

Transparent Resilience for Chapel

Konstantina Panagiotopoulou



Thesis submitted for the degree of Doctor of Philosophy in the School of
Mathematical and Computer Sciences.

January, 2020

The copyright in this thesis is owned by the author. Any quotation from the thesis or use of any of the information contained in it must acknowledge this thesis as the source of the quotation or information.

TRANSPARENT RESILIENCE FOR CHAPEL

Konstantina PANAGIOTOPOULOU

January, 2020

© Konstantina PANAGIOTOPOULOU, 2020

Department of Computer Science

School of Mathematical and Computer Sciences

Heriot-Watt University

Edinburgh

Examination committee

Dr. Hans Vandierendonck, Queen's University Belfast

Dr. Manuel Maarek, Heriot-Watt University

Supervisors

Dr. Hans-Wolfgang Loidl, Heriot-Watt University

Prof. Sven-Bodo Scholz, Heriot-Watt University

Abstract

High-performance systems pose a number of challenges to traditional fault tolerance approaches. The exponential increase of core numbers in large-scale distributed systems exposes the growth of permanent, intermittent, and transient faults. The redundancy schemes in use increase the number of system resources dedicated to recovery, while the extensive use of silent-failure mode inhibits systems' capability to detect faults that hinder application progress. As parallel computation strives to survive the high failure rates, software shifts focus towards the support of resilience.

The thesis proposes a mechanism for resilience support for Chapel, the high performance language developed by Cray. We investigate the potential for embedded transparent resilience, to assist uninterrupted program completion on distributed hardware, in the event of component failures. Our goal is to achieve graceful degradation; continued application execution when nodes in the system suffer fatal failures. We aim to provide a resilience-enabled version of the language, without application code modifications. We focus on Chapel's task- and data-parallel constructs, and enhance their functionality with mechanisms to support resilience.

In particular, we build on existing language constructs that facilitate parallel execution in Chapel. We focus on constructs that introduce unstructured and structured parallelism and constructs that introduce locality, as derived by the Partitioned Global Address Space programming model. Furthermore, we expand the resilient support to cover data distributions on library-level.

The core implementation is on the runtime level, primarily on Chapel's tasking and communication layers; we introduce mechanisms to support automatic task adoption and recovery by guiding the control to perform task re-execution. On the data-parallel track, we propose a resilience enabled version of the Block data distribution module. We develop an *in-memory data redundancy* mechanism, exploiting Chapel's concept of locales. We apply the concept of *buddy locales*, as the primary means to store data redundantly and adopt remote workload from failed locales.

We evaluate our resilient task-parallel mechanism with respect to the overheads introduced by embedded resilience. We use a set of constructed micro-benchmarks to evaluate the resilient task-parallel implementation, while for the evaluation of resilient data-parallelism we demonstrate results on the STREAM triad benchmark and the N-body all-pairs algorithm, on a 32-node Beowulf cluster. In order to assist the evaluation, we develop an error injection interface to simulate node failures.

To my mom and Christos

Acknowledgements

The completion of this PhD has been a challenging but rewarding journey. It is marked by the succession of periods of accomplishment and periods of frustration, when things did or did not work out. It has been a time of exploration of knowledge and definitely a lesson on how not to quit when things look bad. This thesis has been completed thanks to the support of many people.

First of all, I need to thank my supervisor Dr Hans-Wolfgang Loidl, for offering me the opportunity to do research. He has dedicated time and effort in providing guidance throughout this period and has always been present when needed. His patience throughout this process, especially after I got a full-time job and moved to Greece, has been of great value. I cannot thank him enough for his persistence to keep me focused and for the time he spent reviewing drafts of this thesis.

I also thank Prof. Sven-Bodo Scholz, Dr Michele Weiland, Prof. Greg Michaelson and Dr Chris Fensch, who gave advice early in this PhD, when the main focus was still fuzzy. I also thank the members of my examination committee, Dr Hans Vandierendonck and Dr Manuel Maarek, for their constructive comments to improve the thesis.

I am also grateful to the Department for supporting me for 3 years through the James Watt Scholarship program and the Scottish Informatics and Computer Science Alliance (SICSA) for sponsoring me to fulfil a 3-month research internship with Cray in Seattle, WA.

I thank the people from Cray, especially Brad Chamberlain and Tom MacDonald for their guidance. Brad has been great help when I struggled with the internal details of Chapel, but it is his vision and enthusiasm to make people adopt the language that inspired me the most.

Also, I thank my fellow PhD students, friends and members of the Dependable Systems Group, in particular, Evgenij Belikov and Prabhat Tottoo, with whom I shared an office, but most importantly, a relaxed environment to work, and debate each others research and beer tastes.

I am grateful to my family, especially my mom Liana, who would check on me every day and try to be supportive. A big shout-out to my Rose Street flatmates, Kika and Pepi, whom I consider family. Also, a big thanks to all my Edinburgh friends for the fun times we shared.

Last but not least, I am forever grateful to Christos who has been by my side throughout this process, though we were located in different timezones for the most part. His unconditional love and support throughout these years and his constant nudging to get things done have been of tremendous importance to the completion of this work.

Research Thesis Submission

Please note this form should be bound into the submitted thesis.

Name:	Konstantina Panagiotopoulou		
School:	Mathematical and Computer Sciences		
Version: <i>(i.e. First, Resubmission, Final)</i>	Final	Degree Sought:	PhD in Computer Science

Declaration

In accordance with the appropriate regulations I hereby submit my thesis and I declare that:

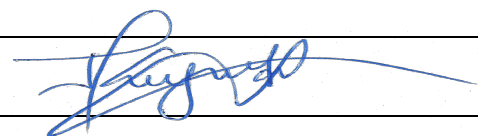
1. The thesis embodies the results of my own work and has been composed by myself
2. Where appropriate, I have made acknowledgement of the work of others
3. The thesis is the correct version for submission and is the same version as any electronic versions submitted*.
4. My thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available for loan or photocopying and be available via the Institutional Repository, subject to such conditions as the Librarian may require
5. I understand that as a student of the University I am required to abide by the Regulations of the University and to conform to its discipline.
6. I confirm that the thesis has been verified against plagiarism via an approved plagiarism detection application e.g. Turnitin.

ONLY for submissions including published works

Please note you are only required to complete the Inclusion of Published Works Form (page 2) if your thesis contains published works)

7. Where the thesis contains published outputs under Regulation 6 (9.1.2) or Regulation 43 (9) these are accompanied by a critical review which accurately describes my contribution to the research and, for multi-author outputs, a signed declaration indicating the contribution of each author (complete)
8. Inclusion of published outputs under Regulation 6 (9.1.2) or Regulation 43 (9) shall not constitute plagiarism.

* Please note that it is the responsibility of the candidate to ensure that the correct version of the thesis is submitted.

Signature of Candidate:		Date:	06/01/2020
-------------------------	---	-------	------------

Submission

Submitted By <i>(name in capitals)</i> :	
Signature of Individual Submitting:	
Date Submitted:	

For Completion in the Student Service Centre (SSC)

Limited Access	Requested	Yes		No		Approved	Yes		No	
E-thesis Submitted <i>(mandatory for final theses)</i>										
Received in the SSC by <i>(name in capitals)</i> :						Date:				

Contents

List of Tables	vi
List of Figures	viii
List of Abbreviations	xi
List of Publications	xii
1 Introduction	1
1.1 Thesis Statement	3
1.2 Contributions	6
1.3 Context and Limitations	8
1.4 Thesis Structure	11
2 Background	13
2.1 Distributed Systems Terminology	13
2.1.1 Concurrency and Parallelism	14
2.1.2 Parallel Architectures	14
2.1.3 High-Performance Systems towards the Exascale	17
2.2 Dependability of Distributed Systems	22
2.2.1 Properties of Dependable Systems	22
2.2.2 System Failures on Distributed Systems	26
2.2.3 Failure Rates	28
2.3 Resilience Principles	30
2.3.1 CAP Theorem	30
2.3.2 Software Level Fault Tolerance	31
2.3.3 Hardware Level Fault Tolerance	35

2.4	Resilient Store	37
2.4.1	External File Systems	39
2.4.2	In-Memory Replication Techniques	40
2.5	Related Work	41
2.5.1	Languages with resilience capabilities	41
2.5.2	Runtimes with resilience capabilities	50
2.6	Summary	57
3	The Chapel Parallel Programming Language	58
3.1	A Brief History of Chapel	58
3.1.1	Partitioned Global Address Space Programming Model	59
3.2	Chapel Base Language	61
3.3	Chapel’s Approach to Parallelism	63
3.4	Task Parallelism in Chapel	63
3.4.1	Unstructured Task Parallelism	63
3.4.2	Structured Task Parallelism	64
3.4.3	Remote Task Spawning	64
3.4.4	Synchronisation and Atomicity	66
3.5	Data Parallelism	67
3.5.1	The <code>forall</code> loop	67
3.5.2	Domains and Arrays	68
3.6	Locality	70
3.6.1	The <code>locale</code> type	70
3.6.2	The <code>on</code> construct	71
3.6.3	Domain Maps, Layouts, and Distributions	72
3.7	Chapel’s Runtime Environment	74
3.7.1	The Communication Layer	75
3.7.2	The Tasking Layer	78
3.8	Chapel’s Resilience Directions	83
3.9	Summary	84
4	Resilient Task Parallelism	85
4.1	Foundations of Resilience	85

4.1.1	Data Structures and Resilient Store	87
4.2	Design Aspects	88
4.2.1	Assumptions	89
4.2.2	Failure Notification	90
4.2.3	Tasking Interface	91
4.2.4	Task Synchronisation	91
4.3	Communication Protocol Extensions	93
4.3.1	Serial Remote Task Execution	93
4.3.2	Distributed Remote Task Execution	97
4.3.3	Optimisations and extensions	105
4.4	Testing Interface	107
4.4.1	Micro-benchmarks	107
4.4.2	Failure Injection Mechanism	110
4.4.3	Experimental Setup	112
4.4.4	Benchmarking and Threshold Standards	114
4.5	Evaluation of the Resilient Task Parallel Implementation	117
4.5.1	Blocking Fork	117
4.5.2	Non-blocking Fork	122
4.6	Summary	136
5	Resilient Data Parallelism	139
5.1	The Block Distribution	139
5.1.1	Overview	141
5.1.2	Implementation Details	141
5.1.3	Leader-Follower Iterators	144
5.2	Resilient Block Distribution: Data redundancy	145
5.2.1	Data structure additions to the Block distribution	145
5.2.2	Buddy locale configuration	147
5.2.3	Initialisation of the Block distribution	147
5.2.4	Redundant domain initialisation	149
5.2.5	Redundant array initialisation	151
5.3	Resilient Block Distribution: Recovery and adoption	153
5.3.1	Iteration over redundant data: <code>Forall</code> loop	153

5.3.2	An example of multi-failure recovery	155
5.3.3	Failure Tolerance Threshold	157
5.3.4	Parallelisation of Recovery	160
5.4	Evaluation of the Resilient Data Parallel Implementation	164
5.4.1	Experimental setup	165
5.4.2	STREAM: Sustainable Memory Bandwidth in High Performance Computers	165
5.4.3	N-body: approximation of particle motion	177
5.5	Portability to Other Predefined Distributions	183
5.6	Summary	184
6	Conclusion	188
6.1	Summary	188
6.2	Contributions	191
6.3	Limitations and Future Work	192
A	Supportive Data	198
A.1	Combinations of failures	198
A.2	Snapshot of the internal topology of a Beowulf node	200
A.3	HDFS integration in Chapel	201
A.4	Mandelbrot microbenchmark	202
A.4.1	Mandelbrot Chapel code	202
A.4.2	Mandelbrot Result Fractal Sets	204
A.5	STREAM: Historical measurements	205
B	Benchmarks and Data Sets	206
B.1	Microbenchmarks	206
B.1.1	Serial distributed micro-benchmarks	207
B.1.2	Task parallel micro-benchmarks: <code>begin+on</code>	210
B.1.3	Task parallel micro-benchmarks: <code>cobegin+on</code>	213
B.1.4	Task parallel micro-benchmarks: <code>coforall+on</code>	217
B.2	Data parallel micro-benchmarks	218
B.2.1	STREAM triad	218
B.2.2	N-body all-pairs	223

CONTENTS

B.3 Celestial bodies input datasets	227
References	230

List of Tables

2.1	Key Research Areas Towards Resilience	18
2.2	Classification of fault-tolerance techniques	28
2.3	Probability of Faults	30
2.4	Comparison of Resilient X10 and Resilient Chapel implementations .	47
2.5	Cross-language/ framerwork overview of resilience approaches	56
3.1	Index partitioning formula of the Block distribution	73
3.2	Overview of Chapel’s runtime system layered architecture.	75
3.3	Internal properties of the tasking layer of Chapel’s runtime system . .	82
4.1	Example configuration of buddy and guest locales	105
4.2	Exchange of communication messages across locales	132
5.1	Index partitioning formula of the Block distribution	140
5.2	Layout of remote index sets of the Block distribution	148
5.3	Runtime level recovery strategy from multiple failures	158
5.4	A hypothetical data-parallel program executing on 4 locales without and with failure.	161
5.5	Average runtime of the Mandelbrot data-parallel micro-benchmark on a set of different failure injection configurations, with excution on 4 locales.	162
5.6	Vector kernels of the STREAM benchmark	166
5.7	STREAM triad full measurements set	168
5.8	Local and remote block sizes persisted per locale for a configuration with two buddies	170

5.9	STREAM full measurement set, runtime regular Chapel, resilient Chapel without failures and resilient Chapel on an increasing number of failures	176
5.10	Complete measurement set of absolute overhead results of the N-body execution with 5K and 10K iterations and 20 and 40 bodies on an increasing number of failures	180
5.11	Complete measurement set of relative overhead results of the N-body execution with 5K and 10K iterations and 20 and 40 bodies on an increasing number of failures	182
A.1	Mandelbrot fractal sets with 2010x2010 input array sizes executing on 4 locales without and with a single failure.	204
A.2	STREAM triad Memory Bandwidth Results	205

List of Figures

2.1	Distributed Memory Architectural Topology	15
2.2	Shared Memory Architectural Topology	16
2.3	Laprie’s taxonomy of dependability	26
2.4	Categorisation of causes of faults	27
2.5	Resilience Rate Projection to the Exascale	29
2.6	Communication flow of in-memory replication of tasks	38
2.7	The Hadoop Distributed File System Architecture	39
3.1	Program control flow of <code>on</code> , <code>begin</code> , <code>cobegin</code> and <code>coforall</code>	65
3.2	Active Message communication between two parties	77
4.1	Communication flow of in-memory replication of tasks	86
4.2	Execution flow of the blocking remote <code>on</code> statement	94
4.3	Execution flow of the resilient blocking <code>on</code> statement without and with a single failure	95
4.4	Execution flow of a remote non-blocking <code>on</code>	98
4.5	Example of two failures in a multi-locale task-parallel Chapel program	100
4.6	Execution flow of a resilient non-blocking fork	101
4.7	Execution flow of a resilient non-blocking fork task recovery with multiple buddy locales	104
4.8	Constructed micro-benchmarks for functionality and overhead testing	107
4.9	Task spawning flow within a <code>coforall+on</code> loop	109
4.10	Resilient X10: Sparse matrix dense vector multiply calculation	115
4.11	Success rates of the failure recovery mechanism for the resilient block- ing fork implementation	118
4.12	Performance results of the resilient blocking fork implementation . . .	120

4.13	Overhead results of the resilient blocking fork implementation	121
4.14	Overhead results of the resilient non-blocking <code>begin+on</code> fork imple- mentation	124
4.15	Runtime results of the resilient non-blocking <code>begin+on</code> fork imple- mentation	125
4.16	Average, minimum, maximum runtime results for <code>begin+on</code> per dis- tinct locale	127
4.17	OpenMP's fork-join execution model	128
4.18	Overhead results of the resilient non-blocking <code>cobegin+on</code> fork im- plementation	129
4.19	Runtime results of the resilient non-blocking <code>cobegin+on</code> fork imple- mentation	130
4.20	Average, minimum, maximum runtime results for <code>cobegin+on</code> per distinct locales	131
4.21	Runtime results of the resilient non-blocking <code>coforall+on</code> fork im- plementation	133
4.22	Overhead results of the resilient non-blocking <code>coforall+on</code> fork im- plementation	134
4.23	Runtime results for <code>coforall+on</code> on 16 locales with an increasing number of failures	135
4.24	Average, minimum and maximum runtime results for <code>coforall+on</code> on 16 locales with an increasing number of failures	136
5.1	Layout of the internal classes of the Block distribution	142
5.2	Layout of indices of the BlockDom class for a two-dimensional array .	146
5.3	Layout of the internal classes of the Resilient Block distribution . . .	149
5.4	Stepped timeline of multiple locale failures and recovery	156
5.5	Recoverable combinations on a sample configuration of 8 locales with 2 buddies	159
5.6	A hypothetical data-parallel program executing on 4 locales with a single failure and parallelised recovery on the buddy locale.	163
5.7	Runtime results of the STREAM triad on an increasing number of failures	167

5.8	Runtime results of the STREAM triad, baseline and resilient version without failures	170
5.9	Runtime results of application code of the STREAM triad on an increasing number of failures	174
5.10	Runtime results of the STREAM triad on 12 locales with up to 8 failures	176
5.11	Runtime results of the Nbody executions for 20 and 40 bodies with 5K and 10K iterations on an increasing number of failures	178
5.12	Absolute overhead results of the N-body execution for 20 and 40 bodies with 5K and 10K iterations on an increasing number of failures	179
5.13	Relative overhead results of the N-body execution for 20 and 40 bodies with 5K and 10K iterations on an increasing number of failures . . .	181
A.1	Combinations of Locale Id's with a configuration of 8 target locales and 2 buddies per locale	199
A.2	Internal topology of a Beowulf node, produced with the hwloc package.	200

List of Abbreviations

AM	Active Messages
BER	Backward Error Recovery
CAF	Co-Array Fortran
CAP	Consistency, Availability, Partition Tolerance
DPE	Dead Place Exception
DSP	Digital Signal Processor
FER	Forward Error Recovery
FLOPS	Floating Point Operations per Second
FMA	Fused Multiple-Add
HDFS	Hadoop Distributed File System
HEC	High-End Computing
HPC	High Performance Computing
KDD	Knowledge Discovery in Databases
LLC	Last Level Cache
MPI	Message Passing Interface
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
NUMA	Non-Uniform Memory Access
OFI	Open Fabric Interface
OpenMP	Open Multi-Processing
PGAS	Partitioned Global Address Space
POSIX	Portable Operating System Interface for Unix
RDD	Resilient Distributed Datasets
ROC	Recovery-Oriented Computing
SCC	Singlechip Cloud Computer
SDC	Silent Data Corruption
SEU	Single Event Upsets
SPOF	Single Point of Failure
UMA	Uniform Memory Access
UPC	Unified Parallel C

List of Publications

Part of the work presented in the thesis is derived from the following publications :

1. Panagiotopoulou, Konstantina and Hans-Wolfgang Loidl(2015). *Towards Resilient Chapel: Design and Implementation of a Transparent Resilience Mechanism for Chapel*. In: Proceedings of the 3rd International Conference on Exascale Applications and Software. EASC 15. Edinburgh, UK: University of Edinburgh, pp. 8691. ISBN: 978-0-9926615-1-9.

The paper describes the design and the prototype implementation of transparent resilience support for Chapel. We discuss the design considerations of basic resilience support embedded in Chapels runtime system. We use automatic task adoption and data redundancy as the focal points of our implementation. We provide preliminary results on constructed micro-benchmarks, focusing on the functionality of the blocking fork operation using patterns of deep nesting of tasks.

2. Panagiotopoulou, Konstantina and Hans-Wolfgang Loidl (2016). *Transparently Resilient Task Parallelism for Chapel*. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 15861595. DOI:10.1109/IPDPSW.2016.102

The paper focuses on the design and implementation of transparent resilience for Chapel’s task parallelism. Our goal is to ensure program termination in the presence of hardware failure of one or multiple nodes in the system. We provide evaluation results on a set of five synthetic micro-benchmarks. We discuss the design challenges and demonstrate low overheads of our mechanism on a commodity Beowulf cluster, with a maximum of 2.6% in the failure-free case and a maximum of 4.64% for recovery of a single failure.

Following the final submission of the thesis, we plan to contribute the resilient version of Chapel to the official Chapel repository¹ to make it available for user testing and possible extensions. At the moment, there are no further publication plans.

¹The official Chapel repository on Github: <https://github.com/chapel-lang/>

Chapter 1

Introduction

Hardware is becoming increasingly parallel, complex and error-prone. Parallel programs are required to efficiently exploit the power of the underlying hardware. Novel parallel languages are developed to exploit the computational power, with two prevailing approaches; either by requiring the programmer to handle the low-level management of threads, task synchronisation, communication and data access or by employing layers of abstraction; high-level language components and middleware.

Both approaches come with advantages and disadvantages; writing a parallel program using a low-level language can be challenging to debug and requires more effort to reach a correct solution, since all parallel aspects and possible pitfalls need to be taken into consideration. The programmer is responsible for the coordination of parallel tasks, while avoiding race conditions and deadlocks. On the other hand, using a language equipped with high-level components requires less effort to get to an initial solution, while the part of performance tuning becomes complex as the high-level abstractions often obscure the causality of poor performance or introduce overhead.

The Partitioned Global Address Space (PGAS) programming model emerged in the early 2000's, and is an attempt to simplify parallel programming while providing control over the low-level components of the programming stack via powerful high-level abstractions, with the general aim of increasing the programmers' productivity on massively parallel machines. PGAS programming languages allow application code to refer to any lexically visible variable, residing in local or remote memory. The runtime system and the compiler are responsible for the coordination of data

access operations over the communication layer.

Nonetheless, even after a careful choice of programming language and given the non-trivial effort dedicated to performance tuning, a computationally-intensive application is still prone to fail due to the intrinsic poor reliability of the hardware components that make up the underlying large-scale system on which the application executes. As the component count increases to the thousands the probability of failures increases as well. The rate of fatal failures increases further when executing on commodity hardware. Research results show that such failure events are clustered temporally and spatially (Hacker, Romero, and Carothers, 2009), thus increasing the complexity of both the failure analysis and the design of parallel applications. Such failures cost computing time and power resources.

Central to this research is *resilience* and its implementation in the runtime of a Chapel, a representative language of the PGAS programming model. Resilience is the ability of a system to execute in a timely manner in the presence of failures. In the context of a programming language, the produced programs are expected to maintain the properties of correctness and timely execution. Correctness refers to the avoidance of data corruption, while timely execution refers to performance and efficiency considerations of the executing program. Failures are defined as both hardware and software events that impede the progress of the application program. A relevant term to resilience, *fault tolerance*, is more commonly used to describe the *ability to tolerate failures and take recovery action* using mechanisms that are segregated from the application or are provided by a third party software or hardware component, as documented by Morin and Puaut, 1997 and Gärtner, 1999.

A fair amount of research and bibliography (Brewer, 2000; Cristian, 1991; Lamport, 1977 and Patterson, Brown, Broadwell, et al., 2002) has been dedicated to recovery strategies to alleviate the effect of failures during execution of large-scale applications. Most prominently, we find techniques that persist the application’s state in memory and restart the computation from logs, following a failure, commonly categorised as *checkpoint-restart*. Variations of checkpoint-restart are used extensively today, especially in data intensive applications. The issue then remains, how expensive it is to pause the computation in order to capture snapshots of the application state and what is the performance overhead when recovering from a

checkpoint. Other questions also arise, for example, for how long the system remains in an erroneous state; how can we ensure that a checkpoint-ed state is valid and what is the performance penalty of using such mechanisms when no failures occur. in the following chapter (Chapter 2) we review resilience techniques and recovery mechanisms.

In our design we propose a different approach, based on the principles of *forward error recovery*, *transparent resilience* via the technique of *graceful degradation*. Forward error recovery, as opposed to checkpoint-restart, employs mechanisms to handle failures while the execution progresses. Transparency refers to the ability of the resilient system to handle failures without user-assistance; in the context of Chapel this directly translates to enabling resilience without changes on application-level. Finally, graceful degradation aims to preserve the *liveness* (Lamport, 1977) of the system, allowing the program to progress with less compute nodes as failures occur during program execution. Our design focuses on runtime-level mechanisms for task-adoption and recovery from in-memory stored data, without the use of checkpointing.

1.1 Thesis Statement

The key hypothesis of the thesis is that it is possible and beneficial to provide support for transparent resilience embedded in Chapel (Chamberlain, Callahan, and Zima, 2007); implemented solely within the runtime system and on library-level. No program changes should be necessary to enable resilience on application-level.

In particular the thesis is organized into two main parts; we look at *task parallelism* and its implementation within the communication and the tasking layers of the language and at *data parallelism*; implemented within Chapel’s *data distributions* on module-level and assisted by the runtime system.

Our design addresses resilience for Chapel, one of the first generation PGAS languages. Chapel, is actively developed by Cray as an open-source project and it is designed from first principles for High Performance Computing. It aims to cover general parallelism needs, it incorporates the basic principles of the programming model

with parallelism and locality as its focal points, and builds on a multi-resolution design allowing the use of high- and low-level features, as required by the application programmer.

Partitioned Global Address Space (PGAS) programming languages were motivated by the need to increase productivity in parallel programming, by exposing details of the low-level system on application-level in a succinct elegant manner ; such as task- and data-locality, and by providing control over the number and the synchronisation of executing tasks. As such, programmers can reason about the location of data, statically or dynamically, using language semantics or a number of available execution-time queries.

The work presented in this thesis is an exploration of techniques in order to allow parallel programs to progress and complete execution, producing correct results when components —nodes of a large-scale system, experience fatal failures during execution. We apply the key design concepts of *task-adoption* and *task-recovery* by migrating the computation and the data on the remaining system nodes. We also apply data redundancy techniques and we look into maintaining an up-to-date image of the system, including status checks and data updates, without the use of checkpointing.

We argue that the globally-visible data used in PGAS can be exploited to provide resilience without requiring programming effort or modifications on application level. Parallel applications execute till completion and produce correct results in cases where the system experiences failures due to a node malfunction, an unscheduled node shutdown or a network partition.

To evaluate our design and implementation we provide experimental results on a set of constructed micro-benchmarks, a linear algebra benchmark used as a stress-test and a non-trivial application. We are primarily concerned with correctness, timely execution and the overheads introduced by the adoption, recovery and data redundancy mechanisms. We evaluate overheads both during normal execution, when no failures occur, and for a range of failure scenarios, including single and multiple node failures during execution.

The research aims to answer the following main research questions:

- Built-in resilience: Can a resilience mechanism be integrated in the internal design of a parallel programming language? What mechanisms of the lower-level runtime system can be employed to assist recovery?

Erlang is the most widely-used language providing built-in resilience, but it started as a purpose-specific language for telephone networks. Erlang employs recovery when worker threads fail. Can this idea be ported to a larger scale, where system components -or entire nodes- fail? And how are PGAS languages –and in particular, Chapel– good candidates for a parallel language with built-in resilience? We aim to provide solutions for embedding resilience in Chapel and we advocate that designing a language with resilience in-mind from the early stages of the development is the best approach towards a complete solution.

- Transparency: Can a resilience mechanism be efficient without user-assistance and manual tuning? In particular, is transparent resilience beneficial in the context of a general-purpose parallel programming language?

Fault-tolerance and resilience mechanisms are notoriously difficult to understand, use in programs and maintain, this is the reason behind the success of external frameworks that provide relevant functionality. On hardware level, processor design towards hardware-level fault tolerance has the potential of eliminating failures. On application level though, it is often difficult to acknowledge a failure; on the programmer's side, a component failure is not immediately evident and most programmers will seek the reason for abrupt termination or incorrect results within their application code. On the other hand, alternating between application code and resilience code requires in-depth analysis of the application's behaviour and eliminates the portability potential across applications. We seek to substantiate the benefits of language built-in resilience, and minimize the overheads of the added computation and data management, taking advantage of the global namespace characteristics. In PGAS, we advocate that providing parallel programmers with a complete built-in solution for resilience is an added benefit towards writing efficient code that executes despite systemic disruptions.

- Application programming: When failures occur in a system with a given num-

ber of resources, is it beneficial for an application to attempt to recover the failed tasks and proceed with less computational power?

In large scientific computations we often encounter precision tolerance limits in the expected results. This practice is used as a trade off to the large execution times expected and the subsequent higher possibility of failure. We seek to answer whether a resilience-embedded language can provide comparable execution times to a non-resilient version in the context of a failure-free execution without sacrificing the precision of the computation.

1.2 Contributions

The main research contributions of this thesis are:

- a design approach to built-in resilience on PGAS parallel programming languages;
- an implementation of transparent resilience in the lower-level runtime system and within Chapel’s standard libraries;
- the experimental evaluation of the proposed mechanism on micro-benchmarks, as well as larger-scale applications on a distributed system.

The detailed contributions of the thesis are presented below:

1. On language level, the design of a transparent resilience framework for Chapel’s task-parallel language constructs : *begin*, *cobegin* and *coforall* loop (Chapter 4) and a *blocked distribution* module with resilience capabilities, one of the standard data-parallel modules used in Chapel (Chapter 5). The resilience design covers all task parallel constructs offered in the language, while Chapel offers in total four standard distributions; *Block*, *Cyclic*, *BlockCyclic* and *2D Dimensional*. Towards the end of Chapter 5 (Section 5.5) we discuss the main steps required to port our resilient implementation to other predefined distributions.
2. On system-level, an in-memory data redundancy mechanism to assist recovery using data copying and taking advantage of Chapel’s data locality principles. This mechanism utilizes the notion of *buddies* that store data owned by affiliated nodes, as an alternative to the use of external file systems for data

redundancy. We provide the algorithm for building buddy sets; while advanced users can implement their own algorithms, based on application and systemic requirements.

3. An implementation of resilience for task-parallel features, integrated within the runtime system, with modifications on the communication and tasking layers (Chapter 4). We provide an evaluation of the implementation on a set of constructed micro-benchmarks and we discuss future optimisations.
4. The implementation of a resilient version of the *blocked data distribution*, using higher- and lower-level recovery strategies. We provide an evaluation of the overhead introduced by the resilience mechanism in configurations with single and multiple failures. We discuss the mechanism's behaviour during application execution and propose possible optimisations to improve efficiency. We analyse the results from testing on two applications: the STREAM benchmark and the N-body algorithm (Chapter 5).
5. A fault injection mechanism for Chapel programs that simulates failures of Chapel *locales* in a distributed setup; as an auxiliary functionality for testing, due to the lack of a hardware or software mechanism to simulate component failures in a controlled manner. In an effort to simulate node failures more accurately, we require that locales that suffer failure remain idle after the failure and till execution completion. To this end, we have added a local status check, that works as a *hook* into the runtime system's components, potentially extensible to accommodate added functionality, for example a dynamic load-balancing mechanism. The fault injection mechanism is also applicable to other languages or systems that utilize GASNet as the lower-level implementation of their communication layer (Chapters 4 and 5).
6. A review of resilience and fault tolerance in high performance systems, covering fault detection and fault avoidance techniques. We provide a critical review of predominantly used recovery mechanisms for backward and forward error recovery; we also review existing languages and runtime systems with resilience capabilities (Chapter 2).

1.3 Context and Limitations

This work aims to provide a resilient framework embedded in the runtime of a general-purpose parallel language. Though we aim for a general solution, the distinct characteristics of application programs may be incompatible with the implementation of resilience in this work.

We provide a high-level description of the types of computation that are not currently covered by the resilient Chapel implementation, while we also refer to limitations that are inherited from the language itself, in order to clearly define the scope of the work. We also discuss limitations of the implementation and briefly explain the rationale or reasoning behind the corresponding design decisions.

- As Chapel employs high-level abstractions and a layered runtime stack, the state of the computation in the sense of the progress of an executing thread, is not exposed outside the threading layer to the upper layers; primarily the tasking and the communication layer where the resilience mechanism is implemented. The accessible information on task-level is restricted to whether a task has begun or terminated execution or whether it remains in idle state. To this end, the resilient mechanism is unable to pick-up computation from a specific point, on the subsequent recovery after a failure; we instead execute the calculation from the beginning. We require that tasks are *atomic* in the context of resilience; either they complete successfully or the entire task is re-started.

This design choice is mainly driven by two factors. Firstly, assuming we were able to retrieve detailed information on the progress of a thread, we believe that the added complexity would not necessarily decrease the overheads of resilience. It would require added logic and data structure management to persist such information and added communication to maintain the system up to date. Secondly, as tasks are the main means for introducing parallelism in Chapel, we consider the tasking layer a better candidate for the implementation of resilience. One of our main goals is transparent resilience and part of that is to preserve the semantics of the base language; by continuing to provide the programmer with constructs to reason about parallelism –in the form of tasks.

- Chapel programs that use locality features as part of the computation cannot be covered by the implementation, since their correctness property cannot be maintained after migration on a different location in the system. A simple example is demonstrated in Listing 1.1. The parallel forall loop iterates over the elements of the distributed array A and performs an addition of the element's value and the locale's identifier on which the execution takes place. After a failure, the value `here.id` will be modified due to the migration of the computation to a different place in the system (recovery), thus the correctness of the result will be compromised. Although, relying on such locality information is a bad programming practice, it is not currently prohibited by the language or the type system.

```

1 forall a in A do
2   a = a + here.id

```

Listing 1.1: A parallel forall loop with locality-based computation

Locales are Chapel constructs that define a place in the system; for a distributed system they are commonly mapped to nodes in a cluster. The `here` keyword is Chapel's syntactic equivalent of `MPI_Get_processor_name()` in MPI or the equivalent of `node()` in Erlang.

Chapel's parallelism assumes homogeneous sets of processors, and nodes by extension. The idiom of specific placement is potentially applicable to programs that use non-homogeneous cluster nodes, for example when using a co-processor attached to a node, or when a node handles the connection to an external file system (for example, when implementing the HDFS Chapel interface). As a consequence of task migration during failure recovery, these programs are not covered by our design of resilience. On the other hand, any programs that depend on connectivity to external systems or special types of nodes would possibly be inefficient (if not erroneous) to continue executing when connectivity is lost. In other words, in order to take advantage of the resilient mechanism, applications with placement requirements, are required in this context to use systems with *redundant special components* and provide

a parametric reconnection strategy on application level.

A detailed discussion of Chapel constructs is provided in Chapter 3. In Chapter 5 we provide an in-depth discussion of data parallelism, including parallel loops and distributed arrays.

- Real faults in HPC platforms are the subject of extended study (Schroeder and Gibson, 2010, El-Sayed and Schroeder, 2013, Bosilca, Bouteiller, Guermouche, Herault, Robert, Sens, and Dongarra, 2016, Gainaru, Cappello, Snir, and Kramer, 2012) and their types and correlations are non-trivial to pinpoint. Log analysis indicates spatial and temporal correlations between failures and a correlation between higher workloads and high failure rates. In Di, Guo, Pershey, Snir, and Cappello, 2019, we find a detailed analysis on a number of large scale systems that demonstrate a MTBF between 1 and 42 hours. In the context of Chapel though, nodes are abstracted by locales. The spatial distribution of node faults within a cluster is not directly mapped to the failure distribution on locales within a Chapel program. As such, from a programmer’s perspective failures occur in random patterns.

In the presence of real failures, the evaluation of a resilient mechanism is complex. In this work, we introduce a custom mechanism to simulate failures pseudo-randomly, but also introduces some limitations. We demonstrate failure recovery for failures that are clustered in the beginning of the execution. The current implementation of the testing framework is parametric as to the number of failures we introduce in the system, but not with respect to the point in time that the failures are introduced. For example, when two failures are injected on two random locales in the system, the failure injection mechanism executes alongside the application, performs a sign-on on each remote locale and sends a `Unix`-level signal. The injection is serial, so the number of failures to introduce affects the time that a failure is realised by the system, nevertheless failure signals occur earlier in the execution and are bound by a short time frame.

1.4 Thesis Structure

- *Chapter 2: Background*

The chapter provides a detailed review of parallelism and resilience. We expand on the main ideas of concurrency and parallelism and discuss the most widely used memory models; shared and distributed memory. We provide the background for fault tolerance focusing on the concepts of dependability, failures and faults, and we review failure metrics and software-level fault tolerance techniques. Towards the end of the chapter we review programming languages and frameworks that provide or aspire to provide fault-tolerance capabilities and we draw on their main similarities and differences to this work.

- *Chapter 3: The Chapel Parallel Programming Language*

Chapter 3 serves as an introduction to Chapel. We discuss the guiding principles and the design of the language and we detail the main language components that introduce and/or assist the writing of parallel programs and their internal functionality. We cover the topics of structured and unstructured parallelism, synchronisation, parallel loops and domain distribution. We also expand on the notion of locality which is integral to Chapel’s design. The chapter also covers –the most relevant to this work– parts of the underlying runtime system.

- *Chapter 4: Resilient Task Parallelism*

The chapter covers our main implementation of transparent resilience for task parallelism in Chapel applications. The main design decisions and assumptions are discussed early in the chapter. We then move to a detailed discussion of the functional parts of task execution on the runtime level and discuss the implementation details of resilience support. We focus on the extensions of the underlying communication layer and we evaluate the effect of the modifications on a set of constructed micro-benchmarks. The *evaluation section* targets the task-parallel components, as presented in Chapter 3 and is divided into two sections. A section on correctness; where the focus is on results verification with respect to the corresponding results of the non-resilient version and a section on performance and overhead results, using a range of experimental configurations.

- *Chapter 5: Resilient Data Parallelism*

The chapter covers the implementation of resilience support for Chapel’s data-parallel track, specifically Chapel’s *blocked distribution* module. We detail the design of the mechanisms that assist data redundancy. We then focus on the implementation of the resilient blocked distribution, covering the functional modifications required to support resilient program execution. For the purposes of our *evaluation* we use the *STREAM triad*; a synthetic linear algebra benchmark and the *N-body* all-pairs algorithm. We discuss the performance results with respect to the baseline’s execution runtimes. Towards the end we cover the issue of portability of the resilient implementation to other Chapel predefined distributions.

- *Chapter 6: Conclusion*

In the final chapter we summarize the main outcomes of our work in terms of system design, implementation challenges and performance results as they emerge from the empirical evaluation of resilience. We also discuss current limitations, as they arise from the design and the implementation. Finally, we propose future research directions that stem from this work, and topics that we consider relevant to the thesis.

Chapter 2

Background

In this chapter, we discuss concurrency and parallelism, expanding on the main characteristics of shared and distributed memory systems. We provide a systems' dependability overview, discussing system design and detailing properties and types of failures and widely-used failure metrics. We also discuss the main challenges faced towards the exa-architectures, focusing on failure detection mechanisms and techniques for failure avoidance and recovery. Towards the end of the chapter, we provide an overview of related work with focus on programming languages and runtime systems with resilient capabilities.

2.1 Distributed Systems Terminology

Continuous research and development in the fields of computational technology has brought the programming community into the era of multi-core machines and high speeds. Today, a conventional home machine is at least quad-core and can achieve around 15 gigaflops of computational power. Since sequential performance in terms of clock frequency has stalled and the number of cores per chip increases rapidly, a personal computer today is of equivalent computational power to the machines used in supercomputing a decade ago. According to Berkeley scientists (Shalf, Bashor, Patterson, Asanovic, Yelick, Keutzer, and Mattson, 2009), the goal from a hardware's perspective has been to double the number of cores per chip every 18 months. In order to achieve performance from the growing number of high-performance machines, the development of new sophisticated parallel programming languages be-

comes a necessity.

2.1.1 Concurrency and Parallelism

Concurrency is defined as the progressing of more than one processes or programs at the same time. It is a term that describes the general topic of parallelism in multiprocessing systems (Daintith, 2004).

According to Flynn's classification (Flynn, 1972), parallelism can be explained in terms of instruction and data streams in a system, leading to the below categorisation:

- SISD: single instruction, single data;
- SIMD: single instruction, multiple data;
- MISD: multiple instruction, single data;
- MIMD: multiple instruction, multiple data

SISD represents the computation on a conventional serial processor, while MISD does not occur in real systems. SIMD is a pattern of operation on matrices and vectors, which takes advantage of the inherent parallelism in these data structures. MIMD represents a wide range of architectures from large symmetrical multiprocessor systems, where processors share memory resources, to the small asymmetrical mini-computer/DMA channel combinations, where a master processor controls the data accesses. Shared and distributed memory systems form special subcategories of MIMD systems, and are discussed in detail in the following paragraphs.

2.1.2 Parallel Architectures

Distributed Memory Model

From a hardware's aspect, a distributed memory architecture consists of individual processors, with a dedicated memory and I/O mechanism. The processors rely on a network interface for local and wide-area communication. The address space is formulated by a number of *disjoint address spaces* and the same physical address can refer to different private memory locations, precluding explicit access from a remote processor. This is the type of memory architecture that is used in clusters. An overview of a distributed memory system is shown in Figure 2.1.

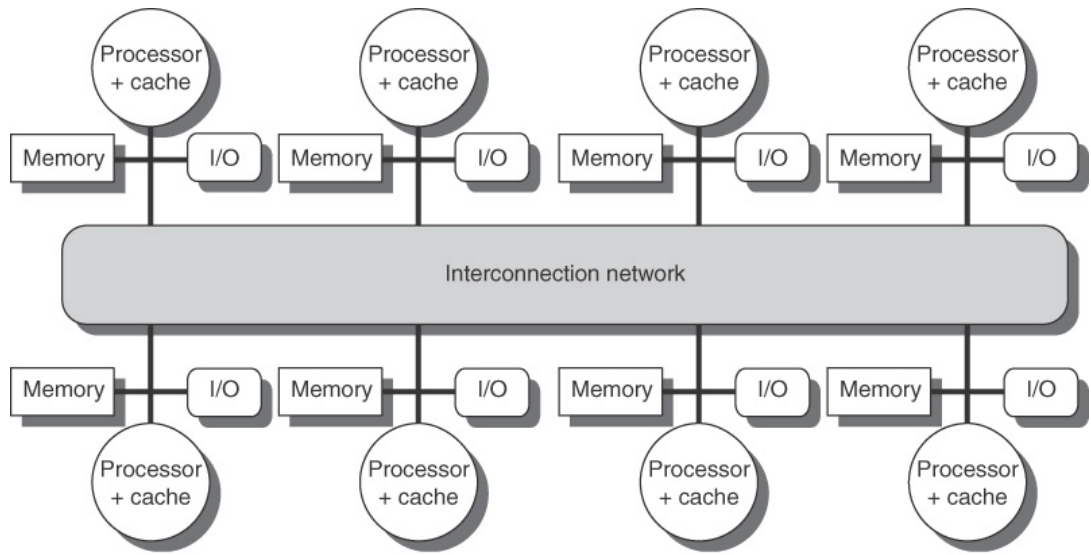


Figure 2.1: Distributed Memory Architectural Topology (Meter, 2016)

From a software's perspective processes in a distributed memory system perform computation on the local part of memory, while different sets of processes may in its simplest form reside on the same physical machine or on remote nodes in the networked environment. Data exchange is implemented by message passing, forming send/receive pairs of operations. Implementations of message passing require the programmer to embed library calls in the source code; they are typically language independent and tuned for a range of underlying architectures. The Message Passing Interface (MPI) (Graham, Dongarra, Geist, et al., 2015) is a standardized interface for performing data exchange on distributed memory architectures - currently at version 3.1.

Due to the high cost of message transfers, it is crucial for parallel performance to allow frequent memory accesses and to be able to hide the latency of message passing by performing other computation. Provided that the majority of memory accesses is local, a distributed memory system can achieve low latency. On the other hand, programming for such systems requires additional effort to coordinate the data exchange.

Shared Memory Model

From a hardware's perspective, a shared memory architecture consists of a number of independent processors, sharing the system's available memory, thus accessing

the same physical address space. Two commonly-used access patterns are: *Uniform Memory Access (UMA)*; where all processors have balanced access to the shared memory, and *Non-uniform Memory Access (NUMA)*; where a part of the shared memory is attached to each processor. An overview of the shared memory model can be found in Figure 2.2.

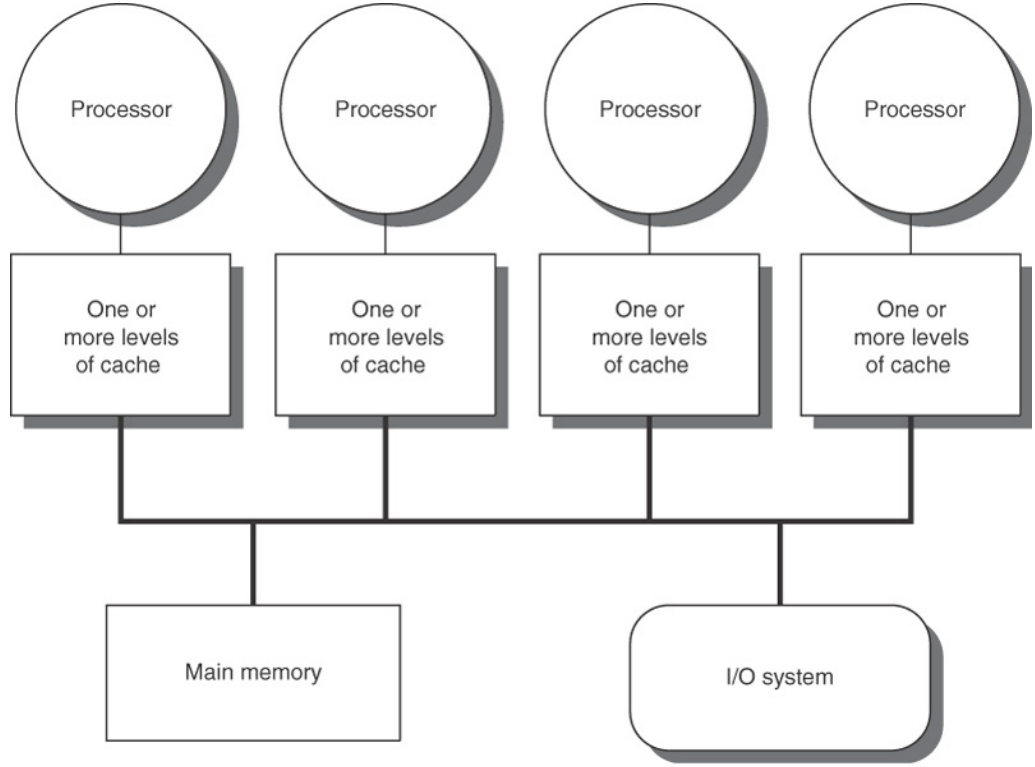


Figure 2.2: Shared Memory Architectural Topology (Ferretti, 2017)

As the number of cores scales, the memory architecture and the design of cache-coherent systems becomes challenging. Ongoing research on the elimination of this complexity proposes an approach based on non-cache-coherency (Cai and Shrivastava, 2016), which reduces data sharing and relies on message passing. An example machine with non-coherent caches is the Singlechip Cloud Computer (SCC) (Gries, Hoffmann, Konow, and Riepen, 2011) by Intel Labs, while the Compiler Microarchitecture Lab of Arizona State University (Jeyapaul, 2012) has proposed a virtual shared memory implementation where cache updates are handled on library-level. Nonetheless, most modern systems remain cache-coherent, mainly due to convenience in programming. A typical server-size machine today, such as the HPE Superdome Flex (Hewlett Packard Enterprise, 2018) consists of 4 up to 32 cores, forming one NUMA region per 4 cores. The NUMA regions are connected in an all-

to-all fashion, using Hewlett Packard’s ASIC technology –the latest generation of SGI’s NUMalink (Woodacre, Robb, Roe, and Feind, 2003), to provide equal latency across nodes.

From a software’s perspective, the memory is mapped to the address space of the processes that share the memory region. There is no inter-process data exchange, so the kernel is not involved. Due to the concurrent memory accesses, synchronization is required when storing or retrieving data to/from a shared memory location. Common synchronization techniques such as mutexes, condition variables, read and write locks, record locks, and semaphores are used widely in systems programming. OpenMP (Barney, 2017), which stands for *Open Multi-Processing* is an API used for multi-threaded parallelism for this type of memory. It consists of a limited set of programming directives and aims to provide a standard for programming shared memory architectures.

Inter-processor coordination and synchronization is managed via the global memory, thus increasing programmability. The main challenges faced in shared memory systems are performance and scalability. The interconnection network is a major bottleneck for performance, while scalability is often hindered by consistency issues between the global memory and the caches.

Hybrid memory architectures have also been proposed to tackle the challenges posed by the two prevailing models, as discussed earlier. For example, an alternative memory architecture may comprise of a small set of processors that function as an individual node within the system, using a single bus interconnect, thus combining shared and distributed memory characteristics on a single system.

2.1.3 High-Performance Systems towards the Exascale

High-performance systems face software challenges towards the exascale era. System utilization and load balancing are considered integral to benefit from the underlying hardware’s capability (Aggarwal and Aggarwal, 2010), while asynchrony and data migration costs continue to challenge programmers. One of the main hindering factors with regard to the scalability of high-performance computing (Dongarra, Graybill, Harrod, et al., 2008) is fault tolerance.

In DeBardeleben, Laros, Daly, Scott, Engelmann, and Harrod, 2009 the authors

identify the key research areas across fields for achieving resilience on High-End Computing (HEC). We expand on the topics of relevance to this thesis, while in Figure 2.1 we show the complete classification.

Theoretical Foundations	Enabling Infrastructure
Metrics and Measurements	Programming Models and Languages
Simulations and Emulation	RAS Systems
Formal Methods	System Software
Statistics and Optimisation	Middleware, Libraries and APIs
Efficiency Modelling and Uncertainty Quantification	Cooperation and Coordination Frameworks
	Tools
Fault Prediction and Detection	Monitoring and Control
Statistical Analysis	RAS Systems
Data and Information Collection	Performability
Anomaly Detection	System Software and Middleware
Visualisation	Tunable Fidelity
Machine Learning	Quality of Services
	End-to-end Data Integrity
	Silent Data Corruption

Table 2.1: Key Research Areas Towards Resilience (DeBardeleben, Laros, Daly, Scott, Engelmann, and Harrod, 2009). We briefly discuss the relevant areas (colour-coded) to systems design and this work.

Enabling Infrastructure *Programming Models & Languages*: Application level resilience introduces a new challenge into programming and it is unclear whether the average domain scientist can handle resilient code. Programming model and language production has been focusing on easing parallel programming complexity while providing performance. DARPA’s HPSC program is an example project targeting productivity coupled with performance. The predominant model for HPC applications today is the Message Passing Interface (MPI) which is historically not tolerant to faults. As it becomes evident by the number of languages and runtimes with resilience support, presented in Section 2.5, the focus of new programming models and languages is shifting today towards the provision of resilience mecha-

nisms.

System Software: HPC systems’ evolution has been driven by performance. For example, Linux OS has made progress in terms of performance over the past decade, but little effort has been invested in enabling reliability or robustness. Currently, the common case in the event of a failure remains the system-wide abort.

Middleware, Libraries & APIs: As scientific libraries (mathematical, data compression) and programming models overlap considerably, a large number of scientific computation depends on libraries – often highly tuned for performance. These libraries take no steps towards reliability, for example protection of data integrity. On the other hand, checkpointing libraries are specifically designed for reliability in the presence of hard failures, but do not ensure the data integrity of checkpoints.

Tools: A variety of tools continues to be developed on the analysis of performance of HPC applications, while very little effort has been put into resilience-related tools. System administrators today use tools that allow them to monitor a range of health metrics for a system, from voltage and temperature to fan speeds and resource managers. Application developers, on the other hand, lack the tools to monitor application related statistics, such as abrupt terminations or transient errors, which could prove useful in determining checkpoint intervals and resource utilisation.

Cooperation & Coordination Frameworks: Due to the lack of standardized interfaces, the components of a system are disassembled, thus inhibiting the cooperation on fault detection and recovery.

When designing for resilience in HPC systems, the above infrastructure is necessary but not sufficient to unilaterally tackle the issue of resilience. Subsequently, approaches to support resilience are often hindered by the lack of supportive mechanisms on other levels of the software/hardware stack. For example, in this work, we propose a mechanism for transparent software level fault tolerance, embedded in the runtime system of a programming language. In order to provide a complete evaluation, we required a mechanism to introduce failures in a controlled manner. Though, such a functionality is outside the scope of this work, we have dedicated time to customize a standalone testing mechanism, as this was integral for our testing. At the time of writing of this thesis and to the best of the author’s knowledge there is no available software to coordinate node failure simulations.

Fault Prediction & Detection Fault prediction and detection are fundamental to achieving resilience, though the difference between faults that impact applications, leading to errors, and of those faults that have no effect on the integrity of the computation, remains unclear. Fault detection also refers to monitoring the health indicators of a system to minimize error latency. Health metrics (Kothamasu, Huang, and VerDuin, 2006) include CPU, I/O and system paging to detect whether a system runs near maximum capacity and also performance alterations that may signify faulty or unoptimised components. Fault prediction in hardware systems is an extensively researched topic with approaches ranging across the analysis of sensor data (Turnbull and Alldrin, 2003) and the study of the temporal correlations of failures (Bouguerra, Gainaru, and Cappello, 2013) to the application of machine learning (Chigurupati, Thibaux, and Lassar, 2016).

The main challenge for *data collection* in the extreme scale is the volume of data and the size of the platform. Today’s techniques lack the ability to identify only relevant data and process them in a scalable and fast manner to reach useful conclusions, without impacting application performance. Similarly to statistical analysis and data collection, *anomaly detection* suffers from the large number of components and since components are expected to behave similarly when subject to the same circumstances, it becomes more difficult to establish causal relationships between failures and components.

Another hindering factor in error detection is the lack of *visualisation tools* and techniques that can allow the detection of irregularities. Additional capabilities, such as varying fidelity and granularity settings, could prove useful to real-time fault detection. Due to the vast amount of data collected during execution, *machine learning* is the most prominent research area for identifying and categorizing patterns of irregularity or degraded state of execution. Machine learning techniques, such as *pattern recognition* and *Knowledge Discovery in databases (KDD)* could also apply to platform monitoring, if provided with known fault indicators and use cases.

Monitoring & Control The focal points in control theory are observability and controllability, thus system monitoring research is concerned with determining the current state of the application or platform and finding ways to impose reliabil-

ity. *Performability* is used to describe the coupling of performance and reliability. Current models for large scale systems do not address reliability and performance equally and/or fail to incorporate power requirements. The discussion around reliability usually involves trade-off's of performance or power consumption. On application level, tunable results are foreign to scientists today, though a critical step towards scalable reliable systems.

Software support for resilience has commonly been vendor-specific and targeted to hardware monitoring, lacking metrics of *quality of service*. As a result, there is no common standard -for example an API- to query platform characteristics, runtime information and scheduling across different systems from different vendors. Such a standardization would assist in cross-platform testing of the proposed resilience mechanisms.

End-to-end Data Integrity Numerous techniques aimed to data integrity are available in literature, most prominently lock-step, bound checks and periodic flushing. The confidence in getting correct results is highly dependant on data integrity, especially in critical mission systems, thus such systems employ data integrity techniques on multiple levels; component, board, software. These techniques require significant time for design and validation and as bit-error rates grow, end-to-end solutions become more attractive compared to per-bit or per-structure approaches.

Silent Data Corruption (SDC) poses a threat to computational tasks and can have multiple causes, such as temperature/voltage, electrostatic discharge and fluctuations. Furthermore, platforms with multiple replicated components are more vulnerable to SDC's. We currently lack the scientific methods to determine the probability of a SDC, when for example writing/reading from disk. SDC errors require both efforts of characterisation and employment of suitable resilience techniques to mitigate their impact.

Theoretical Modelling One aspect of increasing concern is *error latency*; the interval between fault activation and error detection. For real-life systems, there is no theoretical modelling or *simulation & emulation* techniques that can allow an application programmer to make an educated choice among available hardware platforms to run an application with known resource requirements and execution

time.

2.2 Dependability of Distributed Systems

It is experimentally proven that identical systems which operate under closely matching conditions fail at different points in time (Schroeder and Gibson, 2010). Thus, the analysis of a system's dependability is based on a set of fundamental concepts drawn from probability theory. In the following paragraphs we introduce the main properties of dependable systems: *reliability*, *availability*, *safety*, *integrity* and *maintainability*.

2.2.1 Properties of Dependable Systems

Reliability is the probability that a system will perform its indented function within the specified design limits – typically runtime, memory or hardware capabilities.

Mathematically, if we assume that the lifetime of a component is represented by the randomly chosen value T , we can define the survival function of the component as follows:

$$R(t) = P(T > t), t \geq 0 \quad (2.1)$$

The above equation is the mathematical representation of reliability, $R(t)$ is the probability distribution of a failure and T represents a random time of a system failure. Assuming that the system can tolerate a single failure within the time interval at issue, and building on the previous equation of reliability we can define *unreliability* as a measure of failure, the probability that the system will fail by time T , earlier than t :

$$F(t) = P(T \leq t), t \leq 0 \text{ and thus } F(t) = 1 - R(T) \quad (2.2)$$

$F(t)$ is the *probability distribution function*, while $R(t)$ is the *complementary probability distribution function* (survival function). Following from the above, if we assume that T takes its values in $[0, +\infty)$, then the distribution function $F(t)$, and as a consequence the complementary probability distribution function $R(t)$, are

continuous for t and admit a derivative within their definition interval.

Considering the survival function $R(t)$, we can define the probability density of lifetime as the probability that a component or system fails within the interval $[t, t + \Delta t]$, as shown by the equation below:

$$i(t)dt = P(t < T \leq t + \Delta t) = P(T \leq t + \Delta t) - P(T \leq t) \quad (2.3)$$

and thus:

$$i = \frac{dF}{dt} = \frac{d(1 - R)}{dt} = \frac{d(1)}{dt} - \frac{d(R)}{dt} = -\frac{dR}{dt} \quad (2.4)$$

or

$$dR = \int i(t)dt \quad (2.5)$$

Availability (Daintith, 2004) is defined as the probability that the system will function according to specification at any point throughout a stated period of time t . It is a measure for allowing system repair, when failure occurs.

The *Mean Time to Failure* and *Mean Time Between Failures* are commonly used failure metrics to help in the definition of availability. **Mean Time To Failure (MTTF)** measures the average time to failure with a modeling assumption of infinite repair time. Given the reliability function of a system $R(t)$ (as in equations 2.1-2.8) we can define the MTTF as follows:

$$MTTF = \int_0^{\infty} t F(t) dt \quad (2.6)$$

For the case we are concerned, where T is a non negative random variable, and the mean T is described by $\bar{T} = E[T]$ (Kaufmann, Cruon, and Grouchko, 1977), the average of a system's lifetime distribution (**grottke2008ten**). We can assume there exist an $a > 0$ such that $\lim_{t \rightarrow \infty} [e^{at} R(t)] = 0$. As such $R(t)$ tends towards 0 exponentially. The above shows that for a given $\epsilon > 0$, there exists a failure rate such that

$$R(t) = 1 - (1 - \epsilon^{-t}) = \epsilon^{-t} \quad (2.7)$$

The survival function then becomes :

$$R(t) = \int_t^{\infty} f(s) ds \text{ for all } t \geq 0 \quad (2.8)$$

Then:

$$R(t) \leq R(0) = \int_0^\infty f(s) ds = \int_0^i f(s) ds + \int_i^\infty f(s) ds \quad (2.9)$$

Applying 2.5 and performing integration by parts, with the limitation of $tR(t) \rightarrow 0$ as $t \rightarrow \infty$, we reach the following definition:

$$MTTF = \int_0^\infty R(t) dt \quad (2.10)$$

Thus, MTTF is the definite integral evaluation of the reliability function.

Mean Time Between Failures (MTBF) is the measure of reliability of hardware components. For most components, typical uptime varies between thousands and tens of thousands of hours between failures. For example, a commodity hard disk drive has an average MTBF of 300,000 hours. MTBF also applies to systems and represents the expected value of time between failures, implying that a system has previously failed and has been repaired. MTBF is defined by the following relation:

$$MTBF = MTTF + MTTR \quad (2.11)$$

where MTTR is Mean Time To Repair, a measure of the time required for a repair in maintenance studies.

Building on the above metrics, the mathematical representation of availability is defined as follows :

$$Availability = \frac{System\ up\ time}{System\ up\ time + System\ down\ time} = \frac{MTTF}{MTTF + MTTR} \quad (2.12)$$

where MTTF is the Mean Time To Failure and MTTR is the Mean Time To Repair.

In Hacker, Romero, and Carothers, 2009, the authors analyse event logs of two IBM Blue Gene petascale systems and present a prediction model for node failures. They confirm previous findings that failure rates follow a Weibull (Murthy, Xie, and Jiang, 2004) rather than an exponential distribution, while they also investigate the possibility of ranking the system into sets of nodes based on reliability to efficiently guide job scheduling. They also observe variation in the reliability curves of the two

systems, though they are built with identical hardware components, which poses questions on whether MTBF provided by hardware manufacturers is an adequate measure of reliability.

Availability measures success and is used primarily for repairable systems. In the case of unrepairable systems availability is equivalent to reliability.

Safety (Avizienis, Laprie, Randell, and Landwehr, 2004) is defined as the absence of catastrophic consequences for the users and the environment. This property is especially relevant to mission critical systems, such as aircraft monitoring software. Safety assessments determine the impact of design and implementation on the overall systemic safety (Johnson, 1998).

Integrity (Avizienis, Laprie, Randell, and Landwehr, 2004) is the absence of improper system alterations. Integrity is a requirement of system security, and refers to the extent of the data corruption and the ability to recover from it; ensuring the correctness of system upgrades and the stability of the system's state.

Maintainability is the probability that a failed system will be restored to the specified conditions within a given period of time, when maintenance is performed according to prescribed procedures and resources.

In Laprie, 1995, we find a definition of dependability based on *Impairments*, the factors that make a system unreliable; *Means*, the methods to construct a dependable system and *Measures* that overlap with some of the properties explained above. This categorisation is represented in Figure 2.3, below.

According to the above taxonomy, we identify the factors that affect systems' dependability, referred to as *Impairments*. *Errors* are introduced by programmers or designers and may include incorrect numerical values, omissions or typographical errors. Errors lead to software *faults*, which can remain undetected until they become software *failures*. *Failures* occur when the system ceases to deliver the expected results according to the specification's input values. There are four categories based on the severity of the failure (catastrophic, critical, major and minor) which also vary depending on the system or the application. Failures are further discussed in the following section.

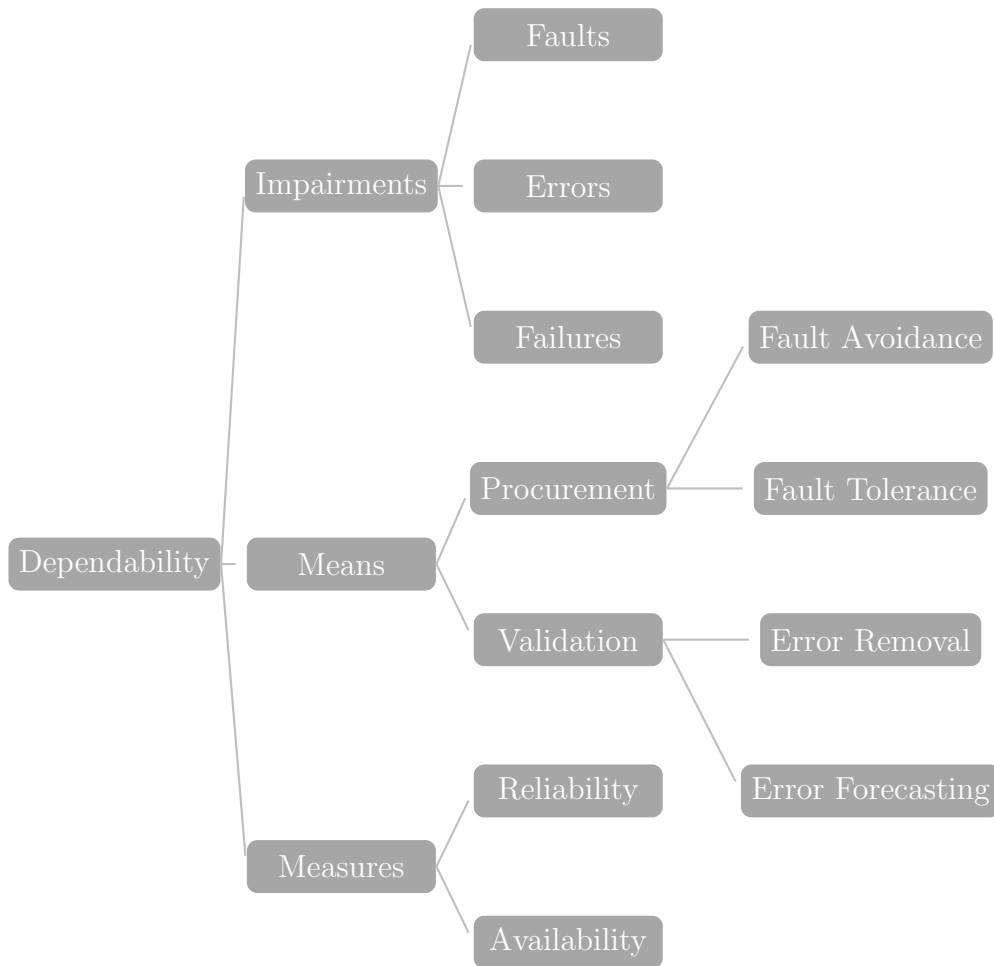


Figure 2.3: Laprie's taxonomy of dependability (Laprie, 1992)

2.2.2 System Failures on Distributed Systems

System failure is the condition in which a system no longer performs the intended function or is not able to do so at a level that equals or exceeds established minimums. Common causes of failure include software and hardware failures, network partitions, power outages and human errors.

Types of failures

Figure 2.4 demonstrates a categorisation of common faults. The categorisation covers the *when* (Creation Phase), *where* (Dimension) and *how* (Persistence) faults occur, while it also takes into account faults related to system security, caused with Malicious Intent.

In the literature, we find fault model taxonomies based on a set of differentiating failure factors. In Schneider, 1993a and Cristian, 1991, the authors propose a

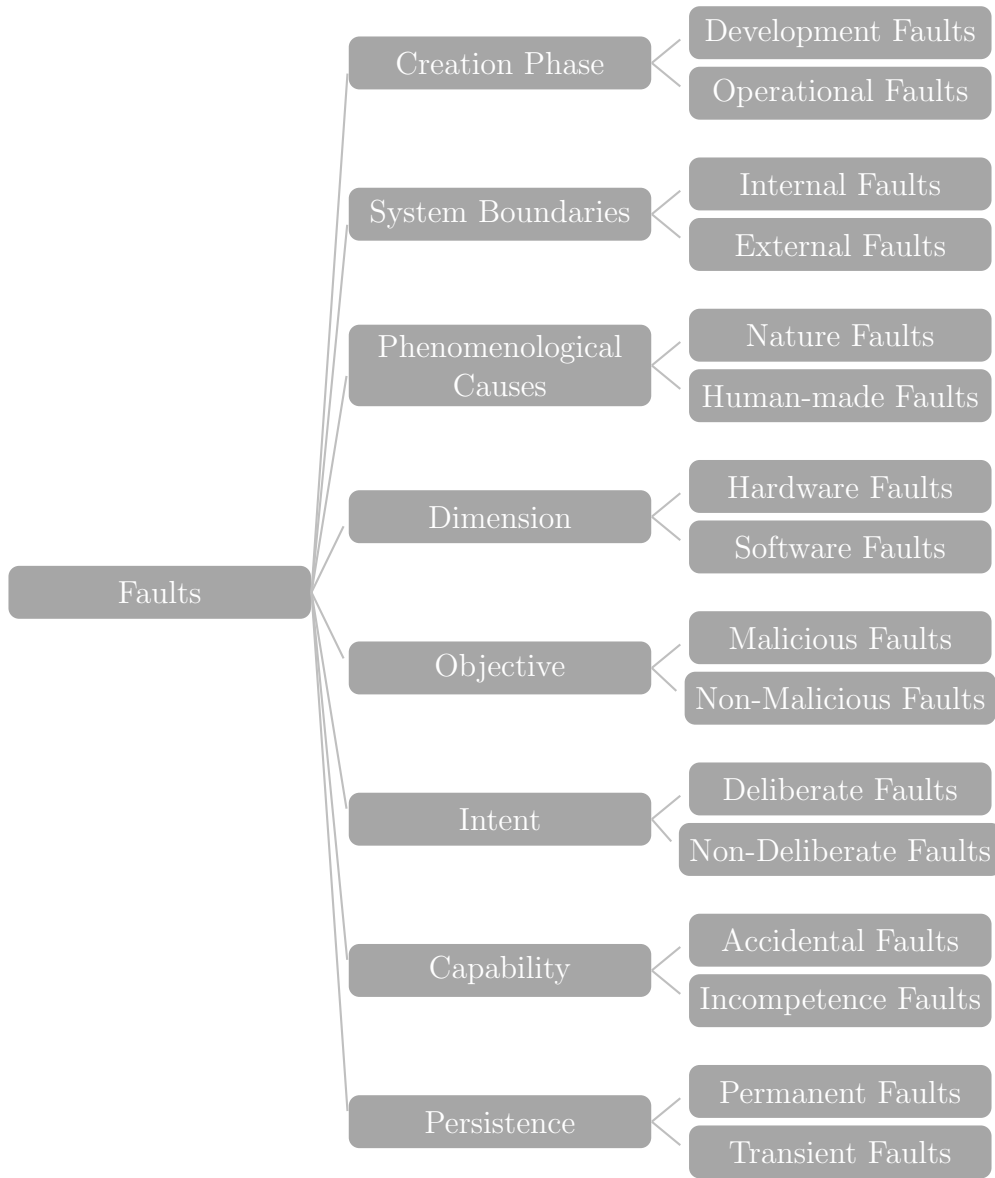


Figure 2.4: Categorisation of causes of faults (Laprie, 1992). In this work, we address internal hardware faults, irrespectively of their persistence, whether permanent or transient.

classification based on the result of the failure on the system. Among system faults we distinguish another taxonomy: crash faults; which cause system-wide aborts, fail-stop faults; where the failure of a subset of the system becomes evident to the rest of the system and Byzantine faults; where the system continues to operate but its behaviour is unpredictable.

The different classifications of fault-tolerance techniques, as described in Gärtner, 1999, take into account the properties the system preserves after failure; *safety* and *liveness*, the two classes of system properties required in order to prove program correctness. According to Lamport, 1977, safety is expressed by an invariant, a set

	live	not live
safe	masking	fail-safe
not safe	non-masking	none

Table 2.2: Classification of fault-tolerance techniques (Gärtner, 1999). Rows represent the preservation of the liveness property and columns represent the preservation of the safety property of a system.

of legal states. As long as the distributed program remains within the invariant, it maintains the safety property. On the other hand, liveness addresses the progress of a system. A common example of liveness is the timely termination of a program, given a correct input.

In Table 2.2, we summarize the fault tolerance techniques as they arise by the combination of the maintenance of safety and liveness in the system. Maintaining neither safety nor liveness (*none*) is equivalent to the absence of any fault tolerance measures. *Fail-safe* techniques aim to preserve the safety property and allow the system to terminate in a non-proper manner or in an unknown state. On the other hand, *non-masking* techniques are concerned with allowing the system to progress, although programs will eventually require the safety property to produce meaningful results. This is a common approach in internet services, as it allows the system to return to normal execution after the failure, for example a the case of a server crash. Finally, *fault-masking*, is the strictest and harder to achieve as it requires that despite failures the program will remain within the invariant and will continue execution, till it terminates properly.

In this taxonomy, our work falls under the fault-masking category, when node failures are introduced in the system. Our system aims to preserve liveness, by avoiding abrupt termination, and safety by introducing mechanisms to adopt orphaned tasks and perform task re-execution as required. On the latter, we also ensure that any further communication to failed nodes is handled transparently to the application level, by the runtime system.

2.2.3 Failure Rates

Technology scaling makes chip-level reliability difficult to achieve, due to *hard failures*, *single-event upsets* and *variability*. The shrinking processor sizes result in the

increase of hard failures and device degradation, leading to thermal stresses and electro-migration. On top of device shrinking, the increased power supply voltage, electric fields and temperature contribute to increased device failure rates. Also, single-event upsets (SEU) occur due to node capacity in integrated circuits. An increase of 8% is predicted in SEU rate per bit in each technology generation (Kogge, Borkar, Campbell, et al., 2008). With smaller transistors, the spatial variation of the electrical characteristics increases, leading to intermittent or permanent faults. The threshold voltage for transistors in a single group can vary by 30%. Figure 2.5 presents a summary of the above effects as a function of time, measured in 2008 and with a projection into the Exascale.

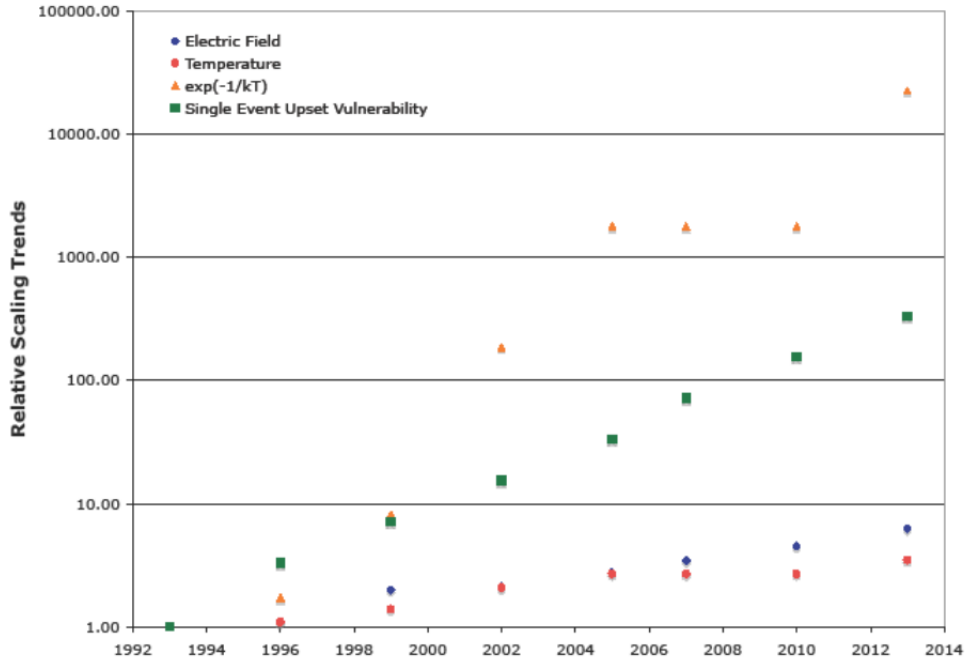


Figure 2.5: Resilience Rate Projection to the Exascale (2008) (Kogge, Borkar, Campbell, et al., 2008)

Table 2.3, summarizes the probability of different types of component faults, their impact probability and the proposed course of action to alleviate the effects of faults. While the failure rate of any particular component is relatively small, the overall resilience of a computing system depends strongly on the number of components that it is comprised of; this is particularly true for data-center class systems. Additionally, the impact of failure of any of the hardware components is

Fault	Probability	Impact	Action
Fans	High	Low	Node down
Power Supply	High	Low	Node down
CPU/SRAM	Very Low	Low	Node down
DRAM	Medium	Low	Reconfiguration
Solder Joints	High	Low	Node down
Sockets	High	Low	Node down
Disks	Mid to High	Low	Reconfiguration
NAND/PCM	Low	Low	Reconfiguration
Soft Errors	Low	High	Clever accounting

Table 2.3: Probability of Faults
(Source: Shekhar Borkar, Intel, Sep' 2014)

marked low with respect to the hardware, taking into account the costs of repair or replacement. In the case of an executing application though, any of these failures would lead to abrupt termination, therefore their impact is high.

2.3 Resilience Principles

Resilience is integrated to the system and works alongside the application's execution, as opposed to fault tolerance which is segregated from the system. Fault tolerance adopts a reactive and fatalistic approach to failures, contrasting resilience's proactive optimistic approach which is based on monitoring and often on data replication to circumvent failures.

2.3.1 CAP Theorem

Brewer (Gilbert and Lynch, 2002) conjectured that a distributed system cannot simultaneously provide all three of the following desirable properties:

- **Consistency** read operations are *aware* of all previously completed write operations;
- **Availability** read and write operations are always successful;
- **Partition tolerance** system properties are maintained even when network failures prevent communication among machines

In (Brewer, 2000) we also find examples of systems that forfeit one of the three properties and the features and mechanisms used. Examples of CA (Consistency-Availability) systems are single-site databases, cluster databases, LDAP and xFS

file systems. Commonly used practices include two-phase commits and cache validation protocols. Examples of systems that forfeit the availability property include distributed databases, distributed locking and majority protocols that employ pessimistic locking or withhold availability of minority partitions. Finally, Coda, web caching and DNS are example systems that forfeit consistency, usually by employing expiration rules and conflict resolution.

Different systems and technologies support combinations of the three properties at different levels. For example, it is widely accepted that we can achieve consistency and availability within a cluster, although it is hard to do so in practice. Also, OS's and networks provide better availability but are practically less consistent, contrasting databases that fulfil consistency rather than availability requirements.

The choice of two out of three properties, naturally results to three combinations of design approaches; CP, AP, and CA. It is widely acceptable that CA is not a coherent option as a system that is not partition tolerant, will be forced to give up either Consistency or Availability. The CAP theorem is thus restated as follows: *during a system partition, a distributed system must choose either Consistency or Availability.*

On component failure, our design prioritizes partition tolerance; by avoiding abrupt termination, and availability; by proactively migrating data and re-actively migrating executing processes on secondary locations. In our design, processes do not share state information; in order to avoid additional system-wide communication costs, instead migrated calculations are restarted on the remote location. Consistency in the sense of ordered task execution or the intermediate results, is not actively maintained. In fact, nodes will prioritize local over "adopted" task execution. This design decision does not violate program semantics since, irrespectively of their ordering, tasks will execute within the scope of the surrounding synchronisation block of the adopting task (or within the implicit synchronisation scope of the application itself).

2.3.2 Software Level Fault Tolerance

There is a number of different approaches when handling a system fault, with the simplest form being to allow the system to fail, though this approach is prohibitive

for industrial and embedded systems. Another common approach is to allow functioning parts of a system or an application to continue after shutting down the affected parts, also referred to as *graceful degradation* (Herlihy and Wing, 1991). This technique requires a component able to distinguish the functioning and faulty parts (Sampath, Sengupta, Lafortune, Sinnamohideen, and Teneketzi, 1995) or a detection mechanism embedded within the components (Yau and Cheung, 1975; Laprie, Arlat, Beoune, and Kanoun, 1990).

We also note the case of *fail-safe fault tolerance*, where a system reaches a safe state and is allowed to continue after the fault (Gärtner, 1999). An alternative approach is *component redundancy*, where multiple components perform the same task and the correct result is obtained using a majority voting algorithm. An instance of this technique is *N-version programming* discussed later in this section, while in (Schlichting and Schneider, 1983) we find a method of organising redundant components using fail-stop processors.

In Schneider, 1993b the author introduces a *self-stabilising* system, which can survive failure of the internal state and allows errors to persist until a correct state is restored, thus surviving transient faults, though the implementation and verification of such a system is hard in practice.

Fault Recovery

The main goal of a system's recovery strategy is to preserve a correct state with respect to the expected behaviour. Error occurrence can be determined as the time the system transitioned from a correct to an erroneous state. The two most common approaches in literature, are *backward error recovery* and *forward error recovery* and will be discussed in the following paragraphs.

Backward Error Recovery (BER) (Smith, 1988) This technique relies on the assumption that the system was in a correct state at some point in the past. At a time when the system is known to have a correct state, a copy of the state information is stored (backup). Once an error is detected, steps are taken to restore the program to the backed up correct state, thus moving the system backwards in time. The major disadvantage of this technique is that the error that caused the rollback

may not have disappeared, causing the system to fail again. To some degree this situation can be dealt with by moving the system forward after the rollback has occurred.

One of the most commonly used BER techniques is *checkpoint/restart*. One of its major selling points is the fact that it is suitable both for transient and permanent failures. The periodic saving of the state in stable storage is called a *checkpoint*. When a failure is detected, one or more processes are restarted using the information stored in the latest checkpoint. The simplest approach is synchronous checkpointing, with execution suspension until the checkpoint is saved. Depending on the type of stable storage used, this process can significantly reduce performance. Alternatively, asynchronous checkpointing strategies allow processes to continue execution, while a checkpoint is written. A number of optimisations has been proposed to minimize the time needed for checkpointing, notably *incremental checkpoints* that only capture the data that have been modified since the last checkpoint. Asynchronous and incremental checkpoint techniques depend on address translation hardware to detect modified data (Morin and Puaut, 1997).

Checkpointing techniques address primarily the communication layer of the runtime. *Consistent checkpointing* requires coordination of all processes on each checkpoint. In this technique, one checkpoint per process is enough to recover a correct state and the checkpoint imposes a barrier beyond which rollback is not necessary. A common optimisation on this technique is to checkpoint only processes that have established communication since the last checkpoint, with the drawback of having to maintain logs of the inter-process communication.

The alternative to consistent checkpoints are *independent checkpoints*, where each process establishes a local checkpoint without synchronisation. Because the set of the latest checkpoints of all processes does not ensure a consistent checkpoint, processes are required to maintain multiple checkpoints. In the event of a failure, inter-process communication is tracked in order to identify a consistent checkpoint and then each process is forced to rollback to one of their checkpoints. In the worst case, all processes must be restarted from the initial state, also referred to as the *domino effect*. Proposed optimisations to address the domino effect and minimize the data stored per process, require message logging. In this case, messages sent

after the checkpoint and before the failure are replayed using the logs.

Forward Error Recovery (FER) Forward Error Recovery (FER) takes steps to recover the state while moving the system forward along the time line. This technique requires prior knowledge of the failure characteristics of the system and system recovery needs to be built-in to the system. Commonly, it requires some form of redundancy scheme to maintain critical information that can be used to recover the state, similarly to the information persisted in checkpoints.

In other cases, the properties of the application need to be taken into consideration, for example a large weather simulation application may be fault-agnostic; the results of a failed computation can be substituted by older data (i.e. results of previous iterations) or failed computation can be excluded, without distorting the final results, a technique commonly referred to as *decimation*.

Forward Error Correction (FEC), a form of Forward Error Recovery, is commonly applied in digital communications to correct erroneous data on the receiving end, most commonly by applying error correcting codes (Puri, Ramchandran, Lee, and Bharghavan, 2001). The error correcting codes are encoded with the data, allowing the receiver to apply correction without the need for added communication. Multicast and broadcast networks, as a primary example of one way communication use forward error correction techniques.

Other Approaches to Software Level Fault Tolerance

As the recovery of the state is an important issue regarding fault tolerance, a number of techniques has been proposed to tackle the issue. *Compensation recovery* (Jin and Yang, 2009) is an approach where the faulty state is assumed to contain enough information to recalculate a correct state. *Fault-masking* (Laski, Szermer, and Luczycki, 1995, Yakovlev, 1993, Benítez-Pérez, Latif-Shabgahi, Haydn, Bennett, Fleming, and Bass, 1999), where in each step recovery is performed without previous fault detection, stems from compensation recovery.

Another recovery approach proposed during the 1970's, *N-version programming* (Chen and Avizienis, 1978), is inspired by hardware reliability techniques and it builds on the same ideas of FER. The base idea is that N versions of software exe-

cute and the correct answer is deducted using a voting algorithm to resolve conflicts. The decision algorithm is based on the increasing independence as N increases, but if failures do not occur independently and are more than $\frac{N-1}{2}$ then the system is guaranteed to fail. Practical aspects of constructing such systems are presented in Avizienis, Gunningberg, Kelly, Strigini, Traverse, Tso, and Voges, 1985 and Avizienis, Lyu, Schütz, Tso, and Voges, 1988 using the Design Diversity Experiment (DEDIX), a supervisor and testing system for multi-versioning, developed by UCLA.

Another approach, researched extensively during the 1970's, is *recovery blocks* (Randell, 1975; Horning, Lauer, Melliar-Smith, and Randell, 1974) which has served as basis for more recent approaches. Recovery blocks build on the idea of reliable segments, to ensure that changes to external variables of the block are performed reliably. *Resilient Distributed Datasets* (Zaharia, Chowdhury, Das, et al., 2012), discussed in detail in Section 2.5.2, draw from the same principle.

Finally, in the literature we also find approaches that aim at fault prevention, often referred to as *fault avoidance* (Smith, 1988) techniques. Verification protocols are used throughout the application to prove correctness, while detailed specifications of the behaviour of the program and continuous testing is employed to preserve correctness. Fault avoidance techniques are considered inefficient in proving the absence of bugs. As Dijkstra noted *program testing can be used to show the presence of bugs, but never to show their absence* (Dahl, Dijkstra, and Hoare, 1972). Program correctness proofs, like mathematical proofs, can only prove what is specified and the complexity of program adds to the complexity of the proof.

2.3.3 Hardware Level Fault Tolerance

Many of the techniques that have been proposed to tackle failures on software level derive from hardware level fault-tolerance techniques. In this section, we discuss a number of prevalent hardware *redundancy* techniques. The guiding principle is that when a fault occurs in real-time systems a set of redundant modules take over the functionality of the failed module(s).

One-for-one redundancy is a technique which assumes two hardware modules, namely the *active* and *standby* component. The standby component monitors the

activity and state of the active component and in the case of failure it takes over the execution of ongoing tasks. While this technique doubles the cost of hardware, it provides high levels of availability, as the probability of both components failing at the same time is decreased.

Another common approach is *N+X Redundancy* where N represents the number of required components to perform the task and X represents the redundant hardware, typically less than N. In case of failure of one of the N components, an X component adopts the local functionality of N. The process is coordinated by a higher-level module which monitors health and decides on which X module will take over N's work when N fails. N+X redundancy alleviates the high hardware cost, but reduces the system's availability in the case of multiple failures. We note also, that one-for-one is a special case of the N+X redundancy scheme.

Finally, *load sharing* is based on a high-level coordination module, responsible for distributing the load among active components and monitoring their health. In the event of a failure, the coordinator will attempt to redistribute the load over the available components. This technique leads to *graceful degradation* of performance on each hardware failure. The cost for redundancy in this case is minimal, but a hardware failure leads the system to perform in a non-optimal way until the failed components are replaced.

Our software-level resilience mechanism is inspired by the load sharing hardware technique. We are concerned with the progress of program execution when one or multiple components fail. As the Chapel runtime is agnostic to any health metrics of the underlying hardware and the communication layer as implemented by GAS-Net applies a strict policy of termination on failure, we introduce a runtime level coordination mechanism. We assume that a component is able to send a message to notify of its failing state. A hardware or software level monitoring module could replace this functionality without any further changes in the runtime.

All of the above mentioned redundancy techniques require synchronisation of the standby by the active components. In bibliography we find different techniques of standby synchronisation, such as *Bus Cycle Level*, *Memory Mirroring*, *Message Level* and *Checkpoint Level Synchronization* and *Reconciliation on Takeover*.

2.4 Resilient Store

To enable applications to recover from failure we require a form of *resilient store* to maintain critical information. The main feature of such a store is the ability to survive failures. The two prevailing techniques to achieve availability of data despite the occurrence of failures are *in-memory replication* and the use of *external file systems*.

In-memory data replication requires that the low-level data structure management is implemented within the runtime system, a requirement that introduces memory overheads. On the other hand, the technique is appealing as it is faster than disk storage (Zaharia, Chowdhury, Das, et al., 2012) and results in self-contained applications. As we discuss in earlier published work (Panagiotopoulou and Loidl, 2016) this approach ensures independence of third-party components and modules and does not require specific knowledge by the application programmer.

Also, as discussed in Dam, Vishnu, and Jong, 2011, the choice of an in-memory redundancy scheme is based on the implicit assumption that the number of the participating processors is chosen with respect to the amount of work to be performed, rather than the amount of memory required by the application.

The above assumption reveals a conflict between in-memory replication and the use of weak scaling. Indeed, the increased input sizes combined with the added memory stress of redundant data may lead applications to hang or fail due to memory limitations. On the other hand, weak scaling is a testing technique to process larger inputs when adding processing power; it is used to test or project performance by examining raw numbers. We argue that the goals of resilience and weak scaling differ, as resilience is intended for long running applications, rather than a sequence of experiments. Furthermore, we argue that weak scaling does not require the use of a resilience mechanism; the main testing requirement of maintaining a fixed per processor workload is violated when nodes (or processors) are lost due to failures.

As a high level design approach for the management of redundant data we define *buddy nodes*, an idea based on the work of Finkel and Tripathi, 1990. Buddy nodes act as *backup* locations to store the required data for task re-execution. Each of the participating nodes in the system is required to define one or multiple such buddy nodes and accordingly we refer the original nodes as *guests*. Each of the buddy

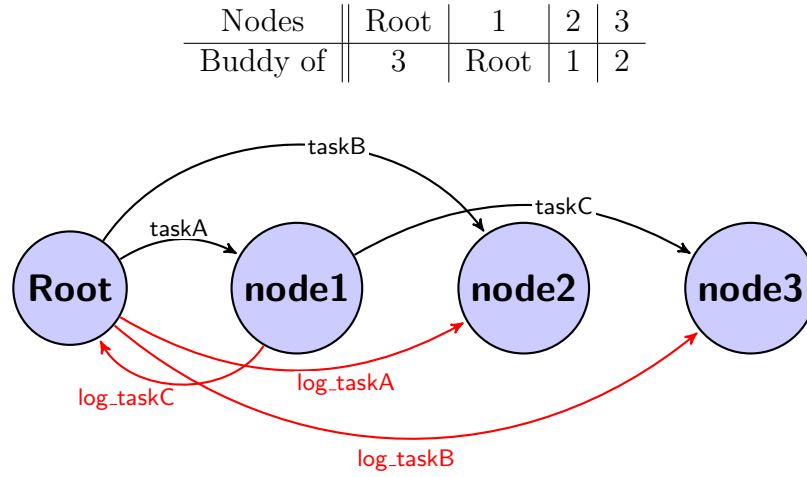


Figure 2.6: Sample communication flow of in-memory replication of tasks (and data) among *guest* and *buddy* nodes. `log_task{X}` represents the copied task descriptor of the remotely spawned task `task{X}` on the buddy node.

nodes is required to maintain data on all its guest nodes, and provide access or take up execution in the place of a dead guest node that has suffered a failure.

Figure 4.1 demonstrates an example of communication flow for in-memory replication among nodes, with a basic *next node* buddy configuration. In the figure, Node 2 stores copies of the tasks (and data) launched on Node 1, in this case *taskA*. The initiating node (*Root*) establishes communication to the buddy node (Node 2) and communicates *log_taskA*, the copy of *taskA*, intended to execute on Node 1.

For the purposes of this work we do not look into inter-dependencies among tasks. In fact, in the context of Chapel, task descriptors contain all the necessary information to re-execute the task from the stored copy. The above configuration (*next node*) is the one applied in this work, but alternative buddy configurations can be employed. Later in Chapter 5 (Section 5.2.2) we discuss the implementation details of this configuration while in Section 5.3.2 we discuss how a multi-locale configuration can take advantage of the simple round robin buddy allocation algorithm to build an efficient recovery strategy. We also detail the criteria that an alternative buddy allocation mechanism should fulfil in order to comply with the implementation in this work.

In the next paragraphs, we discuss two implementations of resilient stores, representative of the two main directions; external file systems such as Hadoop’s HDFS in Section 2.4.1 and in-memory replication with the use of an API, as implemented

by the ZooKeeper framework (Section 2.4.2).

2.4.1 External File Systems

Hadoop Distributed File System (HDFS)

Hadoop’s Distributed File System (HDFS) (Borthakur, 2008) is a file system designed for commodity clusters. HDFS is aimed for large datasets and provides high throughput data access. As the large number of components in a Hadoop system, expose the non-trivial probability of hardware failure, HDFS’s main design goal is to address *detection and recovery* from faults.

HDFS uses the master-slave architectural model, with a central *Namenode* and multiple *Datanodes*, as shown in Figure 2.7. The Namenode is responsible for the management of the filesystem’s namespace and the clients’ access to the files. Also, it handles open, close and rename operations for files and directories and maps the file blocks to the available Datanodes. Datanodes are responsible for local storage and for serving read and write requests, block and file creation, replication and deletion, as instructed by the Namenode.

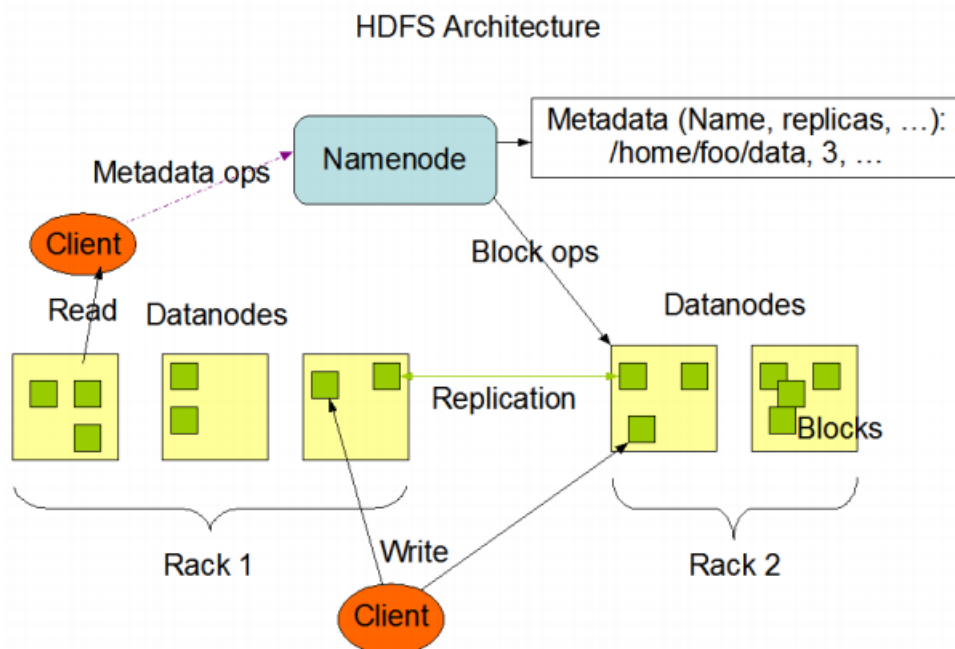


Figure 2.7: The Hadoop Distributed File System Architecture (Borthakur, 2008). The Namenode manages filesystem accesses and Datanodes serve read and write requests.

Fault tolerance in HDFS is achieved via *data partition* and *replication*. Each file is partitioned in equal sized blocks (with the exception of the last block) and it is stored on multiple Datanodes, while extra copies of the block are stored on remote Datanodes for redundancy. Both the blocksize and the replication factor are configurable. A second mechanism, aiming to fault detection, is the *periodical Heartbeat* and *Blockreport* sent from each Datanode to the Namenode. The heartbeat indicates that the node is functioning properly, while Blockreport contains a list of all blocks currently stored on a Datanode.

The Namenode also uses replication for local logs, while the blocks are validated for data corruption using checksums. The Namenode is a single point of failure for the system, while auto-restart and failover functionality to another machine is not currently supported. Chapel provides integration with HDFS. An example is provided in Appendix A.3.

2.4.2 In-Memory Replication Techniques

ZooKeeper

ZooKeeper (Hunt, Konar, Junqueira, and Reed, 2010) developed by Yahoo!, is a coordination service for distributed applications. The main motivation behind the ZooKeeper system is to provide applications with basic memory coordination setup. It uses a shared hierarchical namespace, organized as a regular file system, but with in-memory storage. The main components are data registers named *znodes*.

ZooKeeper is replicated on a number of hosts, while the server machines maintain an image of state, logs of the transactions and snapshots in persistent store. Clients establish a TCP connection to one of the servers and send requests and heartbeats. Each transaction leads to a time stamped update, used for ordering.

The system provides a set of guarantees, including *sequential consistency*, thus ensuring that updates from a client will be performed in the order sent, and transaction *atomicity*. Clients are provided with a *single system image* regardless of the specific server they are connected to. ZooKeeper guarantees that the applied updates will persist until the next update, thus maintaining the *reliability* property. Finally, the image of the system from the client's part, within a time frame, is referred to as *timeliness*.

2.5 Related Work

In this section we discuss a set of existing programming languages and runtime systems that provide resilience capabilities or have fault tolerance characteristics.

2.5.1 Languages with resilience capabilities

Resilient X10

Resilient X10 (Cunningham, Grove, Herta, et al., 2014) is a complete implementation of the X10 language designed for fault tolerance and based on three main design decisions. Firstly, the changes towards resilience only affect the runtime system and libraries. Secondly, a new type of exception, Dead Place Exception (DPE), is introduced and thrown when an X10 *place*'s failure is detected. Finally, a *Happens Before Invariance Principle* is introduced to ensure that in the occurrence of place failures, any modifications to the heap of non-failed places occur in the order specified by the original program. This ensures that tasks residing at failed places will either run to completion or not at all.

Base X10 Design X10's parallelism is based on the `finish` construct that defines synchronised blocks of tasks. A `finish` block waits until all spawned tasks within its context complete, while it also aggregates exceptions thrown by the tasks. A `finish` construct can be explicit in the application code or is created implicitly when a remote task is spawned by a placement `at` construct.

In regular X10, remote tasks are recorded with runtime level *X10RT* messages from the spawning place (*home*) to the remote place. The completion of a task is indicated by a corresponding X10RT message from the remote `place` where the task executes back to the *home place*. The mechanisms in place do not guarantee resilience, as in the event of a place failure, the synchronised block will wait endlessly on a termination notification message.

The runtime system maintains two internal stacks of `finish` states. The first one, called the *synchronisation stack*; it is used to query the closest synchronisation point. The second one is the *explicit stack*, used to locate the closest explicit `finish` which governs new asynchronous tasks (`asyncs`) in the code. All tasks have access

to these stacks and as they are spawned or terminated, they call methods on the associated finish object, which in turn handles wait operations. A finish object is complemented by an API with the following methods: `make`, `wait`, `fork`, `begin`, `join` and `pushEx`. The use of a `finish` construct on application level is demonstrated in Listing 2.1.

```

1 finish {
2   at (dst) async { // An asynchronous task is spawned
3   body();          // to the dst place
4   }
5 }
```

Listing 2.1: An example use of the *finish* construct in X10 (application level), that governs the synchronisation of a remote asynchronous (*async*) task spawned to place `dst`.

Resilient X10 Design Resilient X10 allows `places` (equivalent of locales) to fail asynchronously. Failures are exposed via exceptions and the runtime system is modified to repair the global control structure and ensure that the failure of a place does not alter the happens-before relationship between instances at non-failed places.

Failure detection is not part of the design of Resilient X10; it is assumed that failures are detected by the runtime system and realised in the form of the newly defined Dead Place exceptions. The developers do not report on node *resurrection*, so in the case of false positive failure detection, the relevant exception is thrown and the place is considered failed for the remainder of the execution. In Chapel’s resilient implementation, we provide a explicit sanity check mechanism to restrict new tasks from executing on locales that are detected to have failed.

On implementation level, an API call polls the status of arbitrary places and returns the number of failed X10 `places`. When a failed place is detected, the runtime undertakes the task of clearing the link to the place and continues the execution. A distributed termination detection mechanism is introduced as the number of tasks is not statically known at compile time. Listings 2.2 and 2.3 demonstrate snippets of the runtime API on the source and target place, respectively.

```

1 f=FinishObject.make(current_f);
2 //set activity's current_f to f
3 f.fork(here, dst);
4 x10rt_runAsync(dst, body, f);
5 //
6 //continue with local tasks
7 f.wait();
8 //restore current_f

```

Listing 2.2: The internal implementation of a *finish* at the source place (runtime system level).

```

1 //The remote task, f and src are received from src
2 if(f.begin(src, here)==true){
3   try{
4     body();
5   }catch (e:Exception){
6     f.pushExc(e);
7   }
8   f.join(here);
9 }

```

Listing 2.3: The internal implementation of a *Finish* at the destination place (runtime system level).

Listings 2.2 and 2.3 demonstrate the internal functionality of the finish object on the source and destination places of a remote task, respectively. In Listing 2.3, the conditional *f.begin(src, here) == true* is used to identify the case where the source place has failed after transmitting the message, but before the message has been received at the destination, thus restricting orphaned tasks from executing on the remote locale.

All exceptions are combined into a `MultipleException` *e* and thrown by the `wait` call. If the `finish` is implicit, then exceptions are propagated further towards the initiating place through the `at` construct.

The runtime maintains counters and assumes its own state as resilient. Each finish state object contains an internal *live counter set*, to record the executing tasks and their home places and stores exceptions accumulated via the `pushExc` method. In the case that a source place fails after the destination executes the `begin` statement, then the task may execute on the destination, though it is considered lost. Thus, the finish object has to log the messages in-transit (*transit counters*) and the conditional (`begin`) is only used to avoid execution of the task when the source place has failed.

Based on these semantics, X10 is able to support resilience, except for the case when the home place of the `finish` dies. In this case, tasks become *orphaned*. A parent finish cannot assume that tasks have terminated, as its local counters do not log orphaned tasks. Resilient X10 provides an adoption strategy for orphaned tasks, according to which, *the closest parent finish adopts orphaned tasks* by merging counters and discarding any exceptions. The implementation of the `finish` object is thus modified to maintain two counter sets to keep track of adopted and local tasks, while updates from adopted tasks are redirected to the location of the adopting finish.

X10 Resilient Store Implementations Resilient X10 introduces *three distinct implementations* to persist `finish` states. The implementations share the same abstract design and differ only with regard to the type of resilient store used; *Place-Zero-Based finish* implementation where all data is stored at place 0, *ZooKeeper-Based finish*, where all data is stored on *znodes* and *Distributed Resilient finish* implementation, where backup copies of the states are stored at a place different to the home place. We discuss the three available implementations in the following paragraphs.

In *Place-Zero Based Finish*, place 0 is assumed to never fail, so it implements the resilient data store for the application. In practice operations at other places, invoke asynchronous communication to Place 0, which maintains a database of all `finish` states with additional information on the *home* and the *parent finish* places. On discovery of a dead place, place 0 queries the `finish` objects on the dead place, locates the parent finish and facilitates adoption of orphaned tasks. Each finish that terminates without failure is removed from the database. This practice requires additional communication and makes place 0 a bottleneck, though it is reported to scale reasonably up to hundreds of places. Design-wise, the main idea of Place-Zero Based Finish is similar to our *failure-free root* assumption, as detailed in Section 4.2.1.

The *ZooKeeper Based Finish* approach uses the Zookeeper framewrok (Discussed in Section 2.4.2)) and aims at the elimination of the dependencies to Place 0. In the preliminary version, each *znode* stores a counter and each `finish` state creates

a *znode* with a unique identifier. Children *znodes* contain information on the local *home identifier*, the *parent finish identifier* and an *adoption flag*. Locks are used for mutual exclusion of places that update data on a node, while *znodes* are created dynamically to avoid initialisation overhead. Each API method contacts the ZooKeeper framework, while wait calls use a *watcher callback* mechanism to check that all counters have reached zero before termination.

In the optimised version, each *znode* is created with a unique sequence number, provided by the ZooKeeper server, thus reducing the number of required operations for each spawned task. This implementation, though optimised, is estimated to be 13 times slower than the *Place-Zero Based Finish* implementation, which is attributed to the high cost of ZooKeeper operations.

Finally, the implementation of the *Distributed Resilient Finish* is an effort to improve upon the bottleneck of Place 0, using an *X10-level resilient storage*. The core idea is to store each finish at its home place and maintain a backup copy at a different place. In case the home place is Place 0, then there is no need to use a remote backup and this case defaults to the Place-Zero-Based Finish approach.

Backups receive synchronisation updates from the master and in the case that the master dies, they receive updates from children tasks and facilitate the adoption of orphaned tasks. As such, every operation on the master requires the establishment of synchronous communication to the backup without the need to distinguish between adopted and non-adopted tasks. The backup information is stored in the form of a point-wise sum of the master's counter sets. After adoption, the backup is tagged with an adoption flag and a forward reference.

Each place maintains a backup table to associate the masters' global references to their backups. If a task fails to communicate with the master, it can query the backup tables of other places to find its backup. In the case that no master or backup is found with exhaustive search, a system-wide fatal error occurs. In the event of a failure, each finish queries for children tasks executing at the failed place and uses the children's pointers to find the backup copies and adopt the tasks. Thus, the tree of places remains connected, unless there is a failure of both master and backup places. In our design of resilient Chapel, we introduce backup tables to persist references to other locales (in the form of *buddy-guest* relationships among locales)

in the execution, while we also maintain information on the context of remote tasks in the system.

Resilient Design in X10 and Chapel X10’s resilience mechanism has served as a prototype for our work in Chapel. Some of the assumptions we make in this work, are shared with X10’s resilient version and derive from the PGAS programming model. As an example, in both implementations, tasks on remote locales/places execute till completion or not at all. Also, we follow a similar strategy to the *Place-Zero Based Finish* for our blocking fork execution, and both implementations support a form of in-memory distributed resilient store. In Chapters 4 and 5 we provide an in-depth discussion of our design.

On the other hand, our work differs from X10 in a number of key aspects. For example, X10’s runtime supports exceptions and failures are handled inside *try/catch* blocks that can be aggregated, while in Chapel we handle each failure as a single event. Also, X10 assumes that a remote place maintains up-to-date availability information on the parent place, while in our work we use status update requests, where needed. Furthermore, we do not use global references to other locales, apart from the root, since buddy/guest relationships change dynamically as failures occur; we instead re-calculate the identifiers of remote locales before communication. Finally, X10 assumes a distributed termination detection mechanism, while in Chapel we require the transmission of a short message to notify of a failure. Table 2.4 provides a comparative overview of the various design aspects and their handling in the Resilient X10 implementation and in our work.

Erlang

Erlang (Armstrong, 2007) is a functional programming language designed for distributed and fault-tolerant software. Erlang’s development stems from research within Ericsson towards a programming language tailored to telecommunication systems. It is designed towards fault-tolerance, in the sense that it provides tools for mitigating the impact of failures in the early version of the language. It provides process isolation to avoid propagation of corrupted data in the system and forces processes to crash. Erlang is tailored for distributed, rather than parallel

Design aspects	Resilient X10	Resilient Chapel
Resilient implementations	Three available implementations that adhere to the same design and differ only with respect to the resilient store used.	A single design and implementation with a number of configuration options is provided. A distributed in-memory resilience mechanism is used as resilient store.
Failure realisation	Failures occur in the form of exceptions, either single or aggregated.	Failures are realised as a single systemic event with the transmission of a communication layer signal.
Failure mitigation strategy	Communication links are cleared, the place is marked as dead, execution continues	The locale is marked as dead, the parent (serial execution) or buddy (distributed execution) adopts and re-executes the tasks of the failed locale, and execution continues.
Task adoption strategy	The place of the closest synchronisation point (<code>finish</code>) adopts the orphaned tasks. A number of orphaned tasks can be exposed via an aggregated exception.	The buddy locale adopts each failed task (or the parent locale in the case of the serial resilient implementation). Each task is migrated based on the destination locale it was initially scheduled to execute on and the buddy configuration.
Status calls	Resilient X10 implements calls to query the status of arbitrary places in the execution.	Resilient Chapel implements calls to query the status of buddies, guests and children locales
In-transit message logging	Recording of in-transit messages helps in the detection of incomplete transmissions when a source place dies.	In-transit messages are used between parent-children and buddy-guest locales to communicate task context.
Node resurrection	The number of failed places does not decrease during application execution.	No resurrection is supported for locales during execution, although idle locales have the ability to act as buddies and execute adopted tasks.
Low-level communication layer	The implementation supports only the TCP/IP sockets backend, since the PAMI and MPI backends do not provide one-to-one connections between communicating places.	Resilient Chapel is built on top of the GASNet communication backend, the most general multi-locale setup provided by Chapel. Other available implementations such as <i>ofi</i> , a libfabric-based communication layer, introduced in the more recent versions of the language and <i>ugni</i> , which is a Cray-specific implementation are not supported.

Table 2.4: A comparison of the Resilient X10 and Resilient Chapel design and implementations, including their main similarities and key differences.

HPC systems, but scales well for large numbers of agents.

Failure detection, as well as inter-process communication, in Erlang is implemented via linked processes. Each process in the system can create (and destroy) links to other processes using an API call and the destination's identifier.

When processes terminate, they emit an exit signal including an *exit reason* or they can call an *exit* function, without impacting their calling process. Processes can also receive exit signals from linked processes. Erlang uses two types of storage; *ETS: Erlang term storage* and *DETS: disk ETS*, for RAM and disk storage respectively. ETS data related to a process that has terminated is discarded, while DETS data is persistent and error checked.

In the case of abrupt termination, when a process exits with reason other than *normal*, all linked processes are signalled. The processes can be programmed to evaluate a boolean flag (`trap_exit`) and will either terminate (`trap_exit == false`), thus sending new signals and propagating the failure to the rest of the system or will take recovery actions (`trap_exit == true`). There is a cross-system analogy between trapping an exit and catching an exception.

Erlang introduces the core idea of *supervisor processes*. Supervisors monitor the termination of children processes through links, while they are also able to restart one or more children. In order to maintain consistency, the required data to start or restart a process are stored in the disk tables within a transactional scope. Each child process is complemented by a specification type (Nyström, 2009) shown below:

$$Id, M, F, A, Restart, Shutdown, Type, Modules \quad (2.13)$$

where

- **Id** is the identifier for the child process;
- **M**, **F**, **A** flags indicate how the child should be restarted; by a call to function **F** of module **M** with arguments **A**;
- **Restart** indicates how a termination must be handled and can take one of three values: *permanent* where the child is always restarted; *transient* where the child is only restarted if it fails and *temporary* which indicates that the child shall not be restarted;

- **Shutdown** indicates the timeframe (in milliseconds) within which a child process is allowed to perform shutdown before being abruptly terminated;
- **Type** indicates the role of the process in the system, worker or supervisor; and
- **Modules** specify the appropriate handlers to be used.

Each supervisor task implements a *Restart Strategy*, which can be *one-for-one*; only the terminated child process is restarted, *one-for-all*; all children processes are restarted on failure or *one-for-rest*; all children processes that started after the terminated process are restarted. In order to allow global system recovery, when local restarts cannot tackle the failure, each supervisor has a limited number of restarts to perform. The limits *maxR* and *maxT* indicate that if more than *maxR* restarts occur during a period *maxT* (in seconds) the supervisor task fails. Erlang applications are built using supervision trees to ensure efficient fault recovery. Children processes are connected to their parents via links. The leaves of the tree are worker tasks that perform the actual computation, while non-leaf processes perform monitoring tasks.

In the context of telecommunication systems or web servers, it is common for tasks to have inter-dependencies, thus multiple tasks might have to get restarted on a single failure. The processes are first terminated and subsequently replaced by newly created processes to avoid the use of obsolete data. A detailed discussion of Erlang’s fault tolerance mechanism complemented by a strategy for analysing process structures from source code is published in Nyström, 2009.

Though our work is inspired by Erlang’s fault-tolerance, especially with respect to the aspects of transparent failure recovery, the two systems differ in a number of ways. As Erlang is a functional language, tasks correspond to low-level threads that can fail and get restarted at any time throughout the execution; as such there can be no side-effects from partially executed tasks. Chapel’s runtime on the other hand, is layered, abstracting tasks from their executing threads, and to this end we introduce *task atomicity* as one of our assumptions detailed in Section 4.2.1. Erlang’s fault-tolerance is aimed at tackling task failures irrespectively of their location in the system, while in Chapel we focus on locale failures that closely match the loss of a node in a large-scale system and we require the communication API to handle

any lower-level errors. As such, the difference in the runtime implementations of the two languages does not allow direct comparisons.

FT-SR

FT-SR (Schlichting and Thomas, 1995) is the augmented version of the Synchronizing Resources (SR) (Olsson, Andrews, Coffin, and Townsend, 1992) distributed programming language with features for replication, recovery and failure notification. The programming model is based on the fail-stop model and focuses on processor failures with fail-silent semantics (Andrews and Olsson, 1993). The fault tolerant design is based on the replication of fail-stop modules that perform short atomic operations despite failure. For an N -fold configuration the application can tolerate $N-1$ failures, while failure notifications are generated when the N th fail-stop component fails. The main applications of the language regard two-phase commit server protocols and atomic operations in banking systems.

FT-SR is a domain-specific language with older design principles. There are two main similarities between our work and the FT-SR implementation. Firstly, both approaches work on the same level of the runtime system, attempting to tackle node failures and secondly, the system can use N backup nodes, similarly to the multiple buddy nodes used in this work. In contrast, Chapel is a general-purpose programming language and its use cases cover more than transactional schemes. In that sense, our work is aimed at handling failures on systems with complicated communication patterns, compared to server-client models.

2.5.2 Runtimes with resilience capabilities

Apache Hadoop YARN’s Resource Manager

In Hadoop version 0.23, MapReduce has been reconstructed to MapReduce 2.0 and renamed to YARN (Vavilapalli, Murthy, Douglas, et al., 2013). The driving design idea is the separation of resource management and scheduling, with the use of a global *Resource Manager* (RM) and one *Application Master* (AM) per application. The RM distributes resources among applications in the system, while the AM handles resource requests to the RM, coordinates job execution and monitors the

Node Managers (NM).

Within YARN clusters, the *Resource Manager* is a potential single point of failure, although applications may continue to execute uninterrupted. In this direction, possible restarts of the RM are transparent to the application level. The first step towards resilience is the preservation of application-queues (Phase I). The second step is to preserve the running state of the applications and resume work on restart (Phase II).

Phase I focuses on the reconstruction of the RM to enable it to store application-queues in a persistent state-store and re-read the states automatically on restart. This alleviates users from the burden of re-submitting jobs. Also, existing applications are re-triggered automatically when the RM restarts. If YARN fails to save the running states, the application is responsible to employ recovery mechanisms that allow it to continue execution, otherwise the application is restarted.

In Phase II, the RM is able to combine information from application queues with container-statuses, stored on the *Node Managers*, and allocation requests from *Application Masters*. On restart of the RM, applications only re-sync with the RM without any work loss.

FTC-Charm++

FTC-Charm++ (Zheng, Shi, and Kalé, 2004) is a fault-tolerant runtime designed for fast in-memory checkpointing and restart. It is based on Charm++ (Kalé and Krishnan, 1993), a parallel object-oriented language with high-level parallel constructs. On restart, after a failure, the program can continue to execute on the remaining processors and load balancing is employed to minimize the impact of the failure. The in-memory checkpointing version is suitable for applications with small memory footprint, while, for applications with large memory requirements, there is also an *in-disk checkpoint* variation.

One of the main design ideas in FTC-Charm++ is the disk-less double checkpointing. The system does not assume any kind of reliable storage and process state is stored in-memory in the form of objects. The storing two copies of an object on two different locations in a distributed manner, alleviates network bottlenecks on the server machine. This setup resembles our *buddy locales* system employed for

Chapel’s distributed remote task spawning, as demonstrated in Chapter 4.

At the same time, Charm++ employs load balancing mechanisms to minimize post-failure effects. Recovered applications continue execution on fewer processors. The system uses processor virtualisation (i.e. creates dummy processes in the place of failed processors) on restart and manages workload imbalances by migrating objects to less loaded processors. A design overview of the system can be found in Zheng, Shi, and Kalé, 2004, while in Zheng, Huang, and Kalé, 2006 we find a performance evaluation study.

Resilient Distributed Data Sets

Resilient Distributed Datasets (RDD’s) (Zaharia, Chowdhury, Das, et al., 2012) is a fault-tolerant abstraction for data sharing in cluster applications. The motivation behind RDD’s are iterative algorithms; where intermediate results are used across computations, and interactive data mining tools; where multiple ad-hoc queries are performed on the same data set. In both cases, in-memory data storage can increase performance by an order of magnitude.

The main difference of RDD’s compared to other solutions for in-memory storage on clusters (e.g. distributed shared memory (Nitzberg and Lo, 1991), key-value stores (Ousterhout, Agrawal, Erickson, et al., 2010), Piccolo (Power and Li, 2010)) are the coarse-grained transformations applied to big datasets. Existing abstractions build on fine-grained updates and are restricted to the use of data replication or the logging of updates across nodes. As a result, in data intensive algorithms, there is need for extensive copying, with the subsequent overhead. In RDD’s, fault tolerance is provided by logging the transformations instead of the datasets themselves, allowing for datasets to recover and avoiding expensive replication operations.

An RDD is a read-only partitioned record collection which is created via transformations on data in stable storage or on other RDD’s. Each RDD has information on its lineage, which is enough to recompute a partition from stable storage. In Zaharia, Chowdhury, Das, et al., 2012, we find a performance evaluation of RDD’s fault recovery mechanism using the k-means algorithm on *Spark*; a Scala-based system implemented by UC Berkeley for research and testing. RDD’s are exposed via a language API and represented via objects; the objects for lost tasks are reconstructed

on the failing point. The extra overhead introduced in the particular iteration(s) with failures, is amortized by the distributed re-run of lost task(s) across machines leading to a balanced per iteration runtime for the application.

Piccolo

Piccolo (Power and Li, 2010) is a data-centric programming model with fault-tolerance capabilities aimed at in-memory applications in data-centers. The runtime system employs key-value tables to store mutable state information across the participating machines, while updates on the same key are treated as atomic operations. It uses a master-slave model with the master periodically generating a consistent snapshot of the state. Piccolo, in contrast to our system, requires user-assistance, for checkpointing additional information on control functions and kernel point, in order to perform recovery. On machine failure, the master forces restart of all workers from the latest snapshot. As no information on the state of the master is maintained, failures of the master are handled as worker failures.

Recovery-Oriented Computing

Recovery-Oriented Computing (ROC) (Patterson, Brown, Broadwell, et al., 2002) is a joint Berkeley/Stanford research project, investigating novel techniques for building highly-dependable Internet services. The project is motivated by earlier efforts, such as IBM’s Autonomic Computing Project in 2001 and Microsoft’s shift to “trust-worthy” computing as stated by Bill Gates in 2002. ROC differentiates from traditional approaches for fault tolerance as it emphasizes on failure recovery rather than failure avoidance. It takes into account human errors, hardware failure and software ageing, while it also focuses on maintainability. More specifically, maintainability can be expressed as the ratio of $MTTR/MTTF$ and, in this sense, shrinking the recovery times has the same effect as stretching the time to failure.

The main study areas of ROC include failure pinpointing to speed up the recovery process, provision for graceful degradation, system-level undo functions, while the project also addresses failure categorisation and the design of recovery experiments. The ROC project has led to the design of *undoable* systems (Brown and Patterson, 2002) and system-wide undo functionality support (Brown, 2003) and to prototypes

of hardware-level support for recovery (Oppenheimer, Brown, Beck, et al., 2002).

In Table 2.5 we provide an overview of the languages and the frameworks presented in the Chapter; a summary of the main resilience directions in each system and finally a high-level comparison to the Chapel resilience framework presented in the thesis.

Language / System	Approach to resilience	Resilient Chapel
Resilient X10	<ul style="list-style-type: none"> • Failures are realised in the form of exceptions • Adoption performed by the place at the closest synchronisation point • Three distinct implementations of resilient store • Recording of in-transit communication • Node resurrection is not supported 	<ul style="list-style-type: none"> • Failures are realised as systemic events • Task adoption is performed on <i>buddy locales</i> • Single implementation of resilient store with in-memory data replication • Recording of in-transit communication • Node resurrection is not supported
Erlang	<ul style="list-style-type: none"> • Process-level fault mitigation • Failure isolation • Process linking • Supervisor process • Restart strategies 	<ul style="list-style-type: none"> • Node-level fault mitigation • Restart of failed tasks only • Buddy locales • Single restart strategy

FT-SR	<ul style="list-style-type: none"> • Node replication • Targets silent processor failures • Failure tolerance for two-phase commits • Configurable backup nodes • Failure notifications after failure threshold 	<ul style="list-style-type: none"> • General-purpose programming • Buddy locales as idle backups • Failure notification on detection • Fatal failures threshold
YARN	<ul style="list-style-type: none"> • Resilient store in-use • Read state from resilient store on restart • Automatic application restarts • Future work: application re-sync to YARN without restart 	<ul style="list-style-type: none"> • Restart avoidance • Continued execution • Graceful degradation
FTC- Charm++	<ul style="list-style-type: none"> • Application restart on remaining nodes • Load balancing • In-disk checkpoint variation • In-memory duplication of objects • Processes virtualisation • Dummy processes as placeholders for failed nodes 	<ul style="list-style-type: none"> • Restart avoidance • Continued execution • No load-balancing capability

RDD's	<ul style="list-style-type: none"> • Focuses on iterative algorithms with intermediate results • Logging coarse-grained transformations on the datasets • No data replication • Node lineage information persisted in resilient storage • Reconstruction of lost tasks on the failure point 	<ul style="list-style-type: none"> • In-memory data replication • No external storage • Lost task migration
Piccolo	<ul style="list-style-type: none"> • Persists mutable state information across machines • Support for atomic operations • Consistent snapshots propagated to slave nodes • User-assisted checkpointing • Restart of workers from checkpoint on failure 	<ul style="list-style-type: none"> • Persists status information • Transparency • Restart from local data on buddy
ROC	<ul style="list-style-type: none"> • Targets failure recovery • Targets hardware errors and software ageing • Detection with failure pinpointing • Support for system-level undo functions • Hardware level recovery support prototype 	<ul style="list-style-type: none"> • Failure recovery focus • Software-level support

Table 2.5: An overview of the resilience approaches employed by the languages and frameworks presented in the section.

2.6 Summary

In this chapter we have expanded on the issues of concurrency and parallelism, detailing shared and distributed memory models and the upcoming challenges for Exascale computing. We have provided an overview of systems’ dependability, discussing their properties and types of failures and we have detailed widely used failure metrics. We discussed a range of faults and failure taxonomies, as found in literature.

We reviewed the main challenges of resilient support and addressed the essential steps required to tackle these challenges across scientific fields. We have expanded on software-level fault tolerance techniques and emphasized *fault recovery* as the main mechanism to achieve system-level resilience. We covered two prominent *data redundancy* approaches – external file systems and in-memory replication, using two concrete representative systems, HDFS and ZooKeeper.

Finally, we have provided a critical review of related projects, focusing on programming languages and runtime systems with resilience capabilities. We discussed their enabling mechanisms and compared their functional characteristics to our design goals of *transparency* and *automatic task recovery*.

In the next chapter, we will discuss Chapel’s design principles, focusing on the language constructs that introduce *locality* and *parallelism*. Drawing from this chapter, we will address the main points that require modifications in order to support resilience following the *forward error recovery* approach with in-memory replication, to allow for *graceful degradation* of Chapel applications on failure.

Chapter 3

The Chapel Parallel Programming Language

This chapter serves as an introduction to Chapel, an instance of the Partitioned Global Address Space class of languages, designed by Cray for general parallelism on HPC hardware. We provide a brief discussion of Chapel’s origins and the main programming constructs offered in the base language. We then expand on Chapel’s key concepts to address HPC: *parallelism* and *locality*. We discuss the language constructs that support parallelism and locality and detail their internal functionality, focusing primarily on the communication and tasking layers of the runtime system. Towards the end of the chapter, we discuss Chapel’s latest experimentation towards the support of a data distribution for in-memory replication, which gives insight insight as to Chapel’s future directions regarding resilience.

3.1 A Brief History of Chapel

Cray launched the development of Chapel as part of the second phase of the DARPA High Productivity Computing Systems (HPCS) program in 2003, alongside four other teams, each led by a hardware or software vendor. The HPCS program’s main target was *improved productivity* for HPC programmers focusing on *performance*, *portability*, *programmability*, and *robustness*, while it encouraged contributions across the system stack; including proposals for new hardware architectures and new software, and empirical studies on programmers’ productivity. The teams

investigated opportunities for improvement in memory, processor and network architectures, while also engaged in the design of novel programming languages.

There were three short-listed teams on programming language design, with Cray pursuing *Chapel*, IBM working on *X10* and Sun developing *Fortress*. Chapel’s name was inspired by the Cascade Range, a mountain range at the south east of Seattle. The name is an approximate acronym for *Cascade High Productivity Language* (Bernheim, 2007).

3.1.1 Partitioned Global Address Space Programming Model

Chapel is a member of the wider class of parallel programming languages; the Partitioned Global Address Space (PGAS) languages. The PGAS programming model emerged in the early 2000s. Led by DARPA’s HPCS program (Phase II) and motivated by the independent development of Unified Parallel C (UPC) at Berkeley University, around the same period. UPC is another PGAS language; it emerged as an attempt to combine and refine the successful parallel characteristics and knowledge gained from previous C99 (ISO/IEC JTC 1/SC 22, 1999) extensions, such as Split-C (Culler, Dusseau, Goldstein, Krishnamurthy, Lumetta, Eicken, and Yelick, 1993) and Parallel C Preprocessor (Brooks III, Gorda, and Warren, 1992). Two other representative languages of the PGAS model are Co-Array Fortran (CAF) (Numrich and Reid, 1998; Fanfarillo, Burnus, Cardellini, Filippone, Nagle, and Rouson, 2014), which is an extension to Fortran 95, and Titanium (Yelick, Semenzato, Pike, et al., 1998), also developed at Berkeley and closely matching X10’s design.

Based on their core implementation, Chapel, X10, and Fortress are designed and developed from first principles, while Co-Array Fortran, UPC and Titanium are language extensions. Another distinction can be made based on their development time; Co-Array Fortran, UPC and Titanium belong to the first generation while Chapel, X10, and Fortress are a second generation PGAS. Finally, Co-Array Fortran and UPC are presumed as mainstream PGAS languages, due to the emerging interest of the parallel computing community in them. The development of Fortress officially stopped in July 2012, while the X10 development team have announced their plans to support direct compatibility to Java, abandoning on the C++ backend.

PGAS design principles

The PGAS model exposes both data and task/thread locality attempting to improve on application performance and programmer's productivity, by providing control over the lower-level characteristics via high-level abstractions. The predominant design requirements for modern programming languages, also reflected in DARPA's HPCS program, can be summarized to the following points (Dongarra, Graybill, Harrod, et al., 2008):

- Performance: Targeting the improvement in computational power by 10 to 40 times over current performance rates;
- Programmability: Targeting the decrease of development and maintenance time to 1/10;
- Portability: The results of the HPCS program were required to be applicable across software and hardware architectures;
- Robustness: Reliability and fault tolerance against hardware and software defects;
- Heterogeneity: Ability to address a range of emerging architectures, such as co-processors and GPGPUs and their combinations, with comparable performance results;
- Deterministic parallelism: The property of receiving the same results for the same input in every execution. This is a weaker requirement, though capable of providing a sound formal basis for the produced applications.

The performance requirement refers to the need for powerful machines and predates the development of multi-cores, while the portability goal implies performance portability irrespective of the underlying software and hardware technologies. In the next paragraphs, we discuss how Chapel introduces a number of abstractions and high-level constructs to address the programmability aspect. Throughout the development of Chapel, a number of releases of the language have focused on performance enhancements but the robustness aspect has not been a focus so far. This latter aspect has been the motivation for this work.

In the remainder of this chapter we will introduce language constructs of interest and we will identify a number of design directives illustrating the designers' motivation to address productivity and performance requirements.

3.2 Chapel Base Language

Chapel is a statically typed and type-safe imperative language. It supports first-class functions, reflection and method forwarding. Chapel statements, including procedures (functions) and iterators may have side-effects.

Control Structure

Chapel expresses data and control flow using the Global View Model, where application code may refer to any lexically visible variable, irrespectively of its location in the memory, local or remote. The compiler and the runtime system are responsible for implementing the load and store operations over the network. In contrast to commonly used parallel languages, the Global View Model raises the level of abstraction.

Parallelism is more general than the SPMD model as it imposes fewer restrictions on how parallel threads operate and it is introduced via multi-threading, while the low-level thread management is implicitly handled by the compiler, and aided by the runtime system. This enables Chapel to be architecture-neutral and achieve *performance portability*; enabling application execution across different platforms and maintaining a comparable level of performance (Pennycook, Sewall, and Lee, 2016). The adoptability goal of the language has been the motivation behind designing Chapel on top of architecture neutral components (Balaji, 2015); most prominently the generated C99 executable code, the C++ compiler, and the use of POSIX threads for parallelism. Chapel runs on both commodity hardware, including laptops and small clusters from different vendors, and on custom parallel systems, such as Cray machines.

Task parallelism is supported by constructs that introduce concurrency; **begin**, **cobegin**, and **coforall**, synchronise tasks (**sync/single** variables), introduce or suppress parallelism (block form **sync** and **serial** statements) and **atomics** for atomic operations. Atomic operations are guided by a transactional memory scheme, implemented in the compiler and the libraries.

Chapel **iterators** are another special concept that yield values consecutively or in parallel. Iterators are used in the underlying implementation of parallel loops, but they are also a concept that can be used directly on the application level. The motivation for iterators is to separate the data structure traversal from the com-

putation and avoid the complexity of nested loops. Parallel iterators are used in explicit `forall` loops and they are predefined for `ranges`, while custom implementations are provided for `domains` and `arrays`. In addition, Chapel allows user-level `zippered` iterators, that allow the traversal of multiple ranges and domains within a single loop. An example of a zippered iterator is demonstrated in Listing 3.1.

```
1 for (i, j) in zip(1..3, 4..6) do
2   write(i, " ", j, " ");
```

Listing 3.1: Example of a `zippered` iteration.

Sequential Model

Chapel supports a range of widely-used primitive types; `bool`, `int`, `uint`, `real`, `imag`, `complex` and `string`, while it also allows type casting and built-in constraints. The language supports *enumerated* types, *unions* and *tuples*. Similarly to the majority of modern parallel programming languages conditionals, serial loops and `break` and `continue` statements are also available.

Chapel incorporates the concept of objects from object-oriented programming models to address the programmers' needs for modularity, but without restricting the use of other programming styles, such as traditional imperative programming. Apart from classes, it supports records, which similarly to C#, support value semantics and are allocated in local memory. Procedures (functions) in Chapel support the use of formal argument intents; such as `in`, `out`, `inout`, `const` and `ref` and support type inference.

Chapel also employs parametric polymorphism to reduce code redundancy and promote code reuse, while interoperability to C is also supported via the `extern` statement. C routines can be inlined in Chapel programs, while also Chapel procedures can be used from external code via the `export` linkage specifier. Finally, the language provides a rich set of libraries in the form of base modules, such as `Math`, `Base` and `Type`, and auxiliary modules such as `BitOpts`, `Search`, `Sort` and `Time`.

3.3 Chapel’s Approach to Parallelism

Chapel supports diverse styles of parallelism including task and data parallelism, cooperative task parallelism and synchronisation based concurrent programming all of which can be arbitrarily composed.

3.4 Task Parallelism in Chapel

Chapel uses **tasks** that are able to execute in parallel and express computation. The target architecture is abstracted via **locales**: units with storage and execution capabilities, similar to a core in a multi-core processor, as discussed in our publication (Panagiotopoulou and Loidl, 2015). A multi-locale program begins execution on the root locale and spans out to other locales, as remote tasks are spawned. Chapel uses the term **LocaleTree** to describe the parent-child relationships and the dependencies that are created among the participating locales, as a direct result of the program’s structure. In the following paragraphs, we provide a brief overview on aspects of relevance to the support of resilience.

3.4.1 Unstructured Task Parallelism

The **begin** construct introduces unstructured parallelism by **forking** a new computation to execute on a new thread, while the rest of the program continues. The result of the task is returned at a later point in the execution and its completion is tracked either by an enclosing explicit synchronisation (**sync**) block or the implicit synchronisation of the main function (**join** operation). Listing 3.2 demonstrates an example use of **begin**. We note that the first call of **sumFunct** will execute on a newly created thread and thus the execution order of the two tasks is not guaranteed.

```

1 // sumFunct: prints the sum of two integer input numbers
2 begin sumFunct(4, 5);
3 sumFunct(2, 1);

```

Listing 3.2: Example use of the **begin** construct.

3.4.2 Structured Task Parallelism

`Cobegin` employs block-structured task creation; a task is created for each statement in the block, while the `coforall` loop creates exactly one task per iteration and is the loop-form equivalent of the `cobegin` block. Both `cobegin` blocks and `coforall` loops employ an implicit synchronisation barrier to control the asynchronous tasks that are launched.

```

1 const pivotVal = find Pivot();
2 const pivotLoc = partition(pivotVal);
3
4 serial thresh <= 0 do cobegin {
5   // tasks in the block execute asynchronously
6   pqsort(arr, thresh-1, low, pivotLoc-1);
7   pqsort(arr, thresh-1, pivotLoc+1, high);
8 } // implicit synchronisation point

```

Listing 3.3: Snippet of the `quicksort` algorithm implementation in Chapel, with in-place array modification, using an explicitly synchronised `cobegin` block (Cray Inc, 2015a)

The control flow returns when all tasks have reached the synchronisation point and the program resumes with the execution of the next statement. In Listing 3.3 we demonstrate a part of the main calculation of the *quicksort* algorithm, as implemented in Chapel using a `cobegin` block. The `serial` construct is used to suppress parallelism for performance tuning purposes.

3.4.3 Remote Task Spawning

Explicit remote task spawning is achieved in Chapel using the `on` construct. In Section 3.6.2, we discuss the construct in detail. Figure 3.1 demonstrates the control flow of the `begin`, `cobegin` and `coforall` constructs when combined with the `on` construct. The main difference is the block-wait operation of the master task on the guiding locale. When `begin` is used, the parent locale continues with the execution of other tasks in a non-blocking manner, while when `cobegin` or `coforall` is used, the parent locale block-waits on the completion of every task in the block/loop. The tasks that are created within a `cobegin` or `coforall` execute in a non-blocking fashion with respect to each other, while the guiding thread of the master task contributes to the execution of the tasks behind the scenes. Finally, Chapel applies

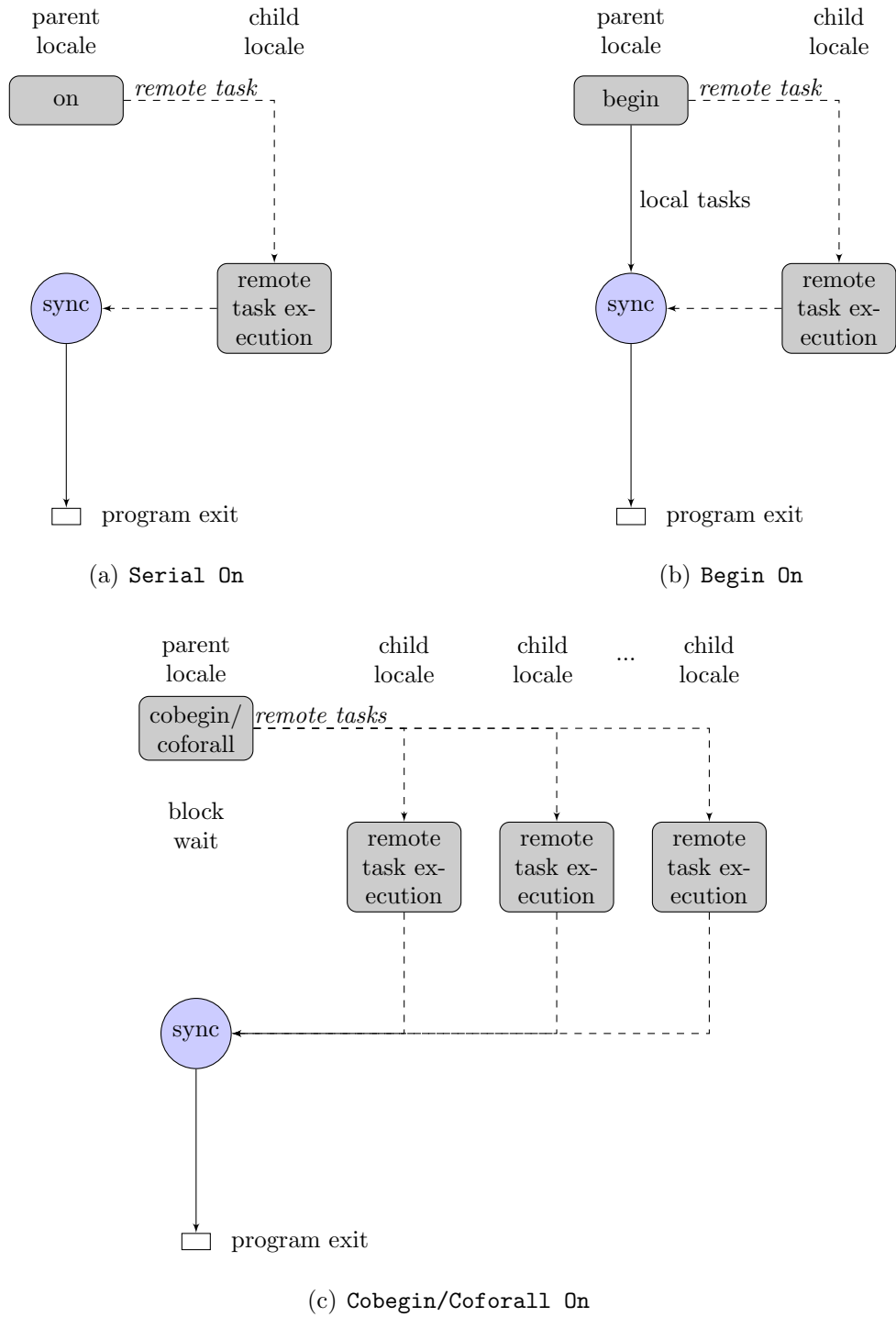


Figure 3.1: Program control flow using 3.1(a) the `on` construct in a serial execution, followed by its combination with task-parallel constructs (3.1(b) `begin` and 3.1(c) `cobegin/coforall`) to produce distributed parallel execution flows in Chapel. In order to achieve remote execution of a task in Chapel, the `on` construct is required for placement and a task parallel construct is potentially combined to produce forking.

a compiler optimisation to execute tasks serially when a `cobegin` block contains less than three tasks.

3.4.4 Synchronisation and Atomicity

Synchronisation in Chapel is supported via `sync` and `single` variables which are associated with a logical full/empty state. Chapel’s primitive types (excluding `complex`), enumerated types and classes can be translated to synchronised types using the `sync` and `single` type constructors.

```

1 var count$: sync int = n; // counter which serves as lock
2 var release$: single bool; // barrier release
3
4 forall t in 1..n do begin {
5   work(t);
6   var myc = count$; // read the count, set state to empty
7   if myc!=1 {
8     write(" ");
9     count$ = myc-1; // update the count, set state to full
10    release$;
11  } else {
12    release$ = true; // release all tasks
13    writeln("done");
14  }
15 }
```

Listing 3.4: Implementation of a split-phase barrier using `sync` and `single` variables (Cray Inc, 2015a)

A synchronisation variable can only be read when in a *full* state and accordingly it can be written when its state is *empty*, while when an initialisation expression is present in the declaration the state is considered full. The difference between the two types is that a `single` variable is immutable, in the sense that it can only be written once. These concepts are similar to Haskell’s (Peyton Jones, Gordon, and Finne, 1996) *MVar* and *Ivar* data types. The `$` notation is used by convention as suffix for `sync` and `single` variable names to provide the programmer with a visual hint about synchronisation in the code.

Listing 3.4 demonstrates a `forall` loop with locks. In each iteration the task reads the `count$` variable and attempts to read the `release$` variable. All tasks will block, until the last task writes the `release$` variable, thus setting it to a full

state and allowing all other tasks to unblock.

Chapel provides support for atomic operations for `bool`, `int`, `uint`, and `real` types with the use of `atomic` variables. Common operations on atomic values include `read`, `write`, `testAndSet` and `fetchAdd`. On the runtime level, Chapel programmers can choose among three implementations for atomic operation, via the `CHPL_ATOMICS` environment variable; `cstdlib`, `intrinsic`s and `locks`. Listing 3.5 demonstrates an example of an atomic variable x with the use of predefined methods for reading and writing.

```

1 config const n = 31;
2 const R = 1..n;
3
4 var x: atomic int;
5 x.write(n);
6
7 if x.read() != n then
8   halt("Error: x (", x.read(), ") != n (", n, ")");
```

Listing 3.5: Example use of `atomic` variables (Cray Inc, 2015a)

3.5 Data Parallelism

Data parallelism in Chapel is expressed with parallel `forall` loops and with a set of implicitly parallel operations on arrays, such as whole-array assignment and reductions. These high-level constructs promote an abstract parallel programming style and assist in the avoidance of common errors, such as deadlocks and race conditions.

3.5.1 The `forall` loop

Chapel’s `forall` loop is the parallel version of the `for` loop, where an arbitrary number of tasks is used to execute iterations in parallel. This is the main point differentiating the `forall` from the `coforall` loop, where each iteration is executed by a distinct Chapel task. The number of tasks in the `forall` is decided by the iterand based on the available hardware resources, while the user may explicitly set a maximum threshold on the creation of tasks.

The *iterations in a forall loop are required to be serialisable* and thus free from inter-task dependencies; serialisability is a guarantee that a set of operations that alter data are equivalent to a serial execution of the same operations. The idea closely matches the *isolation* property of the transactional paradigm in database development (Haerder and Reuter, 1983), alongside the principles of atomicity, consistency and durability. A serialisable execution will preserve correctness, given that each separate iteration in the forall loop preserves correctness. As the forall loop must be able to produce the correct output even in the worst case, when executed by a single task, its serialisability is checked by the compiler during the semantic analysis phase (type checking).

Chapel users are allowed to write custom iterators (Chamberlain, Choi, Deitz, and Navarro, 2011) to guide forall loops. To achieve this, users need to supply the iterators that create the tasks and distribute the iteration space among them. Iterators can be produced using solely base language and task-parallel constructs. The implementation details of iterators are discussed in Section 5.1.3.

3.5.2 Domains and Arrays

Domains are another novel Chapel construct representing ordered sets of Cartesian indices. Domains define iteration spaces, support aggregate operations; such as slicing, control the shape of arrays and drive loops. A domain is defined by its **rank**; indicating the number of its dimensions, **idxType**; which specifies the index type for each dimension, and a boolean **stridable** flag, to indicate whether any of the dimensions results from a strided range. Domains are first-class language constructs and can have both regular and irregular base domain types, such as dense, strided, and associative accordingly. Iteration spaces can be single or multi-dimensional and domain indices can be distributed across multiple locales.

As a high-level abstraction mechanism, **domains** support promotion of scalar operations, which are equivalent to explicit forall loops, with parallel mapping of functions across the domain's indices. They can be used as hash tables, dictionaries and unstructured graphs, while there are also sparse domains that represent subsets of parent domains. Chapel also supports a fixed set of higher-order custom reduction and scan operations (sum product, max/min, bitwise operations) on domains,

as part of its standard modules, similarly to MPI’s collectives. Reductions can flatten the dimension of values, while scans are used to calculate prefix operations in parallel.

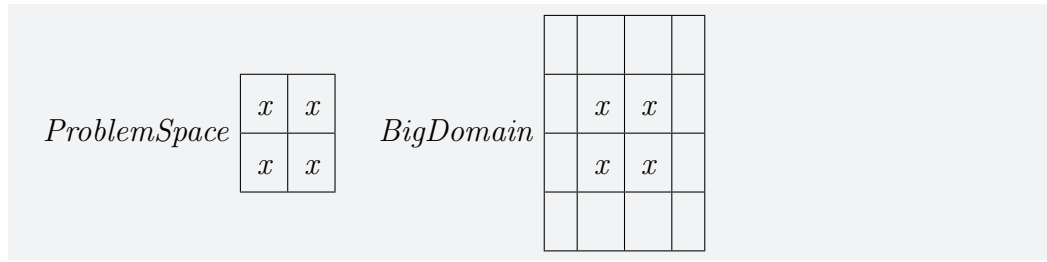
Listing 3.6, demonstrates a snippet of the Jacobi four-point stencil implementation in Chapel. The code is a standard example of the use of domains as iteration spaces and demonstrates the basic syntax for array declaration. The two arrays *X* and *XNew* are declared over *BigDomain* (line 5). A forall loop is used to iterate in parallel over the indices of *ProblemSpace* performing the main calculation on the array’s values. On line 12, a combination of higher-order functions is used to calculate the *delta* value between the two arrays.

```

1 // representation of the initial problem space and
2 // a superset domain
3 const ProblemSpace = {1..n, 1..n},
4           BigDomain = {0..n+1, 0..n+1};
5
6 // declaration of the initial and result arrays
7 // over BigDomain
8 var X, XNew: [BigDomain] real = 0.0;
9
10 // constants to represent the neighbouring nodes
11 // in every direction
12 const north = (-1,0), south = (1,0), east = (0,1), west =
    (0,-1);
13
14 do {
15     forall ij in ProblemSpace do
16         XNew(ij) = (X(ij+north) + X(ij+south)
17                   + X(ij+east) + X(ij+west)) / 4.0;
18     // reduction: compute the max of absolute differences
19     // of the next and current approximations to check
20     // if convergence has been reached
21     delta = max reduce abs(XNew[ProblemSpace] -
22                           X[ProblemSpace]);
23     X[ProblemSpace] = XNew[ProblemSpace];
24     iteration += 1;
25 } while (delta > epsilon);

```

Listing 3.6: Chapel’s implementation of the jacobi algorithm using forall loop and domain operations (Cray Inc, 2015a)



3.6 Locality

Chapel enables programmers to control locality on the lower levels, by specifying the placement of variables and the location of tasks, but also at a higher level with the use of distributed domains and arrays. Chapel’s support of distributed-memory data parallelism, stems from this high-level approach. Locality is distinct from parallelism; in Chapel’s design the two concepts are orthogonal and can be combined freely.

3.6.1 The locale type

Locales are portions of the targeted parallel architecture with processing and storage capabilities (Cray Inc, 2015b). *Locality* is the core Chapel design principle to express an abstraction over both data affinity and the placement of task execution. A locale in Chapel is a primitive type, which allows control over affinity, while it allows tasks to have uniform access to local and remote data, due to the global name space of the PGAS programming model. Locale types support equality operations and often represent a compute node, such as a multi-core node of a conventional parallel architecture. The tasks that execute on a locale have access to local and remote variables, though with different access costs.

Chapel programs execute on a number of locales, specified at execution time. Locales can be referenced from within the code using their `localeId` within the *LocalesArray* and via predefined methods that query locale properties. Examples of such methods return the total number of cores, a locale’s name, the call stack limit and the number of tasks and threads that execute locally at a specific point during the program’s execution.

3.6.2 The `on` construct

In Chapel the programmer can explicitly specify the resource on which a task will execute by using the `on` construct and a single operand specifying the target locale. A variable can also act as an operand, in which case the statement will execute on the locale where the variable is stored. Controlling locality via variables is a better programming practice since it disengages the program from the number of locales. Additionally, programmers can query the locale on which a variable is stored or a task executes, using the `.locale` method or the built-in variable `here`.

```

1 var x: int = 2;
2 on Locales[1 % numLocales] {
3   var y: int = 3;
4   // execution on the locale where x is stored,
5   // syntactic sugar for x.locale
6   on x do
7     task();
8 }
```

Listing 3.7: An example of Chapel’s task migration using the `on` construct (Cray Inc, 2015a). `numLocales` is the execution flag specifying the number of locales on which the application will execute; available also on application level.

The `on` construct, demonstrated in Listing 3.7, is the main language mechanism for task migration. `On` constructs are used in Chapel to explicitly control task locality and guide the execution. The migrated task is a logical continuation of the initial task at a different place in the system. The body of an `on` block is perceived as a single blocking task by the controlling thread of the parent locale.

`On` clauses do not introduce parallelism in the program, hence emphasizing on Chapel’s distinction between parallelism and locality, however they can be combined with both task- and data-parallel constructs to generate different parallel styles.

```

1 forall loc in Locales do
2   // task migration to loc
3   on loc do
4     task1();
```

Listing 3.8: An combination of `forall` and `on` constructs (Cray Inc, 2015b)

Listing 3.8 demonstrates a distributed parallel program with a combination of `on` and `forall` constructs. On the runtime level, a non-blocking fork operation is

used to implement the parallel execution, which includes the task migration on the remote locales.

Fork operations are also used during initialisation of multi-locale programs to establish communication between the root locale (Locale 0) and the rest of the locales in the configuration; such tasks include broadcasting of global variables and locale initialisation on the communication layer.

3.6.3 Domain Maps, Layouts, and Distributions

Domains and **arrays** are governed by **domain maps** that specify their distribution. Each domain map specifies the rules that control how domains or arrays are distributed over a range of locales; how indices and elements are mapped over physically distributed memory and how operations are performed.

When no domain map is provided all indices of a domain are mapped to the current locale and stored in local memory, forming **layouts**. When multiple locales are targeted Chapel employs **distributions** to allow distributed mapping of indices and, consecutively, array elements.

Distributions and layouts are encountered in most PGAS languages as a tool to query domains and arrays that reside on the same locale or are partitioned across multiple locales. Chapel offers commonly-used predefined layouts and distributions as part of its standard module library. In Listing 3.9, we demonstrate example use cases of the **Block** and **Cyclic** distributions. Parallel **forall** loops are used to initialise the values of the distributed arrays using the unique identifiers of the corresponding locales. In Chapter 5 we focus on the internal implementation of the **Block** distribution.

Domain maps are a high-level construct in Chapel as they are implemented on top of other high-level constructs that provide data- and task-parallel features, locality features and base language constructs. The full power of these constructs stems from the ability to develop user defined distributions tailored to the target application (Chamberlain, Deitz, Iten, and Choi, 2010). The explicit placement of data and computation is the main feature that supports locality.

The block size is calculated in the Blocked distribution by dividing the indices of the domain with the number of the target locales. If a remainder exists, then a

larger block size is used, resulting in larger blocks being assigned to the first locales, per their ordering in the target locales array of Table 3.1.

idx	locIdx
$low \leq idx \leq high$	$\text{floor}((idx - low) * N / (high - low + 1))$
$idx < low$	0
$idx > high$	$N - 1$

Table 3.1: The index set partitioning into blocks within the Blocked distribution. Each block is mapped on a locale of the `targetLocales` array. The `boundingBox` is the domain used as guide for the partitions and it is commonly equal to the problem domain.

```

1 // distribution modules
2 use BlockDist, CyclicDist;
3
4 // two-dimensional space
5 const Space = {1..8, 1..8};
6
7 // block distribution over a
8 // two-dimensional domain
9 const B: domain(2) dmapped
10     Block(boundingBox=Space) = Space;
11
12 // cyclic distribution over a
13 // two-dimensional domain
14 const C: domain(2) dmapped
15     Cyclic(startIdx=Space.low) = Space;
16
17 // declaration of blockArray over the
18 // block-distributed domain B
19 var blockArray: [B] int;
20
21 // declaration of cyclicArray over the
22 // cyclic-distributed domain C
23 var cyclicArray: [C] int;
24
25 // parallel loop over blockArray
26 forall a in blockArray do
27     a = a.locale.id;
28 writeln(blockArray);
29
30 // parallel loop over cyclicArray
31 forall a in cyclicArray do
32     a = a.locale.id;
33 writeln(cyclicArray);

```

Listing 3.9: Example mapping of indices for Chapel’s Blocked and Cyclic distributions (Cray Inc, 2015a)

Output (run on 6 locales):

<i>blockArray</i>								<i>cyclicArray</i>							
0	0	0	0	1	1	1	1	0	1	0	1	0	1	0	1
0	0	0	0	1	1	1	1	2	3	2	3	2	3	2	3
0	0	0	0	1	1	1	1	4	5	4	5	4	5	4	5
2	2	2	2	3	3	3	3	0	1	0	1	0	1	0	1
2	2	2	2	3	3	3	3	2	3	2	3	2	3	2	3
2	2	2	2	3	3	3	3	4	5	4	5	4	5	4	5
4	4	4	4	5	5	5	5	0	1	0	1	0	1	0	1
4	4	4	4	5	5	5	5	2	3	2	3	2	3	2	3

The blocks are then calculated based on a the number of indices in the distributed domain and the block size. The calculation is wrapped inside a forall loop over the locales. The first iteration of the computation calculates the lower bound of each of the domain’s dimensions and the second iteration is used to calculate the upper bound. The cyclic distribution applies a slight variation to the above by using strided ranges. In Listing 3.9 the block of Locale 0 in the Block distribution is defined by the range $\{0..3, 0..2\}$ while for the Cyclic distribution the block is defined by the range $\{0..7 \text{ by } 2, 0..7 \text{ by } 3\}$.

Although, the explicit data and task placement contrasts with the shared memory and the SPMD programming model, it solves valuable control issues in distributed memory systems. Due to the global namespace, the declaration of a domain is the only modification required to enable shared-memory code to support distributed-memory operations. To ensure affinity forall loops are implemented in a way that allows iteration over the local index set. Finally, Chapel’s compiler does not assist in maintaining memory consistency, it is left to the programmer to enforce it via synchronisation constructs. As a result of Chapel’s relaxed memory semantics, memory consistency is only guaranteed for race-free programs.

3.7 Chapel’s Runtime Environment

Chapel depends on a number of runtime libraries and API’s that implement the low-level features of the language. The runtime layers, as shown in Table 3.2, are

organised as interfaces; they are implemented in C and linked to the generated code. Each of the layers implements a subset of the required functionality; communication, task, memory management and timing. Each interface provides a number of distinct implementations to support the required semantics. For the remainder of this section we will discuss the main implementations provided for the communication and tasking layers.

Runtime Layers
Communication Layer
Tasking Layer
Threading Layer
Timers
Launchers

Table 3.2: Overview of Chapel’s runtime system layered architecture.

3.7.1 The Communication Layer

Chapel’s communication layer is written in C and it is a thin wrapper of GASNet (Bonachea, 2002), a network- and language-independent communication interface tailored to the PGAS programming model. GASNet supports multiple communication protocols (e.g UDP, MPI) for the implementation of the low-level communication API, with reportedly good performance results (Titus, 2011).

GASNet Core API

The main concepts in GASNet’s Core API are **nodes**, **threads** and **jobs**. Nodes are the main units of control and return values from `gasnet_init()` calls. A node represents an OS-level process, associated to local memory and system resources. Threads within a node share virtual memory and OS-level process identifiers. Threads of a multi-threaded node are equal, no master-slave distinction is made. Control functions execute using a single thread on the node on behalf of every locale. Finally, jobs build a parallel execution environment on a set of nodes that often correspond to physical units.

Errors that occur during a call to the GASNet core functions or the extended API are fatal, except if otherwise specified. If a node within a GASNet job crashes,

aborts, or suffers a fatal hardware fault, GASNet attempts to terminate the remaining nodes in a timely manner to prevent the creation of orphaned processes.

As we will discuss in Chapter 4, since GASNet is not designed for resilience, many of its design choices inhibit the purposes of the resilient implementation, where the main goal is to enable the runtime to handle the orphaned processes safely on a higher level, notably on the tasking layer. We will also discuss in detail the required communication layer changes to ensure that the application does not terminate and the challenges faced in the design of a testing mechanism to simulate locale failures.

Active Messages

GASNet builds on top of Active Messages (Eicken, Culler, Goldstein, and Schauser, 1992), a library that implements dual-message communication with logically matching request and reply operations. Upon receipt of a request message, a request handler is invoked; likewise, when a reply message arrives, the reply handler is invoked. Request handlers can reply at most once to the requesting node. If no explicit reply is issued, the layer may generate one, automatically. Thus a request handler may reply once to the requesting node, while reply handlers cannot issue requests or replies.

A high-level description of an ActiveMessage exchange between two nodes, A and B is described below, where (*) is a wildcard character, used to represent a predefined system function. Figure 3.2 demonstrates a representation of the message exchange between the two nodes.

1. A calls `gasnet_AMRequest*()` and sends a request to B. The call includes arguments, data payload, B's address (node index) and the index of the request handler to run on B when the request arrives;
2. At a later point, B receives the request, and runs the appropriate request handler with the arguments and data (if any) provided in the request call. The request's handler performs calculation on the arguments and issues a reply message (`gasnet_AMReply*()`) before exiting. The reply handler copies the token passed into the request handler, the arguments and data payload and the index of the reply handler to execute when the reply message arrives. A node index is not required as the request handler is only permitted to send

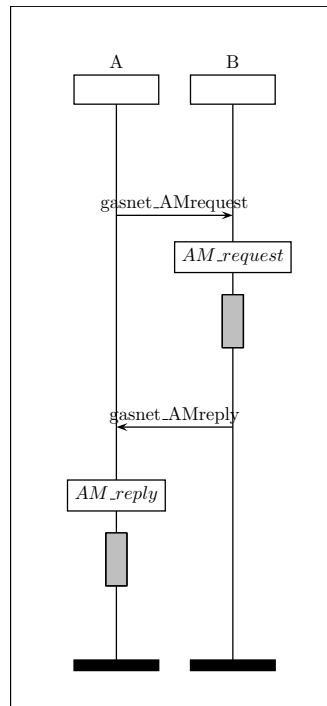


Figure 3.2: High-level representation of an Active Message communication between two nodes. Node A sends a request to Node B, including arguments and data and the index of the handler to execute on Node B on reception of the request. Node B performs the required calculation and issues a reply message to Node A. On reception of the reply, Node A calls the corresponding reply handler, performs necessary calculation and exits. Any further messages from Node A signify the beginning of a new cycle of communication.

a reply to the requesting node;

3. Later in the execution, A receives the reply message from B and runs the corresponding reply handler, with the arguments and data provided in the `gasnet_AMReply*()` call. The reply handler performs any necessary calculation on the arguments and exits. A is not permitted to produce further messages.

From a sender's point of view, request and reply functions are blocked until the message is *sent*. A message is perceived as sent, once it is safe for the caller to reuse the occupied block of memory. For implementations that copy or buffer messages for transmission, messages are defined as *sent*, as soon as the message layer has copied the message.

The layer is *best effort* and any necessary re-transmissions and buffering issues on unreliable networks, are handled transparently to the upper levels. This property of the communication layer is fundamental for our design and relates to our assumption

about *Network partition*, as discussed in Section 4.2.1. We require that the failures of the underlying network are either handled transparently or lead to a system-wide abort.

In either case, *sent* does not necessarily imply *received*. Once the control returns from a request or reply function, clients cannot assume that the message has been received and handled at the destination. The message layer only guarantees that if a request or reply is sent and the receiver occasionally polls for arriving messages then the message will eventually be delivered. From a receiver’s point of view, a message is defined as *received* only once the corresponding handler function is invoked. The contents of partially received messages and messages with unexecuted handlers are undefined.

If the client sends an `AM_Request` or `AM_Reply` to a handler index that has not been registered on the destination node, GASNet will terminate the job. Each specific implementation defines whether the sending or the receiving end handles the `null` check.

Based on the above discussion of the main components of the communication layer we identify the main issues that require special handling in order to support resilience. Firstly, the strict GASNet policy regarding node crashes and fatal failures needs to be tackled in order to allow the runtime to execute the required functions to perform task adoption and re-execution. The second issue that arises is the testing mechanism; specifically, the simulation of node crashes without the system-wide abort dictated by GASNet. We expand on these issues in Sections 4.3.2 and 4.4.2 of the next chapter.

3.7.2 The Tasking Layer

Tasks are the main units of parallelism in Chapel, they express computation that can execute in parallel. Tasks are defined as processes that form a distinct execution context, potentially composed of multiple threads. The underlying **threading** layer is used to execute tasks. From a programmer’s view, tasks are the predominant construct and the control of lower-level features, such as scheduling or mapping of tasks to threads, is deliberately hidden on the application level. The abstraction of tasks promotes programmability and provides flexibility and low-level control on

scheduling for the runtime layer while parallelism, both implicit and explicit, is implemented in Chapel via asynchronous tasks.

The task management interface is primarily responsible to implement the tasks generated by Chapels `begin`, `cobegin` and `coforall` statements and to implement the full/empty semantics required for synchronization variables. The tasking interface supplies calls for startup and shutdown, singleton task creation, management and execution of task lists, in addition to synchronisation and task queries. On program startup, the only existing task is `main()`, while task lists are created dynamically to manage the tasks associated with `begin`, `cobegin`, and `coforall` constructs. Accumulating the tasks related to a construct in a task list is beneficial for executing them in the desirable order with respect to all the other active tasks, given a potentially limited number of hosting threads. In particular, this practice allows the thread that hosts the parent task to run all the children tasks resulting from that construct one after another serially, when all other threads are busy with the execution of other tasks and the maximum number of threads has been reached. This prevents one kind of deadlock due to having more tasks than hosting threads.

FIFO tasks over POSIX threads (*pthreads*), is the most commonly used tasking layer implementation provided by Chapel and though it is heavy-weight, it maintains its portability benefits. Another tasking layer implementation is *Qthreads* developed by Sandia National Laboratories. The benefit of Chapel tasks over *Qthreads* is a light-weight implementation with enhanced support for synchronisation. *Qthreads* became the default tasking layer for Linux platforms from Chapel version 1.12 onwards. Other alternatives include *MassiveThreads* (Nakashima and Taura, 2014), developed by the University of Tokyo and the `muxed` tasking layer implementation, which is specific to Cray systems.

FIFO tasks

In the *FIFO tasking* layer, a Chapel task is mapped to a thread and executes on this thread until completion. The number of active tasks of a program at each point cannot exceed the number of existing threads. If the program, creates more tasks, these are placed into a task pool, until a thread picks them for execution. The name of the tasking implementation implies that POSIX threads will pick up tasks to execute,

preserving their order of creation by the program. As POSIX threads (**pthread**s) creation is expensive, the scheduler does not destroy them when they become idle. Instead, threads continue to check the task pool for new tasks. The number of **pthread**s to be used in a Chapel program is configurable via environment variables. The FIFO implementation of the tasking layer can prove more heavy-weight on some platforms compared to other implementations, mainly because it is tied to POSIX threads, which typically carry a lot of context and may apply restrictions. For example, in Linux, user pthreads are implemented as kernel threads; requiring more time for context switching, and are commonly governed by restrictions on their pre-emption.

Qthreads

Qthreads is a cross-platform parallel runtime environment designed to support programming with light-weight processes. *Qthreads* supports synchronisation and atomic operations, while the runtime assumes shared work queues and work stealing. Atomic operations and the subsequent context switching is performed using function calls outside the kernel. This results in less state information being persisted, and more opportunities for the scheduler to hide communication latency (Wheeler, Murphy, Stark, and Chamberlain, 2011).

Qthreads implements a two level hierarchy with *shepherds*, that control work distribution, and *workers*, that host *Qthreads*. Processes are assumed not to compete with other *Qthreads* over CPU and memory, by default. As Chapel programs often execute with a configuration of one locale per node, *Qthreads* is suitable for high-performance computing applications. Finally, the implementation depends on the third-party **hwloc** library, which provides a description of the locale aware hardware.

Tasking Layer’s Subsidiary Role

The tasking layer also performs the subsidiary role of supporting **on** statements; it executes a dual-phase synthetic task; by firstly scheduling the body of the **on** statement for execution on the target locale (child) and on completion of the execution sets an internal flag on the initiating locale (parent) to mark the completion. The communication layer on the target locale launches the body of the remote task by

calling `chpl_task_startMovedTask()`, a dedicated function of the tasking layer.

While `on` statements are the main mechanism to perform task migration in Chapel, they are part of the design of the communication layer. The implementation takes advantage of the tasking layer’s capabilities to simplify the intended functionality. Due to the acknowledgement-based method for execution ordering, a task within an `on` statement is not added to the global task list, instead the initiating (parent) locale is responsible for monitoring the completion of its execution.

Tasking interface

Tasks are assigned unique identifiers (`taskID`) serially when added to the task pool. Their data type and default value are used to exchange task identifiers between C and Chapel code. In contrast, thread identifiers use negative integer values to make a clearer distinction to tasks.

Two important functions within Chapel’s tasking layer API are `addToTaskList`, which appends tasks to the task list as they get created for the construct on the current locale and `executeTasksInList`, which ensures that all tasks in the task list have begun execution.

Chapel implements a task table, on module level, to monitor tasks in the system, for debugging purposes. Information about the state of tasks and the file and line they are created in the original program is maintained, while the task table is initialised in parallel on each locale, after the initialisation of the tasking and threading layers. To avoid referencing tasks that are created before the table’s initialisation, for example the `main()` task, all operations on the table are checked for membership. Table 3.3 demonstrates the information that is available in the current implementation of the runtime within the task lists and the task pool. The concepts of *task identifiers*, *state* information and *task list* are used extensively in the implementation of resilience on the tasking layer in the next chapter.

Number of Threads

The number of threads used to execute a Chapel program can be controlled by the environment variable `NUM_THREADS_PER_LOCALE`. The default value is zero, indicating that the choice of number of threads is left to the tasking layer. Chapel programs will generate an error if the requested number of threads per locale is out

Task Info	Location	Description
<code>taskID</code>	task pool	Identifier of the task
<code>fn</code>	task list, task pool	Pointer to the function to be executed
<code>arg</code>	task list, task pool	Pointer to the arguments of the function
<code>begun</code>	task list	Boolean, indicates that a task has begun
<code>filename</code>	task list, task pool	Filename where the task was started
<code>lineno</code>	task list, task pool	Line number in the <code>filename</code>
<code>prvData</code>	task list	Task private data
<code>prev/next</code>	task pool	Pointer to previous/next task in the task pool
<code>ptask</code>	task list	Pointer to the task pool
<code>ltask</code>	task pool	Pointer to the task list
<code>state</code>	task table	Task state: pending, active, suspended
<code>tl_info</code>	task table	Information for the tasking layer
<code>sublocale</code>	task table	Requested sublocale for a moved task
<code>task_list_locale</code>	task list, task pool	Indicates the locale on which the task list resides
<code>is_begin_statement</code>	task list, task pool	Boolean, indicates that the statement is a begin

Table 3.3: Properties and functions of tasks in the runtime system. All tasks created within the system are placed in the globally accessible `task pool`. Tasks that are created in the context of a block such as a `cobegin` block or a `coforall` loop form `task lists`.

of bounds. For example, when using the GASNet API on a multi-locale setup, the number of threads is bound to 127 or 255 threads per locale.

For *Qthreads* and *MassiveThreads*, the value of `NUM_THREADS_PER_LOCALE` specifies the number of system threads, or shepherds, used to execute tasks. If the value is 0, the tasking layer creates an equal number of threads to the number of processor cores on each locale.

In the FIFO tasking layer, the value of `NUM_THREADS_PER_LOCALE` indicates the maximum number of threads that can be created on each locale. The threads are created on demand, meaning that in a program with few concurrent tasks the maximum number of threads may never be reached.

For a program with heavy computational load and few inter-task dependencies, it is recommended that the number of threads is set equal to the number of physical cores. On the contrary, a program with fine-grained synchronisation, can use a

larger number of threads without impacting performance, since idle threads will not consume CPU cycles. In particular, a larger number of threads in this case is recommended for effective latency hiding. The *Qthreads* implementation is tuned towards performance, rather than load balancing, which is beneficial to Chapel programs that typically execute on one locale. For a program using multiple locales mapped to a single node though, this scheduling strategy can impact performance negatively due to resource starvation. To this end, the programmer can enable the oversubscription option (`CHPL_QTHREAD_ENABLE_OVERSUBSCRIPTION`) to introduce load balanced scheduling of processes. Finally, setting the number of threads very low, can result in deadlock, if the program produces larger numbers of concurrent tasks.

Task Call Stack

All executing tasks are associated to a call stack, the size of which is specified by the `CALL_STACK_SIZE` environment variable. Tasks that require more space than the amount provided by their call stack, result to stack overflow. Chapel programs are designed to check for overflow by default, though at the moment, these checks are implementation dependent.

For FIFO tasks the typical call stack size is 8MiB, while there is an additional guard page at the end of each call stack. If a task causes stack overflow the guard page will return a bad memory reference; in the case of Linux environments this translates to a *segmentation fault* and leads to program termination.

3.8 Chapel's Resilience Directions

In version 1.16 Chapel introduced support and documentation for the *Replicated Distribution* where each participating locale stores the entire *replicant*, domain or array. As a simplistic example a domain of 4 (`{1..4}`) distributed with the Replicated distribution will store 4 indices per locale. Chapel also supports access operations to the local replicant on each locale, while the consistency between replicants on different locales is not automatically maintained.

The replicated distribution, although still a work in progress, makes clear the intention of the Chapel team to support a form of in-memory data redundancy scheme, and subsequently reveals the interest for resilience support in the future.

Since the Replicated distribution cannot currently be combined with other standard distributions, it has not been the focus of experimentation for this work, since our intention has been to support resilience for a widely used pattern of data distribution, the Blocked distribution.

3.9 Summary

In this chapter we discussed Chapel’s origins and introduced the main constructs of the base language. We have focused on Chapel’s *parallelism* and *locality* design directives and discussed the functionality of task- and data-parallel constructs and mechanisms. We have introduced the main concepts that guide our resilient design, the explicit placement of tasks with the `on` construct, the constructs that introduce parallelism and the data distributions.

In the second part, we expanded on the runtime system, focusing on the communication and tasking layers. We also pointed out the implementation details that require special handling in order to allow resilient support, such as the strict failure policy of GASNet. We will be detailing the design and implementation of resilience in Chapters 4 and 5, for the task- and data parallel part, respectively.

Chapter 4

Resilient Task Parallelism

In this chapter, we present the design and evaluation of our extensions to Chapel’s runtime system to support resilience for serial and task-parallel constructs. Central to our implementation is an *in-memory data redundancy scheme*. We detail the internal runtime system functionality and the required additions and modifications to support uninterrupted program execution in the presence of failures. We evaluate a prototype for serial distributed execution; serial programs that utilize `on` constructs to execute remote tasks, and subsequently extend the functionality to support Chapel’s task-parallel constructs. We run experiments on a commodity cluster configured with up to 16 locales. Additionally, a focal point of the resilient design are *buddy locales*, a core concept that drives the scheduling of task adoption and recovery of failed tasks.

4.1 Foundations of Resilience

In Chapter 2 we have discussed the principles of software level fault-tolerance (Section 2.3.2) and expanded on the concept of the *resilient store* as a data structure able to survive failures. In Section 2.4, we detailed two widely used frameworks for resilient data storage, HDFS, a representative example of external file systems and ZooKeeper, a coordination framework for in-memory store.

Earlier in Chapter 2, we motivated our design decision to implement a custom in-memory data store to ensure redundancy of the data required for task re-execution. In the context of this work, we are not concerned with providing redundancy of

the computation, similar to the technique of the *N-version programming* of Section 2.3.2, though we recognise that the buddy locale mechanism could become the basis for such an approach. In our design a successful task completes execution once, and there is no duplication of the produced results. Such an approach to resilience, is more common either on hardware-level fault tolerance or on software for large production systems, with spare resources that perform redundant calculations and decide based on voting algorithms. Here we are concerned with providing a general purpose mechanism for resilience, available to the average Chapel user (with access to commodity hardware), where all available compute resources perform in the best case scenario (of no failures) only their locally assigned work. In the below paragraphs we detail the implementation of our in-memory resilient data store.

For the purposes of data redundancy, this work employs *in-memory replication* on the networked nodes, as discussed earlier in Section 2.4. We avoid the use of external mechanisms, such as file systems. This is primarily a design decision to propose a resilience mechanism, independent of external systems or third-party software and in accordance to our goal of *transparency*; the Chapel programmer, is able to use the resilient version of the runtime system out-of-the-box, similarly to any Chapel release. In Figure 4.1 we port the simple diagram of Chapter 2, to an one-to-one mapping between nodes and Chapel locales.

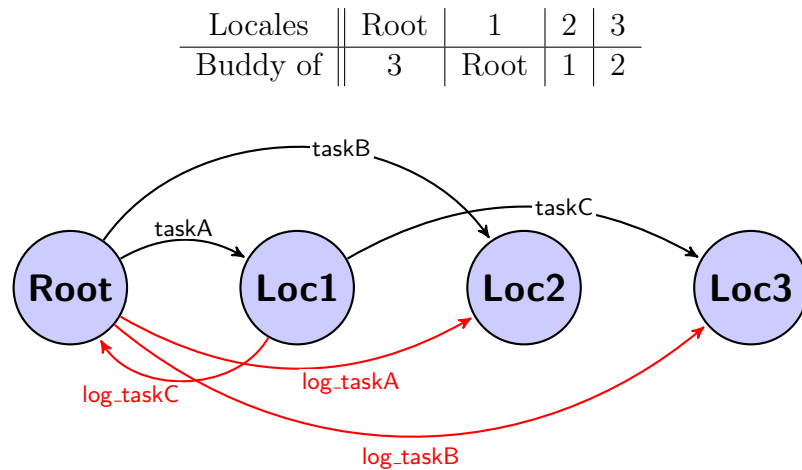


Figure 4.1: Sample communication flow of in-memory replication of tasks (and data) among guest and buddy locales. `log_task{X}` represents the copied task descriptor of the remotely spawned task `task{X}` on the buddy locale.

4.1.1 Data Structures and Resilient Store

The resilient in-memory store is implemented within Chapel’s runtime, on the communication layer, using a simple interface and a C singly-linked list. The list stores *in-transit message* structures, as shown in Listing 4.1, that capture the context of task descriptors and it is populated during execution via a runtime-level message-logging mechanism. The list is updated on execution of the ActiveMessages (AM) handlers of `IN_TRANSIT` and `IN_TRANSIT_DEL` signals. The main operations on the list are *append*, *delete* and linear-time *lookup*. For serial remote execution we use a single list which is persisted on the root locale, while for parallel remote execution, partial shorter lists are created on each locale, as we will explain later in this chapter.

In an earlier instance of the implementation we attempted the use of separate structures for the task descriptors and the variable-sized arguments, in order to provide a cleaner interface. This solution was abandoned due to the overhead of accessing two data structures on every update and lookup operation, and instead motivated the composite in-transit structure direction as a low overhead approach.

```

1 struct chpl_comm_transitMsg{      // in-transit message struct
2   chpl_fn_int_t      fid;         // function id
3   int                mid;         // message id
4   int                src;         // source address
5   int                dst;         // destination address
6   void*              ack;         // return address for ack
7   int                arg_size;    // size of arguments
8   char               arg[0];      // variable-sized arguments
9   chpl_comm_transitMsg* next;    // pointer to the next msg
10 };

```

Listing 4.1: Snippet of the definition of the *in-transit message* the C structure introduced in our implementation to capture the properties of an in-transit message.

Listing 4.1 details the `transitMsg` structure which is complemented by a minimal API for reading, writing and updating the list. The order of read and write operations on the list is imposed by the serial processing of the FIFO message queue, thus consistency is guaranteed.

Furthermore, we implement an array to store information on the status of locales (`failed_t` structs, Listing 4.2). The *status array* is of equal size to the number of locales (*numLocales*) and is used to track the status of each locale. In contrast

to the linked list, we allow relaxed consistency for the status array; the array is updated after the detection of a new failure; when **FAIL** or **TIMEOUT** signals arrive by remote locales or when a failure is detected locally. Since our system does not currently allow *resurrection* of failed locales and GASNet prohibits new nodes from joining the configuration, we can assume the status information to be up to date only with regard to locales that have already been visited and found failed. Thus, a locale is certain to have failed when marked as such, but locales that are not marked as failed may or may not have failed. It is also possible that a failure is detected simultaneously by multiple locales.

```

1 typedef struct{
2     int dId;           // node id
3     int parentId;      // parent node's id
4     int alive;         // status information
5 } failed_t;

```

Listing 4.2: Snippet of the *status* C structure, introduced by our implementation to capture status and hierarchy information on locale *dID*.

For serial remote Chapel operations we store a single copy of the *status array* on the root locale, which poses a scalability concern for applications with fine-grained computation, but lifts the communication costs of broadcasting status updates on failure detection and the memory overhead of maintaining updated copies across locales; both non-negligible when scaling to higher locale numbers. On the other hand, for the non-blocking remote operations, we distribute partial arrays on all participating locales aiming for scalability. In the following sections we detail the implementation for both cases.

4.2 Design Aspects

In this section, we discuss a set of assumptions that govern the design of the resilient mechanism. We expand on Chapel’s design aspects, control flow and internal synchronisation mechanisms; concepts that occur throughout the resilience design and implementation.

4.2.1 Assumptions

In the context of Chapel, when locales are not structured hierarchically (flat locale model), failures of networked nodes are realised as locale failures. We perform recovery using task migration, both for blocking and non-blocking fork operations.

We base our design on a set of assumptions that should be fulfilled for resilience support on serial and task-parallel Chapel constructs. The same assumptions apply to Chapel’s data-parallel constructs, discussed in the following chapter. This set of assumptions is presented below:

- **Failure-free root:** The root locale (Locale 0, by default) is failure resistant and acts as resilient store for redundancy and data retrieval. In the context of baseline (non-resilient) Chapel, the root locale is a single point of failure for a number of critical services such as I/O handling, initialisation and node bootstrapping, and auxiliary tasks. Migration of such tasks during program execution is guaranteed to lead to fatal errors;
- **User-code resilience:** The resilient implementation is supported and applies only to the user’s code; we argue that failures during initialisation cannot lead to a fault-free execution and thus, in the context of this work, we consider such failures unrecoverable;
- **Failure notification:** A failing locale explicitly notifies of its failure. While this can be seen as a contentious design assumption, we argue that it is within the capabilities of modern HPC hardware, as discussed in Chapter 2. The delivery of the failure notification is guaranteed to the extent possible allowed by the underlying communication framework. At the moment, this is a software-based solution to compensate for the lack of a hardware capability; in future implementations this can be complemented by an out-of-band signalling mechanism which monitors health metrics of the participating nodes, similar to the one discussed in our previous work, Panagiotopoulou and Loidl, 2016.
- **Task atomicity:** As a consequence of the fail-stop model (Section 2.2.2) used in this work, the body of a task on a failing locale will either execute to completion or not at all. This is also dictated by Chapel’s modular and layered runtime structure, which poses obstacles in tracking the progress of tasks on subsequent layers (tasking and threading) and, additionally, by state mainte-

nance concerns. This fault model is in accordance to Chapel’s task-parallel design that employs a *fire and forget* pattern without tracking task progress. A similar principle is applied for Resilient X10 (Cunningham, Grove, Herta, et al., 2014) [pg. 2, Introduction], while in the next chapters we detail the mechanisms employed to facilitate the bidirectional exchange of status information on both the source and target locales. We argue that the above assumption can be relaxed for side-effect free computation, which can re-execute safely;

- **Network partition:** We require that any failure of the underlying communication network leads to system-wide abort. It is left to the communication API (in this case GASNet) to provide mechanisms for message replay and apply relevant corrective actions.

4.2.2 Failure Notification

The assumption that a failing locale produces a failure notification is made to surpass the issue of failure detection, as it is not the focal point for the resilient design. A significant number of studies (discussed in Chapter 2) are focused on the issue of failure detection. Software based solutions most commonly assume an external detection or monitoring mechanism that calculates metrics, such as heartbeat frequency, or even historical data of failures. Notably, even mature resilience solutions, such as checkpoint-restart frameworks, do not address the issue of detection.

This assumption is not a requirement for the resilience mechanism to function correctly, rather than a convenient addition to allow testing and sanity checking, as the available testing platform lacks self-monitoring capabilities. The assumption requires a single systemic signal being sent to one –up to the total number of buddy nodes– locale. A number of alternatives can be proposed to replace the failure notification mechanism, all of which require implementation outside the scope of this work. The signal can be replaced by a call to the GASNet interface that implements the failure notification (parametric to the identifier of the failed node). It is also possible to introduce an external mechanism that calculates the identifiers (based on the buddy calculation algorithm) and notifies only the buddy locales via a signalling interface. Finally, another alternative could be a publish/subscribe model, where notifications of failure are received by every node in the system. In the latter

case, our design must be modified to discard failure signals for nodes that are not guests of the locale.

4.2.3 Tasking Interface

Chapel’s default tasking layer implementation for the majority of its target platforms is based on Qthreads (Wheeler, Murphy, and Thain, 2008; Wheeler, Murphy, Stark, and Chamberlain, 2011), a user-level threading package implemented by Sandia National Laboratories. Qthreads provides a lightweight implementation for Chapel’s tasks and an optimized implementation of synchronisation variables.

Qthreads implements *shepherds* that are in charge of the work distribution management, but do not have any fault tolerance capabilities. *Workers* host the Qthreads and correspond to Chapel tasks. A set of environment variables control the distribution of worker threads on the available hardware. The Qthreads implementation assumes by default that processes cooperate without competing over resources (CPU, memory). When combined with GASNet, which allows multiple locales to reside on the same physical node, the Qthreads API applies a set of optimisations, such as node overloading, to tackle resource starvation.

Chapel’s modular design does not expose the details of the underlying tasking (and threading) implementation on the wrapping tasking interface. The programmer is not allowed to alter the scheduling of tasks or access the thread’s stack. As such, in the context of the resilient design, we do not make any assumptions as to where recovery tasks execute. Such an assumption would restrict correct recovery execution, across different threading implementations (where the thread stack is not in use). In the current design, the modifications for resilience are adjusted to Chapel’s design directives. Each recovery is handled as a new task and its scheduling is handled by the underlying implementation, as for any regular Chapel task.

4.2.4 Task Synchronisation

Synchronisation and tracking of parallel tasks in Chapel is achieved with the use of the internal mechanism of `endCounts`. The `endCount` mechanism is implemented within Chapel’s modules and is used to explicitly, synchronise structured blocks of parallel tasks, such as `sync` blocks, as well as the implicit surrounding block of the

main function. They are also used for joining asynchronous `cobegin` and `coforall` task blocks. The implementation, as shown in Listing 4.3, is based on a protected atomic counter and a set of auxiliary methods to allocate, increment/decrement the counter and implement a *wait* functionality. The initiating task of a synchronised block increments the counter before dispatching each new task.

A new `endCount` object is allocated for each synchronised block, while a wrapper function captures the scope of the block. On creation of a task, the `endCount` reference is passed to the wrapper. As tasks are added to task lists for execution the corresponding atomic counter is incremented and on completion of the task the counter is decremented. This mechanism introduces added communication both to and from the initiating locale. The main function itself is governed by an `endCount`.

```

1 // endCount class
2 class _EndCount {
3     var i: atomic int,
4     taskCnt: taskCntType,
5     taskList: _task_list = _nullTaskList;
6 }
7
8 // endCount object allocation
9 proc _endCountAlloc(param forceLocalTypes : bool);
10
11 // free endCount object
12 proc _endCountFree(e: _EndCount);
13
14 // increment task counter
15 proc _upEndCount(e: _EndCount, param countRunningTasks=true)
16     ;
17 // decrement task counter
18 proc _downEndCount(e: _EndCount);
19
20 // wait on counter to become zero
21 proc _waitEndCount(e: _EndCount, param countRunningTasks=
    true);

```

Listing 4.3: The `endCount` class constructor with the corresponding method interfaces to allocate and delete an `endCount`, update the task counter, and perform a wait operation.

The master thread of each block of tasks block-waits on the counter to become zero. The method `executeTasksInList(void)` is invoked, during the wait period, to allow the main thread to contribute to the execution of tasks. When all tasks have completed, the method `freeTaskList(void)` is invoked to free the memory

occupied by the task list.

4.3 Communication Protocol Extensions

In this section, we provide a detailed discussion of the resilient mechanism for Chapel’s serial and parallel remote task execution, focusing mainly on the *communication layer*. We expand on the extensions of the resilient implementation and discuss an optimisation for fast *local recovery* of failed tasks.

4.3.1 Serial Remote Task Execution

Base Chapel An overview of the function invocations that take place within the communication layer during the execution of an `on` statement on a remote locale is demonstrated in Figure 4.2. The operation that takes place under the hood is a blocking fork (`chpl_comm_fork`). A `fork_t` struct, which contains the function and data to execute on, is sent to the child locale for execution. Once the signal is received on the remote locale (child), the corresponding `AM_fork` signal handler executes and the task is wrapped in a new `fork_wrapper` context. The `fork_wrapper` looks up the task identifier in the globally accessible function table and retrieves the arguments of the fork. Subsequently the task is handed to the tasking layer and scheduled for execution. On completion of the task a new active message (`SIGNAL`) is dispatched to the initiating parent locale to indicate the successful remote execution.

Back to the parent locale, there is a pending block-wait on the remote task. The parent locale periodically polls for new messages until it receives the `SIGNAL` message. GASNet’s API requires bootstrapped nodes to poll for incoming signals, both as means to handle new requests and as a *heartbeat* mechanism to indicate that they remain functioning. On receipt of a signal, the parent executes the corresponding handler function and builds a `done_t` struct, of similar functionality to the `endCount` mechanism. Finally, the block-wait is released and the control flow continues with the next statement on the parent locale. Adhering to GASNet’s documentation it is assumed that the condition of a `BLOCKUNTIL` call, can only be modified by the execution of an Active Message handler, in this case the `AM_signal` handler function.

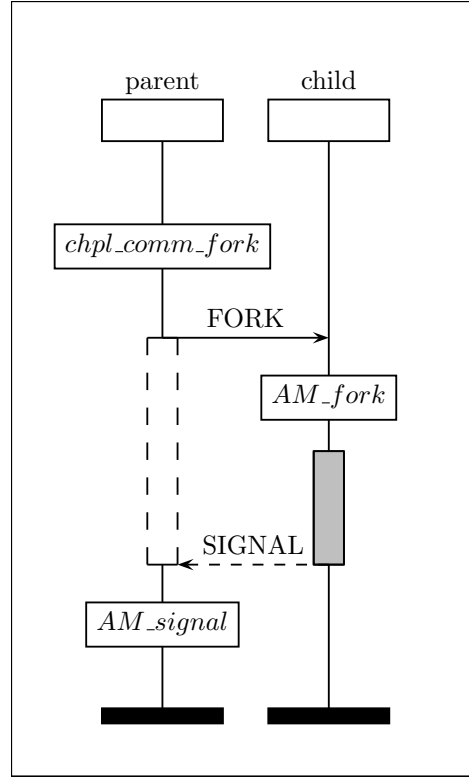


Figure 4.2: Execution flow of a blocking remote `on` statement from a parent to a child locale. The blocking `fork` operation is initiated from the parent locale. The child locale indicates the completion of the remote task with a `SIGNAL` message. Signals are handled by the corresponding Active Message handlers.

Resilient Chapel Figure 4.3 demonstrates the added functionality, including communication, signal handlers and data structures, that support resilience for the blocking fork operation of Figure 4.2. We provide four new signals: `FAIL_UPDATE`, `FAIL_UPDATE_REP`, `FAIL` and `TIMEOUT` and their corresponding handlers. The first two signals are exchanged between the parent locale of a remote task and the resilient store (in this case, the root locale) and are required to query the status of the remote locale, before dispatching the new task. In the case of multiple locale failures in the system, each parent locale will refrain from detecting already known failures, and thus spawning remote tasks that are certain to fail. The latter two signals complement the in-memory resilient store mechanism, as they propagate status information to the resilient store.

The root locale acts as store for status information using the `failed_table`; the status information is updated on detection of failures. The first action on the parent’s side is to request an update on the status of the child locale. This is a proactive detection mechanism to help save additional communication and execution

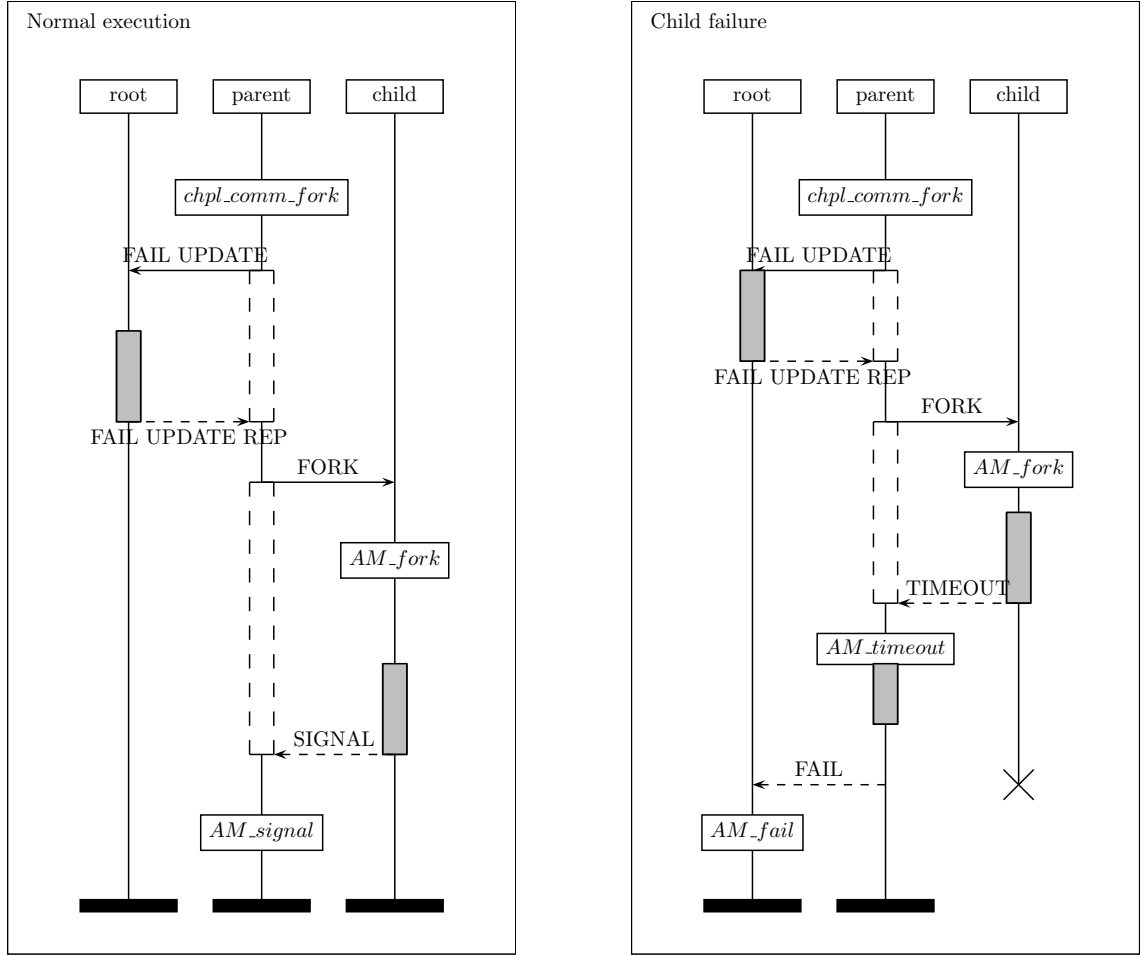


Figure 4.3: Execution flow of the resilient blocking `on` statement from parent to child locale *without* (left) and *with* (right) a single failure on the child locale. The **root** locale maintains information on the status of every other locale in the system. An update (status query) operation precedes the spawning of the fork operation. In the event of a failure (right) the child locale sends a `TIMEOUT` signal to notify of it's failure.

overhead, in the case that the child locale has been detected to have failed during execution of a previous task. In case of failure, this mechanism helps to save one additional message. We handle the exchange of information between the parent and the root locale using `FAIL.UPDATE` and `FAIL.UPDATE.REP` signals. The root locale looks up the child's identifier in the `failed_table` and retrieves the status information.

If the child locale has not been marked as failed, the parent proceeds with dispatching the remote task to the the child. On the child locale, the fork signal is received and the fork wrapper is constructed. On the right of Figure 4.3, we demonstrate the case where a failure occurs on the child locale. In this case, the child

locale notifies the parent for local failure with a `TIMEOUT` signal. On receipt of the `TIMEOUT`, the parent sends an update to the root locale for the newly detected failure (`FAIL`). In the case that the parent is the root locale, the update is consumed locally without any communication.

Recoveries of both previously known and newly detected failures are handled on the parent locale, making use of the function and data information that the parent already owns, by executing the task wrapper. The motivation behind this design is the availability of the evaluation context (function and arguments) on the parent's local memory. Failure of the parent locale, is handled on its immediate live ancestor (as a child failure), one level closer to the root locale.

According to our *task atomicity* assumption and the fail-stop model, a locale will either fail or will execute a local task till completion. When a new remote task is received for execution, the child locale has to obtain information on the current local status. Because our failure injection mechanism is message-based, and since the messages are stored in a queue and processed in order, the processing of a failure signal might be delayed. Thus, we add a detection capability in the local mechanism of message polling (present on every locale) to handle failure signals immediately, by updating the local status information. As such, a locale is always aware of its status and can notify the parent at any point prior to or between the reception of the remote task and the execution of the main body of the task.

In the context of recursive task spawning, drawing from *task atomicity* assumption, we are only concerned with the execution of the top level task. If the initial parent task does not complete then both the parent and children tasks are considered lost. When children subtasks have been spawned to remote locales previously to the parent locale's failure, then these tasks do not execute (since there is no parent to return the execution control), until the top level parent task is rescheduled for recovery. If the children locales have not failed by the time that the subtasks execute, then they will execute as normal tasks, but spawned from a different location—the buddy locale.

The above signals are sufficient to support the exchange of status information and the propagation of newly detected failures. The `failed_table` is maintained up to date, while it is also a single source of truth (SSOT) for already detected failures.

The alternative approach would be a decentralised design where status information is maintained locally on every node. We argue that such an approach would require data structure management on every node, and would eventually produce more communication for the synchronisation of locally stored status information. Furthermore, following one or multiple failures, the location where a recovery task will execute cannot be known statically, so an additional mechanism would be required to query the locale with the most up to date information, or using a more simplistic approach an one-to-all communication pattern would be required to access that information. Our design with the centralised status information and the new signals introduces communication, but alleviates the need for synchronisation of locally stored status information on every node. It also requires minimum data management on the parent locales, since only the status of the prospective child needs to be known at every instance of remote task spawning.

4.3.2 Distributed Remote Task Execution

Base Chapel In Figure 4.4 we demonstrate the non-blocking fork operation without resilience capabilities, implemented within Chapel’s communication layer, which results from the combination of the `on` statement with any of the task-parallel constructs (`begin`, `cobegin` or `coforall`). The parent locale launches the remote fork operation and continues with the execution of the next local statement without waiting on a completion signal. Instead, the `endCount` mechanism (Section 4.2.4) is employed to track completion of the remote task at a later point in the execution. Signals that arrive later are handled in the context of the `endCount` on the next synchronisation point or at the end of the user’s code.

Design Challenges

Resilient Chapel The first important difference when comparing to the *Serial Remote Task Execution*, of Section 4.3.1, is that the parent locale does not block wait on the completion of the remote task(s). As such, there is no guarantee that the task descriptor (function and arguments) will be available on the parent’s memory after the dispatch of the remote task, in case re-execution is required. Therefore, it is of relevance for resilience that this information is stored redundantly in a resilient

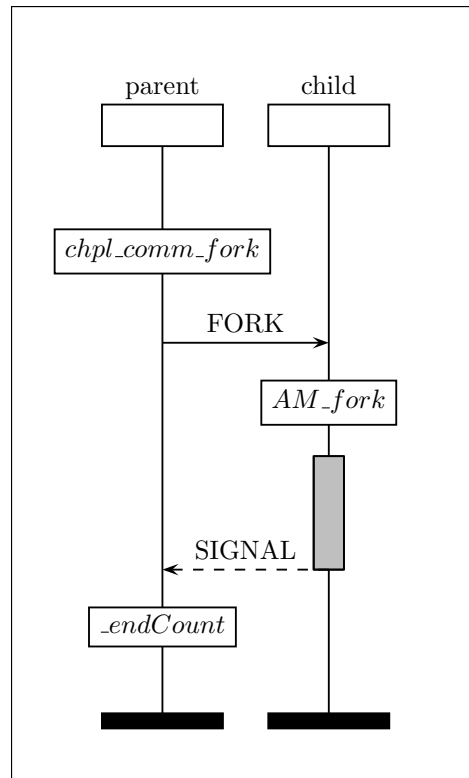


Figure 4.4: Execution flow of a remote non-blocking `on` statement from parent to child locale. The parent initiates the fork operation and proceeds with the execution of local tasks, until a completion signal is received.

store.

The lack of acknowledgements of remote tasks on the parent locale, complicates the tracking of live locales. On one hand, the parent locale does not have information on the status of the remote locale where the task is about to be launched. On the other hand, after launching the task the parent does not wait on task completion (and subsequent acknowledgement), that would verify that the remote locale remains functional. The parent continues execution, possibly launching further remote tasks, and ultimately exiting.

It becomes clear that the idea of sending a failure notification signal to the parent locale is not sufficient in this non-blocking environment, as recovery is not guaranteed in any of the two prevailing scenarios. In the first case, the parent locale may have exited after completing local work. As a result, local memory is garbage collected and when a failure notification arrives, it does not correspond to any local records of the task.

In the second case, the parent locale is in progress of executing local tasks when

the failure notification arrives; presumably task descriptors are available in local memory and can be reused. The challenge then becomes, to ensure that all children tasks either complete successfully, or fail within the lifespan of the parent locale, so recovery tasks can be relaunched. Since, this is a strong requirement for a distributed system operating on top of an asynchronous message-based communication network, a different mechanism is required to relaunch the tasks that fail after the parent has completed local work and exited. The lack of such a mechanism and the complexity of applying different recovery strategies based on merely timing constraints, poses concerns on the state of the computation and complicates the maintenance of correctness of the program and the evaluation of the recovery mechanism. We address this challenge by introducing the concept of *buddy locales* in the following paragraphs.

Another challenge in the context of the distributed remote-task execution is the design of an efficient mechanism to track the status of locales. On one hand, the parent may receive failure notifications for a task it maintains no record of. On the other hand, the child locale might forward a failure notification to a failed parent locale, risking the loss of information.

A naive brute-force approach would be to attempt to maintain the entire system up-to-date with liveness information. This would require the broadcasting of new failures to every locale in the system, resulting in one-to-all blocking operations on every failure, in addition to the overhead of managing the internal status tables on each locale. Assuming that these broadcasting operations do not lead to message congestion and that failure messages do not exceed the limits of local message queues, we would still need to ensure a low execution overhead. Assuming the overhead remains within the acceptable threshold, this approach would still fail to maintain a real-time picture of failures throughout the system due to the FIFO processing of the message queue. As a side-effect, locales that have no links or task dependencies to a failed locale, are burdened with the management of status information.

An example scenario is presented in Figure 4.5. We use a sample program where non-blocking tasks are spawned on Locales 0 to 3. Locale 2 executes `task2` initiated on Locale 1 and spawns `task3` to Locale 3. During execution Locale 2 fails, initiating a set of notifications to every participating locale, including Locale 0. In the next

instance, Locale 3 fails, spawning failure notifications to Locales 0 and 1. Based on the sample program shown in the figure, we note that Notifications 1, 4 and 5 are redundant, since the locales do not share any tasks.

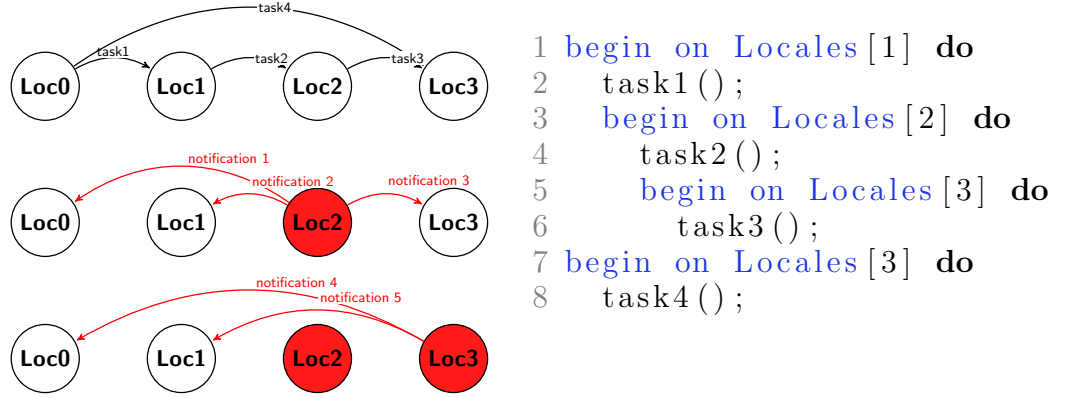


Figure 4.5: An example scenario with two failures (first, on Locale 2, then on Locale 3) in a multi-locale task-parallel Chapel program, using one-to-all failure notifications.

Based on the reasons explained above, we introduce *buddy locales* as an implementation of in-memory redundancy with data replication on a subset of the networked nodes. We create partial in-transit message lists on each buddy locale, which, when combined, cover status information on the entire system. In this setup, each buddy locale performs local *housekeeping* tasks and implements the data-structure management within the runtime system. We should clarify that buddy nodes are not idle, they perform local computation, apart from the computation of adopted tasks.

We adapt our resilient design to address the requirements of the distributed remote task execution according to the following design considerations:

1. The available task descriptors on the parent locale, during the launch of a remote task, are not reused for task recovery;
2. We perform in-memory replication of data on the *buddy locales* to achieve redundancy;
3. Only *buddy locales* receive notifications of a new failure, as opposed to system-wide failure broadcasts;
4. *Buddy locales* are responsible for task recovery in the event of a failure and notify the root locale for newly detected failures;

5. Only the root locale maintains system-wide status information.

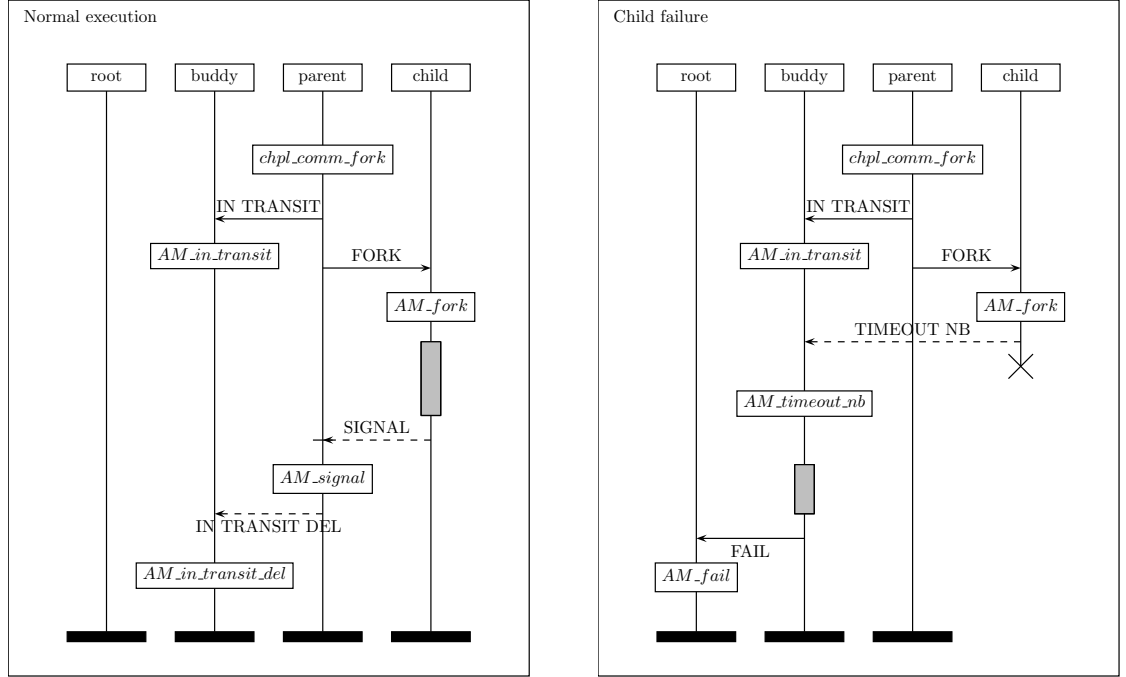


Figure 4.6: Execution flow of a resilient non-blocking fork from parent to child locale, *without* failures (left) and *with* (right) a single failure and recovery on the buddy locale. The parent locale notifies its buddy locale of a new fork operation (IN TRANSIT message) and initiates the operation. On successful completion of the remote operation and following the receipt of the SIGNAL from the child locale, the parent notifies the buddy locale to delete the locally stored task descriptor. In the event of failure, the buddy locale is notified and the remote task executes from locally stored data on the buddy locale.

Figure 4.6 demonstrates the non-blocking fork operation with the modifications to support resilience. We reuse the linked list, introduced earlier in Section 4.1, for the blocking fork case. The list captures the core part of the task descriptor with information on the parent and child locale, the function identifier and the variable-sized arguments; this data is sufficient to perform task recovery following the failure of an executing locale.

Similarly to the blocking fork case, we use a singly-linked list to store the task descriptors and an API to perform the required operations (read, write, update, delete) on the list. Each of the buddy locales maintains information only for a portion of the system; those task descriptors that belong to its guest locales. As such, each list in local memory is shorter in length, alleviating memory overheads and contributing to faster information retrieval. Finally, we maintain the centralised status array, stored on the root locale, which tracks the status of the system and is

of size equal to the number of locales (*numLocales*).

For the non-blocking fork operation we require that the parent locale establishes one-way communication to the buddy of the child locale prior to launching the fork operation. To store the data redundantly, the parent sends the task descriptor to the buddy locale using an `IN_TRANSIT` active message. The arguments are copied using a `memcpy` operation; which is expensive but essential in order to retrieve the arguments. It would be, otherwise, unsafe to use references or single element assignment to the parent locale to retrieve the task descriptor, since the parent is also susceptible to failure. Such a design decision would open the system to the possibility of unrecoverable tasks. The average size of the exchanged task descriptors is $44 + x$ bytes, where x is the length of the arguments array. The use of the primitive `memcpy` operation is generally optimized by compilers for larger data chunks. Nevertheless, Chapel's backend also defaults to `memcpy` for any size, as a clearer simple statement to allow any hardware-level optimizations.

The memory is freed either on successful completion of the task, as signalled by the child locale with an `IN_TRANSIT_DEL` signal or after the successful recovery of an orphaned task. On execution of the ActiveMessages (AM) handlers for `IN_TRANSIT` and `IN_TRANSIT_DEL` signals, the in-transit list, discussed in Section 4.1.1, is updated.

In the event of a failure, the primary buddy locale is responsible for the adoption of the tasks on the failed locale (child). The buddy notifies the root locale for any newly detected failure; information that is persisted in the *status array*. In turn, the buddy locale employs mechanisms to reconstruct the task from local memory and continues with the relaunching of the task.

For load balancing purposes we may choose to re-execute the task either as a new local task on the buddy locale or re-launch the task in a non-blocking manner on another locale in the system (possibly the parent locale), as long as it is known to be alive. Our current implementation complies with the first strategy in order to avoid creating more communication and data copying from further non-blocking fork operations.

The implementation is designed in such way that *the number of buddies per locale is configurable*. When N buddies are used, the application is guaranteed to tolerate $N-1$ failures within the same group of buddies. At the same time, we define the

order in which the buddy locales will attempt task adoption, defining *primary* and *secondary buddies*.

Though, the choice of the number of buddies is left to the programmer, the above communication and data management costs should be considered; for example a configuration with N buddies, will produce on average $2N$ additional messages per remote task in the application, during a failure-free execution. The additional messages correspond to one `IN_TRANSIT` signal from the parent to each buddy locale, in order to store the data of the remote task, and an additional `IN_TRANSIT_DEL` signal to notify each buddy locale to delete the entry in local memory, following the task's successful execution. The additional amount of data that needs to be stored, communicated and managed in the system accounts for another $N * numTasks$, where *numTasks* is the number of remote tasks in the application.

Furthermore, there is a trade-off between the total number of tasks created in the Chapel program and the number of added messages to support resilience. For a small number of independent tasks, a configuration with a maximum of two buddies per locale would be recommended, as it is sufficient to support recovery on a system with low/medium failure rates, while introducing runtime overheads.

As an additional feature to the buddies' mechanism, we require that the selection algorithm is programmable, within certain limitations, thus adding a layer of abstraction and allowing the systems programmer to adjust these parameters according to the application and the configuration. For example, for the execution of a Chapel program on a state-of-the-art HPC system, the programmer may choose to use a single buddy for each locale, aiming to minimize the memory and performance overheads. Regarding the systems configuration, when setting a compute node to act as locale, it would be cost-efficient to pick the neighbouring nodes as buddies, while in a hierarchical node set up, where a locale represents for example one of two NUMA regions on a node, an off-host buddy would be preferable to avoid unrecoverable data loss, as the possibility of both the primary and the buddy locale suffering failures is increased.

In order to allow the system to tolerate multiple failures within the same group of buddies, the buddy selection algorithm needs to fulfill a set of requirements. We consider the first locale in the list of buddy locales to be the *primary* buddy, while

all other locales are *secondary* buddies.

1. The selection algorithm needs to ensure that all locales are unique within the list of buddies;
2. The list of buddies of locale L cannot contain L itself;
3. Secondary buddies need to maintain information on the status of the primary buddy to ensure task recovery when multiple failures occur, thus it is required that secondary buddies are also buddies of the primary buddy locale.

When the buddy selection algorithm adheres to the above requirements the system is guaranteed to recover each orphaned task only once, as expected. The latter requirement ensures that when a primary buddy fails before executing a recovery task, the secondary buddy will be able to adopt the failed recovery task, thus no tasks are lost due to multiple failures. We should note, that the primary buddy is at every point in the execution the first functioning locale within the buddies array.

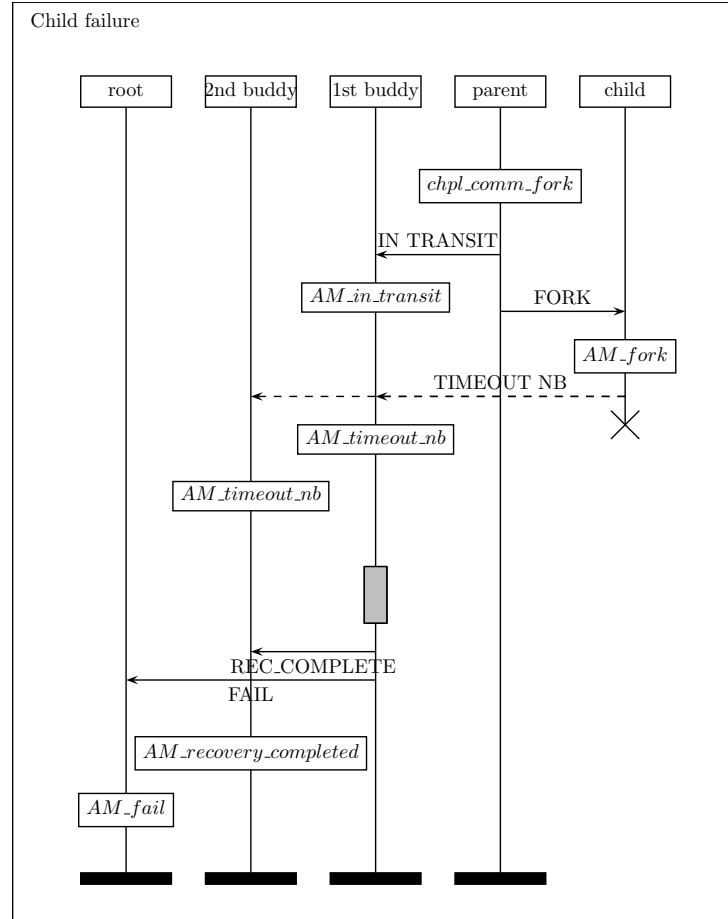


Figure 4.7: Execution flow of a resilient non-blocking fork task recovery with multiple buddy locales. After completion of the recovery task, the primary buddy notifies the secondary buddy (buddies) with a signal of completion.

To accommodate for the latter requirement we introduce a new `REC_COMPLETE` signal. The signal is used by the primary buddy to notify the secondary buddies for the completion of a recovery task. The secondary buddies are by default notified of any status changes of the primary buddy with a `FAIL` signal; i.e. when the primary buddy fails. Based on these two signals, the secondary buddy is able to identify whether to perform recovery for both the newly failed task and the previously detected recovery task. We use Figure 4.7 to demonstrate this functionality.

To clarify the above requirements we consider the example of Table 4.1. The system is configured with four locales and two buddies per locale. In Table 4.1 (A) we demonstrate the buddies per locale, i.e. Locale 2 acts as primary buddy for Locale 1, while Locale 3 is secondary buddy for Locale 1. Assuming that Locale 1 fails, both Locales 2 and 3 are notified. Locale 2 is responsible for performing recovery. As shown in Table 4.1 (B), Locale 3 is responsible for recovering primarily Locale 2 and secondarily Locale 1, thus Locale 3 has information on the status of both locales. When Locale 1 fails, Locale 2 begins recovery of the failed task of Locale 1, and sends a `REC_COMPLETE` signal to Locale 2. In the case that the primary buddy has also failed, Locale 3 will apply recovery for both the initial tasks of Locale 1 as well as for the orphaned tasks of Locale 2. For a configuration with multiple buddies per locale, buddy groups can overlap. Currently, the implementation does not support assignment of new buddy locales during program execution. When the buddy locales configuration complies to the above pattern, we eliminate redundant communication for recovery coordination on secondary buddy locales.

(A) Buddy Locales					(B) Guest Locales				
Locales	0	1	2	3	Locales	0	1	2	3
primary	1	2	3	0	primary	3	0	1	2
secondary	2	3	0	1	secondary	2	3	0	1

Table 4.1: Example configuration of 4 locales with two buddies per locale as captured (A) on the side of the locales and (B) on the side of buddy locales.

4.3.3 Optimisations and extensions

Earlier in the chapter we discussed the failure detection capabilities we have implemented within the polling functionality of the incoming communication. In order

to take full advantage of this mechanism we have added a fast recovery capability on the parent locale, which we refer to as *recovery on-the-fly*. On the parent locale, before launching the non-blocking remote fork, we check the following two boolean conditions:

1. *failed_table*[*node*].*alive* == 1
2. *buddyLocales*[0] == *chpl_nodeID*

In the local status table, the **alive** attribute is initially set to 0 for all locales. For the prospective child locale in the non-blocking fork operation we check whether the locale is known to have failed. If there is a known failure, we calculate the identifiers of the buddies of the child node and locate the primary buddy. If both conditions are satisfied and the current node is the primary buddy we proceed to perform recovery locally on the parent locale, without launching a remote fork operation.

A common technique to support fault recovery in large HPC systems is to maintain backup nodes, which are initially idle and replace the main nodes as they fail. In our system, the choice of buddy locales is based on the number of locales present in the configuration, persisted in Chapel’s *targetLocales* array and configured in the **SSH_SERVERS** list when building the runtime system. As such, we can configure Chapel programs to execute on a larger number of locales than the locales required in the program. Our design allows the additional locales to act as buddy locales, able to adopt tasks of failed locales, although they do not perform local work. Thus, our software-based solution can also be used on a system configured with back up nodes without modifications. As an example, we could build the non-blocking snippet of Listing 4.4 with **SSH_SERVERS=node0,node1,node2,node3**, choosing the next node as buddy locale and executing on four locales ($-nl4$). In the event of failure of Locale 3, Locale 2, even though it does not perform any local work, is responsible for the recovery of *task2()*.

```

1 on Locales [1] do begin
2   task1 ();
3 on Locales [3] do begin
4   task2 ();

```

Listing 4.4: Example Chapel snippet using two non-blocking remote tasks on two target locales.

4.4 Testing Interface

In this section, we present the constructed micro-benchmarks, used in the performance evaluation of the resilient task-parallel implementation. We discuss the design of the failure injection mechanism and detail the experimental setup. We also define the acceptable threshold of overhead for our resilient mechanism, taking into account the performance results of Resilient X10 as reported in bibliography.

4.4.1 Micro-benchmarks

We present a set of synthetic micro benchmarks to evaluate the resilience mechanism for serial and task-parallel Chapel programs. Figure 4.8 provides a graphical representation of the remote tasks launched in each micro-benchmark. We note that all Chapel programs begin execution on Locale 0.

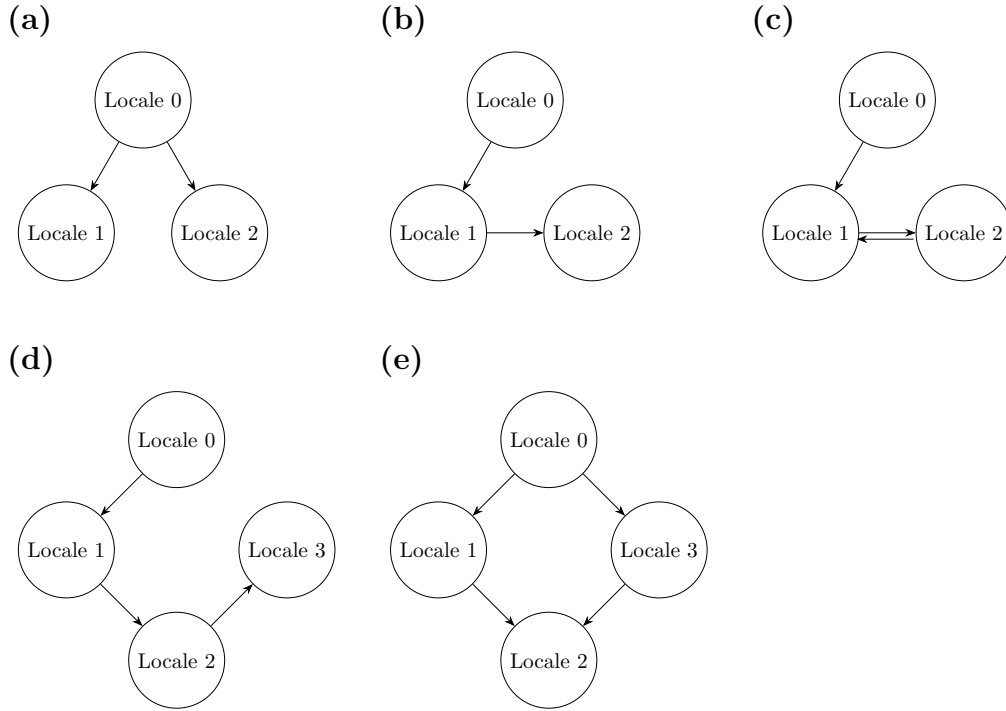


Figure 4.8: Graphical representation of the constructed micro-benchmarks used in functionality and overhead testing.

Listing 4.5 demonstrates a condensed version of the distributed serial micro-benchmarks for the blocking fork case. We later modify the code to introduce parallelism. The complete programs can be found in the Appendix (Section B.1).

simpleons.chpl <pre> on Locales[1] do monteCarlo(); on Locales[2] do monteCarlo(); </pre>	simpleontest.chpl <pre> on Locales[1] do { monteCarlo(); on Locales[2] do monteCarlo(); } </pre>	back.chpl <pre> on Locales[1] do { monteCarlo(); on Locales[2] do{ monteCarlo(); on Locales[1] do monteCarlo(); } } </pre>
three_on.chpl <pre> on Locales[1] do { monteCarlo(); on Locales[2] do{ monteCarlo(); on Locales[3] do monteCarlo(); } } </pre>	two_two_on.chpl <pre> on Locales[1] do { monteCarlo(); on Locales[2] do monteCarlo(); } on Locales[3] do{ monteCarlo(); on Locales[2] do monteCarlo(); } </pre>	

Listing 4.5: Snippets of the serial constructed micro-benchmarks used for performance testing. The micro-benchmark functionality can be summarized to the following: **simpleons.chpl**: two respective tasks launched on two distinct locales; **simpleontest.chpl**: a pair of nested tasks launched on two distinct locales; **back.chpl**: three nested tasks, where the first and the last are launched on the same locale; **three_on.chpl**: two pairs of nested tasks launched on three distinct locales and **two_two_on.chpl**: four tasks, nested in pairs, with the inner tasks launched on the same locale.

We name our constructed programs to concisely indicate their structure as follows: (a) **simpleons**: two respective tasks launched on two different locales; (b) **simpleontest**: a pair of nested tasks launched on different locales; (c) **back**: three nested tasks, where the first and the last are launched on the same locale; (d) **three_on**: three nested tasks launched on three different locales and finally, (e) **two_two_on**: two pairs of tasks, nested in pairs, with the inner tasks launched on the same locale.

Subsequently, we construct the task-parallel non-blocking versions using **begin** and **cobegin** constructs, as shown in Listing 4.6 for the **simpleons** micro-benchmark.

For the construct combination of **coforall+on** we launch one independent task

<pre> simpleons.chpl begin on Locales[1] do monteCarlo(); begin on Locales[2] do monteCarlo(); </pre>	<pre> simpleons.chpl cobegin { on Locales[1] do monteCarlo(); on Locales[2] do monteCarlo(); } </pre>
--	--

Listing 4.6: The task-parallel version of the `simpleons` constructed micro-benchmark using `begin` and `cobegin` constructs

per locale and execute the programs on an increasing number of locales, as shown in Listing 4.7, while the corresponding graphical representation is shown in Figure 4.9.

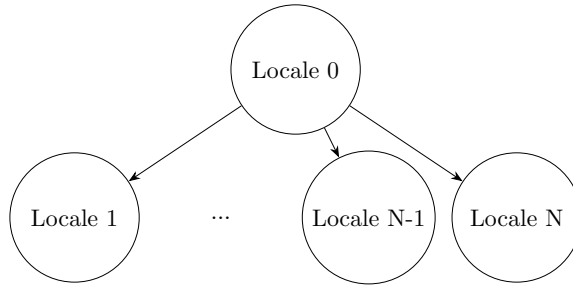


Figure 4.9: Representation of task spawning during the execution of a `coforall+on` loop. The loop initiates on Locale 0 and all locales execute the same task in parallel. When the iterable expression is `Locales` (i.e. every locale in the configuration) Locale 0 also executes local computation.

```

1 coforall loc in Locales
2   on loc do
3     monteCarlo();

```

Listing 4.7: Task-parallel `coforall` loop executing the Monte Carlo Pi approximation algorithm

The micro-benchmarks are designed with deep and shallow nesting of tasks, to be used as *stress tests* for the evaluation of the task recovery mechanism. We use the Monte Carlo Pi approximation method as the independent long running computation, launched on multiple locales, serially and in parallel. The same input size and same seed value are used on every locale.

The focus of this part of the evaluation is to study the overhead of the task parallel resilient mechanism, both in the absence and presence of failure. We have opted for a small number of locales, that enables us to demonstrate overheads with respect to the nesting of locales, for each task parallel construct. We are also able

to pinpoint the impact of failure on specific locales, taking into account factors such as the overall workload and the buddy locale correlations (Figures 4.16 and 4.20). The small scale of the micro-benchmarks allows us to discuss the resilient mechanism design in more depth, while also studying the performance penalties in configurations with increasing numbers of buddies per locales and cases with multiple failures (latter part of Section 4.5.2). Especially in the cases of `begin` and `cobegin` constructs, scaling up the number of locales would result in bulkier and difficult to study benchmarks with deeper nesting patterns, while these cases would not be representative of common idioms in real applications, as per our goal here.

4.4.2 Failure Injection Mechanism

As part of our assumptions, discussed in Section 4.2.1, we require that failing locales explicitly notify of their failure. This assumption is based on the rationale that modern HPC provides hardware monitoring devices that facilitate graceful degradation. In order to emulate failure(s) in the system, we use a signal-based failure-injection mechanism, written in Python and `bash` shell. We override the default behaviour of the POSIX *User defined signal 1* (SIGUSR1) handler, as specified in GASNet, and instruct the communication layer to ignore the signal via GASNet’s environment variables. A similar mechanism for node shutdown is implemented on the lower communication layer. GASNet (Bonachea, 2002) utilizes a *SIGQUIT* signal when it detects a node that has called the exit function or has crashed. Upon signal reception, nodes perform system-specific shutdown (if applicable) and call the exit function to end the local process.

We run the error injection script alongside the application, using Python’s multi-threading module. As specified in the `flat` locale model, the common and best performing case is the direct one-to-one mapping of locales to nodes in the system. As a first step the script iterates over the nodes, specified in the `SSH_SERVERS` environment variable during Chapel’s multi-locale setup. We choose one or multiple node Id’s and login remotely to the node(s). We also use a secondary testing mode (*random*), in which node identifiers are chosen pseudo-randomly, using Python’s random value generator.

The fault injector is abstracted from the application execution, in order to be

generic enough and reusable throughout the experiment. Furthermore, the failure injection mechanism looks into Unix-level processes executing within a set of nodes. Implementing an *in-app* fault-injection mechanism would require non-trivial effort on application level; the added functionality should not be taken into account in the overhead results, while we would also need an effective mechanism of mapping locales to lower-level processes; a direction opposite to the language’s layered architectural design.

On each node, we obtain the process identifier (`pid`) of the executing Chapel program, using bash commands (`ps`, `grep`, `awk`) and send `USR1` signals to the corresponding processes. Once the `USR1` signal is received by the process(es), the locale(s) will produce the failure notification and exit. As discussed in Section 3.7.1 the node will not shut down as this would lead to system-wide abort, according to GASNet’s failure policy. On the runtime level though the locale is considered inoperative; it will not process any further signals and it will cease execution of ongoing tasks and signal handlers. As a realistic node shutdown cannot be emulated without causing disturbances in the execution of the program, and for the purposes of testing the resilient implementation, we consider this testing interface flexible enough to emulate locale failures, in order to assess our implementation.

We use two testing modes: *all* and *rand*. The first mode simulates failures on every locale apart from the root locale and it qualifies as a stress test, while the latter simulates failures on a random number of locales within the bootstrapped set of nodes. The locales are picked pseudo-randomly using Python’s `random` generator module, hence *rand* emulates a failure scenario, where a random number of locales fails independently, following a pattern of uniform random selection from locales in the list without replacement.

The main goal of our testing is to evaluate the overhead introduced by the resilient implementation, while the testing interface itself poses two limitations. Firstly, the interface is not able to simulate failures at different times during execution; thus the failures occur serially with respect to each other, and are clustered in time. Secondly, we need to allow a short time frame in the beginning of the execution, after initialisation and before launching the remote tasks, to send the failure signals. We assume that the runtime requires fixed and comparable initialisation

time across locales, iterations, and test cases. The fixed delay is not accounted for in the demonstrated runtime results of the next section.

The evaluation section demonstrates a failure injection pattern with a left-skewed distribution over time. Locales will often fail early rather than late. This could potentially lead to an underestimation of overhead but taking into account the overhead of local updates for buddy locales, late failures also mean that the locales that will eventually fail, will perform administrative work for buddy locales for longer periods, thus introducing a larger overhead.

4.4.3 Experimental Setup

We present two sets of results representing the two different stages in the implementation. The first set addresses our prototype implementation for serial remote tasks (blocking fork operations) and it is built with Chapel version 1.9.0 and GASNet (v1.22.0). The rest of the setup for the serial remote tests is configured as follows: default flat locale model; where locales are homogeneous in terms of cores and cores have equal distance from memory, *fifo* tasks over POSIX threads; default memory (standard C malloc commands) and intrinsics. We use the GNU compiler suite (gcc v 4.4.7) and the *amudprun* launcher for 64-bit Linux platforms.

The second experimental set demonstrates the performance of the resilient distributed remote task spawning (non-blocking fork implementation). We use version 1.12¹ of Chapel, configured with *Qthreads*, and for our resilient version we test with one buddy per locale. As in the previous case, we use the flat locale model and the default memory configuration. In this setup, our system can tolerate one failure within each group of buddies.

We map locales to nodes and select the next neighbouring locale in the bootstrapped set to act as the buddy (Locale 2 is the buddy of Locale 1 and so on). We note that the system nodes that represent Locale 1 and Locale 2 might not be adjacent within the `SSH_SERVERS` list.

We perform 20 iterations for each test case using up to 16 nodes and present the

¹Between implementation phases, we ported our mechanism to the latest at the time Chapel version (version 1.12). From version 1.12 onwards, Chapel adopted *Qthreads* as the default tasking layer implementation, but the switch between implementations has been kept transparent on Chapel's tasking layer. We use version 1.12 for the remainder of the thesis, while, at the time of writing this thesis, the latest stable release is 1.18.

arithmetic mean for each test case. The experiments were performed on a 32-node Beowulf cluster (Sterling, 2003). Each node comprises of two Intel Xeon E5506 quad-core CPU's at 2.13 GHz, sharing 12 MB of RAM and connected via Gigabit Ethernet. The nodes run CentOS Linux release 7.5.1804 (Core) x86_64. Each core uses a 32KB L1i cache for instructions and 32KB L1d cache for data and an L2 cache of 256KB. Each quad-core also shares an L3 cache of 4096KB. A snapshot of the internal topology of a Beowulf node, as produced by the `hwloc` package, is available in the Appendix (Section A.2).

HPC platforms

Both in the following evaluation section (Section 4.5) and in Section 5.4 of the next chapter, we provide an extended discussion of overheads, often focusing on the communication costs. We do not provide experimental evaluation results on a modern HPC system both due to constraints imposed by the testing interface (linux-level signaling permissions) and due to time constraints. Nevertheless, we argue that the communication overheads that contribute to a large part of the *fixed cost* of the resilient mechanism, would not benefit from a high performance interconnect (HPI) of an HPC system.

In Bortolotti, Carbone, Galli, et al., 2011 the authors demonstrate comparative results for UDP over Ethernet and Infiniband interconnects for point-to point data transfers. The main conclusion is that for UDP frames smaller than 1500 Bytes the throughput is the same between the systems. Infiniband (RC mode) outperforms the Ethernet link for frames larger than 4000 Bytes, while the receiver's side is often burdened with higher CPU workloads. Based on the above study, and taking into account that the size of the exchanged messages in our task-parallel implementation is on the lower spectrum ($44 + x$, x being the variable sized arguments), we believe that the system would not benefit from a faster message rate of an advanced HPI. The bottleneck is in our case the communication protocol imposed by GASNet and Active messages. As we discussed in Section 3.7.1, on the sender's side the protocol imposes CPU idle time while blocking to send a message, while on the receiver's side there are delays introduced by the queueing and ordered processing of messages.

4.4.4 Benchmarking and Threshold Standards

In order to provide context for the evaluation of the resilient implementation we define a threshold of acceptable overhead. As Resilient X10 is closely related to our work, we look into the overheads, reported in Cunningham, Grove, Herta, et al., 2014; Grove, Hamouda, Herta, et al., 2017, as a baseline for our evaluation. As discussed in Section 2.5.1, Resilient X10 provides a *Place Zero-Based Finish* and a *Resilient Distributed Finish* implementation. Our resilient implementation shares common semantics with both approaches. The Place-Zero Based Finish, assumes that the first **place** (X10’s equivalent construct to Chapel’s locale) never fails, which is similar to our *Failure-free root* assumption, although in our resilient implementation we only store status information on Locale 0, instead of data used in the computation. In that matter, our implementation is comparable to the *Resilient Distributed Finish* approach, since we use the internal mechanism of buddy locales to maintain the data redundantly on the memory of the networked nodes and X10, uses two **places** to store the state, one of which is always the parent **place** of the remote task.

The initial results as presented by the X10 team are based on a set of micro-benchmarks, representative of common patterns in X10 calculations and are performed on a 23 node AMD Linux cluster and on a 13 node IBM Power775 cluster. For executions with the resilient version without failures the place-zero implementation for tasks that initiate on place zero is reported to perform closely to the default X10 implementation, whereas for other parent places, the overhead increases by *one order of magnitude*. For micro-benchmarks where each place spawns 100 local tasks, the overhead of the resilient implementation increases by two orders of magnitude compared to regular X10. This is attributed to the added synchronous communication to Place 0. The distributed implementation is reportedly more scalable and faster; an overhead of more than *one order of magnitude* is introduced in the case without failures.

The X10 team also reports on an iterative sparse matrix dense vector multiply calculation, used in analytics applications. As part of the computation each matrix element is read from memory and a multiply-add instruction is performed. Figure 4.10 demonstrates the mean runtime of 30 iterations for variable input sizes for

regular X10, Resilient X10 without failures and Resilient X10 with 1 Dead Place, while they also provide results on a Hadoop implementation for comparison.

Size	Hadoop	X10	Res. X10	1 Dead Place
100K	3301	12	12	14
200K	3390	20	19	20
400K	3563	32	30	32
800K	5392	73	70	76
1M	6820	96	96	105
1.2M	8737	128	129	142
1.5M	12559	180	182	199
2M	21773	293	290	317
2.5M	33664	434	438	480
3M	52075	596	595	656

Figure 4.10: Runtimes in seconds of the sparse matrix dense vector multiply calculation over 23 places for the Resilient X10 implementation, as reported by Cunningham, Grove, Herta, et al., 2014.

We note that the X10 resilient version without failures produces small speedups for the variable input sizes, with the larger speedup of 6.45% for the input size of 400K elements. The authors do not analyse the possible causes of these speedups. These small speedups also appear in the next section in our evaluation discussion for resilient task-parallelism in Chapel and in the evaluation of the data-parallel implementation in Sections 5.4.2 and 5.4.3 of Chapter 5.

Throughout the results the resilient version’s runtime is comparable to the regular X10 version, while for the case with a single place failure, the overhead increases by an average of 7.62%. In the worst case, for the input of 100K elements the overhead increases to 15.38%. The authors also report on runtimes with 3 failures (Grove, Hamouda, Herta, et al., 2017) with an average runtime overhead of 3.26%. The latter result of lower overheads with a larger number of failures, also comes up in the evaluation of our mechanism.

The X10 team has not, to the best of our knowledge, published detailed results with multiple failures but one of their main conclusions is that the total number of failures and the time that failures occur in the execution, directly affect the cost of resilience support. In other words, they point out a direct link between performance and the number of recoveries required. Based on Resilient X10’s design, a parent place that initiates tasks on multiple children places, will be burdened with the

cost of recovery management as more children places suffer failures. The time that failures occur is another factor that affects performance; for example, a place failure that occurs just before completion of a long-running task, will introduce larger overheads, compared to an early failure.

Based on X10’s design principle that results from failed tasks are discarded and the tasks are scheduled for re-execution, we expect the time for recovery to be equal to the time required to successfully complete the task plus the time spent from the beginning of the task execution till the time of the failure. Though, the X10 team does not provide relevant empirical data to quantify the above costs, when we look only at the runtime of a single task, we need to factor-in the time till the detection of the failure, the time spend in the management of the internal data structures and the re-execution scheduling time.

The authors also report on results of the K-means algorithm, using the *decimation* technique. According to this technique, when a place failure occurs, the remaining state is used to continue with the execution. For a small number of independent failures the calculation produces equivalent results with the trade-off of accuracy, but the correctness of the results is affected when adjacent places fail. This part of the implementation is not comparable to our Chapel resilient implementation, firstly because checkpointing is required to preserve the state of the computation on the different places and secondly because failure recovery is not performed. It is clear, that the viability of this approach depends on the algorithm. Since, we aim for a general solution to resilience for Chapel, we follow a more conservative design approach, where we require that the accuracy of the results remains unchanged even in the presence of failures. As such we have excluded the reported results on decimation, from our discussion.

Considering that our implementation shares common design directives with both the Place-Zero Based Finish and the Distributed Finish of Resilient X10, we take into account the reported overheads. Based on the worst case overhead as reported by X10 for the K-means micro-benchmark and the overhead of one order of magnitude for both design approaches without failures, we consider a 30% overhead increase a reasonable threshold to set across our testing cases for Chapel’s resilient runtime implementation.

In the following sections (Sections 4.5.1 and 4.5.2) and in Chapter 5 we will report on performance results on a range of Chapel programs, including constructed micro-benchmarks, the STREAM triad benchmark (Section 5.4.2) and the N-body algorithm (Section 5.4.3). We discuss overheads using the resilient implementation without failures, to evaluate the initial costs of resilient support and results with single and multiple locale failures.

4.5 Evaluation of the Resilient Task Parallel Implementation

In this section, we present a performance evaluation of the runtime-level resilient implementation². We are concerned mainly with Chapel’s serial and task-parallel constructs and their combinations with remote task spawning language constructs. The discussion focuses primarily on the *overheads* introduced for the set of micro-benchmarks presented earlier in Section 4.4.1, with the use of the resilient mechanism. We analyse the results taking into account the different testing modes (Section 4.4.2) and execution parameters.

4.5.1 Blocking Fork

Functionality and Correctness As a first step, we demonstrate the functionality of the resilient serial remote task spawning (blocking fork) implementation for the programs of Section 4.4.1, looking into the correctness property of the program (i.e. the correctness of the results). This metric serves the purpose of a *sanity check* for the implementation and it is not meant to assess its quality. We use the *Monte Carlo Pi approximation* as the independent long-running computation, we execute 30 iterations with both testing modes; *all* and *rand*, and validate that the output matches the result of the non-failing computation. In the *all* testing mode, we inject failures on every locale in the execution, apart from Locale 0, while in the *rand* mode, failures are injected on a random number of pseudo-randomly selected locales in the execution.

²The datasets and the source code used in the evaluation sections of Chapter 4 and Chapter 5 are uploaded on Github: <https://github.com/konsP/ChapelTransparentResilience>.

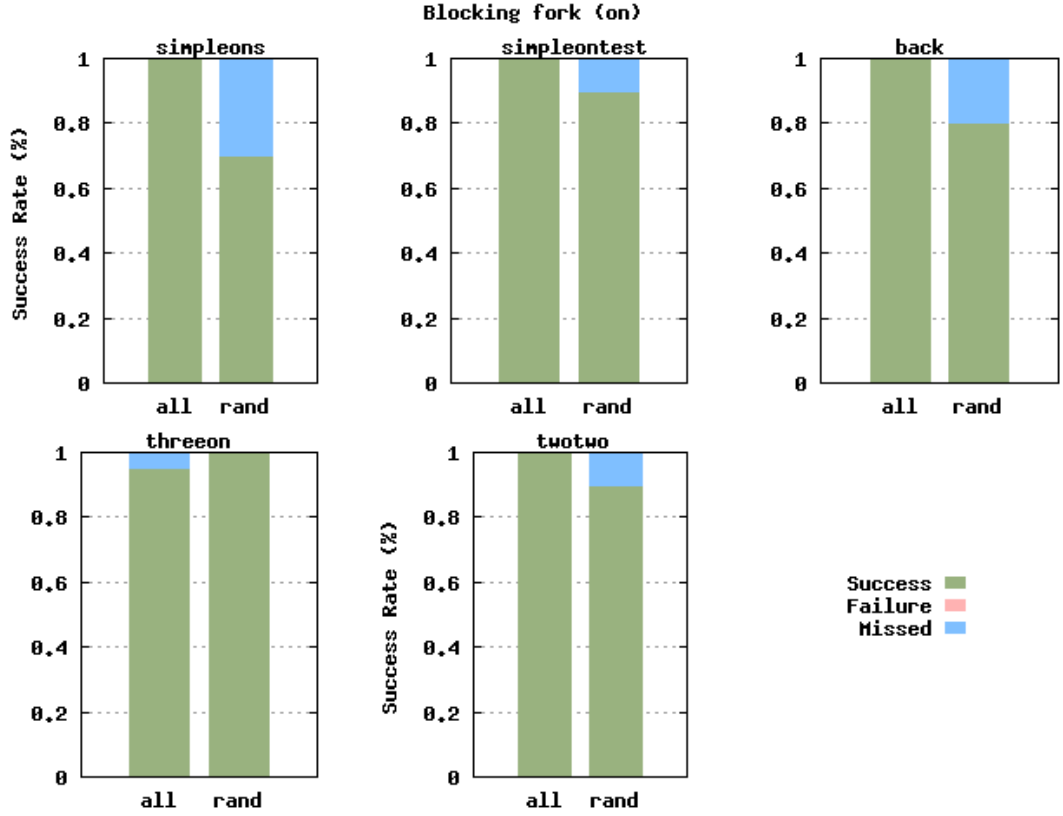


Figure 4.11: The plots demonstrate the success rate (%) of the failure recovery mechanism for the resilient blocking fork implementation. The data corresponds to the full set of measurements (30 iterations) performed per micro-benchmark and for both testing modes. The cases marked *Missed* indicate a failure on Locale 0 and lead to system-wide abort.

Figure 4.11 shows the success rates of failure recovery in five plots; the results correspond to each micro-benchmark and we present results for the *all* and *rand* testing modes respectively. In this plot we considered an execution successful (marked as *Success* in the figure) when all locale failures are handled and the program reaches completion with correct results. This requirement implies that the runtime successfully detects the locale failures and performs the recovery of tasks transparently with respect to the application domain.

According to our design of the resilient blocking fork, recovery is undertaken by the parent locale. In the *all* testing mode, recovery tasks are handled on the root locale, following failures on all other locales in the execution. In the *rand* testing mode, a failure of both the parent and the child locale will result in recovery on the immediate ancestor of the parent locale, assuming that this ancestor is alive, or one level higher on the locale tree. As more failures occur, recovery tasks are handled

further up towards the root locale, and in the worst case, the *rand* mode becomes equivalent to the *all* mode.

As shown in the figure, we obtain high success rates up to 100% on both testing modes confirming the correctness of the calculation in the presence of failures. The behaviour also demonstrates that the produced implementation is in accordance to our design, while the cases marked as *Missed* in Figure 4.11 are an immediate result of the limitations of the testing interface.

On one hand, Chapel’s multi-locale initialisation mechanism (driven by GASNet) dictates that locale identifiers are assigned to bootstrapped nodes transparently to the rest of the implementation and the application code. The connection between locale identifiers and node identifiers is not exposed to the programmer or the Chapel application code. On the other hand, the failure injection mechanism is based on Linux OS-level process identifiers where the Chapel computation is seen as a black box. As far as the failure injection mechanism is concerned, locales are OS-level processes. Though, in the common case the first node of the bootstrapped set is chosen to act as the root locale (Locale 0), such a convention is not explicitly stated in the GASNet specification – showing the developers’ intention to provide a general API for transparent cross-platform integration. Thus, in the test cases where the root locale is not the first node in the `SSH_SERVERS` list, the testing interface will (*all* mode) or may (*rand* mode) simulate failures on the root locale, resulting in system-wide abort. In order to comply to our assumption that the root locale is failure-free, such cases are excluded from the performance evaluation of our mechanism. This case remains currently a limitation of the testing interface (see also Section 6.3).

Finally, we observe the absence of logical failures in the results, for example locale failures that remain unhandled or task recoveries that occur on other than the intended adopting locale. This fact is evidence of the ability of the resilient implementation to tolerate locale failures and apply corrective action, within the above stated scope of limitations. Furthermore, all failures are handled on the expected adopting locale, in compliance with our design.

Runtime and Overhead In Figures 4.12 and 4.13 , we demonstrate the runtime (measured in seconds) and the respective runtime overheads introduced by our

resilient blocking fork implementation for each of the micro-benchmarks. We use Chapel version 1.9 as the baseline and we observe small overheads ranging between 0.29% and 1.29%, compared to the regular Chapel implementation, when no locale failures are introduced (red bar). We attribute these overheads to the added communication, the management of the internal data structures, and the serial lookup operations for status checks, required to support resilience. The results exhibit that an application with remote task spawning that executes on a reliable system without failures, will experience small performance penalties by due to the use of our resilient mechanism.

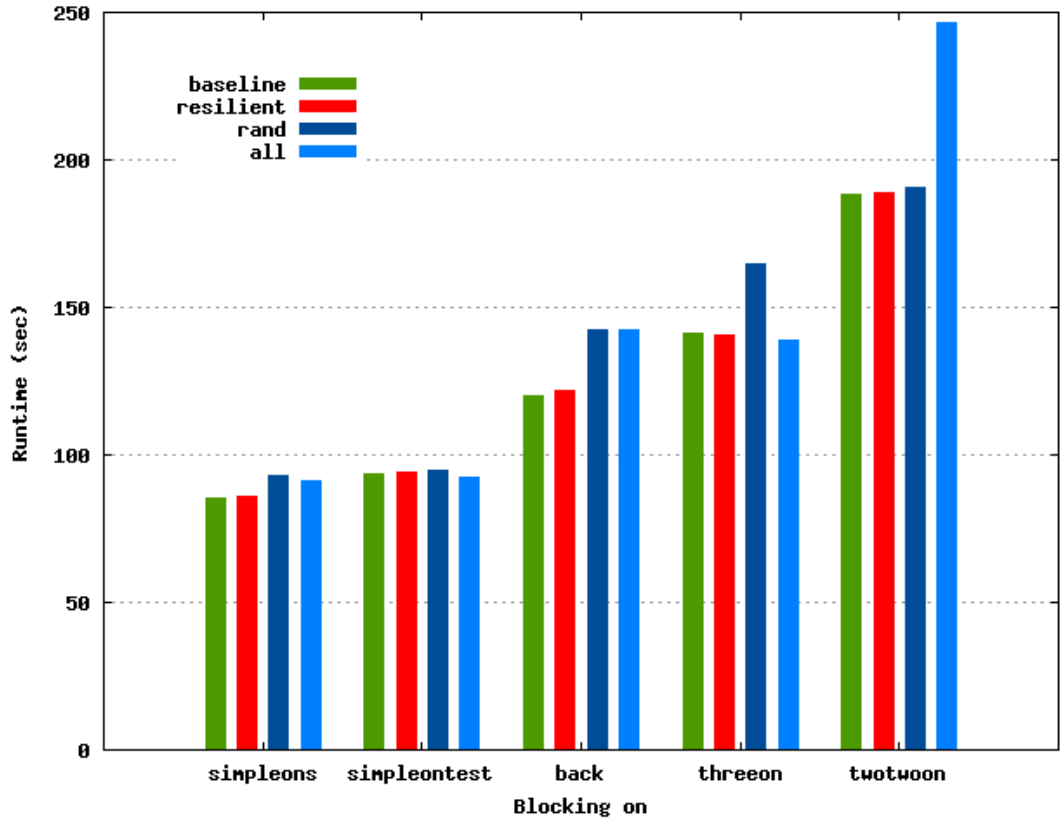


Figure 4.12: Runtime (in seconds) of the micro-benchmarks using the resilient blocking fork implementation. The plot demonstrates the runtimes of regular Chapel, resilient Chapel without failures, and the runtimes when using the `all` and `rand` testing configurations.

The next two bars in the plot of Figure 4.13 represent the two testing modes, `rand` and `all`. The overhead rates on failure depend highly on the structure of the *LocaleTree* in each micro-benchmark, especially with respect to shallow and deep nesting. We make the implicit assumption that the independent Monte Carlo Pi calculations perform similarly on every locale. For three out of the five test cases,

failure with the *all* mode performs better compared to failures of a random number of locales, as the recovery of the latter results in one or multiple *straggler* task(s) on the adopting locale(s).

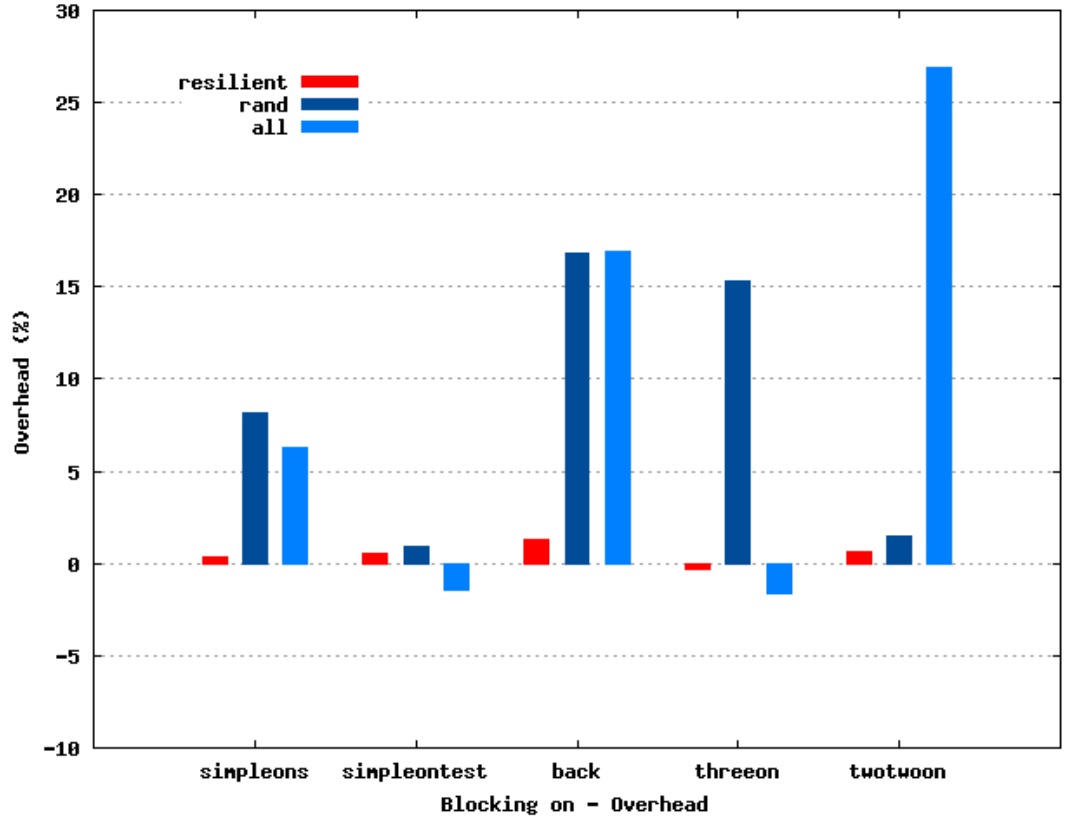


Figure 4.13: Overhead (%) of the five micro-benchmarks using the resilient blocking fork implementation. The graph demonstrates the runtimes of regular Chapel, resilient Chapel without failures, and the runtimes with the *all* and *rand* testing configurations.

In detail, for *simpleons* the mean overhead across executions in the *all* testing mode introduces an overhead of 6.25% compared to the 8.15% of the *rand* mode. When both Locales 1 and 2 fail, the root locale is immediately notified and initiates the recovery process for the two independent tasks. The *back* micro-benchmark performs comparably in both test modes (16.79% and 16.87% respectively) due to the reuse of Locale 1 in the code. For *simpleontest* and *threeon*, we note small speedups of 1.45% and 1.58%. Similarly to the case of *simpleons*, and due to the nesting of tasks the root locale does not establish costly communication to Locale 1 and instead handles *local recoveries* immediately. As more locales fail, the recovery rolls back towards Locale 0; due to the lack of communication costs, the replacement of status checks by the local array lookup and the increased task granularity on

Locale 0, we demonstrate increased performance.

Apart from cases where both the parent and the child locale fail, when multiple failures occur on independent locales, the recovery tends to be balanced, without straggler tasks. Furthermore, recovery tasks that execute on the root locale tend to be cheaper when detected early in the execution. As the root locale is up to date with the failures that occur across the system, a failure of an immediate child locale allows for early detection. It is hence possible to skip the communication phase (to the failed child) and perform *local recovery*. Due to the structure of our micro-benchmarks, the root locale does not perform any local computation, thus recovery is cheaper than recoveries on other locales.

For the *threeon* benchmark we note a higher overhead (15.27%) for the *rand* mode. Due to the structure of this benchmark, with the three nested tasks, we can get unbalanced execution patterns, depending on the locales that fail each time. In the worst case the first failure occurs on Locale 2, followed by task adoption on Locale 1 and a subsequent failure on Locale 3. As a result, Locale 1 is burdened with the execution of two recovery tasks on top of its local workload.

For the *twotwoon* benchmark, there are two nested tasks executing on the same inner locale. The high overhead of the *all* mode shows that communication between parent/child pairs is established before any failures are detected. It is also worth noting the case where the order of failures is {Locale 2, Locale 1, Locale 3}. When Locale 2 fails, both Locales 1 and 3 begin the recovery process. After their failure, the root locale will adopt all the tasks, accumulating the costs of the previous recoveries. Eventually, all four tasks will execute on the root locale, leading to the increased overhead of 26.86%. The mean overhead is thus highly affected by scenarios as the above, but remains within the 30% threshold, as defined in Section 4.4.4.

4.5.2 Non-blocking Fork

In this section we focus on Chapel’s non-blocking remote task spawning mechanism. Programmatically, non-blocking distributed Chapel programs use combinations of `begin`, `cobegin` and `coforall` with the `on` construct, to achieve task migration and asynchronous execution. We re-use the micro-benchmarks of Section 4.4.1 and apply the above construct combinations to produce parallel versions. The complete

micro-benchmarks can be found in the Appendix (Section B.1).

For the resilient non-blocking fork implementation we focus on results with the injection of a single failure. This experimental setup has been chosen instead of the *all* and *rand* testing modes in order to more clearly demonstrate the effects of a failure in a non-blocking environment. In the discussion we do not report on the cases with failure of the root locale, both due to the critical role of Locale 0 –dictated by Chapel’s design– and also in accordance to our *failure-free root* assumption. As per our conclusions from the blocking fork prototype and the evaluation results of Resilient X10, later in this section we discuss how the placement of failures affects the performance of the execution.

begin We demonstrate the mean, maximum and minimum runtime of (i) the baseline implementation, (ii) the resilient version without failures and (iii) the resilient version with a single failure, for the combination of **begin** and **on** constructs over 20 iterations. Each individual task in the micro-benchmarks is launched in a non-blocking fashion. The runtime system is configured to use the two subsequent locales in the configuration as buddy locales in a circular order. For example the buddies of Locale1 are Locales 2 and 3, while the buddies of Locale 3 are Locales 0 and 1. We focus mainly on the mean overheads, demonstrated in Figure 4.14, but we present the execution runtimes in Figure 4.15 for completeness.

The error bars mark the minimum and maximum runtime that occurred across iterations for each test case. In the case of overhead graphs the average, minimum and maximum are calculated with respect to the corresponding average, minimum and maximum runtimes of the baseline. For this reason the average overhead appears outside the error bar range. For example for the *simpleontest* micro-benchmark with 1 failure shown in Figure 4.14, the average overhead is close to 6% (comparing to the average runtime of the baseline) while the minimum overhead is 7% slower comparing to the minimum runtime of the baseline.

For the first micro-benchmark (*simpleons*) the average overhead of the resilient implementation in the case without failures is 0.7% compared to the baseline and an overhead of 1.34% is introduced in the case with a single failure. The results suggest that in programs with parallel tasks that execute on locales without inter-

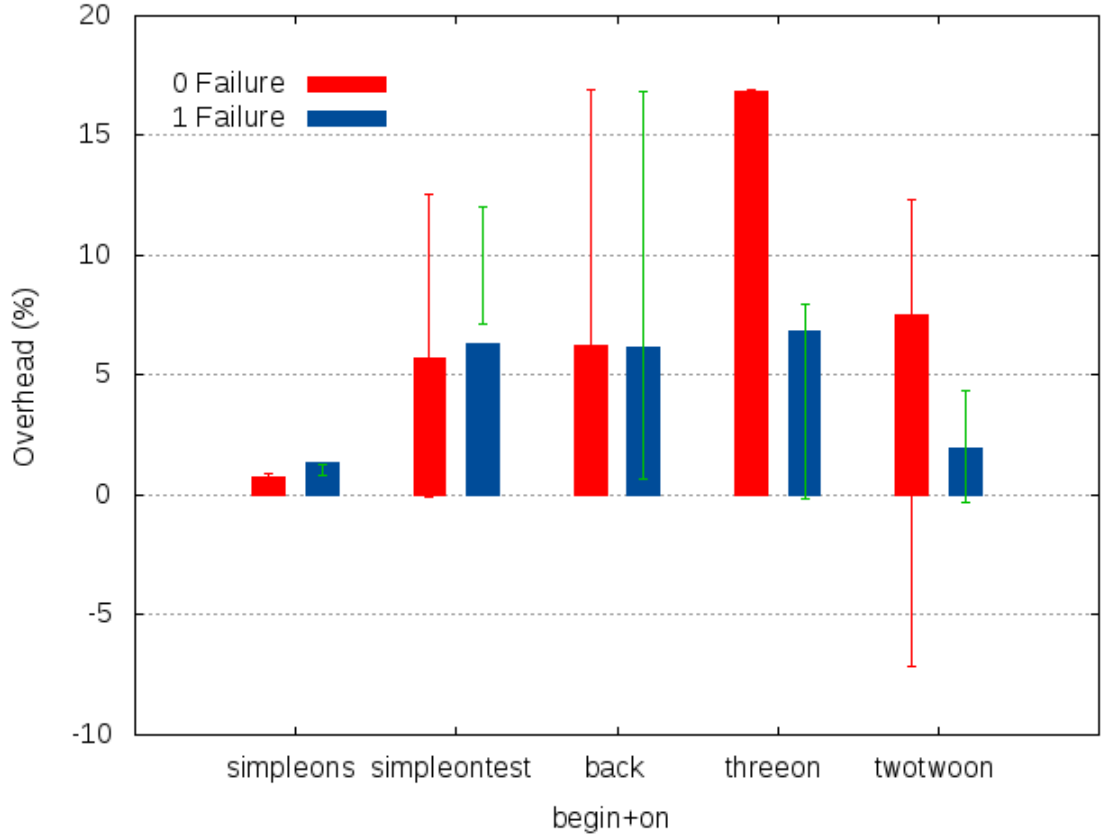


Figure 4.14: Overhead (%) of the resilient non-blocking fork for the combination of `begin+on`. The plot demonstrates the runtime mean overheads of the resilient implementation *without* failures and *with* a single failure, compared to the baseline Chapel implementation. The maximum and minimum values in each case are shown with error bars, and correspond to the percentage difference of the respective maximum, and minimum values of the baseline.

locale dependencies, recovery is cost-effective and the use of the resilient version when no failures occur achieves comparable performance to the baseline Chapel implementation, a result that agrees to the figures reported for Resilient X10.

In the case of the *simpleontest* benchmark, where the two tasks are nested we note average overheads of 5.7% and 6.27% for the case without failure and with a single failure, respectively. In the next two micro-benchmarks we launch three nested tasks, thus increasing the number of parent/child pairs and the inter-locale communication. For *back*, the average case with failure remains close to 6.18%, achieving comparable performance to the failure-free case.

For *threeon*, the average overhead with failures accounts for a runtime increase of 6.83%, while the minimum runtime with recovery is close to the baseline’s average runtime. When no failures occur, we note larger overheads (16.9%) with negligible

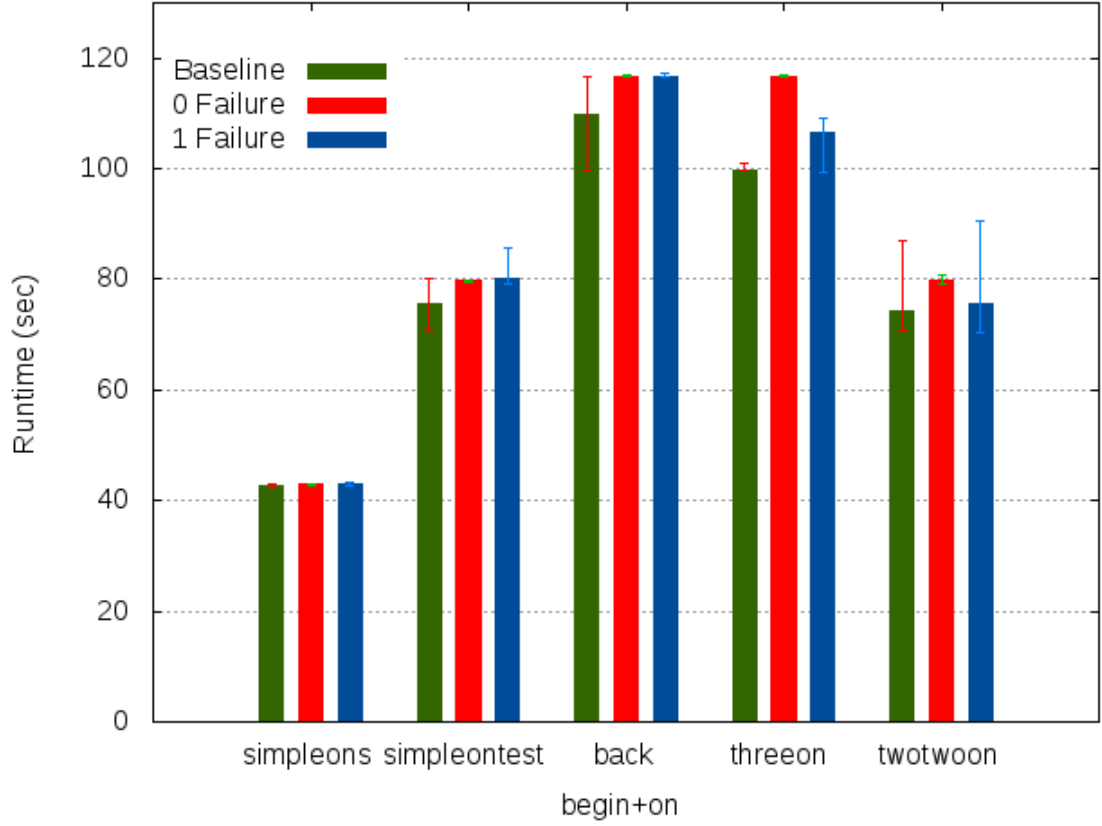


Figure 4.15: Runtimes (in seconds) of the resilient non-blocking fork for the combination of `begin+on`. The plot demonstrates the runtime average results for the baseline (regular Chapel) implementation, the resilient implementation without and with a single failure. The maximum and minimum values in each case are shown with error bars.

divergence between maximum and minimum runtimes. The overheads in this case approach the maximum overhead of the *back* micro-benchmark in the failure-free execution.

Both programs showcase a triple-nesting pattern of tasks and locales, though in *back* there is a locale dependency between locales 1 and 2. In the failure-free case, the inter-dependency of locales in *back* alleviates some of the communication costs, regarding the status checks –Locale 2, as the primary buddy, has information on the status of Locale 1. In the *threeon* micro-benchmark, the liveness check needs to be performed for each non-blocking task, since the task nesting follows the opposite order of the buddies structure. This means that none of the parent locales possesses liveness information on the child locale, since the children are primary or secondary buddies of the parent, a fact that is also suggested by the low (close to 1%) divergence of the maximum and minimum overheads, when no failures occur.

Thus, the communication established to obtain liveness information before each of the four tasks is launched, contributes to the larger overhead.

When comparing the two micro-benchmarks in terms of runtime (Figure 4.15) in the single failure case, we notice different execution behaviour. For the *back* micro-benchmark, as Locale 1 appears twice in the program and according to the *next neighbour* recovery pattern, there is limited variability between the minimum and the maximum overhead. After Locale 1 fails, Locale 2 is burdened with the recovery of two tasks on top of the local workload, less communication is required and task granularity is increased. Similarly, when Locale 2 fails, the recovery is handled immediately on Locale 0, where there is no local workload. The runtime results show improved performance of the *threeon* benchmark as there are only two failure scenarios. Firstly, failures that occur on Locales 1 or 2 which will increase task granularity on the buddy locales (Locales 2 or 3 respectively) and secondly a failure on Locale 3 which is handled on the root locale.

Finally, for the last micro-benchmark (*twotwoon*) the average overhead in the case with failure is 1.98% and in the failure-free case it reaches 7.55%. The results of the failure-free case show high variability, with the minimum case demonstrating a 7% speedup. According small speedups are also reported in Resilient X10's results.

In an effort to understand how the runtime is affected by the failures on the different locales we use Figure 4.16 to showcase the runtime of the execution for the separate failures on each locale. We note small divergence for the first three micro-benchmarks (*simpleons*, *simpleontest* and *back*) in the average case, showing that the cost of failure on any of the two participating locales is comparable. In the case of *threeon*, failure on any of the three participating locales introduces smaller overheads compared to the failure-free case, as demonstrated in Figure 4.16(d). Here, each locale failure is handled on the subsequent locale, which results in decreased inter-locale communication, since no new communication originates from the failed locale, after the failure notification.

In Figure 4.16(e), we note that the larger overheads occur in the case of failure on Locale 2. This outcome is expected as Locale 2 appears twice in the program for both the inner nested tasks, so the buddy locale's workload (in this case, Locale 3) triples, thus increasing the total runtime. The overhead compared to the failure-free

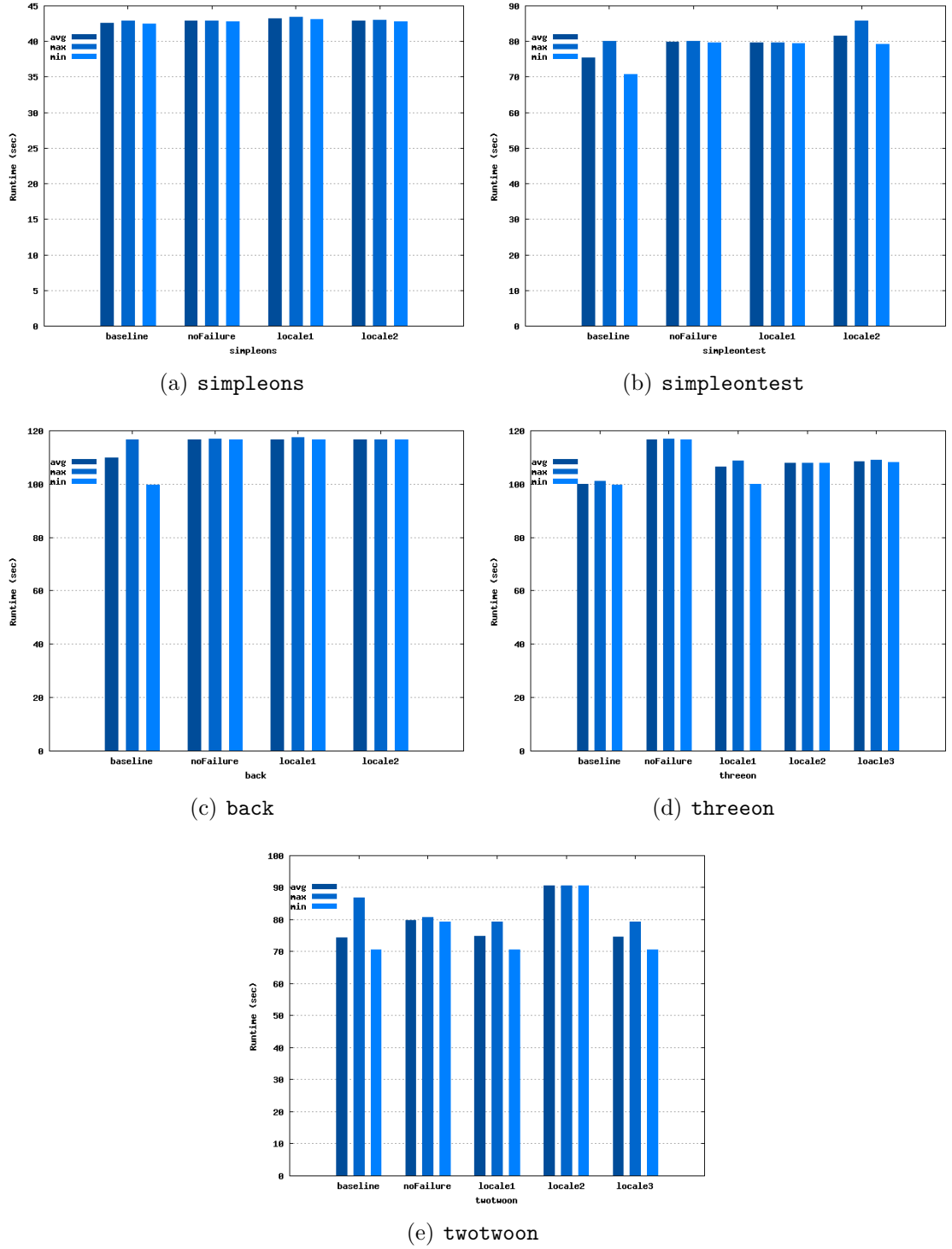


Figure 4.16: The plots demonstrate the average, maximum and minimum runtimes of the micro-benchmarks, as obtained by the previous experimental configuration, with respect to the location of a single failure on the locales on the x-axis. Runtimes (in seconds) of constructed programs for (i) the baseline (regular Chapel 1.18), (ii) resilient Chapel without failures and (iii-v) resilient Chapel with failure on the participating locales.

case is increased by 11.7%.

Finally we point out that for the non-blocking tasks created by the combination of `begin` and `on` the mean overhead across test cases with a single failure is at most 6.83% and occurs in the case of triple nesting. For the failure-free case, the mean overhead is 16.9%, which occurs, as before, in the case of triple nesting, with minimal divergence between minimum and maximum values.

cobegin In this section, we discuss the runtimes of our `cobegin+on` micro-benchmarks. The main point that differentiates the `cobegin` construct compared to the case of `begin` of the previous paragraph, is that `cobegin` is designed with the *producer/-consumer* pattern in mind. The produced parallelism matches closely OpenMP’s execution model (demonstrated in Figure 4.17) with a parallel region of tasks and a synchronisation point to join threads (or tasks in Chapel’s context).

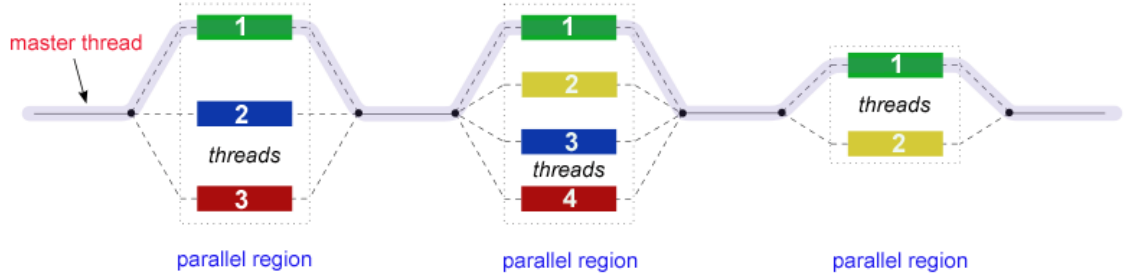


Figure 4.17: Fork-join execution model as implemented by OpenMP (Barney, 2017).

According to Chapel’s documentation, programs that contain less than three tasks in the `cobegin` block, are optimised by the compiler to execute serially. For the micro-benchmarks with nested tasks we have used `cobegin` constructs on the inner nested levels, to force the creation of multiple parallel tasks. For example, in Listing 4.8 we demonstrate the final version of the `threeon` micro-benchmark following the above refactoring. Another example is the case of triple-nested tasks, where we use one `cobegin` in the scope of each locale on top of the surrounding `cobegin`. For some of the micro-benchmarks we have also used `begin` constructs to enforce the non-blocking execution of tasks, so for this set of experiments we expect higher overheads for the micro-benchmarks that create more non-blocking tasks.

```

1 on Locales[1] do cobegin{
2   montCarlo();
3   on Locales[2] do cobegin{
4     montCarlo();
5     on Locales[3] do {
6       montCarlo();
7     }
8   }
9 }

```

Listing 4.8: The `three_on` micro-benchmark using a `cobegin+on` combination on nested tasks.

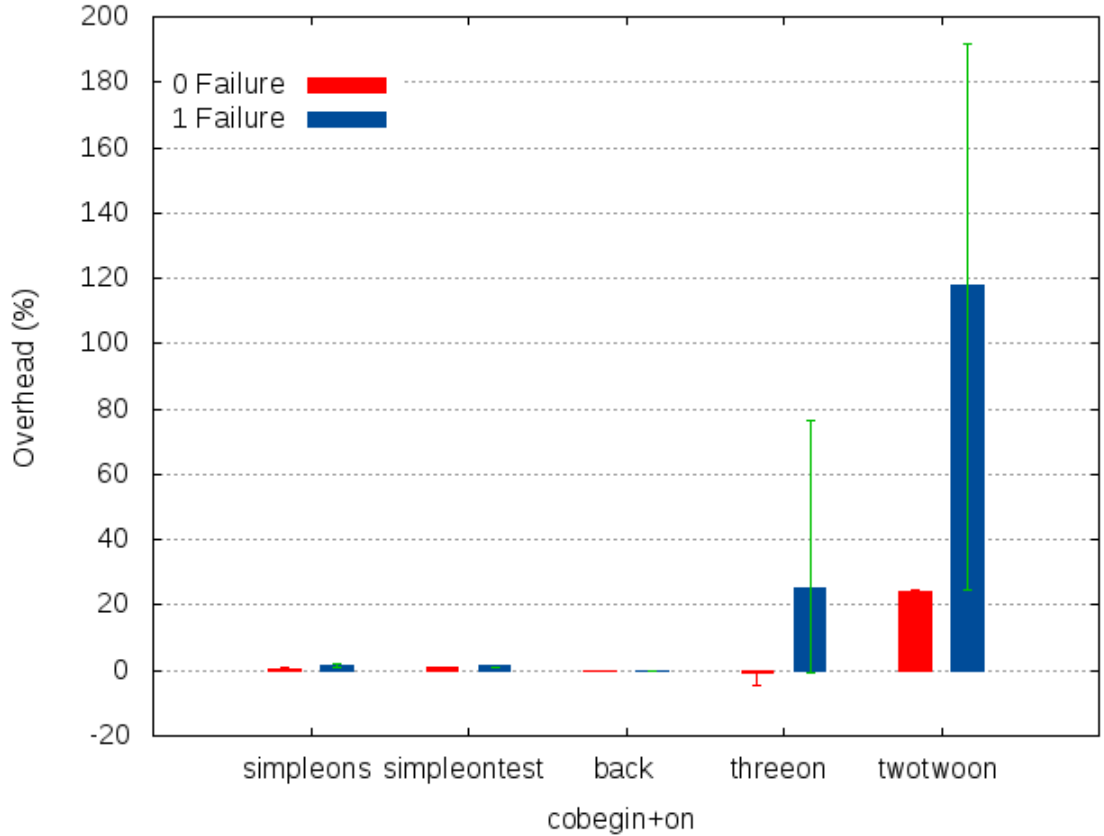


Figure 4.18: Overhead (%) of the resilient non-blocking fork for the combination of `cobegin+on`. The plot demonstrates the runtime average overheads of the resilient implementation without and with a single failure, compared to the baseline Chapel implementation. The maximum and minimum values in each case are shown with error bars, and show the percentage difference to the respective maximum and minimum values of the baseline.

Figure 4.18 demonstrates the runtime overheads of our micro-benchmarks with the construct combination of `cobegin` and `on`, while Figure 4.19 demonstrates the absolute runtimes in seconds.

We observe that the failure-free and single failure cases perform comparably;

indeed small overheads, of up to 1%, are introduced for the first three micro-benchmarks, *simpleons*, *simpleontest* and *back* in the average case.

For the micro-benchmark with three nested tasks (*threeon*) the average overhead increases significantly approaching 24.9% on average for the single failure case. The structure of *threeon*, as demonstrated in Listing 4.8, employs two `cobegin` blocks, which by design introduce two synchronisation points. When failures occur on locales that initiate a `cobegin` the recovery is bound by the implicit synchronisation. More specifically, in the case of *threeon* this translates to Locales 1 and 2. Our explanation is validated by the data of Figure 4.20(d), where the failures of Locales 1 and 2 show significantly high maximum runtimes; thus impacting the mean overhead of the entire micro-benchmark.

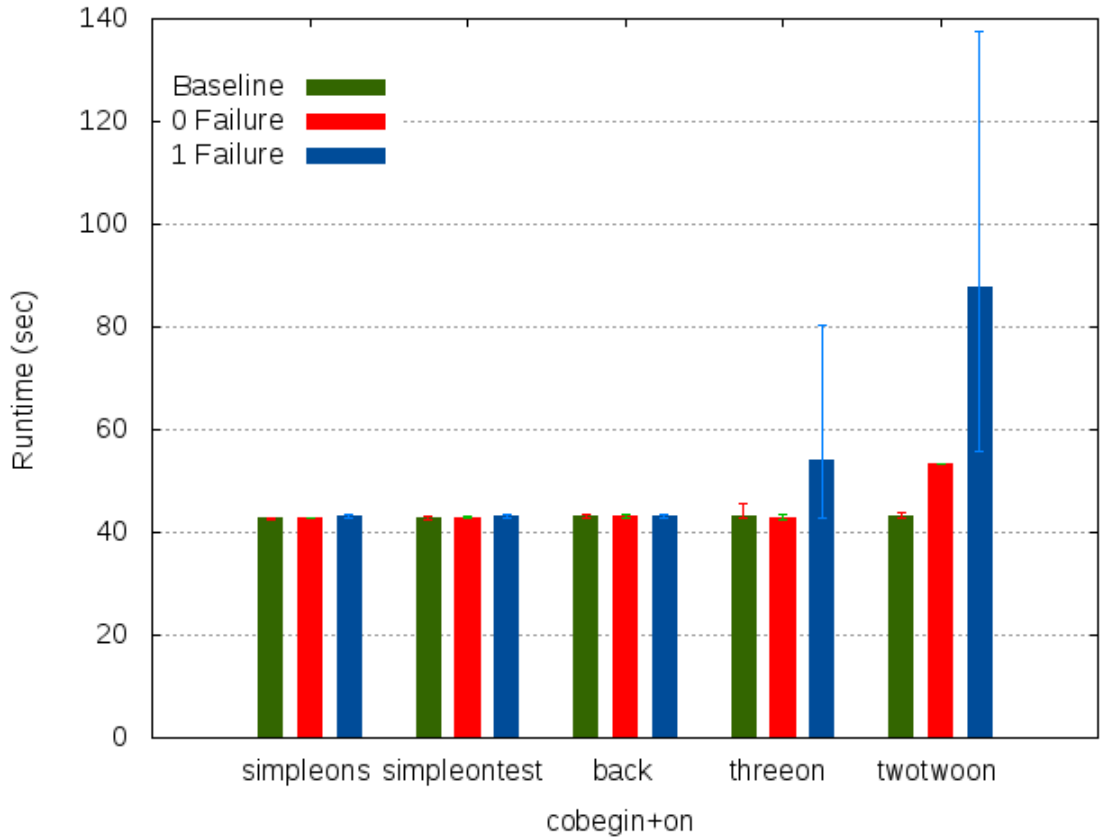
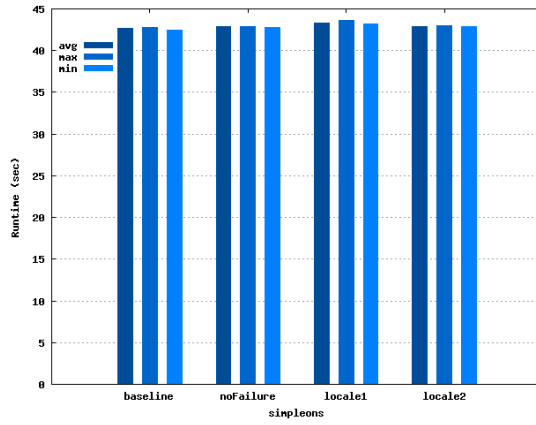
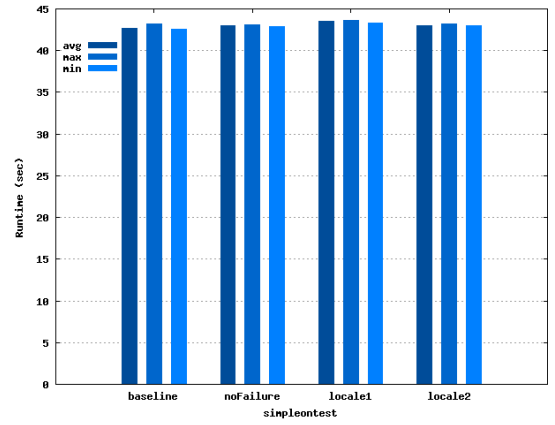


Figure 4.19: Runtimes (in seconds) of the resilient non-blocking fork for the combination of `cobegin+on`. The plot demonstrates the runtime average results for the baseline (regular Chapel) implementation, the resilient implementation without and with a single failure. The maximum and minimum values in each case are shown with error bars.

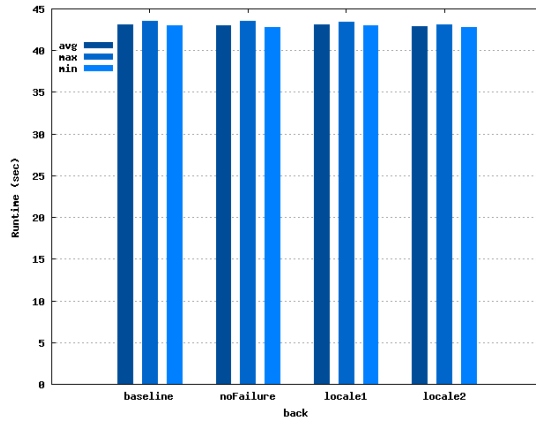
The micro-benchmark with two nested tasks (*twotwoon*) is the worst performing case in our setup for the *cobegin* and *on* combination, where in the average case



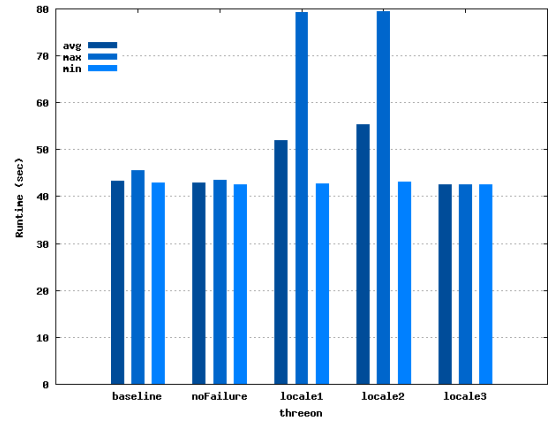
(a) simpleons



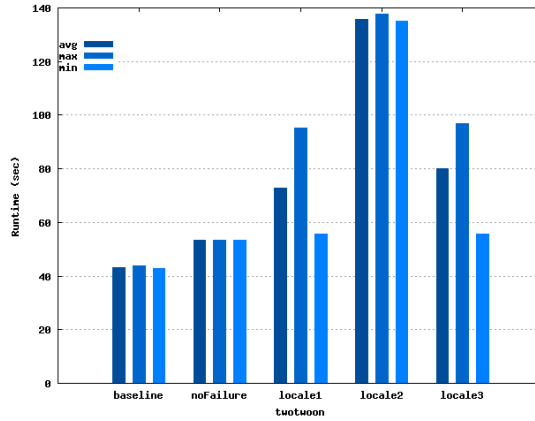
(b) simpleontest



(c) back



(d) threeon



(e) twotwoon

Figure 4.20: Non-Blocking 'On' : Failures on the participating locales for the combination of `cobegin+on`. Runtimes (in seconds) of constructed programs for (i) the baseline (regular Chapel 1.12), (ii) resilient Chapel without failures and (iii-v) resilient Chapel with failure on the participating locales.

without failure the overhead is close to 24%. In the case with a single failure the runtime doubles compared to the baseline, with a 120% overhead. The maximum runtime occurs, as expected, in the case of failure on Locale 2, as this locale is reused in the execution of the inner nested tasks. Thus, the computational workload triples for Locale 3, while Locale 3 is also bound by the synchronisation point of the `cobegin`, turning it into a *straggler* for the application.

As the last micro-benchmark performs significantly slower in the case with failure, we inspect Figure 4.20, showing the separate runtimes obtained by the failures of each locale. As confirmed by the plot of Figure 4.20(e), we demonstrate the increased runtime of the program in the cases of failure on Locale 2.

coforall For the combination of `coforall` and `on` the results are balanced across executions, showing negligible variability between *min* and *max* runtimes, as demonstrated in Figure 4.21 with the error bars. In the `coforall` case, tasks are independent and span up to 16 locales. For the case without failure, the overheads of the resilience mechanism remain fairly low, between 0.4% and 2.6%.

# messages	#2	#4	#8	#12	#16
coforall+on					
(i) baseline	6	18	42	66	90
(ii) 0 failures	6	20	48	76	104
(iii) 1 failure	7	21	49	77	105

Table 4.2: Number of messages exchanged among locales in the `coforall+on` execution, when executing on 16 locales.

We note slightly larger overheads in cases with failure, between 0.9% and 4.69% as shown in Figure 4.22. The latter occurs when using 4 locales, while when the maximum number of locales is used (16 locales) the overhead remains close to 3%.

In contrast to previous experiments, when using the `coforall` loop, Locale 0 participates in the execution by performing local computation, hence recovery costs are equivalent across locales. As demonstrated in Table 4.2, there is increased communication between locales with 90 messages exchanged for the baseline case, when the computation spans up to 16 locales. The communication is comprised of one message per buddy group for copying of the task descriptors, the initial bootstrapping of up to 16 locales and the fork/join operations. The case with failure produces

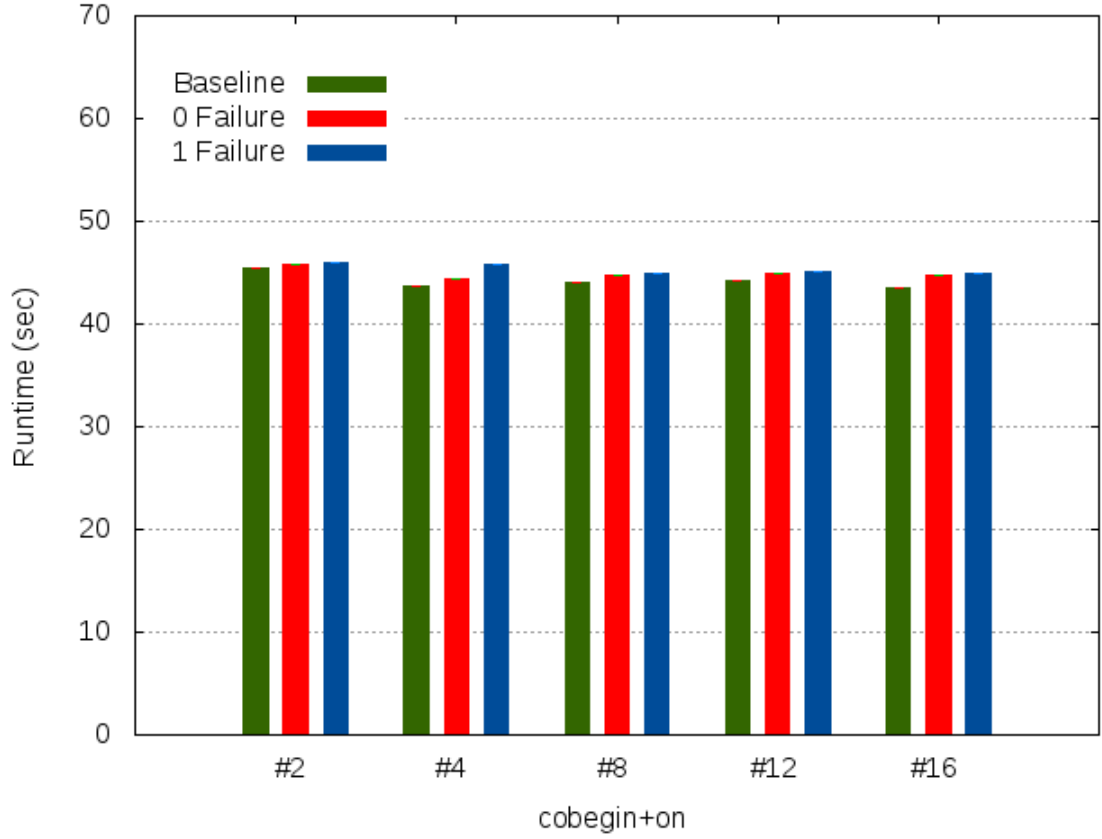


Figure 4.21: Runtimes (in seconds) of the resilient non-blocking fork for the combination of `cforall+on`. The x-axis represents the number of participating locales in each execution, while the y-axis represents the runtime in seconds. The plot demonstrates the runtime average results for the baseline (regular Chapel) implementation, the resilient implementation *without* failures and *with* a single failure for a distributed parallel loop executing the Monte Carlo Pi approximation algorithm on every locale. The maximum and minimum values in of each run are shown with error bars.

one extra communication operation compared to the failure-free case; the failure notification message. For the resilient version with a single failure, we can assume that the maximum performance penalty will occur in the case of task recovery on the slowest buddy locale.

Multiple failures In Figure 4.23 we showcase the runtimes of the combination of the `cforall` and `on` constructs, when run on 16 locales with an increasing number of failures (up to 6 failures). We use two buddies per locale in the configuration and the application can tolerate up to two failures within each group of buddies.

As shown in the plot of Figure 4.23, we obtain the maximum mean overhead of 25.9%, compared to the baseline version of Figure 4.21, when injecting four random

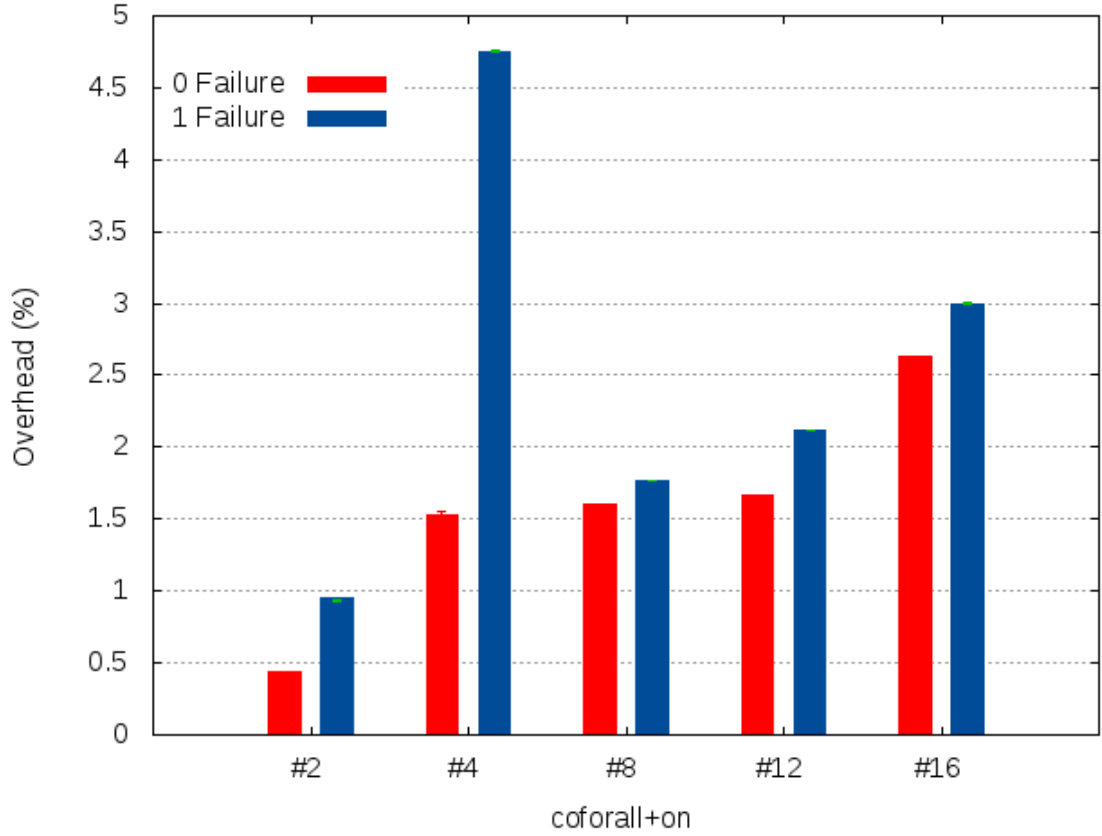


Figure 4.22: Overhead (%) of the resilient non-blocking fork for the combination of `coforall+on`. The plot demonstrates the runtime average, maximum and minimum overheads of the resilient implementation *without* failures and *with* a single failure, compared to the baseline Chapel implementation for a distributed parallel loop executing the Monte Carlo Pi approximation algorithm on every locale.

failures. We note, as expected, that although the overheads remain within the 30% threshold, as multiple failures occur the overheads tend to increase.

The results show high variability between minimum and maximum values. The fact that this variation is present in the case without failures and the comparison to the execution on 16 locales without failures and a single buddy per locale of Figure 4.21, leads to the conclusion that the high variability is introduced by the use of multiple buddies; in this case two buddies per locale. The high variability is also present in the next paragraph (Figure 4.24) where we increase the number of buddies. The communication and copying of remote data across multiple locations, appears to impact the worst case performance with a fixed penalty, close to 30% compared to the average case, when moving from single to multiple buddy configurations.

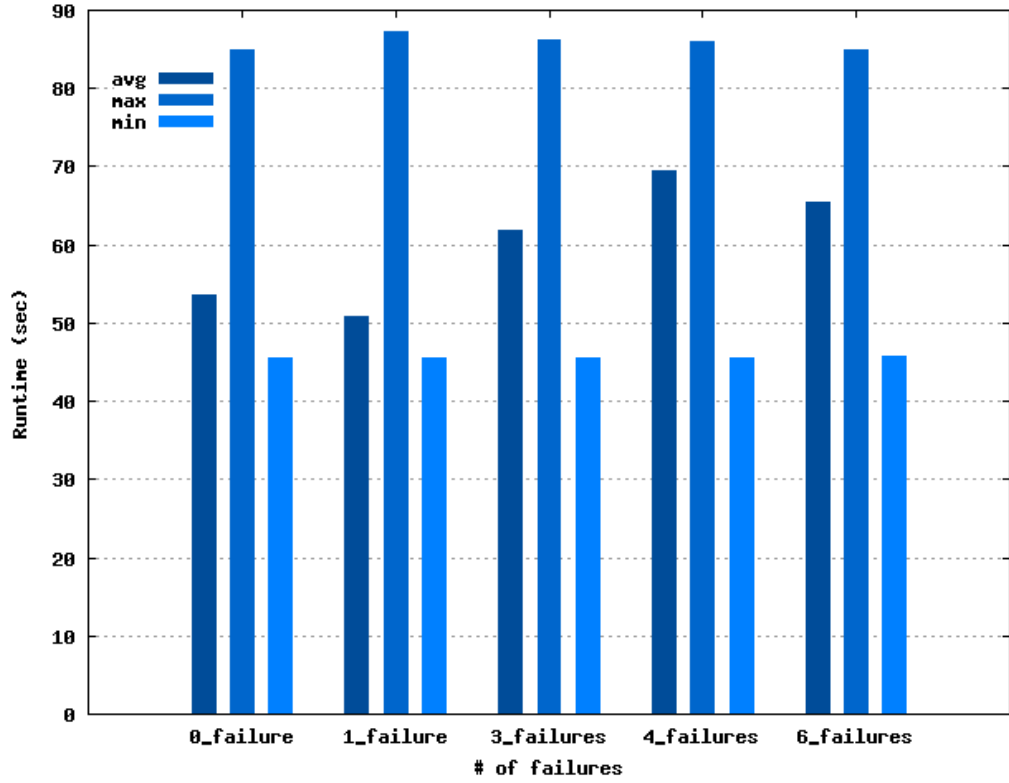


Figure 4.23: Runtimes (in seconds) for the `coforall+on` combination on 16 locales, with an increasing number of failures. The plot demonstrates the average, maximum and minimum runtime for executions with 0 to 6 failures.

Multiple buddies In Figure 4.24 we demonstrate the case of `coforall` and `on` constructs executing on 16 locales without failures and with varying numbers of buddies per locale. The results show that the number of buddies in the configuration does not impact the runtime with added overheads in the cases without failure, despite the management of larger internal data structures and the added communication for the exchange of in-transit messages, as the processing of the additional messages does not alter the per locale execution flow.

From the above experiments we conclude that when using a `coforall` loop to launch independent fine-grained tasks on multiple locales the runtime does not increase linearly to the number of failures; though the placement of failures can introduce high variability. Furthermore, in an execution without failures, larger numbers of buddies per locale do not introduce performance overheads.

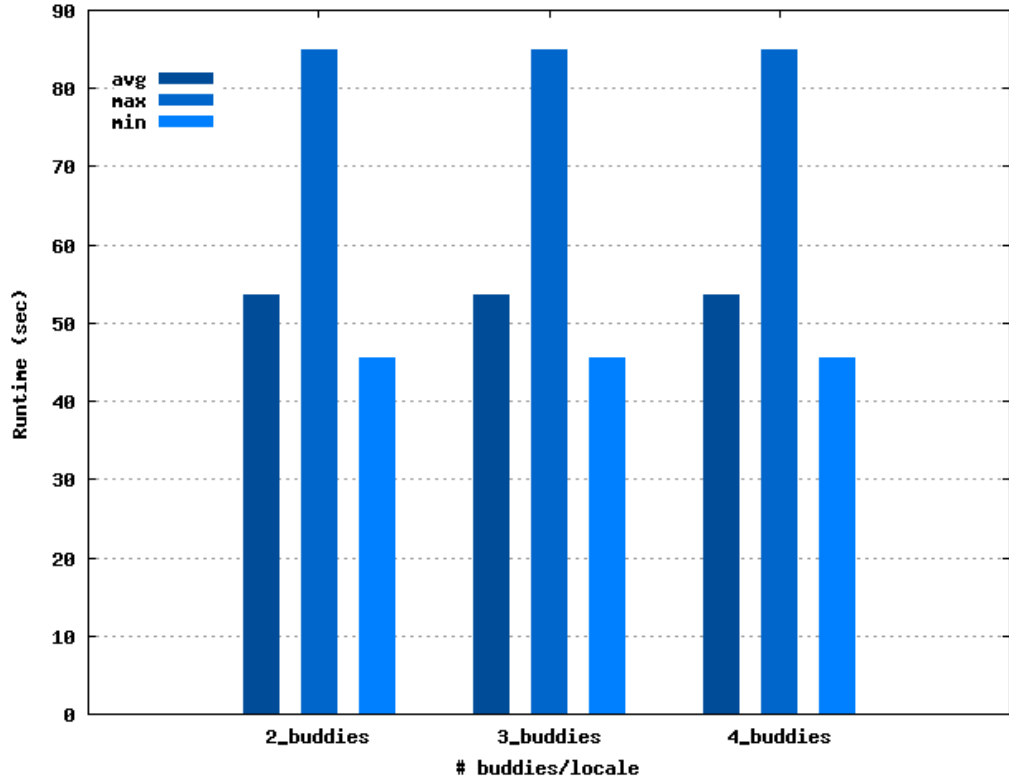


Figure 4.24: Runtimes (in seconds) for the `coforall+on` combination on 16 locales, with an increasing number of buddies. The plot demonstrates the average, maximum and minimum runtime for executions without failures, when using 2 to 4 buddies per locale.

4.6 Summary

We have presented the design and implementation of our transparent resilience mechanism for Chapel’s remote spawning task-parallel constructs. We have detailed the focal points of our design; the *internal data redundancy scheme* and the *buddy locales* mechanism and we have presented the required implementation within Chapel’s runtime system to support the two mechanisms. All aspects of the implementation including the detection and recovery mechanism remain transparent to the programmer. The resulting version of the runtime allows Chapel programs to execute till completion, despite the occurrence of failures on one or multiple participating locales. We have also discussed the resilient mechanism for serial remote task execution as the prototype implementation.

We have provided an evaluation of our mechanism primarily with respect to the overheads introduced by the implementation of resilient support. We presented evaluation results for five synthetic micro-benchmarks, representative of patterns of

nested task-parallelism in Chapel applications. Our performance results show low overheads, with a maximum of 25% performance decrease, on a 32-node Beowulf distributed-memory architecture. when using the resilient Chapel version without failures, which demonstrates that our mechanism can be used on reliable systems without introducing prohibitive performance penalties.

For the cases with failures, we note that the overheads of the different construct combinations vary, depending on the structure and nesting depth of the micro-benchmarks in use. For 80% of the testing cases the average overheads remain below 7%, while we also demonstrated two cases with larger overheads when performing experiments with the task-parallel `cobegin` construct, with implicit synchronisation.

As a measure of comparison we report on published overhead measurements for Resilient X10 using the Place-Zero resilient store. In Kawachiya, 2014, the authors report on the performance penalties of using the resilient (compared to base X10) version *without failures* on eight places. For MontePi; the calculation of π using the Monte Carlo algorithm, with two remote tasks spawned per place, the overhead of Resilient X10 is 2.2 %. For the KMeans benchmark, Resilient X10 introduces a 9% overhead, while for the Heat Transfer benchmark, which includes a stencil computation with frequent creation of remote tasks, the authors report on a 6.5-fold slowdown. While an one-to-one comparison is not possible, our *cobegin on* micro-benchmark uses multiple remote tasks and synchronisation points, similarly to the smaller benchmarks reported for Resilient X10. With the exception of the *twotwoon* micro-benchmark, we demonstrate negligible mean overheads of up to 1% for the case without failures when executing on four locales, compared to the 2% best result reported for MontePi and 9% of KMeans in X10. In the case of deeper nesting of remote tasks with dependencies among locales (*twotwoon*) the overhead increases up to 24% but remains lower than the 6.5-fold slowdown of HeatTransfer in Resilient X10.

The evaluation results presented in this section are representative of usages of Chapel’s task parallel constructs within other applications. Some of the micro-benchmarks; particularly *three_on* and *two.two_on*, serve as stress tests for deep nesting of tasks, since in Chapel programs one would rarely find deeper than two-level nesting patterns. We are at this point, unable to provide a formula to statically

calculate the produced overheads, as these primitive tasking constructs can be used in larger more complex applications. The existence of multiple synchronisation points, the scheduling of tasks as it arises by a certain algorithm, the frequent writes to memory, are only some of the factors that can impact application performance. The evaluation of the micro-benchmarks explicitly looks into the specified patterns of task parallelism. To that end, one can obtain an indication of overhead for the task-parallel sections, but it is left to the application programmer to obtain an estimate of the overheads of the entire application (when resilience is enabled) via testing.

As explained in Section 3.4.3, we have provided a single implementation of resilience to cover Chapel’s non-blocking execution model, irrespectively of the construct used to introduce task parallelism (`begin`, `cobegin` or `coforall`). The different runtime behaviour is expected as the internal design of each construct differs by default in baseline Chapel. In this work we take a holistic approach to resilience, using the constructs as the test cases, rather than as tools to design custom resilient versions. We currently consider an implementation of resilient constructs outside the scope of this work, and instead we focus on the runtime-based approach.

To the best of our knowledge, this is the first attempt to support transparent resilience for task parallelism within Chapel’s runtime system. While our performance results have been obtained from a set of micro-benchmarks, the implementation is general enough to apply to uses of task parallelism within larger Chapel applications. In the next chapter (Chapter 5), we will discuss two more substantial benchmarks, as part of our performance evaluation of the resilient data-parallel implementation for the Block data distribution.

Chapter 5

Resilient Data Parallelism

In this chapter, we discuss the implementation of resilience in the context of Chapel’s data-parallel support. Reflecting Chapel’s design, our resilient design focuses on data distributions and the parallel forall loop; both core components of data parallel support in Chapel and in other PGAS languages (X10, UPC). Chapel’s data distributions are central to data parallelism, as discussed earlier in Section 3.6.3, while *forall* loops are based on the concept of parallel iterators (Section 3.5.1).

To demonstrate our approach to resilience we focus on one of Chapel’s built-in data distributions; the *Blocked distribution* (**BlockDist**), and detail the required design modifications to support resilience. The implementation builds on our resilient runtime support, as presented in Chapter 4, with the modifications that support failure discovery, and expands on library-level module, in the internal design of the Blocked distribution. The implementation targets two focal points; the *distribution and management of redundant data* across locales (Section 5.2) and the *tuning of parallel iterators* to support adoption and recovery (Section 5.3) building on the *buddy locales* mechanism.

5.1 The Block Distribution

Earlier in Section 3.6.3, we discussed a simple Chapel program of initialisation and parallel iteration over a block distributed two-dimensional array.

When executing on six target locales the program of Listing 5.1 will produce the output shown in the array (left-hand side). The iteration space is distributed

```

use BlockDist;

const Space = {1..8, 1..8};
const A: domain(2) dmapped
  Block(boundingBox=Space) = Space;
var blockArray: [A] int;
forall a in blockArray do
  a = a.locale.id;

```

0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
2	2	2	2	3	3	3	3
2	2	2	2	3	3	3	3
2	2	2	2	3	3	3	3
4	4	4	4	5	5	5	5
4	4	4	4	5	5	5	5

Listing 5.1: The listing demonstrates a snippet of a distributed rectangular two-dimensional array declaration mapped with the block distribution. The array is initialized using a parallel forall loop, and the value of the *owning* locale identifier is assigned in each array cell. On the right side, we provide a sample mapping of the index space over 6 locales using Chapel’s block distribution. The number of locales is specified using the `numLocales` or `nl` parameter on program execution. When executing on a distributed system the number of locales used in the program must match or be smaller than the number of hosts in the configuration.

across the available locales (right-hand side) in a blocked manner. The distribution captures all the required data to execute (and recover) the distributed tasks, while global variables are stored on the root locale and are accessible via global references from other locales. Where provided, the size of the *bounding box* is taken into account for the calculation of the block size, while by default the blocks are computed to span across all the available locales, aiming for equal -or closely matching- block sizes.

Formally, the formula to compute the index `locIdx` of the locale on which a domain array index `idx` will be mapped, in the one-dimensional case, where the *boundingBox* defines a $\{low..high\}$ index range of the problem domain; and *targetLocales* $[0..N - 1]$ *locale* define the array of locales that participate in the execution, is shown in Table 5.1, below.

<code>idx</code>	<code>locIdx</code>
$low \leq idx \leq high$	$\text{floor}((idx - low) * N / (high - low + 1))$
$idx < low$	0
$idx > high$	$N - 1$

Table 5.1: The index partitioning formula of the Block distribution. Each block is mapped to one of the locales in the `targetLocales` array. The `boundingBox` is the domain used as guide for the partitioning and defaults to the problem domain (Cray Inc, 2015b).

In the case of multidimensional arrays, both `idx` and `locIdx` are index tuples, so

the above formula is applied to each dimension. Domain indices outside the bounding box are mapped to the same locale as the nearest index within the bounding box.

Chapel distributions provide a set of methods to facilitate queries on the underlying distributed space such as the `localSubdomain` method which returns the index set on the current locale, when the sub-domain is represented as a single sub-domain and `.localeId`, which when called on a variable, returns the identifier of the locale on which the variable is stored.

5.1.1 Overview

The Block distribution adheres to the Object Oriented programming paradigm, using classes to define its internal structure. Parent classes are accessible via global references from any location in the program during execution, and build upon the concepts of domains and domain maps of Section 3.6.3. Local instances of the parent classes are automatically created across the participating locales to capture the local sub-domains and sub-arrays, as defined by the data distribution. The distributions maintain information on the above mentioned local instances, via auxiliary data structures; mainly internal arrays.

Internally, the block distribution uses forall loops for parallel iteration over domain indices, array elements and locale arrays. Locality constructs, especially the `on` construct, are used extensively for data placement and for the coordination of remote read and write operations. Finally, remote copying is also used extensively during the initialisation phase of the distribution.

5.1.2 Implementation Details

Base Chapel The Block distribution is comprised of six classes; the `Block` distribution class, the `BlockDom` domain class, the `BlockArr` array class and their per locale instances; `LocBlock`, `LocBlockDom` and `LocBlockArr`, respectively. When a parent class instance is created (distribution, domain, or array), a corresponding local class instance is created on each locale (`loc0` to `locN`) in the target locales. As demonstrated in Figure 5.1, each of the local classes is mapped by the parent class, using the auxiliary arrays; `locDist`, `locDoms` and `locArr`, respectively. In the

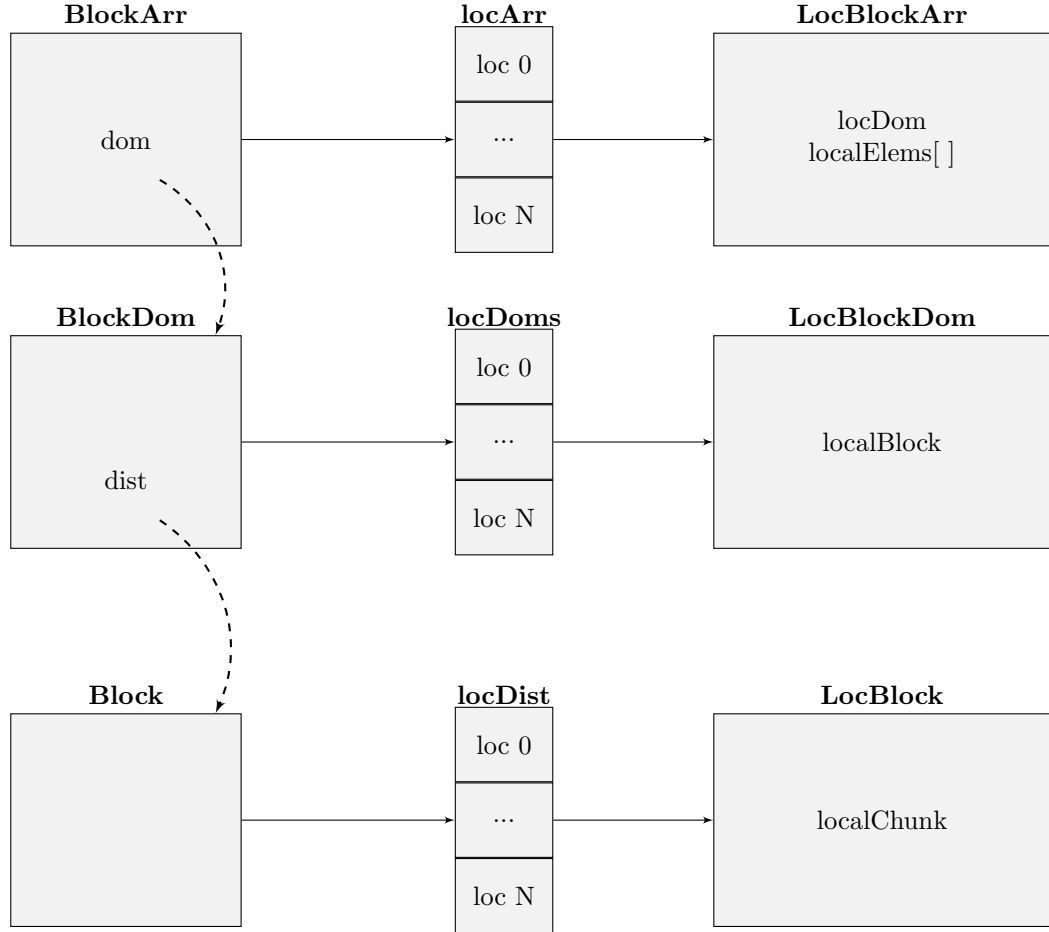


Figure 5.1: The layout of the internal classes of the blocked distribution, implemented within the *BlockDist* module.

following paragraphs, we detail the main components of the Block distribution and their internal functionality.

Block Distribution Class, Block: The *rank* of a block class defines the number of dimensions of the domain mapped by the distribution. Each Block class defines a *boundingBox* parameter; a local domain used to assist the partitioning of the index space across locales, and a local *targetLocDom* domain over which it defines the array of *targetLocales* and the array of local distribution classes. Finally, *locDist* is a local array of per locale distribution classes. Each of these classes points to the participating locale’s *LocBlock*, a per locale object representing the locally owned block.

The block distribution also defines a set of configuration flags to control the number of tasks used on each locale during parallel iteration: the *dataParTasksPer-*

Locale option limits the number of running tasks to the available parallel resources; the *dataParMinGranularity* option defines a minimum threshold of granularity per parallel task and the *dataParIgnoreRunningTasks* option controls the creation of new tasks.

Local Block Distribution Class, LocBlock: During the setup of the Block distribution a local *LocBlock* class is created on each target locale. LocBlock inherits the rank of the block distribution and defines a domain, representing the local set of indices; *localChunk*. For resilience, we extend this logic to redundantly persist the index sets of each locale on the buddy locales.

Block Domain Class, BlockDom: The *BlockDom* class inherits from *BaseRectangular*, the base class of all rectangular domains in Chapel. *BlockDom* describes the indices of the blocked domain. It maintains a reference to the parent Block class and specifies the layout of indices (normal or sparse). The BlockDom class maintains an auxiliary array (*locDoms*) of per locale domain classes and a domain (*whole*) that represents the complete index set of the distribution. Using the combination of local arrays, the system and the application code are able to query the location of a domain index, with respect to the index space, defined by the distribution over the target locales.

Local Block Domain Class, LocBlockDom: LocBlockDom represents the local part of BlockDom on each locale. It defines *localBlock*, a local rectangular domain, that locales may query to access their local index set.

Block Array Class, BlockArr: The class defines the elements of specified type of the distributed array. *dom* is a reference to the parent domain class and *myLocArr* is an optimized reference to the current location's (as specified by the **here** locality construct) array of elements. BlockArr also defines an array of per locale block array class instances, *LocBlockArr*.

Local Block Array Class, LocBlockArr: The local block array is the sub-array of the distributed array, and it is assigned to each of the locales in the execution.

Local element values are stored in the *localElems* local array, while there is also a reference from this class to the parent local domain class. This class represents the *remote* block of the array where element values are stored during array initialisation and are read and/or updated during application execution. For the resilient version, we support a new local array to persist the element values of the remote locales.

5.1.3 Leader-Follower Iterators

In Section 3.2 we introduced the concept of iterators as a construct of the base language. Iterators are the building blocks of data-parallel **forall** loops. They are essential to the internal functionality of distributions, but also a powerful user-level construct.

In the baseline implementation of Chapel, **zippered forall** loops, such as the one demonstrated in Listing 5.2, are based semantically on *leader-follower* iterators (Cray Inc, 2015b). The leader iterator handles the higher level iterable construct, either a range or a domain, that is iterated over.

```

1 forall (a, b, c) in zip(A, B, C) do {
2     task();
3 }
```

Listing 5.2: A sample **zippered forall** loop with A in the role of the leader iterator

Leader and follower iterators have distinct roles in the execution of parallel loops. The leader is responsible for the creation of parallel tasks and their association to target locales. A single leader guides each forall loop and assigns work using Chapel’s main task-parallel constructs; **begin**, **cobegin**, **coforall**. A leader iterator uses locality constructs, such as the **on** construct, to place parallel tasks on locales. Each task yields locally owned work to assist the leader in the distribution of tasks. The follower iterator receives input work as yielded by the leader. The follower iterates and yields the values of each iteration serially and in the order indicated by the leader.

The block distribution implements a pair of leader/follower iterators for the parallel traversal of the distributed domain (**BlockDom**) and another pair for the distributed array (**BlockArr**). In the context of the distributed domain, the leader

iterator, traverses the target locales, slicing and assigning per locale blocks of indices, the follower then iterates over the indices of each local block. The same logic applies to the distributed array, with iterations over the distributed and local element arrays.

For distributed multi-locale executions, Chapel will normally map one locale per system node. This behaviour is compatible with the Qthreads implementation of the tasking layer, as Qthreads prioritizes performance over load-balancing and assumes that tasks do not compete over system resources, as we detailed in Section 3.7.2. As such in the context of parallel iterators, the leader will create only a small number of tasks per locale—lower than the number of available cores per node, to prevent resource starvation. The above runtime behaviour provides opportunities to hide recovery latency in the case of the resilient implementation, by parallelising recovery and local tasks within buddy locales. We will discuss this behaviour in detail in Section 5.3.4.

5.2 Resilient Block Distribution: Data redundancy

In Chapter 4, we discussed the concept of *buddy locales*, as the central mechanism to assist recovery of failed tasks. In the context of distributed arrays, we reuse the mechanism of buddy locales and implement additional functionality to persist the copies of the index sets redundantly within the **Block** distribution module. In the next paragraphs, we detail the module-level design and implementation.

5.2.1 Data structure additions to the **Block** distribution

Resilient Chapel In contrast to the task-parallel resilient implementation, where locales have uniform access to the task pool, in the data-parallel implementation we need to persist essential data to allow recovery in the event of a failure. Following the design of the block data distribution of Section 5.1.2, we require that each local block instance (**LocBlock**) computes and persists in local storage the remote *chunks* of guest locales, apart from the locally owned *chunk*. The chunks will be used to perform adoption and recovery in the case of guest locale failures. To this end, we introduce an one dimensional array of remote chunks with size equal to the number of guests per locale. We name this array **localResChunks** to conform to the naming

conventions in place.

Similarly, on the local block domain class, a new data structure is required to store the blocks of guest locales. We also require a mechanism to point to the location of guest locales within the array of target locales. The latter array; `localResIndices`, stores $rank * int$ integers, where rank is the underlying domain's dimension. For an initial 2-dimensional array mapped with the block distribution, each locale requires an index set or tuple to refer to every other location, as shown in Listing 5.2. Using the buddy's index, we are able to specify the location to copy to during the initialisation of the local block domain class. We also introduce the `localResBlocks` array, to store the index blocks of guest locales.

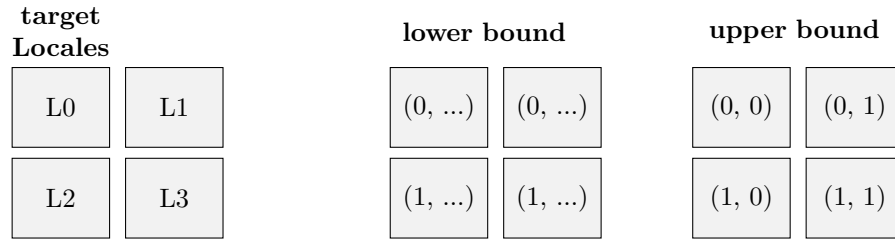


Figure 5.2: The layout of indices of the BlockDom class when mapping a two-dimensional array on four locales. It is used to compute the location of each chunk in the distribution.

Finally, on array level, we create an array of arrays to persist the element values of each guest locale. Remote element values are copied into the `localResElems` array during the initialisation of the elements on the source (guest locale). We maintain the redundant copies up to date with the source's element values. After the adoption of a failed locale by a buddy, the array serves as the source to perform task recovery. This is a critical step towards the correctness of the computation, as it ensures that in the case of failure, a recovery task will execute using valid data. The array also serves as source for the update of redundant copies on secondary buddy locales, thus ensuring correctness in the case of further failures within the same group of buddy locales.

On the local part of the block array, we also maintain the identifiers of the adopted locales and the indices of the locales under recovery, to avoid duplicate recoveries.

5.2.2 Buddy locale configuration

Resilient Chapel For the implementation of resilient task parallelism the buddy configuration required a calculation of the buddy identifiers of each locale, taking into account only the locale’s index, as this was sufficient to adopt tasks from the distributed task list, as maintained by the runtime level. The base calculation provides functionality to return the index of the guest locale. In this implementation we provide a simple round-robin assignment of buddy locales, where a locale adopts the next *numberofbuddies* locales in the incremental order of locale identifiers.

As shown in Listing 5.3, **place** specifies the order of a buddy locale, the first to N-th buddy of the current locale. This ordering is not exposed to the user, but it is required for the management of auxiliary data structures, as introduced earlier in Section 5.2.1. Furthermore, the order of buddies exposes a *primary-secondary buddy relationship* between locales.

```

1 proc _computeBuddyId(loc:int, numberofbuddies:int,
2     place:int, numlocs:int):int {
3     assert(place <= numberofbuddies);
4     if (loc+place >=0 && loc+place <=(numlocs-1)) then
5         return loc+place;
6     else
7         return loc+place-numlocs;
8 }
```

Listing 5.3: Computation of the id of a buddies based on **loc**; the current locale index

As an example, and building on the previous configuration of four locales of Figure 5.2 with a setup of two buddies per locale, we calculate the guests’ local domain positioning, as shown in Table 5.2. A functional example of this configuration, is described in Section 5.3.2

5.2.3 Initialisation of the Block distribution

Resilient Chapel During the execution of application code, the declaration of a distributed array on a multi-locale setup, triggers the initialisation of the block distribution and the subsequent instantiation of its comprising classes. A sample block distributed array declaration is shown in Listing 5.4.

Locale	Primary guest	Primary guest's main do-	Secondary guest	Secondary guest's main do-
Loc0	Loc3	(1, 1)	Loc2	(1, 0)
Loc1	Loc0	(0, 0)	Loc3	(1, 1)
Loc2	Loc1	(0, 1)	Loc0	(0, 0)
Loc3	Loc2	(1, 0)	Loc1	(0, 1)

Table 5.2: A sample configuration of four locales with two buddy locales and their corresponding remote locations to be persisted during the initialisation of the Block distribution.

The code snippet triggers the instantiation of the six main classes of the block distribution. The constructors are enhanced to support the declaration of the new data structures, as discussed in Section 5.2.1. The per locale instances of `LocBlock`, and `LocBlockDom` are populated during initialisation, since the target locales, their indices and the indices of their buddy domains are known or can be computed at this stage.

```

1 const n : int = 100;           // the domain's size
2 const space = {1..n};
3
4 // the number of locales to use in the execution
5 var localeView={0..#numLocales};
6 var myLocales: [localeView] locale =
7   reshape(locales , localeView);
8
9 const ProblemSpace =>space dmapped
10   Block(boundingBox=space , targetLocales=myLocales);
11
12 var A: [ProblemSpace] int;

```

Listing 5.4: Declaration of the block distributed array A of rank 1 over the distributed domain ProblemSpace

A detailed snippet of the implementation, demonstrating the calculation of guests' blocks, is provided in the following section (Section 5.2.4). As the distributed array A of Listing 5.4 (line 10) is initialised with element values, we use the array `localResElements` of the `BlockArr` class as the redundant copy destination. We describe the implementation details in Sections 5.2.4 and 5.2.5, while Figure 5.3 provides an overview of the data structures that have been added to the Block distribution.

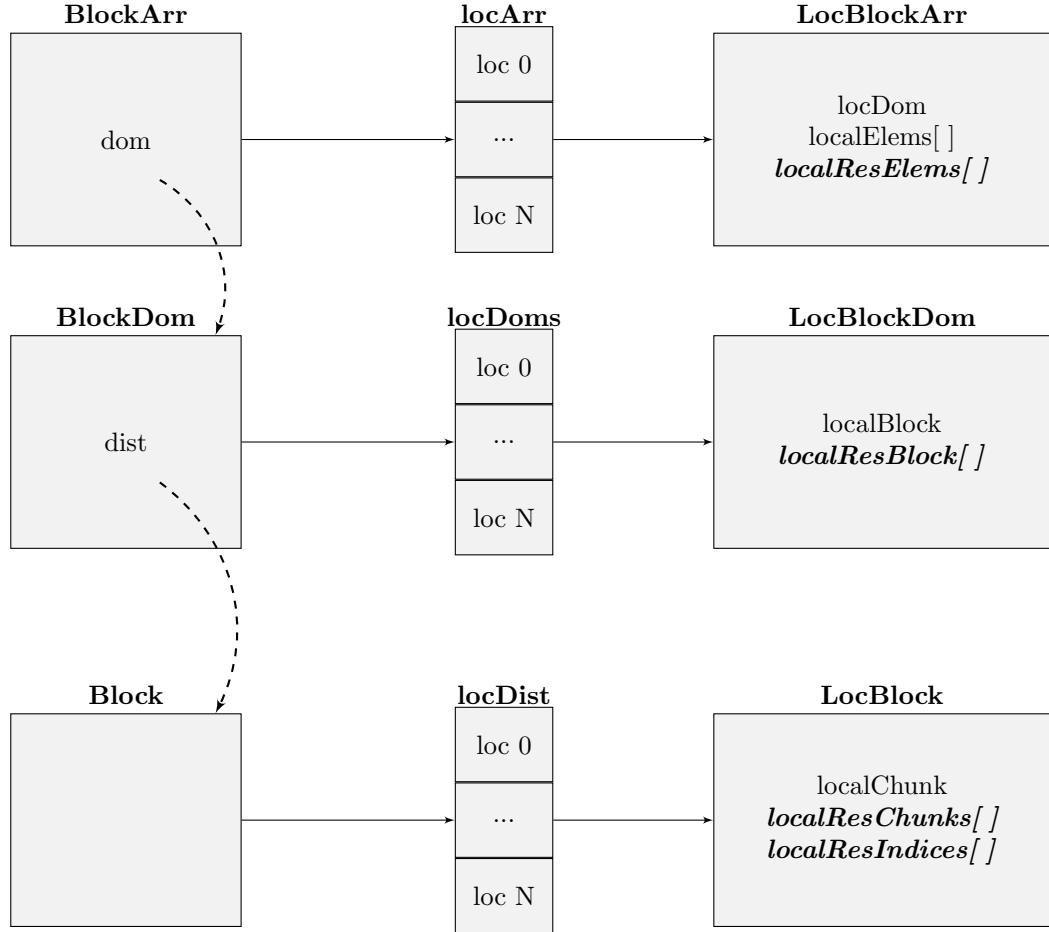


Figure 5.3: Layout of Chapel’s Block distribution classes of the *BlockDistRes* module. The additions to each class to support resilience are shown in bold italics.

5.2.4 Redundant domain initialisation

Resilient Chapel Within the `Block` constructor (`Block.Block`) the `locDist` array is populated at runtime with the instances of local blocks, using a task-parallel `coforall` loop. We introduce a nested zippered iteration to compute the indices of the guest locales and define the array `localResIndices`. On each of the local block classes a locale computes the local chunk; the bounds of the locally owned index set, and the redundant chunks using the auxiliary function `_computeBlock`, passing the local and remote locale’s indices as parameters. At this point we maintain information on the bounds of each blocked domain, including the bounds of the domains of guest locales.

The next point which requires enhancements to support the functionality of buddy locales is the place where the global and local domains are populated. The default Chapel implementation provides the auxiliary method `getChunk(domain, id)`

to allow locales to calculate their local sub-domains. Drawing on the above, we implement a new method to access the chunk of a guest locale, using the locale's identifier. With the above constructs we are able to initialize the local instances of the block domain, by setting the lower bounds of the blocks on each locale. The first step towards this calculation is to compute the identifiers of each guest locale. Once the lower bounds of the blocks are set, we proceed with computing the upper bounds, assigning them accordingly in the arrays `localBlock` and `localResBlocks`. We finally, iterate over the target locales, copy the guests' identifiers in the resilient indices array and the remote guests' local blocks into the local resilient blocks array.

```

1 if locDoms(dist.targetLocDom.low) == nil {
2
3   // Creation/calculation of the lower bound of the
4   // local domain on each locale and the
5   // corresponding guest locales
6   coforall localeIdx in dist.targetLocDom do {
7     on dist.targetLocales(localeIdx) do{
8
9       locDoms(localeIdx) = new LocBlockDom(rank, idxType,
10        stridable,
11        dist.getChunk(whole, localeIdx),
12        dist.locDist[localeIdx].localResIndex,
13        new localResBlocks[]);
14
15       for i in 1..numBuddies do
16         localResBlocks[i] = dist.getResChunk(whole,
17         localeIdx, i);
18     }
19   } else {
20     // Calculation of the upper bound
21     coforall localeIdx in dist.targetLocDom do {
22       on dist.targetLocales(localeIdx) do{
23
24         locDoms(localeIdx).localBlock = dist.getChunk(whole,
25         localeIdx);
26
27         for i in 1..numBuddies do
28           locDoms(localeIdx).localResBlock = dist.getResChunk(
29           whole, localeIdx, i);
30       }
31     }
32   }

```

Listing 5.5: Computation of a guest locales' remote sub-domain by the buddy locale `loc`; the buddy locale's index. This is part of the block distribution's internal functionality, with the addition of the guest sub-domains calculation (Lines 14-16 and 25-27).

Listing 5.5 demonstrates a snippet of the computation of the redundant local chunks of guests on each buddy locale (Lines 14-16 & 25-27). Each buddy iterates over the local one-dimensional array of guest locales, calculating the local resilient blocks. Since the chunks are not statically known, the computation of redundant chunks must take place during program execution.

Listing 5.6 demonstrates the calculation of a remote chunk, based on the local identifier, the guest locale’s identifier and Chapel’s internal utility methods.

```
1 chunk = locDist(locid)
2   .localResChunk((...inds.getIndices()));
```

Listing 5.6: Snippet of the `getResChunk` method calculation of remote index set.

Based on the above code, for an example configuration of four participating locales with two buddies per locale, each locale is required to persist the domain of two guest locales in local memory.

5.2.5 Redundant array initialisation

Resilient Chapel Following the configuration of buddy locales, we require a mechanism to persist the guests’ element values on the buddy locales. The resilient configuration does not currently support array slicing or re-assignment to a different-sized array, as such we can assume that index blocks that are assigned to each locale during the initialisation of the block distribution, continue to *belong* to that locale throughout the implementation; i.e. index sets belong to the initial locale until program completion. As discussed earlier in Section 5.2.1, in order to ensure application correctness, we require that the redundant element values remain up-to-date with the elements on the source locale.

Chapel provides uniform access to arrays; whether local or distributed, on user-level, allowing transparent access and update of local and remote element values, a design principle that stems from the PGAS programming model. The simplest form is *single assignment* of elements, while the user can also iterate serially or in parallel over distributed arrays and assign element values. A second common operation is *bulk copying* between arrays ($A = B$), performed by a task-parallel loop over the target locales, in the form of an aggregate operation.

Single element assignment The read and write operations on an array element in Chapel are performed by calls to the `dsiAccess` method. The method yields the value of the specified element, as part of the *localElems* array or in the case of remote elements, using a global reference to the element, which requires the remote locale's index. In terms of updating the redundant copies on the buddy locales, single element access is an expensive operation; the current implementation requires *numberofbuddies* remote write operations on each `dsiAccess` call.

Bulk array copying For bulk array copying, Chapel performs a zippered task-parallel iteration over the locales and the source and destination arrays. If both arrays comprise of local elements, then the operation is handled as a regular local copying operation. The implementation tackles remote arrays based on their dimension. For arrays of *rank == 1*, Chapel uses the consecutive chunks of the underlying domain as the iterable expression to perform copying. In the case of multi-dimensional arrays each of the ranks is calculated and copied separately. Chapel uses the internal method `array_get` to provide a `this` object reference functionality for classes and to optimize copy operations. As bulk copying element assignment is a more complex operation compared to single element assignment, we introduce a new `updateBuddies` method to update the values on buddy locales. The method call follows the bulk transfer call and involves the calculation of guest locales' indices.

The above operations are used throughout the program to propagate value updates and maintain redundant copies up to date with the values on the guest locale (source). Accesses of an array element on application level directly affect the local elements array on module level, more specifically, the local part of the distributed array assigned to the operating locale. On initialisation of a blocked array, and in order to initialise the redundant data, in the general case, we require *(number of buddies) * (domain indices)* copy operations.

5.3 Resilient Block Distribution: Recovery and adoption

5.3.1 Iteration over redundant data: Forall loop

Resilient Chapel To support task recovery, we require that the guiding iterators of the parallel forall loop are redirected, as discussed in Section 5.1.3, to yield the indices and data of the in-memory redundant copies, in the event of a failure.

The design of resilient leader iterators, builds on the same assumptions we discussed in Section 4.2.1 for the resilient task-parallelism part; *task atomicity* and the *fail-stop model*. The task atomicity requirement is a guard against memory inconsistencies, to ensure that partial data updates from an incomplete execution are not propagated to the redundant copies on the buddy locales.

The leader iterator of the block array class, yields the array elements accessed by the application code, using an internal call to the underlying blocked domain leader iterator. The `BlockDom` iterator yields the required element indices to perform the calculation. We have introduced two enhancements on leader iterators to assist the recovery from redundant data.

Firstly, when a failure is realised on a primary buddy locale, the leader iterator will yield the required indices corresponding to the failed locale after its local indices. The buddy locale prioritizes local over recovery work, similarly to our task-parallel recovery strategy. The possible re-ordering of tasks after a failure does not pose state maintenance concerns, since the tasks within a forall loop are by design independent, as guaranteed by the compiler. Thus, the relative time of computation of a data block, compared to other blocks, does not affect the correctness of the overall computation.

The second modification, arises mainly from the requirements of the evaluation mechanism; since locale failures are only simulations in our testing framework and locales are signalled to interrupt local execution, we need to ensure that the locales considered as failed by the system and the testing interface, remain idle. Thus, we have added a local status check within the `BlockDom` iterator. If the status indicates local failure then the method exits without yielding values. In the scope of a realistic failure, a node would be unable to perform local computation, or the

results would be lost, for example in the case of a network partition; as a result the indices of the local block would not be propagated. Outside the scope of this work, on system level, we assume either a communication layer method set to perform retries to reconnect to the node and/or a system wide abort event; the latter is the policy currently applied by GASNet on locale failure.

As a result of the above, the testing infrastructure is not limited to the external mechanism that kills system processes, but expands on the runtime level in the form of a graceful degradation mechanism. The internal *hooks* into the Chapel components can be used to potentially integrate added functionality, including dynamic load-balancing with reconfiguration of the number of executing tasks, or possibly the detection of data corruption assisted by a resilience-enabled external module.

The data indices and values to be yielded are accessed on leader level. Since the leader iterator is instrumental to the implementation of the forall loop, we are able to guide the iterations that are propagated to the follower iterators.

On leader-level the data indices are calculated dynamically, based on the amount of local data, the size of the local chunks and the available resources. The indices are then propagated to the follower iterators, at which point the application code dictates the type of data transformation to be applied. Neither the leader nor the follower iterators maintain historical data on the indices accessed in each iteration. The part of the local data that is accessed and/or modified in the context of the calculation is only available at the time of the calculation, while the leader does not maintain any lineage of the data that have been updated. The fact that the data transformations cannot be recovered at a later point, and changes to the task state are difficult to rollback in the existing language infrastructure, has led our implementation towards embedding the remote buddy updates within the context of local updates. We additionally guard the correctness of our resilient design by the *task atomicity* assumption, as stated in Section 4.2.1. As such we maintain a relatively synchronous snapshot of the original and redundant copies, throughout the calculation.

Taking into account our task atomicity assumption, another design direction would be to perform update of remote copies after the successful completion of a task. The first performance issue arises from the implementation of bulk copying

operations in Chapel, since they are performed as single element writes. The second performance consideration is the requirement for a synchronisation point to separate the local calculation from the remote update phase. Although the operative domain is accessible on leader-level (and the remote domains can be calculated), the update phase would require a new type of follower iterator, dedicated to remote element updates. This would lead to the internal restructuring of the block distribution to include the update phase; following the synchronisation point. The essential modification on the leader iterator would require the calculation of the remote buddy (sub)domains and the propagation of both the local and remote domains to the follower. The new follower iterator would then perform the remote write operations using the local domain as source and the remote domains as target.

As the addition of a synchronised update phase would be costly and would change the semantics of parallel iterators within the data distributions, we have decided to combine the remote updates with the local calculation. In our design, we perform the remote write operations we dynamically calculate remote indices on the follower iterator and we use data that is available in memory during the calculation.

5.3.2 An example of multi-failure recovery

Building on the next neighbour buddy locale configuration (demonstrated in Table 5.2) Locale 2 is the primary guest of Locale 3, while also the secondary guest of Locale 0. There is also a primary buddy relation between Locales 3 and 0. This setup ensures data availability on the buddy locales after multiple locale failures. The recovery strategy in the case of multiple-failures, is summarized in the following stepped timeline of Figure 5.4.

In the general case, the event of a failure on the first guest locale, will trigger the adopting locale (primary buddy) to retrieve the index block and element values in the first position of the arrays `localResBlocks` and `localResElems`, and accordingly for subsequent guest locales.

More specifically for the above sample configuration, after the first failure (Locale 2) is realised on the primary buddy (Locale 3) (step 2), the buddy will begin recovery action; Locale 3 will yield the requested values corresponding to the locally stored copies for Locale 2 (step 4). Locale 0, as the secondary buddy, continues with local

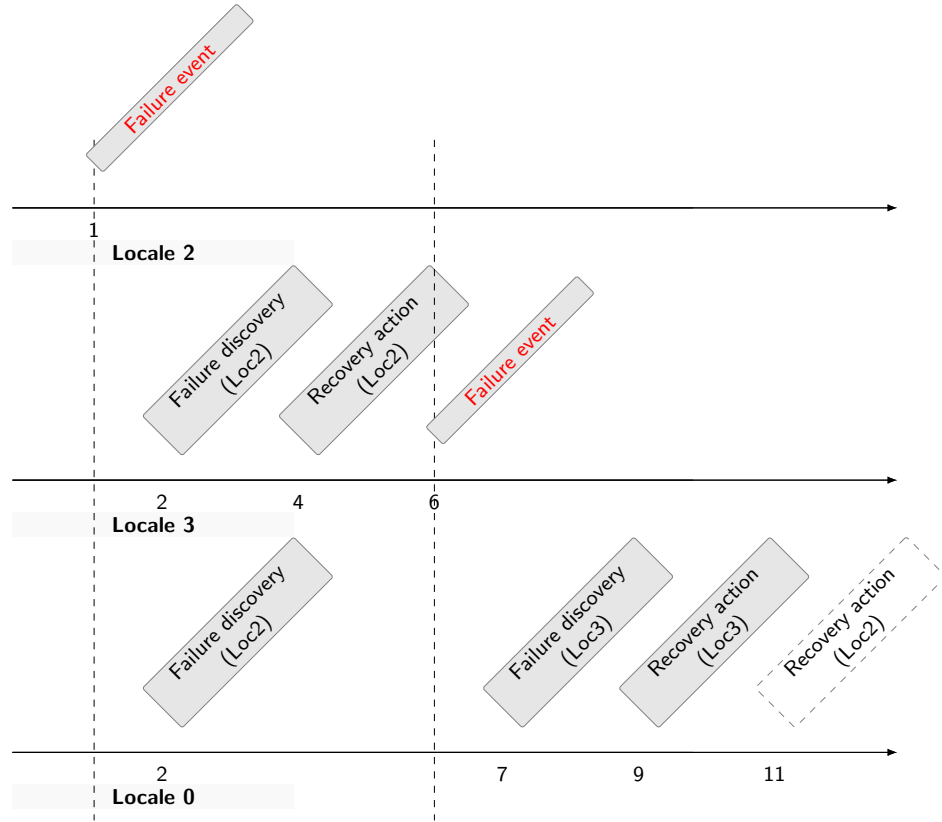


Figure 5.4: A timeline of multiple locale failures and recovery on the buddy locales. The x-axis represents points in time and the y-axis represents the participating locales.

work, until the second failure (Locale 3) is realised (step 7). Locale 0 now begins recovery action for Locale 3 (step 9). At this point we require a mechanism to differentiate between a previously discovered failure that has been recovered and a failure for which task adoption has not begun. In the former case, Locale 0 will only perform recovery for Locale 3.

In order to ensure that secondary₁ buddies have enough information to decide on whether to begin or skip a recovery task, each primary buddy locale signals the secondary buddies of the adopted locale with the completion of recovery tasks. In the above example, on step 7 of Figure 5.4, if Locale 0 has not received the signal of recovery completion by the time Locale 3 fails, then Locale 0 will uptake the recovery of Locale 2 as well, by yielding the elements of Locale 2 from local redundant copies. Locale 0 can begin recovery for Locale 2, based on the following information:

1. Locale 2 has failed
2. Locale 3 is the primary buddy of Locale 2

3. Locale 3 is *alive*; recovery has begun or is scheduled to begin
4. Locale 3 has failed
5. Recovery for Locale 2 has not completed

Table 5.3 demonstrates an abstracted snippet of the decision making process on Locales 0 (right side) and 3 (left side), as multiple failures occur, based on the events of Figure 5.4. More specifically, the snippet gives an overview of the required coordination on the different layers, to support resilience. Firstly, events such as failure signals and recovery completion notifications are realised on the *communication layer* and handled by the registered Active Message handlers. Secondly, the information on the status of locales is maintained on the tasking layer, and the recovery (and regular) tasks are initiated on the *tasking layer*, while the distributed data are maintained on *module-level* within the Block distribution. Each time a recovery completion signal arrives, the according guest locale's status is marked as recovered. From the above discussion and pseudo-code, we can summarize that in order to make decisions about recovery, we require information on failure state including:

1. The status of each locale, within each buddy group;
2. The array of guest locales (on each buddy locale);
3. The array of secondary buddy locales; (on each buddy locale) and
4. The recovery completion information by the primary buddy locale.

We note that this simple mechanism of maintaining information on locales' statuses, can also accommodate a resilient strategy with node *resurrection* or *replacement*. The required adjustments are a new signal and corresponding signal handler to indicate that a node has re-joined the system and is ready to perform work and a bulk-copy operation to restore the local data from remote copies.

5.3.3 Failure Tolerance Threshold

Due to data availability constraints, as opposed to the task-parallel resilience mechanism, we require a calculation for the maximum number of failures that the resilient data-parallel implementation can tolerate, without compromising the correctness requirement. In the task-parallel implementation the computational context is assigned from a centralised task pool, and are available to every executing task via

<pre> 2 // Locale 3 3 // consume messages from queue 4 // event 2 5 // call AM_signal handler 6 guests[0] = 1; //statusFailed 7 8 // event 4 9 // recovery from local data 10 computation(myResElems[0][1..N]); 11 12 // successful recovery 13 // send REC_COMPLETE 14 // to Locale0 15 16 // event 4 17 // local failure 18 localStatus = 1; 19 20 // send SIGNAL to Locale0 21 // send SIGNAL to Locale1 </pre>	<pre> // Locale 0 // consume messages from queue // event 2 // call AM_signal handler guests[1] = 1; guests[0] = 0; // call AM_Recovery_Completed // handler guests[1] = 2; // event 7 // call AM_signal handler guests[0] = 1; // event 9 // recovery from local data computation(myResElems[0][1..N]); // event 11 if(guests[1] != 2) { // recovery from local data computation(myResElems[1][1..N]); } </pre>
--	--

Table 5.3: Pseudo-code of the design for multi-locale failure recovery as implemented on the tasking layer and on the block distribution module level assisted by the ActiveMessages signals of the communication layer implementation. The pseudo-code demonstrates the decision making process triggered by the events of Figure 5.4 on Locale 3 (left side) and Locale 0 (right side).

global references. Here, *the system is able to tolerate at most $N-1$ failures within each buddy group*, where N is the number of locales per group, but in the general case, we need to take into account the overlaps of buddy groups, especially for configurations with multiple buddies. We need to consider the first and last locales in the target locales array, since they also form buddy groups. In order to calculate the number of locale failures that the system can recover from, we provide the below formula:

$$\begin{aligned}
 Fail_max = & (numLocales / (numBuddies + 1)) \times numBuddies + \\
 & (numLocales \bmod numBuddies)
 \end{aligned}
 \tag{5.1}$$

where

$numLocales$ = Number of locales in the execution and

$numBuddies$ = Number of buddies per locale

The maximum number of failures ($Fail_max$), as per Formula 5.1, requires that at least one locale within each buddy group remains live. This requirement applies also to the locales in the beginning and end of the locale range. The requirement could also be considered to set the $Fail_min$ threshold; the minimum number of failures that is guaranteed to bring the system down. In other words, any number of failures larger than $Fail_max$, or failure of all buddies within a buddy group, is guaranteed to terminate the execution.

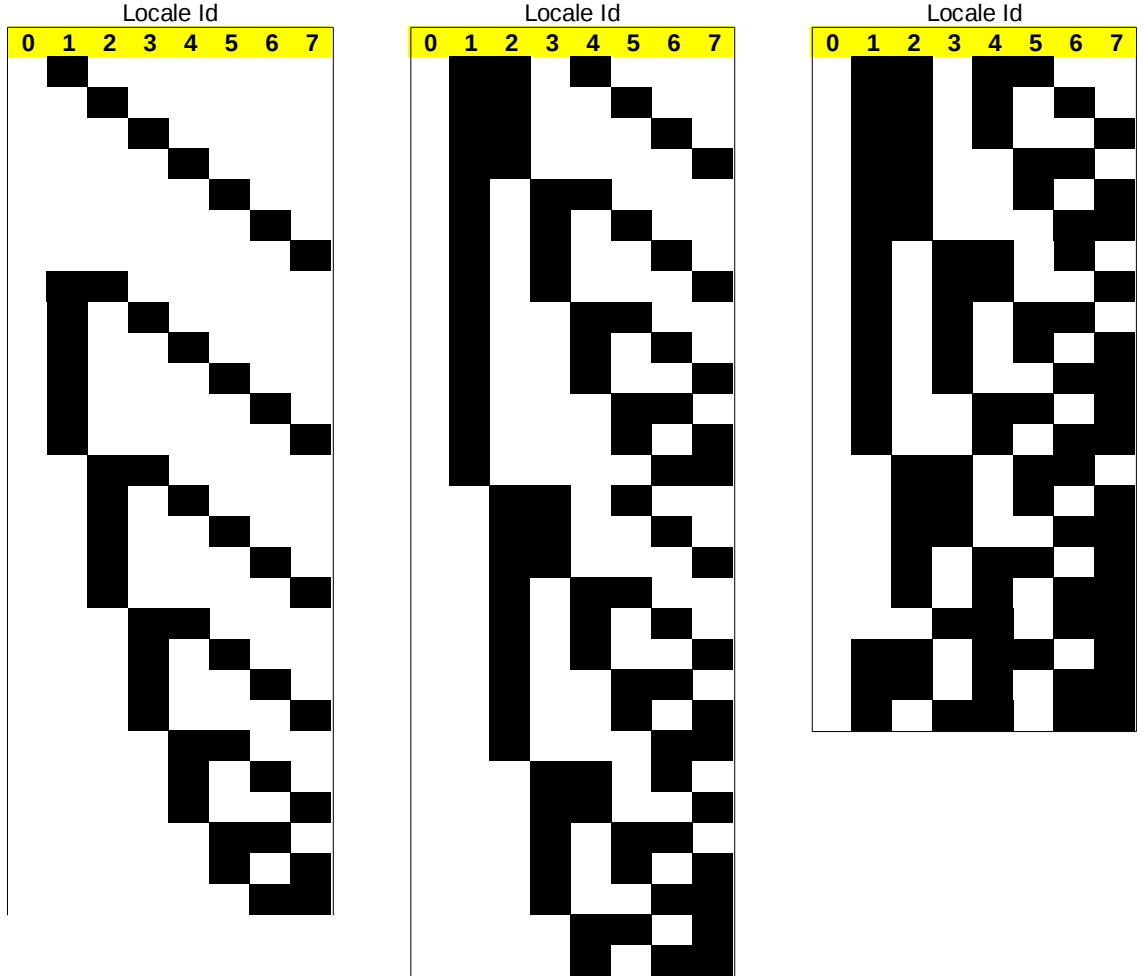


Figure 5.5: The set of recoverable combinations on a sample configuration of 8 locales with 2 buddies

Accordingly, Figure 5.5 demonstrates the set of recoverable failures on the shaded locale Id's for an execution with 8 locales the system is guaranteed to tolerate up to $(8/3 \times 2) + (8 \bmod 3 - 1) = 5$ failures. The combinations shown, comply to the requirement that the failures do not affect all N nodes of an N -buddy group, and Locale 0 is failure-free. In Appendix A.1, we provide a Python script to calculate

all the combinations of a range of Id's, while we also demonstrate the full set of combinations for the configuration with 8 participating locales.

Another conclusion is that, if the programmer requires a different algorithm for the assignment of buddy locales; for example a two buddy configuration of previous and next neighbours, then they will also need to implement a mechanism to query the identifier and the status of the primary buddy of each guest locale. This step is essential to avoid recovery duplication and preserve the integrity of the redundant data.

5.3.4 Parallelisation of Recovery

In Section 5.1.3, we have discussed how the leader iterator of a parallel loop is tuned to prevent resource starvation in the system. This strategy is applied across the participating locales, but also within each locale, since the leader will normally create less tasks compared to the available processors on a node. In contrast to Chapel's task-parallel constructs which allow the programmer to force the creation of new tasks, for example when using `begin` or `coforall`, the number of tasks in a data-parallel `forall` loop is governed solely by the leader iterator.

In the context of resilient Chapel, we are concerned with how the system's *capacity*, which reduces with each new locale failure, may affect the overall runtime of a program. In the hypothetical case, of an *embarrassingly parallel* algorithm that executes without communication and recovery costs, we would expect that the reduced capacity, when losing one node in the setup, would lead to the re-execution of this task after the completion of local work on the buddy node. Essentially, this would lead to a single straggler task —corresponding to a single failure, and the subsequent increase of the overall runtime by the execution of the recovery task.

In Table 5.4 we demonstrate the above scenario. We show two instances of the same execution in the failure-free case (top) and in the case with a single failure (bottom). In the latter case, Locale 2 fails (step 1) and the recovery task is restarted on the buddy locale (Locale 3) after the completion of local work, on step 4. Again, we assume 0% communication costs and 0% overhead for the management of redundant data, so the recovery task can begin immediately after the local work has completed. Also, in this case, the total capacity of the system is reduced after the failure (step

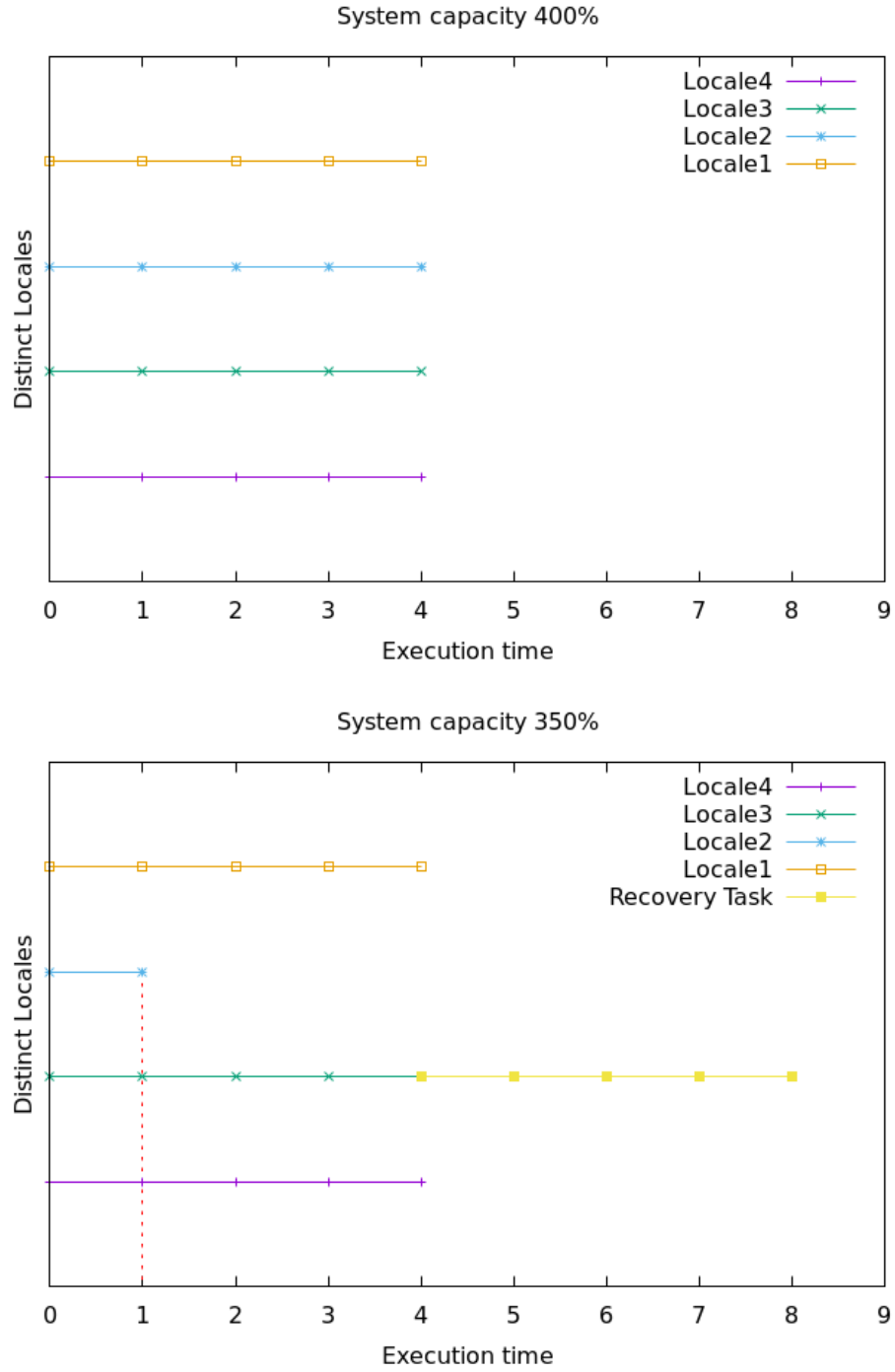


Table 5.4: A hypothetical program execution run on 4 locales, perfectly load-balanced and using full capacity. The x-axis represents the stepped execution time, while the y-axis represents the execution on the distinct locales. The first plot (top) demonstrates a load-balanced execution of a parallel forall loop on 4 locales without failures, while the second plot (below) demonstrates the case with one failure on Locale 2 and the subsequent execution of recovery on the buddy locale (locale 3) after completion of the local work.

1) by 25%, leading to an average overall capacity of 325%.

In order to further investigate the effect of reduced total capacity on the overall execution runtime, we run a data-parallel version of the Mandelbrot micro-benchmark¹ on 4 locales. We use an input set of 2010 rows and columns, corresponding to pixels in the produced representation of the complex fractal set. We run with the resilience enabled version of the runtime, using a single buddy per locale and four failure configurations:

- (a) execution without failures;
- (b) a failure injected before 10% of the total runtime of the failure-free execution;
- (c) a failure injected at exactly 10% of the execution; and
- (d) a failure injected at 15% of the execution.

Configuration	(a)	(b)	(c)	(d)
Mean runtime (mins)	309.50	306.28	312.340	287.10

Table 5.5: Average runtime (in minutes) of the Mandelbrot data-parallel micro-benchmark on a set of different failure injection configurations, with execution on 4 locales.

We run 10 iterations for each setup and present the mean runtimes in Table 5.5. Our results for this realistic example show small speedups for configurations (b) and (d) and a small overhead for configuration (c), when comparing to the failure-free execution (case (a)). We use the default size of arrays (2010 \times 2010), which leads to one bulk task being created per locale. We also consider the communication costs and the internal management of the data structures introduced by the resilient version. The results clearly contradict the hypothetical runtime we demonstrated in Table 5.4, and show that the overall runtime does not increase by the runtime of the additional execution (recovery task). In contrast, the overall runtime benefits from the reduced communication and remote copying costs, when one locale in the system is lost. When injecting a failure at 15% of the execution we note a speedup close to 7.5%, which shows that for larger local chunks and frequent data updates (as required by the algorithm), there is a heavy communication penalty. This is particularly evident in this example, following the failure of the guest locale

¹The Mandelbrot micro-benchmark is included in Chapel’s release. The code can be found under <https://github.com/chapel-lang/chapel/tree/master/test/exercises/Mandelbrot/solutions>.

(Locale 2) there is no communication for data updates, since no secondary buddies are configured. Furthermore, the increased workload on the buddy locale, leads to the creation of more parallel tasks, which execute on previously idle processors. In the Appendix (Section A.4) we provide the data-parallel micro-benchmark and the output of the Mandelbrot execution for the above four configurations.

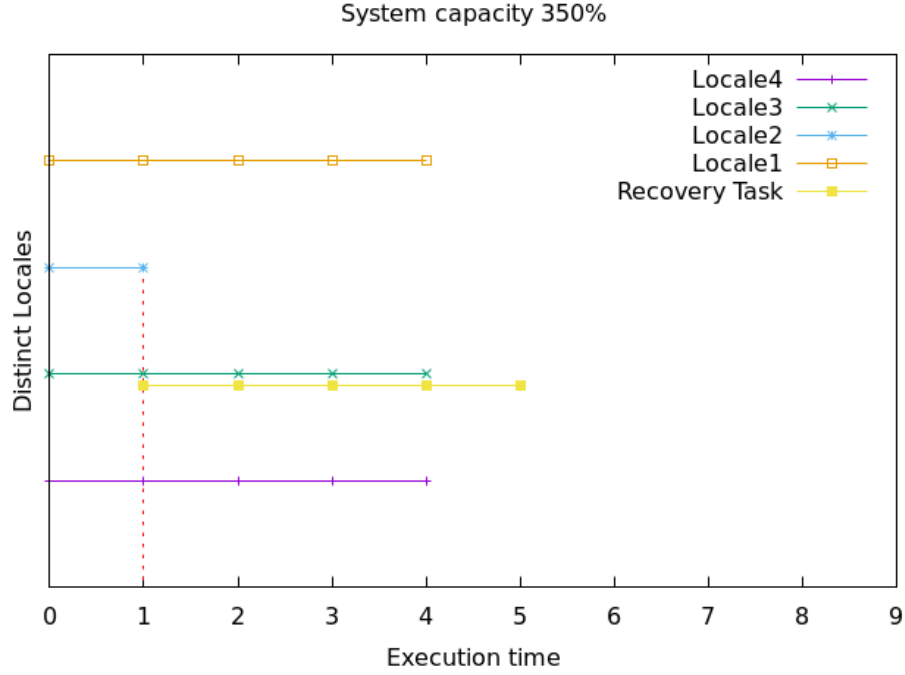


Figure 5.6: A hypothetical program execution run on 4 locales, perfectly load-balanced, before the failure. The x-axis represents the stepped execution time, while the y-axis represents the execution on the distinct locales. The recovery task on the buddy locale begins executing immediately after the failure has occurred and alongside the local tasks on the buddy. Thus the total runtime of the program (bound by the implicit synchronisation of the forall loop), is increased by a small percentage, until the recovery task completes.

Based on the above discussion of design choices both in Chapel and in the Qthreads tasking library, but also from the empirical data of the Mandelbrot micro-benchmark, we conclude that the increase of workload on a buddy locale due to task recovery, guides the leader iterator of a forall loop to create more tasks, thus taking advantage of the idle cores. In Figure 5.6 we demonstrate how the recovery task of the hypothetical scenario of Table 5.4 would be handled within Chapel’s resilient runtime. The above considerations will assist in the discussion of our evaluation results of the resilient block distribution in the following section (Section 5.4).

5.4 Evaluation of the Resilient Data Parallel Implementation

In this section we report on the performance results of the two benchmarks used in our experimental evaluation for Chapel’s data-parallel resilient implementation, the STREAM benchmark (5.4.2) and the N-body algorithm (5.4.3). We validate that the system can recover every time for failures below the *Fail_max* threshold and within the limitations in place. We are particularly interested in the costs associated to the resilient support mechanism when multiple failures below the *Fail_max* threshold occur during execution.

For the STREAM benchmark we initially use a small number of locales to assess the overheads of the resilient implementation. We have chosen a set of fixed input sizes that we can use across testing cases when scaling the number of failures. We then move to larger configurations demonstrating results with significantly increased input sizes, up to 1 million elements ². For the N-body application, we have chosen to scale the number of iterations on two different datasets, in order to demonstrate the correctness and adaptability of our resilient mechanism when employing graceful degradation, due to multiple failures. The smaller size of the datasets has allowed us, with respect to time constraints, to test with multiple failure configurations and provide a larger set of measurements for comparison.

The choice of benchmarks in this section is guided by two main factors; firstly, we require data-parallel applications, since our goal is to verify and evaluate the resilient design of parallel iterators. Secondly, since our mechanism addresses the blocked data distribution, the datasets must be in the form of dense rectangular arrays or matrices. We have also opted for applications in which the calculation on the input data do not introduce unbalanced loads, as the most appropriate candidates for a blocked distribution. We have reviewed existing benchmarks that are commonly used on blocked datasets; linear solvers such as Linpack (Dongarra, 1992), vector operations such as in STREAM triad (McCalpin, 1995) and transforms such as the FFT (Cochran, Cooley, Favon, et al., 1967). In this work we have chosen STREAM triad due to the larger memory requirements of the benchmark, to enable

²Use of one million element arrays for the execution of STREAM is the general recommendation provided by the creator and maintainer of the benchmark (McCalpin, 2002)

us to evaluate the memory overheads when combined with the data replication of the resilient mechanism. For our larger benchmark, we have chosen the N-body simulation, as it is a commonly used application for benchmarking data parallelism across programming languages. The base algorithm is easy to implement and the datasets to execute on are straightforward to produce.

We begin by detailing the experimental setup and we provide empirical evidence to demonstrate the runtime and performance of applications, under different failure configurations. We discuss the results and provide analysis on the sources of the demonstrated overheads.

5.4.1 Experimental setup

For the resilient data-parallel evaluation we have used the same hardware as described in Chapter 4; a 32-node Beowulf cluster. We re-state the specifications for completeness, here. Each node comprises of two Intel Xeon E5506 quad-core CPU's at 2.13 GHz, sharing 12 MB of RAM and connected via Gigabit Ethernet. The nodes run CentOS Linux release 7.5.1804 (Core) x86_64. Each core uses a 32KB L1i cache for instructions and 32KB L1d cache for data and an L2 cache of 256KB. Each quad-core also shares an L3 cache of 4096KB.

5.4.2 STREAM: Sustainable Memory Bandwidth in High Performance Computers

The STREAM benchmark is a synthetic algorithm designed as a memory bandwidth (in MB/s) stress test. STREAM evaluates the performance of four simple vector kernels. Memory bandwidth affects the performance of `read` and `write` operations and, as a result, it drives the performance of data-intensive applications; programs with regular access to in-memory stored data. High bandwidths ensure that data can be retrieved or written by the processor with small performance penalties.

STREAM is composed of four micro-benchmarks; *copy*, *scale*, *sum* and *triad*. Table 5.6 details the four kernels and the number of bytes read or written per iteration. The table also summarizes the floating point operations per second (FLOPS) required per kernel.

In this section, we provide some background on the composition of the benchmark and its impact on memory and compute components of an executing processor.

Name	Kernel	Bytes/Iteration	FLOPS/Iteration
copy	$a(i) = b(i)$	16	0
scale	$a(i) = q * b(i)$	16	1
sum	$a(i) = b(i) + c(i)$	24	1
triad	$a(i) = b(i) + q * c(i)$	24	2

Table 5.6: The vector kernels that compose the STREAM benchmark (McCalpin, 2002). For each kernel we provide the total bytes that are read and written per iteration and the number of floating point operations required.

The *copy* benchmark measures transfer rates in the absence of arithmetic. It is one of the least expensive, but also common memory operations, and consists of the retrieval of two values from memory and a write operation on one of the values.

The *scale* micro-benchmark adds a simple arithmetic operation, by updating element b before writing it to a . This simple scalar operation serves as the basis for building more complex operations, thus *scale*'s performance is an indicator of the performance of larger calculations.

Sum adds a third operand; with three values being retrieved from memory. For larger arrays, the processor's pipeline will fill quickly, so memory bandwidth can be tested. The benchmark approximates a computation often used in real-life applications and it was originally used to perform multiple load/store operations on vector machines.

Finally, the *triad* micro-benchmark uses fused multiple-add (FMA) (Fog, 2012) operations. It builds on *sum* by adding an arithmetic operation to the values retrieved from memory. The *triad* micro-benchmark is directly associated with application performance (McCalpin, 2002; McCalpin, 1995) given that FMA operations are regularly part of basic computations, such as dot products, matrix multiplication, polynomial evaluation, Newton's function evaluation method and digital signal processor (DSP) (Verbauwhede, Schaumont, Piguet, and Kienhuis, 2004) operations.

A common experimental configuration for STREAM is to execute the four micro-benchmarks and then construct an average value, such as the geometric mean, as a way to compare performance on different platforms using a single metric. A set of historical measurements for STREAM on Intel processors can be found in

Appendix A.5.

In this work, we focus on the *triad* benchmark, as it more closely resembles computations that take part in practical applications. The *triad* is used as a stress-test to the efficiency of the resilient implementation. Due to the extensive use of read and write operations on in-memory data, we do expect higher overheads compared to applications with heavier computation, such as the N-body algorithm of Section 5.4.3.

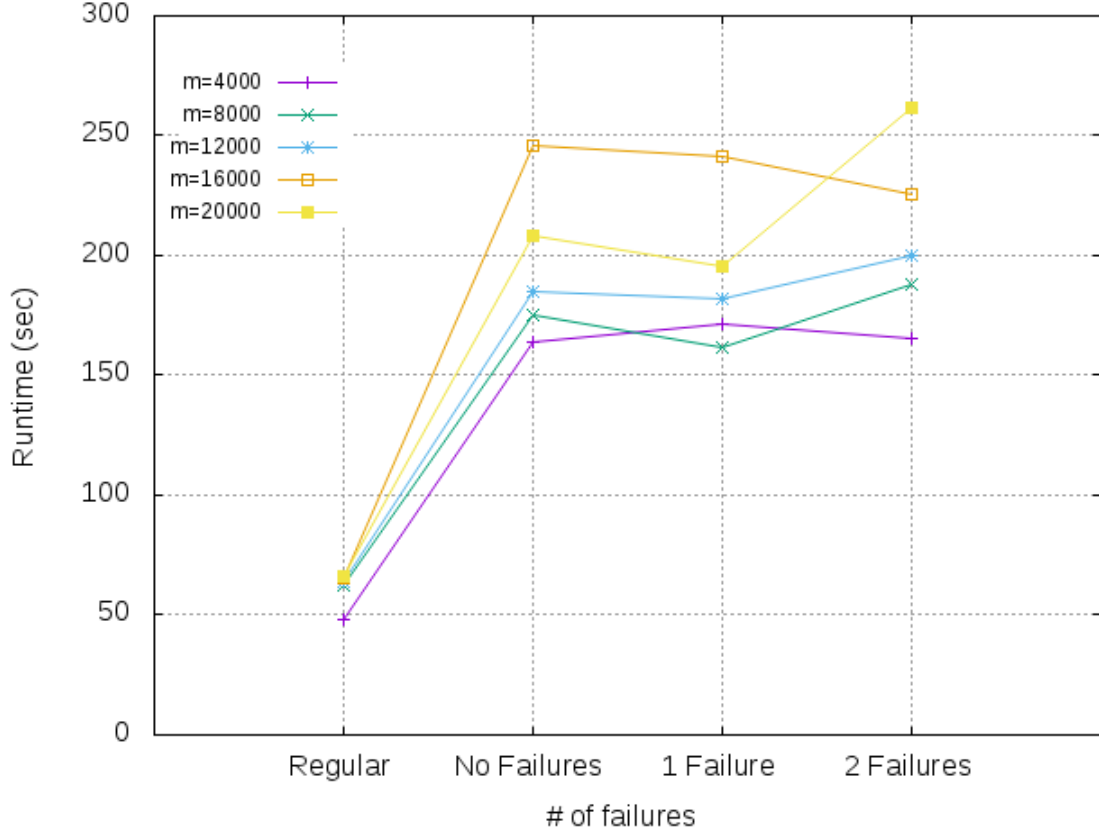


Figure 5.7: Runtime (in seconds) of the STREAM triad execution on four locales with regular Chapel and resilient Chapel for input arrays A and B of size 4K up to 20K and 0 to 2 failures. The end-to-end execution runtimes are measured.

Figure 5.7 demonstrates the execution times for STREAM triad on 4 locales using the resilient blocked distribution. The results show runtimes (in seconds) for input array sizes of 4K up to 20K elements for the initial one-dimensional arrays A and B. We measure the end-to-end execution runtime and in Table 5.7 we provide the full set of percentage differences of the different configurations (0-2 failures) comparing to the regular Chapel version. We also provide the mean overhead percentage across input sizes per failure injection setup.

We plot the runtime (in seconds) with respect to the number of failures and we use the regular version of the block distribution for comparison, for five different input sizes, as the main graph to assess the overhead of resilience on an increasing number of failures (Figure 5.7). The baseline regular Chapel version is marked in the x-axis as **Regular**, followed by the resilient version without failures and the cases with failures.

	A. Runtime (seconds)					
m	4000	8000	12000	16000	20000	
Regular	48.19	62.67	63.89	65.38	66.19	
No failures	164.12	175.24	185.09	246.23	207.96	
1 Failure	171.60	161.28	181.97	241.66	195.14	
2 Failures	165.19	187.72	199.63	225.70	261.79	
	B. Percentage difference to baseline (%)					
m	4000	8000	12000	16000	20000	Mean
No failures	109.20%	94.63%	97.34%	116.07%	103.41%	104.13%
1 Failure	112.29%	88.06%	96.05%	114.82%	98.68%	101.98%
2 Failures	109.66%	104.42%	103.01%	110.15%	119.27%	109.30%
	C. Percentage difference to resilient version without failures (%)					
m	4000	8000	12000	16000	20000	Mean
1 Failure	4.45%	-8.30%	-1.70%	-20.90%	-6.36%	-2.09%
2 Failures	0.64%	6.87%	7.55%	-8.70%	22.92%	4.84%

Table 5.7: Full measurements set of the mean runtime in seconds (A), mean percentage difference (%) to the regular Chapel runtime version (B), on an increasing number of failures and mean percentage difference (%) to the resilient Chapel runtime version (C) of the STREAM triad execution on four locales with regular Chapel and resilient Chapel with 0-2 failures for the input arrays A and B of size 4K up to 20K. We also calculate the mean percentage difference per execution setup compared to regular Chapel.

The algorithm specifies a precision tolerance ϵ value of 0.1, we perform 10K iterations and execute 10 experiments per setup. We have used a configuration of two buddies per locale, and as discussed in Section 5.3.3, for the four participating locales the system’s *Fail_max* is two locale failures. In other words, injecting more than two failures is guaranteed to produce incorrect results. In the case of Locale 0, the failure will lead to fail exit, while failures on other locales, over the *Fail_max* threshold, cause silent data corruption and possibly deadlocks due to faulty task counter update.

In each iteration for the case with failures we introduce a fixed number of failures on random locales up to *Fail_max*. For example in the case of one failure, the locale to fail is chosen randomly out of the set of existing locales, excluding Locale 0. We note from the plot that the input size of arrays affects both the baseline and the resilient version.

When comparing the regular and resilient version without failures we note that the difference in input sizes directly affects both runtimes. For regular Chapel the fivefold increase of the input size (from 4K to 20K elements) produces an overhead of 29.4%, while for the resilient version without failures the overhead is 23.5% (Table 5.7 A). Figure 5.7 demonstrates clearly that, irrespectively of failures, there is a fixed overhead across the different input sizes for the resilient version when comparing to regular Chapel, ranging between 98.6% and 119.2% as per Table 5.7.

Although, the distinct costs per case depend on the number of buddies and the input size of the application, we consider this a *fixed cost* in the sense that it comes up in every execution with the resilient version. The fixed cost includes the initialisation of the block distribution on the target locales, the configuration of the buddy locales, the initialisation of the blocked domains and the blocked array accesses to produce the redundant copies on the buddy locales, as detailed earlier in Sections 5.2.2 through 5.2.5. As discussed earlier, the majority of the bulk copying operations occur during program startup, following the buddy configuration, a fact that accounts for a higher cost to enable the resilient infrastructure. A notable source of overhead are the remote memory access operations that occur throughout the execution to maintain the redundant data up to date; these costs are more difficult to isolate and measure.

To clearly demonstrate the fixed cost we provide Figure 5.8. The figure shows a condensed and restructured version of the average execution runtimes over input size of Figure 5.7, including only the regular Chapel version and the resilient version without failures, for the different input sizes. The average overhead accounts to 103.4% between the two versions, while there is also a peak in the runtime for the input size of 16K elements.

For the input of 16K elements we note a larger runtime increase in the setup without failures; with an overhead of 13% compared to the larger input size of 20K

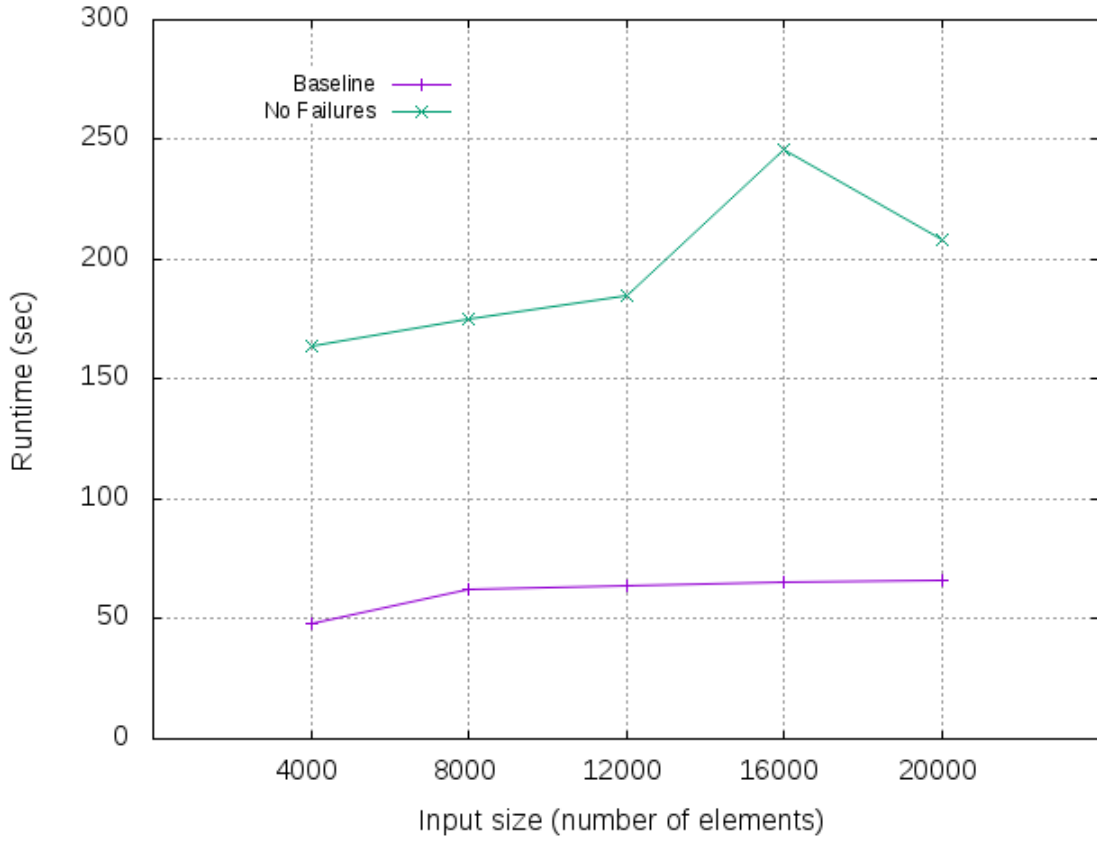


Figure 5.8: **Absolute overhead** Runtime (in seconds). Comparison of the baseline and the resilient version without failures for STREAM triad for the input size arrays of 4K to 20K.

elements. Taking into account the underlying hardware characteristics and the fact that the application occupies a single core per locale, with each locale corresponding to one node in the execution, we believe that the demonstrated runtime increase is related to the L2 cache latency effects.

Input size	4K	8K	12K	16K	20K
Local array block size	1K	2K	3K	4K	5K
Local array blocks	3	3	3	3	3
Redundant array blocks	6	6	6	6	6
Total elements	9K	18K	27K	36K	45K
Total memory space	72KB	144KB	216KB	288KB	360KB

Table 5.8: Block sizes persisted per locale per input size. The calculation includes the local block sizes and the blocks persisted redundantly on each buddy locale for the configuration with two buddy locales. The array elements are double precision `real`'s of 64 bit width.

In Table 5.8 we summarize the total elements persisted per locale during program initialisation, i.e. the additional memory requirements per locale for the resilient

block distribution. We take into account the default bit width of 64 of Chapel’s `real` (double precision) data type, used for the array elements. The per locale memory requirements are calculated based on Formula 5.2, below, while in our experiments the local blocks on each locale are of equal size. The memory requirements are fixed throughout a failure-free execution, while for executions with failures the total required memory decreases. As less nodes participate in the execution, less redundancy is maintained as the redundant copies become primary sources of data.

$$\begin{aligned} \text{Memory_per_locale} = \\ (localBlockSize + firstBuddyBlockSize + \dots + NthBuddyBlockSize) \quad (5.2) \\ \times numArrays \times dataTypeLength \end{aligned}$$

where

localBlockSize = The size of the local block after the distribution;

numArrays = The number of block distributed arrays in the application;

dataTypeLength = The memory space required for the data type of array elements, and

XthBuddyBlockSize = The size of the local block of the *Xth* buddy after the distribution.

Each core on our machine is equipped with an L1d cache of 32KB, an L2 cache of 256KB and a shared L3 cache of 4MB for every 4 cores that form a NUMA node. The cache line size is 64 bytes and the L1d and L2 caches follow the exclusive model. According to Intel’s Software Developer’s Manual (Documentation, 2011), the primary cause of misses on L1d and L2 caches is poor data locality combined with a large dataset. In this sense, the pipeline is stalled waiting for memory, since most of the Last Level Cache (LLC) misses end up accessing the RAM.

In the case of STREAM’s resilient execution without failures, although the leader iterators yield local consecutive chunks to follow within the parallel loop, the accesses to update the redundant copies as they are retrieved from remote memory locations, become more expensive, due to the offloading of *hot data* from the cache. On the other hand, following a failure, the data accessed by a buddy locale show better

data locality, as these are accesses to locally stored consecutive chunks, due to the configuration of the buddies. The latter is an example where modifications to the runtime system have performance side-effects beyond the immediate overhead of managing the required data to enable re-computation of results.

For the case of 16K elements the memory requirements of the application in the failure-free case, account for 288KB which is the combined size of the L1d and L2 caches. That leads to the conclusion that the loading of a new cache line from L1d cache is followed necessarily by the eviction of another line to L2, leading to progressively more expensive offloading operations. In contrast, for the input size of 20K elements, the data set is larger than the L2 cache (360KB), which makes the processor pre-fetching more visible, as the wait time for the next cache line is shorter compared to the access time to the L2 cache, thus leading to smaller runtimes for the input of 20K elements.

Sources of runtime overhead The main sources for the demonstrated runtime overhead with respect to the additions and modifications to the Block distribution, are summarized in the following list:

- *Remote copying* in the beginning of the execution; for example, for the configuration with 2 buddy locales per locale and for the input size of 12K arrays each locale is assigned three blocks (arrays A, B, and C) of 3K elements each. Each locale copies 3K elements per input array per guest locale, resulting in a $2 \text{ guests} * 3 \text{ arrays per locale} * 3K \text{ blocksize} = 18K$ of element copying from guest locales. The overall copying for redundancy when using a two buddy configuration accounts to 72K elements for all four locales. This is a fixed cost in the sense that it occurs once in the beginning of each execution.
- *Single-element access*: each time an element is read or written in the program, either serially or within a loop, a remote copying operation takes place to update the corresponding element on the buddy locales. In the regular case, a remote access in the context of Chapel's distributions requires a set of checks on the runtime in order to define whether a local or remote element is written and a recalculation of the index sets of the participating locales in the remote case. To allow value updates in the resilient case, we handle the updates of redundant

data on the guest locales. As such, the update of a single local value requires one local and two remote write operations. Thus a loop which updates the values of all elements over the initial problem space of 12K elements, requires $2(\text{arrays}) * 3K \text{ local write operations} + 2(\text{buddies}) * 2(\text{arrays}) * 3K \text{ remote write operations}$ adding up to 18K write operations in total. In general, the write operations required for a full update are $\text{localChunkSize} + \text{numBuddies} * \text{remoteChunkSize}$, for a distribution with equally-sized chunks across locales. This is a variable cost in the sense that is related to memory read/write costs and communication overheads.

- *Added functionality* on the runtime level, in the forms of:
 - status checks of remote locales, which require on-the-fly recalculation of buddy indices, as this information is not available on the blocked array’s level; and
 - initial configuration of buddy locales, including the calculation of remote index sets, which accounts for a low percentage of the overall overhead.

When comparing the resilient version without failures to the cases with failure injections of Figure 5.7, we note that the costs of task adoption and recovery are significantly smaller; 4.4% overhead for the input size of 4K elements with one failure, while in the majority of the cases, these costs are amortized by improved data locality, thus leading to speedups of up to 8.7% (for the input size of 1K elements with two failures). This is consistent to our explanation of the cache effects; as more failures occur and the working data set is accumulated on the single available (“live”) locale, the program shows better data locality. Our conclusions of recovery parallelisation of Section 5.3.4 are also applicable in this case, since the STREAM triad calculation occupies 2 to 3 cores per locale in the failure-free case.

In Figure 5.9, we demonstrate the runtime of the application code as measured by Chapel’s **Time** module. The timing includes the STREAM triad calculation. As before, the element updates on the buddy locales during each iteration and the task adoption and data-parallel recovery in the cases with failures, are also included. The difference to the plot of Figure 5.7 is the initialisation cost of the block distribution, which takes into account the initialisation of the distributed array and the remote write operations to initialize the redundant copies.

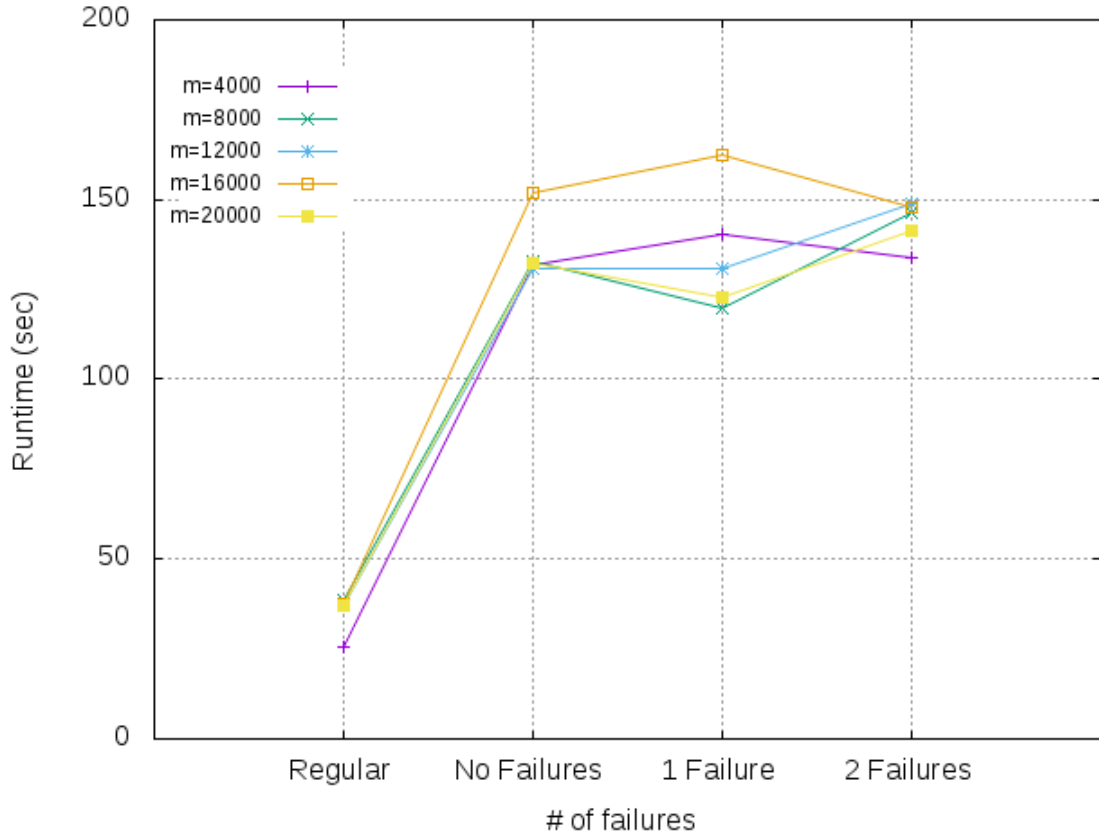


Figure 5.9: Application code mean runtime (in seconds) of the STREAM triad execution on four locales with baseline Chapel and resilient Chapel with 0-2 failures for input arrays A and B of size 4K up to 20K. The runtime is measured using Chapel’s `Time` module.

The first thing to note when comparing the Figures 5.7 and 5.9, is that in the case of regular Chapel, the overheads of the initialisation of the arrays in the distribution are particularly high. When excluding this part from our measurements the runtime difference decreases by an average of 54.4% across input sizes. In comparison, the resilient version without failures shows an average decrease of 35.1% in runtime, compared to the end-to-end runtime measurements. Since the same input datasets are used per input size, the approximate 20% difference of the runtime difference is a rough indication of the average overhead introduced by the initialisation of the redundant copies on the buddy locales, across input sizes.

The difference is also evident in the cases with failures. Here, for the case with two failures, the runtime converges between 130-150 seconds across input sizes. The illustration of the performance of the main calculation indicates that the overhead of recovering two tasks, when comparing the cases without failures and with 2 failures

is on average 9.7%, while the largest runtime decrease is 38.8% (demonstrated in the case of 16K elements), though the latter is affected by the memory latency issues discussed earlier in this section.

In Figure 5.10 we demonstrate a set of measurements for STREAM triad for the larger input sizes of 500.000 and 1,000.000 elements. Based on the formula of Section 5.3.3 for 12 participating locales, the system is guaranteed to recover a maximum of 8 failures.

The results demonstrate, as before, an initial fixed overhead when using the resilient block distribution. For the 500K input size of the resilient version, the overhead accounts for 68.5% while for the larger input of 1 million elements the overhead accounts for 44.8%, which implies better processor utilization due to the larger dataset.

We should also note that the recovery of two failures compared to the resilient version without failures introduces larger overheads compared to the previous experiments. More specifically, the difference between the *No failures* case and the case of *2 failures* demonstrates overheads of 21% and 52.2%, respectively for the two input sizes. As opposed to the previous experiments, the recovery of two failures for the case of 1 million elements poses non-trivial overheads. The higher local workload on the buddy locale does not provide, in this case, opportunities to parallelise the task recovery.

On the other hand, after the threshold of 2 failures, the cost of further failure recoveries is normalized, with 8 failures introducing only 6.8% overhead compared to the case with 2 failures, for the input of 1 million elements. This means that although the system’s capacity is reduced by 66.6%, the recovery workload is amortised by the reduced communication for updates of the redundant data and the improved data locality. As the recovery of 2 failures remains a straggler for the system, the cost of subsequent recoveries is, in large part, obscured. For completeness, we also provide the full set of runtime measurements in Table 5.9.

The above results make obvious the fact that the added communication and the remote copying operations are the main factors that introduce overhead. These costs, though dependent on the input size and the number of buddy locales per configuration, are consistently higher compared to the cost of task-adoption and re-

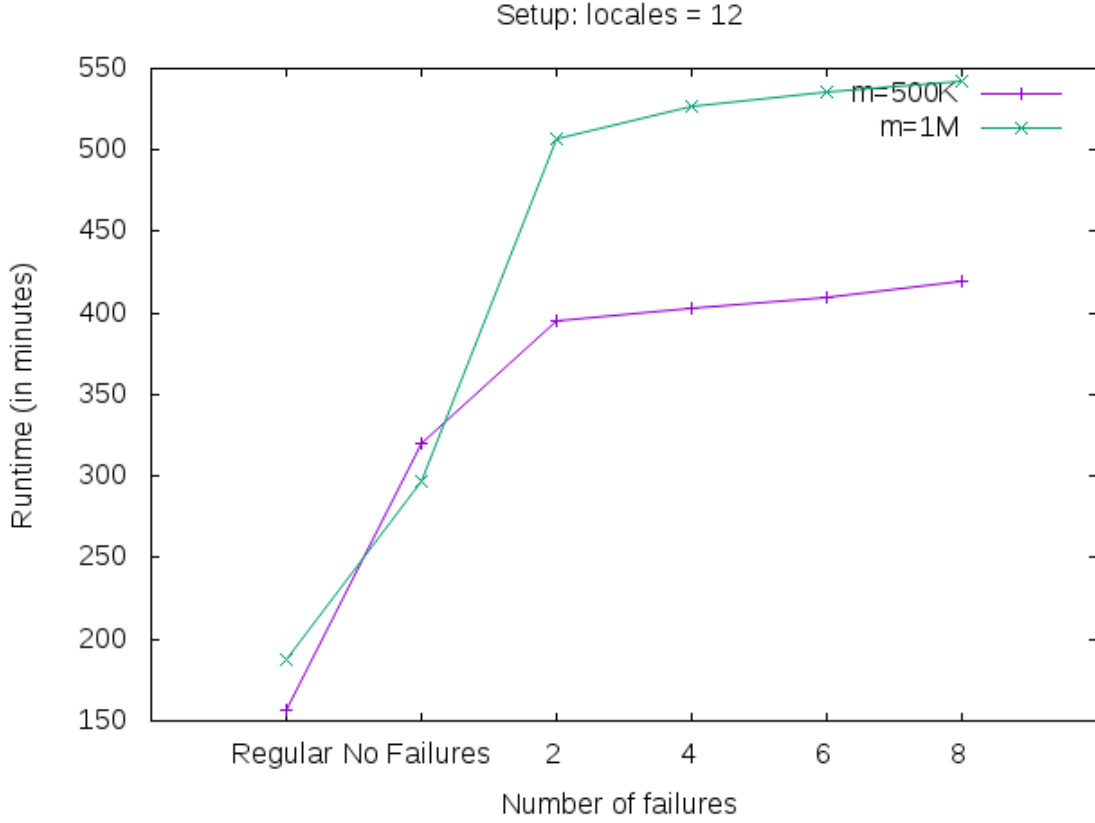


Figure 5.10: Runtime (in seconds) of the STREAM triad, comparison of the baseline and resilient version without failures and the resilient version across 2-8 failures for STREAM triad for the input size arrays of 500K and 1M elements.

m	500000	1000000
Regular	156.7898	188.0277
No Failures	320.48077	296.6098
2 Failures	395.7574	506.3997
4 Failures	402.7511	527.2684
6 Failures	409.7312	535.7154
8 Failures	420.0953	542.5347

Table 5.9: Full measurements set of runtime (in minutes) of the STREAM triad execution on twelve locales with baseline Chapel and resilient Chapel with 0-8 failures for input arrays A and B of size 500K up to 1M.

covery from redundant data. The above are indicated by the reduced performance of the application with failures when compared to the resilient version without failures.

As stated initially, the memory-intensive STREAM triad benchmark is a stress-test for our resilience implementation, which results in the above mentioned high overheads. In the following section, we apply our resilience approach to a real-life data-intensive application with substantial computational costs.

5.4.3 N-body: approximation of particle motion

The general N-body simulation problem describes the evolutionary motion of particles within an isolated system. The particles interact through physical forces; more commonly the gravitational Newtonian force when describing celestial bodies, but similar simulations cover the movement of atoms within gas clouds. The problem has been studied extensively by mathematicians and physicists, including Gauss, Lagrange and Laplace, but a general analytic solution has not been provided yet by researchers (Heggie and Hut, 2003; Wisdom and Holman, 1991).

In the bibliography, we find implementations of several Nbody algorithms (such as *all-pairs*, *Barnes-Hut*, and *finite multipole*) across languages, where in the common case, the positions, velocities and masses of the participating bodies are initialised with pseudo-random values. In this implementation we use the all-pairs algorithm³, a simple solution for Nbody. Though it is not highly tuned for performance, it is simple enough to allow a straightforward evaluation of the impact of the resilient mechanism on performance. We use two predefined input sets of twenty (20) and forty (40) solar bodies to verify the correctness of our results. The datasets can be found in Section B.3 of the Appendix.

In each time step, the bodies are forced by the gravitational power to develop speed and move in the space. The bodies advance for a total number of *iterations* and from a physics perspective we are interested in measuring the initial and final energy produced in the system. For the purposes of this work we are interested in the completion of the entire set of calculations that describes the movement and interaction of the bodies, when executing on a parallel system with node failures.

Figure 5.11 demonstrates the mean execution runtime of both the baseline and the resilient version for datasets of 20 and 40 participating bodies with 5K and 10K iterations on 16 locales. We use a two-buddy configuration per locale and we introduce up to 10 failures, while we perform 10 experiments per test case.

As demonstrated in previous experiments, the use of the resilient version introduces overheads compared to the regular Chapel runtime, irrespectively of the presence or the number of failures. This is particularly evident when comparing

³The block distributed parallel version is adjusted from the serial Nbody Chapel implementation, which can be found under <https://github.com/chapel-lang/chapel/tree/release/1.12/test/release/examples/benchmarks/shootout>

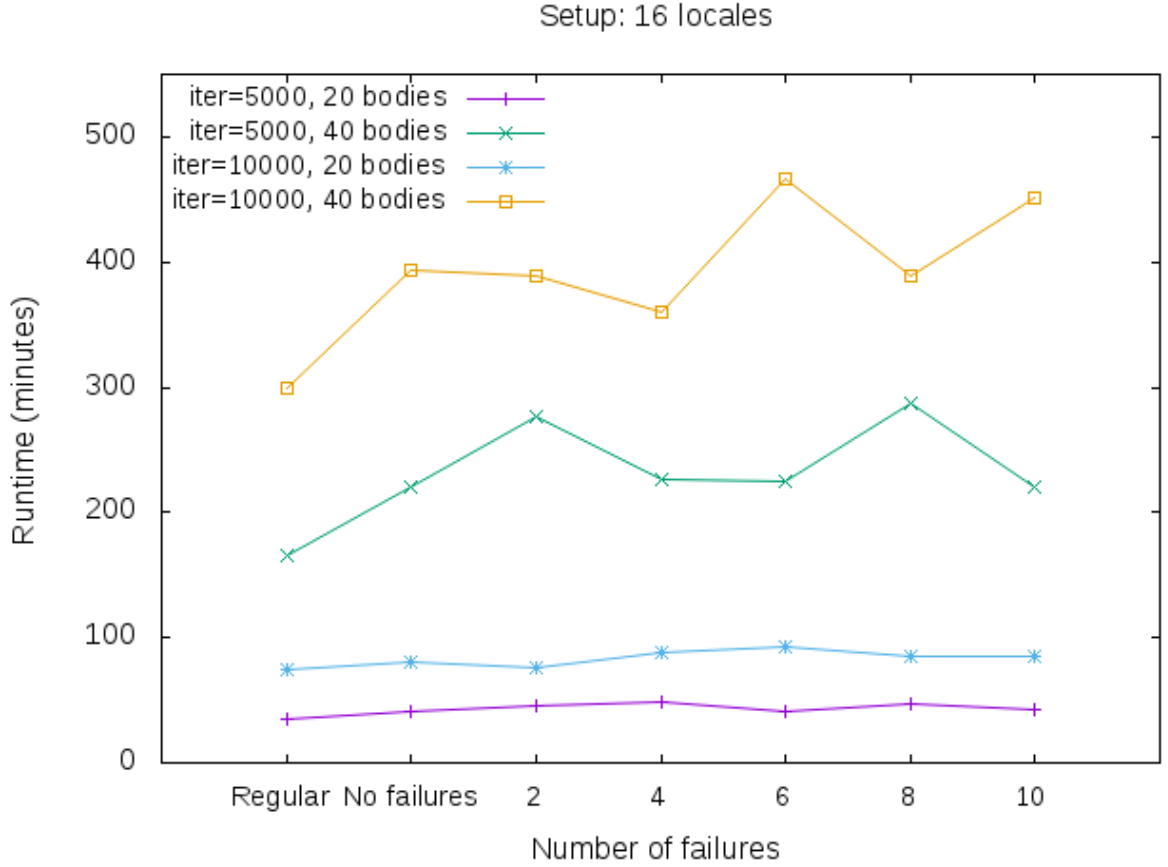


Figure 5.11: Runtime (minutes) of the Nbody execution for the datasets of 20 and 40 bodies with 5K and 10K iterations on 16 locales. We provide the runtime of the baseline Chapel runtime, the resilient version without failures and the runtimes with 2-10 failure injections.

the baseline to the resilient version without failures; corresponding to the first two points on the x-axis of Figure 5.11, for the experiments with 40 bodies.

Figure 5.12 demonstrates the *absolute overhead* of the resilient runtime version for the datasets of 20 and 40 celestial bodies with an increasing number of failures compared to the results of the baseline Chapel implementation for both executions of 5K and 10K iterations. The results demonstrate high variability across the different input sizes and across the range of injected failures.

The management of the additional data structures and the performance overhead of remote write operations to initialise the redundant copies during program initialisation and throughout the execution to update the copies, accounts for the demonstrated increase of 16.7% in the case of 5K iterations and 7.6% in the case of 10K iterations for 20 bodies, when moving from the baseline to the resilient version

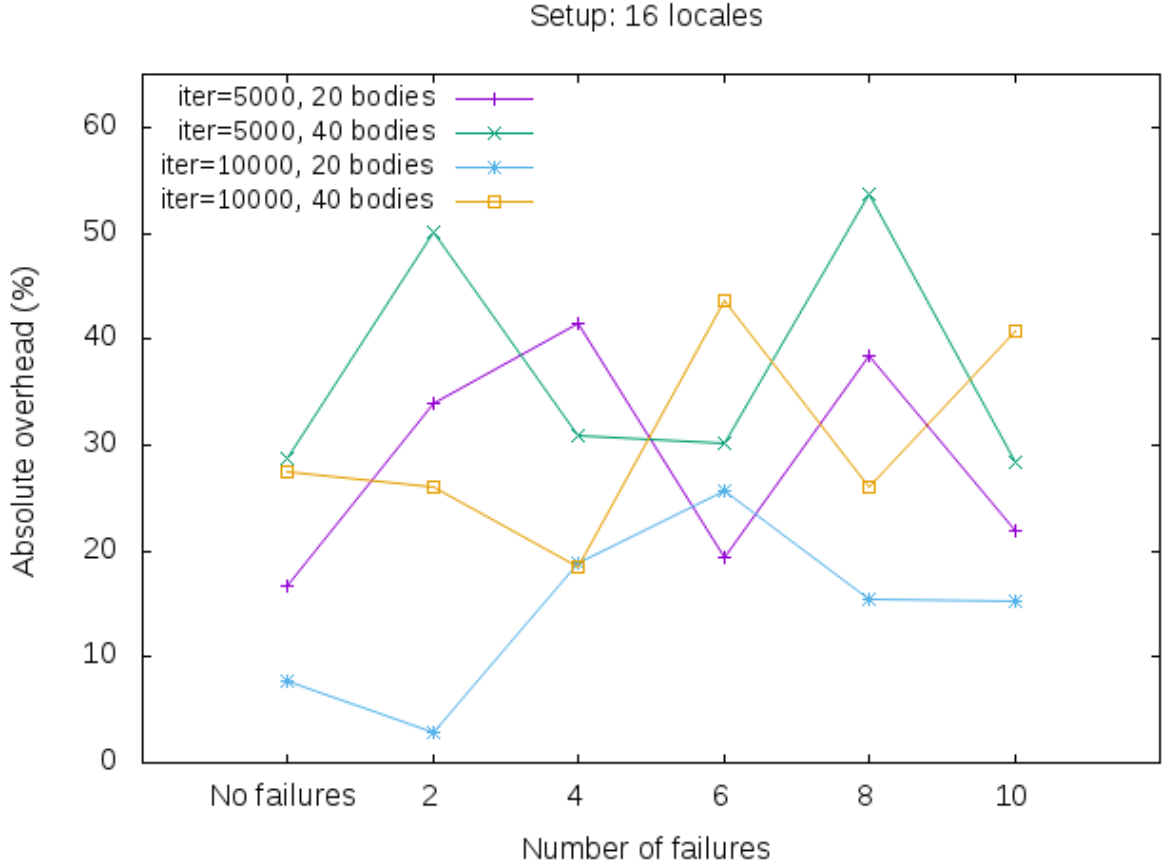


Figure 5.12: **Absolute overhead** (%) of the N-body execution with 5K and 10K iterations for 20 and 40 bodies on 16 locales *compared to baseline Chapel*, over an increasing number of failures.

without failures. The overheads for the larger dataset are respectively 28.6% and 27.4%. As discussed earlier, we consider these costs fixed, in the sense that the additional operations are required to enable the resilient support per our design.

When comparing the test cases (both datasets for 5K and 10K iterations) and based on Figure 5.12 and Table 5.10 of the full set of absolute overhead measurements, we observe lower overheads and improved overall performance for this realistic example, compared to the results for STREAM triad, in the previous section (Section 5.4.2). For the Nbody calculation, the total execution time is larger and the computation is less memory-bound compared to STREAM triad. More specifically, for the calculation of 10K iterations with 20 bodies, the cost of resilience is on average smaller, irrespectively of the number of failures. With the exception of the case with 6 failures, the plot shows better performance for the execution with 10K iterations compared to the execution of 5K iterations. Specifically, the average

overhead with failures for 10K iterations is 15.59% while for the 5K execution the average overhead is 31.04%, close to the 30% threshold set earlier. This is also linked to task creation as dictated by the leader iterator of the parallel loop.

	20 bodies		40 bodies	
Iterations	5000	10000	5000	10000
No Failures	16.74%	7.68%	28.65%	27.39%
2 Failures	33.96%	2.78%	50.05%	26.07%
4 Failures	41.52%	18.93%	30.89%	18.52%
6 Failures	19.36%	25.63%	30.12%	43.65%
8 Failures	38.42%	15.46%	53.62%	26.00%
10 Failures	21.95%	15.18%	28.29%	40.78%

Table 5.10: Complete measurements set of absolute overhead (%) of the N-body execution with 5K and 10K iterations for 20 and 40 bodies on 16 locales when *compared to baseline Chapel*, over an incremental number of failures.

Nonetheless, for the version with 20 bodies and 5K iterations we note overheads of 41.5% and 38.4% when four and eight randomly selected locales fail, respectively. The smallest overhead occurs in the case of six failures (19.36%) , while the resilient version without failures introduces a 16.7% on top of the baseline’s runtime. Finally, the resilient version without failures for the 10K iterations case is on average 7.6% slower compared to the baseline. We note that the overheads for 5K iterations present a *standard deviation of 9.67*; a high value compared to the values in the working set. The high standard deviation shows that the runtime points are spread out compared to the mean. The main parameter that introduces variability in the runtime and subsequent overheads is the placement of failures.

For the dataset of 40 bodies, the granularity of the local calculation per locale is increased, but we also observe irregular initial workloads across the target Locales, stemming from the distribution of the dataset. The average runtime (in minutes) and the corresponding absolute overhead of the resilient implementation compared to the baseline Chapel version, are demonstrated in Figures 5.11 and 5.12, respectively.

For the input set of 40 bodies we demonstrate larger absolute overhead compared to the baseline, with the largest percentage increase occurring for 5K iterations with 8 locale failures (53.6%). The mean absolute overhead for the dataset of 40 bodies is 36.9% for 5K iterations and 30.4% for 10K iterations. Although the average performance remains close to the initial threshold of 30%, Figure 5.12 shows that

the overheads present high variability, with standard deviation values of 10.6 and 8.8 respectively for the 5K and 10K iterations runs.

In Figure 5.13 we demonstrate the *relative overheads* for both datasets and iteration configurations, compared to the resilient version of the runtime. Our goal here is to investigate the *variable cost* of the resilient mechanism associated to the cases with failures and identify its possible causes. In Table 5.11 we also provide the full set of measurements of relative overhead across test cases.

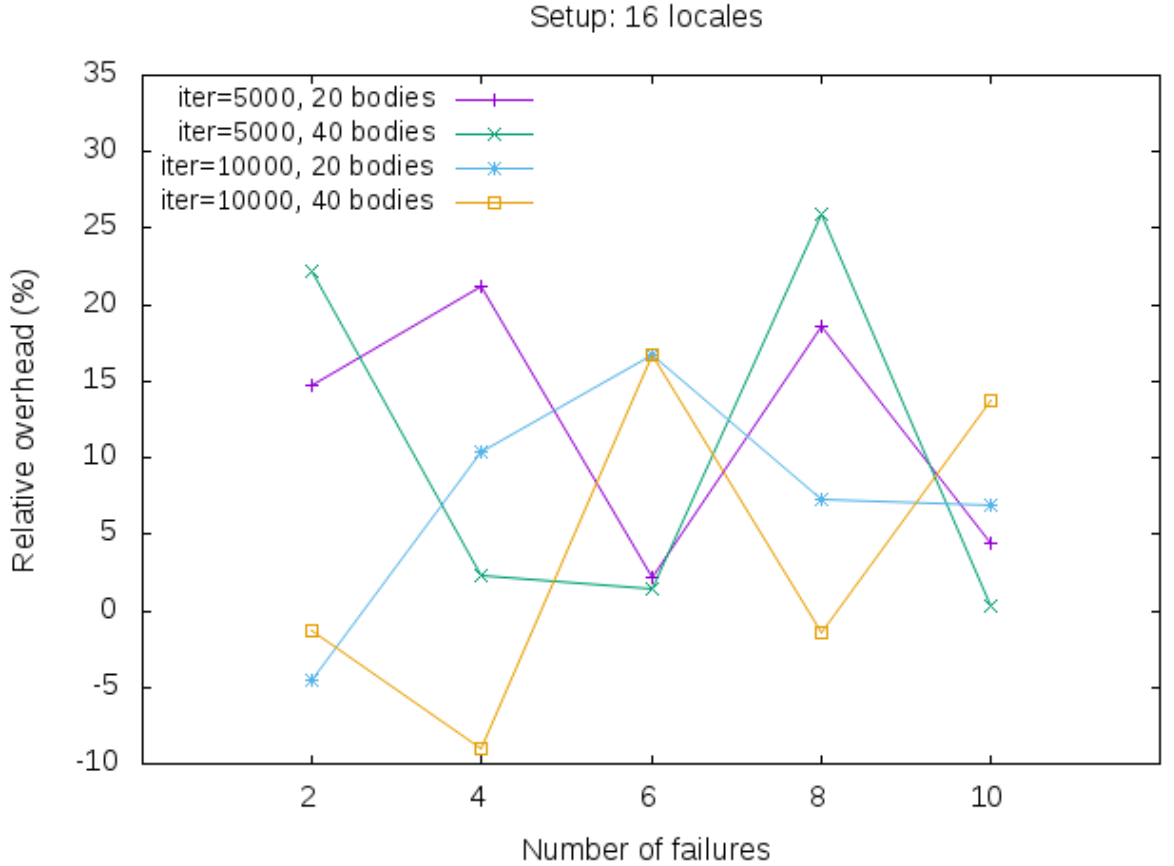


Figure 5.13: **Relative overhead** (%) of the N-body execution with 5K and 10K iterations for 20 and 40 bodies on 16 locales *compared to the resilient version*, over an increasing number of failures.

The maximum relative overhead for 20 bodies is 21.2% and occurs for 5K iterations when 1/4 of the total target locales in the execution suffer fatal failures, while we also note a decrease of 5% in the runtime for the execution of 10K iterations with 2 locale failures. For 40 bodies, the maximum relative overhead of 25.9% occurs for 8 locale failures and 5K iterations.

We attribute the high variability of these results, along with the few cases of

	20 bodies		40 bodies	
Iterations	i=5000	i=10000	i=5000	i=10000
2 Failures	14.74%	-4.54%	22.19%	-1.34%
4 Failures	21.21%	10.44%	2.29%	-8.97%
6 Failures	2.24%	16.67%	1.49%	16.76%
8 Failures	18.56%	7.22%	25.96%	-1.40%
10 Failures	4.45%	6.96%	0.37%	13.77%

Table 5.11: Complete measurements set of relative overhead (%) of the N-body execution with 5K and 10K iterations for 20 and 40 bodies on 16 locales when compared to Chapel’s resilient version, over an incremental number of failures.

speedups compared to the case without failures, to factors, related to the characteristics of the resilient mechanism, but also the configuration of the experiments.

First, we need to clarify that the *fixed cost* required to introduce the resilience mechanism is high by our design. The fixed costs include the remote read and write operations to initialise the additional data structures on top of the regular costs of initialisation of Chapel. Thus when focusing on the cases with failures the relative overheads are low, as they are amortized by the initial cost. Furthermore, the small datasets and the subsequent low workload assigned on each target locale, leads to lower task creation and thus lower utilisation on the participating nodes. As a result, the systemic overhead in the event of failures and the subsequent task adoption is dominated by the communication cost rather than the actual cost of execution of the recovery tasks, which are in many cases parallelised.

Also, the datasets used in the experiments are distributed in uneven chunks, with a number of locales in the execution being assigned double the workloads, according to the formula of the block distribution (Section 5.1). In this sense, the placement of failures can introduce large variability in the runtime measurements. More specifically, failures on the locales with lower workloads increase task granularity on the adopting locales, but the total overhead of recovery in this case is obscured in part by the runtimes of the loaded locales due to the implicit synchronisation point of the forall loop. In contrast, failures on locales with higher workloads may lead to unbalanced execution and straggler tasks, for example in the case of 4 failures with 5K iterations.

A secondary factor are early locale failures; failures that are clustered in the beginning of the execution, due to the limitations of our testing mechanism (Sec-

tion 1.3). When multiple failures occur early in the execution, primary buddy locales that adopt the failed tasks are able to begin recovery immediately after the failed status has been propagated. This also reduces the total communication cost for updating the remote copies on secondary buddies throughout the execution, since according to the recovery completion mechanism, discussed in Section 5.3.2, as more locales fail the size of the buddy sets is reduced, leading to less communication for updates.

In this section, we have demonstrated comparative runtime and overhead results of a widely-used scientific application benchmark using Chapel’s resilient block data distribution on an increasing number of locale failures. The results show that the penalty for using the resilient version instead of the baseline is associated to the problem size and the granularity of the distributed tasks. We have also discussed how the placement of failures can affect the absolute and relative overheads of the resilient execution, a factor that is also pointed out in the evaluation of Resilient X10. Specifically, when running 10K iterations of the Nbody algorithm with 20 bodies the runtime of the resilient version is 5.5 minutes higher than the baseline, where the execution requires on average 75 minutes.

We have discussed the issues of the *fixed costs* introduced by our mechanism and the main factors that introduce variability. We have also demonstrated, that for a program with significant computational load and infrequent memory accesses, such as the N-body algorithm, the relative overhead of the resilient version without failures remains below or close 30% across test cases and input sizes.

5.5 Portability to Other Predefined Distributions

The high-level design of the Block distribution, as discussed in Section 5.1.2, is shared among Chapel’s predefined distributions, such as *Cyclic* and *BlockCyclic*. The distributed and local domains, local chunks, and local arrays are present, while the major difference is the index sets calculation that is custom per distribution.

The main implementation steps to support resilience for a Chapel distribution, following the resilient design in this work, can be summarized to the following:

- A buddy locale implementation, with methods to query the status of remote

locales and methods to calculate the indices of redundant data, customised to the distribution.

- A set of new data structures to persist the redundant data, calculated in the previous step. We do not expect the data structures of redundant data to differ significantly to the ones presented in this work.
- An new implementation of iterators which respects the status of locales (*alive* or *failed*) and handles the traversal of the regular and redundant data structures, to accommodate the recovery functionality.

Taking into account the shared design principles across Chapel’s predefined distributions, we believe that the above implementation steps and the resilient design as discussed throughout the chapter, is flexible enough to enable the resilient mechanism within other Chapel predefined distributions. We should note though, that the mapping of indices to the target locales is specialised to each distribution. We identify the resilient implementation of the leader and follower iterators, as the most time consuming part of our implementation, especially concerning the modifications to allow the retrieval of data from redundant copies.

5.6 Summary

In this Chapter, we have discussed and evaluated our resilient implementation for Chapel’s block data distribution. The implementation spans across the runtime level and the block distribution module and allows transparent resilience in the event of locale failures, on parallel block distributed application programs. While the larger part of the implementation is integrated within the runtime system; communication and tasking layers, the tuning of the internal iterators is implemented on library-level, and more specifically within the blocked data distribution and the distribution utility modules. We have discussed the implementation details, and described the need for redundant copies in the system. We have provided a resilient version of the block distribution with support for task recovery and *in-memory redundant data storage*, with the use of *buddy locales*. Our mechanism provides a guarantee on the number of failures that can be recovered by the system.

We provided a set of benchmarks to evaluate the performance of our mecha-

nism. We presented evaluation results for STREAM a micro-benchmark performing a set of common data operations which is used for memory testing and for the all-pairs N-body algorithm, a data intensive physics application for particle movement simulation.

Our results on the STREAM micro-benchmark, the stress test application used for evaluation, show large overheads for small datasets, accounting to a 220% runtime increase, when comparing the resilient version without failures to the baseline’s end-to-end runtime, and slightly smaller overheads when measuring the main calculation’s execution. This cost is directly associated with the management of the new data structures and the extensive copying used to initialize and maintain the redundant data up to date. Unless the computation itself depends heavily on remote memory accesses and after the initial fixed cost of using the resilient block distribution, the results for the cases with locale failures show small variability and even demonstrate some speedups of up to 10% compared to the case without failures.

In the case of the N-body benchmark, our results show, that although the initialisation costs of the block distribution account for overheads of up to 40%, as the input datasets become larger, the overheads in the experiments with failures decrease significantly down to 13.3% when performing 10K iterations. We note that, for the cases with failures, the overheads vary significantly, depending on the placement of the locales that suffer failures and their assigned workloads.

As the main sources of overhead we have identified the additional data structure management on locale level, and the placement of failures with respect to the per-locale assigned workloads. We have also provided a parametric formula to calculate the number of failures that our system is guaranteed to tolerate, based on the number of target locales in the execution and the configuration of the number of buddies per locale.

From a programmer’s perspective, it is a question of the type of the application and the purpose it serves, to make the decision of whether to use the resilient version. For example, assuming that the N-body program is part of a calculation on a mission critical system, then a resilience-enabled program is preferable. Furthermore, resilience is more desirable for long-running parallel applications, in which cases the high fixed costs are amortised by the longer total runtime with the lower variable

costs. On the other hand, if there are time constraints or if the program executes multiple times and only a rough estimate is required, for example if we require only the order of magnitude of the produced energy in the N-body calculation, then the baseline version is able to cover this need, as it is sufficient to calculate the average, taking into account only the results of successful executions.

Another factor to put under consideration is the type of system the application executes on. A system built for reliability will provide software or hardware mechanisms to avoid, mediate or mask failures; transient faults, fatal failures or both, so the use of a resilience enabled version might be redundant. Vendors will often distinguish between High Availability (HA) servers and Fault tolerant (FT) server configurations. High availability servers have independent components with failover capabilities; the servers monitor one another based on a set of health metrics while in the event of a failure the application is migrated and restarted on a live server. *Windows Server Failover Clustering (WSFC)* is an example of a high availability solution offered in the Windows Server suite. Fault tolerant solutions, on the other hand, focus on providing redundancy of the computation. Here the hardware is tightly coupled, executing a single instance of the operating system and multiple instances of the application in lock step. Every instruction is executed across components, similarly to a set of mirrored machines. When a failure occurs the surviving system can take over the application execution with minimum downtime. An example of a fault tolerant system is the *Endurance 6200* by Marathon.

Though the execution platform is an important factor to consider for resilience, we assume that the average programmer will typically have access to commodity clusters, with the subsequent high failure risks. In this case, the demonstrated overhead may not be prohibitive, compared to the delays of abrupt termination.

The most important characteristic of the implementation in our view, is the programmability aspect. Faced with the above scenarios, or executing on a number of systems with different reliability factors, a programmer may switch between the resilient and the baseline implementation only with the modification of the library module in use, that adds up to modifying a single line of code and recompiling the application code.

We have worked towards an automated solution that does not require user-

assistance. To this end, we have not provided comparisons to user-assisted techniques, such as checkpoint-restart. In specific scenarios, checkpointing the redundant data on buddy locales could prove beneficial to performance, since it would remove parts of the communication required to update remote data. On the other hand, checkpoint-restart poses new design challenges, such as the choice of when to perform checkpointing and state correctness considerations. Since we have opted for a transparent automated solution, we argue that the most important reason for not supporting checkpoint-restart (or a hybrid solution) is the fact that the mechanism is not automated, it requires user assistance and support on application layer. For the above reasons, we argue that the use of checkpointing mechanisms remains outside the scope and goals of this work.

This implementation is to the best of our knowledge the first attempt to support resilience on module-level and assisted by the runtime for data distributions in Chapel. Although, we focus on the block distribution, the main concepts are applicable to other Chapel predefined distributions; such as *Cyclic* and *BlockCyclic*, and possibly to similar constructs in other PGAS, such as X10's distributed arrays and CAF's co-arrays. Finally, we have discussed possible implementation challenges and we have identified the main steps required for porting the resilient design.

Chapter 6

Conclusion

6.1 Summary

The thesis investigates the design and implementation of embedded transparent resilience support in Chapel –a modern high-level parallel programming language. Our goal is to provide graceful degradation for long-running parallel applications, in the presence of component failures. We have demonstrated that support for transparent resilience can be embedded in Chapel, as per our key hypothesis, and that no application-level modifications are required to enable resilience. We have demonstrated *automatic adoption* and *recovery* for orphaned tasks of failed locales, embedded in the runtime system, primarily on the tasking and communication layers.

Resilience is identified as one of the main challenges to tackle in order to achieve exascale performance, as the rapid scaling of component count in HPC systems introduces increased failure rates. On the other hand, modern parallel languages strive for programmability, offering powerful abstractions to lift the burden of explicit synchronization control for application programmers. The thesis is an attempt to reconcile the conflicting requirements of *programmability* in a modern programming language and *high performance* in the presence of failures. A programming language with embedded resilience lifts many of the programming challenges of traditional fault-tolerance approaches, such as failure discovery and coordination, while also alleviates the need for user-assisted mechanisms and third-party monitoring.

Our *design* focuses on the runtime level; particularly the communication and

tasking layers, and within the libraries; particularly, within the *blocked data distribution*. We employ the concept of *buddy locales* to handle the adoption and recovery of tasks and to act as a decentralised distributed *resilient store*. Our resilient design tackles the main issue of abrupt program termination in the occurrence of *fail-stop* failures, and performs automatic task-adoption and recovery, without added programming effort or modifications on the application-level. This functionality can introduce higher runtime overheads when compared to other user-assisted approaches, such as optimised checkpointing.

The basis of the design is the migration of lost calculation on task-level to remote functioning nodes and its rescheduling and re-execution from locally persisted redundant data, thus taking advantage of the locality property. Our design shares common principles to a number of other languages and frameworks with resilience capabilities. Most prominently, the design primarily addresses the runtime layer and libraries (similarly to Resilient X10); in-memory replication of redundant data (such as in YARN and Resilient X10); failure mitigation (similarly to Erlang), node links (such as in Erlang and Resilient X10). A detailed design comparison of Chapel to other fault tolerant systems has been addressed in Chapter 2 and summarised in Tables 2.4 and 2.5.

Chapel is a programming language of the Partitioned Global Address Space programming model. It is actively developed by Cray, built from first principles, and it supports a variety of programming styles. Chapel builds on a set of high-level constructs and abstractions, providing opportunities for embedding our resilient mechanisms. More specifically, we take advantage of the inherent *distinction of parallelism and locality*, and use them as the basis for building a task-adoption and recovery mechanism and a distributed in-memory *data redundancy scheme*.

Our *implementation* is embedded within the runtime system and in the case of data-parallel applications, within the library modules. We extend and modify the functionality on the communication and the tasking layer to guide the adoption of tasks. We also extend existing library code to support the resilient mechanism and to accommodate the resilient store. This allows for portability of the resilient application code across different systems, only by rebuilding the runtime system and re-compiling the application code.

We have provided an empirical evaluation of our resilient mechanism and have demonstrated its applicability on both task- and data-parallel applications, ensuring uninterrupted program execution and correctness. The design and implementation of task-parallel resilience are detailed in Chapter 4, while in Chapter 5 we focus on the resilient blocked distribution. As one core foundational achievement of our work, we develop a formula (Chapter 5) to calculate the guaranteed number of failures that can be recovered, based on the number of participating locales and the configuration of buddy locales. The formula derives from our design and it is validated through testing.

For resilient data-parallelism, we have implemented an in-memory data redundancy mechanism assisted by buddy locales, to ensure data availability in the event of failures. The mechanism, implemented within the block distribution module, relies on remote copying operations to initialise and maintain the redundant copies up to date. Our results show that memory-intensive applications have significant fixed performance penalties compared to the baseline version of the runtime system, but execution costs are amortised as more failures are introduced in the system.

The analysis of the experimental results, has revealed a pattern of fixed costs to enable the resilient mechanism, while the results for the cases with multiple failure injections show significantly variable costs. The variability of the latter is a strong indication of how the placement of failures can affect the execution runtime.

More specifically, our results for the task-parallel implementation show that the resilient mechanism, as tested on a set of synthetic micro-benchmarks, introduces overheads of up to 17% for unstructured task-parallelism. The task-parallel micro-benchmarks are designed to test task nesting patterns, commonly used in Chapel applications. For structured task-parallelism we have demonstrated overheads of up to 26% for task-parallel loops with up to 6 failures on 16 locales and we have also shown that the number of buddies in a failure-free task-parallel application does not affect the execution runtime.

We have also demonstrated experimental results on two more realistic sample applications using the block data distribution; the STREAM triad memory benchmark and the N-body all-pairs algorithm. For STREAM triad, as an application with frequent accesses to memory, we demonstrated significant performance penalties with

higher mean overheads. For N-body, the increased per locale task granularity following the task adoptions on the buddy locales provides the runtime system with opportunities to hide communication and memory latency. The absolute overheads of resilience without failures account for 27% of the runtime in the worst case for small bodies datasets. The relative overheads demonstrate high variability, with a 21.2% maximum overhead for 5K participating bodies with 4 failures and 13.7% for the input of 10K bodies with 10 failures, both on a 16 locale configuration. For the larger datasets of 500K and 1M celestial bodies, the initial fixed costs reach 68%, while the variable costs when multiple failures occur, are bound by the cost of single recovery.

6.2 Contributions

The main research contributions of the thesis can be summarized to the following:

- We have developed a *design for transparent resilience* on a representative language of a class of high-level programming languages (PGAS), focusing primarily on maintaining transparency and programmability.
- We have *implemented* our mechanism within the runtime system and within Chapel’s standard module library, building on top of existing programming abstractions and taking into consideration the design principles of the underlying programming model.
- We have provided an *empirical validation and experimental evaluation* of the proposed mechanism on sample applications on a large-scale system and we have identified the factors that introduce overhead, aiming to provide context for programmers on the trade-off’s of using the resilient implementation.

The detailed contributions of the thesis are:

1. The design of a transparent resilience framework for Chapel’s task-parallel language constructs: covering the *begin* and *cobegin* task-parallel constructs and the task-parallel *coforall* loop. The design of a transparent resilient version of the *blocked distribution* one of Chapel’s pre-defined data distributions. The system is able to recover from multiple failures, and we provide guaranteed recovery up to a *Fail_max* threshold.

2. The design and implementation of an in-memory data redundancy mechanism with distributed data copies, inspired and guided by Chapel’s data locality features. We have used the construct of *buddy locales*, as an alternative solution to external file systems, to provide a self-contained implementation.
3. An implementation of the resilient design for Chapel’s task-parallel constructs. The implementation is integrated on the runtime system, more specifically on the communication and the tasking layers. We have provided an evaluation of the resilient task-parallel mechanism, with empirical results on a set of constructed micro-benchmarks.
4. The implementation of the resilient mechanism for the block data distribution. The implementation builds on the runtime system modifications and expands on library-level. On module-level, we integrate the data redundancy mechanism and implement the internal parallel iterators to manage the task adoption and task recovery on the *buddy locales*.
5. A fault injection mechanism for simulating node failures in a distributed setup on top of the GASNet lower-level communication library. This serves as an auxiliary implementation to facilitate the experimental evaluation. An instrumental part of the mechanism embeds *status awareness* capabilities for the participating locales in the runtime system. As such, it can serve as the basis for potential extensions: including the integration of mechanisms that provide health metrics information, or the design of a checkpointing strategy with the use of an external file system mechanism.
6. To provide context for our work we have provided a critical review of resilience and fault tolerance in high performance systems, covering predominantly used fault detection and recovery mechanisms. We also review existing languages and runtime systems with resilience capabilities.

6.3 Limitations and Future Work

In this section, we provide a list of current limitations of this work and we propose improvements and possible topics for extension and future work.

Limitations

- We have made the design choice to perform re-execution of the *lost* calculation from the beginning, as the progress and state of threads is not exposed from the lower runtime on the tasking and communication layers. Since Chapel is a language with side-effects, one of the assumptions we use as basis for our design is *task atomicity*; we require that the tasks either complete successfully or fail, and the entire task is re-run on recovery. The migration of the computation does not automatically introduce side-effects, with the exception of programs with explicit synchronisation in the form of locks, which may lead to deadlocks or livelocks. We do not currently provide a method to statically determine side-effects.
- The current implementation does not cover programs with explicit computation placement requirements, as the recovery design is based on the migration of the computation in the event of failure.
- The out-of-band failure injection mechanism executes alongside the application, as such the tunability in the distribution of failure injection is limited. Failures are injected serially with respect to each other and in a time-clustered manner.
- Currently, the number of buddies and the algorithm for the placement of buddies is not exposed on application level. This is a limitation in the tunability of the implementation; the advanced programmer is required to access the runtime implementation to tune these parameters.
- In this work we assume that dead nodes can be detected, while from a systems perspective a timeout or lack of heartbeat is only an indication of a failed node. Slow or unresponsive nodes may resume and make progress.

It is common for resilience implementations to disregard the detection part. Our assumptions makes clearer the distinction between *detection* and *recovery*. Even mature implementations, such as checkpoint restart, assume a type of external detection mechanism in order to decide when to perform recovery; in this case restart from the checkpoint. In the context of a resilient programming language implementation, the lower runtime layers should be responsible for the part of detection. Here, the most prominent candidate is the GASNet communication layer, possibly linked or assisted by an external mechanism.

The possibility that a dead node does make progress is covered by our design with the propagation of status information. Once a node is presumed dead (even falsely) it is no longer part of the execution. Following task adoption and update of the status tables the incoming signals are *programmatically discarded*.

Assuming such a testing case, our experimental methodology would have to perform modifications within the GASNet implementation for example by adding delays for outgoing signals from nodes that are considered dead; non-trivial to implement since GASNet only exposes a minimal API for integration. New method signatures should be added, while we would also need to define a new signal to indicate that a node enters *slow mode*; trivial to implement via linux-level signals.

On the resilient part though, since the above checks are in place, slow nodes would be handled in strictly the same way as dead nodes. We should expect small delays when consuming messages from the message queue, though the messages from "dead" nodes would be discarded.

Future Work

Performance To reduce the overheads of recovery, a more sophisticated strategy could be investigated in the context of multiple buddy locales, to promote load-balanced re-executions. A possibility would be to use load information during failure discovery and instruct the least loaded buddy locale to perform task adoption and recovery, thus introducing a dynamic re-configuration of primary buddy locales during program execution.

To reduce the initialisation costs, we could investigate the use of parallel loops for buddy allocation and remote data copying. Another optimization is the use of **zippered** parallel loops on the internal arrays that are used to persist guest data. Nevertheless, this optimization is only applicable to array blocks of the same size, as instructed by the design of **zippered** iterators.

To improve performance, we could look into an optimistic remote copying implementation based on health statistics, historical information on component upgrades, component life-expectancy, MTBF rates and other per node metrics. As such, we

could perform bulk data copying from nodes that are identified as more prone to fail. Alternatively, we could look into a strategy of bulk remote updates, based on a minimum threshold of individual element updates, or on frequent updates of a minimum block size, before initiating communication to the buddy locales.

Monitoring On the runtime level, we would be interested in disassembling the execution of local and recovery tasks within synchronised blocks. Currently, the *endCounts* mechanism; responsible for tracking parallel tasks, is based on a task counter to perform synchronisation (*join* operations). We would be interested in the use of separate counters for regular and recovery tasks, both as a debugging mechanism and as a further refinement of the communication layer implementation.

Tunability We would be interested in providing a user-tunable recovery strategy, to allow programmers to decide on whether the termination of the application execution is preferable compared to the projected costs of resilience. The mechanism could be designed as a configuration variable or an execution flag, setting a threshold for recovery, based for example on the maximum number of failures or the maximum number of recovery tasks that can execute.

Data distributions We have provided discussion of the main steps required to port our mechanism to other Chapel pre-defined data distributions, such as Cyclic and BlockCyclic. We would be interested in the implementation of the resilience mechanism for these data distributions and the evaluation of the associated costs for suitable applications. We note that the current implementation of the resilient runtime system does not pose any fundamental obstacles in implementing resilience within other pre-defined distributions.

Extensions and other directions We would be interested in integrations with a third-party system or a software/hardware component with health monitoring capabilities, to use co-operatively with the resilient runtime system. To this end, we could take advantage of the opportunities provided by our testing mechanism. For example, the *runtime-level hooks* could be extended to integrate a signalling mechanism to support node replacement from backup nodes. We could also integrate

functionality for performance counters via external libraries or tools. Furthermore, we could investigate alternative failure injection frameworks, with the ability to inject spatially distributed and time-stepped failures, to further test our resilient implementation.

To allow recovery of failed tasks from the failure point, and avoid re-execution of the entire computation, we could possibly look into combining in-memory checkpointing of the task state on the threading layer. Ideally, the checkpointed state information should be available across the system with the use of global references. We could also take advantage of our assumption of the failure-free root to persist this information.

Though, outside the scope of this work, and since the GASNet library remains to this point the most commonly used implementation of the communication layer in Chapel, another direction of future work would be the implementation of a relaxed failure policy for GASNet, possibly including the signals and signal handlers we introduced in Chapel’s communication layer in this work.

Regarding the aspect of *false positive locale failures* (which are considered failed in the current implementation), we could look into historical data of node response times and allow a threshold before launching recovery tasks on the adopting locale. Drawing from the above point, with a relaxed failure policy in GASNet in-place and assuming that message queues are not discarded during *idle locale time*, the statistical information on each node can be exchanged during application startup. In the event of a slow node that resumes communication, the status information on the node should be updated across locales.

In order to relax our *Task Atomicity* assumption of Section 4.2.1, a possible future work item would be to look into the detection of side-effect free computation, which can safely re-execute during failure recovery. Possible approaches may rely on static analysis or into adding annotations on language-level to identify *pure functions*.

Another possible enhancement would be to embed the configuration of the number of buddies, and potentially, the buddy placement, within Chapel applications. This would require compiler changes to provide support for a system configuration variable or an execution flag. The idea could also be extended to support predefined patterns of buddy locale placement, to support hierarchical locales, in the form of

predefined library modules.

Appendix A

Supportive Data

A.1 Combinations of failures

To produce the full set of combinations of locale Id's within the input range of target locales, we use the Python script of Listing A.1. In Figure A.1, below we provide, all the combinations of locale Id's for 8 participating locales when configured with 2 buddies per locale. We mark the combinations that do not comply with our initial assumptions:

- Combinations that include Locale 0;
- Combinations that include all the locale Id's of a buddy group.

The rest of the cases, comply to our assumption and can be recovered by the resilience mechanism.

```
#!/usr/bin/python
2 from itertools import chain, combinations
  localeIds = [0, 1, 2, ...] // input range
4
  def all_subsets(ss):
6     return chain(*map(lambda x: combinations(ss, x), range(0, len(ss)
+1)))
8 for subset in all_subsets(localeIds):
    print(localeIds)
```

Listing A.1: Python script to produce all the distinct combinations in an input range

(0.)	(0, 1)	(0, 1, 2)	(0, 1, 2, 3)	(0, 1, 2, 3, 4)	(0, 1, 2, 3, 4, 5)	(0, 1, 2, 3, 4, 5, 6)
(1.)	(0, 2)	(0, 1, 3)	(0, 1, 2, 4)	(0, 1, 2, 3, 5)	(0, 1, 2, 3, 4, 6)	(0, 1, 2, 3, 4, 5, 7)
(2.)	(0, 3)	(0, 1, 4)	(0, 1, 2, 5)	(0, 1, 2, 3, 6)	(0, 1, 2, 3, 4, 7)	(0, 1, 2, 3, 4, 6, 7)
(3.)	(0, 4)	(0, 1, 5)	(0, 1, 2, 6)	(0, 1, 2, 3, 7)	(0, 1, 2, 3, 5, 6)	(0, 1, 2, 3, 5, 6, 7)
(4.)	(0, 5)	(0, 1, 6)	(0, 1, 2, 7)	(0, 1, 2, 4, 5)	(0, 1, 2, 3, 5, 7)	(0, 1, 2, 4, 5, 6, 7)
(5.)	(0, 6)	(0, 1, 7)	(0, 1, 3, 4)	(0, 1, 2, 4, 6)	(0, 1, 2, 3, 6, 7)	(0, 1, 3, 4, 5, 6, 7)
(6.)	(0, 7)	(0, 2, 3)	(0, 1, 3, 5)	(0, 1, 2, 4, 7)	(0, 1, 2, 4, 5, 6)	(0, 2, 3, 4, 5, 6, 7)
(7.)	(1, 2)	(0, 2, 4)	(0, 1, 3, 6)	(0, 1, 2, 5, 6)	(0, 1, 2, 4, 5, 7)	(1, 2, 3, 4, 5, 6, 7)
	(1, 3)	(0, 2, 5)	(0, 1, 3, 7)	(0, 1, 2, 5, 7)	(0, 1, 2, 4, 6, 7)	(0, 1, 2, 3, 4, 5, 6, 7)
	(1, 4)	(0, 2, 6)	(0, 1, 4, 5)	(0, 1, 2, 6, 7)	(0, 1, 2, 5, 6, 7)	
	(1, 5)	(0, 2, 7)	(0, 1, 4, 6)	(0, 1, 3, 4, 5)	(0, 1, 3, 4, 5, 6)	
	(1, 6)	(0, 3, 4)	(0, 1, 4, 7)	(0, 1, 3, 4, 6)	(0, 1, 3, 4, 5, 7)	
	(1, 7)	(0, 3, 5)	(0, 1, 5, 6)	(0, 1, 3, 4, 7)	(0, 1, 3, 4, 6, 7)	
	(2, 3)	(0, 3, 6)	(0, 1, 5, 7)	(0, 1, 3, 5, 6)	(0, 1, 3, 5, 6, 7)	
	(2, 4)	(0, 3, 7)	(0, 1, 6, 7)	(0, 1, 3, 5, 7)	(0, 1, 4, 5, 6, 7)	
	(2, 5)	(0, 4, 5)	(0, 2, 3, 4)	(0, 1, 3, 6, 7)	(0, 2, 3, 4, 5, 6)	
	(2, 6)	(0, 4, 6)	(0, 2, 3, 5)	(0, 1, 4, 5, 6)	(0, 2, 3, 4, 5, 7)	
	(2, 7)	(0, 4, 7)	(0, 2, 3, 6)	(0, 1, 4, 5, 7)	(0, 2, 3, 4, 6, 7)	
	(3, 4)	(0, 5, 6)	(0, 2, 3, 7)	(0, 1, 4, 6, 7)	(0, 2, 3, 5, 6, 7)	
	(3, 5)	(0, 5, 7)	(0, 2, 4, 5)	(0, 1, 5, 6, 7)	(0, 2, 4, 5, 6, 7)	
	(3, 6)	(0, 6, 7)	(0, 2, 4, 6)	(0, 2, 3, 4, 5)	(0, 3, 4, 5, 6, 7)	
	(3, 7)	(1, 2, 3)	(0, 2, 4, 7)	(0, 2, 3, 4, 6)	(1, 2, 3, 4, 5, 6)	
	(4, 5)	(1, 2, 4)	(0, 2, 5, 6)	(0, 2, 3, 4, 7)	(1, 2, 3, 4, 5, 7)	
	(4, 6)	(1, 2, 5)	(0, 2, 5, 7)	(0, 2, 3, 5, 6)	(1, 2, 3, 4, 6, 7)	
	(4, 7)	(1, 2, 6)	(0, 2, 6, 7)	(0, 2, 3, 5, 7)	(1, 2, 3, 5, 6, 7)	
	(5, 6)	(1, 2, 7)	(0, 3, 4, 5)	(0, 2, 3, 6, 7)	(1, 2, 4, 5, 6, 7)	
	(5, 7)	(1, 3, 4)	(0, 3, 4, 6)	(0, 2, 4, 5, 6)	(1, 3, 4, 5, 6, 7)	
	(6, 7)	(1, 3, 5)	(0, 3, 4, 7)	(0, 2, 4, 5, 7)	(2, 3, 4, 5, 6, 7)	
		(1, 3, 6)	(0, 3, 5, 6)	(0, 2, 4, 6, 7)		
		(1, 3, 7)	(0, 3, 5, 7)	(0, 2, 5, 6, 7)		
		(1, 4, 5)	(0, 3, 6, 7)	(0, 3, 4, 5, 6)		
		(1, 4, 6)	(0, 4, 5, 6)	(0, 3, 4, 5, 7)		
		(1, 4, 7)	(0, 4, 5, 7)	(0, 3, 4, 6, 7)		
		(1, 5, 6)	(0, 4, 6, 7)	(0, 3, 5, 6, 7)		
		(1, 5, 7)	(0, 5, 6, 7)	(0, 4, 5, 6, 7)		
		(1, 6, 7)	(1, 2, 3, 4)	(1, 2, 3, 4, 5)		
		(2, 3, 4)	(1, 2, 3, 5)	(1, 2, 3, 4, 6)		
		(2, 3, 5)	(1, 2, 3, 6)	(1, 2, 3, 4, 7)		
		(2, 3, 6)	(1, 2, 3, 7)	(1, 2, 3, 5, 6)		
		(2, 3, 7)	(1, 2, 4, 5)	(1, 2, 3, 5, 7)		
		(2, 4, 5)	(1, 2, 4, 6)	(1, 2, 3, 6, 7)		
		(2, 4, 6)	(1, 2, 4, 7)	(1, 2, 4, 5, 6)		
		(2, 4, 7)	(1, 2, 5, 6)	(1, 2, 4, 5, 7)		
		(2, 5, 6)	(1, 2, 5, 7)	(1, 2, 4, 6, 7)		
		(2, 5, 7)	(1, 2, 6, 7)	(1, 2, 5, 6, 7)		
		(2, 6, 7)	(1, 3, 4, 5)	(1, 3, 4, 5, 6)		
		(3, 4, 5)	(1, 3, 4, 6)	(1, 3, 4, 5, 7)		
		(3, 4, 6)	(1, 3, 4, 7)	(1, 3, 4, 6, 7)		
		(3, 4, 7)	(1, 3, 5, 6)	(1, 3, 5, 6, 7)		
		(3, 5, 6)	(1, 3, 5, 7)	(1, 4, 5, 6, 7)		
		(3, 5, 7)	(1, 3, 6, 7)	(2, 3, 4, 5, 6)		
		(3, 6, 7)	(1, 4, 5, 6)	(2, 3, 4, 5, 7)		
		(4, 5, 6)	(1, 4, 5, 7)	(2, 3, 4, 6, 7)		
		(4, 5, 7)	(1, 4, 6, 7)	(2, 3, 5, 6, 7)		
		(4, 6, 7)	(1, 5, 6, 7)	(2, 4, 5, 6, 7)		
		(5, 6, 7)	(2, 3, 4, 5)	(3, 4, 5, 6, 7)		
			(2, 3, 4, 6)			
			(2, 3, 4, 7)			
			(2, 3, 5, 6)			
			(2, 3, 5, 7)			
			(2, 3, 6, 7)			
			(2, 4, 5, 6)			
			(2, 4, 5, 7)			
			(2, 4, 6, 7)			
			(2, 5, 6, 7)			
			(3, 4, 5, 6)			
			(3, 4, 5, 7)			
			(3, 4, 6, 7)			
			(3, 5, 6, 7)			
			(4, 5, 6, 7)			

Recoverable combinations
Combinations including Locale 0
Combinations including all locales in a buddy group

Figure A.1: Combinations of Locale Id's with a configuration of 8 target locales and 2 buddies per locale

A.2 Snapshot of the internal topology of a Beowulf node

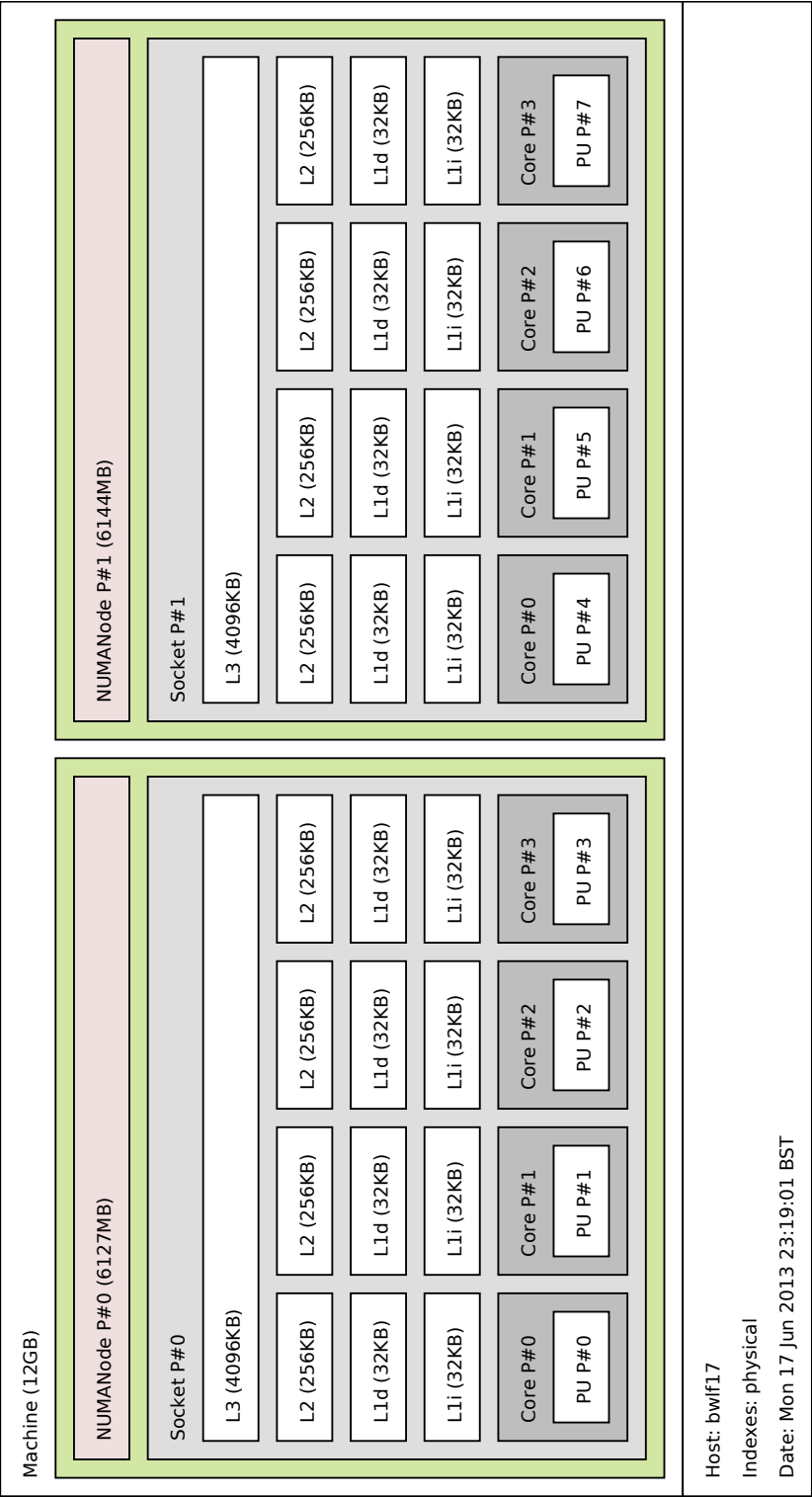


Figure A.2: Internal topology of a Beowulf node, produced with the hwloc package.

A.3 HDFS integration in Chapel

```

1 use HDFS;
2 // Connect to HDFS
3 var hdfs = hdfsChapelConnect("default", 0);
4
5 // Create one file per locale
6 var gfl = hdfs.hdfsOpen("afile.txt", iomode.r);
7
8 //...
9
10 // Get the local file of the current locale
11 var fl = gfl.getLocal();
12
13 // Create reader channels on the file and
14 // perform read operations
15 // ...
16
17 // Close the files and disconnect from HDFS
18
19 gfl.hdfsClose();
20 hdfs.hdfsChapelDisconnect();

```

Listing A.2: Example usage of the HDFS interface in Chapel.

A.4 Mandelbrot microbenchmark

A.4.1 Mandelbrot Chapel code

```

1 // mandelbrot: Resilient distributed data-parallel implementation
2
3 use MPlot;
4 use Time;
5 use BlockDistResMultiTaskRecovery;
6
7 //
8 // Dimensions of image file
9 //
10 config const rows = 2010,
11                                     cols = rows;
12
13 //
14 // Maximum number of steps to iterate
15 //
16 config const maxSteps = 50;
17
18 proc main() {
19
20     // added delay
21     var t= new Timer();
22     t.start();
23     while(t.elapsed() <20){} /
24     t.stop();
25
26     // timestamp
27     writeln("Chapel time: ", getCurrentTime());
28
29     //
30     // The set of indices over which the image is defined. Note that
31     // 0..#n means "a range with n indices starting at 0", i.e., 0..n-1
32     //
33     var LocImgSpace = {0..#rows, 0..#cols};
34     var ImgSpace = LocImgSpace dmapped Block(boundingBox=LocImgSpace);
35
36     //
37     // An array to store the resulting Image.
38     //
39     var Image: [ImgSpace] int;
40
41     //
42     // Compute the image, in parallel
43     //
44     for (i,j) in ImgSpace do
45         Image[i,j] = compute(i,j);
46
47     // timestamp
48     writeln("Chapel time: ", getCurrentTime());
49
50     //
51     // Plot the image
52     //
53     plot(Image);
54 }
55
56
```

```

58 // Compute the pixel value as described in the handout
59 //
60 proc compute(x, y) {
61     const c = mapImg2CPlane(x, y); // convert the pixel coordinates to a
62     complex value
63
64     var z: complex;
65     for i in 1..maxSteps {
66         z = z*z + c;
67         if (abs(z) > 2.0) then
68             return i;
69     }
70     return 0;
71 }
72
73 //
74 // Map an image coordinate to a point in the complex plane.
75 // Image coordinates are (row, col), with row 0 at the top.
76 //
77 proc mapImg2CPlane(row, col) {
78     const (rmin, rmax) = (-1.5, .5);
79     const (imin, imax) = (-1i, 1i);
80
81     return ((rmax - rmin) * col / cols + rmin) +
82           ((imin - imax) * row / rows + imax);
83 }

```

Listing A.3: Data-parallel Mandelbrot implementation, adjusted from Chapel's release to use the resilient version of the block data distribution. The initial micro-benchmark can be found under <https://github.com/chapel-lang/chapel/tree/master/test/exercises/Mandelbrot/solutions>.

A.4.2 Mandelbrot Result Fractal Sets

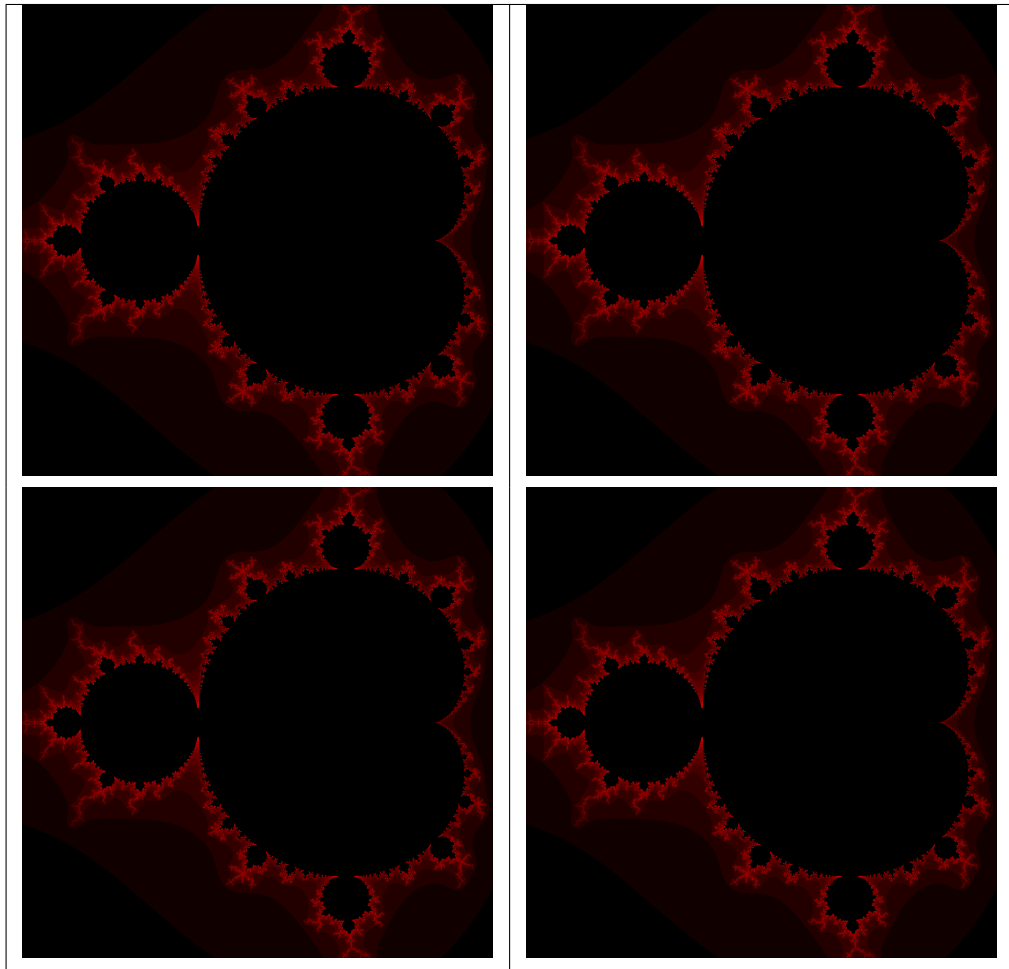


Table A.1: Mandelbrot fractal set output from execution on 4 locales without and with a single failure at different points in the execution. The results of the executions with failures (b, c, and d) have been validated for correctness using the Linux `diff` utility on the produced `.ppm` images.

A.5 STREAM: Historical measurements

Table A.2 lists the memory bandwidth as measured using the STREAM micro-benchmark over a period of six years, focusing on Intel processors which have undergone a number of changes. The total memory bandwidth measured found is with the use of all the cores on the node. The Raspberry Pi results provide a comparison on the lower end of memory bandwidth scale.

Processor	No. of cores/node (sockets)	Total Memory BW(GBps)	Memory BW/core (GBps)
Harpertown (2007)	8 (4)	7.2	0.9
Harpertown (2007)	8 (4)	32	4
Nehalem-EP (2009)	12 (6)	42	3.5
Westmere-EP (2010)	8 (4)	42	5.25
Sandy Bridge EP (2012)	16 (8)	78	4.88
Sandy Bridge EP (2012)	12 (6)	78	6.5
Sandy Bridge EP (2012)	8 (4)	78	9.75
Ivy Bridge EP (2013)	24 (12)	101	4.21
Ivy Bridge EP (2013)	20 (10)	101	5.05
Ivy Bridge EP (2013)	16 (8)	101	6.31
Ivy Bridge EP (2013)	12 (6)	101	8.42
Haswell EP (guess)	32 (16)	120	3.75
Haswell EP (guess)	24 (12)	120	5
Raspberry Pi v1	1	0.25	0.25
Raspberry Pi v2	1	0.26	0.26

Table A.2: STREAM triad Memory Bandwidth Results Layton, 2002

Appendix B

Benchmarks and Data Sets

B.1 Microbenchmarks

```
1 module MonteCarlo{
2
3   proc monteCarlo(){
4     const n = 100000000,      // number of random points to
5       try
6         seed = 589494289;    // seed for random number
7         generator
8
9     var rs = new RandomStream(seed, parSafe=false);
10    var count = 0;
11    for i in 1..n do
12      if (rs.getNext()*2 + rs.getNext()*2) <= 1.0 then
13        count += 1;
14      var pi = count * 4.0 / n;
15      writeln("pi=", pi, " on locale ", here.id);
16    delete rs;
17  }
```

Listing B.1: MonteCarlo module

B.1.1 Serial distributed micro-benchmarks

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  on Locales[1] do {
12    monteCarlo();
13  }
14  on Locales[2] do{
15    monteCarlo();
16  }
17  total.stop();
18  writeln("Total elapsed:", total.elapsed());
19 }

```

Listing B.2: Serial simpleons.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   var total = new Timer();
8   total.start();
9
10  on Locales[1] do {
11    monteCarlo();
12    on Locales[2] do{
13      monteCarlo();
14    }
15  }
16  total.stop();
17  writeln("Total elapsed:", total.elapsed());
18 }

```

Listing B.3: Serial simpleontest.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   var total = new Timer();
8   total.start();
9
10  on Locales[1] do {
11    monteCarlo();
12    on Locales[2] do{
13      monteCarlo();
14      on Locales[1] do {
15        monteCarlo();
16      }
17    }
18  }
19  total.stop();
20  writeln("Total elapsed:", total.elapsed());
21 }

```

Listing B.4: Serial back.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   var total = new Timer();
8   total.start();
9
10  on Locales[1] do {
11    monteCarlo();
12    on Locales[2] do{
13      monteCarlo();
14      on Locales[3] do{
15        monteCarlo();
16      }
17    }
18  }
19  total.stop();
20  writeln("Total elapsed:", total.elapsed());
21 }

```

Listing B.5: Serial three.on.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   var total = new Timer();
8   total.start();
9
10  on Locales[1] do {
11    monteCarlo();
12    on Locales[2] do{
13      monteCarlo();
14    }
15  }
16  on Locales[3] do{
17    monteCarlo();
18    on Locales[2] do{
19      monteCarlo();
20    }
21  }
22  total.stop();
23  writeln("Total elapsed:", total.elapsed());
24 }

```

Listing B.6: Serial two_two_on.chpl

B.1.2 Task parallel micro-benchmarks: begin+on

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  on Locales[1] do begin {
12    monteCarlo();
13  }
14  on Locales[2] do begin {
15    monteCarlo();
16  }
17  total.stop();
18  writeln("Total elapsed:", total.elapsed());
19 }

```

Listing B.7: Task parallel begin+on simpleons.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  on Locales[1] do begin{
12    monteCarlo();
13    on Locales[2] do begin{
14      monteCarlo();
15    }
16  }
17  total.stop();
18  writeln("Total elapsed:", total.elapsed());
19 }

```

Listing B.8: Task parallel begin+on simpleontest.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  on Locales[1] do begin {
12    monteCarlo();
13    on Locales[2] do begin {
14      monteCarlo();
15      on Locales[1] do begin {
16        monteCarlo();
17      }
18    }
19  }
20  total.stop();
21  writeln("Total elapsed:", total.elapsed());
22 }

```

Listing B.9: Task parallel `begin+on` back.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  on Locales[1] do begin {
12    monteCarlo();
13    on Locales[2] do begin {
14      monteCarlo();
15      on Locales[3] do begin {
16        monteCarlo();
17      }
18    }
19  }
20  total.stop();
21  writeln("Total elapsed:", total.elapsed());
22 }

```

Listing B.10: Task parallel `begin+on` three_on.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  on Locales[1] do begin {
12    monteCarlo();
13    on Locales[2] do begin {
14      monteCarlo();
15    }
16  }
17  on Locales[3] do begin {
18    monteCarlo();
19    on Locales[2] do begin {
20      monteCarlo();
21    }
22  }
23  total.stop();
24  writeln("Total elapsed:", total.elapsed());
25 }

```

Listing B.11: Task parallel `begin+on` two_two_on.chpl

B.1.3 Task parallel micro-benchmarks: cobegin+on

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  cobegin{
12    on Locales[1] do {
13      monteCarlo();
14    }
15    on Locales[2] do {
16      monteCarlo();
17    }
18  }
19  total.stop();
20  writeln("Total elapsed:", total.elapsed());
21 }

```

Listing B.12: Task parallel cobegin+on simpleons.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  on Locales[1] do {
12    cobegin{
13      monteCarlo();
14      on Locales[2] do {
15        monteCarlo();
16      }
17    }
18  }
19  total.stop();
20  writeln("Total elapsed:", total.elapsed());
21 }

```

Listing B.13: Task parallel cobegin+on simpleontest.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10  on Locales[1] do {
11    cobegin{
12      monteCarlo();
13      on Locales[2] do {
14        cobegin{
15          monteCarlo();
16          on Locales[1] do {
17            monteCarlo();
18          }
19        }
20      }
21    }
22  }
23  total.stop();
24  writeln("Total elapsed:", total.elapsed());
25 }

```

Listing B.14: Task parallel cobegin+on back.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  on Locales[1] do {
12    cobegin{
13      monteCarlo();
14      on Locales[2] do {
15        cobegin{
16          monteCarlo();
17          on Locales[3] do {
18            monteCarlo();
19          }
20        }
21      }
22    }
23  }
24  total.stop();
25  writeln("Total elapsed:", total.elapsed());
26 }

```

Listing B.15: Task parallel cobegin+on three.on.chpl

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  cobegin{
12    on Locales[1] do {
13      monteCarlo();
14      on Locales[2] do {
15        monteCarlo();
16      }
17    }
18    on Locales[3] do {
19      monteCarlo();
20      on Locales[2] do {
21        monteCarlo();
22      }
23    }
24  }
25  total.stop();
26  writeln("Total elapsed:", total.elapsed());
27 }

```

Listing B.16: Task parallel cobegin+on two_two_on.chpl

B.1.4 Task parallel micro-benchmarks: coforall+on

```

1 use Time, Random, MonteCarlo;
2
3 proc main() {
4   var t = new Timer();
5   t.start();
6   while(t.elapsed() < 20){}
7   t.stop();
8   var total = new Timer();
9   total.start();
10
11  coforall loc in Locales{
12    on loc do
13      monteCarlo();
14    }
15  total.stop();
16  writeln("Total elapsed:", total.elapsed());
17 }

```

Listing B.17: Task parallel coforall+on coforall.chpl

B.2 Data parallel micro-benchmarks

B.2.1 STREAM triad

```

1 //
2 // Use standard modules for Block distributions, Timing
  routines, Type
3 // utility functions, and Random numbers
4 //
5 use BlockDistResMultiTaskRecovery, Time, Types, Random;
6
7 //
8 // Use shared user module for computing HPCC problem sizes
9 //
10 use HPCCProblemSize;
11
12 //
13 // The number of vectors and element type of those vectors
14 //
15 const numVectors = 3;
16 type elemType = real(64);
17
18 //
19 // Configuration constants to set the problem size (m) and
  the scalar
20 // multiplier, alpha
21 //
22 config const m = 400,
23           alpha = 3.0,
24           printDbg = false;
25
26 //
27 // Configuration constants to set the number of trials to
  run and the
28 // amount of error to permit in the verification
29 //
30 config const numTrials = 1,
31           epsilon = 0.0;
32
33 //
34 // Configuration constants to indicate whether or not to use
  a
35 // pseudo-random seed (based on the clock) or a fixed seed;
  and to
36 // specify the fixed seed explicitly
37 //
38 config const useRandomSeed = true,
39           seed = if useRandomSeed then
40               SeedGenerator.currentTime else 314159265;
41
42 //
43 // Configuration constants

```

```

44 //
45 config const printParams = true ,
46           printArrays = false ,
47           printStats = true;
48
49 //
50 // The program entry point
51 //
52 proc main() {
53     printConfiguration(); // print the configuration
54
55     var t, tt = new Timer();
56     t.start();
57     while(t.elapsed() <20){}
58     t.stop();
59
60     //
61     // ProblemSpace describes the index set for the three
        vectors. It
62     // is a 1D domain that is distributed according to a Block
63     // distribution. In this case, the Block distribution is
        1D
64     // distribution computed by blocking the bounding box 1..m
        across
65     // the set of locales. The ProblemSpace domain also
        contains the
66     // indices 1..m.
67     //
68     const MonoSpace = {1..m};
69     var MonoLocaleView={0..#numLocales};
70     var MyMonoLocales:[MonoLocaleView] locale = reshape(
        Locales , MonoLocaleView);
71     const ProblemSpace => MonoSpace dmapped Block(boundingBox=
        MonoSpace, targetLocales=MyMonoLocales);
72
73
74     //
75     // A, B, and C are the three distributed vectors, declared
        to store
76     // a variable of type elemType for each index in
        ProblemSpace.
77     //
78     var A, B, C: [ProblemSpace] elemType;
79     initVectors(B, C);
80
81     var execTime: [1..numTrials] real; // an array of
        timings
82
83     for trial in 1..numTrials { // loop over the trials
84         const startTime = getCurrentTime(); // capture the start
            time
85         if(printDbg) then
86             writeln("Trial #", trial);
87
88         //

```

```

89      // The main loop: Iterate over the vectors A, B, and C
      // in a
90      // parallel, zippered manner storing the elements as a,
      // b, and c.
91      // Compute the multiply-add on b and c, storing the
      // result to a.
92      //
93
94      forall (a, b, c) in zip(A, B, C) do {
95      a = b + alpha * c;
96      }
97      execTime(trial) = getCurrentTime() - startTime; //
      // store the elapsed time
98  }
99
100  if(printDbg) then
101    printTimes(execTime);
102
103  var v = verifyResults(A, B, C);
104  writeln("Verification = ", v, "\n");
105 }
106
107 proc prettyPrint(space, A){
108
109   var count =1;
110   for i in space do{
111     if (i<count*1000000) then{
112       write( A[i], " ");
113     }else{
114       write("    on ", A[i].locale.id, "\n\n");
115       count=count+1;
116     }
117   }
118 }
119
120 //
121 // Print the problem size and number of trials
122 //
123 proc printConfiguration() {
124   if (printParams) {
125     if (printStats) then
126       printLocalesTasks();
127     printProblemSize(elemType, numVectors, m);
128     writeln("Number of trials = ", numTrials, "\n");
129   }
130 }
131
132 //
133 // Initialize vectors B and C using a random stream of
      // values and
134 // optionally print them to the console
135 //
136 proc initVectors(B, C) {
137   var randlist = new RandomStream(seed);

```



```

138
139 randlist.fillRandom2(B);
140 randlist.fillRandom2(C);
141
142 if (printArrays) {
143     writeln("B is: ", B);
144     writeln("C is: ", C );
145
146     for b in B do
147         writeln(b.locale.id);
148
149     for c in C do
150         writeln(c.locale.id);
151 }
152 delete randlist;
153 }
154
155 //
156 // Verify that the computation is correct
157 //
158 proc verifyResults(A, B, C) {
159     if (printArrays) then
160         writeln("A is: ", A, "\n"); // optionally print A
161
162     //
163     // recompute the computation, destructively storing into B
164     // to save space
165     forall (b, c) in zip(B, C) do
166         b += alpha * c;
167
168     if (printArrays) then
169         writeln("A-hat is: ", B, "\n"); // optionally print A-
170         hat
171     //
172     // Compute the infinity-norm by computing the maximum
173     // reduction of
174     // the absolute value of A's elements minus the new result
175     // computed in B. "[i in I]" represents an expression-
176     // level
177     // loop: "forall i in I"
178     //
179     const infNorm = max reduce [(a,b) in zip(A,B)] abs(a - b);
180
181     return (infNorm <= epsilon); // return whether the
182     error is acceptable
183 }
184
185 //
186 // Print out success/failure, the timings, and the GB/s
187 // value
188 //
189 proc printResults(successful, execTimes) {

```

```

186  writeln("Validation: ", if successful then "SUCCESS" else
    "FAILURE");
187  if (printStats) {
188      const totalTime = + reduce execTimes,
189          avgTime = totalTime / numTrials,
190          minTime = min reduce execTimes;
191      writeln("Execution time:");
192      writeln("  tot = ", totalTime);
193      writeln("  avg = ", avgTime);
194      writeln("  min = ", minTime);
195
196      const GBPerSec = numVectors * numBytes(elemType) * (m /
          minTime) * 1e-9;
197      writeln("Performance (GB/s) = ", GBPerSec);
198  }
199 }
200
201 proc printTimes(execTimes) {
202     if (printStats) {
203         const totalTime = + reduce execTimes,
204             avgTime = totalTime / numTrials,
205             minTime = min reduce execTimes;
206         writeln("Execution time:");
207         writeln("  tot = ", totalTime);
208         writeln("  avg = ", avgTime);
209         writeln("  min = ", minTime);
210
211         const GBPerSec = numVectors * numBytes(elemType) * (m /
            minTime) * 1e-9;
212         writeln("Performance (GB/s) = ", GBPerSec);
213     }
214 }

```

Listing B.18: Data parallel STREAM triad source code.

B.2.2 N-body all-pairs

```

1 //
2 // Use the resilience enabled data block distribution module
3 //
4 use BlockDistResMultiTaskRecovery;
5
6 //
7 // Use the module where bodies datasets are defined
8 //
9 use Morebodies;
10
11 //
12 // The number of timesteps to simulate
13 //
14 config const n = 10000;
15
16 //
17 // The number of bodies to be simulated
18 //
19 const numbodies = bodies.numElements;
20
21 //
22 // The declaration of the distribution domain
23 //
24
25 const MonoSpace = {1..#numbodies};
26 var MonoLocaleView={0..#numLocales};
27 var MyMonoLocales:[MonoLocaleView] locale =
28     reshape(Locales, MonoLocaleView);
29 const ProblemSpace => MonoSpace dmapped
30     Block(boundingBox=MonoSpace, targetLocales=MyMonoLocales
31         );
32 //
33 // The declaration of the distributed arrays
34 //
35 var A: [ProblemSpace] body;
36 var C: [MonoSpace] body =bodies;
37
38 //
39 // The computation involves initializing the sun's velocity,
40 // writing the initial energy, advancing the system through
41 // 'n'
42 // timesteps, and writing the final energy.
43 //
44 proc main() {
45 //
46 // Add some delay
47 //
48     var t, total = new Timer();

```

```

49  t.start();
50  while(t.elapsed() <20){}
51  t.stop();
52
53  //
54  // Initialisation of the distributed arrays
55  //
56  initArrays();
57  initSun();
58
59  total.start();
60  //
61  // Print the original energy of the system and the
    configuration
62  //
63  writef("%.9r\n", energy());
64  for 1..n do
65      advance(0.01);
66  var finalEnergy = energy();
67  writef("%.9r\n", finalEnergy);
68
69  total.stop();
70  verifyResults(finalEnergy);
71
72  writeln("Configuration: numbodies= ", numbodies, " n=", n)
    ;
73  writeln("Total elapsed: ", total.elapsed(), "\n\n");
74 }
75
76 //
77 // Compute the sun's initial velocity
78 //
79 proc initSun() {
80     const p = + reduce (for b in bodies do (b.v * b.mass));
81     bodies[1].v = -p / solarMass;
82 }
83
84 proc initArrays() {
85     for i in 1..numbodies {
86         A[i] = bodies[i];
87     }
88     if debug then printArrays();
89 }
90
91 //
92 // Advance the positions and velocities of all the bodies
93 //
94 proc advance(dt) {
95
96     forall i in 1..numbodies {
97         forall j in i+1..numbodies {
98             ref b1 = A[i],
99             b2 = A[j];
100

```

```

101      const dpos = b1.pos - b2.pos ,
102             mag = dt / sqrt(sumOfSquares(dpos))**3;
103
104      b1.v -= dpos * b2.mass * mag;
105      b2.v += dpos * b1.mass * mag;
106  }
107 }
108
109 for b in A do
110     b.pos += dt * b.v;
111 }
112
113 //
114 // Compute the energy of the bodies
115 //
116 proc energy() {
117     var e = 0.0;
118
119     for i in 1..numbodies {
120         const b1 = A[i];
121
122         e += 0.5 * b1.mass * sumOfSquares(b1.v);
123
124         for j in i+1..numbodies {
125             const b2 = A[j];
126
127             e -= (b1.mass * b2.mass) / sqrt(sumOfSquares(b1.pos -
128                 b2.pos));
129         }
130     }
131     return e;
132 }
133
134 //
135 // Verify the final result
136 //
137 proc verifyResults(finalEnergy) {
138
139     writeln("%.9r\n", energyC());
140     for 1..n do {
141
142         for i in 1..numbodies {
143             for j in i+1..numbodies {
144                 ref b1 = C[i],
145                 b2 = C[j];
146
147                 const dpos = b1.pos - b2.pos ,
148                        mag = 0.01 / sqrt(sumOfSquares(dpos))**3;
149
150                 b1.v -= dpos * b2.mass * mag;
151                 b2.v += dpos * b1.mass * mag;
152             }
153         }
154     }

```

```

155     for b in C do
156         b.pos += 0.01 * b.v;
157     }
158     var xx = energyC();
159     writef("%.9r\n", xx);
160
161     if(xx == finalEnergy) then
162         writeln("Verify: Success ");
163     else
164         writeln("Verify: Failed ");
165 }
166
167 //
168 // Compute the energy of the bodies
169 //
170 proc energyC() {
171     var e = 0.0;
172
173     for i in 1..numbodies {
174         const b1 = C[i];
175
176         e += 0.5 * b1.mass * sumOfSquares(b1.v);
177
178         for j in i+1..numbodies {
179             const b2 = C[j];
180
181             e -= (b1.mass * b2.mass) / sqrt(sumOfSquares(b1.pos -
182                 b2.pos));
183         }
184     }
185     return e;
186 }
187
188 //
189 // A helper to compute the sum of squares of a 3-tuple's
190 // components
191 //
192 inline proc sumOfSquares(x)
193     return x(1)**2 + x(2)**2 + x(3)**2;

```

Listing B.19: Data parallel Nbody source code.

B.3 Celestial bodies input datasets

Below we provide the bodies datasets used in the experiments for the N-body all-pairs algorithm.

	P _{ox}	P _{osy}	P _{osz}	V _x *daysPerYear	V _y *daysPerYear	V _z *daysPerYear	Mass*solarMass
1	0	0	0	0	0	0	1
2	4.84143144246472090e+00	-1.16032004402742839e+00	-1.03622044471123109e-01	1.66007664274403694e-03	7.69901118419740425e-03	-6.90460016972063023e-05	9.54791938424326609e-04
3	8.34336671824457987e+00	4.12479856412430479e+00	-4.03523417114321381e-01	-2.76742510726862411e-03	4.99852801234917238e-03	2.30417297573763929e-05	2.85885980666130812e-04
4	1.28943695621391310e+01	-1.51111514016986312e+01	-2.23307578892655734e-01	2.96460137564761618e-03	2.37847173959480950e-03	-2.96589568540237556e-05	4.36624404335156298e-05
5	1.53796971148509165e+01	-2.59193146099879641e+01	1.79258772950371181e-01	2.68067772490389322e-03	1.62824170038242295e-03	-9.51592254519715870e-05	5.15138902046611451e-05
6	-5.996130	7.365330	-0.431370	5.309727	-5.760521	-5.106896	8.931437
7	9.696543	5.340199	1.281590	9.319191	5.530926	5.281424	5.454451
8	4.449270	-7.322360	1.748141	8.405575	6.757387	-3.780701	8.708323
9	-4.857735	5.615505	-6.154079	-0.959717	7.221056	9.439165	2.294621
10	9.708160	2.828011	-0.427657	8.265324	-7.198768	1.337771	1.761434
11	7.561939	3.018993	-0.557335	8.242748	7.488066	4.161242	7.211702
12	7.838152	4.015871	4.578490	9.432579	7.258484	-1.640808	9.275744
13	-2.338464	-0.491522	6.132675	8.621252	2.287422	6.693512	3.673369
14	-3.517514	5.605034	6.461754	-1.782839	2.803802	-4.876018	9.978596
15	-2.934857	-6.321287	-7.405713	-1.177606	-3.809353	-1.566956	3.965903
16	2.193786	-8.114434	-4.943103	2.761208	-5.372918	6.697706	2.036953
17	2.821056	7.285711	-6.897996	5.022684	-7.672948	8.466814	5.836308
18	-1.064412	-9.127116	6.493452	7.512236	8.458475	9.879862	3.211262
19	-8.700751	-0.117617	5.459934	9.367001	6.023586	3.113340	4.708029
20	9.252983	-5.499310	7.807762	8.374489	8.264580	4.752002	8.738186

	P _{ox}	P _{osy}	P _{osz}	V _x * daysPerYear	V _y * daysPerYear	V _z * daysPerYear	Mass * solarMass
1	0	0	0	0	0	0	1
2	4.84143144246472090e+00	-1.16032004402742839e+00	-1.03622044471123109e-01	1.66007664274403694e-03	7.699011118419740425e-03	-6.90460016972063023e-05	9.54791938424326609e-04
3	8.3436671824457987e+00	4.12479856412430479e+00	-4.03523417114321381e-01	-2.76742510726862411e-03	4.99852801234917238e-03	2.30417297573763929e-05	2.85885980666130812e-04
4	1.28943695621391310e+01	-1.51111514016986312e+01	-2.23307578892655734e-01	2.96460137564761618e-03	2.37847173959480950e-03	-2.96589568540237556e-05	4.36624404335156298e-05
5	1.53796971148509165e+01	-2.59193146099879641e+01	1.79258772950371181e-01	2.68067772490389322e-03	1.62824170038242295e-03	-9.51592254519715870e-05	5.15138902046611451e-05
6	-5.996130	7.365330	-0.431370	5.309727	-5.760521	-5.106896	8.931437
7	9.696543	5.340199	1.281590	9.319191	5.530926	5.281424	5.454451
8	4.449270	-7.322360	1.748141	8.405575	6.757387	-3.780701	8.708323
9	-4.857735	5.615505	-6.154079	-0.959717	7.221056	9.439165	2.294621
10	9.708160	2.828011	-0.427657	8.265324	-7.198768	1.337771	1.761434
11	7.561939	3.018993	-0.557335	8.242748	7.488066	4.161242	7.211702
12	7.838152	4.015871	4.578490	9.432579	7.258484	-1.640808	9.275744
13	-2.338464	-0.491522	6.132675	8.621252	2.287422	6.693512	3.673369
14	-3.517514	5.605034	6.461754	-1.782839	2.803802	-4.876018	9.978596
15	-2.934857	-6.321287	-7.405713	-1.177606	-3.809353	-1.566956	3.965903
16	2.193786	-8.114434	-4.943103	2.761208	-5.372918	6.697706	2.036953
17	2.821056	7.285711	-6.897996	5.022684	-7.672948	8.466814	5.836308
18	-1.064412	-9.127116	6.493452	7.512236	8.458475	9.879862	3.211262
19	-8.700751	-0.117617	5.459934	9.367001	6.023586	3.113340	4.708029
20	9.252983	-5.499310	7.807762	8.374489	8.264580	4.752002	8.738186
21	-1.501240	0.085436	-1.279516	-2.131567	-7.034532	4.00093	-0.114315
22	-2.808461	0.090542	-0.196506	-2.308134	-5.657036	2.817179	-0.083894
23	-2.265666	0.057676	-2.609833	-2.998204	-1.179703	2.922209	-0.164753
24	-1.338059	0.054886	-2.063638	-2.430338	-10.018402	3.381674	-0.260204
25	-2.471469	0.040688	-1.654777	-2.463079	-3.918562	4.010200	-0.168750
26	-3.651031	0.095394	-2.426275	-0.904886	-12.890277	3.584466	-0.012760
27	-1.182438	0.056138	-0.828468	-2.137824	-2.072036	0.599011	-0.229874
28	-1.083913	0.085883	-0.382518	-1.501623	-6.115369	3.343080	-0.093931
29	-0.377924	0.064018	-0.035444	-1.766623	-0.705942	2.309968	-0.127547
30	-1.541501	0.054150	-2.539693	-0.1119370	-7.431612	2.686822	-0.1115642
31	-1.773963	0.052958	-1.100589	-1.832969	-1.364395	2.636378	-0.098315
32	-1.145477	0.078912	-2.813053	-1.382221	-9.719300	0.267281	-0.325595
33	-0.278223	0.088058	-0.595940	-0.406811	-7.811395	0.862891	-0.245388
34	-2.839112	0.077387	-0.196584	-2.715980	-4.730591	3.739578	-0.320120
35	-4.054068	0.056741	-0.735103	-0.513379	-1.454963	2.508780	-0.264910
36	-0.939786	0.025492	-0.113363	-1.292863	-0.660535	3.999435	-0.302026
37	-3.459945	0.004066	-2.379608	-2.317664	-13.818573	4.066582	-0.307602
38	-2.209989	0.020231	-1.739912	-0.030443	-14.642679	2.646544	-0.310080
39	-1.457757	0.055759	-2.401717	-0.317331	-1.875087	0.186987	-0.093790
40	-1.110016	0.065750	-0.074573	-1.270877	-13.686308	0.256845	-0.289455

References

- Aggarwal, Monika and Sumit Aggarwal (2010). “Dynamic load balancing based on CPU utilization and data locality in distributed database using priority policy”. In: *2010 2nd International Conference on Software Technology and Engineering*. Vol. 2, pp. V2–388–V2–391. DOI: 10.1109/ICSTE.2010.5608781.
- Andrews, Gregory R. and Ronald A Olsson (1993). *The SR Programming Language: Concurrency in Practice*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc. ISBN: 0-8053-0088-0.
- Armstrong, Joe (2007). *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf. ISBN: 193435600X, 9781934356005.
- Avižienis, Algirdas, Per Gunningberg, John PJ Kelly, Lorenzo Strigini, Pascal Traverse, Kam Sing Tso, and Udo Voges (1985). “The UCLA DEDIX system: A Distributed Testbed for Multiple-version Software”. In: *Digest of 15th Annual International Symposium on Fault-Tolerant Computing*, pp. 126–134.
- Avižienis, Algirdas, Jean-Claude Laprie, Brian Randell, and Carl Landwehr (2004). “Basic Concepts and Taxonomy of Dependable and Secure Computing”. In: *IEEE Trans. Dependable Secur. Comput.* 1.1, pp. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.
- Avižienis, Algirdas, Michael RT Lyu, Werner Schütz, Kam Sing Tso, and Udo Voges (1988). “DEDIX 87A Supervisory System For Design Diversity Experiments at UCLA”. In: *Software Diversity In Computerized Control Systems*. Springer, pp. 129–168.
- Balaji, Pavan (2015). *Programming models for parallel computing*. MIT Press. ISBN: 0262528819.
- Barney, Blaise (2017). *OpenMP*. URL: <https://computing.llnl.gov/tutorials/openMP/> (visited on 03/2018).

- Benítez-Pérez, Héctor, Golam R Latif-Shabgahi, Thompson Haydn, Stuart Bennett, Peter J Fleming, and Julian M Bass (1999). “Integration and Comparison of FDI and Fault Masking Features in Embedded Systems”. In: *IFAC Proceedings Volumes* 32.2, pp. 7712–7717. ISSN: 1474-6670. DOI: 1474-6670(17)57316-4.
- Bernheim, Laura (2007). “Powerful and Productive Parallel Programming With Chapel: Building an Open-Source Language That Scales From Laptops to Supercomputers”. In: *HostingAdvice.com [Blog]*. URL: <https://www.hostingadvice.com/blog/powerful-and-productive-parallel-programming-with-chapel/> (visited on 02/2010).
- Bonachea, Dan (2002). *GASNet Specification, V1.1*. Tech. rep. Berkeley, CA, USA.
- Borthakur, Dhruba (2008). “HDFS architecture guide”. In: *Hadoop Apache Project* 53, pp. 1–13.
- Bortolotti, Daniela, Angelo Carbone, Domenico Galli, Ignazio Lax, Umberto Marconi, Gianluca Peco, Stefano Perazzini, Vincenzo Maria Vagnoni, and Maria Zangoli (2011). “Comparison of UDP Transmission Performance Between IP-Over-InfiniBand and 10-Gigabit Ethernet”. In: *IEEE Transactions on Nuclear Science* 58.4, pp. 1606–1612.
- Bosilca, George, Aurelien Bouteiller, Amina Guermouche, Thomas Herault, Yves Robert, Pierre Sens, and Jack Dongarra (2016). “Failure detection and propagation in HPC systems”. In: *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, pp. 312–322.
- Bouguerra, Mohamed Slim, Ana Gainaru, and Franck Cappello (2013). “Failure prediction: what to do with unpredicted failures”. In: *28th IEEE international parallel and distributed processing symposium*. Vol. 2.
- Brewer, Eric A. (2000). “Towards Robust Distributed Systems”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC ’00. Portland, Oregon, USA: ACM, pp. 7–. ISBN: 1-58113-183-6. DOI: 10.1145/343477.343502.
- Brooks III, Eugene D, Brent C Gorda, and Karen H Warren (1992). “The parallel C preprocessor”. In: *Scientific Programming* 1.1, pp. 79–89.

- Brown, Aaron B (2003). *A recovery-oriented approach to dependable services: repairing past errors with system-wide undo*. Tech. rep. California University Berkeley, Dept. Of Electrical Engineering And Computer Sciences.
- Brown, Aaron B and David A Patterson (2002). “Rewind, Repair, Replay: Three R’s to Dependability”. In: *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, pp. 70–77. DOI: 10.1145/1133373.1133387.
- Cai, Jian and Aviral Shrivastava (2016). “Software Coherence Management on Non-Coherent Cache Multi-cores”. In: *Proceedings of 29th International Conference on VLSI Design (VLSID)*. (Best Student Paper Award).
- Chamberlain, B.L., D. Callahan, and H.P. Zima (2007). “Parallel Programmability and the Chapel Language”. In: *Int. J. High Perform. Comput. Appl.* 21.3, pp. 291–312. ISSN: 1094-3420. DOI: 10.1177/1094342007078442.
- Chamberlain, Bradford L., Sung-eun Choi, Steven J. Deitz, and Angeles Navarro (2011). “User-Defined Parallel Zippered Iterators in Chapel”. In: *Proc. PGAS 2011: Fifth Conference in Partitioned Global Address Space Programming Models*.
- Chamberlain, Bradford L., Steven J. Deitz, David Iten, and Sung-Eun Choi (2010). “User-defined Distributions and Layouts in Chapel: Philosophy and Framework”. In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*. HotPar’10. Berkeley, CA: USENIX Association, pp. 12–12. URL: <http://dl.acm.org/citation.cfm?id=1863086.1863098>.
- Chen, Liming and Algirdas Avizienis (1978). “N-version Programming: A Fault-tolerance Approach to Reliability of Software Operation”. In: *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, pp. 3–9.
- Chigurupati, Asha, Romain Thibaux, and Noah Lassar (2016). “Predicting Hardware Failure Using Machine Learning”. In: *Reliability and Maintainability Symposium (RAMS), 2016 Annual*. IEEE, pp. 1–6.
- Cochran, W. T., J. W. Cooley, D. L. Favon, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch (1967). “What is the fast Fourier transform?” In: *Proceedings of the IEEE* 55.10, pp. 1664–1674. DOI: 10.1109/PROC.1967.5957.

- Cray Inc (2015a). *Chapel*. URL: <https://github.com/chapel-lang/chapel>.
- (2015b). *Chapel specification v 0.98*. Tech. rep., pp. 208–209. URL: <https://chapel-lang.org/spec/spec-0.98.pdf>.
- Cristian, Flavin (1991). “Understanding Fault-tolerant Distributed Systems”. In: *Communications of the ACM* 34.2, pp. 56–78.
- Culler, David E, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick (1993). “Parallel Programming in Split-C”. In: *Supercomputing’93. Proceedings*. IEEE, pp. 262–273.
- Cunningham, David, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu (2014). “Resilient X10: Efficient Failure-Aware Programming”. In: *ACM SIGPLAN Notices*. Vol. 49. 8. ACM, pp. 67–80.
- Dahl, Ole-Johan, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare (1972). *Structured programming*. Academic Press Ltd.
- Daintith, John (2004). *A Dictionary of Computing*. Vol. 5. Oxford Paperbacks. Oxford University Press. ISBN: 9780198608776.
- Dam, Hubertus J. J. van, Abhinav Vishnu, and Wibe A. de Jong (2011). “Designing a Scalable Fault Tolerance Model for High Performance Computational Chemistry: A Case Study with Coupled Cluster Perturbative Triples”. In: *Journal of Chemical Theory and Computation* 7.1. PMID: 26606219, pp. 66–75. DOI: 10.1021/ct100439u.
- DeBardeleben, Nathan, James Laros, John T Daly, Stephen L Scott, Christian Engelmann, and Bill Harrod (2009). “High-end Computing Resilience: Analysis of issues facing the HEC community and path-forward for research and development”. In: *Whitepaper, Dec*.
- Di, Sheng, Hanqi Guo, Eric Pershey, Marc Snir, and Franck Cappello (2019). “Characterizing and Understanding HPC Job Failures Over The 2K-Day Life of IBM BlueGene/Q System”. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, pp. 473–484.

- Documentation, Intel (2011). “Intel® 64 and IA-32 Architectures Software Developers Manual”. In: *Volume 3B: System programming Guide, Part 2*. URL: <https://software.intel.com/en-us/articles/intel-sdm>.
- Dongarra, Jack, Robert Graybill, William Harrod, Robert Lucas, Ewing Lusk, Piotr Luszczek, Janice McMahon, Allan Snavey, Jeffrey Vetter, Katherine Yelick, Sadaf Alam, Roy Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy Meredith, and Mustafa Tikir (2008). “DARPA’s HPCS Program: History, Models, Tools, Languages”. In: *Advances in Computers High Performance Computing*. Ed. by Marvin V. Zelkowitz. Vol. 72. Advances in Computers. Elsevier, pp. 1–100. DOI: 10.1016/S0065-2458(08)00001-6.
- Dongarra, Jack J. (1992). “Performance of Various Computers Using Standard Linear Equations Software”. In: *SIGARCH Comput. Archit. News* 20.3, pp. 22–44. ISSN: 0163-5964. DOI: 10.1145/141868.141871.
- Eicken, Thorsten von, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser (1992). “Active Messages: A Mechanism for Integrated Communication and Computation”. In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ISCA ’92. ACM, pp. 256–266. ISBN: 0-89791-509-7. DOI: 10.1145/139669.140382.
- El-Sayed, Nosayba and Bianca Schroeder (2013). “Reading between the lines of failure logs: Understanding how HPC systems fail”. In: *2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, pp. 1–12.
- Fanfarillo, Alessandro, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson (2014). “OpenCoarrays: Open-source Transport Layers Supporting Coarray Fortran Compilers”. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, p. 4.
- Ferretti, Marco (2017). *Shared Memory Multiprocessors*. URL: <http://www-5.unipv.it/mferretti/cdol/aca/Charts/07-multiprocessors-MF.pdf> (visited on 02/2018).
- Finkel, David and Satish K Tripathi (1990). “A Performance analysis of a Buddy System for Fault Tolerance”. In: *Performance Evaluation* 11.3, pp. 177–185.

- Flynn, Michael J (1972). “Some Computer Organizations and their Effectiveness”. In: *IEEE transactions on computers* 100.9, pp. 948–960.
- Fog, Agner (2012). “The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers”. In: *Copenhagen University College of Engineering*, pp. 02–29.
- Gainaru, Ana, Franck Cappello, Marc Snir, and William Kramer (2012). “Fault prediction under the microscope: A closer look into hpc systems”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, p. 77.
- Gärtner, Felix C (1999). “Fundamentals of Fault-tolerant Distributed Computing in Asynchronous Environments”. In: *ACM Computing Surveys (CSUR)* 31.1, pp. 1–26.
- Gilbert, Seth and Nancy Lynch (2002). “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services”. In: *SIGACT News* 33.2, pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601.
- Graham, Richard, Jack Dongarra, Al Geist, Bill Gropp, Rainer Keller, Andrew Lumsdaine, Ewing Lusk, and Rolf Rabenseifner (2015). *MPI: A Message-passing Interface Standard, Version 3.1*.
- Gries, M., U. Hoffmann, M. Konow, and M. Riepen (2011). “SCC: A Flexible Architecture for Many-Core Platform Research”. In: *Computing in Science Engineering* 13.6, pp. 79–83. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.109.
- Grove, David, Sara Hamouda, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Josh Milthorpe, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Olivier Tardieu (2017). *Failure Recovery in Resilient X10*. Tech. rep.
- Hacker, Thomas J, Fabian Romero, and Christopher D Carothers (2009). “An Analysis of Clustered Failures on Large Supercomputing Systems”. In: *Journal of Parallel and Distributed Computing* 69.7, pp. 652 –665. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2009.03.007.
- Haerder, Theo and Andreas Reuter (1983). “Principles of Transaction-oriented Database Recovery”. In: *ACM Computing Surveys (CSUR)* 15.4, pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291.

- Heggie, Douglas and Piet Hut (2003). *The Gravitational Million-Body Problem: a Multidisciplinary Approach to Star Cluster Dynamics*. IOP Publishing. ISBN: 978-0521774864.
- Herlihy, Maurice P and Jeannette M Wing (1991). “Specifying Graceful Degradation”. In: *IEEE Transactions on Parallel and Distributed Systems* 2.1, pp. 93–104.
- Hewlett Packard Enterprise (2018). *HPE Superdome Flex server architecture and RAS*. Tech. rep. [Technical White Paper]. Hewlett Packard Enterprise. URL: <https://h20195.www2.hpe.com/v2/Getdocument.aspx?docname=a00036491enw> (visited on 06/2019).
- Horning, James J, Hugh C Lauer, Peter M Melliar-Smith, and Brian Randell (1974). “A Program Structure for Error Detection and Recovery”. In: *Operating Systems OS 1974, Lecture Notes in Computer Science*. Vol. 16. Springer, Berlin, Heidelberg, pp. 171–187. ISBN: 978-3-540-06849-5. DOI: 10.1007/BFb0029359.
- Hunt, Patrick, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed (2010). “ZooKeeper: Wait-free Coordination for Internet-scale Systems”. In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC’10. Boston, MA: USENIX Association, pp. 11–11. URL: <http://dl.acm.org/citation.cfm?id=1855840.1855851>.
- ISO/IEC JTC 1/SC 22, Programming languages (1999). *ISO/IEC 9899:1999 Programming Language C*. Revised by ISO/IEC 9899:2018. ISO/IEC.
- Jeyapaul, Reiley (2012). “Smart Compilers for Reliable and Power-efficient Embedded Computing”. PhD thesis. Arizona State University. URL: http://aviral.lab.asu.edu/bibadmin/uploads/pdf/ReileyJeyapaul_PhDThesis.pdf.
- Jin, Xiao-Zheng and Guang-Hong Yang (2009). “Robust Adaptive Fault-tolerant Compensation Control with Actuator Failures and Bounded Disturbances”. In: *Acta Automatica Sinica* 35.3, pp. 305–309. ISSN: 1874-1029. DOI: 10.1016/S1874-1029(08)60079-8.
- Johnson, Leslie A (1998). “DO-178B, Software considerations in airborne systems and equipment certification”. In: *RTCA, Radio Technical Commission for Aeronautic* 199.

- Kalé, Laxmikant V. and Sanjeev Krishnan (1993). “CHARM++: a portable concurrent object oriented system based on C++”. In: *ACM Sigplan Notices*. Vol. 28. 10. ACM, pp. 91–108.
- Kaufmann, Arnold, Roger Cruon, and Daniel Grouchko (1977). *Mathematical Models for the Study of the Reliability of Systems*. Elsevier.
- Kawachiya, Kiyokuni (2014). *Writing fault-tolerant applications using Resilient X10*. Tech. rep. Technical Report RT0960, IBM Research Tokyo.
- Kogge, Peter, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, and Robert Lucas (2008). “ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems”. In: *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Technical Representative* 15.
- Kothamasu, Ranganath, Samuel H. Huang, and William H. VerDuin (2006). “System Health Monitoring and Prognostics — A Review of Current Paradigms and Practices”. In: *The International Journal of Advanced Manufacturing Technology* 28.9, pp. 1012–1024. ISSN: 1433-3015. DOI: 10.1007/s00170-004-2131-6.
- Lamport, Leslie (1977). “Proving the Correctness of Multiprocess Programs”. In: *IEEE Transactions on Software Engineering* 3.2, pp. 125–143. ISSN: 0098-5589. DOI: 10.1109/TSE.1977.229904.
- Laprie, Jean-Claude (1992). “Dependability: Basic Concepts and Terminology”. In: *Dependability: Basic Concepts and Terminology: In English, French, German, Italian and Japanese*. Ed. by Jean-Claude Laprie. Vienna: Springer Vienna, pp. 3–245. ISBN: 978-3-7091-9170-5. DOI: 10.1007/978-3-7091-9170-5_1.
- (1995). “Dependable Computing and Fault-tolerance, Concepts and Terminology”. In: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years*, pp. 2–11. DOI: 10.1109/FTCSH.1995.532603.
- Laprie, Jean-Claude, Jean Arlat, Christian Beounes, and Karama Kanoun (1990). “Definition and Analysis of Hardware -and Software- Fault-tolerant Architectures”. In: *Computer* 23.7, pp. 39–51. ISSN: 0018-9162. DOI: 10.1109/2.56851.

- Laski, Janusz, Wojciech Szermer, and Piotr Luczycki (1995). “Error masking in computer programs”. In: *Software Testing, Verification and Reliability* 5.2, pp. 81–105.
- Layton, Jeff (2002). *Benchmarking Memory Bandwidth*. Ed. by Admin HPC New Day [Magazine]. URL: <http://www.admin-magazine.com/HPC/Articles/Finding-Memory-Bottlenecks-with-Stream> (visited on 06/2017).
- McCalpin, John D (1995). “Memory bandwidth and machine balance in high performance computers”. In: *IEEE Technical Committee on Computer Architecture Newsletter*. Pp. 19–25. URL: <http://www.cs.virginia.edu/stmam>.
- (2002). *STREAM Benchmark*. URL: <http://www.cs.virginia.edu/stream> (visited on 08/2017).
- Meter, Rodney van (2016). *Systems: Distributed-Memory Multiprocessors and Interconnection Networks*. [Lecture notes]. URL: <http://web.sfc.keio.ac.jp/~rdv/keio/sfc/teaching/architecture/architecture-2008/lec10-dsm.html> (visited on 01/2017).
- Morin, Christine and Isabelle Puaut (1997). “A survey of recoverable distributed shared virtual memory systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 8.9, pp. 959–969. ISSN: 1045-9219. DOI: 10.1109/71.615441.
- Murthy, Prabhakar D N, Min Xie, and Renyan Jiang (2004). *Weibull models*. Vol. 505. Wiley Series in Probability and Statistics. John Wiley and Sons. ISBN: 9780471473268. DOI: 10.1002/047147326X.
- Nakashima, Jun and Kenjiro Taura (2014). “MassiveThreads: A Thread Library for High Productivity Languages”. In: *Concurrent Objects and Beyond*. Vol. 8665. Springer, pp. 222–238. DOI: 10.1007/978-3-662-44471-9_10.
- Nitzberg, Bill and Virginia Lo (1991). “Distributed Shared Memory: A Survey of Issues and Algorithms”. In: *Computer* 24.8, pp. 52–60. ISSN: 0018-9162. DOI: 10.1109/2.84877.
- Numrich, Robert W and John Reid (1998). “CoArray Fortran for parallel programming”. In: *ACM Sigplan Fortran Forum*. Vol. 17. 2. ACM, pp. 1–31. DOI: 10.1145/289918.289920.
- Nyström, Jan Henry (2009). “Analysing Fault Tolerance for Erlang Applications”. PhD thesis. ISBN: 978-91-554-7532-1.

- Olsson, Ronald A, Gregory R Andrews, Michael H Coffin, and Gregg M Townsend (1992). *SR: A Language for Parallel and Distributed Programming*. Tech. rep. TR 92-09, Dept. of Computer Science, The University of Arizona.
- Oppenheimer, David, Aaron Brown, James Beck, Daniel Hettena, Jon Kuroda, N Treuhaft, David A Patterson, and Katherine Yelick (2002). “ROC-1: Hardware Support for Recovery-Oriented Computing”. In: *IEEE Transactions on Computers* 51.2, pp. 100–107. ISSN: 0018-9340. DOI: 10.1109/12.980002.
- Ousterhout, John, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman (2010). “The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM”. In: *SIGOPS Oper. Syst. Rev.* 43.4, pp. 92–105. ISSN: 0163-5980. DOI: 10.1145/1713254.1713276.
- Panagiotopoulou, K. and H. W. Loidl (2016). “Transparently Resilient Task Parallelism for Chapel”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1586–1595. DOI: 10.1109/IPDPSW.2016.102.
- Panagiotopoulou, Konstantina and Hans-Wolfgang Loidl (2015). “Towards Resilient Chapel: Design and Implementation of a Transparent Resilience Mechanism for Chapel”. In: *Proceedings of the 3rd International Conference on Exascale Applications and Software*. EASC ’15. Edinburgh, UK: University of Edinburgh, pp. 86–91. ISBN: 978-0-9926615-1-9. URL: <http://dl.acm.org/citation.cfm?id=2820083.2820100>.
- Patterson, David A, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, et al. (2002). *Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies*. Tech. rep. UCB//CSD-02-1175. UC Berkeley Computer Science. URL: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2002/5574.html>.
- Pennycook, Simon J, Jason D Sewall, and Victor W Lee (2016). “A metric for performance portability”. In: *Computing Research Repository (CoRR) in arXiv*. URL: <http://arxiv.org/abs/1611.07409>.

- Peyton Jones, Simon, Andrew Gordon, and Sigbjorn Finne (1996). “Concurrent Haskell”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’96. ACM, pp. 295–308. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237794. URL: <http://doi.acm.org/10.1145/237721.237794>.
- Power, Russell and Jinyang Li (2010). “Piccolo: Building Fast, Distributed Programs with Partitioned Tables”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. Vancouver, BC, Canada: USENIX Association, pp. 1–14. URL: <http://dl.acm.org/citation.cfm?id=1924943.1924964>.
- Puri, Rohit, Kannan Ramchandran, Kang-Won Lee, and Vaduvur Bharghavan (2001). “Forward Error Correction (FEC) Codes Based Multiple Description Coding for Internet Video Streaming and Multicast”. In: *Signal Processing: Image Communication* 16.8, pp. 745–762. DOI: 10.1016/S0923-5965(01)00005-4.
- Randell, Brian (1975). “System Structure for Software Fault Tolerance”. In: *IEEE Transactions on Software Engineering*. Vol. 1. 797478. ACM, pp. 220–232. DOI: 10.1109/TSE.1975.6312842.
- Sampath, Meera, Raja Sengupta, Stéphane Lafortune, Kasim Sinnamohideen, and Demosthenis Teneketzis (1995). “Diagnosability of Discrete-event Systems”. In: *IEEE Transactions on Automatic Control* 40.9, pp. 1555–1575. DOI: 10.1109/9.412626.
- Schlichting, Richard D and Fred B Schneider (1983). “Fail-stop processors: an approach to designing fault-tolerant computing systems”. In: *ACM Transactions on Computer Systems (TOCS)* 1.3, pp. 222–238.
- Schlichting, Richard D. and Vicraj T. Thomas (1995). “Programming language support for writing fault-tolerant distributed software”. In: *IEEE Transactions on Computers* 44.2, pp. 203–212.
- Schneider, Fred B. (1993a). “Distributed Systems (2nd Edition)”. In: ed. by Sape Mullender. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. Chap. What Good Are Models and What Models Are Good?, pp. 17–26. ISBN: 0-201-62427-3. URL: <http://dl.acm.org/citation.cfm?id=302430.302432>.

- Schneider, Marco (1993b). “Self-stabilization”. In: *ACM Computing Surveys (CSUR)* 25.1, pp. 45–67. ISSN: 0360-0300. DOI: 10.1145/151254.151256.
- Schroeder, Bianca and Garth Gibson (2010). “A Large-Scale Study of Failures in High-Performance Computing Systems”. In: *IEEE Transactions on Dependable and Secure Computing* 7.4, pp. 337–350. ISSN: 1545-5971. DOI: 10.1109/TDSC.2009.4.
- Shalf, John, J Bashor, David Patterson, K Asanovic, Katherine Yelick, Kurt Keutzer, and Tim Mattson (2009). “The Manycore Revolution: Will HPC Lead or Follow?” In: *Journal of SciDAC Review* 14, pp. 40–49. URL: <http://www.icsi.berkeley.edu/pubs/arch/manycorerevolution09.pdf>.
- Smith, Jonathan M (1988). *A survey of software fault tolerance techniques*. Tech. rep. DOI: 10.7916/D8639W1J.
- Sterling, Thomas Lawrence (2003). *Beowulf cluster computing with Linux*. 2nd ed. MIT press. ISBN: 0262692929.
- Titus, Greg (2011). “The Chapel Runtime”. In: *Proceedings of the 11th Annual Workshop on Charm++ and its Applications*. URL: https://charm.cs.illinois.edu/workshops/charmWorkshop2013/slides/CharmWorkshop2013_ext_chapel.pptx.
- Turnbull, Doug and Neil Alldrin (2003). *Failure prediction in hardware systems*. Tech. rep. University of California, San Diego.
- Vavilapalli, Vinod Kumar, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler (2013). “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: ACM, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633.
- Verbaauwhede, Ingrid, Patrick Schaumont, Christian Piguet, and Bart Kienhuis (2004). “Architectures and design techniques for energy efficient embedded DSP and multimedia processing”. In: *Proceedings of the conference on Design, automation and test in Europe*. IEEE Computer Society. IEEE, pp. 988–993. ISBN: 0-7695-2085-5. DOI: 10.1109/DATE.2004.1269022.

- Wheeler, Kyle B, Richard C Murphy, Dylan Stark, and Bradford L Chamberlain (2011). “The Chapel Tasking Layer over Qthreads”. In: *Cray Users’ Group Conference (CUG 2011)*.
- Wheeler, Kyle B, Richard C Murphy, and Douglas Thain (2008). “Qthreads: An API for programming with millions of lightweight threads”. In: *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*. IEEE, pp. 1–8. DOI: 10.1109/IPDPS.2008.4536359.
- Wisdom, Jack and Matthew Holman (1991). “Symplectic maps for the n-body problem”. In: *The Astronomical Journal* 102, pp. 1528–1538. ISSN: 0004-6256. DOI: 10.1086/115978.
- Woodacre, Michael, Derek Robb, Dean Roe, and Karl Feind (2003). “The SGI® Altix™ 3000 global shared-memory architecture”. In: *Silicon Graphics, Inc. [White Paper]*.
- Yakovlev, Alex (1993). “Structural technique for fault-masking in asynchronous interfaces”. In: *IEE Proceedings E (Computers and Digital Techniques)* 140.2, pp. 81–91. ISSN: 0143-7062.
- Yau, Sik-Sang and RC Cheung (1975). “Design of self-checking software”. In: *ACM SIGPLAN Notices*. Vol. 10. 6. ACM, pp. 450–455. DOI: 10.1145/390016.808468.
- Yelick, Kathy, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken (1998). “Titanium: a high-performance Java dialect”. In: *Concurrency and Computation: Practice and Experience* 10.11-13, pp. 825–836. DOI: 10.1002/(SICI)1096-9128(199809/11)10:11/13<825::AID-CPE383>3.0.CO;2-H.
- Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica (2012). “Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing”. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI’12. San Jose, CA: USENIX Association, pp. 2–2. URL: <http://dl.acm.org/citation.cfm?id=2228298.2228301>.
- Zheng, Gengbin, Chao Huang, and Laxmikant V. Kalé (2006). “Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for MPI and Charm++”.

In: *SIGOPS Oper. Syst. Rev.* 40.2, pp. 90–99. ISSN: 0163-5980. DOI: 10.1145/1131322.1131340.

Zheng, Gengbin, Lixia Shi, and Laxmikant V. Kalé (2004). “FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI”. In: *Cluster Computing, 2004 IEEE International Conference on*, pp. 93–103. DOI: 10.1109/CLUSTER.2004.1392606.