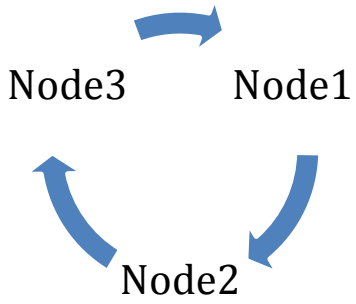

Figure 1: Logical ring for failure detection

3 Revised Recovery Protocol

In D5.5, we detailed a revised version of the recovery protocol (compared to D5.1), to be implemented in the resilience manager. While implementing the protocol, we realized that the implementation of a simulated protocol into a distributed runtime raises additional challenges, which are not evident in a simulator. The protocol needed to be further revised.

The protocol, as detailed in D5.5, Sect. 2.4, and demonstrated in a simulator in D5.5, Sect. 3, has shortcomings, which were apparent during actual distributed runs. Before a task can be scheduled via the AllScale scheduler on a remote locality (called *enqueing* a task in our terminology), this task needs to be recorded. The protocol previously outlined initiates a backup only once a task reaches the remote locality it is scheduled to run on. The backup was proposed to follow the static guard-protectee relationship along the logical ring of

localities, which was the same as the failure detector ring. This raises two important issues:

- In this design, the period between enqueing remotely a task, and scheduling it once it arrives, is not safe, and a node crash during that phase means that the task is lost and never recorded.
- In addition, this design has significant performance implications, since the cross-node latency needs to be carried twice: once, for transferring a task, and a second time, for backing it up.

In addition, in D5.5 we purge subtasks of failed tasks (called 'orphan' tasks). We have not implemented the purging of tasks in the current version, since this complicates the protocol and is not required when restarting work without associated global data. However, we may need to consider it during the integration of data item backups.

We realized the issues with the static guard-protectee backup scheme only during the actual work with the distributed scheduler; they were not apparent with the simulator, since the simulator carried no overhead in transferring tasks.

3.1 Backup Scheme

The modifications mean that we propose a modified back up scheme of work, which is not the same as the static guard-protectee detection, but is **scheduler-dependent**:

- Every time the distributed scheduler enqueues a task on a remote locality, this task is locally backed up in a *remotely_scheduled_tasks* list. Only then the task is submitted for scheduling on a remote locality.
- Every time a task signals its completion from the remote locality, the task is removed from the local *remotely_scheduled_tasks* list

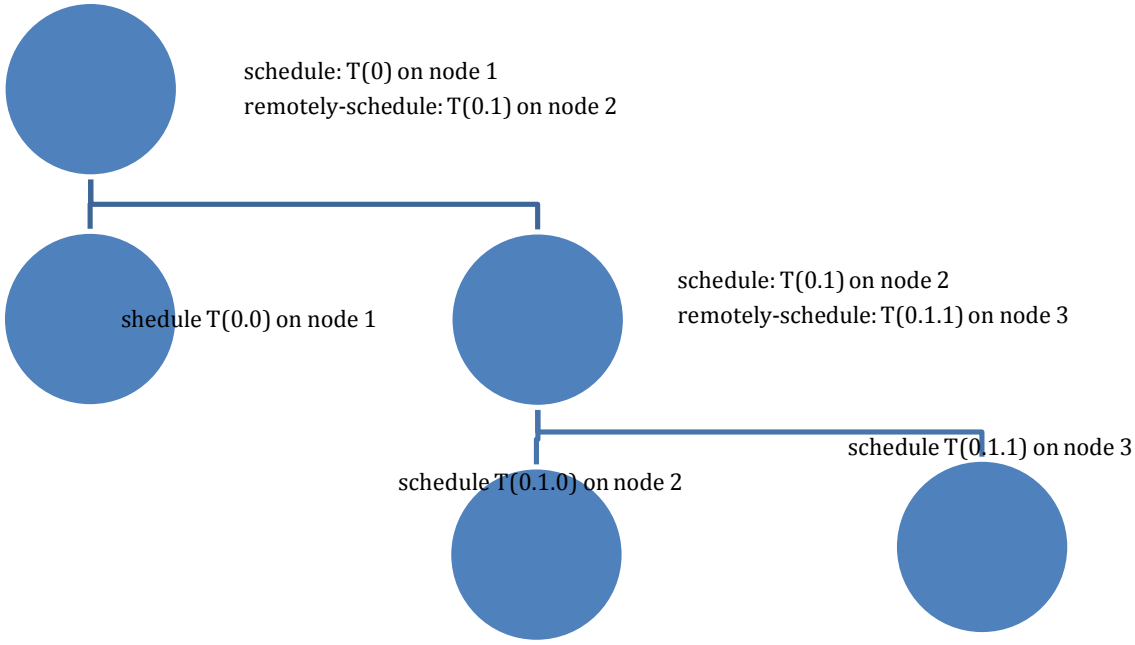

Figure 2: Illustration of new backup scheme for work and data items

We illustrate this scheme in Figure 2, on the example of a task 0, which is run on 3 nodes. A split happens into tasks 0.0 and 0.1. The distributed scheduler enqueues 0.1 for node 2, but records that on node 1 first. 0.0 remains locally scheduled. A similar split and back up happens further with 0.1, which is split into the locally scheduled 0.1.0, and the remotely enqueued 0.1.0, which is locally backed up on node 2 first.

3.2 Recovery Scheme

We propose the following modified recovery after a failure is detected:

- Only the guard node detects the failure of its own protectee. It triggers a global recovery on all surviving processes of
  - The guard-protectee ring
  - The list of available localities, which is needed by the distributed scheduler
- The guard node then broadcasts to all surviving nodes the ID of its protectee. All localities check their *remotely_scheduled_tasks*, and reschedule all tasks remotely scheduled on the failed locality

This protocol has been implemented and committed in the current release of the resilience manager (AllScale, 2018) for work items (but not yet for data items). It has been tested for 2 and 3 node settings for work items on the example of Fibonacci, but not for the associated data items yet, which is ongoing work.

4 Conclusions and Future Work

The future work will focus on two central aspects:

- Support for data items, as an extension of the current support for work items. This requires close collaboration with WP3. The support for data items should not require fundamental changes to the outlined protocol in Sect. 3.

- Verification of the protocol for the pilot applications. The current protocol is being tested on an artificial Fibonacci benchmark, in distributed mode.

5 Bibliography

AllScale. (2018). *AllScale Runtime*. Retrieved from AllScale Runtime: https://github.com/allscale/allscale_runtime

Bosilca, G. (2017). *A Failure Detector for HPC.* HAL. Retrieved from https://hal.inria.fr/hal-01453086