Load Balancing, Fault Tolerance, and Resource Elasticity for Asynchronous Many-Task Systems

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)

Vorgelegt im Fachbereich 16 – Elektrotechnik/Informatik
der Universität Kassel

von M.Sc. Jonas Posner

Betreuerin & Erstgutachterin:
Prof. Dr. Claudia Fohry

Zweitgutachter:
Prof. Dr. Martin Schulz

Eingereicht am 22. Dezember 2021 in Kassel
Verteidigt am 12. Juli 2022 in Kassel

doi:10.17170/kobra-202207286542

Danksagung

Diese Arbeit wäre ohne die wohlwollende Unterstützung verschiedener Personen nicht möglich gewesen.

Prof. Dr. Claudia Fohry, ich bin dir äußerst dankbar, dass du meine Doktormutter bist und mir die Möglichkeit gegeben hast zu promovieren. Ohne dich hätte ich diesen Weg damals nicht in Betracht gezogen. Vielen Dank, dass du dir seit Jahren immer Zeit für meine Anliegen nimmst und mich stets mit Rat und Tat unterstützt. Danke für die vielen Diskussionen, Anregungen und Vorschläge.

Prof. Dr. Martin Schulz, vielen Dank für dein großes Vertrauen in mich, für unsere Zoom Treffen, und für die Begutachtung meiner Arbeit.

Während meiner gesamten Zeit als Doktorand hatte ich das große Glück in einer angenehmen Atmosphäre mit netten Kollegen zusammenarbeiten zu dürfen. *M. Sc. Niko Luke*, danke für deine Unterstützung bei jeglichen Arbeiten auf dem Cluster. *Dr. Marco Bungart*, uns verbindet eine langjährige Freundschaft, die während unserer Zeit im gemeinsamen Büro entstanden ist. Wir haben nicht nur tagtäglich über unsere Forschung diskutiert, sondern uns auch über viele private Themen ausgetauscht. Ich denke gerne an unsere schöne gemeinsame Zeit zurück, vielen Dank. *M. Sc. Lukas Reitz*, durch unsere produktive Zusammenarbeit und den Austausch mit dir hat meine Forschung stark profitiert, vielen Dank. Ich freue mich schon auf deine Verteidigung :) *Claudia Huerkamp*, vielen Dank für das ideale Lösen aller anfallenden bürokratischen Hürden und für die vielen auflockernden Gespräche abseits des Arbeitsalltags.

Großen Dank an alle *meine Freunde*, mit denen ich meine freie Zeit genießen darf. Ob beim Kaffee oder beim Bier, in der Heimatstadt oder auf Städtetrips, beim Festival oder im Stadion, ihr sorgt zuverlässig dafür, dass ich vom Alltag abschalten kann und neue Energie sammeln kann.

Herzlichen Dank an meine Eltern *Petra und Frank* für eure bedingungslose lebenslange Unterstützung.

Abschließend möchte ich meiner wundervollen Partnerin *Katja* meine tiefste Dankbarkeit aussprechen. Du bist ein großartiger Mensch, bereicherst jeden Tag mein Leben und bist in jeglichen Lebenslagen immer für mich da. Ich liebe dich mehr als Worte sagen können <3

Danke,
Jonas

Zusammenfassung

High-Performance Computing (*HPC*) ermöglicht die Lösung komplexer Probleme aus verschiedenen wissenschaftlichen Bereichen, einschließlich gesellschaftlicher Probleme wie z.B. COVID-19. In letzter Zeit gibt es neben traditionellen Simulationen immer mehr irreguläre Anwendungen, welche die Vorhersagbarkeit der Berechnungen einschränken. Die Anwendungen werden auf HPC-Maschinen ausgeführt, die aus immer mehr Hardwarekomponenten bestehen und von mehreren Benutzern gleichzeitig verwendet werden.

Um eine effiziente und produktive Programmierung heutiger und zukünftiger HPC-Maschinen zu ermöglichen, muss eine Reihe von Problemen bewältigt werden, u.a.: *Lastenausgleich* (gleichmäßiges Auslasten der Ressourcen), *Fehlertoleranz* (Bewältigen von Hardwareausfällen) und *Ressourcenelastizität* (Hinzufügen/Entfernen von Ressourcen).

In dieser Dissertation adressieren wir die Probleme im Kontext der *Asynchronous Many-Task (AMT)* Programmierung. Bei AMT teilen Programmierer eine Berechnung in viele feingranulare Ausführungseinheiten (engl. *Tasks*) auf, die von einem Laufzeitsystem dynamisch an Recheneinheiten (z.B. Threads) zugewiesen werden. Während sich AMT für Einzelrechner immer mehr etabliert, konzentrieren wir uns auf Cluster-AMTs, bei denen es sich derzeit lediglich um Prototypen mit eingeschränktem Funktionsumfang handelt.

Hinsichtlich Lastenausgleich schlagen wir eine Work-Stealing-Technik vor, die Tasks transparent Ressourcen zuweist und so die Last über alle Recheneinheiten balanciert. In diesem Kontext führen wir mehrere Tasking-Konstrukte ein. Experimente zeigen eine gute Skalierbarkeit, und eine Produktivitäts-Evaluierung zeigt eine intuitive Handhabung.

Hinsichtlich Fehlertoleranz schlagen wir vier Techniken für den transparenten Schutz von Programmen vor. Nach einem Fehler wird die Ausführung eines Programms mit weniger Ressourcen fortgeführt. Drei Techniken schreiben unkoordinierte Sicherheitskopien in einen resilienten Speicher: Eine speichert alle offenen Task-Deskriptoren, die zweite speichert nur einen Teil davon, und die dritte protokolliert Stealing-Ereignisse um die Anzahl der Sicherheitskopien zu reduzieren. Die vierte Technik schreibt keine Sicherheitskopien, sondern nutzt Duplikationen während des Work-Stealings. Experimente zeigen keinen eindeutigen Sieger, z.B. hat die erste Technik bei schwacher Skalierung einen Mehraufwand ohne Fehler von unter 1% und für Wiederherstellungen von unter 0,5 Sekunden. Simulationen einer Menge von Jobs zeigen eine Reduzierung der Ausführungszeit um bis zu 97%.

Hinsichtlich Ressourcenelastizität schlagen wir eine Technik zum Hinzufügen und Entfernen von Rechenknoten vor, die Tasks entsprechend transparent verlagert. Experimente zeigen Kosten für das Hinzufügen/Entfernen unter 0,5 Sekunden. Simulationen einer Menge von Jobs zeigen eine Reduzierung der Ausführungszeit um bis zu 20%.

Abstract

High-Performance Computing (*HPC*) enables solving complex problems from various scientific fields including key societal problems such as COVID-19. Recently, traditional simulations have been joined by more diverse workloads, including irregular ones limiting the predictability of the computations. Workloads are run on HPC machines that comprise an increasing number of hardware components, and serve multiple users simultaneously.

To enable efficient and productive programming of today's HPC machines and beyond, it is essential to address a variety of issues, including: *load balancing* (i.e., utilizing all resources equally), *fault tolerance* (i.e., coping with hardware failures), and *resource elasticity* (i.e., allowing the addition/release of resources).

In this thesis, we address these issues in the context of *Asynchronous Many-Task (AMT)* programming. In AMT, programmers split a computation into many fine-grained execution units (called *tasks*), which are dynamically mapped to processing units (e.g., threads) by a runtime system. While AMT is becoming established for single computers, we are focusing on cluster AMTs, which are currently merely prototypes with limited functionalities.

Regarding load balancing, we propose a work stealing technique that transparently schedules tasks to resources of the overall system, balancing the workload over all processing units. In this context, we introduce several tasking constructs. Experiments show good scalability, and a productivity evaluation shows intuitive use.

Regarding fault tolerance, we propose four techniques to protect programs transparently. All perform localized recovery and continue the program execution with fewer resources. Three techniques write uncoordinated checkpoints in a resilient store: One saves descriptors of all open tasks; the second saves only part of them; and the third logs stealing events to reduce the number of checkpoints. The fourth technique does not write checkpoints at all, but exploits natural task duplication of work stealing. Experiments show no clear winner between the techniques. For instance, the first one has a failure-free running time overhead below 1% and a recovery overhead below 0.5 seconds, both for smooth weak scaling. Simulations of job set executions show that the completion time can be reduced by up to 97%.

Regarding resource elasticity, we propose a technique to enable the addition and release of nodes at runtime by transparently relocating tasks accordingly. Experiments show costs for adding and releasing nodes below 0.5 seconds. Additionally, simulations of job set executions show that the completion time can be reduced by up to 20%.

Contents

Acronyms

APGAS	**A**synchronous **P**artitioned **G**lobal **A**ddress **S**pace. Java library that brings the parallel programming concepts of X10 to Java [16], see Section 2.2.
$APGAS_{hyb}$	*APGAS* extension providing **hyb**rid load balancing and novel tasking constructs, see Section 3.3.
GLB	**G**lobal **L**oad **B**alancing via lifeline-based work stealing for dynamic independent tasks [17], see Section 2.3.
GLB_{X10}	Original *GLB* implementation in X10 [18], see Section 2.3.4.
GLB_{coop}	*GLB* implementation in *APGAS* using ***coop**erative* work stealing, see Section 3.2.2.
GLB_{split}	*GLB* implementation in *APGAS* using *coordinated* work stealing with a **split** queue as task pool data structure, see Section 3.2.3.
GLB_{multi}	*GLB* implementation in *APGAS* allowing **multi**ple workers per process [19], see Section 2.4.
X10-FT	**F**ault **T**olerance technique tailored to *GLB_{X10}*, which involves a sophisticated orchestration of several asynchronous protocols [20], see Section 1.3.3.
$X10\text{-}FT_{GLB}$	Implementation of *X10-FT* extending *GLB_{X10}*, see Section 2.3.4.
TC	**T**ask-level **C**heckpointing technique using a resilient store, see Section 4.3.2.

TC_{GLB}	Implementation of *TC* extending GLB_{coop}, see Section 4.3.4.
IncTC	**Inc**remental and selective **T**ask-level **C**heckpointing technique, see Section 4.4.1.
$IncTC_{GLB}$	Implementation of *IncTC* extending GLB_{coop}, see Section 4.4.1.3.
SST_{NFJ}	**S**upervision with **S**teal **T**racking fault tolerance technique for **N**ested **F**ork-**J**oin programs [21], see Section 4.4.3.1.
SST	SST_{NFJ} for dynamic independent tasks, see Section 4.4.3.2.
SST_{GLB}	Implementation of *SST* extending GLB_{coop}, see Section 4.4.3.3.
LogTC	Combination of *TC* and SST_{NFJ}, **log**s timestamps of stealing events, see Section 4.4.2.
$LogTC_{GLB}$	Implementation of *LogTC* extending GLB_{coop}, see Section 4.4.2.1.
TRE	**T**ask-level **R**esource **E**lasticity technique, see Section 5.2.
TRE_{GLB}	Implementation of *TRE* extending GLB_{multi}, see Section 5.3.

Existing	***New***	**Chapter**
① **APGAS**		*1. Parallel Programming*
② **GLB_{X10}** Implemented in X10 ③ **$\text{GLB}_{\text{multi}}$** Implemented in ①	⑤ **GLB_{coop}** Implemented in ① **$\text{GLB}_{\text{split}}$** Implemented in ① **$\text{APGAS}_{\text{hyb}}$** Extends ①	*3. Load Balancing*
$\text{X10-FT}_{\text{GLB}}$ Extends ② ④ **SST_{NFJ}**	⑥ **TC_{GLB}** Extends ⑤ **$\text{IncTC}_{\text{GLB}}$** Extends ⑤ **$\text{LogTC}_{\text{GLB}}$** Combines ⑥ and ④, extends ⑤ **SST_{GLB}** Implements ④, extends ⑤	*4. Fault Tolerance*
	TRE_{GLB} Extends ③	*5. Resource Elasticity*

Figure 1: Overview of acronyms with contributions being marked in blue color

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Contents

1.1 Motivation

Today's *High-Performance Computing (HPC)* systems, also called *supercomputer*s, are key for various scientific disciplines to solve complex challenges of societal importance. Such systems are becoming increasingly accessible providing more and more computing power, so that results can be achieved that were not practically computable only a short while back. Traditional HPC applications include simulations of various types, from climate to financial risk, and especially recently HPC applications are becoming more diverse and often have irregular workloads, i.e., a limited predictability of the computation. Examples of new application areas include *Data Analytics* and *Artificial Intelligence* [22, 23]. A recent prominent example combining several aspects is the computations for COVID-19 research [24, 25].

In the last decades, the increase in computing power has been enormous. Currently, we are in the beginning of the *exascale* era, which means that a supercomputer can execute up to 10^{18} floating point operations per second [26, 27]. According to `top500.org`, currently the fastest supercomputer in the world is Fugaku, providing almost half an exaflop. It uses 158,976 compute nodes interconnected as a cluster comprising a total of 7,630,848 cores [28]. The first full exascale supercomputer is expected to be available in the next few months [29].

For a certain period, new performance milestones have been reached by increasing processor clock rates and scaling the number of processors and processor cores. Nowadays, the overall performance of supercomputers is still increased by scaling the number of processors, but especially by innovative technologies that affect the entire architecture, including memory, communications, etc. Regarding computing power, recent techniques include heterogeneous architectures using accelerators, such as *Graphics Processing Units (GPUs)* and *Field Programmable Gate Arrays (FPGAs)* for specific purposes. Other techniques focus on increasing memory bandwidth and storage capacities by extending the memory hierarchy. They include *High Bandwidth Memory (HBM)* and non-volatile *Storage Class Memory (SCM)*, which can complement or even replace traditional hard disks. While these innovative technologies allow the total performance of supercomputers to grow, at the same time architectures are becoming more complex.

This increasing complexity and the emergence of diversified workloads makes it more and more challenging to develop efficient parallel applications. Key issues include *load balancing*, *fault tolerance*, and *resource elasticity*. Addressing them is essential to enable efficient use of exascale supercomputers and beyond.

In this thesis, we first discuss the three key issues, as well as *parallel programming* in general, providing their state of the art and identifying open research questions. In the main part of this thesis, we address these questions in a specific context, which is the *Asynchronous Many-Task (AMT)* programming paradigm introduced in Section 1.3, and propose a unified approach towards the three issues. Regarding supercomputer architectures, this thesis restricts consideration to clusters of homogeneous multi-core nodes.

1.2 Key Issues

1.2.1 Efficient and Productive Parallel Programming

For as long as parallel computing systems exist, writing parallel applications that effectively exploit all resources and achieve high performance at massive scale has been challenging, even for experts. In general, parallelization requires decomposing a large problem into smaller work packages allowing multiple processing units (e.g., processes or threads) to solve the large problem in parallel.

There are several approaches to decomposing a large problem, including *recursive decomposition* and *exploratory decomposition*. Recursive decomposition follows a divide-and-conquer strategy and recursively divides the problem into smaller sub-problems (work packages) until a specified size is reached. Exploratory decomposition divides a large solution space, e.g., of search or optimization problems, into smaller parts (work packages) that can be processed in parallel.

After identifying the work packages, they must be mapped to the processing units. Efficient mapping aims to achieve an equal execution time on all processing units. This is often denoted as *load balancing* and described in the following Section 1.2.2.

In addition to the above challenges, there are others, such as *data locality* and *race conditions*, see e.g., [30, 31, 32]. All of the challenges must be considered and resolved by the programmer. This leads to a greatly increased programming effort compared to traditional sequential programming which significantly compromises *programmer productivity*, i.e., the human efficiency in writing and maintaining applications.

Traditionally, parallelization challenges have been tackled by optimizing parallel applications to run on a particular supercomputer. While such optimizations can maximize the performance of the application on this machine, they limit the *portability*, i.e., the application must be manually re-optimized to run on others.

1.2.2 Load Balancing

To achieve maximum performance and scalability, the work packages from the decomposition must be distributed *fairly* across the processing units so that ideally each processing unit requires the same amount of time to compute its share. To ensure fairness, one may need to consider both the computation load and the required communication. An *unfair* distribution may, for example, result in *load imbalance*, i.e., some processing units complete their work packages faster than others and become idle, thus limiting performance and scalability.

The difficulty of accomplishing fair load balancing increases with the complexity of both the problem to be computed and the supercomputer architecture. In particular, *irregular* applications may generate new computational load and data at runtime and may be segmented into multiple phases with different resource requirements. Therefore, a fair load balancing for irregular workloads is a major challenge.

This thesis considers *dynamic* load balancing that distributes the work packages at runtime while taking newly generated work packages into account, which makes it appropriate for irregular workloads.

1.2.3 Fault Tolerance

As modern supercomputers are built with more and more hardware components, the probability of hardware failures increases [33, 34, 35]. Important failure types include temporary *soft errors* and permanent *fail-stop failures.*

Soft errors can be caused by different phenomena, e.g., cosmic rays, and may manifest as bit flips that falsify computed results. Nowadays, hardware components are equipped with integrated error correction codes, which detect and correct most of the soft errors. If soft errors are not detected, they are categorized as *silent errors* and may lead to wrong results.

Fail-stop failures can be caused by a crash of a hardware component or by a software error. Both situations lead to process failures, but are always detected. If no precautions are taken, the affected applications fatally abort.

Mean Time Between Failures (MTBF) is a well-known metric for describing the frequency of failures. Even if a single node may have an MTBF of, as an optimistic example, a century, a supercomputer of 100,000 such nodes will experience a failure every 9 hours on average [34]. Due to the increasing number of components in supercomputers, it is expected that the overall MTBF will continue to decrease. As a result, the probability of successful execution of parallel applications decreases as well, with long-running applications on many nodes being particularly affected.

This thesis considers fail-stop failures, and throughout the remainder of this work, terms like *failure*, *fault*, *error*, *fault tolerance* and *resilience* refer to this context.

1.2.4 Resource Elasticity

Supercomputer users do not execute applications directly, but submit them in the form of jobs via batch scripts. These scripts comprise commands and parameters describing the execution of the user application, as well as requirements such as the number of nodes, size of main memory etc. Afterwards, a job scheduler assigns the jobs to starting times and nodes, trying to maximize the overall resource utilization of the supercomputer while keeping waiting times low. Conventional jobs are *rigid*, i.e., they retain the same set of resources throughout their lifetime.

Rigidity limits the job scheduler's flexibility. Consider the example of a cluster with 5 nodes and 2 jobs, of which one needs 2 nodes in the first half of its computation, and 3 nodes afterwards; and the other needs 3 nodes first and 2 nodes later. If the jobs are *elastic*, they can run in parallel; if they are rigid, each occupies 3 nodes, and the jobs must be run in sequence.

As this example shows, rigidity is often not application-inherent, but rather a convention. For instance, jobs with irregular workloads may be segmented into phases with different resource requirements. Other jobs may be able to accommodate a different number of resources: They run faster with more resources, but still finish earlier with less resources if this allows for an earlier start. This is particularly true when the jobs can incorporate additional resources later on. Another source of flexibility for the job scheduler is an allowance to pull resources from a running job to reassign them to another.

1.3 State of the Art

1.3.1 Efficient and Productive Parallel Programming

Today's de facto parallelization standards for HPC applications are the *Message Passing Interface (MPI)* [36] for distributed-memory systems and *Open Multi-Processing (OpenMP)* [37] for shared-memory systems. In a recent survey among participants of the

US Exascale Computing Project [38], 73% of the respondents reported that they are using MPI, and 45% are using OpenMP. When targeting hybrid architectures, composed of shared-memory and distributed-memory, MPI and OpenMP are often combined. Thereby, *inter*-process parallelism is realized with MPI and *intra*-process parallelism is realized with OpenMP, commonly running one process per NUMA domain (Non-Uniform Memory Access).

As the name of MPI implies, its core idea is explicit messaging. Processes do not have direct access to the data of other processes, but they send and receive messages to exchange data. MPI offers numerous communication functions such as blocking and non-blocking point-to-point messages as well as collective operations like broadcast and reduce. While this low-level programming enables high performance, programmers must consider hardware-related aspects and deal with the fragmented memory model, which hinders programmer productivity. So far, the MPI standard does not support resource elasticity and fault tolerance only in a rudimentary way.

OpenMP programs are parallelized through the use of compiler directives, with the majority of thread management and distribution of the workload to threads being automatically handled by OpenMP. Thereby, programmers do not have to bother with threading details unlike in traditional low-level execution models, such as POSIX threads. Traditional OpenMP supports loop level parallelism, i.e., the workload of loops can be scheduled to threads in different ways, including *statically* and *dynamically.* The static approach schedules the loop iterations to threads in equal-sized chunks incurring only minimal management overhead. In contrast, the dynamic approach schedules the loop iterations at runtime incurring higher management overhead. Thus, the static approach is suitable if all iterations require about the same computation time, and the dynamic approach is well suited if the iterations require different computation times. However, OpenMP supports only loops with regular structure, but does not support irregular workloads in which new computational load is generated at runtime.

As both traditional MPI and OpenMP are reaching their limits (e.g., in terms of programmer productivity and features such as fault tolerance and resource elasticity) in the exascale era, for years there has been a growing interest in programming environments that increase programmer productivity through a higher level of abstraction and additionally support features such as fault tolerance and resource elasticity. As a result, numerous programming environments have been proposed that differ in performance, features, ecosystem, and last but not least in their acceptance by the community.

For distributed-memory programming, one well-known approach that facilitates programmer productivity is the *Partitioned Global Address Space (PGAS)* model [39]. PGAS is a high-level abstraction describing the memory of all processing units involved in the computation as one global memory. Thus, a unified semantic for local and remote operations can be provided. A *place* denotes a memory partition and associated computing resources. Typically, each place corresponds to one NUMA domain. Every place can access every memory partition, but local accesses are faster than remote ones. This way, PGAS hides the complexities of network communication, but not their existence, from the programmer.

PGAS has been implemented in several languages and libraries. For example, Co-Array Fortran (CAF) [40] is a standalone programming language; OpenSHMEM [41], Unified Parallel C (UPC) [42], and UPC++ [43] are libraries for C/C++; and Titanium [44] and PCJ [45] are libraries for Java.

Another promising approach is *Asynchronous Many-Task (AMT)* programming. Here, programmers split the computation of a large problem into many fine-grained execution units, called *task*s. They implement the tasks by using appropriate tasking constructs provided by the AMT environment (briefly *AMT*). Then, a runtime system of the AMT dynamically maps the tasks to a set of processing units (e.g., processes, threads), called *worker*s. Typically, the number of tasks is far higher than that of the workers. This way, AMTs support flexible and efficient solutions for dynamic load balancing of irregular workloads, see Section 1.3.2.

AMTs are gaining acceptance in the community for programming shared-memory systems, which is reflected in numerous commercially supported environments such as OpenMP tasks [37], oneAPI Threading Building Blocks (oneTBB) [46], and Java's Fork/Join Pool [47, 48].

In contrast, AMTs for programming distributed-memory systems are, so far, mainly research-oriented prototypes. Examples include new languages such as X10 [49] and Chapel [50], and libraries such as Legion [51], StarPU [52] and HPX [53] for C/C++, and *APGAS* [16] for Java. Additionally, there are increasing research activities regarding AMTs for heterogeneous systems, e.g., with accelerators such as GPUs [54, 55, 56, 57].

AMTs are not only interesting for conventional HPC applications but also receive attention for workflows [58] and data analytics [59]. AMTs differ in their use of a base programming language, which is often C/C++ for HPC and Java for data analytics, as well as in the targeted task granularity, and in the importance of high performance.

In general, AMTs have many differences in the support of features such as task cancellation, task priorities, task-internal parallelism, grain-size control, fault tolerance, and resource elasticity. Moreover, there are differences in the way tasks are allowed to cooperate (*task model*), which we classify based on [54] as follows.

- *Independent*: Each task computes a task result, and the overall result is computed by reduction over all task results using an associative and commutative operator, e.g., integer sum. Independent tasks can be *statically* known in advance or *dynamically* spawned at runtime. Except for parameter passing from parent tasks to child tasks (if applicable), independent tasks are not allowed to cooperate and must be free of side effects. Typically, all independent tasks execute the same code. *Dynamic Independent Tasks (DIT)* are supported by, e.g., YewPar [60, 61], Blaze-Tasks [62], and the *Global Load Balancing (GLB)* library [17, 18], and are useful for tree-based algorithms solving search, optimization, and approximation problems [60, 61, 62].

- *Nested Fork-Join*: A computation always starts with a single root task. Tasks may spawn new child tasks at runtime, resulting in a computation tree. Parents typically pass parameters to children. Moreover, parents wait for the result returns of their children. The final result is calculated by integrating task results upwards in the tree. In the pure form of nested fork-join, tasks may only communicate by passing parameters and returning results. Example AMTs include Java's Fork/Join pool [47, 48], Cilk [63], and Satin [64]. Nested fork-join programs are useful for divide-and-conquer algorithms.

- *Dataflow*: Tasks send their outputs to other tasks that need these outputs as input. Thereby, a directed acyclic task graph is formed. Tasks can either start immediately after having been spawned and then block their execution when they are waiting for inputs, or be started automatically when all inputs are available. Example AMTs include Legion [51], StarPU [52], PaRSEC [65] and HPX [53].

- *Side Effects*: Tasks cooperate through read and write accesses to shared or PGAS memory. Concurrent accesses must be synchronized to ensure data integrity. Through side effects, tasks may synchronize with other tasks. Side effects are popular and frequently used. They can be used alone or in combination with any of the above task models. Several AMTs utilize side effects strongly, for instance Legion, Chapel, X10, and OpenMP. However, *data locality*, i.e., executing tasks as close as possible to the required data, is an important issue, e.g., [66].

Since each AMT has advantages and disadvantages [54, 55, 57, 67], it is currently unclear which, if any of the existing ones, will become widely established. In addition, several research questions are open:

- How can AMT tasking constructs look so that they are generic, flexible, and user-friendly? Identifying such constructs is essential to facilitate programmer productivity while covering multiple application areas.

- Is it possible to design the above tasking constructs so that the AMT runtime system can extract all necessary information from them to support fault tolerance and resource elasticity, without requiring additional programming effort?

1.3.2 Load Balancing

Fair load balancing is essential for high performance at massive scale. Especially for irregular workloads, a well-established strategy is to perform load balancing *dynamically* so that the workload is balanced at runtime. Implementing and optimizing an efficient dynamic load balancing by hand for a given problem is challenging and time-consuming because many details are critical, such as the frequency and the granularity of the balancing.

AMTs address this issue by providing dynamic load balancing techniques transparently at the level of their runtime systems. Well-known task mapping techniques include *list-scheduling* and *work stealing/sharing.*

The basic idea of *list-scheduling* is to create a list of tasks before the first task is executed. For dynamic load balancing, the list of tasks is recalculated and reordered at runtime. Typically, list-scheduling is coupled with heuristic policies or performance models [52].

Work stealing is an especially capable and widely used technique [68] in which each worker maintains a *task queue* (also called *task pool*). A worker repeatedly takes tasks out of its pool, processes them, and inserts any new tasks into its pool. When a pool runs empty, the worker maintaining this pool, called *thief*, attempts to steal tasks from a co-worker, called *victim.* Victim selection may follow different schemes, a simple way is random selection.

Stealing can be accomplished in two different ways: 1) *cooperative* work stealing: either a thief requests tasks from a victim and then the victim extracts tasks (called *loot*) from its pool and sends them to the thief, or 2) *coordinated* work stealing: the thief extracts the tasks from the victim's pool itself. While the cooperative approach requires victims to occasionally interrupt task processing to answer steal requests, the coordinated approach requires synchronized access to the pools. The cooperative and coordinated approaches have been shown to be roughly equivalent in theory and practice [69], but this reference only considered shared-memory systems with random victim selection. Nowadays, however, work stealing is also used for distributed-memory systems.

The behavior during task generation can be distinguished into two policies: *help-first* (also called *child-stealing*) and *work-first* (also called *parent-stealing*). Under help-first, a worker spawning a child task continues processing the parent task and inserts the child task into its pool so that it can be stolen by another worker. Under work-first, the opposite is done; the worker inserts the parent task into the pool so that it can be stolen by another worker and processes the child task. Typically, help-first is deployed for dynamic independent tasks, and work-first is deployed for nested fork-join programs.

Work sharing follows a somewhat opposite approach to work stealing: Instead of empty workers seeking new tasks, overloaded workers share tasks with other workers. Occasionally, for hybrid architectures, work stealing and work sharing are combined [19, 70].

Furthermore, AMTs differ in details such as the number of tasks to be stolen/shared as well as the mechanism for termination detection. For the latter, the computation must be stopped as soon as each worker has no tasks left. Consequently, with a large number of workers, appropriate scalable algorithms are required.

Since most AMTs for distributed-memory systems and their dynamic load balancing techniques are currently research-oriented prototypes, several research questions are open:

- Which dynamic load balancing technique (e.g., work sharing, cooperative/coordinated work stealing, work-/help-first – or some combination or a new technique) achieves the best performance at massive scale?
- For which application areas (e.g., simulations, data analytics, artificial intelligence) is dynamic load balancing well suited? In short, can AMT be a one-fits-all solution?

1.3.3 Fault Tolerance

Resilience to failures can be implemented at different levels of the software stack. However, resilient programs are not yet state of the art in HPC. In the previously mentioned recent survey among participants of the US Exascale Computing Project [38], only 2% of the respondents reported that their applications are currently fault-tolerant, whereas 67% were planning to add this to their applications. One reason for these low numbers is probably that, in general, fault tolerance results in higher running times, even in failure-free runs.

In the following, we discuss several resilience techniques and classify them into two major levels: *system-level* and *application-level*. Additionally, each resilience technique has certain characteristics that differ in their impact on application performance and programmer productivity. In general, the more specific a resilience technique is designed to an application or an application area, the more efficient it can be.

A well-known technique that can be adapted at both system-level and application-level is *checkpoint/restart*. Here, checkpoints containing the state of the running application are written at regular time intervals. When a failure occurs, the application is restarted from the last valid checkpoint [34, 71]. The main drawback of checkpoint/restart, especially if adapted at system-level, is a high running time overhead caused by the checkpoint writing. Depending on the specific variant, the time for writing a checkpoint to a shared file system may, for instance, be 30 minutes [72].

We have seen that checkpoint/restart can be provided at system-level, e.g., by libraries such as DMTCP [73] and BLCR [74]. In this way, checkpoint/restart is provided transparently, i.e., the user code is left unchanged. Alternatively, checkpoint/restart can be provided at application-level, relying on libraries such as SCR [75] and FTI [76]. It can also be implemented from scratch for a particular application [77, 78]. At application-level, the application code must be adapted, resulting in an increased programming effort. For instance, when using an application-level library, a programmer must identify data that are sufficient for recovery. Hence, compared to system-level approaches, the checkpoint sizes can be smaller making checkpoint writing more efficient.

Typically, checkpoints are written to a shared file system. Enhanced variants strive to reduce the overhead of checkpoint writing by, e.g., updating the checkpoints incrementally, or by using a combination of several levels in the memory/storage hierarchy. Moreover, checkpoint writing can be performed in an uncoordinated way [79], so that each process decides autonomously when to update its checkpoint, and it can be combined with other techniques such as logging messages that can be re-sent during recovery [80].

At application-level, numerous alternative techniques to checkpoint/restart have been proposed in recent years. They include *Naturally Fault-Tolerant Algorithms* [81], *Algorithm-Based Fault Tolerance (ABFT)* [82, 83, 84, 85], and *replication* [86]. All of them may be designed to allow the application to continue running with less resources after a failure (called *shrinking recovery*). Thus, a restart of the application can be avoided, as it is required with checkpoint/restart. However, one drawback of all application-level techniques is an increased programming effort. The effort required depends on the problem to be solved, the resilience technique, and the programming environment.

All application-level techniques require resilience support from the programming environment, such as failure notification. However, the MPI standard does only provide rudimentary support, and there are only a few programming environments that do so, e.g., the Reinit++ [87] and the *User Level Failure Mitigation (ULFM)* [88] extensions of MPI, X10 [89], and APGAS [16]. While Reinit++ extends the MPI runtime with a global recovery solution, the basic concept of ULFM involves the return of error codes from MPI calls, to which the user program can react. In addition, approaches such as FENIX [90] and CRAFT [91] build on ULFM for checkpointing interfaces supporting recovery.

Both X10 and APGAS throw an exception in the event of a failure, which can be caught and used to react to the failure. In addition, programmers can register user-defined handlers on each process, which are automatically triggered in the event of a failure.

In this thesis, we consider *task-level resilience* techniques at the level of an AMT. Such techniques can combine the typical advantages of system-level techniques, namely no additional programming overhead, and application-level techniques, namely low runtime overhead.

Outside this thesis, only a limited number of task-level resilience approaches have been suggested so far. They can be categorized as follows:

- *Supervision and steal tracking.* For nested fork-join programs, several approaches for dealing with the loss of task subtrees have been proposed [92, 64, 21, 93]. In general, victims act as *supervisor*s, logging stolen tasks. In the event of a failure, the supervisors initiate recovery to reprocess the failed tasks. The most advanced variant [21] improves the recovery efficiency by avoiding re-execution of *intact* stolen children of failed tasks, which instead return their result to the grandparent (or another ancestor). Thereby, ancestors are discovered with the help of history information that is piggybacked onto loot deliveries. After failures, this information is globally collected in a so-called steal tree.

- *Data-centric bookkeeping.* Ma and Krishnamoorthy [94] proposed a bookkeeping technique for tasks with side effects. Their technique logs all updates of the shared-memory. In the event of a failure, this information is used to determine the failed tasks and avoid duplicate updates when failed tasks are re-executed, i.e., the memory is *idempotent*. The bookkeeping information is also used to restore lost data.

- *Checkpointing data.* Zheng *et al.* [95] proposed a checkpointing technique for Charm++ [96] that stores objects in-memory and/or on disk. These objects include dataflow-coupled tasks and data. In the event of a failure, a program restart is required, but this may be performed with more or less resources. Lion and Thibault [97] proposed a checkpointing technique for StarPU [52], which checkpoints the data sent between dataflow-coupled tasks. They considered static tasks, which can be easily restarted.

- *Checkpointing task descriptors.* Fohry *et al.* [20] proposed a checkpointing technique for dynamic independent tasks and implemented it by extending the Global Load Balancing (*GLB*) library of X10. We denote their technique by *X10-FT* (***F**ault*

T*olerance*) and the extended fault-tolerant library by *X10-FT*$_{GLB}$. Their technique periodically stores for each worker its task descriptors and results into the memory of another worker. These checkpoints are updated in events such as stealing and recovery. *X10-FT* supports shrinking recovery, which in most parts is performed by the worker holding the checkpoint to be recovered. While *X10-FT* can be quite efficient because only little data needs to be checkpointed, it is implemented prototypically, and involves a sophisticated orchestration of several asynchronous protocols for checkpoint writing, stealing, failure detection and restore. This makes the algorithm hard to understand, implement, and verify. Moreover, the algorithm is restricted to use of a single checkpoint copy per data item. Thus, a simultaneous crash of two or more workers in an unfavorable constellation cannot be tolerated.

Despite all these different task-level resilience approaches, several research questions are still open:

- Are general resilience approaches – such as checkpointing – well suited to be provided at task-level? Which algorithmic techniques achieve both low running time overhead and low recovery costs?
- Are the above task-level resilience techniques specific to a particular task model and load balancing technique, or are they more generally applicable to a spectrum of task models and load balancing techniques?
- Can the above techniques be simple to facilitate their own implementation, maintenance, etc.?
- How can the performance of the above techniques be quantified to determine which of them works best in a given scenario without experimentally running it? More specifically, how can the overall running times including failure handling be predicted for different applications, number of resources, MTBF rates, etc.?
- Given that the above task-level resilience techniques are deployed in jobs on supercomputers, by how much does the effective throughput of the supercomputers increase from using these techniques?

1.3.4 Resource Elasticity

Feitelson and Rudolph [98] distinguish jobs in *rigid* (resources fixed by the user), *moldable* (resources fixed by the job scheduler at program start), *evolving* (resources changeable at runtime on application initiative) and *malleable* (resources changeable at runtime on job scheduler initiative). We use the term *elastic* [99] to express that an application can change the number of resources regardless of who takes the initiative.

As of yet, elastic applications are not widely used in practice in HPC. In the previously mentioned recent survey among participants of the US Exascale Computing Project [38], 39% of the respondents reported that their applications can change the number of processes via restarting from a checkpoint, and only 16% reported that their applications can dynamically change the number of processes at runtime. As with fault tolerance, one reason for these low numbers is likely the fact that additional programming effort is generally required to make applications elastic.

This thesis focuses on malleability and considers nodes, respectively processes, as resources. Since both malleability and resilience deal with changes in the number of processes at runtime, they are closely related and share some commonalities.

Malleability must be backed by at least three major layers: (1) job scheduler, (2) programming environment, and (3) algorithm/application. Moreover, a communication layer between (1) and (2) is required. Recent research has addressed these layers, but no comprehensive solution has yet been widely accepted in practice.

Scheduling strategies for malleable jobs on supercomputers have been studied with the goals of improving the effective throughput and reducing energy consumption, e.g., [100, 101, 102, 103]. However, these research outcomes have not made their way into everyday job schedulers, which currently provide only rudimentary, if any, support for malleable jobs and their scheduling. In research, corresponding extensions for the popular job schedulers Slurm [100, 101] and Torque [102] have been proposed. In addition, resource management can be performed hierarchically [104].

Malleability and elasticity for applications can be achieved with different approaches. For instance, checkpoint/restart can provide elasticity through the same mechanisms traditionally used to handle failures [34]. This way, after writing a checkpoint, the application can be terminated and restarted with a new number of processes. Example implementations include the SCR library [75], as well as the MPI extension PCM [105] and the MPI implementation AMPI [106]. As for failure tolerance, the performance of checkpoint writing and restarting can be enhanced by writing checkpoints in-memory [107].

Again, as for fault tolerance, enhanced approaches at the application-level can increase efficiency and avoid the need to restart. Research on elastic algorithms has primarily focused on iterative computations that provide natural synchronization points at which the application can adapt to resource changes with reasonable ease [101]. Nevertheless, the application code must be adapted to accommodate the resource changes, which may require a non-negligible additional programming effort.

Again, the programming environment must support elasticity, but there are only a few environments that do so, e.g., ULFM [88], X10 [49] and *APGAS* [16].

Since AMTs perform resource management at runtime, they offer appealing potential for dynamically changing the number of processes, similar to task-level resilience techniques. Therefore, AMTs may enable malleability in an efficient way without requiring synchronization points or application code modifications. When processes are removed, the AMT needs to move tasks and data from the leaving workers to the remaining ones. Although this has some similarities to handling the loss of a worker, the main difference is that no prior actions are required, as the removal is performed in a controlled way. When processes are added, the AMT needs to include the new workers into the ongoing dynamic load balancing so that they can start processing tasks.

Although the elasticity potential of AMTs has been observed before (e.g., [108]), as of yet only few AMTs actually implement elasticity (e.g., [96, 49, 16]). The authors of *X10-FT*$_{GLB}$ proposed an elasticity extension [109] based on a growth protocol to integrate new processes into the work stealing, but they did not include a shrink protocol.

This work addresses task-level elasticity, for which several research questions are open:

- Which algorithmic techniques achieve low response time to external resource requests, low running time overhead for performing resource changes, and negligible running time overhead if no resource changes are performed?

- Can the above techniques be simple, in order to facilitate their own implementation, maintenance, etc.?

- How can the performance of the above techniques be quantified to determine which of them works best in a given scenario without experimentally running it? More specifically, how can the overall running times be predicted, including the impact of resource changes, and associated running time overheads?

- Given that the above techniques are deployed in jobs on supercomputers, by how much does the effective throughput of the supercomputers increase from using these techniques?

- How can job schedulers support malleability in a user-friendly way while maximizing the effective throughput?

1.4 Contributions

The rest of this thesis addresses the four previously outlined issues – *efficient and productive parallel programming, load balancing, fault tolerance, and resource elasticity* – with focus on the open research questions formulated in Section 1.3. While the questions have been phrased in a broad and general manner, we restrict our answers to a specific context: *dynamic independent tasks in distributed-memory AMTs.*

More specifically, throughout this thesis, we build on the *APGAS library* (briefly *APGAS*) [16], which is a parallel programming environment, and on *lifeline-based work stealing* [17], which is a dynamic load balancing technique. Both are briefly described below.

APGAS is a branch of IBM's X10 project and brings the parallel programming concepts of X10 to Java by using lambdas. Like X10, *APGAS* realizes an asynchronous variant of the PGAS model [110] and supports fault tolerance and resource elasticity. *APGAS* is open source [111] and comprises less than 10,000 lines of code, which allow us to modify and extend the code for the purposes of this thesis.

Lifeline-based work stealing was introduced by the *Global Load Balancing* (*GLB*) [18] library that was originally implemented in X10. The original *GLB* library has less than 2,000 lines of code, and we denote it by GLB_{X10}. The library uses cooperative help-first work stealing and deploys a low-diameter graph for victim selection and termination detection. For our studies, we build on GLB_{X10}, but port, modify, and extend the library to address different issues.

In the following, we briefly outline our specific contributions, devoting one chapter to *load balancing, fault tolerance, and resource elasticity*, respectively. Contributions to *efficient and productive parallel programming* are addressed at appropriate places within these three chapters.

1.4.1 Load Balancing

Our first contribution refers to the open research question of which dynamic load balancing technique achieves the best performance. More specifically, we compare *cooperative* and *coordinated* work stealing in the context of *GLB* (denoted by GLB_{coop} and GLB_{split}, respectively) and show that there are only minor performance differences between them. Then, we design a novel hybrid work stealing technique achieving both *intra-* and *inter*-process load balancing. We implement the hybrid work stealing technique by extending *APGAS* and introduce several novel tasking constructs. The result is denoted

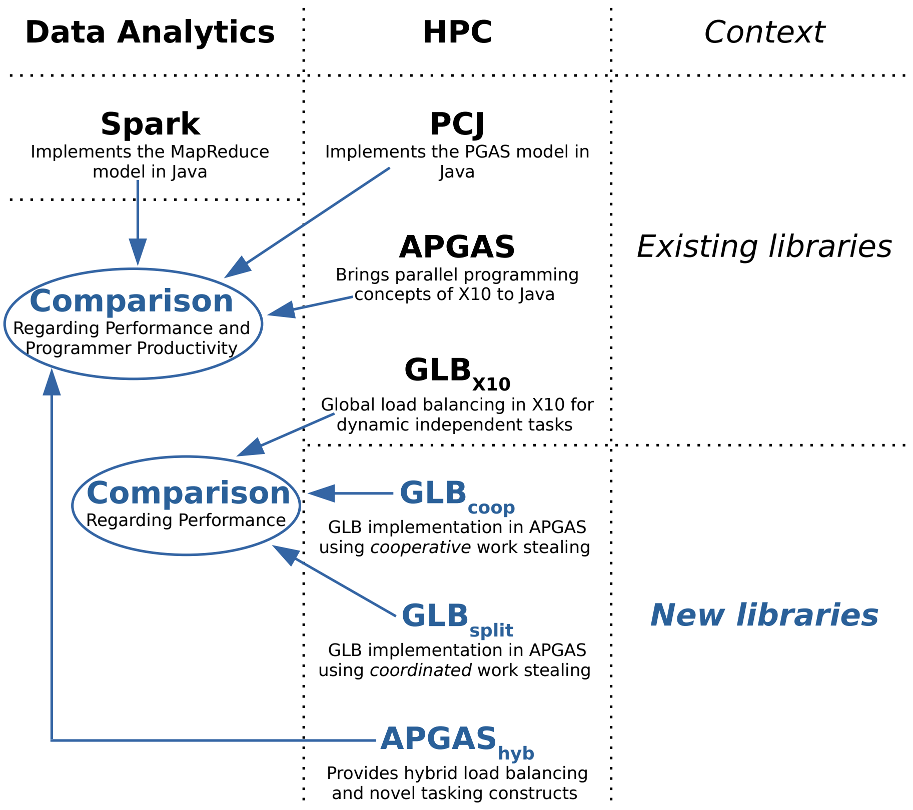

Data Analytics
HPC
Context
Spark
Implements the MapReduce model in Java
PCJ
Implements the PGAS model in Java
APGAS
Brings parallel programming concepts of X10 to Java
Existing libraries
Comparison
Regarding Performance and Programmer Productivity
GLB$_{X10}$
Global load balancing in X10 for dynamic independent tasks
Comparison
Regarding Performance
GLB$_{coop}$
GLB implementation in APGAS using *cooperative* work stealing
GLB$_{split}$
GLB implementation in APGAS using *coordinated* work stealing
New libraries
APGAS$_{hyb}$
Provides hybrid load balancing and novel tasking constructs

Figure 1.1: Overview of load balancing contributions and acronyms with contributions being marked in blue color

by *APGAS*$_{hyb}$. Finally, we contribute to the open research question whether AMT can be a one-fits-all solution by comparing *APGAS*$_{hyb}$ with *APGAS*, the data analytics library *Spark* [112], and the HPC library *PCJ* [45]. For our comparison, we evaluate the performance of the programming environments as well as the programmer productivity offered by their constructs for both HPC and data analytics applications.

Figure 1.1 gives an overview of the load balancing contributions and acronyms with contributions being marked in blue color. In the following, we describe the contributions in more detail.

1.4.1.1 Cooperative vs. Coordinated Work Stealing

Prior to this thesis, Acar *et al.* [69] had shown that *coordinated* and *cooperative* work stealing are roughly equivalent in theory and practice, but they had only considered random victim selection and shared-memory systems. We extend their studies by performing the comparison for lifeline-based victim selection and distributed-memory systems.

Prior to our work, lifeline-based work stealing had only been implemented in the cooperative way in *GLB*$_{X10}$ [18]. We add a *coordinated GLB* variant and compare coordinated and cooperative work stealing.

For practical reasons, we implement the two *GLB* variants in *APGAS*. The cooperative variant, denoted by *GLB*$_{coop}$, resembles *GLB*$_{X10}$, but replaces some specific X10 constructs, which are not available in *APGAS*, with Java synchronization constructs. For the coordinated variant, denoted by *GLB*$_{split}$, we adopt a *split queue* [113] as task pool data structure. Thus, we enable efficient task extraction by thieves while victims can continue processing tasks. In experiments, *GLB*$_{coop}$ outperforms *GLB*$_{split}$ in most cases.

1.4.1.2 Hybrid Work Stealing

GLB allows only one worker per process, thus multiple processes must be started to exploit a multicore node, resulting in unnecessary communication and increased memory consumption. We address this shortcoming by designing a novel hybrid work stealing technique that supports multiple workers per process. For that, we combine Java's Fork/Join Pool [48] for *intra*-process load balancing with the coordinated lifeline-based *GLB* variant, sketched in Section 3.2.3, for *inter*-process load balancing.

We implement the hybrid work stealing technique by extending *APGAS*. In this context, we introduce novel constructs for spawning dynamic independent tasks and several related constructs, e.g., for result reduction. Moreover, unprocessed tasks are now cancelable, which is useful for several applications such as search problems. The resulting *APGAS* variant is denoted by *APGAS*$_{hyb}$. In experiments, *APGAS*$_{hyb}$ shows good scalability in most cases.

1.4.1.3 Evaluation of APGAS for HPC and Data Analytics

We contribute to the one-fits-all open research question by evaluating whether *APGAS*$_{hyb}$ is appropriate for programming both HPC and data analytics applications. To this end, we compare *APGAS*$_{hyb}$ with *APGAS*, *Spark* [112], and *PCJ* [45] in terms of performance and programmer productivity using objective metrics.

Spark is a distributed, multi-threaded, fault-tolerant library for data analytics that implements the MapReduce model [59] and is widely used in this domain. *PCJ* implements the PGAS model in a fundamentally different way than *APGAS* and won the HPC Challenge Class 2 Best Productivity Award in 2014 [114].

For the comparison, we select typical benchmarks from the two domains of HPC and data analytics. Regarding performance, experiments show *APGAS*$_{hyb}$ as a clear winner.

Regarding programmer productivity, $APGAS_{hyb}$ requires the lowest number of different library constructs and only a few lines more than *Spark*. Moreover, we find that it was most intuitive to use. Hence, $APGAS_{hyb}$ might be a good candidate for programming both HPC and data analytics applications using the same programming environment.

1.4.2 Fault Tolerance

One open research question asks whether general resilience techniques could be applied effectively and easily at the task-level. We address this question by designing a novel uncoordinated checkpointing technique at the task-level that tolerates simultaneous process failures and performs recovery in a local and shrinking way. We denote this technique by *TC* (***T**ask-level **C**heckpointing*).

To briefly describe *TC*, each worker independently writes checkpoints of its task pool contents and its current result. The checkpoints are written regularly at fixed time intervals, as well as in the events of stealing and recovery. During stealing, the loot is checkpointed as well. All checkpoints are written to a resilient store that uses replication internally, preventing data loss despite simultaneous process failures. Upon process failure, all other processes are notified to take appropriate actions. In particular, a designated *backup partner* takes over the tasks of the failed process. Overall, our technique is reasonably simple, thanks to the resilient store.

To experimentally evaluate *TC*, we implement it by extending GLB_{coop} and denote the result by TC_{GLB}. Experiments show a failure-free overhead below 1% and a recovery overhead after failures below 0.5 seconds, both for smooth weak scaling. In addition, we compare the performance of TC_{GLB} with that of DMTCP [73], which, as explained before, is a well-known user-space checkpoint/restart library. The results clearly show that task-level resilience has significantly lower running time overhead and less recovery overhead than DMTCP.

In addition to *TC*, we develop three more enhanced variants, all of which aim to further reduce these overheads. Figure 1.2 gives an overview of the variants and acronyms, with blue color marking the contributions of this thesis. The variants (named *IncTC*, *LogTC*, and *SST*) are developed in collaboration with co-authors, see Section 1.5. With two of the variants, *LogTC* and *SST*, we simultaneously address the open research question whether task-level resilience techniques are specific to a particular task model. For these two variants, we adapt a resilience technique that was originally designed for nested fork-join programs with work-first work stealing [21], denoted by SST_{NFJ}, to dynamic independent tasks. SST_{NFJ} combines ***S**upervision and **S**teal **T**racking* to provide resilience, and we

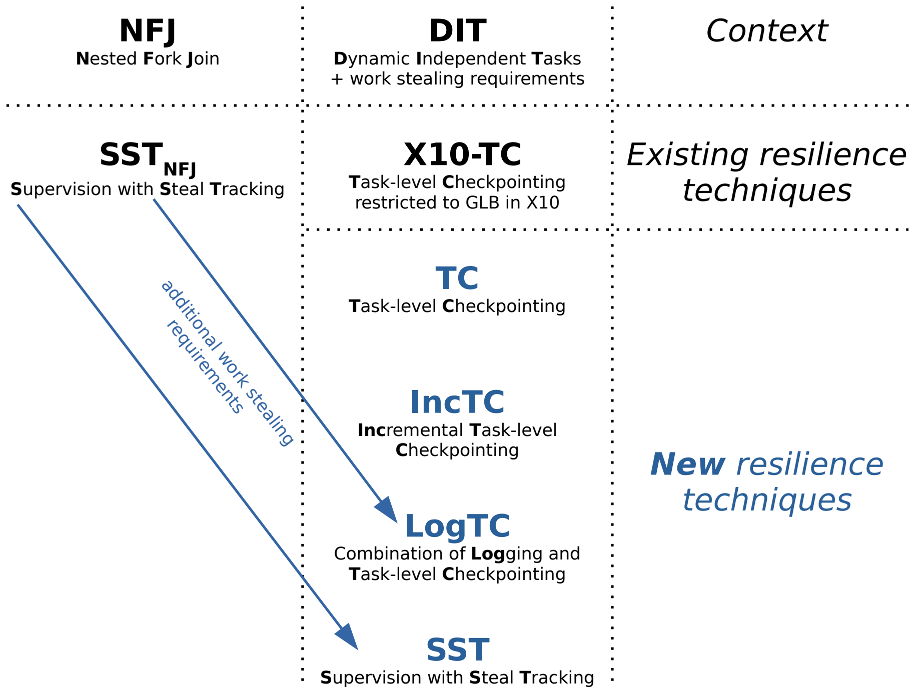

NFJ
Nested Fork Join
DIT
Dynamic Independent Tasks + work stealing requirements
Context
SST_NFJ
Supervision with Steal Tracking
X10-TC
Task-level Checkpointing restricted to GLB in X10
Existing resilience techniques
additional work stealing requirements
TC
Task-level Checkpointing
IncTC
Incremental Task-level Checkpointing
LogTC
Combination of Logging and Task-level Checkpointing
SST
Supervision with Steal Tracking
New resilience techniques

Figure 1.2: Overview of fault tolerance contributions and acronyms with contributions being marked in blue color

adapt it in two different ways. As for *TC*, we implement all variants by extending GLB_{coop} and evaluate them experimentally. In the following, we briefly describe the variants:

- *IncTC*: While *TC* always writes checkpoints comprising the entire task pool content, this variant writes checkpoints **inc**rementally and for "stable" tasks only. Experiments showed that the larger the task descriptor size, the more effective *IncTC* is compared to *TC*.

- *LogTC*: This variant combines SST_{NFJ} with *TC*. Thus, in the event of stealing, *LogTC* does not write checkpoints, but **log**s timestamps of stealing events and saves them in the resilient in-memory store. Experiments showed no clear winner between *TC* and *LogTC*.

- *SST*: This variant transfers SST_{NFJ} (**S**upervision with **S**teal **T**racking) itself from the context of nested fork-join programs to our specific context – dynamic independent tasks with help-first work stealing. Thus, *SST* does not write checkpoints at all, but enables fault tolerance with supervision and steal tracking. Experiments showed slightly lower running time overhead in failure-free runs for *SST* compared to *TC*, but *TC* has lower and less varying recovery costs.

To address the next open research question of how to determine which technique performs best in a given scenario, we derive formulas. These formulas predict the overall running times of applications written with *TC* and *SST*, including failure handling. The formulas depend on MTBF, number of workers, and steal rate. Based on the formulas, we predict the execution times of single long-running applications under failures.

We then addressed the open research question of how much the effective throughput of supercomputers can be increased using protected jobs. To this end, we simulate the execution of job sets on two supercomputers with varying MTBFs to quantify the impact of protected jobs on the overall completion time. The results show that application protection by *TC* or *SST* is effective, and that the difference between the two is rather low. *SST* performs slightly better in all currently realistic scenarios, but *TC* is ahead in systems with an order of millions of processes and for particularly small MTBFs.

1.4.3 Resource Elasticity

To address the open research question which resource elasticity techniques could be applied effectively and easily at the task-level, we design a novel ***T**ask-level **R**esource **E**lasticity* (*TRE*) technique that enables the transparent adaptation of applications to the addition or release of multiple nodes. *TRE* can handle resource changes without modifying the user application.

To describe the technique briefly, resource changes are accomplished by the runtime system that relocates tasks to added nodes and away from released nodes. Process 0 oversees a protocol, in which it starts new workers, instructs others to withdraw etc., and shuts down the processes when it is safe to do so. As its greatest achievement, the protocol involves only a small subset of workers and continues task processing as far as possible during the adaptation. Overall, we keep the technique simple to facilitate its implementation

To experimentally evaluate *TRE*, we implement it by extending the *multi-worker Global Load Balancing* (GLB_{multi}) library [19] and denote the result by TRE_{GLB}. GLB_{multi} is an enhanced *GLB* variant that combines lifeline-based work stealing across processes with work sharing among the workers of a process to enable multiple workers per process. We choose GLB_{multi} instead of GLB_{coop} or GLB_{split} for this study, since it is more advanced and was available at the time when we conducted this study, but not when we conducted those from Sections 1.4.1 and 1.4.2.

We then address the open research question of how to quantify the performance of *TRE*. The overhead for adding or releasing nodes can not just be measured, since it is impossible to run an application with a particular resource scenario without causing the overhead. Therefore, we derive formulas that estimate the overhead-free running time of work stealing applications with a changing number of resources. We then subtract these estimated values from experimentally measured running times to determine the running time overheads for adding or releasing nodes. The results demonstrate that the time required to add and release up to 64 nodes is less than 0.5 seconds. If no resource changes are performed, the running time overhead is negligible.

Building on these results, we address the open research questions on the impact of malleable jobs on the throughput of supercomputers. To this end, we simulate the execution of job sets on two supercomputers. Results show an increase of the throughput by up to 20% if most jobs are malleable. Addressing the open research question of how to enable job malleability in a user-friendly way, we propose that jobs must be parametrized with a minimum, maximum, and preferred number of nodes. Additionally, we introduce a heuristic for determining appropriate values for these parameters.

1.5 Publications

All content of this thesis has already been published in the following peer-reviewed publications (sorted ascending by publication date):

[P1] Jonas Posner and Claudia Fohry. “Cooperation vs. Coordination for Lifeline-Based Global Load Balancing in APGAS”. in: *Proceedings SIGPLAN Workshop on X10*. ACM, 2016, pp. 13–17. DOI: 10.1145/2931028.2931029
Own contribution: Most of the algorithm development and implementation of GLB_{coop} and GLB_{split}; execution and evaluation of experiments; formulation of parts of the manuscript.

[P2] Jonas Posner and Claudia Fohry. “Fault Tolerance for Cooperative Lifeline-Based Global Load Balancing in Java with APGAS and Hazelcast”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS) Workshops (APDCM)*. IEEE, 2017, pp. 854–863. DOI: 10.1109/ipdpsw.2017.31
Own contribution: Most of the algorithm development and implementation of *TC*; execution and evaluation of experiments; formulation of parts of the manuscript.

[P3] Jonas Posner. “A Generic Reusable Java Framework for Fault-Tolerant Parallelization with the Task Pool Pattern”. In: *International Parallel and Distributed Processing Symposium (IPDPS), Ph.D. Forum*. Poster. 2017

[P4] Jonas Posner and Claudia Fohry. "A Java Task Pool Framework providing Fault-Tolerant Global Load Balancing". In: *Special Issue International Journal of Networking and Computing (IJNC)* 8.1 (2018), pp. 2–31. DOI: `10.15803/ijnc.8.1_2`
Extended version of [P2]: New benchmarks; re-execution and re-evaluation of experiments (on a more recent supercomputer); re-formulation of the manuscript.
Own contribution: Most of the extension.

[P5] Jonas Posner and Claudia Fohry. "A Combination of Intra- and Inter-place Work Stealing for the APGAS Library". In: *Proceedings Parallel Processing and Applied Mathematics (PPAM) Workshops (WLPP).* Springer, 2018, pp. 234–243. DOI: `10.1007/978-3-319-78054-2_22`
Own contribution: Algorithm development and implementation of $APGAS_{hyb}$; execution and evaluation of experiments; formulation of parts of the manuscript.

[P6] Jonas Posner and Claudia Fohry. "Hybrid Work Stealing of Locality-Flexible and Cancelable Tasks for the APGAS Library". In: *The Journal of Supercomputing* (2018), pp. 1435–1448. DOI: `10.1007/s11227-018-2234-8`
Extended version of [P5]: Added new benchmarks; extension of the algorithm and implementation of a new cancellation feature; re-execution and re-evaluation of experiments; re-formulation of the manuscript.
Own contribution: Most of the extension.

[P7] Claudia Fohry, Jonas Posner, and Lukas Reitz. "A Selective and Incremental Backup Scheme for Task Pools". In: *Proceedings International Conference on High Performance Computing & Simulation (HPCS).* 2018, pp. 621–628. DOI: `10.1109/HPCS.2018.00103`
Own contribution: Most of the implementation of *IncTC*; most of the execution and evaluation of experiments; formulation of parts of the manuscript.

[P8] Jonas Posner, Lukas Reitz, and Claudia Fohry. "Comparison of the HPC and Big Data Java Libraries Spark, PCJ, and APGAS". in: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC) Workshops (PAW-ATM).* ACM, 2018, pp. 11–22. DOI: `10.1109/PAW-ATM.2018.00007`
Own contribution: Supervision of the implementations in the context of an undergraduate project; execution and evaluation of experiments; formulation of parts of the manuscript.

[P9] Jonas Posner, Lukas Reitz, and Claudia Fohry. "A Comparison of Application-Level Fault Tolerance Schemes for Task Pools". In: *Future Generation Computing Systems (FGCS)* 105 (2019), pp. 119–134. DOI: `10.1016/j.future.2019.11.031`
Extended version of [P7]: Added new fault tolerance technique *LogTC*; added new benchmarks; re-execution and re-evaluation of experiments (on a larger supercomputer); re-formulation of the manuscript.
Own contribution: Supervision of algorithm development and implementation of *LogTC* in the context of a bachelor thesis.

[P10] Jonas Posner. "System-Level vs. Application-Level Checkpointing". In: *Proceedings International Conference on Cluster Computing (CLUSTER), Extended Abstract.* IEEE, 2020, pp. 404–405. DOI: `10.1109/CLUSTER49012.2020.00051`

[P11] Jonas Posner. "Locality-Flexible and Cancelable Tasks for the APGAS Library". In: *EuroHPC Summit Week, PRACEdays.* Poster. 2021

[P12] Jonas Posner, Lukas Reitz, and Claudia Fohry. "Checkpointing vs. Supervision Resilience Approaches for Dynamic Independent Tasks". In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS) Workshops (APDCM).* IEEE, 2021, pp. 556–565. DOI: `10.1109/IPDPSW52791.2021.00089`
Own contribution: Supervision of algorithm development and implementation of *SST* in the context of a master thesis; collaborative execution and evaluation of experiments; formulation of parts of the manuscript.

[P13] Jonas Posner. "Resource Elasticity at Task-Level". In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS), Ph.D. Forum, Extended Abstract.* IEEE, 2021. DOI: `10.1109/IPDPSW52791.2021.00160`

[P14] Jonas Posner and Claudia Fohry. "Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing". In: *Proceedings International Conference on Parallel Processing (ICPP) Workshops (P2S2).* ACM, 2021. DOI: `10.1145/3458744.3473361`
Own contribution: Algorithm development and implementation of *TRE*; execution and evaluation of experiments including simulations; writing parts of the manuscript.

[P15] Jonas Posner, Lukas Reitz, and Claudia Fohry. "Task-Level Resilience: Checkpointing vs. Supervision". In: *Special Issue International Journal of Networking and Computing (IJNC)* 12.1 (2022), pp. 47–72. DOI: `10.15803/ijnc.12.1_47`
Extended version of [P12]: Added simulation of the execution of job sets on

supercomputers; re-formulation of the manuscript.
Own contribution: Most of the extension.

An important contribution of this thesis is to put the research outcomes from the various publications into a coherent context for the first time. Most publications are the result of joint research with co-authors, and a precise breakdown of individual contributions by authors is hardly possible. Nevertheless, we annotate the titles of specific sections that are chiefly the work of co-authors with footnotes, while all other sections are chiefly the work of the thesis author. In addition, we specify in the chapter introductions which section was adapted from which publication.

1.6 Structure

The remainder of this thesis is structured into the following chapters. After reading Chapter 2, each chapter can be read independently.

- *Chapter 2* provides background information on PGAS, *APGAS*, *GLB*, and dynamic independent tasks, as well as the benchmarks and the hardware environments used for experimental evaluations.
- *Chapter 3* focuses on the issue of load balancing. It starts by comparing cooperative and coordinated work stealing, followed by a description of the hybrid work stealing technique and the novel tasking constructs. Finally, *APGAS* is compared to *Spark* and *PCJ*, regarding performance and programmer productivity.
- *Chapter 4* addresses the issue of fault tolerance and describes the resilience techniques we have developed. After an introduction of the general ideas, it provides several implementation details, followed by experimental evaluations. Finally, we present formulas for predicting running times including recovery, and simulations of the execution of job sets on supercomputers.
- *Chapter 5* covers the issue of resource elasticity and describes the resource elasticity technique we have developed. Moreover, we present formulas for predicting running times for applications with a changing number of resources and a heuristic to determine job malleability parameters. Finally, the performance of the developed technique is evaluated experimentally and by simulating the execution of job sets.
- *Chapter 6* closes this thesis with conclusions and future work.

Chapter 2

Background

Contents

2.1 PGAS

The *Partitioned Global Address Space (PGAS)* model [39] aims to simplify the parallel programming of supercomputers. Thus, it facilitates programmer productivity while enabling high performance at scale.

PGAS is a high-level abstraction describing the memory of all computing resources involved in the computation as one global memory. Thus, a unified semantic for local and remote operations is provided. A *place* denotes a memory partition and associated computing resources. Typically, when using a supercomputer, each place corresponds to one node. Figure 2.1 shows an example with homogeneous multi-core nodes, to which this thesis is restricted. However, PGAS is generally not limited to this setting. For example, one can also start multiple places per node. Moreover, computational resources can also include, e.g., heterogeneous architectures such as GPUs.

PGAS programmers can easily access each memory partition from the global memory by using straightforward constructs. However, local access is faster than remote access. This way, PGAS hides the complexities of network communication, but not their existence, from the programmer.

Early PGAS implementations such as CAF [40], Titanium [44], and UPC [42] follow the *Single Program Multiple Data (SPMD)* execution style. At application launch, a fixed number of worker threads are spawned on each place, all executing the same code. However, each worker thread has an ID enabling different code paths for different worker threads.

Newer PGAS implementations such as X10 [49] and *APGAS* [16] introduced an asynchronous variant of the PGAS model as described in the following Section 2.2.

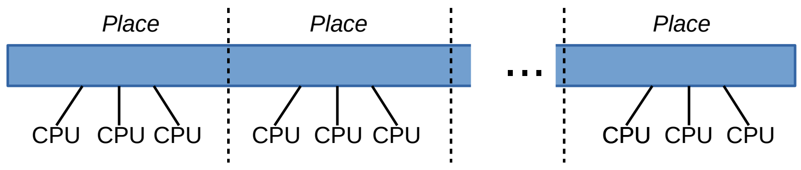


Figure 2.1: Partitioned Global Address Space Model

2.2 APGAS

The *APGAS* library for Java (briefly *APGAS*) [16] was released in 2015 as a branch of IBM's X10 project [115] and is available as open source [111]. *APGAS* brings the parallelization concepts of X10 to Java by using lambdas. There is also an *APGAS* variant for Scala [116], but throughout this thesis we refer to the Java variant.

The X10 project started in 2004 as part of the *Productive, Easy-to-use, Reliable Computing System (PERCS)* project funded by *DARPA's High Productivity Computing Systems (HPCS)* program [117]. X10 [49] is a parallel, object-oriented, and class-based programming language. The name X10 reflects the language's goal to raise programmer productivity by a factor of 10.

For parallelization, X10 and *APGAS* deploy an asynchronous variant of the PGAS [110] model. As in the previously described original PGAS, a *place* denotes a memory partition and associated computing resources. However, in the asynchronous variant of PGAS, computations are accomplished by (asynchronous) *activities*, which are executed by worker threads. At program start, one activity is started on Place 0 and executes the `main` method. Activities are mapped to computing resources of the respective place.

As formulated in [16], *APGAS "supports resilient, elastic, parallel, distributed programming on clusters of Java Virtual Machines (JVMs)"*. Technically, each place is realized by one process running one JVM, and workers are realized by Java threads managed by Java's `ForkJoinPool` [47, 48]. *APGAS* programmers express activities as Java lambdas.

APGAS provides different launchers for deploying applications on supercomputers. For instance, the `SSHLauncher` starts the places by using `ssh` on the remote nodes. The places are interconnected with the help of the Hazelcast Java library [118], whereby they are consecutively numbered with IDs starting from 0.

In the following, we present *APGAS* constructs and mechanisms for place-internal concurrency control. We then describe distributed data structures provided by *APGAS*, followed by the supported fault tolerance and resource elasticity functionalities. Finally, we show concrete code examples.

2.2.1 Constructs

APGAS programmers can use the following constructs, where *`{...}`* represents a user-implemented lambda:

- `here()`: Returns the ID of the calling place.
- `places()`: Returns a list of all places.
- `place(`*`<ID>`*`)`: Returns a place object matching the passed ID.
- `async({...})`: Spawns an *asynchronous* activity on the calling place. Depending on worker availability, the activity is executed immediately or later. In all cases, the `async` call returns immediately.
- `at(`*`<Place>`*`, {...})`: Spawns a *synchronous* activity on the passed place. The `at` call blocks until the spawned activity has terminated. If an exception is raised on the remote place, it can be caught on the original place by a surrounding `try-catch` block. Moreover, an `at` invocation transparently copies variables from the original place that are accessed on the remote place and sends them along. Thus, the lambda and the sent data must be serializable. All remote write accesses refer to the copied data and not to the original ones. In addition, programmers can return a value from the remote place to the original one.
- `asyncAt(`*`<Place>`*`, {...})`: Combines `async` and `at` to spawn an asynchronous activity on the passed place.
- `finish({...})`: Activities can be spawned within a `finish` block. If so, the parent activity waits at the end of the block until all spawned asynchronous activities, including recursively and/or remotely spawned activities, have terminated. Only then, the program execution is resumed. Figure 2.2 visualizes a `finish` block that encapsulates multiple nested activities on multiple places. Inside a `finish` block, multiple exceptions thrown inside asynchronous activities are combined into one exception. The latter can be caught by a `try-catch` block surrounding the `finish` on the original place.
- `uncountedAsyncAt(`*`<Place>`*`, {...})`: Resembles `asyncAt`, but the termination of the spawned activity is not tracked by any surrounding `finish` block. Moreover, exceptions thrown by the activity are ignored.
- `immediateAsyncAt(`*`<Place>`*`, {...})`: Resembles `uncountedAsyncAt`, but starts a new Java thread on the remote place, which immediately executes the transferred activity. Such a thread runs concurrently to the worker threads.

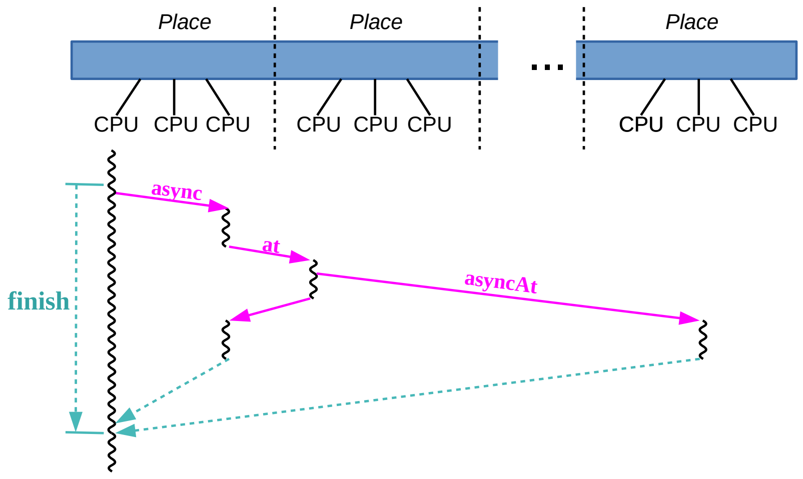


Figure 2.2: *APGAS*: Multiple activities within a `finish` block

2.2.2 Place-Internal Concurrency Control

APGAS does not provide constructs for place-internal concurrency control, but since workers are implemented by Java threads, the extensive capabilities of Java can be used. In the following, we briefly outline some Java concurrency capabilities that we will use later in this thesis. Further capabilities can be found in the official documentation of Java [119].

Java concurrency capabilities include the keyword `synchronized`, which can lock a whole method or a specified code block. `synchronized` does not guarantee fairness regarding the execution order of pending activities.

Further examples for Java's concurrency control include the data structures `AtomicBoolean` and `ConcurrentLinkedQueue`. `AtomicBoolean` is a thread-safe implementation of a traditional `boolean` variable. `ConcurrentLinkedQueue` implements a linked queue but manages data structure access in a thread-safe way.

Moreover, Java offers a wait-and-notify mechanism, which enables thread communication in locking situations. If a thread inside a `synchronized` block discovers that some condition is not fulfilled, it can call the function `wait()` on the lock object. Then, the lock is released and the thread waits until another thread calls the function `notify()` on this lock object. Because of the lock release, another thread can enter the `synchronized` block and apply changes that cause the condition to become fulfilled. In this case, it calls `notify()`.

2.2.3 Distributed Data Structures

APGAS offers several distributed data structures that allow objects to be stored across multiple places and accessed from any place. `GlobalRef<T>` is a global reference to a single object of type `T`. This object is stored on one place, but the other places can dereference the `GlobalRef` by calling `get()` and then access the object via network communication (`at`, `asyncAt`, etc.).

`GlobalRef<T> (List<Place>, {...})` stores one object of type `T` on each place of the passed list. Often, `places()` is used for this list if all places should be used. The objects are initialized by the passed lambda. If such a `GlobalRef` object is dereferenced on a particular place by calling `get()`, the respective local object (if any) is returned.

`IMap<K, V>` is an automatically distributed, concurrent, and fault-tolerant variant of a traditional key-value store provided by Hazelcast [120]. Fault tolerance functionalities are described in Section 2.2.4. Internally, an `IMap` is divided into partitions that are distributed evenly over all places. If a new place joins the computation, the partitions are automatically redistributed, so that they remain balanced. The same applies if a place leaves the computation, either intentionally or after a failure. Each key is unambiguously assigned to a single partition with a hash function, with entries also evenly distributed.

The `IMap` is thread-safe and provides a wide set of functions with different trade-offs between usability and performance. The most simple and efficient access functions are `get()` and `set()`, which correspond to thread-safe variants of the traditional map functions. Each call of `get()` and `set()` is independent and performs its own network operation. Therefore, multiple calls of those functions are carried out in any order. The `get()` and `set()` functions do not lock their `IMap` entry.

If multiple access operations refer to the same entry, they can be encapsulated in an `EntryProcessor`. It locks and unlocks the entry automatically and retains the order of operations. Such an `EntryProcessor` plus a key can be passed to the function `executeOnKey()`, which processes the `EntryProcessor` directly on the key owner. Therefore, network traffic is reduced, and the operations are executed atomically.

Moreover, multiple data structure accesses can be bundled in a transaction. Transactions guarantee atomicity, consistency, isolation, and durability of the included code. Instead of executing operations immediately on an `IMap`, transactions first lock all involved entries. Then, they perform the changes locally on data copies in a transaction context. Only if all operations have been successfully performed, the changes are committed to the real `IMap`, and afterwards the entries are unlocked. In cases of error, all previously executed operations are rolled back and a `TransactionException` is thrown. Error cases include place failures. If the transaction terminates successfully, all operations have been executed.

2.2.4 Fault Tolerance

APGAS supports user-level failure handling with respect to permanent place failures and provides the following types of failure notifications:

- A `DeadPlaceException` is thrown when a place failure occurs. Figure 2.3 extends Figure 2.2, but adds a visualization of the catching of thrown exceptions.
- Programmers can register a `PlaceFailureHandler` on each place, which is automatically invoked when any other place crashes.
- As part of Chapter 4, we extended *APGAS* and added the new construct `isDead(`*`<Place>`*`)` with which the liveliness of the passed place can be inquired.

As mentioned earlier, the `IMap` is fault-tolerant, which is achieved by automatically replicating each partition to other places internally. The number of replicas is configurable between zero and six. If a place leaves the computation, a backup of its hold data is still available (if the number of replicas has been set to at least one). So, the lost data can be automatically restored. A high number of replicas increases the availability of data, but also the network traffic.

If all owners of a replica leave the computation before the restore has been finished, a partition may get lost. For handling this case, programmers can register a `PartitionLostListener` on each place. It is automatically triggered and executes user-defined code.

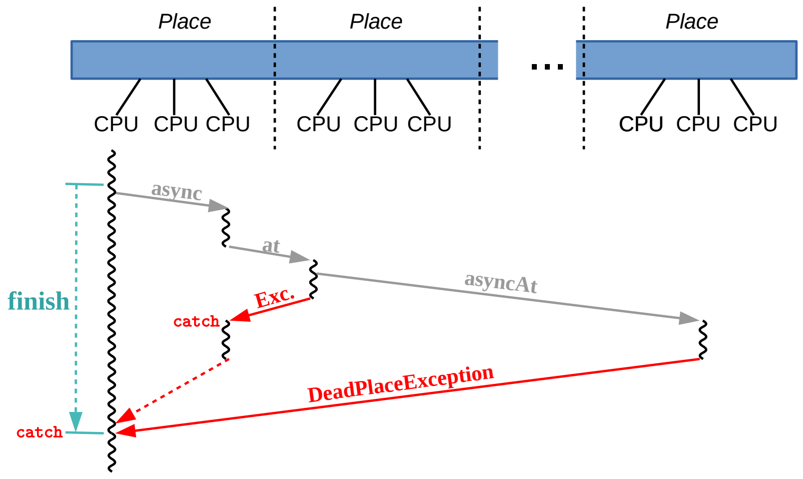


Figure 2.3: *APGAS*: Catching `Exceptions`

2.2.5 Resource Elasticity

APGAS supports user-level resource elasticity: It can integrate new places and release running ones on-the-fly. However, the usability of the resource elasticity of the original *APGAS* [16] is limited. In particular, new places must be started manually by users from outside the application on a respective remote node while passing parameters such as IP address and port of place 0. In addition, the original *APGAS* does not provide constructs allowing programmers to release places.

As part of Chapter 5, we have significantly improved the usability of resource elasticity and have resolved these shortcomings. Thus, we extended *APGAS* and added constructs for adding and releasing places. For that, Place 0 now maintains a bidirectional mapping of nodes to places, as well as all information (such as IP addresses) required to start new places on remote nodes and to safely shutdown places.

2.2.6 Code Examples

Listing 2.1 shows a simple, compilable *APGAS* program. *APGAS* constructs are highlighted in green. The program starts asynchronous activities (see Line 7) on Place 0 within a for-loop. All activities print *"Hello World"* and are encapsulated within a `finish` block (see Line 5) so that the program terminates only after all activities have been executed. Without the `finish` block, the program would terminate without guaranteeing that all activities have been executed.

```
 1 import static apgas.Constructs.*;
 2 
 3 public class HelloWorld{
 4   public static void main(String[] args) {
 5     finish(() -> {
 6       for (int i = 0; i < 42; i++) {
 7         async(() -> {
 8           System.out.println("Hello World");
 9         });
10     });
11   }
12 }
```

Listing 2.1: *APGAS*: `HelloWorld.java`

Listing 2.2 starts on each place (see Line 3) one synchronous activity (see Line 4) that prints out a *"Hello from"* plus the ID of the executing place (see Line 5). Due to the use of synchronous `at`, no `finish` block is required, and the output is sorted in ascending order by place IDs.

```java
public class HelloPlacesAt{
  public static void main(String[] args) {
    for (final Place p : places()) {
      at(p, () -> {
        System.out.println("Hello from " + here().id());
      });
    }
  }
}
```

Listing 2.2: *APGAS*: `HelloPlacesAt.java`

Listing 2.3 resembles Listing 2.2 but uses asynchronous activities (see Line 5). Consequently, a `finish` block is used (see Line 3), and the output is printed in an arbitrary order due to the asynchronism.

```java
 1 public class HelloPlacesAsyncAt{
 2   public static void main(String[] args) {
 3     finish(() -> {
 4       for (final Place p : places()) {
 5         asyncAt(p, () -> {
 6           System.out.println("Hello from " + here());
 7         });
 8     });
 9   }
10 }
```

Listing 2.3: *APGAS*: `HelloPlacesAsyncAt.java`

2.3 GLB

Global Load Balancing (*GLB*) provides dynamic load balancing for dynamic independent tasks (*DIT*) at runtime by deploying lifeline-based cooperative help-first work stealing. The lifeline scheme was first formulated formally [17] and then *GLB* was implemented as a reusable open source library as part of the official X10 distribution [115]. We denote the X10 implementation of *GLB* by GLB_{X10} [18].

In the following, we first define the DIT setting, followed by a description of *GLB*'s dynamic load balancing algorithm. We then outline the technical requirements of *GLB* that users/programmers must meet. Finally, we discuss internal implementation details of GLB_{X10}, as this is the starting point for our *GLB* variants in *APGAS* in Section 3.2.

2.3.1 DIT Setting

The dynamic independent tasks (*DIT*) setting has the following characteristics:

- Tasks are free of side effects.
- Tasks behave deterministically from the outside.
- Each task generates a result and possibly one or several child tasks.
- Each worker accumulates task results locally.
- All results have the same type.
- The final result is computed from these worker results by a reduction operation.
- The reduction operator is commutative and associative such as integer sum or maximum.

Listing 2.4 shows a DIT example as pseudocode (*not GLB* code) calculating the number of valid placements of N queens on an $N \times N$ chessboard (NQueens). It is invoked by calling `nqueens(new PosList(), 0)`. Upon termination, the result may be queried from the programming environment. Each `nqueens(...)` call represents a task.

```
void nqueens(PosList queens, int d) {
  if (d == n) {
    incrementResult();
  } else {
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        if (isValidPosition(queens,i,j)) {
          spawn nqueens(add(queens,i,j), d+1);
        }
      }
    }
  }
}
```

Listing 2.4: Dynamic Independent Tasks: NQueens

2.3.2 Dynamic Load Balancing

In *GLB*, each worker maintains its own local task pool. From there, the worker takes out tasks and processes them. Moreover, the worker inserts any newly generated tasks into its local task pool. At program start, at least one local pool contains at least one task.

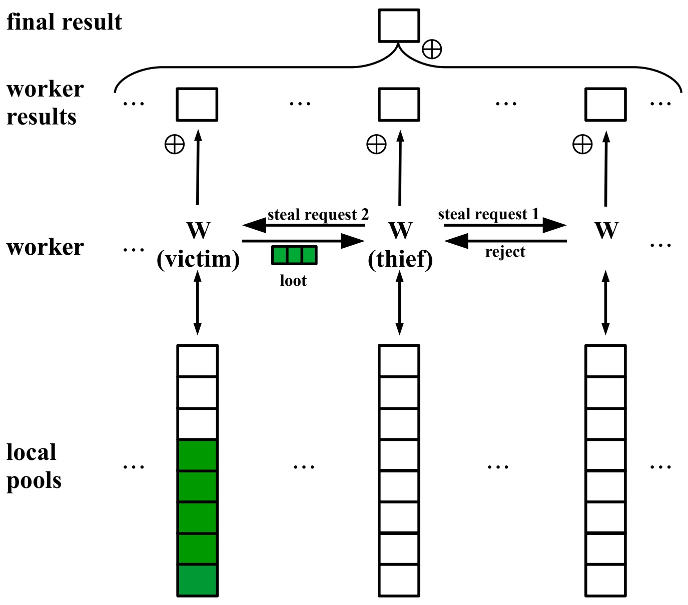


Figure 2.4: *GLB*: Cooperative work stealing

If initially there is only one task, a *dynamic* start is performed, and all other tasks are generated at runtime. If more than one task is known at program start, a *static* start can be performed. If so, each worker gets initially assigned a set of tasks and begins to process them. The distribution of the initial tasks to the workers can be defined by the user. On each place, exactly one worker is started. The specific data structure for the task pools must be implemented by the user.

If the local task pool of a worker runs empty, the worker tries to steal tasks from another worker. Correspondingly, the involved workers are called *thief* and *victim*, respectively. First, a thief tries to steal tasks from up to `w` random victims. If these attempts were unsuccessful, the thief tries to steal tasks from so-called *lifeline buddies*. For this, the workers are arranged in a so-called *lifeline-graph* [17]. Its N graph nodes represent the workers, and the outgoing edges are called *lifelines*. The lifeline graph is low-degree, low-diameter, and directed. The corresponding neighbor workers are called *lifeline buddies*. We consider a z-dimensional torus with up to l nodes per dimension. If $N < l^z$, the last dimension is filled up level-wise. Thus, a worker has either `z` or `z - 1` lifeline buddies. `w` and `z` are configurable by the user.

If a thief tries to steal tasks from a lifeline buddy, the corresponding lifeline from the thief to the victim is activated. This is accomplished via a flag on the thief worker indicating that there is an open request to this victim. A thief does not steal from a lifeline buddy if the lifeline is already activated. When a lifeline buddy sends tasks to the thief, the corresponding lifeline is deactivated.

GLB implements *cooperative* work stealing. As visualized in Figure 2.4, a thief sends a steal request and waits for the answer. Thieves are not allowed to access the victim's task pool directly. After receipt of the steal request, the victim responds by sending tasks, called *loot*, or a reject message if it has no tasks to share. If the victim is a lifeline-buddy and has no tasks, it additionally stores the request and tries to send tasks later.

If all `w + z` steal requests were unsuccessful, the thief goes inactive. An inactive worker can only be reactivated by a lifeline-buddy that sends tasks later. On Place 0, the inactivity of all workers are recognized. If so, the overall computation has been completed. Then, on Place 0 the final result is computed by collecting and reducing all partial results.

With a static program start, all lifelines are closed at program start.

2.3.3 Technical Requirements

GLB users must implement the task pool data structure, called `TaskQueue`, that supports the following operations:

- `boolean process(n)`: Takes out up to `n` tasks from the local task pool, processes them, and inserts any newly generated tasks into the local task pool again. Thus, it contains the specific implementation of the computation. The method returns `true` if the task pool is not empty after processing the `n` tasks and `false` otherwise.

- `TaskBag split()`: Extracts a set of tasks from the pool and returns them as a `TaskBag` object. `TaskBag` must also be implemented by the *GLB* user and represents a simple container for tasks. The `split()` method is called GLB internally by victims, and the return value corresponds to a piece of loot. The number of tasks to be extracted depends on the user implementation, typically it is half of the current tasks. However, *GLB* requires that at least one task must remain in the pool.

- `void merge(TaskBag)`: Inserts the loot contained in the passed `TaskBag` into the pool. This method is called by thieves when receiving loot.

- `GLBResult<T> getResult()`: Returns the result of the worker as a `GLBResult<T>` object. `GLBResult<T>` is a user class that represents a computed result in the form of an array, where the element type `T` of the array is defined by the *GLB* programmer.

2.3.4 X10 Implementation GLB$_{\mathrm{X10}}$

In this section, we discuss several internal implementation details of $\mathit{GLB}_{\mathit{X10}}$. Even if *GLB* users do not need to be aware of them, they are certainly relevant for the development of our variants in *APGAS* in Section 3.2.

Listing 2.5 shows a simplified version of a $\mathit{GLB}_{\mathit{X10}}$ worker's main loop as pseudocode. Steal attempts (Line 7) and task deliveries (Line 5) are realized by asynchronous remote activities. Since $\mathit{GLB}_{\mathit{X10}}$ permits only a single thread per place, all activities of a place are queued until the worker calls `Runtime.probe()` (Line 4). The parameter `n` in Line 3 controls the frequency of interrupting task processing. When `Runtime.probe()` is reached, all queued activities, which correspond to steal requests and task deliveries from other workers, are run sequentially in any order.

Steal attempts check whether the remote task pool is empty by inspecting an `empty` flag. If so, they inform the thief, who essentially blocks waiting for an answer in the meantime. If not, they register the thief by entering its identity in a registration queue at the victim place. When the victim resumes after `Runtime.probe()`, it sends out tasks to the registered thieves. If there are not enough tasks for everybody, a rejection message is sent to the remaining thieves.

```
1 while (tasks available) {
2   while (task pool is not empty) {
3     process up to n tasks;
4     Runtime.probe();
5     send tasks to registered thieves;
6   }
7   attempt to steal from up to w+z victims;
8 }
```

Listing 2.5: $\mathit{GLB}_{\mathit{X10}}$: Worker's main loop

2.4 Multi-Worker GLB

$\mathit{GLB}_{\mathit{multi}}$ is an extension of *GLB* that resolves *GLB*'s restriction of one worker per place, thus allowing multiple workers per place. $\mathit{GLB}_{\mathit{multi}}$ has first been proposed for X10 [70], and later for *APGAS* [19]. We refer to the *APGAS* variant, which is available as open source [121].

GLB$_{multi}$ workers are threads that run independently. Each worker maintains its own local task pool, from which it takes tasks for processing, and where it inserts any new tasks. These local task pools are strictly private, i.e., only the owning worker has access. Additionally, on each place two shared pools, called `intra-pool` and `inter-pool`, are maintained to which all local workers have access.

GLB$_{multi}$ combines lifeline-based work *stealing* for *inter*-place load balancing with work *sharing* for *intra*-place load balancing. The `intra-pool` serves for sharing work between the local workers, and the `inter-pool` is primarily used to answer steal requests from other places. As in the previously described *GLB* variant, each worker alternates between task processing and communication phases. A communication phase starts after processing `n` tasks and comprises the following actions in sequence:

- If there is an inactive local worker, a certain number of tasks are extracted from the local task pool and given to it, and it is started.
- If `intra-pool` is empty, a certain number of tasks are extracted from the local task pool and inserted into `intra-pool`.
- If an incoming inter-place steal was answered by sending tasks since the worker's last communication phase, a certain number of tasks are extracted from the local task pool and inserted into `inter-pool`.

If a worker runs out of tasks, it takes out all tasks from `intra-pool` and continues. If `intra-pool` is empty, it takes out all tasks from `inter-pool` instead and continues. If both `intra-pool` and `inter-pool` are empty, the worker becomes inactive. If all local workers are inactive, the inter-place work stealing begins.

When a place runs out of tasks and becomes a thief, it first contacts up to `w` random victims, and if not successful, up to `z` lifeline buddies afterwards (as in Section 2.3). If a victim's `inter-pool` is empty, the victim responds with a reject message. Otherwise, it sends all tasks from the `inter-pool` as a response. Lifeline buddies record rejected steal requests. If all `w + z` steal attempts were unsuccessful, the thief goes inactive. An inactive thief is restarted with one local worker when a lifeline buddy sends tasks in reaction to an earlier recorded steal request. When all places have become inactive, all tasks have been processed, and the final result is computed.

2.5 Benchmarks

In the following, we describe the benchmarks we used to experimentally evaluate our implementations.

2.5.1 Unbalanced Tree Search (UTS)

UTS [122] dynamically generates a highly unbalanced tree at runtime, starting from a root value. Each tree node is represented by an individual hash value. The hash value of the root node is generated using the initial seed. The calculation of a node is based on its hash value and generates new nodes, each with a new individual hash value. Thus, the total number of nodes is initially unknown, and each node must be computed to obtain the complete tree. The final result of a program run is the total number of tree nodes, i.e., a single `long` value.

The tree properties can be configured by the user as follows:

- tree depth `d`,
- branching factor `b`, i.e., maximum number of children per node,
- initial seed `s`, and
- tree shape (binomial or geometric distribution).

UTS is well suited to simulate irregular workloads and thus to evaluate dynamic load balancing techniques. The X10 distribution contains an *UTS* implementation using GLB_{X10} [115]. This implementation starts with only one task, which represents the root tree node. This task is processed by worker 0, and thus new tasks are generated. These new tasks are subject to work stealing such that dynamic load balancing is achieved.

The GLB_{X10} implementation stores the tasks in a compact format: tree nodes are not explicitly stored as self-contained tasks, but a local task pool uses three arrays (`lower`, `upper`, and `hash`) to store all nodes. Each entry `i` describes the hash value of a parent `hash[i]` and the numbers of children (`upper[i]`, `lower[i]`) to be calculated. Thus, three related values from `hash`, `lower`, and `upper` exactly represent $(upper - lower)$ tasks. We denote this compact variant by UTS_C.

Unfortunately, this compact format cannot be implemented as a dequeue, which is a requirement of some of our fault tolerance techniques, e.g., *IncTC*. Fohry *et al.* [20] used an *UTS* variant that explicitly stores each tree node as a distinct task. We denote this explicit variant by UTS_E. We have transferred both UTS_C and UTS_E to Java in a straightforward way.

2.5.2 Betweenness Centrality (BC)

BC [123] computes a score for each node in a given directional, weighted, and acyclic graph. This score rates the centrality of a node x and is calculated as follows:

$$score(x) = \sum_{s \neq v \neq t} \frac{\sigma_{s,t}(x)}{\sigma_{s,t}},$$

where

- $\sigma_{s,t}$ is the number of shortest paths from node s to node t, and
- $\sigma_{s,t}(x)$ is the number of shortest paths from node s to node t that pass through node x.

A high score of a node indicates that this node is part of many shortest paths. The graph properties can be configured by the user with the number of nodes $2^{\mathtt{n}}$ and the initial seed `s`. The given graph is generated using the initial seed. The result of a program run is an array of `double` values that stores for each node its score.

Like for *UTS*, the X10 distribution contains a *BC* implementation using GLB_{X10} [115]. Here, a task executes the following steps:

1. For each node s, all shortest paths from s to all other nodes are calculated.
2. For all nodes x that are on a shortest path from s to another node t, the value of $score(x)$ is updated.

In contrast to *UTS*, the graph is known from the beginning, therefore all tasks are also known from the beginning and no new tasks are generated at runtime. Therefore, a static start can be performed, i.e., each worker gets initially assigned a set of tasks and begins to process them.

Like for *UTS*, the GLB_{X10} implementation stores the tasks in a compact format. Here, entries in the arrays `lower` and `upper` define intervals of nodes.

Again, Fohry *et al.* [20] introduced an explicit variant for *BC*, and we denote the compact variant by BC_C and the explicit one by BC_E. We have transferred both BC_C and BC_E to Java in a straightforward way.

2.5.3 NQueens

NQueens [124] calculates the number of placements of N queens on an $N \times N$ chessboard such that no queen threatens any other queen. The result of a calculation is a single `long` value. `N` can be configured by the user.

We adapted the algorithm from reference [125]. A task represents a chessboard with $k < N$ validly positioned queens. The computation is started with a single task that represents an empty chessboard.

Processing a task checks for each free square whether a new queen can be placed on it. If a queen can be placed on a square, a new task is created, which represents the chessboard extended by the queen. If there are only a specified number of queens (`threshold`) left to be placed, the task is processed sequentially, and no new tasks are created. The `threshold` can be configured by the user.

2.5.4 Pi

This benchmark approximates π using the well-known Monte Carlo algorithm [126]. It is a simple, compute-intensive benchmark that generates `n` points within a unit square and counts the number of points that fall within the corresponding unit circle. Then, the value of π is calculated with the following formula:

$$\pi = 4 \cdot \frac{numInside}{n}$$

The number of random points per task is calculated by the formula $n/(numWorker \cdot tasksPerWorker)$, where n and $tasksPerWorker$ can be configured by the user. All tasks are statically known and are evenly distributed at the beginning. No new tasks are generated at runtime. The final result of the computation is, obviously, a `float` value that is approximately π.

We have implemented this benchmark from scratch.

2.5.5 Matrix Multiplication

MatMul multiplies two square matrices of `double` values and produces a new square matrix as result. Each matrix is divided into `N` blocks of `M` `double` values. `N`, `M`, as well as an initial seed `s` can be configured by the user. The two matrices are generated using the initial seed.

We adapted the implementation from reference [127]. All tasks are statically known and are evenly distributed at the beginning. No new tasks are generated at runtime. For a multiplication of two blocks, a simple technique using a triple-nested loop is implemented.

2.5.6 Word Count

WordCount is a canonical map-reduce application. A specified number of input files are read line-by-line, and the occurrences of each word are counted and stored as individual key/value pairs. For parallelization, reading and counting are distributed, and partial results are computed locally first. The reduction step accumulates the partial results. The result of a run is a key/value map.

We have implemented this benchmark from scratch.

2.5.7 Travel Salesman Problem

TSP [128] determines the shortest round trip through a list of cities that are connected by different distances. We deploy a parallel branch and bound algorithm with heuristics [129]. Each place holds a local optimum. Whenever a worker discovers a new local optimum, it propagates its new local optimum to the other workers. The computation is started with a single task. If there is only a specified number of cities (`threshold`) left to visit for a round trip to be computed, the rest is processed sequentially, and no new tasks (for that combination) are created.

The number of cities `c`, the `threshold`, and the initial seed `s` can be configured by the user. The city graph is generated using the initial seed.

We have implemented this benchmark from scratch.

2.5.8 Synthetic Benchmarks

We have developed two groups of synthetic benchmarks, both designed to evaluate specific scenarios as described in the following.

2.5.8.1 Simulation of Large Task Pools

We designed the first group of synthetic benchmarks with the aim of simulating large task pools. Thus, their task descriptor size is adjustable by the user with a dummy ballast `b`. The benchmarks compute π with a Monte Carlo algorithm and user-defined precision `g`.

We denote the first benchmark by *VBS* – **V**ariable **B**allast for **S**tatic tasks. The static tasks are evenly distributed across the workers at the beginning. The number of tasks `t` can be configured by the user.

The second benchmark is called *VBD* – **V**ariable **B**allast for **D**ynamic tasks. It starts with a single task. Each task dynamically spawns multiple child tasks at runtime until a user specified depth `d` is reached. The number of child tasks is randomly selected from a user specified range.

2.5.8.2 Smooth Weak Scaling

The second group of synthetic benchmarks was designed with the aim of supporting smooth weak scaling. Therefore, these benchmarks perform some placeholder computations but the user provides a desired running time $\widehat{T}(p)$. A run takes time $T(p) = \widehat{T}(p) + \epsilon$ with p workers, where ϵ reflects the costs of dynamic load balancing. In other terms, a run takes time $T(p) = \widehat{T}(p) \cdot (1 + L)$ with p workers, where L reflects the overhead for load balancing as a percentage of $\widehat{T}(p)$.

Moreover, the user can influence the number of tasks per worker `t`, as well as specify a fluctuation range `f` for the task durations.

We denote the first benchmark by *SWSS* – **S**mooth **W**eak **S**caling for **S**tatic tasks. It deploys static tasks, which are evenly distributed across the workers at the beginning. Task durations in *SWSS* are varied per worker. For example, for an average task duration of 10 ms and `f` = 20%, one worker may obtain `t` tasks with a duration of 8 ms each, and another `t` tasks with a duration of 12 ms each (random times within the fluctuation range).

The second benchmark is denoted by *SWSD* – **S**mooth **W**eak **S**caling for **D**ynamic tasks. It starts with a single task. *SWSD* generates a perfect m-ary task tree dynamically at runtime, where m and `t` are automatically chosen/adjusted.

2.6 Hardware Environments for Experiments

We deployed the following hardware environments for our experimental evaluations:

- *Kassel* [130] provides a partition containing 12 homogeneous InfiniBand-connected nodes, each with two 6-core Intel Xeon E5-2643-v4 CPUs and 256 GB of main memory. The partition is owned by the department and was the only one available without time constraints for the entire period of this thesis.
- *Goethe*-HLR [131] provides homogeneous InfiniBand-connected nodes, each with two 20-core Intel Xeon Gold 6148 CPUs and 192 GB of main memory.
- *Lichtenberg* I Phase 1 [132] comprised InfiniBand-connected nodes, each equipped with two 8-core Intel Xeon E5-2670 CPUs and 32 GB of main memory. It was discontinued in 2020.

On all machines, we mapped the processes/places *cyclically* onto the lowest possible number of nodes. For instance, when we used *Kassel* and started 24 places on two nodes, we mapped Place 0 onto Node 0, Place 1 onto Node 1, Place 2 onto Node 0, and so on.

Chapter 3

Load Balancing

Contents

3.1 Introduction

In this chapter, we present our contributions to the open research questions regarding load balancing. As outlined in Section 1.4.1, we first contribute to the open research question of which load balancing technique has the best performance. For that, we compare cooperative and coordinated work stealing in the context of *GLB* in Section 3.2. Experimental results show minor performance differences between the two variants.

Then, in Section 3.3, we design and implement the novel hybrid work stealing technique achieving both *intra*- and *inter*-process load balancing. We implement our technique by extending *APGAS* and introduce several novel tasking constructs. In doing so, we aim to make sure that the tasking constructs are generic, flexible, feature-rich, and user-friendly, as prompted by one the open research questions. We denote the result by $APGAS_{hyb}$. Experimental results show good scalability in most cases.

Afterwards, in Section 3.4, we contribute to the open research question whether AMT can be a one-fits-all solution. To this end, we compare $APGAS_{hyb}$ with *APGAS*, the data analytics library *Spark* [112], and the HPC library *PCJ* [45]. Our comparison evaluates the performance of the programming environments for HPC and data analytics applications as well as their programmer productivity. $APGAS_{hyb}$ turns out best, making it a strong candidate for programming both HPC and data analytics applications.

Finally, we conclude this chapter with related work and conclusions in Sections 3.5 and 3.6, respectively.

This chapter was adapted from publications as follows: Section 3.2 from [P1], Section 3.3 from [P5, P6, P11], and Section 3.4 from [P8].

3.2 Cooperative vs. Coordinated Work Stealing

3.2.1 Problem Description

Work stealing can be accomplished in either a *cooperative* or a *coordinated* way. In the cooperative approach, the thief sends a message to the victim, asking for tasks. Workers occasionally interrupt task processing to answer such requests, and then send tasks or a reject message. Local pools are therefore private to their respective owners.

In the coordinated approach, in contrast, a thief directly accesses the victim's pool and takes out tasks if available. Pool data structures must consequently support concurrent access. A well-known data structure for this purpose is the split queue [113]. It deploys a ring buffer and divides it into a private and a public portion. The owner operates on the private portion, whereas thieves take out tasks from the public portion. Between the two portions, tasks can be moved at low cost.

Cooperative and coordinated work stealing have been shown to be roughly equivalent in theory and practice [69]. Whereas the analysis in [69] only considered random victim selection and shared-memory systems, we extend their studies by performing the comparison for lifeline-based victim selection and distributed-memory systems.

Prior to our work, lifeline-based work stealing had only been implemented in the *cooperative* way in GLB_{X10} [18]. However, the core idea of the lifeline algorithm is neither tied to cooperation nor to sequentialization of activities, as GLB_{X10} does. We devise a *cooperative GLB* variant without sequentialization and a *coordinated GLB* variant, and then compare coordinated and cooperative work stealing.

For practical reasons, we implement both *GLB* variants, *cooperative* and *coordinated*, in *APGAS*. The cooperative variant is denoted by GLB_{coop} and the coordinated variant is denoted by GLB_{split}.

This section is organized as follows. Sections 3.2.2 and 3.2.3 explain the cooperative and coordinated schemes, respectively. Section 3.2.4 discusses experimental results, and Section 3.2.5 wraps up the obtained results.

3.2.2 Cooperative Scheme

GLB_{coop} resembles GLB_{X10} but resolves the restriction that all activities at the same place are sequentialized. For that we replaced GLB_{X10}'s `Runtime.probe()`-centric approach (see Listing 2.5) by Java synchronization constructs, in particular by `synchronized` sections. Thus, some activities, such as the registration of a thief at a victim place, can run in parallel to task processing. The replacement was also necessary since *APGAS*, unlike X10, offers no mechanism to enforce a single thread per place. Moreover, *APGAS* does not provide `Runtime.probe()` or a similar functionality for activity interruption.

Our cooperative scheme implements exclusive access to the local pools with Java's `synchronized` blocks. Listing 3.1 shows a worker's main loop of GLB_{coop} as simplified pseudocode. The worker's main loop of GLB_{X10} was shown in Listing 2.5.

Note that both `synchronized` blocks refer to the same local worker. At a remote place, in contrast, steal attempts do not acquire a lock. Locking is not needed, since steals do

```
 1 while (tasks available) {
 2   while (task pool is not empty) {
 3     synchronized (worker object) {
 4       process up to n tasks;
 5       send tasks to registered thieves;
 6     }
 7   }
 8   synchronized (worker object) {
 9     attempt to steal from up to w+z victims;
10   }
11 }
```

Listing 3.1: *GLB$_{coop}$*: Worker's main loop

not access the remote pool, but only refer to an `empty` flag and, possibly, the registration queue. Concurrent access to these variables is implemented with Java's `AtomicBoolean` and `ConcurrentLinkedQueue` constructs, respectively.

In consequence, steal attempts can be performed in parallel to the remote worker's task processing (Line 4) and stealing (Line 9), where possible conflicts are resolved by the `empty` flag. Steal attempts can also be performed in parallel to task deliveries (Line 5).

Unlike steal attempts, task deliveries deploy a `synchronized` block at the remote place, to safely insert their tasks into the task pool. This synchronization approach comes close to cooperative work stealing insofar as the worker keeps control of its pool to the extent that it may decide when others get access.

Java's `synchronized` blocks do not guarantee fairness. Therefore, it may happen that the worker regains access to the next `synchronized` block right after leaving the previous one in Line 6, although there are other activities waiting. This does not compromise the program's progress, since the other activities can be task deliveries only. Task deliveries are invoked with `async`, and thus their delay does not restrain the sender. It does not restrain the receiver either, since it has enough tasks in its pool in Lines 2-7. Moreover, resource consumption of the queued activities is unproblematic since their number is limited by `z`.

Line 9 depicts all steal attempts of a victim together. The implementation of this line is somewhat more complex. Within the `synchronized` block, the worker calls `wait()`, and thus releases the lock, whenever it waits for an answer. Consequently, task deliveries and reject messages can get into the block. Note that task deliveries may origin from the victim just attempted, or from a lifeline buddy who recorded the thief long ago. In any case, an arriving delivery or reject wakes up the worker with `notify()`.

Apart from differences in synchronization, we tried to keep *GLB$_{coop}$* as close as possible to *GLB$_{X10}$*. This was straightforward due to the close correspondence between X10 and *APGAS*. Further implementation details can be found in [133].

3.2.3 Coordinated Scheme

GLB$_{split}$ allows efficient task extraction by thieves while victims can continue processing tasks. This requires a data structure that allows concurrent access.

We use the split queue from [113] that deploys a ring buffer divided into a private and a public portion. The owner operates on the head of the private portion, whereas steals and task deliveries access the tail of the public portion. Accesses to the private portion need not be locked, which speeds up the owner's regular task processing. Accesses to the public portion must be locked, however, to enable concurrent access by the owner (see below), as well as by one or several thieves and lifeline-based task deliveries (see below).

As for *GLB*$_{coop}$, locking is accomplished by `synchronized` blocks. Unlike there, thieves directly take out tasks from the pool, such that steal attempts return instantly. At the thief place, the stolen tasks are inserted at the tail of the public pool by the worker itself. If no tasks were found, the steal attempt returns instantly, as well. If it was a lifeline steal, it priorly records the thief in a `ConcurrentLinkedQueue`, as in *GLB*$_{coop}$.

Between the private and public portions, data are moved with methods `release()` and `reacquire()`. Method `reacquire()` moves tasks from the public to the private portion. It is called by the owner when it has consumed the last task. Method `release()` moves tasks from the private to the public portion. It must be called regularly, to adjust the ratio between the two portions. Both methods are fast, since they only move the division line between the portions by incrementing/decrementing an index. They do not physically move tasks.

As argued in [113], method `reacquire()` needs a lock, since the owner and a thief may try to take out tasks at the same time. In contrast, method `release()` is lock-free since it only adds tasks, which cannot interfere with a thief's task removal, and likewise not with lifeline-based task deliveries, since these operations access the opposite end of the public portion. At the worst, a thief may see a too small size of the public portion and therefore steal fewer tasks than would be possible with the correct size. Since the index of the division line is an integer, correctness of this approach also relies on Java's atomicity for integer access.

As mentioned in Section 2.3, *GLB* is structured in such a way that the task pool data structure must be implemented in user code. Therefore, we programmed a split queue in Java, and integrated it into the *UTS*$_E$ and *BC*$_E$ benchmarks. We chose the explicit variants of the benchmarks because the split queue requires the tasks to be stored explicitly. In addition, *GLB* itself had to be modified, to accommodate direct access to the remote pools. Listing 3.2 shows a worker's main loop of *GLB*$_{split}$ (again in simplified pseudocode).

```
1 while (tasks available) {
2   while (task pool is not empty) {
3     process up to n tasks
4     release();
5     send tasks to registered lifeline thieves;
6   }
7   attempt to steal from up to w+z victims;
8 }
```

Listing 3.2: *GLB$_{split}$*: Worker's main loop

In comparison to the main loop of *GLB$_{coop}$* in Listing 3.1, the `synchronized` blocks have become superfluous, since the `release()` (Line 4) and `steal()` (Line 7) methods comprise `synchronized` blocks. Furthermore, explicit task delivery is only needed for lifeline thieves that could not help themselves for lack of tasks at the time of their access. If these deliveries reactivate the remote worker, the method `reacquire()` is called on the remote place to move tasks to the public portion. Finally, the regular `release()` call was added to the main loop (Line 4) in order to balance the private/public ratio periodically.

Our split queue implementation provides an option to configure the ratio between the private and public portions, as well as the number of tasks that are stolen. We carried out systematic parameter studies as will be described in Section 3.2.4. They showed that ratio 50:50 in combination with a steal-half policy performs best in general. Individual program runs can slightly profit from specific parameter settings, but we consistently used 50:50 and steal-half for all experiments in Section 3.2.4.

3.2.4 Experiments

Experiments were carried out on *Lichtenberg* [132]. We started up to 512 places which were cyclically assigned to up to 128 nodes. Thus, we started 4 places per node as discussed below. Java was used in version 8.0. *APGAS* and X10 2.5.4 were used in their revisions of February 29, 2016, from the official git repository [115]. X10 programs were compiled into Java with Managed X10.

As benchmarks, we utilized *UTS$_E$* and *BC$_E$*. As described in Section 3.2.3, the explicit variants of the benchmarks are required for *GLB$_{split}$*, but for comparability they were used consistently in all experiments. Both benchmarks were run with a large and a small configuration (see below). Values for the *GLB* parameter n (the number of tasks per step) were determined experimentally, so as to minimize the running time.

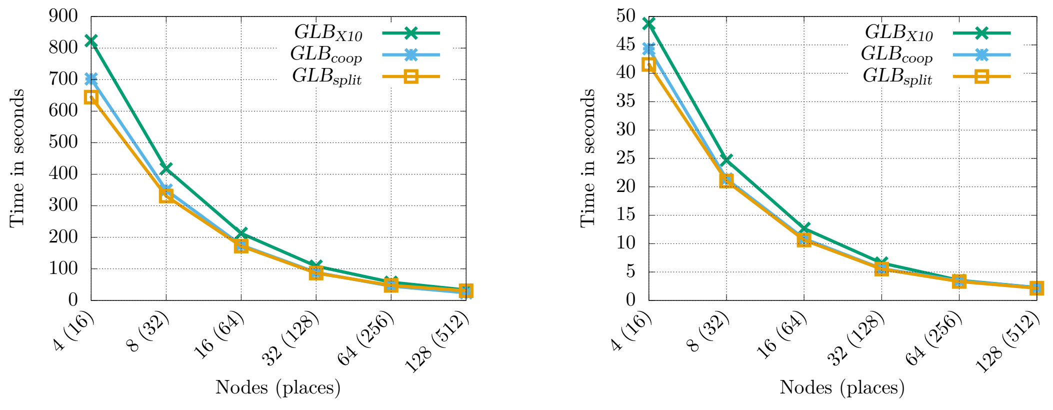
- *UTS$_E$*: geometric tree shape, branching factor $b = 4$, initial seed $s = 19$, tree depth $d = 13$ (small configuration) and $d = 17$ (large configuration), $n = 511$.

- BC_E: initial seed $s = 2$, number of graph nodes $N = 2^{14}$ (small configuration) and $N = 2^{16}$ (large configuration), $n = 127$.

Figure 3.1 depicts the results for the large configurations. Note that the horizontal axes are log-scaled. Results for the small configurations were similar. The place assignments for Figure 3.1 mapped 4 places to each node and left some cores idle. However, we observed a similar performance in preliminary experiments with the small UTS_E configuration, when mapping up to 8 places to each node. When mapping 16 places to each node, the performance of all program variants declined.

In all experiments, GLB_{X10} consistently had the highest running time. For UTS_E on up to 128 places, GLB_{coop} is the fastest variant, with GLB_{split} being slower by 2% to 9%. On 256 and 512 places, GLB_{split} performs best, running up to 22% faster than GLB_{coop}. For BC_E, GLB_{split} is the fastest variant consistently. Here, the difference between the GLB_{coop} and GLB_{split} is 2% to 6%

In summary, both the GLB_{coop} and GLB_{split} variants have lower execution times than GLB_{X10}, but between GLB_{coop} and GLB_{split} there is no clear winner. A possible reason for the performance gain of both GLB_{coop} and GLB_{split} over GLB_{X10} may be the place-internal parallelism. In GLB_{coop}, thieves are registered and theft attempts are rejected in parallel to task processing. Thus, GLB_{coop} makes better use of the victim place's processing resources. Moreover, thieves are notified earlier about unsuccessful steal attempts, and can contact the next victim right away. In GLB_{split}, tasks can be stolen in parallel to task processing thanks to the split queue, which also leads to a better use of the victim place's processing resources.



(a) UTS_E **(b)** BC_E

Figure 3.1: Performance of GLB_{X10}, GLB_{coop}, and GLB_{split}

3.2.5 Wrap Up

In this section, we have introduced a cooperative and a coordinated variant of *GLB* for *APGAS*, denoted by GLB_{coop} and GLB_{split}, respectively. Both outperform the original GLB_{X10}, when compiled with Managed X10. The performance difference between GLB_{coop} and GLB_{split} is small, confirming a previous result for shared-memory architectures from the literature. However, GLB_{split} requires a specific user-implemented data structure for the task pools that allows effective concurrent access.

3.3 Hybrid Work Stealing

3.3.1 Problem Description

As stated in the Introduction, clusters of multicore nodes are the prevalent architecture in high-performance computing today. To use them efficiently, parallel programs should simultaneously exploit both node-internal shared-memory parallelism and inter-node distributed-memory parallelism.

Various applications deploy locality-flexible tasks that may run on any resource of the overall system. For such tasks, programmers do not want to specify a placement, as it is required for all activities in *APGAS*. Instead, it is preferable that a runtime system places the tasks dynamically to balance the load.

The HabaneroUPC++ programming library tackles this issue by introducing an `asyncAny` construct for spawning locality-flexible tasks [134]. Unfortunately, the cited publication does not report speedup values, and we were unable to obtain speedups with the code provided by the authors [125].

Other asynchronous PGAS programming environments such as X10 and *APGAS* from [16] do not yet support locality-flexible tasks, and *GLB* allows only a single worker per place. As a result, multicore nodes could so far only be exploited by starting multiple places per node, which causes unnecessary communication and increases the memory load.

In this section, we introduce a hybrid work stealing technique that supports multiple workers per place. For that, we adopt the concept of `asyncAny` tasks from HabaneroUPC++. However, we develop a fundamentally different algorithm, implement our technique directly in *APGAS*, and denote the result by $APGAS_{hyb}$. We combine the *intra*-place load balancing of Java's `ForkJoinPool` [47, 48] with the *GLB* lifeline scheme for *inter*-place load balancing. HabaneroUPC++, in contrast, uses the SLAW scheduler [135] for intra-place load balancing and, in inter-place load balancing, selects a suitable victim with the help of *Remote Direct Memory Access* (RDMA). Moreover, the HabaneroUPC++ technique contacts an unlimited number of remote victims, whereas our technique contacts a fixed number of random victims and lifeline buddies as defined by the lifeline scheme.

In addition to `asyncAny`, we introduce several related constructs. They include a new `finish` construct, called `finishAsyncAny`, that is optimized for `asyncAny` tasks, and support for calculating an overall result by reduction from task results. Another group of constructs enables the cancellation of `asyncAny` tasks, or more specifically, the dequeuing of all unprocessed tasks. Cancellation is useful for search problems.

This section is organized as follows. Section 3.3.2 introduces the novel tasking constructs and demonstrates their usage with an example. The hybrid work stealing technique and its implementation are described in Section 3.3.3. Experimental results are presented and discussed in Section 3.3.4. Finally, we wrap up the obtained results in Section 3.3.5.

3.3.2 Programming with `asyncAny` Tasks

To enable the usage of our hybrid working stealing technique, we provide the following novel tasking constructs in $APGAS_{hyb}$:

- `asyncAny`: Submits a locality-flexible task. The task is initially placed in the local task pool and can later be stolen away by other workers.
- `finishAsyncAny`: Suspends until all `asyncAny` tasks that have been directly or recursively spawned in an associated block have been processed.
- `staticInit`: Creates a copy of static data (e.g., constants) on each place.
- `staticAsyncAny`: Resembles `asyncAny`, but expects a list of tasks as parameter.
- `mergeAsyncAny`: Merges a passed task result into the partial result of the local worker.
- `reduceAsyncAny`: Computes the current global result by reduction over the partial results of all workers and returns it.
- `cancelableAsyncAny`: Resembles `asyncAny`, but the submitted task can be canceled.
- `cancelableStaticAsyncAny`: Combines the above `cancelableAsyncAny` and `staticAsyncAny` constructs.

- `cancelAllCancelableAsyncAny`: Cancels all unprocessed `cancelableAsyncAny` and `cancelableStaticAsyncAny` tasks and prohibits spawning new ones.

Listing 3.3 shows a *Hello World* example using `asyncAny`. The `finishAsyncAny`-call detects when all `asyncAny` tasks that are spawned in the loop have been processed. The `asyncAny` tasks just print *“Hello World”* and the ID of the executing place.

```
finishAsyncAny(() -> {
  for (int i = 0; i < N; i++) {
    asyncAny(() -> {
      System.out.println("Hello from " + here());
    });
  }
});
```

Listing 3.3: *APGAS*$_{hyb}$: Submitting N `asyncAny` tasks within a `finishAsyncAny` block

Codes for *Pi* using both *APGAS* and *APGAS*$_{hyb}$ are presented in Section 3.4.4, in Listings 3.5 and 3.6, respectively.

3.3.3 Hybrid Work Stealing Algorithm and Implementation

As described in Section 2.2, *APGAS* realizes the place-internal pools with Java’s `ForkJoinPool`. Consequently, all workers of a place share a single task pool. A call to `asyncAny` initially inserts the new task into the local `ForkJoinPool`, just like a call to `async`. However, `asyncAny` tasks can later be stolen away by other workers. For that, we combine the intra-place work stealing scheme of the `ForkJoinPool` class with the lifeline-based global load balancing scheme for inter-place work stealing. The latter is performed by one dedicated management worker per place.

3.3.3.1 Management Worker

When `finishAsyncAny` is called, one management worker is started on each place. This worker runs in a dedicated Java thread, which is not part of the thread group maintained by the `ForkJoinPool`. Listing 3.4 shows the pseudocode of the management worker’s main loop, and Figure 3.2 visualizes the principal design. The management worker carries out one loop iteration per second (Line 11), except when it is inactive. This approach roughly corresponds to that of *GLB*, where a main loop iteration is carried out after processing `n` tasks.

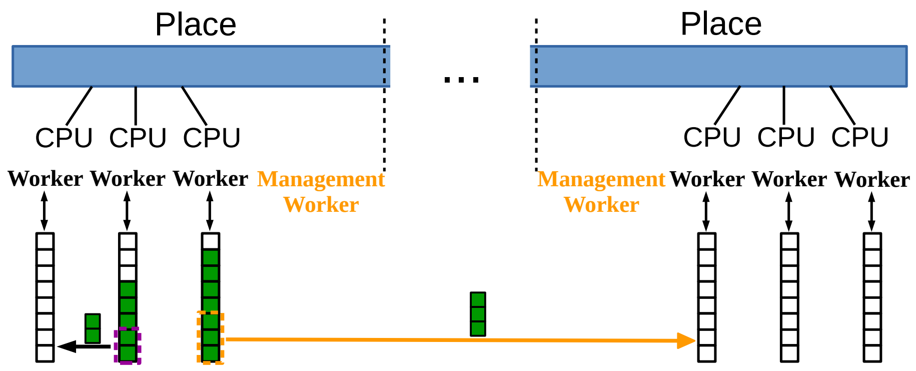


Figure 3.2: *APGAS$_{hyb}$*: Hybrid work stealing for intra- and inter-place load balancing

```
 1 while (tasks available) {
 2   send loot to recorded lifeline thieves;
 3   if (not enough local tasks left) {
 4     try to steal from up to w+z victims;
 5   }
 6   if (all local tasks have been executed &&
 7       all potential victims were contacted) {
 8     notify place 0;
 9     go inactive;
10   }
11   sleep one second;
12 }
```

Listing 3.4: *APGAS$_{hyb}$*: Management worker's main loop

If the number of unprocessed tasks in the local task pool falls below the number of local workers (Line 3), the management worker starts with work stealing (Line 4). This differs from *GLB*, where stealing is started only when a worker has no tasks left. To avoid steal operations having to wait at the victim place, they are performed using `immediateAsyncAt`.

Our hybrid work stealing technique is *coordinated*, i.e., the thief tries to pull half of the unprocessed tasks out of the victim's internal pool itself. For that, we modified Java's `ForkJoinPool` to enable pulling out multiple tasks at once. Since the `ForkJoinPool` deploys internal synchronization, tasks can be removed from the pool concurrently to the running computation of the victim. This is an alternative to the split queue from *GLB$_{split}$*.

If the victim is out of work, the `immediateAsyncAt` invokes another `immediateAsyncAt` to notify the thief about the result. If the unsuccessful steal request was a lifeline request, the thief is additionally added to a `ConcurrentLinkedQueue`. This queue needs to be thread-safe, because multiple steal requests can be received and executed concurrently. If the victim has work, the loot is sent to the thief using `staticAsyncAny`. This construct directly inserts the tasks from the loot into the thief's local `ForkJoinPool`.

When `w + z` steal attempts have returned unsuccessfully, the management worker notifies Place 0 (Line 8) and changes into inactive state (Line 9). Further details are provided in Section 3.3.3.2. An inactive management worker is reactivated when at least one `asyncAny` task is inserted into the local `ForkJoinPool`.

3.3.3.2 `FinishAsyncAny`

The original `finish` implementation tracks the termination of all spawned tasks. This induces a high overhead when the number of tasks is large. Therefore, we implemented a new `finishAsyncAny` construct, which observes only the loot. A call to `finishAsyncAny` starts a new Java thread on Place 0, which regularly checks whether 1) the internal pool contains unprocessed tasks, or 2) there are unprocessed tasks in remote pools.

The first condition is checked with standard `ForkJoinPool` methods. For checking the second condition, each place maintains an `int` array named `stealCounts` with one entry per place. It is initialized with 0. Before a victim sends loot, it increments its local `stealCounts[thief]`. When a thief receives loot, it decrements its local `stealCounts[thief]`. Just before a management worker goes inactive, it sends its `stealCounts` array to Place 0, included in Line 8 in Listing 3.4. On Place 0, all `stealCounts` arrays are added. If all entries in the sum array are 0, the second condition from above must be met, and all management workers have become inactive. Thus, the `finishAsyncAny`-thread on Place 0 terminates all management workers, as well as itself.

3.3.3.3 Task Cancellation

If tasks are submitted with `cancelableAsyncAny`, they can be canceled later by calling `cancelAllCancelableAsyncAny`. This call dequeues all unprocessed tasks and prevents new submissions. In case of an attempted new submission an exception is thrown. Since tasks are independent, a cancellation of all unprocessed tasks cannot leave the program in an inconsistent state.

To locate all cancelable tasks, each place maintains a map of type `ConcurrentHashMap<Long, Task>`. The first parameter is a system-wide unique task ID. A call of `cancelableAsyncAny` inserts the task into the local `ForkJoinPool`, and additionally stores it in the map. Tasks remove themselves from the map as their last operation. When tasks are stolen, they are removed from the victim's map and added to the thief's map.

A call to `cancelAllCancelableAsyncAny` starts an `immediateAsyncAt` on each place. It iterates through all map entries and calls the Java method `cancel()` (from class

`ForkJoinTask`) for each task. To prevent new task submissions, each place maintains a `boolean` flag, which is checked before each task submission by `cancelableAsyncAny`. The flag is set by the `immediateAsyncAt` and reset at the end of the `finishAsyncAny` block.

3.3.4 Experiments

We conducted the following three groups of experiments.

3.3.4.1 Intra- and Inter-Place Speedups on *Kassel*

In the first group of experiments, we used *Kassel* [130]. We measured intra-place speedups by starting a single place on one node and varying the number of workers from 1 to 12, since a node comprises 12 CPU cores. Then, we measured inter-place speedups by varying the number of places from 1 to 12. Here, each place was mapped to a separate node with 12 workers. Java was used in version 9.0.1.

As benchmarks, we deployed the following ones:

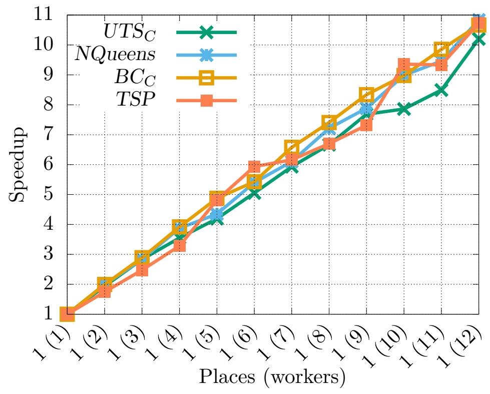
- UTS_C: geometric tree shape, branching factor $b = 4$, initial seed $s = 19$, and tree depth $d = 17$.
- *NQueens*: number of queens and chessboard size $N = 17$. New `asyncAny` tasks are spawned only until 11 queens are unplaced (`threshold`), as in the HabaneroUPC++ implementation [125].
- BC_C: initial seed $s = 2$, number of graph nodes $N = 2^{15}$ for intra-place experiments, and $N = 2^{17}$ for inter-place experiments.
- *TSP*: number of cities = 25. New `asyncAny` tasks are only spawned until 15 cities are left for the current path (`threshold`), which we observed to perform best.

Figure 3.3 depicts the measured speedups for the four benchmarks. In the intra-place case (Figure 3.3a), all benchmarks achieve good scalability. The deviation from linear speedup is up to 13.99% for *NQueens*, up to 22.78% for UTS_C (both with 11 workers), up to 11.12% for BC_C (with 12 workers), and up to 18.71% for *TSP* (with 9 workers).

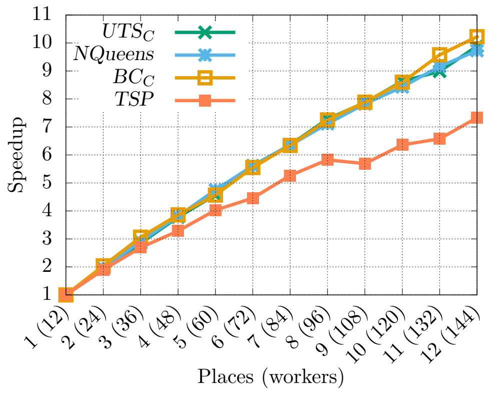
In the inter-place case (Figure 3.3b), the deviation is somewhat higher with 14.78% for BC_C and 18.95% for *NQueens* (both on 12 places). UTS_C, on the other hand, has a smaller deviation of 18.27% (on 11 places). The highest deviation was measured for *TSP* with 40.22% (on 11 places). We expect it to be due to the latency of propagating a new global optimum to an increasing number of workers. The more workers are processing tasks, the more tasks are unnecessarily computed before the new bound becomes effective in the branch-and-bound scheme.

The overall slightly lower performance of inter-place work stealing is probably caused by communication costs. Comparing the performance of the benchmarks with 144 workers to the sequential base variants, UTS_C achieves a speedup of 100, *NQueens* of 105, and *TSP* of 83.

In an additional group of experiments, we started 11 workers per place instead of 12, reserving one CPU core for the management worker. These experiments were run on up to 4 places. We still measured a nearly linear increase of speedup from 11 to 12 workers per place, from which we conclude that the management workers need almost no computational resources. Consequently, it does not pay off to reserve a core for them.



(a) Intra-place speedups



(b) Inter-place speedups

Figure 3.3: *$APGAS_{hyb}$* on *Kassel*: (a) Intra-place speedups over sequential running time, and (b) inter-place speedups over running time with one place with 12 workers

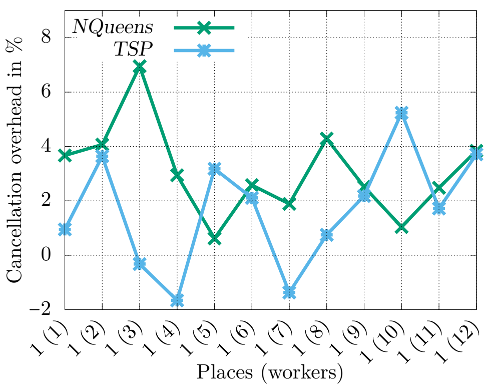
3.3.4.2 Cancellation Overhead on *Kassel*

In the second group of experiments, we evaluated the cancellation functionality on *Kassel*. For that, we implemented cancelable variants of *NQueens* and *TSP*. Here, the user specifies a limit on the number of *NQueens* placements, and a desired maximum length of the roundtrip, respectively. The benchmarks check once per second if the limit is reached by calling `reduceAsyncAny`. If so, a call to `cancelAllCancelableAsyncAny` destroys all unprocessed tasks. These variants solve search problems for a sufficiently good solution within a shorter period of time.

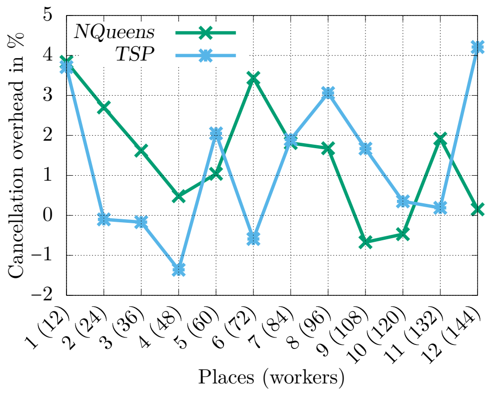
First, we tested the cancelable program variants to make sure that cancellation works properly. As expected, it takes less time to compute a given number of *NQueens* placements as compared to the total number of placements, and it takes less time to compute an approximate *TSP* solution as compared to the optimal one.

Second, we estimated the management overhead for cancellation, which includes the internal `cancelableAsyncAny` management of *APGAS*$_{hyb}$ and the periodic user calls of `reduceAsyncAny`. For these experiments, we configured the cancelable program variants so that they compute all *NQueens* placements, and the optimum *TSP* path, respectively. I.e., we let them solve the same problem as the non-cancelable variants and compared the respective running times. The overheads of the cancelable variants compared to the non-cancelable variants are depicted in Figure 3.4.

As shown in Figure 3.4a, the intra-place overhead was at most 6.95% for *NQueens*, and 5.24% for *TSP*. As shown in Figure 3.4b, the inter-place overhead was at most 3.44% for *NQueens* and 4.21% for *TSP*. Variations may be due to differences in the task processing order.



(a) Intra-place cancellation overhead



(b) Inter-place cancellation overhead

Figure 3.4: *APGAS*$_{hyb}$ on *Kassel*: (a) Intra-place and (b) inter-place performance overhead of `cancelableAsyncAny` compared to `asyncAny`

3.3.4.3 Inter-Place Speedups on *Goethe*

In a last group of experiments, we used *Goethe* [131]. Again, we measured inter-place speedups, but on a significantly larger scale than before. We used up to 128 nodes and started one place per node. Since each node comprises 40 cores, we started 40 workers on each place, resulting in a maximum of 5120 workers. Java was used in version 11.0.1.

As benchmarks, we deployed the following ones:

- *UTS*$_C$: geometric tree shape, branching factor $b = 4$, initial seed $s = 19$, and tree depth $d = 19$.

- *NQueens*: number of queens and chessboard size $N = 18$. New `asyncAny` tasks are spawned only until 10 queens are unplaced, which we observed to perform best.
- BC_C: initial seed $s = 2$, number of graph nodes $N = 2^{19}$.
- *TSP*: number of cities = 28. New `asyncAny` tasks are only spawned until 17 cities are left for the current path, which we observed to perform best.
- *Pi*: tasks per worker = 32, overall points = $2 \cdot 10^{13}$.
- *MatMul*: number of blocks $n = 512$, block size $m = 96$.

Figure 3.5 depicts the measured speedups. Comparing the running times with 5120 workers (128 nodes) to the the running times with 40 workers (1 node), UTS_C achieves a speedup of 100, *NQueens* of 120, BC_C of 128, *TSP* of 65, *Pi* of 107, and *MatMul* of 109.

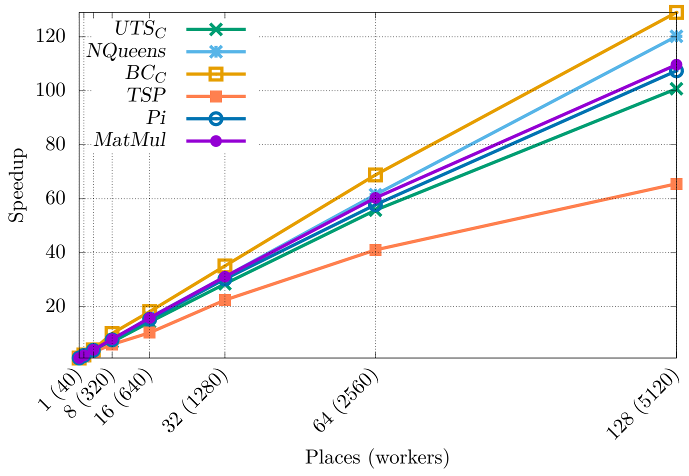


Figure 3.5: $APGAS_{hyb}$ on *Goethe*: Inter-place speedups over running time with one place with 40 workers

3.3.5 Wrap Up

In this section, we have presented a hybrid work stealing technique enabling intra- and inter-place load balancing for *APGAS*. Programmers can now submit locality-flexible tasks using the new construct `asyncAny`. These tasks are automatically scheduled over all cores and places at runtime. Moreover, programmers can cancel all unprocessed `cancelableAsyncAny` tasks of a `finishAsyncAny` block.

We have described the usage and implementation of *APGAS*$_{hyb}$. Moreover, we ported six benchmarks and observed good scalability in most cases. We have also implemented cancelable variants of two benchmarks and observed a management overhead for the cancellation below 7%.

3.4 Evaluation of APGAS for HPC and Data Analytics

3.4.1 Problem Description

The convergence between the two communities of HPC and data analytics is a hot topic in research. Data analytics programming environments have their strengths in, e.g., programmer productivity and fault tolerance, whereas HPC programming environments have their strengths primarily in performance, e.g., [136, 137]. One hindrance to common approaches across the two fields is the use of different programming languages in the two communities. Typical HPC applications use C/C++ in combination with MPI and/or OpenMP. Typical big data applications, in contrast, use JVM-based languages such as Java or Scala with libraries such as Hadoop [138] or Spark [112]. While Java is far from prominent in HPC, there are a few notable Java-based libraries such as *PCJ* [45] and *APGAS* [16].

The gap between HPC and data analytics can be bridged with interfaces such as *Spark+MPI* [136], *SWAT* [139] and *Alchemist* [140]. These interfaces come at a cost in terms of programmer productivity and computing time. Therefore, the use of a unified environment would be more appealing.

In this section, we are exploring the perspective of a common Java foundation, by comparing *APGAS*$_{hyb}$, *APGAS*, *Spark*, and *PCJ* in terms of programmer productivity and performance. Although part of the libraries can also be used with other languages, we always refer to the Java versions for a meaningful comparison. The use of Java may set *Spark* at a disadvantage, since its native language is Scala. For the comparison, we selected benchmarks from both HPC and data analytics. Moreover, we took care to implement the same algorithm in each system.

Regarding programmer productivity, we first detail our subjective impressions from developing benchmarks. The discussion is supported by codes for *Pi*. Second, programmer productivity is assessed with two objective metrics: *number of different library constructs* *used* (*NLC*) and *lines of code* (*LOC*). *LOC* metric indicates learning overhead and complexity.

This section is structured as follows. Section 3.4.2 provides background on *Spark* and *PCJ*. Then, Section 3.4.3 describes and discusses our performance measurements. Section 3.4.4 is devoted to programmer productivity, and includes both personal impressions and metrics. This section finishes with a wrap up in Section 3.4.5.

3.4.2 Background

3.4.2.1 Spark

Spark [112] is an open source, distributed, multi-threaded, in-memory, fault-tolerant library for data analytics, and widely used in this domain. The library was initially released in 2010. Meanwhile, it is maintained by the Apache Software Foundation, and is available as a repository [141]. Examples of typical *Spark* applications include iterative processing in machine learning, and interactive data analytics. *Spark* is implemented in Scala, but can also be used with Java, Python and R.

Like Hadoop [138], *Spark* implements the MapReduce model [59]. It addresses Hadoop's I/O performance bottleneck by maintaining data in memory rather than on disk. This yields a speedup by a factor of up to 100 for iterative algorithms [112]. If MapReduce is not well suited for a particular problem, *Spark* can cause a significant overhead [140].

Spark's primary data abstraction is called *Resilient Distributed Dataset* (`RDD`) [142]. An `RDD` is a resilient, immutable, and distributed data structure. It contains a set of elements and provides operations for

- producing new `RDD`s (called *transformations*), and
- computing return values (called *actions*).

Common examples of transformations and actions are the well-known operations map and reduce, respectively.

The creation of `RDD`s is lazy, i.e., transformations are only triggered when an action is called. This prevents unnecessary memory usage and minimizes computations. Fault tolerance for `RDD`s is achieved by storing the operation sequence and recomputing lost data after failures. This technique, called *ancestry tracking*, does not require checkpoints.

A *Spark* program starts by creating a `SparkContext` object, to which a `SparkConf` object is passed. The `SparkContext` object lives in an extra JVM, which is called a *driver*. The `SparkConf` object contains information about the execution, e.g., on how many JVMs the program should run, how many JVMs are mapped to each node, the number of worker threads per JVM, and an upper bound on memory usage. JVMs are called *executors*. Each executor has the same number of workers.

Since the driver executes the `main` method, transformations and actions are called within the driver's JVM. When an action is called, the driver splits all computations of the required operations into tasks and distributes them over executors. The result of the action is returned to the driver.

An `RDD` can be created, e.g., by passing a local data collection to method `parallelize()` of the `SparkContext` object, or by passing URIs of text files to function `textFile()`. Further transformations include:

- `map()`: Applies a passed function to each data element and returns the resulting `RDD`.
- `flatMap()`: Similar to `map()`, but the passed function may produce for each data element multiple new elements instead of a single one.
- `filter()`: Returns a new `RDD` containing the data elements that match a passed `boolean` function.
- `reduceByKey()`: Returns a new `RDD`, in which each executor's subset of the data is reduced to a single value.

Actions include:

- `count()`: Returns the number of data elements in an `RDD`.
- `collect()`: Returns an array with all data elements of an `RDD`.
- `reduce()`: Pulls all data elements to the driver, aggregates them with a passed function, and returns a single value.

Spark offers several deployment options. For instance, standalone mode is the simplest option on a private cluster, and Mesos [143] is a dedicated cluster manager.

3.4.2.2 *PCJ*

PCJ [45] implements the PGAS model in Java and is open source [144]. *PCJ* is targeted at large-scale HPC applications but was also observed to be suitable for data analytics applications [145, 146, 147]. *PCJ* won the HPC Challenge Class 2 Best Productivity Award on Supercomputing in 2014 [114] and achieves a better performance than MPI with

Java bindings [146]. In some situations, however, the performance of *PCJ* is up to three times below that of MPI with C bindings [146].

PCJ is shipped as a single jar file without dependencies on other libraries. This is an advantage over *Spark*, which involves many dependencies, and over *APGAS*, which involves Hazelcast. Currently, *PCJ* offers no fault tolerance mechanism, but it is in development [148].

Execution units in *PCJ* are called *worker*s. Technically, a worker is realized by a Java thread that maintains its own local memory. Each place runs in a separate JVM. Different places can use a different number of workers. The numbers of places and workers are configurable, see below.

PCJ adopts the SPMD execution style, i.e., all workers are invoked at program startup and execute the same `main` method, which is specified in a `Startpoint` interface. Variables are private to each worker, such that different workers can follow different code paths. *PCJ* provides several methods to exchange data between workers in a synchronous or asynchronous way.

Variables can be declared as shared to permit access by other workers. For the declaration, the variables must simultaneously be fields of an `enum`, and of a class that implements `StartPoint`. Moreover, the `enum` must be annotated with `@Storage(`*`Class`*`)`, and the class should be annotated with `@RegisterStorage(`*`Enum`*`)`.

Details of the communication between workers are hidden from *PCJ* programmers. Instead, the *PCJ* runtime automatically determines whether a communication is place-internal or global, and selects an appropriate communication mechanism. For place-internal communication, it deploys Java's concurrency constructs, and for global communication it deploys network sockets. Global communication is further optimized by arranging places in a graph.

The *PCJ* library includes a launcher, which starts applications on multiple cluster nodes. It takes as input a text file that contains a list of nodes that may contain duplicates. The launcher starts one place for each of the different entries in the list, and one worker for each individual entry. *PCJ* includes the following methods and classes:

- `deploy()`: Deploys a *PCJ* application across a cluster. The method passes a text file to the launcher, as described before.
- `myID()`: Returns the id of the calling worker. The ids are consecutive numbers starting with 0.
- `threadCount()`: Returns the total number of workers system-wide.
- `getNodeCount()`: Returns the number of places.

- `getNodeId()`: Returns the id of the calling place.
- `barrier()`: Synchronizes all workers. When a worker reaches this call, it stops execution and only resumes when all workers have reached the same line in their respective codes. All workers must execute this line. A variant of the method supports pairwise synchronization of two workers.
- `get()`: Synchronously reads the value of a shared variable from the local memory of a particular worker.
- `put()`: Synchronously stores a value into a shared variable in the local memory of a particular worker.
- `asyncGet()`: Asynchronous variant of `get()`. Returns a `PcjFuture` object.
- `asyncPut()`: Asynchronous variant of `put()`. Returns a `PcjFuture` object.
- `PcjFuture`: Provides a `get()` method, which waits for the completion of the underlying operation, and returns the result (if any).
- `broadcast()`: Sends a value to all workers and writes it into their respective instances of a shared variable with the same name, which is passed as a parameter.
- `waitFor()`: Waits until the value of a given shared variable has changed.

3.4.3 Experiments

Experiments were conducted on *Kassel* [130]. For *Spark*, we used version 2.3.0. For *PCJ*, we used version 5.0.6 as of May 29, 2018, from the official repository [144]. For *PCJ* and *APGAS*, we deployed Java release 10.0.1. Since *Spark* was not compatible with this Java version at that time, we used Java version 8.0 for all *Spark* benchmarks. We did not specifically configure the JVMs, but used the default settings of the respective Java versions.

In the first group of experiments, we measured intra-node performance. We utilized one JVM, and varied the number of workers from 1 to 12, since a node comprises 12 CPU cores. For *Spark*, we also started the driver on the same node. Then, we measured inter-node performance, for which we varied the number of nodes from 1 to 12. Here, one JVM with 12 workers was run on each node. Therefore, we started up to 144 workers. For *Spark*, the driver was placed on one of the 12 nodes.

For each benchmark, we used strong scaling (fixed global problem size). To jump ahead briefly, $APGAS_{hyb}$ performs best in most cases. Therefore, to illustrate performance differences clearly, Figure 3.6 depicts the *overhead* of the other libraries compared to $APGAS_{hyb}$. The overhead is specified as a percentage, and is calculated with the formula $\texttt{time}_x$ / $\texttt{time}_{APGAS_{hyb}} - 1$, where $x \in \{Spark,\ PCJ,\ APGAS\}$.

3.4.3.1 Pi

As parameters for *Pi*, we used $n = 2^{40}$, and tasks per worker = 64, which we experimentally determined as the best value for all systems.

Figure 3.6a depicts the overheads over $APGAS_{hyb}$ as percentages. Overall, $APGAS_{hyb}$ has the best performance, and a speedup of 127.30 with 144 workers.

Spark has a large overhead over $APGAS_{hyb}$ for low worker counts, e.g., 57.57% with two workers. Between 10 and 144 workers, the overhead of *Spark* ranges between 2.63% and 9.60%.

PCJ has an overhead over $APGAS_{hyb}$ of at most 10.84% with 72 workers. With 5 workers, *PCJ* performs 0.42% better than $APGAS_{hyb}$. Between 10 and 144 workers, the overhead of *PCJ* ranges between 6.99% and 10.84%.

The performance of *APGAS* differs only slightly from that of $APGAS_{hyb}$. Sometimes, *APGAS* is better, by a maximum of 4.82% with 4 workers. With more than 60 workers, *APGAS* is consistently slower than $APGAS_{hyb}$, with an overhead of at most 6.27% with 144 workers.

3.4.3.2 UTS

Since UTS_E generates an irregular workload at runtime, and *APGAS*, *Spark*, and *PCJ* do not support automatic system-wide load balancing, we initially calculate the tree sequentially up to a certain depth, called *seqTreeDepth*. The resulting tree nodes are then split into $numWorker \cdot tasksPerWorker$ tasks, which are distributed evenly to all workers. We set both $tasksPerWorker = 64$ and $seqTreeDepth = 6$. These values were determined experimentally, and performed best for all systems.

We configured UTS_E with geometric tree shape, branching factor $b = 4$, initial seed $s = 19$, and tree depth $d = 16$.

Figure 3.6b depicts the overheads over $APGAS_{hyb}$ as percentages. Overall, $APGAS_{hyb}$ has the best performance, and a speedup of 89.97 with 144 workers.

Spark has its lowest overhead of 10.84% over $APGAS_{hyb}$ with one worker. For multiple JVMs, the overhead varies between 38.32% (122 workers) and 57.10% (48 workers).

PCJ performs better than *APGAS*$_{hyb}$ by 0.49% with one worker. Otherwise, *PCJ* performs worse. For multiple JVMs, the overhead varies between 40.60% (144 workers) and 89.65% (24 workers).

APGAS's overhead over *APGAS*$_{hyb}$ is rather low inside a place. On multiple JVMs, it varies between 20.62% (144 workers) and 68.41% (24 workers).

3.4.3.3 WordCount

Like [145], we selected two novels as input for *WordCount*: Lev Tolstoy's *War and Peace* [149] (written in English, 3.3 MB), and Georges des Scudéry's *Artamène ou le Grand Cyrus* [150] (written in French, 10 MB). Both are encoded in UTF-8. To achieve a larger amount of data, each file was read 32,768 times, resulting in 105 GB and 320 GB of input data, respectively. The count of 32,768 was evenly distributed over all workers, such that each worker read and processed 32,768/$totalWorkers$ files successively. In *APGAS*$_{hyb}$, 32,768 locality-flexible tasks were spawned.

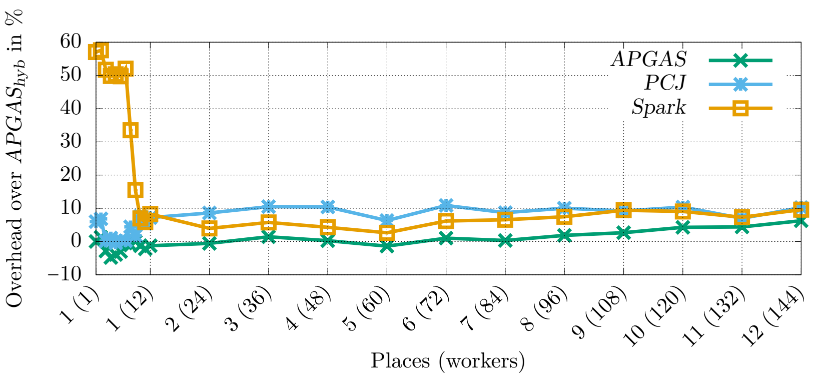
Figures 3.6c and 3.6d depict the overheads over *APGAS*$_{hyb}$ as percentages. When using *War and Peace*, *APGAS*$_{hyb}$ always has the best performance, and a speedup of 63.80 with 144 workers. When using *Artamène ou le Grand Cyrus*, *APGAS*$_{hyb}$ again has the best performance, and a speedup of 96.78 on 144 workers. The only exception occurs for one worker, where *PCJ* is faster by 2.28%.

For the first novel, *Spark* has an overhead over *APGAS*$_{hyb}$ of at most 133.54% with 9 workers. With an increasing number of workers the overhead decreases, but with 144 workers it is still at a high value of 93.11%. Results are similar for the second novel, where *Spark*'s overhead over *APGAS*$_{hyb}$ increases with the number of workers from 53.43% with 1 worker to 131.58% with 144 workers.

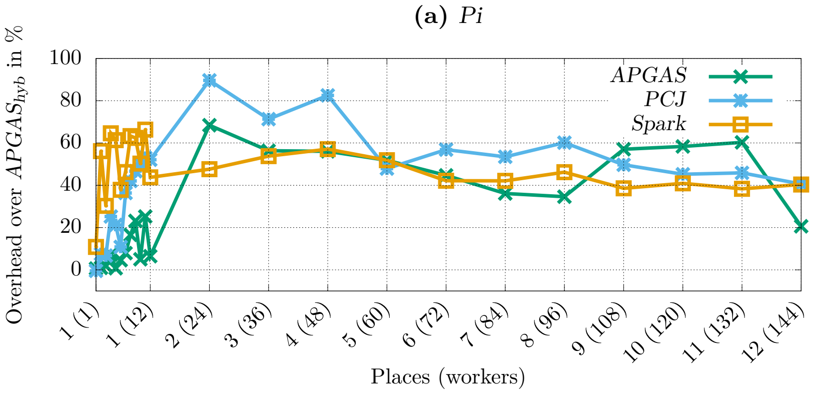
The overheads of *PCJ* and *APGAS* over *APGAS*$_{hyb}$ tend to increase with the number of workers. The overhead of *PCJ* varies between 0.65% (6 workers) and 20.19% (144 workers) for the first novel, and is up to 24.94% (144 workers) for the second novel. The overhead of *APGAS* varies between 0.08% (24 workers) and 13.07% (144 workers) for the first novel, and is up to 10.90% (144 workers) for the second novel.

3.4.3.4 Discussion

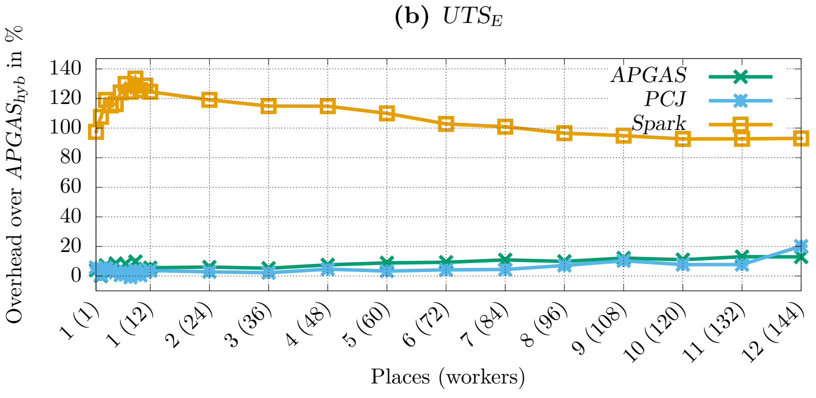
Overall, *APGAS*$_{hyb}$ outperforms the other systems, probably because it is the only system that provides load balancing at both the intra- and inter-node levels. As expected, the advantage is particularly clear for the dynamic workloads of *UTS*$_{E}$. For static workloads, like those of *Pi* and *WordCount*, the advantage is smaller, but still noticeable.



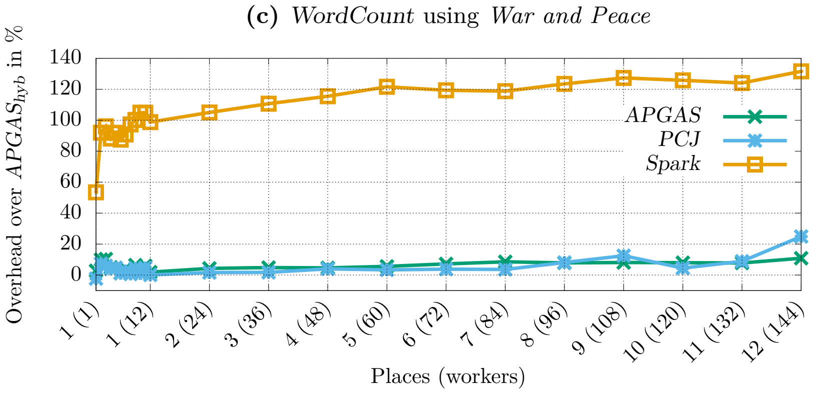
(a) *Pi*



(b) *UTS_E*



(c) *WordCount* using *War and Peace*



(d) *WordCount* using *Artamène ou le Grand Cyrus*

Figure 3.6: Running time overheads of *APGAS*, *PCJ*, and *Spark* over *$APGAS_{hyb}$* on *Kassel*

The *PCJ* programs can be extended manually by dynamic load balancing. Since, in our experiments, *PCJ* and *APGAS* achieved a similar performance, we expect that such an extension may at best bring the *PCJ* performance close to $APGAS_{hyb}$, but at the price of an even lower programmer productivity than described in Section 3.4.4.

Spark does not provide appropriate constructs for manually implementing dynamic load balancing. Overall, the *Spark* programs have the lowest performance. Surprisingly, *WordCount*, which is a typical data analytics benchmark, needed approximately twice the time of its $APGAS_{hyb}$ counterpart. We do not know the reason for this result. Possible explanations include the use of an older Java version, deployment of Java instead of Scala, or a rather low machine size.

3.4.4 Programmer Productivity

The term programmer productivity denotes human efficiency in writing and maintaining applications. Of course, productivity is somewhat subjective, since it depends on the programmer to some degree. To be as fair as possible, all benchmarks were developed by the same person, namely the second author of the corresponding publication [P8], who at that time had no previous experience with any of the systems. In the following, we describe and discuss his impressions, referring to code examples. Moreover, the discussion includes two objective metrics: *number of different library constructs used* (*NLC*), and *lines of code* (*LOC*). The *NLC* metric reflects learning overhead and complexity. For *LOC*, we only count lines containing code.

Listings 3.5–3.8 depict the source codes for *Pi*. The codes are almost complete, except that a few code snippets have been shortened. In particular, the listings for *Spark* and both *APGAS* variants only show the contents of the `main` method, because the associated classes comprise standard elements only. Since *PCJ* requires class annotations to declare variables as shared, the class is included for *PCJ* in Listing 3.7.

Table 3.1 reports the *NLC* values of our codes. As constructs, we count methods, classes etc., as provided by the libraries. Counted constructs are colored in green in Listings 3.5–3.8. As shown in Listing 3.7, the *PCJ* constructs refer to worker control (e.g., Line 29), worker communication (e.g., Line 33), shared variable declarations (e.g., Line 11), and system control (e.g., Line 21). *APGAS* constructs, as depicted in Listing 3.5, chiefly refer to task spawning (e.g., Line 13) and distributed data structures (e.g., Line 7). $APGAS_{hyb}$ constructs, as depicted in Listing 3.6, refer to task spawning (e.g., Line 8) and result reduction (e.g., Line 18). Finally, as depicted in Listing 3.8, *Spark* constructs

	APGAS	***APGAS$_{hyb}$***	***Spark***	***PCJ***
Pi	6	7	7	12
UTS$_E$	6	5	6	15
WordCount	7	6	10	12

Table 3.1: Number of different library constructs used (*NLC*)

	APGAS	***APGAS$_{hyb}$***	***Spark***	***PCJ***
Pi	36	31	29	67
UTS$_E$	64	38	28	78
WordCount	75	74	46	76

Table 3.2: Lines of code (*LOC*)

include transformations (e.g., Line 16), actions (e.g., Line 24), and system control (e.g., Line 8).

The *APGAS* variants have the lowest *NLC* value, with a minor advantage for *APGAS$_{hyb}$*. *Spark* always ranks third, requiring up to four constructs more than *APGAS$_{hyb}$*. *PCJ* always has the highest *NLC* value, and needs about twice as many constructs as *APGAS$_{hyb}$*.

One reason for this outcome can be seen in *Spark*'s and *PCJ*'s use of constructs to start the library runtime, see Lines 8 and 9 in Listing 3.8 and Lines 2, 21, 22, 25 and 26 in Listing 3.7, respectively. In contrast, an *APGAS* program is started automatically when calling the first construct.

Table 3.2 reports the *LOC* metric for all codes. For a fair comparison, codes were styled in the same way, according to the Google Java Style Guide [151].

As the table shows, *Spark* always has the lowest *LOC* value, while *APGAS$_{hyb}$* ranks second, *APGAS* third, and *PCJ* fourth. *PCJ* and both *APGAS* variants need more lines than *Spark*, because storing and reducing the result has to be implemented explicitly. However, since *APGAS$_{hyb}$* offers some support for this, it ranks second.

For example, *Spark*'s *Pi* code in Listing 3.8 only needs one call of `reduce()` in Line 24 to accumulate all distributed results. In contrast, the *PCJ* code in Listing 3.7 reduces the results manually (Lines 48–59, excluding the output in Line 58), and defines the partial results as shared variables (Lines 1, 2, 7, 11, 12, 13 and 47). In the *APGAS* code in Listing 3.5, each task adds its result to the overall result on Place 0 (Lines 7, 21–23 and 29). In the *APGAS$_{hyb}$* code in Listing 3.6, each task merges its result into the local worker result (Lines 9 and 15). After all tasks have been processed, the overall result is computed by reduction on Place 0 (Line 18).

When developing the benchmarks, *APGAS$_{hyb}$* was felt to be most productive. Its use was intuitive, simple, and efficient. In particular, the locality-flexible tasks simplified the implementation, because there was no need to think about load balancing. Moreover, *APGAS$_{hyb}$* provides several handy constructs for storing and reducing results. Our impressions were confirmed by both metrics. Personally, our test person felt that the *NLC* metric better reflects his subjective impressions of programming difficulty and time consumption. In the subjective comparison, he ranked *APGAS* second, because task distribution and result reduction had to be implemented by hand. Aside from that, *APGAS* was just as easy to understand and use as *APGAS$_{hyb}$*.

The described differences between *APGAS* and *APGAS$_{hyb}$* become clear in Listings 3.5 and 3.6. In the *APGAS$_{hyb}$* code, all tasks are spawned by a single construct in Line 8. Note that the number of tasks is defined in the same call, but in Line 16. In contrast, the *APGAS* code manually distributes the tasks evenly over all places, see Lines 11–13.

Spark required more time than *APGAS* to get familiar with, and the algorithms had to be adapted to the MapReduce model. Still, the resulting source code is short and easy-to-understand. The code in Listing 3.8 creates a task list in Lines 11 and 12, and distributes it evenly in Line 14. However, the list itself is not really needed, but only used to distribute consecutive numbers in Line 14. This feels cumbersome but is the easiest and officially recommended way.

We needed a little more training time for *PCJ* than for *Spark*. This was related to the fact that, even for simple problems, more constructs are needed. Both the *PCJ* and *APGAS* codes explicitly take care of task distribution. However, more effort was required for that in *PCJ* than in *APGAS*, compare Listing 3.7 Lines 28–34 and Listing 3.5 Lines 10–13, respectively. Moreover, in *PCJ*, replicating values such as `tasksPerPlace` requires much programming effort, see Listing 3.7 Lines 1, 2, 8, 11, 12, 15, 31, 32 and 33. In contrast, the other systems automatically copy final variables into lambdas for remote reading, see e.g., Listing 3.8 Lines 6 and 18. Even after some time, our test person found the syntax and use of shared variables in *PCJ* difficult and error prone.

Spark and both *APGAS* variants provide automatic intra-node work balancing, but *PCJ* does not. *PCJ* programmers may manually implement it.

For testing our programs, we first installed all libraries on local workstations. These installations did not cause any trouble, although the *Spark* installation was by far the most complicated. When deploying the libraries on a typical HPC cluster with Slurm as workload manager, our experiences varied. Writing a submit-script for *Spark* programs, which starts Spark in standalone mode and sets all environment variables correctly, was quite time consuming and challenging. In contrast, both *PCJ* and *APGAS* offer launchers, which we could use without much effort.

```
long points = 1L << Integer.parseInt(args[0]);
int tasksPerWorker = Integer.parseInt(args[1]);
int allWorkers = localWorkers() * places().size();
int numTasks = allWorkers * tasksPerWorker;
long pointsPerTask = points / numTasks;

GlobalRef<AtomicLong> result = new GR<>(new AL());

finish(() -> {
 for (Place p : places()) {
  for (int j = 0; j < workerPerPlace; ++j) {
   for (int t = 0; t < tasksPerWorker; t++) {
    asyncAt(p, () -> {
     long tmpCount = 0;
     for (long i = 0; i < pointsPerTask; ++i) {
      double x = 2 * randomDouble() - 1.0;
      double y = 2 * randomDouble() - 1.0;
      tmpCount += (x * x + y * y <= 1) ? 1 : 0;
     }
     long transferCount = tmpCount;
     asyncAt(result.home(), () -> {
      result.get().addAndGet(transferCount);
     });
    });
   }
  }
 }
});
long count = result.get().get();
println("Pi is roughly " + 4.0 * count / points);
```

Listing 3.5: *APGAS*: Code for *Pi*

```
long points = 1L << Integer.parseInt(args[0]);
int tasksPerWorker = Integer.parseInt(args[1]);
int allWorkers = localWorkers() * places().size();
int numTasks = allWorkers * tasksPerWorker;
long pointsPerTask = points / numTasks;

finishAsyncAny(() -> {
 staticAsyncAny(() -> {
  long tmpCount = 0;
  for (long j = 0; j < pointsPerTask; ++j) {
   double x = 2 * randomDouble() - 1.0;
   double y = 2 * randomDouble() - 1.0;
   tmpCount += (x * x + y * y <= 1) ? 1 : 0;
  }
  mergeAsyncAny(tmpCount, PLUSLONG);
 }, numTasks);
});
long count = reduceAsyncAnyLong(PLUSLONG);
println("Pi is roughly " + 4.0 * count / points);
```

Listing 3.6: *APGAS$_{hyb}$*: Code for *Pi*

```
@RegisterStorage(PCJPi.Shared.class)
public class PCJPi implements StartPoint {

 public static int n = 0;
 public static long tasksPerWorker = 0;
 public long points = 0;
 public long c = 0;
 public long tasksPerPlace = 0;
 public static AtomicLong remainingTasks;

 @Storage(PCJPi.class)
 enum Shared {
  c,
  points,
  tasksPerPlace
 }

 public static void main(String[] args) {
  n = Integer.parseInt(args[1]);
  tasksPerWorker = Integer.parseInt(args[2]);
  NodesDescription n = new NodesDescription(args[0]);
  PCJ.deploy(PCJPi.class, n);
 }

 @Override
 public void main() {
  points = 1L << n;
  int nodes = PCJ.getNodeCount();
  int worker = PCJ.threadCount();
  int workerPerPlace = nodes / worker;
  tasksPerPlace = workerPerPlace * tasksPerWorker;
  PCJ.barrier();
  long myTPP = PCJ.get(0, Shared.tasksPerPlace);
  remainingTasks = new AtomicLong(myTPP);
  points = PCJ.get(0, Shared.points);
  long nAll = points;
  long pointsPerTask = nAll / (myTPP * nodes);
  long tmpCount = 0;
  PCJ.barrier();
  while (remainingTasks.decrementAndGet() >= 0) {
   for (long i = 0; i < pointsPerTask; i++) {
    double x = 2 * randomDouble() - 1.0;
    double y = 2 * randomDouble() - 1.0;
    tmpCount += (x * x + y * y <= 1) ? 1 : 0;
   }
  }
  c = tmpCount;
  PCJ.barrier();
  if (PCJ.myId() == 0) {
   PcjFuture<Long> cL[] = new PcjFuture[worker];
   long c0 = c;
   for (int p = 1; p < worker; p++) {
      cL[p] = PCJ.asyncGet(p, Shared.c);
     }
     for (int p = 1; p < worker; p++) {
      c0 = c0 + cL[p].get();
     }
     println("Pi is roughly " + 4.0 * c0 / nAll);
    }
  }
}
```

Listing 3.7: *PCJ*: Code for *Pi*

```
long points = 1L << Integer.parseInt(args[0]);
int totalWorker = Integer.parseInt(args[1]);
int tasksPerWorker = Integer.parseInt(args[2]);
int totalTasks = totalWorker * tasksPerWorker;
long points = 1L << n;
long pointsPerTask = points / numTasks;

SparkConf sparkConf = new SC().setAppName("Pi");
JavaSparkContext jsc = new JSC(sparkConf);

List<Int> list = new ArrayList<>(totalTasks);
for (int i = 0; i < totalTasks; i++) list.add(i);

JavaRDD<Int> rdd = jsc.parallelize(list, totalTasks);

long count = rdd.map(integer -> {
 long tmpCount = 0;
 for (long i = 0; i < pointsPerTask; ++i) {
  double x = 2 * randomDouble() - 1.0;
  double y = 2 * randomDouble() - 1.0;
  tmpCount += (x * x + y * y <= 1) ? 1 : 0;
 }
 return tmpCount;
}).reduce((int1, int2) -> int1 + int2);
println("Pi is roughly " + 4.0 * count / points);
```

Listing 3.8: *Spark*: Code for *Pi*

3.4.5 Wrap Up

In this section, we have compared the data analytics library *Spark* and the HPC libraries *PCJ* and *APGAS*. For *APGAS*, we included both the original version and our extended *APGAS*$_{hyb}$ variant. The comparison was based on Java implementations of three benchmarks, which were partly taken from HPC and the data analytics domain, respectively. All implementations were conducted by the same test person, who had no previous experience with any of the programming environments. Furthermore, we took care to implement the same algorithms.

On one hand, we evaluated programmer productivity, based on personal impressions and objective metrics. *APGAS*$_{hyb}$ turned out best, closely followed by *APGAS* and *Spark*. *APGAS*$_{hyb}$ was most intuitive to use, required the lowest number of different library constructs, and its code was by only a few lines longer than that of the *Spark* variant.

On the other hand, we carried out performance measurements with up to 144 workers. They showed *APGAS*$_{hyb}$ as a clear winner. All *APGAS* programs scaled well. With 144 workers, their execution time was up to 28.88% less than that of the *PCJ* programs, and up to 56.81% less than that of the *Spark* programs.

Overall, our results suggest that *APGAS*$_{hyb}$ may be a strong candidate for programming both HPC and data analytics applications with the same system.

3.5 Related Work

Task pools have received continuous attention in the literature, e.g., [152, 153, 154, 155]. Different variants of task pools target shared- and/or distributed-memory architectures, use a central or distributed data structure, and are deployed in runtime systems or at user-level. Task exchange is accomplished with sender-initiated and receiver-initiated approaches, also denoted as work sharing and work stealing, respectively. Work stealing has become popular with Cilk [68] but work sharing can have competitive performance [156].

Our cooperative scheme in Section 3.2 resembles the receiver-initiated algorithm from Acar *et al.* [69]. Like our scheme, their algorithm registers thieves in parallel to task processing. However, their algorithm refers to a shared-memory setting, and uses a compare-and-swap operation for the registration. Unlike in our scheme, only one thief can register at a time. The authors compare their algorithm to a coordinated scheme based on concurrent deques [157]. In line with our results, the cooperative and coordinated schemes have similar performance. For different applications, their running time varies by up to 18% in both directions.

The split queue data structure has been introduced by Dinan *et al.* [158]. We referred to the slightly modified variant from [113], which adopts a lockless `release()` operation and deploys the steal-half strategy. Recently, a more advanced variant has been proposed that uses atomic operations to enhance the performance [159].

For *APGAS*$_{hyb}$, we adopted the concept of locality-flexible `asyncAny` tasks from HabaneroUPC++ [134]. However, our implementation of hybrid work stealing is fundamentally different. The HabaneroUPC++ scheme does not limit the number of random remote victims and evaluates them with the help of RDMA. In contrast, we deploy the *GLB* lifeline scheme [17], and thus steal from up to `w + z` remote victims, which are selected without RDMA. Like HabaneroUPC++, we utilize a dedicated management worker thread for the inter-place work stealing. However, HabaneroUPC++ runs the management worker on a dedicated CPU core that does not participate in the actual computation, whereas we use as many computation workers as cores. Finally, HabaneroUPC++ binds to C++, and $APGAS_{hyb}$ to Java.

Yamashita and Kamada [70] presented multistage execution and multithreading for *GLB* in X10. Each worker maintains an own queue, and each place holds two shared queues for a combination of inter-place work stealing and intra-place work sharing. However, the overall scheme is quite complicated, and the implementation has problems with network message scheduling. Recently, an *APGAS* variant has been proposed that adds tuning mechanisms to dynamically adjust the task granularity to improve the performance [19]. We will build on this *APGAS* variant, without the tuning mechanism, in Chapter 5.

Paudel *et al.* investigate hybrid task placement in X10 with work stealing and work dealing, respectively [160, 161]. Both papers deploy a dedicated thread for inter-place communication. Programmers use annotations to distinguish tasks into location-sensitive and location-flexible ones. Hybrid work stealing for nested fork-join programs is handled in [21]. Recently, work sharing and work stealing have been compared for nested fork-join programs, with no significant difference observed in experiments [162].

A classification and evaluation of task cancellation techniques is given in [163]. Most task-based programming environments do not support task cancellation [55]. An exception is OpenMP [37], where users can cancel parallel regions, sections, loops, and taskgroups. Moreover, HPX [53] supports canceling individual tasks that have not yet been started or are currently blocked. To the best of our knowledge, our work is the first that provides a cancellation feature for locality-flexible tasks.

Filling the gap between HPC and big data systems has received much attention in recent years. For example, Asaadi *et al.* [164] give a survey of MPI, OpenMP, OpenSHMEM, *Spark* and Hadoop. They discuss different system characteristics and performance. These authors conclude that a new programming model should be developed that combines the best of both worlds.

Several researchers have combined *Spark* with typical HPC systems. For example, Spark+MPI [136] exchanges serialized data between Spark and an existing MPI library via a shared-memory file system. Since a data exchange requires several seconds, the system is only useful for long-running Spark computations.

In contrast, Alchemist [140] uses sockets for the data transfer between Spark and MPI. All data must be stored twice: in a *Spark* `RDD`, and in a distributed matrix on the MPI side. Still, using Alchemist in *Spark* programs significantly improves the performance.

SWAT [139] enables for *Spark* the usage of GPUs. Users can still write their *Spark* programs in Java, but SWAT generates OpenCL code from the JVM bytecode at runtime. The generated code is then executed on GPUs. The authors report a speedup by a factor of 3.24 on six machines.

Previous work by Bała *et al.* [145, 146, 147] compared *PCJ* to Apache Hadoop. These authors argue that *PCJ* is easier to use than Hadoop, and *PCJ* programs are 5 to 500 times faster. Moreover, *PCJ* was observed to perform better than MPI with Java bindings, but up to three times worse than MPI with C bindings.

Suter *et al.* compared the Scala version of *APGAS* to Akka [165], which is an actor-based concurrency library [116]. These authors conclude that *APGAS* and Akka are similar in both program complexity and performance.

HabaneroUPC++ [125] is another asynchronous library implementation of the PGAS model. It allows direct global memory access, but has no support for fault tolerance and elasticity. A more detailed comparison of *APGAS* and HabaneroUPC++ was conducted by Scherbaum [166].

3.6 Conclusions

In this chapter, we have contributed to the open research questions regarding load balancing. To help identify which load balancing technique performs best, we have compared cooperative and coordinated work stealing. The performance differences between them were minor. In addition, we have developed a novel hybrid work stealing technique, for which we introduced novel tasking constructs such as `asyncAny`. This way, we have contributed to the open research question for generic, flexible, feature-rich, and user-friendly tasking constructs.

Moreover, we have contributed to the open research question whether AMT can be a one-fits-all solution by comparing *APGAS* with *Spark* and *PCJ*. First, we have evaluated productivity, based on personal impressions and objective metrics. Second, we have carried out performance measurements. Both categories have shown $APGAS_{hyb}$ as the winner. Thus, we conclude that $APGAS_{hyb}$ might be a good candidate for programming both HPC and data analytics applications with the same programming environment.

Chapter 4

Fault Tolerance

Contents

4.1 Introduction

In this chapter, we present our contributions to the open research questions regarding fault tolerance. We start by defining the failure types that can be handled by our fault tolerance techniques in Section 4.2. Then, we describe our novel ***T**ask-level **C**heckpointing* technique *TC* in Section 4.3. We state general requirements of *TC* on work stealing in Section 4.3.1. The description of *TC* is first formulated in a general way in Section 4.3.2 so that *TC* can be applied to a spectrum of task models, as asked for by one of the open research questions. In addition, we describe our concrete adaptation to lifeline-based work stealing in Section 4.3.3, and our implementation TC_{GLB} in Section 4.3.4. Furthermore, we discuss the correctness of TC_{GLB} informally in Section 4.3.5, and compare TC_{GLB} with the related predecessor $X10\text{-}FT_{GLB}$ in Section 4.3.6.

Then, in Section 4.4, we present the three related fault tolerance techniques *IncTC*, *LogTC*, and *SST*. *IncTC* resembles *TC* but reduces the checkpointing overhead by writing checkpoints incrementally and for “stable” tasks only. *LogTC* combines SST_{NFJ}, the supervision and steal tracking approach for nested fork-join programs [21] mentioned in Section 1.3.3, with *TC*'s checkpointing scheme to avoid updating the checkpoints in the event of steals. *SST* transfers SST_{NFJ} from the context of nested fork-join programs to our context of dynamic independent tasks. Consequently, *SST* does not write checkpoints at all. With *LogTC* and *SST*, we contribute to one of the open research questions by showing that task-level resilience techniques need not be specific to a particular task model.

Another open research question asked whether task-level fault tolerance can be provided in a way that does not require additional programming effort. We show that using our extended *GLB* variants requires only negligible additional programming effort.

Afterwards, in Section 4.5, we experimentally evaluate the performance of our techniques. First, we compare *TC* with the checkpoint/restart library DMTCP [73]. Results clearly show that task-level techniques pay off, as *TC* has significantly lower failure-free running time overheads and recovery costs than DMTCP. To explore the open research question of which technique works best at task-level, we compare our techniques with each other. Experimental results show only small performance differences in failure-free runs of *TC*, *IncTC*, *LogTC*, and *SST* for small task pools, whereas for large pools, *IncTC* and *LogTC* outperform *TC* and *SST*.

To address the open research question of how our techniques may impact the throughput of supercomputers, we derive formulas that predict running times including failure handling of applications protected with *TC* and *SST*. The formulas, which are derived in Section 4.6, depend on the number of workers, steal rate, and MTBF. Based on the formulas, we predict running times in larger-scale settings than in our experiments and determine conditions under which *TC*/*SST* are superior in single application runs in Section 4.7. Moreover, we perform simulations to determine overall completion times of job sets, in which either all jobs are protected with *TC* or *SST*, or none of the jobs use any resilience technique (unprotected jobs). Results show that the completion time can be reduced by up to 97% if all jobs are protected with *TC* or *SST* and each worker fails on average once a year. Differences are rather small. We find that *SST* performs slightly better in all currently realistic scenarios, but *TC* takes over in systems with an order of millions of processes.

Finally, we conclude this chapter with related work and conclusions in Sections 4.8 and 4.9, respectively.

This chapter was adapted from publications as follows: Section 4.2 from [P12, P15], Section 4.3 from [P2, P3, P4], Section 4.4 from [P7, P9, P12, P15], Section 4.5 from [P4, P9, P10, P15], Section 4.6 from [P15], and Section 4.7 from [P15].

4.2 Failure Model

Our fault tolerance techniques handle *permanent* (also called *fail-stop*) failures of workers and assume reliable network communication. Different workers that run on the same node are allowed to fail independently, although in practice they will usually go lost together. Any number of workers may fail at any time, including unsuitably correlated times such as during recovery. However, we do not permit failure of the resilient store, in which checkpoints are stored (for *TC*, *IncTC*, and *LogTC*), and failure of worker 0 (for *SST*). These cases lead to program abort if no further precautions are taken. Failures never compromise the correctness of a computed result.

We presume that all workers are notified of failures, possibly with a delay. Recovery is performed locally and does not interrupt task processing at unaffected workers. After a failure, the program continues with the smaller number of intact workers.

4.3 Task-Level Checkpointing (TC)

In this section, we describe the ***T****ask-level* ***C****heckpointing* technique *TC*. For that, we start by defining work stealing requirements in a general way in Section 4.3.1. Then, we describe the actual fault tolerance algorithm *TC* in a general way in Section 4.3.2, so that *TC* can be applied to a spectrum of task models. Although *TC* is formulated generally, we had to choose a concrete task model for the implementation. We selected *GLB*, and describe the conceptual adaptation of *TC* to *GLB* in Section 4.3.3. Afterwards, we describe several details of our *TC* implementation, which extends GLB_{coop} and is denoted by TC_{GLB}, in Section 4.3.4. We then discuss the correctness of TC_{GLB} in Section 4.3.5, compare TC_{GLB} with $X10\text{-}FT_{GLB}$ in Section 4.3.6, and provide a full code example using TC_{GLB} in Section 4.3.7.

4.3.1 Requirements on Work Stealing

In the following, we state general requirements of *TC* on work stealing. As explained below, they can be established for a spectrum of task models, possibly at the price of a loss in efficiency:

(R1) While a worker's local task pool is not empty, the worker must perform a sequence of *worker steps*, or briefly *steps*. A step consists of the following worker actions:

- take out one or several tasks from the pool,
- process all tasks taken, in any order,
- combine the results of these tasks with the worker result, and
- insert all child tasks that were generated during task processing into the local task pool.

When all tasks taken have been handled by the worker this way, the step ends.

As illustrated in Figure 4.1, between the end of a worker step and the beginning of the next one, there is a *gap*, during which the worker is allowed to communicate. Within a step, however, communication by the worker is forbidden. In particular, the worker must neither deliver nor accept loot.

(R2) Only one steal from the same thief to the same victim may be in progress at a time.

(R3) A steal should leave at least one task in the local task pool.

To establish *(R1)*, we restrict our consideration to cooperative work stealing. Since the victim and the thief actively participate in the stealing, they can postpone their activities until the end of a worker step. For *(R2)*, thieves can remember open steal requests and remove duplicates. For *(R3)*, the victim can reject steal requests that would leave its local task pool empty. Requirement *(R3)* is not strictly necessary but appears sensible and occasionally simplifies bookkeeping.

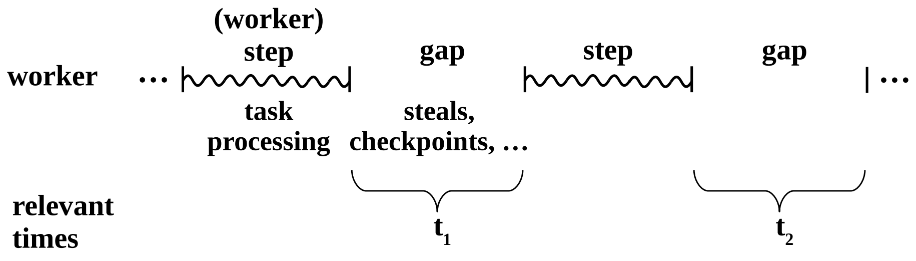


Figure 4.1: Steps, gaps, and relevant times

4.3.2 Fault Tolerance Algorithm

In this section, we formulate *TC* in a general way so that it can be applied to a spectrum of task models. As *TC* is composed of checkpointing and recovery, we first outline the checkpointing scheme in Section 4.3.2.1 and then the recovery scheme in Section 4.3.2.2.

4.3.2.1 Checkpointing

The checkpointing of *TC* is uncoordinated, i.e., each worker autonomously decides when to write a next local *checkpoint*. The term *checkpoint* refers to both the saved data and the event of writing them. Any new checkpoint replaces the previous one.

Checkpoints contain copies of the local task pool contents and the current worker result at the time of their writing, as well as some status information explained later. They are written in the gaps between two worker steps. As illustrated in Figure 4.1, we occasionally refer to these gaps as (relevant) *times*. Note that checkpoints always capture a consistent worker state that includes the complete outcome of all previous tasks (result, child tasks). Checkpoints are written on the following occasions:

- right after initialization of the worker (called *initial checkpoints*),
- at regular time intervals (called *regular checkpoints*),
- in the event of stealing on both the victim and thief sides (called *steal checkpoints*),
- during restore (called *restore checkpoints*), and
- right before the worker becomes inactive (called *final checkpoints*).

Initial checkpoints contain the initial tasks assigned to the worker and an empty worker result. Final checkpoints do not contain any tasks, but the final worker result. The length of the time period between successive regular checkpoints is denoted by r and is measured in seconds.

In each gap between worker steps, the worker checks whether r is over. If so, it writes a regular checkpoint. If a steal or restore checkpoint was performed during the time period, the regular checkpoint is postponed accordingly. Similarly, if both a regular and another type of checkpoint are scheduled for the same gap, the regular checkpoint is omitted.

Checkpoints are saved in the resilient store by a synchronous write operation. No particular type of store is required, but the store must support

- failure-safe storage and retrieval of data,
- transactions to access multiple pieces of data in concert, and
- concurrent access by multiple workers.

Steal checkpoints are part of a steal protocol, which is illustrated in Figure 4.2 and works as follows:

1. The thief contacts the victim, asking for tasks.
2. The victim answers at its earliest convenience. It either sends a reject message (not shown in Figure 4.2), or decides to share tasks.
3. In the second case, the victim extracts the loot from its local pool, and saves it in the resilient store (independent of checkpoints).
4. The victim writes a steal checkpoint.
5. The victim delivers the loot to the thief.
6. The thief inserts the loot into its local task pool.
7. The thief writes a steal checkpoint.
8. The thief notifies the victim about the task adoption.
9. The victim removes the loot from the resilient store.

While a piece of loot is kept in the resilient store, it is called *open*.

All resilient store entries have a unique owner. For checkpoints, it is the worker whose data are saved. For open loot, it is the respective victim. During failure-free operation, all

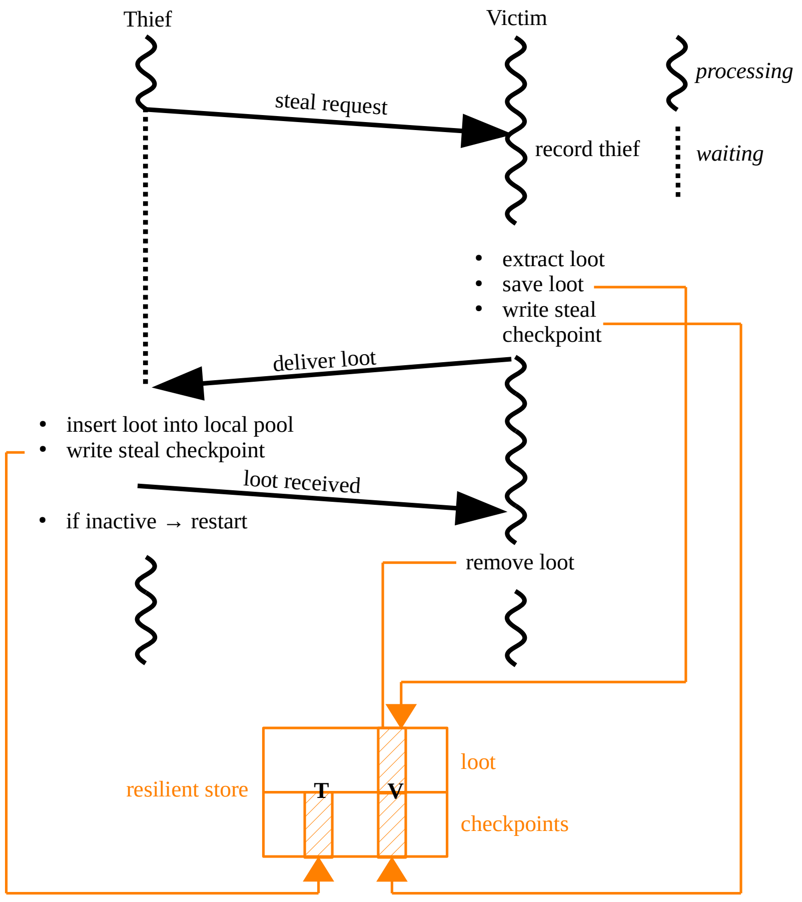


Figure 4.2: *TC*: Steal protocol

accesses are performed by the owner. Thus, there is no need for synchronization. After a failure of a worker x, other workers take care for x's entries. Thus, only after a failure of worker x, non-owners are allowed to access x's entries. To avoid interference with accesses by x, which may arise late because of network delays, each accessing non-owner marks x's entries as `done`. When this flag is set, owner accesses are discarded. This behavior can be programmed with a transaction, i.e., perform data access only if `done` is not set.

4.3.2.2 Recovery

We assume that *all* workers are notified when a worker x failed, although not necessarily at the same time. If a system lacks support for global notification, a worker who observes the failure could notify the others. When receiving a failure notification, a worker executes a *recovery procedure* in the next gap.

In particular, a designated *backup partner* takes over the failed worker's tasks, among other things. This role can be taken by any worker, and possibly at a later time, since the data are held in the resilient store. From an efficiency point of view, timely recovery may pay off, though. A definition of backup partners must meet the following requirements:

- Each worker x must have a unique backup partner. If the backup partner fails before or during its business, succession must be clear.
- A successor of a failed backup partner must not re-execute any actions.
- The backup partner must be able to process the adopted tasks.

A simple deployment may designate worker 0 as the backup partner of all others and crash the program when worker 0 fails. In contrast, we consider workers as being arranged in a ring, according to their numbers and with wraparound. For any worker p, its backup partner is the closest predecessor of p in this ring that is alive. A worker p can easily find out whether it is the backup partner of x, e.g., by inspecting x's ID and the liveliness of all workers from its right neighbor up to x.

The recovery procedure is described in the following.

Recovery from a Single Worker Failure

For simplicity, we first assume that only a single failure occurs, and that each worker can be sure of that. Of course, this assumption is unrealistic. Therefore, the recovery algorithm is actually more complex and involves precautions for multiple-failure cases. After the single failure case, we add the missing details and introduce the complete algorithm.

In the following, we describe the recovery procedure that a worker p performs step by step when notified of the failure of worker x.

1. Worker p records the failure of worker x, to avoid future communication with x. In some task pool variants, further actions may be required to adjust future victim selection. In *GLB*, if x is p's lifeline buddy, p activates the corresponding lifeline. Therefore, p will not send any lifeline steal requests to x in the future. If p has already sent a steal request to x, p considers the request as rejected. If p has recorded a steal request from x, p discards it.
2. For any open loot that has been sent from victim p to thief x, p checks whether this piece of loot is already contained in x's checkpoint. This can be accomplished by inspecting a loot identifier, called `lid`. The `lid`s are consecutive numbers and

system-wide unique. They are sent along with loot deliveries. Each worker records the most recent `lid` of loot sent, and the most recent `lids` of loot received from each worker. From requirement *(R2)*, only the most recent `lids` must be covered. They are held locally and are included in checkpoints.

If x's checkpoint contains the loot, p deletes the loot in the resilient store. Otherwise, p re-merges the loot into its local task pool, deletes the loot in the resilient store, writes a new checkpoint, and marks x's checkpoint as `done`. All actions from operation 2 are carried out in a transaction. Otherwise, it could happen that p inspects the checkpoint and decides to re-merge the loot, but the loot arrives late in x's checkpoint before p has set `done`.

3. In addition to loot sent *to* x, recovery must deal with the tasks in x's checkpoint. The result in x's checkpoint needs not be dealt with, but it is simply kept in the resilient store until the final reduction. The loot sent *from* x will be handled in operation 4.

   If p is x's backup partner, p merges all tasks from x's checkpoint into its local pool and marks the checkpoint as `done`. Afterwards, p writes a restore checkpoint of its own local task pool. These actions are carried out in a transaction

4. If p is x's backup partner, p additionally checks whether x has open loot. If so, p essentially re-sends the loot to the respective thieves, because it is unknown whether x has actually sent the loot. Any receiver makes sure that it does not incorporate the same piece of loot twice, by inspecting the `lids`. Re-sending is done synchronously (see the following description of multiple worker failures). Afterwards, the respective thief has taken over the loot, either in reaction to the original sending or to the re-sending. Therefore, p can now safely remove the loot from the resilient store.

Recovery from Multiple Worker Failures

We consider again the backup partner ring structure described above. With that definition, independent failures, i.e., failures that occur at different times and/or regard disjoint subsets of workers can be handled like a sequence of single failures.

In the following, we examine dependent failures. First of all, we discuss which worker is responsible for restoring a failed worker x, i.e., for performing operations 3 and 4 from the single failure case described above. In contrast to the single failure case, it cannot always be the original backup partner, since that worker may likewise be affected by a failure.

Let us consider the following situation as an example:

$$p \quad x_1 \quad x_3 \quad x_2 \quad x_4 \quad x_0 \quad x_5 \quad q$$

This example shows a sequence of workers in ring order, i.e., p comes first in the ring, then x_1, x_3, and so on. Workers named x_i fail, and the numbers indicate the order of failure, i.e., x_0 fails first, then x_1, x_2, and so on.

When x_0 and x_1 fail, they are restored by x_4 and p, respectively. The failures are independent of each other, and both are handled as in the single failure case described above.

When x_2 fails, x_3 is responsible for the restore of x_2. However, if x_3 fails before or during restoring x_2, p and all others are notified about x_3's failure as usual. Then, the recovery procedure on p takes over responsibility for restoring *both* x_3 and x_2. On the assumption that x_4 crashes shortly after x_3, p is even responsible for restoring x_4.

In general, the recovery procedure of a backup partner of the failed worker iterates over all workers to its right in the ring, until the next place alive. The current worker in this loop is named `iterWorker`. For each `iterWorker` covered, the recovery procedure checks whether restore is needed, i.e., whether the worker's checkpoint contains tasks and/or the worker has open loot.

If a restore is needed, the recovery procedure performs the restore. In our example, upon failure notification for x_3, `iterWorker` takes on values x_3, x_2 and x_4. Assuming that x_5 fails *after* the iteration, it is restored later by p, when p has become x_5's backup partner. Eventually, p becomes the backup partner of q.

Overall, the recovery procedure of each worker p first performs operations 1 and 2 from the single failure case described above and then carries out the steps of the flow diagram in Figure 4.3.

The flow diagram in Figure 4.3 starts with a loop over all failed workers to the right. If no `iterWorker` candidate is found anymore, the recovery has finished ⑩. If an `iterWorker` is found ⓪, the recovery procedure first performs a transaction that carries out operation 3 from the single failure case described above. Use of a transaction avoids, e.g., that the tasks from `iterWorker`'s checkpoint are merged into p's local task pool but remain in `iterWorker`'s checkpoint. If the transaction fails, which is the case if p fails during its execution, the checkpoint remains unchanged and can be restored later by another worker.

Afterwards, the recovery procedure carries out operation 4 from the single failure case described above (handling the loot sent *from* x), which, again, iterates over all open loot in the resilient store of `iterWorker` ①. The current loot in this loop is called `iterLoot` in Figure 4.3. As in the single failure case, it is made sure that the loot is no longer contained

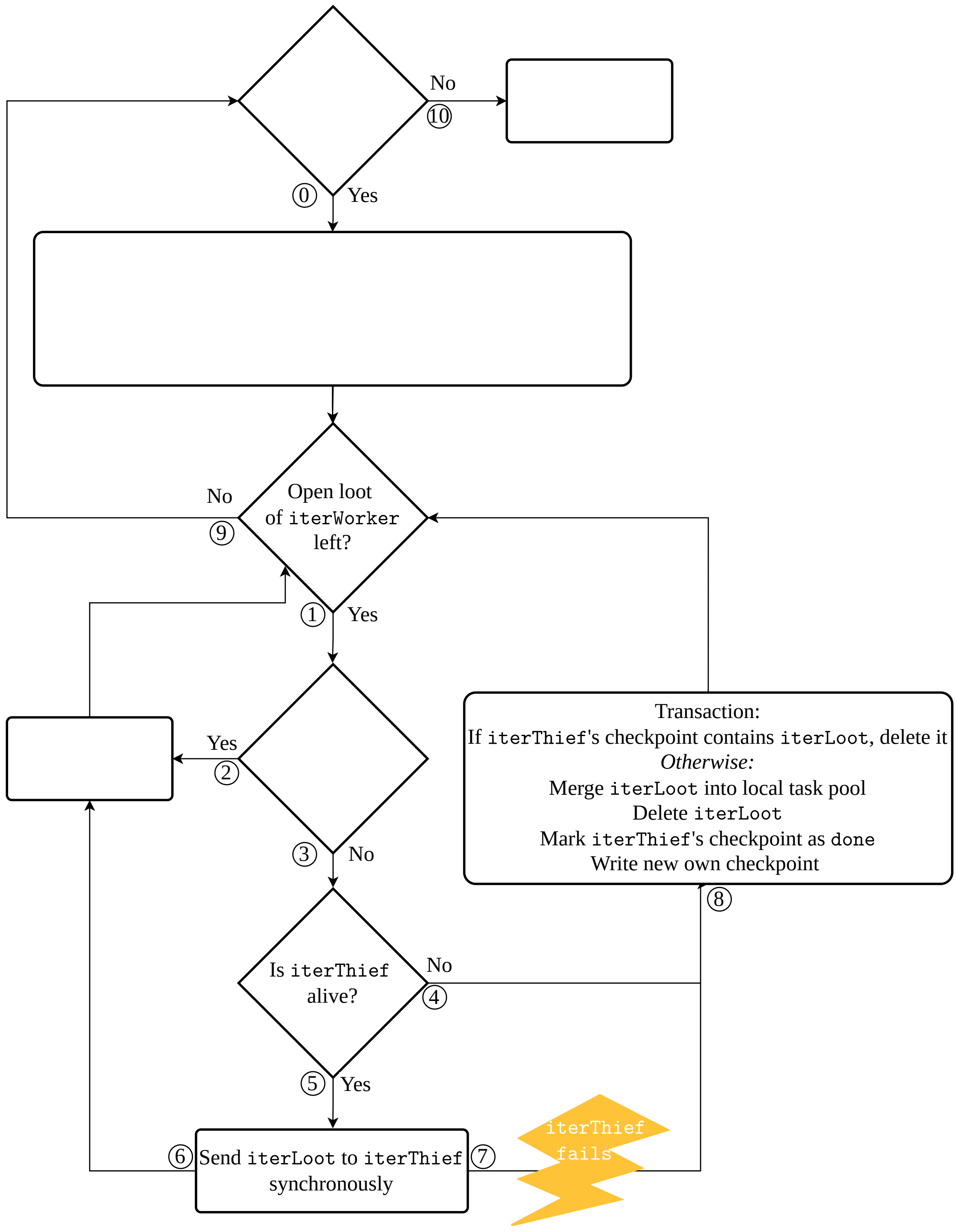

No
10
0
Yes
Open loot of iterWorker left?
No
9
1
Yes
Yes
2
3
No
Transaction:
If iterThief's checkpoint contains iterLoot, delete it
Otherwise:
Merge iterLoot into local task pool
Delete iterLoot
Mark iterThief's checkpoint as done
Write new own checkpoint
8
Is iterThief alive?
No
4
5
Yes
6
Send iterLoot to iterThief synchronously
7
iterThief fails

Figure 4.3: *TC*: Flow diagram of recovery from multiple worker failures

in `iterWorker`'s checkpoint, and otherwise just deletes it ②. If no more `iterLoot` is found for the current `iterWorker`, the next `iterWorker` is dealt with ⑨.

Normally, the recovery procedure tries to re-send the loot to the respective thief, called `iterThief` ③. While not strictly necessary, we first check whether `iterThief` is alive, before synchronously re-sending it. Use of synchronous communication for the re-sending allows one to react immediately to a potential worker failure.

If `iterThief` is alive ⑤, successful return of the synchronous communication for the re-sending ⑥ indicates that the loot has been delivered for sure and can be deleted. This has already been discussed in the single failure case described above. If `iterThief` has already failed ④ or fails during the re-send ⑦, the recovery procedure checks whether `iterThief`'s checkpoint contains the loot ⑧. Otherwise, *p* takes over the loot by merging the tasks into its own local task pool, deleting it from the resilient store, marking `iterThief`'s checkpoint `done`, and writing a new checkpoint ⑧.

The lookup of `iterThief`'s checkpoint and the respective actions in ⑧ need to be performed within a transaction. Otherwise, it may happen, e.g., that `iterLoot` arrives at `iterThief`'s checkpoint after the lookup, but `done` has been set, or that `iterLoot` is deleted but not inserted into *p*'s checkpoint. Again, if *p* fails during the transaction, *p*'s backup partner will find and handle `iterLoot`.

In rare cases, multiple failures may dissect the lifeline graph, such that workers form subgraphs and cannot steal any more from the others [20]. In this case, load balancing malfunctions, but still all tasks are processed by the rest of the workers. Thus, the efficiency drops but the correctness is not compromised. It may pay off to occasionally reconstruct the graph, as suggested in reference [109]. We did not implement that for *TC*.

4.3.3 Adaptation to GLB

Whether the lifeline scheme fulfills *TC*'s requirements on work stealing from Section 4.3.1 depends on details of its implementation, as well as that of the local task pool data structure that is provided by a *GLB* user alongside its application.

In our benchmarks, we implemented the local pools as dequeues. In all cases, access functions for stealing and deliveries operate on one end of the pool, and access functions for the worker's own task processing operate on the other. Steal requests extract at most 50% of the tasks in the pool (benchmark-dependent). We adopt a help-first strategy, i.e., child tasks are inserted into the pool and the processing continues with the parent. *GLB* leaves open whether the *n* tasks for a worker step are taken from the pool as a block or individually. Our implementations take a single task at a time, and completely process it before taking the next one.

In the following, we comment on the validity of *TC*'s requirements.

Requirement *(R1)* is naturally met by the lifeline scheme's computation structure. Each step corresponds to the processing of n tasks, and workers correspond to worker activities. *GLB* synchronization ensures that workers do not communicate during a step. In each step, all child tasks are entered into the pool, and the task results are combined with the worker result.

Requirement *(R2)* may only be violated when a random steal request is sent to a lifeline buddy that has already recorded a lifeline request before. We enforce *(R2)* by discarding the random request on the victim side and treating the lifeline request as if it would have arrived just now.

Requirement *(R3)* is always fulfilled as we extract at most 50% of the pool contents.

4.3.4 Implementation

In this section, we describe several details of our *TC* implementation. For that, we extend GLB_{coop} and denote the result by TC_{GLB}. We first describe some general implementation details in Section 4.3.4.1, followed by unrecoverable situations in Section 4.3.4.2 and our solutions to several technical issues in Section 4.3.4.3.

4.3.4.1 General

As the resilient store, we selected Hazelcast's `IMap` data structure [120] because it nicely fits the stated requirements for a resilient store, see Section 2.2.3. We use separate instances of the `IMap` for checkpoints and loots to simplify the implementation. Our deployed `IMap` operations are described later in Section 4.3.4.3. As described above, we use a ring structure for the backup partners.

The `lids` have type `Integer`. To make the `lids` system-wide unique, they are assigned consecutively to the pieces of loot sent by each victim and combined with the victim's place ID. To keep a record of all loot received in the past, a thief maintains an array that holds one entry per victim. It is held in the local task pool data structure and is thus automatically included in the checkpoints. Similarly, a victim stores its last `lid` sent.

As mentioned before, without failures, each entry in the resilient store can only be accessed by its unique owner. Consequently, locking is only required for accesses by different concurrent *APGAS* activities on the same place. This is necessary because a worker can receive tasks from its lifeline buddy at any time and this is technically executed in two concurrent *APGAS* activities. Recall that in *GLB*$_{coop}$ accesses to the local task pool are protected by a `synchronized` section, see Section 3.2.2. *TC*$_{GLB}$ uses the same lock to protect access to the place's entries in the `IMap` instances.

Listing 4.1 presents simplified pseudocode for the *TC*$_{GLB}$ worker main loop. It extends the corresponding *GLB*$_{coop}$ code from Listing 3.1 and is explained below.

```
1  while (tasks available) {
2    while (task pool is not empty) {
3      synchronized (worker object) {
4        process up to n tasks;
5        for each recorded steal request {
6          write steal checkpoint;
7          send tasks to recorded thief;
8        }
9        if (regular checkpoint time interval is reached) {
10         write regular checkpoint;
11       }
12     }
13   }
14   synchronized (worker object) {
15     attempt to steal from up to w+z victims;
16   }
17 }
18 write final checkpoint;
```

Listing 4.1: *TC*$_{GLB}$: Worker's main loop

As noted before, there are five types of checkpoints: Regular and final checkpoints are invoked in Lines 10 and 18 of Listing 4.1, respectively. Steal checkpoints are invoked in Line 6 and, indirectly, in Line 15. Restore checkpoints are written when tasks are merged during restore, they are not shown in Listing 4.1. Initial checkpoints are written right after initialization of the worker, they are also not shown in Listing 4.1.

Checkpoints of any type include the same data from the local task pool: all tasks, the partial result, the `lids` array, `done`, and possibly other information that a user may have accommodated in the task pool class, see Section 4.3.7.

As described before, Figure 4.2 depicts the steal protocol. We now describe how we have implemented the flow technically. The arrows denote activities invoked with `asyncAt`. If the wavy line is interrupted, the corresponding operation(s) are performed in a `synchronized` section (e.g., `remove loot`), so that no tasks are processed concurrently. Otherwise, the operation is executed concurrently to other local activities, which is the case for `record thief`.

The protocol starts when the thief runs out of tasks or receives a reject message. The steal request shown in Figure 4.2 corresponds to one of the activities started by Line 15 of Listing 4.1. Recall that Line 15 starts one or several activities on up to `w + z` victim places and after each steal attempt waits for an answer by calling `wait()`.

As in *GLB$_{coop}$*, the steal request activity determines whether the victim has tasks to share, just by inspecting a flag. The activity either answers by sending a reject message (not shown in Figure 4.2), or it records the thief as shown.

In the depicted case in Figure 4.2, the victim worker activity notices the steal request in Line 5 of Listing 4.1. It extracts loot, increments it's sent `lid`, stores a (loot, lid) pair in the appropriate `IMap` instance and writes a steal checkpoint to the other `IMap` instance, as shown in Figure 4.2 (after `record thief`). Only then, loot and `lid` are sent to the thief, which is called `deliver loot` in Figure 4.2.

When the thief receives the loot, it checks whether it has already received the same piece of loot before. This may be the case if a piece of loot is sent twice during our recovery protocol. In that case, the second delivery is ignored. The check can be performed easily by comparing the received `lid` with the appropriate entry in the `lids` array.

Usually, the value of this appropriate entry is less than the received `lid`, and the thief merges the received loot into its local task pool. Thereafter, the thief updates its `lids` array, writes a steal checkpoint, and notifies the victim, called `loot received` in Figure 4.2. On the victim place, the `loot received` activity removes the loot from the appropriate `IMap` instance. Finally, if the thief worker was inactive, it is restarted to process the received tasks.

As described before, all workers must be notified about a failure. Technically, *TC$_{GLB}$* registers a `placeFailureHandler` on each place. The *APGAS* runtime invokes these handlers automatically when a place `x` fails and passes `x`'s ID to each handler. The handler performs all actions for recovery, as described in Section 4.3.2.2.

4.3.4.2 Unrecoverable Situations

As noted in Section 4.2, *TC* has no inherent limitations on the number of failures that can be tolerated. Nevertheless, for *TC$_{GLB}$*, failure of Place 0 or loss of an `IMap` partition leads to an unrecoverable situation. The first case is detected by the `placeFailureHandler`s, and the second by the `partitionLostListener` of the `IMap` which is triggered automatically by Hazelcast, see Section 2.2.4. A Place 0 crash cannot be tolerated by *APGAS*, since *APGAS* is not able to migrate the outer `finish` from Place 0 to another place. An `IMap` partition gets lost if all backup copies are gone.

4.3.4.3 Technical Issues

In the following paragraphs, we discuss several interesting technical issues and solutions of our implementation.

DeadPlaceExceptions A place change can be performed with `at` or `asyncAt`. If the remote place is dead when calling such a construct, *APGAS* throws a `DeadPlaceException`. To timely catch these exceptions, we surround each place change by a `try-catch` block. Moreover, there is a `try-catch` block around the outer `finish`, to catch exceptions that are raised during the `asyncAt` blocks. Since our algorithm handles place failures in the `placeFailureHandler`s, all `catch` blocks are left empty, except for the `catch` block that handles the `DeadPlaceException` in Figure 4.3 (marked ⑦).

PlaceFailureHandler We register one `placeFailureHandler` at each place by passing a method reference to the worker constructor. As noted before, the `placeFailureHandlers` are automatically invoked by the *APGAS* runtime when a place crashes. Our implementation of the handlers performs the recovery actions described in Section 4.3.2.2. Note that the handler is invoked, no matter whether the worker activity is active. At the place of an inactive worker activity, the `placeFailureHandler` may merge tasks into the local task pool. In this case, it restarts the worker activity and binds it to the outer `finish` as explained below in paragraph *Restart-Daemon*.

PartitionLostListener We register one `partitionLostListener` on each `IMap` instance. If a partition of an `IMap` instance gets lost, Hazelcast automatically executes the handler code. In our implementation, it prints an error message. Moreover, it terminates the program by starting an asynchronous activity on each place alive that terminates the respective JVM.

Thread-Safe Hazelcast Operations We use two kinds of thread-safe Hazelcast operations to access the two `IMap` instances. Writing checkpoints is performed with the function `executeOnKey()`. This function transfers code to the owner of the respective entry. According to this code, the owner only updates the checkpoint when `done` is `false`. Otherwise, no action is performed.

We implemented our transactions by calling Hazelcast's function `executeTransaction()`. Transactions perform either all or none of the operations passed in a lambda parameter, as has been explained in Section 2.2.4. We surround each transaction by a `try-catch` block to catch a potentially thrown `TransactionException`.

InMemoryFormat Hazelcast supports two storage formats for entries: `Binary` format is more efficient for accesses to whole entries, and `Object` format is more efficient for entry processing. To facilitate our use of `executeOnKey()`, we use `Object`.

Restart-Daemon When a place crashes, the *APGAS* runtime invokes all registered `placeFailureHandlers` by starting a new asynchronous activity on each place which executes the handler code. Unfortunately, there is no option for *APGAS* programmers to bind this activity to a user-defined `finish`. In TC_{GLB}, this results in difficulties, because a `placeFailureHandler` may have to restart an inactive worker and bind the new worker activity to the outer `finish` for correct termination detection.

Therefore, we implemented a workaround. It deploys a so-called *restart-daemon*, which is executed by an additional asynchronous activity on Place 0. We start the daemon in the first line of the outer `finish` block. It runs until the system-wide task pool is empty. The main purpose of the daemon is to restart inactive workers, which received new tasks by a `placeFailureHandler`.

We first added the attribute `restartPlaces` of type `ConcurrentLinkedQueue` on Place 0. If an inactive worker has to be re-started, its ID is inserted into `restartPlaces` by the corresponding `placeFailureHandler`. The restart-daemon cyclically checks `restartPlaces` and, if needed, restarts a worker activity on the corresponding place.

Since the execution of the `placeFailureHandler` is subject to Java scheduling, it can be executed immediately after triggering or at a later time. Therefore, it may happen that the outer `finish` is already terminated when a delayed `placeFailureHandler` starts. This case may result in wrong results because several tasks may not have been processed. To avoid such situations, the daemon finishes only when all handlers have been executed. Therefore, we added an attribute `countHandler` of type `HashMap<Integer, HashMap<Integer, Boolean>>` on Place 0. The first argument, `Integer`, represents the ID of the failed place. The second argument, `HashMap<Integer, Boolean>>`, indicates whether a place has already executed its handler.

For each failure, the first `placeFailureHandler` invoked creates the `countHandler` entry and initializes all HashMap entries with `false`. When a `placeFailureHandler` ends, it writes `true` to its entry as the last operation. The daemon runs until all entries are `true`.

This technique still does not avoid the situation that all `placeFailureHandlers` are delayed and all workers have gone inactive. In this situation, the program could finish and the result would be wrong. To solve this problem, the daemon finishes only if the size of `countHandler` plus the number of places alive is equal to the initial number of places at program start.

Finally, the daemon terminates only when `iMapOpenLoot` is empty. It never handles any open loot itself, however, because delayed `placeRemovedHandlers` will do that. So, the daemon just waits. Checking all entries in `iMapOpenLoot` causes much network traffic, but this operation is only executed in the rare case that all workers are inactive and the last place alive crashes.

Distribution of `IMap` Entries Internally, Hazelcast assigns `IMap` entries to a fixed number of partitions, and evenly distributes these partitions across places. Our `IMaps` contain as many entries as places, and therefore the standard distribution is imbalanced. Our implementations equate the number of partitions with the number of places and deploy a user-defined distribution that saves the checkpoint of place i on place $i-1$. As usual, replicas are distributed randomly.

4.3.5 Correctness

Recall that correctness requires the program to output the correct result or terminate with an error message. At its core, TC_{GLB} correctness is established by the correctness guarantees of Hazelcast and *APGAS*:

- `IMap` entries are safe despite failures. If needed, the `partitionLostHandler` is triggered automatically by Hazelcast, and the program aborts.
- *APGAS* guarantees that all place failures are recognized and the `placeFailureHandlers` are invoked. A Place 0 failure leads to program abort.

In our fault tolerance algorithm, every `IMap` entry exactly captures the *subset* of tasks that are assigned to a worker at the time of checkpoint writing. This includes:

- *finished tasks*, which are captured by the partial result,
- *open tasks*, which are contained in the local task pool, and
- *future tasks*, which have not yet been generated but are encoded in the parent task descriptor.

Each task belongs to one of these groups, since checkpoints are written outside task processing.

Tasks are moved between the subsets of different workers only during stealing and restore. These moves modify multiple `IMap` entries simultaneously, but transactions ensure data integrity despite possible failures.

For stealing, the steal protocol with its handshaking and checkpoint writing on both sides guarantees that the task subsets of victim and thief remain consistent. In particular, the case that failures occur while the loot is in transit is unraveled with the help of the corresponding `IMap` entry.

In recovery, exactly one worker takes over the failed worker's open and future tasks. The partial result is not touched, and therefore the finished tasks stay in the failed worker's subset. Moreover, the recovery protocol from Section 4.3.2.2 guarantees that the loot is taken over by exactly one worker. Altogether, in both stealing and recovery, each task remains in exactly one worker's subset.

The algorithm makes no assumptions on message ordering on system level, but takes care that late messages do no harm. In particular:

- Successive checkpoints of a worker are written one after the other, since checkpoint writing is a synchronous Hazelcast operation.
- Late checkpoints from a failed worker are refused by inspecting the `done` attribute beforehand.
- A victim will only send out further loot to the same thief, if the previous loot was acknowledged.

Termination of the algorithm follows from the continuity of task processing. Workers only interrupt task processing when they perform protocol operations such as answering a steal request, or restoring a worker. All of these operations perform a finite number of actions. When all tasks have been processed, termination is detected by the outer **`finish`**, as described in Section 2.3.

Beyond the theoretical establishment of correctness, we tested our implementation experimentally, by provoking critical situations with `System.exit()` calls, see Section 4.5.1.

4.3.6 Comparison with *X10-FT*$_{GLB}$

Since *TC*$_{GLB}$ and *X10-FT*$_{GLB}$ [20] share some similarities in their designs, we compare them in this section.

A major difference between *X10-FT*$_{GLB}$ and *TC*$_{GLB}$ is the use of the programming languages X10 and Java/APGAS, respectively. *TC*$_{GLB}$'s use of Java allows to utilize Hazelcast and especially its `IMap` data structure. Java has the advantage of being widely used. X10 applications can be compiled to Java or C++. However, *X10-FT*$_{GLB}$ only compiles to C++, otherwise unexpected errors occur.

Another important difference between *X10-FT*$_{GLB}$ and *TC*$_{GLB}$ regards the fault tolerance algorithm. While *TC*$_{GLB}$ relies on a resilient store (the `IMap`), *X10-FT*$_{GLB}$ does not delegate responsibilities to a resilient store. In particular, the *X10-FT*$_{GLB}$ algorithm explicitly deals with the case that a checkpoint gets lost after a place crash. *TC*$_{GLB}$, in contrast, deploys Hazelcast to manage those cases.

Moreover, *X10-FT*$_{GLB}$ manually monitors the liveness of places, whereas *TC*$_{GLB}$ utilizes *APGAS*' `placeFailureHandler`, which is automatically invoked after a failure.

On the positive side, *X10-FT*$_{GLB}$ has no dependencies on foreign libraries. In particular, checkpoint-related communication was implemented the same way as stealing-related communication. Nevertheless, the *X10-FT*$_{GLB}$ algorithm is more complex than the *TC*$_{GLB}$ algorithm, which delegates many responsibilities to the underlying Hazelcast layer. Consequently, *X10-FT*$_{GLB}$ is more difficult to maintain and extend than *TC*$_{GLB}$.

The *X10-FT*$_{GLB}$ algorithm consistently adopts asynchronous communication, which keeps workers responsive. *TC*$_{GLB}$ achieves responsiveness by running multiple concurrent activities on each place. Still, most communication is asynchronous to improve the performance via parallelism between communication and task processing. *TC*$_{GLB}$ uses synchronous communication only for `IMap` accesses, and for re-sending loot after failure.

As mentioned earlier, the `IMap` is fault-tolerant, which is achieved by automatically replicating each partition to other places internally. The number of replicas is configurable from zero to six. If a place leaves the computation, a backup of its hold data is still available (if the number of replicas has been set to at least one). So, the lost data can be automatically restored. A high number of replicas increases the availability of data, but also the network traffic.

TC$_{GLB}$ has the advantage that the number of checkpoint replicas is easily configurable, while *X10-FT*$_{GLB}$ never replicates checkpoints. The *X10-FT*$_{GLB}$ setting could only be changed with a major redesign of the algorithm. Consequently, *TC*$_{GLB}$ can tolerate more cases of simultaneous place failures than *X10-FT*$_{GLB}$. As another advantage, Hazelcast automatically re-writes a backup after loss of a copy for which the original entry is still available. In *X10-FT*$_{GLB}$, that re-writing has to be initiated manually.

TC$_{GLB}$ and *X10-FT*$_{GLB}$ handle partial results in different ways. In *X10-FT*$_{GLB}$, backup partners adopt them after failures. In *TC*$_{GLB}$, they remain in the failed place's `IMap` entry. Consequently, the final result is computed from the partial results of live places in *X10-FT*$_{GLB}$, and from all `IMap` entries in *TC*$_{GLB}$. The *TC*$_{GLB}$ approach is enabled by `IMap` persistence.

Another difference between *TC*$_{GLB}$ and *X10-FT*$_{GLB}$ regards checkpoint handling during stealings. While *TC*$_{GLB}$ writes checkpoints on both the victim and thief sides, *X10-FT*$_{GLB}$ only writes them at the victim side. On the thief side, instead, it stores the victim's identity. Finally, *X10-FT*$_{GLB}$ writes a single steal checkpoint for multiple steals from the same victim. The *TC*$_{GLB}$ approach is simpler but has a higher communication volume.

A performance comparison between *TC*$_{GLB}$ and *X10-FT*$_{GLB}$ can be found in [P4]. The comparison does not show a clear winner, but an older version was used for *TC*$_{GLB}$ than in this thesis, and the version in this thesis is more efficient and scalable [P9].

4.3.7 Usage of *TC*$_{GLB}$

4.3.7.1 Framework Contracts

As noted in Section 2.3.3, *GLB* users must implement the task pool data structure. When using *TC*$_{GLB}$, programmers must additionally implement the following methods:

- `TaskBag getAllTasks()`: Returns all tasks from the local task pool as a `TaskBag` object. The method is called during restore by the backup partner. Note that the tasks are not deleted from the local task pool.

- `void clearTasks()`: Deletes all tasks from the local task pool.

Programmers may also add application-specific fields and methods to the task pool class. These fields may be excluded from checkpointing by marking them with the Java keyword `transient`.

4.3.7.2 Pi

This section illustrates the usage of *TC*$_{GLB}$ with a simple example: the calculation of π with the help of integrals. For this application, we provide the complete code. The example is naturally coded with a static initial work distribution, since all tasks are known at program start. However, for demonstration purposes, we provide implementations for both static and dynamic initial work distributions. The code is depicted in Listings 4.2 to 4.5 and is explained in the following.

`Bag.java` (Listing 4.2) represents a piece of loot, which consists of a set of tasks. Here, a task is represented by a single `Integer` value. This class must implement the TC_{GLB} interface `TaskBag`.

`Queue.java` (Listing 4.3) contains the sequential computation of the actual problem (Line 32), the local task pool (Line 4) and the partial worker result (Line 5). It has to implement the TC_{GLB} interface `TaskQueue`. The second element type is set to `Double`, which defines the type of the result. The first element type is set to `Queue`, which defines the return type of some functions.

Workers process tasks by invoking the function `process()` (Line 32), which pops and processes up to n tasks successively. The function `split()` (Line 22) extracts a set of tasks from the pool and returns them as a `TaskBag` object. The application decides how many tasks are stolen (Line 23).

Since the function `getResult()` (Line 47) returns an instance of `GLBResult`, the inner class `PiResult` (Line 67) extends the TC_{GLB} class `GLBResult`.

The function `init()` (Line 19), which initializes the local task pool, is only invoked when using dynamic initial work distribution (`StartStatically.java`).

`StartDynamically.java` (Listing 4.4) starts this application dynamically so that all initial tasks are located on Place 0. Therefore, the `Queue` constructor (Line 4) and the `GLB` constructor (Line 6) are invoked with `true` as the last parameter. The `result` (Line 8) is an array of size 1, in which index 0 contains pi's value.

`StartStatically.java` (Listing 4.5) starts this example statically by invoking the `Queue` constructor (Line 4) and the `GLB` constructor (Line 6) with `false` as the last parameter. The result has the same form as above.

```
public class Bag implements TaskBag {
  public Deque<Integer> list = new LinkedList<>();

  @Override
  public int size() { return list.size(); }
}
```

Listing 4.2: TC_{GLB}: `Bag.java`

```
1 public class Queue implements TaskQueue<Queue, Double> {
2   int N;
3   double deltaX;
4   Deque<Integer> list = new LinkedList<>();
5   double result = 0;
6
7   public Queue(int n, boolean dynamicDistribution) {
    N = n;
    deltaX = 1.0 / N;
    if (dynamicDistribution == false) {
      int step = N / places().size();
      int start = here().id * step;
      int end = Math.min(start + step, N);
      for (int i = start; i < end; i++)
        list.add(i);
    }
  }

  public void init() { for (int i = 0; i < N; i++) list.add(i); }

  @Override
  public TaskBag split() {
    int size = size() / 2;
    if (size <= 0) return null;
    Bag bag = new Bag();
    for (int i = 0; i < size; i++)
      bag.list.add(list.poll());
    return bag;
  }

  @Override
  public boolean process(int n) {
    double r = 0;
    for (int i = 0; i < n; i++) {
      double x = (list.pop() + 0.5) * deltaX;
      r += 4.0 / (1 + x * x);
      result += r * deltaX;
      if (size() <= 0) break;
    }
    return (size() > 0);
  }

  @Override
  public void merge(TaskBag taskBag) { list.addAll(((Bag) taskBag).list); }

  @Override
  public GLBResult<Double> getResult() { return new PiResult(); }

  @Override
  public void mergeResult(TaskQueue<Queue, Double> that) {
    result += that.getResult().getResult()[0];
  }

  @Override
  public int size() { return list.size(); }

  @Override
  public void clearTasks() { list = new LinkedList<>(); }

  @Override
  public TaskBag getAllTasks() {
    Bag bag = new Bag();
    bag.list.addAll(list);
    return bag;
  }

  public class PiResult extends GLBResult<Double> {
    @Override
    public Double[] getResult() { return new Double[]{result}; }
}
```

Listing 4.3: TC_{GLB}: `Queue.java`

```
public class StartDynamically {
  public static void main(String... args) {
    int N = 1000000;
    SerializableCallable<Queue> init = () -> new Queue(N, true);
    GLBParameter para = new GLBParameter();
    GLB<Queue, Double> glb = new GLB<>(init, para, true);
    Runnable start = () -> glb.getTaskQueue().init();
    Double[] result = glb.run(start);
  }
}
```

Listing 4.4: TC_{GLB}: `StartDynamically.java`

```
public class StartStatically {
  public static void main(String... args) {
    int N = 1000000;
    SerializableCallable<Queue> init = () -> new Queue(N, false);
    GLBParameter para = new GLBParameter();
    GLB<Queue, Double> glb = new GLB<>(init, para, false);
    Double[] result = glb.runParallel();
  }
}
```

Listing 4.5: TC_{GLB}: `StartStatically.java`

4.4 Variants[1]

4.4.1 Incremental and Selective Checkpointing (IncTC)

The ***Inc**remental and selective **T**ask-level **C**heckpointing* technique *IncTC* resembles *TC* but reduces the checkpoint volume by saving less tasks. The technique imposes some additional constraints on the task pool variant to those from Section 4.3.1:

(I1) The owner must operate on one end of the local task pool, and stealings and task deliveries must operate on the other. For simplicity, we denote the owner's end by top, and the other by bottom.

(I2) Each worker step must process a single task.

(I3) The reduction operator should be approximately size-preserving, i.e., the result should have about the same number of bits as each operand.

Constraints *(I1)* and *(I2)* are needed for correctness, whereas constraint *(I3)* impacts efficiency. *IncTC* combines two ideas:

- Checkpoints cover the worker state at some suitable time in the recent past, and
- checkpoints are written incrementally.

In the following, we explain regular checkpoints, stealing, and recovery.

4.4.1.1 Regular Checkpoints

Let us consider any particular local task pool and its worker. From constraints *(R1)* and *(I2)*, each worker step removes the topmost task from the pool, possibly adds one or several tasks at the top, and keeps the rest of tasks in the pool untouched.

Figure 4.4 depicts an example for the evolution of pool contents over time. Only gaps between worker steps, i.e., the so-called (relevant) times, are shown. At times $\tilde{t}$ and t, successive regular checkpoints are written.

The figure depicts task pool examples $P_0 \dots P_5$, where P_1 denotes the task pool right after $\tilde{t}$, and P_5 denotes the task pool right before t. For each depicted pool, R denotes the current worker result. The topmost task A is drawn as a brown grid, and the other tasks are represented by a green striped area. The number of "striped" tasks is denoted by s.

[1]This section is chiefly the work of the co-authors of publications [P7, P9, P12, P15]

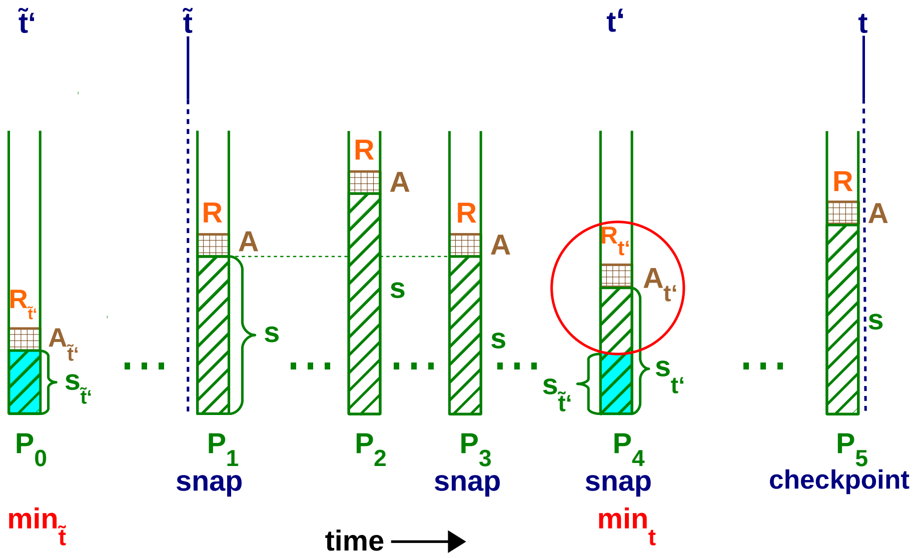


Figure 4.4: *IncTC*: Selective and incremental checkpointing

These tasks remain in the pool during a worker step, and thus we call them *stable*. While s denotes their number, $\mathcal{S}$ denotes the actual tasks.

At any (relevant) time, the *state* of a computation can be represented by the triple $(A, \mathcal{S}, R)$. Note that such a triple includes the outcome of all previous tasks and thus captures a valid state. Our selective checkpointing scheme is based on the following idea: Whenever a regular checkpoint is due at a time t, the state from a recent time $t' \leq t$ is written, where t' minimizes checkpoint size.

To determine t', each active worker monitors s. A and R need not be monitored since their sizes are approximately constant (from *(I3)*). At $\tilde{t}$, and whenever s reaches a minimum, the worker takes a *snapshot*, i.e., it *locally* saves the tupel (A, s, R). Snapshot times are marked by "*snap*" in the figure. They include times when the same minimum is encountered again, since then R and A are more recent. Each snapshot replaces its predecessor in the local store.

Note that the second parameter of a snapshot is a number, whereas states contain tasks. From a snapshot, the corresponding state can be reconstructed by taking s tasks from the bottom of the pool. At checkpointing time t, a *minstate* (denoted min_t) is defined as the state that belongs to the current snapshot. Thus, in Figure 4.4, min_t belongs to snapshot $(A_{t'}, s_{t'}, R_{t'})$. At t, min_t can be reconstructed by taking the bottommost $s_{t'}$ tasks, since the pool contents did not fall below $s_{t'}$ between t' and t.

IncTC combines the above idea with incremental checkpointing, i.e., the scheme does not re-send tasks that are already contained in the current checkpoint. That checkpoint was written at $\tilde{t}$ and contains the state at the last snapshot time $\tilde{t}'$ of the preceding time interval (or was an initial checkpoint). We distinguish two cases:

1. $\boldsymbol{s_{\tilde{t}'} \leq s_{t'}}$: The bottommost $s_{\tilde{t}'}$ tasks (highlighted in Figure 4.4) stayed in the pool from P_0 to P_4, because of the minstate property. Therefore, they are not re-sent. Instead, the checkpoint at t consists of the data marked by a circle in the figure: $A_{t'}$, the $s_{t'} - s_{\tilde{t}'}$ upper striped tasks, and $R_{t'}$.

2. $\boldsymbol{s_{\tilde{t}'} > s_{t'}}$ (not shown in Figure 4.4): Since the bottommost $s_{t'}$ tasks stayed in the pool from P_0 to P_4, they are not re-sent. So, the checkpoint at t includes: $A_{t'}$, $s_{t'}$ (just a number!), and $R_{t'}$. Task $A_{t'}$ is included, since there may have been a previous minstate of the same size.

The checkpoint at t *updates* the previous checkpoint in the resilient store by inserting or deleting the respective tasks.

A slight drawback of *IncTC* over *TC* strikes after failures, when the failed computation must be repeated from t' instead of from t. However, the additional time period is limited by $|t - t'| \leq |t - \tilde{t}| \approx r$.

4.4.1.2 Extension to Stealing and Recovery

Since stealing and recovery are only performed in gaps, the worker's state is clearly defined then.

IncTC deploys the same steal protocol as *TC*, except that the checkpoints contain less tasks. A new snapshot is taken after all types of checkpoint writings. In the following, we modify the notation from Figure 4.4 as follows:

- t denotes the time at which the current (steal) checkpoint $\texttt{check}_t$ is written.
- t' denotes the time at which the current snapshot was taken.
- $\tilde{t}'$ denotes the time at which the pool was in the state that is represented by the previous checkpoint $\texttt{check}_{\tilde{t}} = (A_{\tilde{t}'}, \mathcal{S}_{\tilde{t}'}, R_{\tilde{t}'})$.
- $s_{\texttt{loot}}$ denotes the loot size.

We distinguish several cases:

a) Victim side, $s_{\texttt{loot}} \leq s_{\tilde{t}'}$ *and* $s_{\texttt{loot}} \leq s_{t'}$: Since the loot tasks stayed in the pool from $\tilde{t}'$ to t, $\texttt{check}_t$ equals $(A_{t'}, s_{\texttt{loot}}, R_{t'})$, plus administrative information such as a hint to case a). After receipt, the bottommost $s_{\texttt{loot}}$ tasks are removed from the saved checkpoint.

b) Victim side, $s_{\texttt{loot}} > s_{\tilde{t}'}$ and $s_{\texttt{loot}} \leq s_{t'}$: After t, the worker's computation can no longer be reconstructed from $\texttt{check}_{\tilde{t}}$. So the checkpoint is based on the minstate, i.e., $\texttt{check}_t = (A_{t'}, \widehat{S}, R_{t'})$, where $\widehat{S}$ denotes the $s_{t'} - s_{\texttt{loot}}$ tasks above the loot in the pool at t. This checkpoint *replaces* $\texttt{check}_{\tilde{t}}$.

c) Victim side, $s_{\texttt{loot}} > s_{t'}$: The tasks that remain in the pool after stealing have been generated after t'. Thus, neither $\texttt{check}_{\tilde{t}}$ nor the minstate are suitable to reconstruct the state at t. Thus, $\texttt{check}_t = (A_t, \widehat{S}, R_t)$, where $\widehat{S}$ denotes the $s_t - s_{\texttt{loot}}$ tasks above the loot in the pool at t. Again, this checkpoint *replaces* $\texttt{check}_{\tilde{t}}$.

d) Thief side, empty local task pool: Checkpoint $\texttt{check}_t$ consists of the loot, including its topmost task, and the current worker result. It *replaces* $\texttt{check}_{\tilde{t}}$.

e) Thief side, non-empty local task pool: This case may occur in some task pool variants such as ahead-of-time stealing [167]. The checkpoint sent contains the loot only, and $\texttt{check}_{\tilde{t}}$ is *updated* by including these tasks.

Only checkpoints according to cases b), c) and d) postpone the next regular checkpoint. Otherwise, if checkpoints of different types are due at the same time, the regular checkpoint is written first, followed by victim-side steal checkpoints, and thief-side steal checkpoints (different from *TC*).

The same recovery procedure as in *TC* can be applied since, like there, we always have a valid checkpoint from which a failed worker's computations can be reproduced. The fact that the checkpoint is possibly older does not matter for recovery. We apply one modification: Restore checkpoints, which are written after task adoptions, save less tasks than in *TC*. They are technically the same as steal checkpoints at the thief side, and are handled by cases d) and e)

4.4.1.3 Adaptation to *GLB* and Implementation

Our local task pool implementation from *TC*, see Section 4.3.3, meets Constraint *(I1)*.

Constraint *(I2)* could in principle be established by setting *GLB* parameter $n = 1$. However, that would increase the synchronization costs. Therefore, we introduce two levels of steps: *"Small steps"* are the steps according to requirement *(R1)* from Section 4.3.3, i.e., they process one task and incorporate its children/result. *"Large steps"*, in contrast, are the units after which communication operations are allowed, i.e., the length n of a large step expresses how many tasks must have been processed before communication is allowed. Thus, we typically have $n > 1$, as in *GLB*. Constraint *(I2)* is fulfilled for the small steps. The fact that communication is more rare does not compromise *(R1)*'s correctness. Note that the two-level step structure can only be imposed if the user program takes one task from the pool at a time, as we do. This is a restriction to *IncTC*$_{GLB}$ usage, though.

Constraint *(I3)* is application-dependent. Our benchmarks use the sum operator, which is size-preserving.

The implementation of *IncTC*$_{GLB}$ was performed in a straightforward way by extending *GLB*$_{coop}$.

4.4.2 Combination of Checkpointing and Logging (LogTC)

Like *IncTC*, *LogTC* reduces the checkpointing volume of *TC*. However, it follows a different approach than *IncTC*: **Log**ging timestamps of steals. Moreover, *LogTC* writes checkpoints in parallel to task processing. The technique imposes some additional constraints on the task pool variant to those from Section 4.3.1, which differ from those of *IncTC*:

(L1) All tasks that are in a local task pool at a time must originate from the same task delivery, or from the initial task assignment, respectively. We call such a set of tasks a *task bag*.

(L2) Computations inside worker steps and local task pool accesses must be deterministic. In particular, a take operation must always yield the same task(s), including task order, when applied to the same pool. Moreover, the tasks must be processed in the same order, and child tasks must be inserted into the pool immediately after their generation.

Constraint *(L1)* can be established by incorporating some additional handshaking between victim and thief. For instance, the thief may reject any task deliveries if its pool is non-empty.

LogTC reduces the checkpoint volume of steal checkpoints at the victim side, whereas initial, final, regular, and thief-side steal checkpoints are identical to their *TC* counterparts. We occasionally denote these (identical) checkpoints as *standard*. Restore checkpoints are not needed, as will be explained later.

In both *TC* and *IncTC*, victim-side steal checkpoints contained tasks: more tasks in *TC*, and less tasks in *IncTC*. In *LogTC*, victim-side steal checkpoints never contain any tasks. Instead, their main content is a timestamp for the stealing event. Timestamps specify the number of worker steps that have been executed by the respective worker thus far. Thus, they uniquely correspond to times. If multiple steals are answered at the same time, a separate checkpoint is written for each of them, and the timestamps are supplemented by a sequence number to clarify ordering (see below). From now on, we denote victim-side steal checkpoints as *logs*. In addition to timestamps, logs may contain the loot size, if it is not clear otherwise. We will see later that a sequence of logs, together with the last standard checkpoint, allows to reproduce the victim pool contents after the steals.

Beside changing the content of steal checkpoints, *LogTC* differs from *TC* by asynchronous checkpoint writing. For that, a local copy of the tasks and the worker result is created, and then task processing continues while these data are sent. Consequently, checkpoint writing is separated into starting the checkpoint writing, and waiting for its completion, respectively.

Due to the asynchrony, successive write operations to the resilient store may overtake each other. To avoid race conditions among standard checkpoints, a worker always waits for the completion of a previous checkpoint before starting the next one. Concurrency between standard checkpoints and logs will be discussed later.

The *LogTC* steal protocol is depicted in Figure 4.5. It is asynchronous and combines the formerly independent victim and thief side steal checkpoints into a single transaction:

1. The thief waits for the completion of the previous checkpoint and then sends a steal request.
2. The victim answers with a reject message (not shown), or decides to share tasks.
3. In the second case, the victim extracts the loot from the pool. Then, it invokes a transaction on the resilient store, which is composed of:

   3a) a log, which writes the timestamp (and loot size) to the *victim*'s store entry, and

   3b) a thief-side steal checkpoint, which writes the loot to the *thief*'s store entry.

   The transaction is performed asynchronously. While it is in progress, the victim continues task processing, but it is not allowed to invoke another transaction.
4. When the transaction is completed, the victim asynchronously delivers the loot to the thief.
5. The thief inserts the loot into its local task pool and starts processing it.
6. The victim removes the loot.

Step 1 ensures that the thief-side steal checkpoint (Step 3b) is written after the previous standard checkpoint. The use of transactions in Step 3 ensures that logs cannot overtake each other. Consequently, sequence numbers are clearly defined. To untangle concurrently written checkpoints and logs, checkpoints are extended by a timestamp, as well, which reflects their startup time.

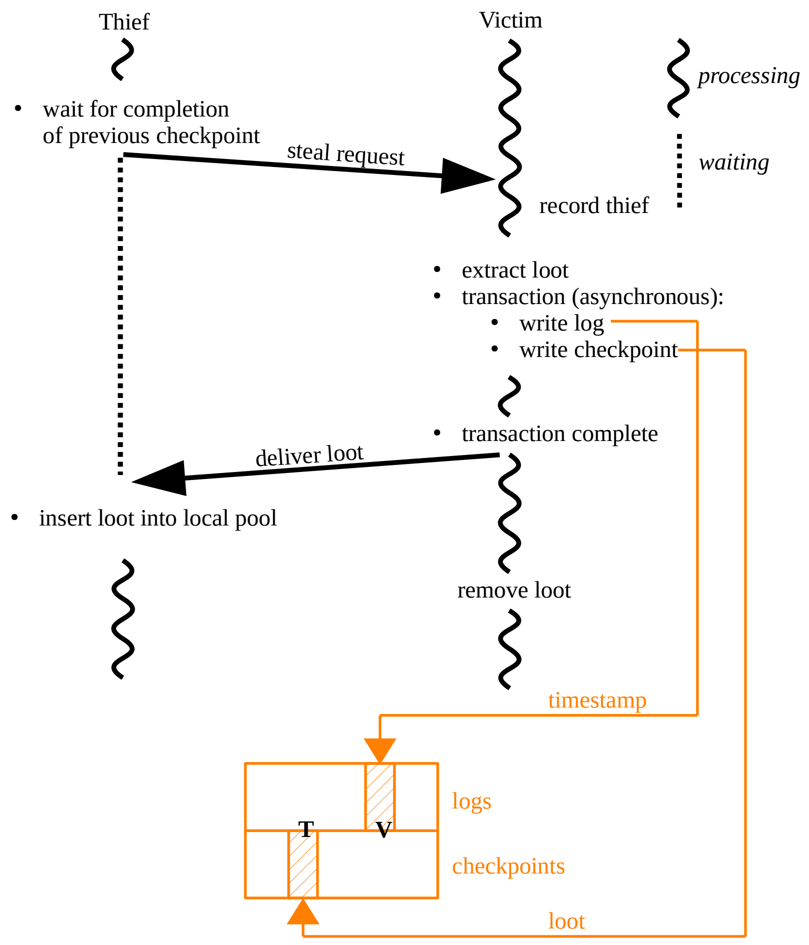


Figure 4.5: *LogTC*: Steal protocol

The recovery procedure resembles that of *TC*. Like there, a backup partner is responsible for handling the failed worker's checkpoint and logs in the resilient store. It first marks these entries as `done`, and then collects the checkpoint and all logs into a *replay unit*.

The following proposition describes how the failed worker's state can be reproduced from the replay unit. Afterwards, we discuss the overall recovery procedure, including the question who performs this recovery.

Proposition 1. *From a replay unit, the victim pool contents after the contained steals can be reproduced.*

Proof. First, all late logs, i.e., logs that have been overtaken by a checkpoint, are removed, since the checkpoint already contains the effects of their steals. Then, without loss of generality, let us consider the first time t_s at which one or several steals of the replay unit took place. The standard checkpoint was written at $t_b \leq t_s$. If $t_b = t_s$, then it was written first. Otherwise, tasks have been processed during (t_b, t_s). Their calculations can be repeated by re-starting the task pool computation from the pool in the standard checkpoint. From *(L2)*, this yields the same pool contents as in the original execution. From *(L1)* and *(R3)*, no loot was received during the time interval $(t_b, t_s]$. Next, the steals are re-applied to the pool, ordered by sequence numbers. For each steal, as many tasks as indicated by the loot size are extracted from the pool and thrown away. From *(L2)*, this yields the same pool contents as in the original execution. The process is repeated for all times at which steals took place (in chronological order). □

Note that a replay unit may contain early logs, i.e., logs that overtook their standard checkpoint. Proposition 1 re-applies them as any others. This is justified as the correctness of the method from Proposition 1 described in the proof does not depend on the frequency of regular checkpoints. Moreover, from Step 4 of the steal protocol, all logs refer to the current task bag. Finally, early logs can be safely consumed, since their "right" checkpoint will not arrive anymore.

The *TC* recovery procedure includes occasional task adoptions: 1) A victim may need to re-adopt the loot sent, and 2) The backup partner may need to adopt the failed worker's saved tasks. Obviously, case 2) is different in *LogTC*, insofar as the tasks must first be reproduced from a replay unit.

Beyond that, task adoptions can violate constraint *(L1)*. To account for that, the recovery procedure *omits* all task insertions into a non-empty pool. Instead, the corresponding worker creates a *description record*, which contains sufficient information to carry out the adoption later. For instance, the description record may contain a link to the failed worker's entries in the resilient store. The description record is inserted into a *replay list*, which is saved in the resilient store. This list must support concurrent accesses.

Entries in the replay list are processed at a more suitable time later. Any worker can do this processing. To avoid increasing the running time of failure-free runs, one may, e.g., adopt the following scheme: The creator of a description record locally saves a link to this record. When it later runs out of tasks or receives a steal request, it resorts to the linked tasks in place of a normal loot. At the very end, a designated worker makes sure that no records are left in the list.

4.4.2.1 Adaptation to *GLB* and Implementation

GLB's lifeline scheme may violate constraint *(L1)*, since task deliveries from lifeline buddies may arrive at any time after the steal request. Therefore, we included some additional handshaking between victim and thief for *LogTC*. In particular, before loot delivery a lifeline buddy first asks its partner whether it is still in need of tasks by spawning an activity on the partner's place. This activity may have to wait for ongoing communication before the answer is clear.

Constraint *(L2)* is naturally fulfilled by our *TC*'s benchmark implementations, see Section 4.3.3.

The *LogTC* description above did not prescribe the design and handling of description records. In our $LogTC_{GLB}$ implementation, a backup partner with a non-empty pool creates a description record that just contains the failed worker's number. It further maintains a local counter for the number of description records that were created but not yet processed. Whenever the backup partner's pool runs empty or it receives a steal request, it consults and possibly decreases this number, and resorts to the corresponding tasks. At the very end, worker 0 checks the resilient store for any left entries.

4.4.3 Supervision with Steal Tracking (SST)

In this section, we describe the ***S****upervision with* ***S****teal* ***T****racking* technique *SST*. As it transfers SST_{NFJ} from the context of nested fork-join programs to our context of dynamic independent tasks, we start by describing SST_{NFJ} in Section 4.4.3.1. We then describe the redesign for dynamic independent tasks in Section 4.4.3.2 and conclude with the adaptation to *GLB* in Section 4.4.3.3.

4.4.3.1 SST_{NFJ} for Nested Fork-Join Programs

We start by describing the original ***S****upervision with* ***S****teal* ***T****racking* technique SST_{NFJ} for ***N****ested* ***F****ork-****J****oin* (*NFJ*) programs [21]. NFJ was introduced in Section 1.3.1, and Listing 4.6 shows an example code for NFJ, which computes Fibonacci numbers and is invoked by calling `fib(n)`. The parent task waits for the results of all children with an explicit `sync`.

```
int fib(int n) {
  if (n < 2) return n;
  int x = spawn fib(n-1);
  int y = spawn fib(n-2);
  sync;
  return x + y;
}
```

Listing 4.6: Nested Fork-Join: Fibonacci

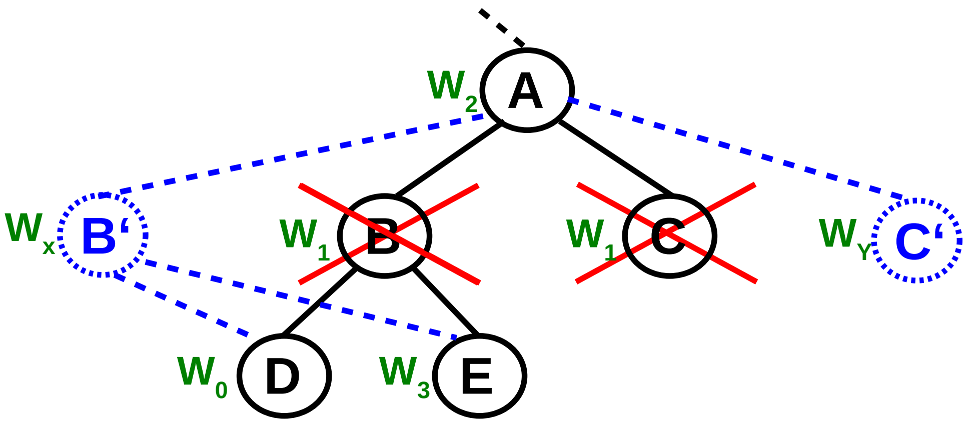


Figure 4.6: SST_{NFJ}: Recovery

SST_{NFJ} refers to a cluster implementation of NFJ [21]. The initial task (here `fib(n)`) is processed by worker 0. At each spawn, a worker branches into the child and places the continuation of the parent task into its local task pool (work-first). Continuations technically have the form of stack *frame*s.

Each steal takes the oldest frame from the local task pool. Thus, the thief processes the parent frame or an ancestor, and the victim processes the child. When a child is finished, the victim keeps the result. When a thief encounters a `sync`, it returns the parent frame to the victim, where it is matched with the child result using a frame ID. Depending on timing, the frame is either sent back to the thief or kept at the victim. The other worker steals a new frame. Matching may have to be applied transitively at a chain of victims.

At each steal, the victim keeps a copy of the stolen frame. If a failure occurs, it initiates re-computation using this copy. This enables recovery from any number of failures, except failure of worker 0. However, a naive re-spawn of the children would cause potentially expensive re-computations of their entire subtrees.

Therefore, the major achievement of SST_{NFJ} is the incorporation of *all* intact subcomputations beneath a faulty one. Figure 4.6 illustrates this concept, with thieves (continuations) drawn below victims. In the example, worker W_1 has stolen tasks `B` and `C` from worker W_2. (It took `C` when `B` was finished, but `D` and `E` had not yet returned.) Similarly, `D` and `E` were stolen by W_0 and W_3, respectively. When W_1 fails, the recovery is led by node `A`, that is, by worker W_2. This worker initiates the re-computations of `B` and `C` (called `B'` and `C'`, respectively), and incorporates the intact subcomputations `D` and `E`, as marked by blue dotted lines.

The feasibility of the approach relies on the following concepts:

- A *steal tree* [168] is a graph with nodes representing frames and edges representing steals, as in the solid line parts of Figure 4.6. Each node is labeled with a *frame ID*, the *history* of this frame (see below), and the *rank* of the processing worker. Frame ID and history are computed at the victim, and then piggybacked onto the loot delivery message from victim to thief and stored at the thief. Frame IDs are quadruples:

  $$frame\ ID = (stage,\ level,\ step,\ victim\ rank),$$

  where *stage* denotes the number of frames that the victim itself had stolen before it was stolen from, and level and step identify the particular frame taken from the victim during this stage. For example, the call `fib(n)` gives rise to two children at the next *level*, and three *steps* for the three continuations encountered (the three subcomputations corresponding to a complete `fib` function, and the remainders after each spawn, respectively). Note that each ID uniquely identifies a frame.

  The *history* of a frame encompasses the IDs of the frame itself, all predecessors in the steal tree, and all pending older siblings of frame/predecessors (e.g., `B` for `C`).

- Upon failure, each worker checks whether it is a victim of the failed worker and has not yet received the result (pending steal). If so, it issues a system-wide call to collect all histories that include the lost frame (e.g., W_2 collects the histories of `D`, `E`). It compresses these histories into a *replay tree*, which supports rapid access to orphaned grandchildren.

- At any following steal, the replay tree is given away to an *alias* worker. Prior to processing the tree, this worker communicates its rank to the orphaned grandchildren. The tree processing itself differs from normal operation. In particular, 1) the stealing of previously unstolen subframes is suppressed, 2) the stealing of lost subframes is enforced (a replay tree is constructed for them beforehand), and 3) the subframes available in orphans are discarded, and the orphan frames are patched instead. Details can be found in [21].

SST$_{NFJ}$ can handle any number of non-root failures. Resiliency during recovery is achieved via bookkeeping of aliases. Moreover, a ForwardUnify protocol reduces data losses in return chains. Details and a discussion of correctness can be found in [21].

4.4.3.2 Redesign of *SST* for Dynamic Independent Tasks

Summarizing the previous definitions, our transformation of *SST*$_{NFJ}$ into *SST* must handle the following differences between DIT and NFJ:

(i) All DIT tasks synchronize with a single ancestor (one-level async-finish structure), whereas each NFJ task synchronizes with its immediate parent.

(ii) DIT results are calculated independently from the spawn tree, by accumulating and combining worker results, whereas NFJ tasks are calculated upwards in the tree.

(iii) Multiple initial DIT tasks may be assigned to one or several workers, whereas NFJ always deploys a single initial task.

(iv) DIT stealing obtains child tasks (help-first), whereas NFJ stealing obtains parent frames (work-first). The DIT tasks are processed from beginning to end, whereas the NFJ tasks are split into continuations.

(v) DIT stealing refers to task bags as opposed to single tasks, and, unlike in NFJ, these bags need not be taken from the pool bottom.

To handle the differences, the design of *SST* imposes two additional work stealing requirements in addition to the DIT guarantees:

1) Staged operation: Each worker must repeatedly steal a task bag, process all tasks from this bag (possibly with the help of thieves), send back the result, steal the next task bag, and so on. Thus, it processes exactly one task bag in each stage (but may store others whose results are still open). Note that stages and phases are different concepts.

2) Determinism: Repeated executions of the same operations on the same local task pool must always yield the same pool contents. Determinism concerns the selection of tasks to be stolen, the extraction of tasks to be processed, and the insertion order of spawned tasks.

To handle difference *(i)*, *SST* imposes an artificial fork-join structure, requesting that thieves report back to their victims when they have finished a bag. Unlike in NFJ, this structure is not visible at the program level. As illustrated in Figure 4.7, the new synchronization granularity is finer than normally in DIT, but coarser than in NFJ.

In Figure 4.7, two initial tasks are processed by four workers marked by different colors. Work stealing gives rise to task bags $B_1 \dots B_5$. For instance, B_2 is stolen by the orange worker from the green one. The orange worker reports back when B_2, including B_3 and B_5, is finished.

The steal tree is defined analogously to SST_{NFJ}, except that nodes represent task bags. For our example, it is shown in Figure 4.7 (right). Analogously to SST_{NFJ}, nodes are labeled by bag ID, history, and worker rank.

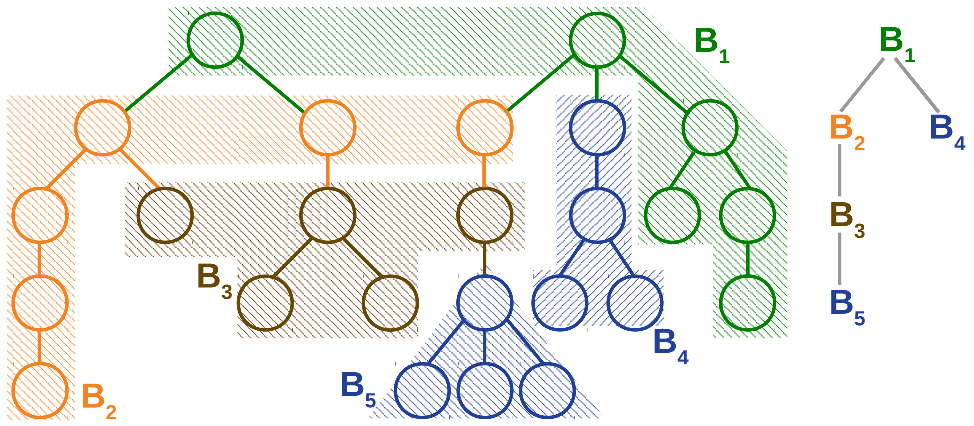


Figure 4.7: *SST*: Task bags and steal tree

To handle difference *(ii)*, results are accumulated per task bag, and are included when the thief reports back to the victim. This is actually simpler than in SST_{NFJ}, as nothing more needs to be done with a finished bag (unlike for SST_{NFJ}'s frames). Thus, after a result return, the thief always proceeds to steal, whereas SST_{NFJ} distinguishes two cases. Similarly, *SST* does not require transitive matching.

The above structure causes the async-finish synchronization to be superfluous, and we therefore omit it.

To handle difference *(iii)*, an artificial root node is inserted into the steal tree if the initial tasks are assigned to different workers (not shown in the figure). This node is labeled with a bag of all initial tasks and assigned to worker 0, whereas its children hold the initial task bags of the different workers. Like a victim, the root node acts as a supervisor and waits for its children's results.

To handle differences *(iv)* and *(v)*, we need bag IDs instead of frame IDs, with the following new definition:

$$bag\ ID = (stage,\ step,\ substep,\ loot\ size,\ victim\ rank).$$

In the definition:

- *stage* is the same as in SST_{NFJ} (using the staged operation requirement),
- *step* is the number of tasks that the victim has processed in this stage before extracting the bag,
- *substep* is the number of tasks that the victim has given away at this step before extracting the current bag, and

- *loot size* is the number of tasks in the bag.

Analogously to SST_{NFJ}, an ID uniquely describes a bag. The history is defined as in SST_{NFJ}, except that we reduce the data volume by omitting the stages and substeps of siblings.

Recovery is performed analogously to SST_{NFJ}. For example, where SST_{NFJ} discards a frame, *SST* removes in the corresponding step as many tasks from the pool as indicated by the loot size. It is the same tasks as in the original execution, according to the determinism requirement. Like SST_{NFJ}, the scheme can handle any number of non-root failures, following the case-by-case analysis in [21].

4.4.3.3 Adaptation to *GLB* and Implementation

We implemented *SST* by extending GLB_{coop} and denote the result by SST_{GLB}. In GLB_{coop}, a worker may receive loot from lifeline buddies while it is still processing another task bag. To ensure staged operation, in SST_{GLB}, we reject such loot with a certain protocol [169]. Moreover, we omit the loot sizes from bag IDs, because *GLB* presumes a predetermined size such as steal-half.

4.5 Experiments

We conducted the following four groups of experiments:

- Correctness tests in Section 4.5.1.
- Comparison and analysis of the running times of *TC* and DMTCP, first in failure-free runs, and then under failures, in Section 4.5.2.
- Comparison and analysis of the memory footprints as well as the running times of *TC* and *IncTC* and *LogTC*, in failure-free runs and under failures, in Section 4.5.3.
- Comparison and analysis of the running times of *TC* and *SST*, in failure-free runs and under failures, in Section 4.5.4.

Note that *TC* is included in all four groups because it is our base checkpointing technique. In addition, the analyses become more and more detailed from group to group, especially regarding the recovery costs.

4.5.1 Correctness Tests

We tested *TC_{GLB}* by provoking process failures with `System.exit()` calls. After each test, we made sure that all workers involved performed the correct actions, by inspecting log files.

The tests were run on *Kassel* [130]. We started 144 workers, which we mapped cyclically onto 12 nodes. We considered the following situations:

1. Worker 2 crashes after processing its first n tasks, but before answering any recorded steal requests.
2. Worker 2 crashes right before it goes inactive.
3. Worker 2 is the victim of a random steal request and crashes right after it has extracted loot, saved it in the `IMap`, and wrote a steal checkpoint.
4. Like situation 3, but worker 2 crashes after sending the loot to the thief.
5. Like situation 3, but for the case of a lifeline steal request.
6. Like situation 3, but additionally the thief crashes when it receives the loot from worker 1 (backup partner of crashed worker 2) during recovery.
7. Like situation 6, but worker 1 (backup partner of crashed worker 2) crashes during handling the thief crash from situation 6.
8. Worker 2 crashes during task processing, approximately in the middle of the overall computation.
9. Worker 2 crashes after merging received loot into its queue and writing the corresponding steal checkpoint, but before sending the `loot received` message to the victim.
10. Like situation 1, but we manipulated the program so that the backup partner waits at the beginning of its `placeFailureHandler` until the rest of the computation has finished. This way, we test the restart-daemon.
11. Like situation 5, but additionally, worker 1 (the backup partner) crashes inside the `merge()` call.
12. Like situation 1, but 90% of the workers crash. This case provokes program abort and tests the detection of unrecoverable situations.

We tested *$IncTC_{GLB}$*, *$LogTC_{GLB}$*, and *SST_{GLB}* in similar ways.

4.5.2 Performance of TC and DMTCP

In this section, we compare and analyze the running times of *TC* and DMTCP, first in failure-free runs, and then under failures.

DMTCP is a user-space library, which checkpoints parallel programs transparently and restarts them from a checkpoint. DMTCP supports many programming languages and HPC environments.

The experiments were run on *Goethe* [131]. We started up to 320 workers, which we mapped cyclically onto a maximum of 8 nodes, since each node comprises 40 CPU cores.

We used DMTCP in version 3.0 (March 7, 2020) [170], and Java in version 1.8.0_221. The resilience mode of *APGAS* was switched on for *TC* runs only. Thus, the numbers reported below include both the overheads of our own technique, and those of resilient *APGAS*.

As a benchmark, we deployed *SWSS* – the synthetic benchmark that was designed with the aim of supporting **S**mooth **W**eak **S**caling for **S**tatic tasks, see Section 2.5.8.2. We configured *SWSS* as follows:

- base computation time $\widehat{T}(p) = 100s$,
- number of tasks per worker 6,000, and
- fluctuation range for the task durations 20%.

Recall, that a failure-free and unprotected run takes time $T(p) = \widehat{T}(p) + \epsilon$ with p workers, where ϵ reflects the costs of dynamic load balancing.

For *TC*, we set the parameter r, which specifies the time period between successive regular checkpoints, to 10 seconds. The value was determined on the basis of the Daly formula [171], which, depending on inputs such as system MTBF, gave us $r = 10 \dots 1000$ seconds. We conservatively used the minimum from this range, to avoid reporting too optimistic results.

4.5.2.1 Running Times of Failure-Free Runs

Figure 4.8 depicts the total running times of non-resilient GLB_{coop}, *TC*, and DMTCP, reporting averages over 5 runs. Note that the y-axis starts at $100s$. DMTCP writes checkpoints of GLB_{coop} and was configured as follows:

- *shm once:* one global checkpoint is written at half of $\widehat{T}(p)$; each worker uses its local memory as storage.

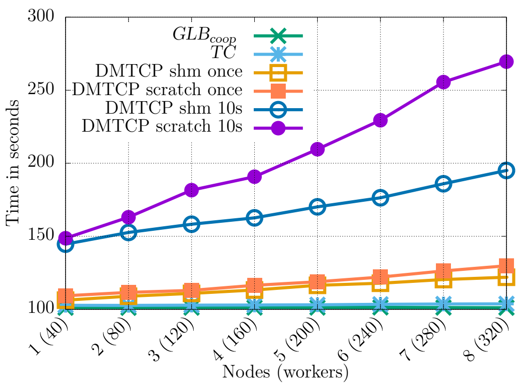


Figure 4.8: Failure-free running times of *SWSS* on *Goethe*

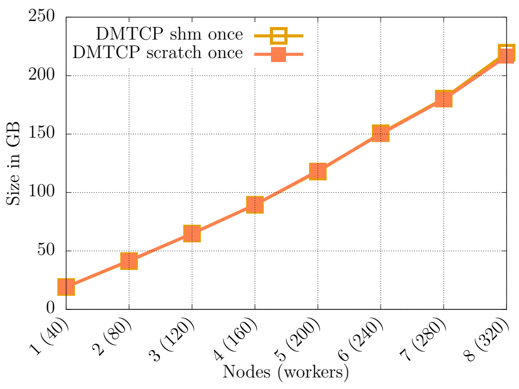


Figure 4.9: Size of DMTCP checkpoints of *SWSS* on *Goethe*

- *scratch once:* one global checkpoint is written at half of $\widehat{T}(p)$; the parallel file system is used as storage.
- *shm/scratch* 10*s:* global checkpoints are written periodically every 10 seconds of processing time (using local memory or parallel file system, respectively). This low value is actually unrealistic in practice but was only used for comparison purposes with *TC*.

DMTCP checkpoints are saved uncompressed, resulting in an average size of 0.6 GB per worker, see Figure 4.9. Naturally, checkpoints written to local memory would be lost in the event of a node failure. However, the results of the *shm* runs can be treated as a lower bound, since local memory offers higher bandwidth and scalability with the number of nodes, in contrast to parallel file systems. For the *scratch* runs, we ensured that the parallel file system was idle.

We measured negligible overheads when we configured DMTCP to not write any checkpoints, not shown in the figures. As noted above, Figure 4.8 shows failure-free runs for which the following observations can be made:

- GLB_{coop} constantly needs approximately $101s$, for any number of workers. Since we configured $\widehat{T}(p) = 100s$, this is as expected and reflects the smooth scaling and low cost (about $1s$) of work stealing.
- *TC* writes an average of 64 checkpoints per worker (8 of which are regular). Since regular checkpoints would only be written every $10s$, the high value reflects the additional checkpointing costs caused by work stealing. This results in an overhead of up to $2.64s$ with 320 workers compared to non-resilient GLB_{coop}. The overhead

increases gently with the number of workers. As noted before, a more detailed performance analysis of *TC* follows in the next sections.

- *DMTCP shm once* takes 5.66s for writing one global checkpoint with 40 workers; and 23.30s with 320 workers. The time required for writing increases with the number of workers and is caused by an increasing coordination overhead when writing checkpoints, because the local memory bandwidth remains unchanged.

- *DMTCP scratch once* takes 8.55s for writing one global checkpoint with 40 workers; and 29.05s with 320 workers. The difference to *shm* increases with the number of workers, caused by the limited (non-scaling) bandwidth of the parallel file system.

- *DMTCP shm/scratch 10s* takes 81.44s using the local memory, and 175.11s using the parallel file system, both with 320 workers. Since DMTCP is capable of optimizing the writing of multiple checkpoints, the total time required does not increase linearly with the number of checkpoints.

Overall, *TC* shows significantly lower failure-free running times overhead and much better scalability than DMTCP. This is mainly because TC only checkpoints selected data and DMTCP checkpoints full program states. Again, a more detailed analysis of *TC* follows in the next sections.

4.5.2.2 Recovery Overhead

Failures are handled differently: *TC* performs a shrinking recovery, whereby a program continues execution with fewer workers. In contrast, a program that was checkpointed by DMTCP aborts if a failure occurs, but the program can be restarted from its last checkpoint.

Figure 4.10 presents the DMTCP restart times using the original number of nodes. It can be observed that the restart costs increase with the number of workers. The increase is stronger for *scratch* than for *shm*, because, again, the limited bandwidth of the parallel file system is a dominating factor.

Figure 4.11 depicts measured running times with 40 workers, where we crashed up to 12 random workers at random times. We executed each configuration 100 times and report averages. Overall, it can be observed that *TC* is by far faster than both DMTCP *shm*/*scratch* 10s.

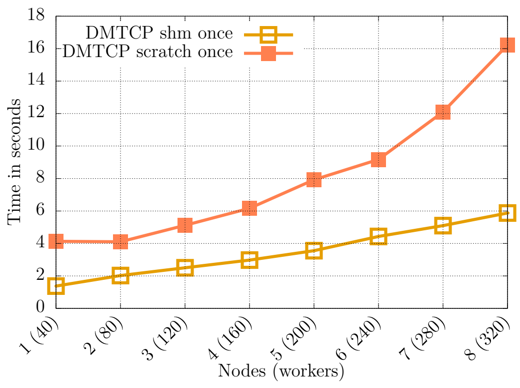


Figure 4.10: DMTCP restart times of *SWSS* on *Goethe*

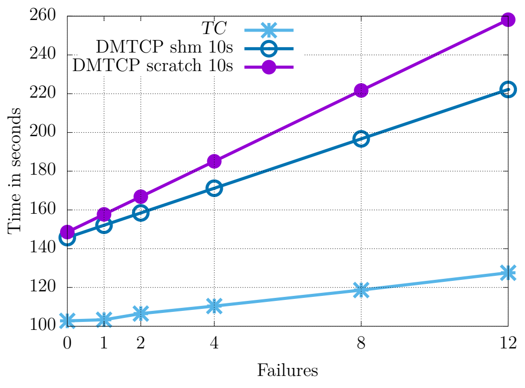


Figure 4.11: Total running times for failures out of 40 workers for *SWSS* on *Goethe*

4.5.2.3 Summary

In this section, we have compared the performance of *TC* and DMTCP. The results show *TC* as the clear winner, with both a significantly lower failure-free running time and a significantly lower recovery overhead than DMTCP. As noted before, a more detailed performance analysis of *TC* follows in the next sections.

4.5.3 Performance of TC, IncTC, and LogTC

In this section, we compare and analyze *TC*, *IncTC*, and *LogTC* to evaluate

- the running time overheads of failure-free runs,
- the memory footprints, and
- the recovery overheads.

Experiments were run on *Kassel* [130]. We started up to 144 workers, which we mapped cyclically onto a maximum of 12 nodes, since each node comprises 12 CPU cores.

We deployed Java in version 11.0.2. The resilience mode of *APGAS* was switched on for *TC*, *IncTC*, and *LogTC* runs only.

In the following, we list our used benchmarks and parameter settings. Values for the *GLB* parameter n (the number of tasks per step) were determined experimentally for each benchmark, so as to minimize the running time for different steal rates and task granularities. Similarly, we experimentally determined the percentages of tasks that are extracted in steals as 10% for *UTS*$_E$ and *VBD*/*VBS*, and 50% for *NQueens* and *BC*$_E$. As noted in Section 2.5.8, *VBD* and *VBS* – **V**ariable **B**allast for **D**ynamic/**S**tatic tasks – are synthetic benchmarks that were designed with the aim of simulating large task pools.

For all benchmarks, we increased the problem size with the number of workers by adjusting benchmark-specific parameters, such that all running times are in the 100 . . . 1000 seconds range. The ellipsis below indicates the corresponding parameter range, and Table 4.1 contains examples of concrete running times.

Both *VBD* and *VBS* were always run with 144 workers. Instead, we increased the dummy ballast. Here are the benchmark parameters:

- *UTS*$_E$: geometric tree shape, branching factor $b = 4$, initial seed $s = 19$, tree depth $d = 15 \ldots 19$, $n = 511$.
- *NQueens*: $N = 16 \ldots 18$, threshold $t = 10 \ldots 12$, $n = 511$.
- *BC*$_E$: initial seed $s = 2$, number of graph nodes $N = 2^{16 \ldots 19}$, $n = 127$.
- *VBD*: number of child tasks randomly selected from $[1, ..., 27]$, tree depth $d = 7$, precision $g = 1$, ballast $b = 0 \ldots 10$ MB, $n = 127$.
- *VBS*: number of tasks $t = 100000$, precision $g = 300$, ballast $b = 0 \ldots 0.4$ MB, $n = 12$.

In all runs, we kept the regular checkpointing interval r at 10 seconds, as in Section 4.5.2. The impact of r on the running time will be further discussed below.

	non-resilient *GLB*$_{coop}$	***TC***	***IncTC***	***LogTC***
***UTS*$_E$**, $d = 19$	886.78	890.53	898.85	889.00
***BC*$_E$**, $N = 2^{19}$	718.56	724.73	748.23	758.57
***BC*$_E$**, $N = 2^{19}$, realistic r	718.56	719.71	727.39	725.57
NQueens, $N = 18$	541.82	543.21	543.99	543.62
VBD, $b = 10\ MB$	503.78	2101.37	802.37	669.78
VBS, $b = 0.4\ MB$	56.73	115.18	57.78	65.83

Table 4.1: Running times in seconds with 144 workers on *Kassel*

4.5.3.1 Running Times of Failure-Free Runs

This section refers to overheads instead of absolute running times, to make the presentation more clear. The overhead is specified as a percentage, and is calculated with the formula $\texttt{time}_\texttt{x}/\texttt{time}_\texttt{GLB} - 1$, where $x \in \{TC, IncTC, LogTC\}$.

Figures 4.12a and 4.13 depict the running times of failure-free runs of *UTS*$_E$, *NQueens*, and *BC*$_E$, respectively. In the figures, grey colored areas mark worker ranges that were run with the same benchmark-specific parameters.

***UTS*$_E$:** For *UTS*$_E$, the overheads are at most 2.96% (for *IncTC* with 2 workers), see Figure 4.12a. The overheads are lowest for *LogTC*, which is closely followed by *TC*, while *IncTC* has the largest overheads.

To better explain the results, Figure 4.12b shows the number of steals, and Figure 4.12c shows the number of checkpoints per worker and second. The curves share some similarities with those in Figure 4.12a, such as the peak at 72 workers.

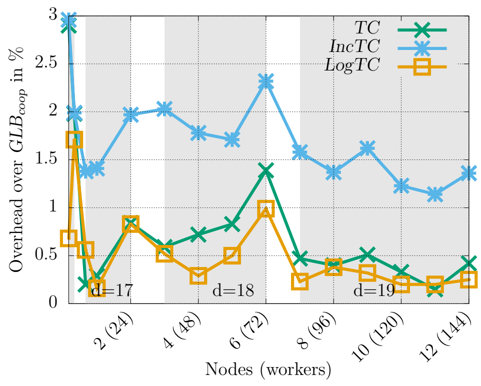
As Figure 4.12b shows, the steal rate increases within each grey colored area. The reason for this is that with the same total work and an increasing number of workers, each worker gets fewer tasks, and thus it has to steal more often.

Figure 4.12c shows that the number of checkpoints per worker and second grows in the same way as the steal rate, which is obviously due to a growing number of steal checkpoints. Additionally, one can see that regular checkpoints, which are written every 10 seconds, play only a minor role in the running time overhead. For example, with 144 workers, each worker writes a checkpoint about every 3 seconds. Correspondingly, Table 4.2 shows that about 88% of the checkpoints are steal checkpoints.

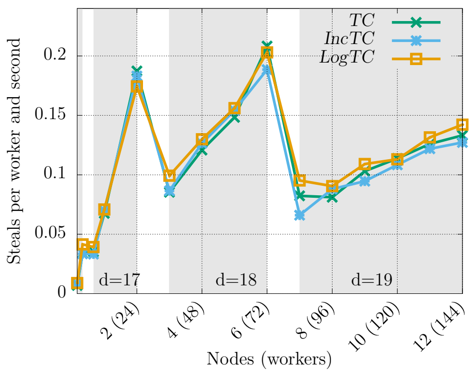
***NQueens*:** For *NQueens*, the overheads are at most 1.39% (for *TC* with 72 workers), see Figure 4.13a. The overheads fluctuate without a clear winner.

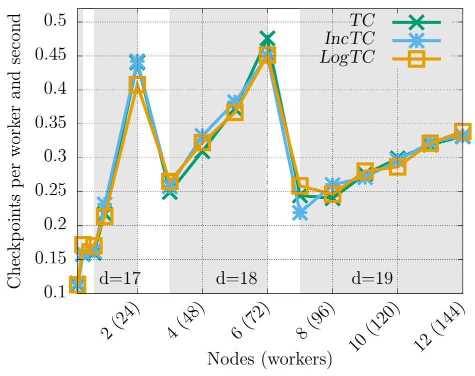
***BC*$_E$:** For *BC*$_E$, the overheads are at most 6.06% (for *LogTC* with 132 workers), see Figure 4.13b. Above 24 workers, they are quite stable, except for *IncTC*, with a relatively clear ranking: *TC* performs best with overheads up to 0.86%, followed by *IncTC* with overheads up to 4.13%, and *LogTC* with overheads up to 5.57% (all with 144 workers).

In general, the *BC*$_E$ overheads are higher and more stable than those of *UTS*$_E$ and *NQueens*. This can be attributed to different benchmark characteristics: First, all tasks are known from the beginning, and thus the steal rate is significantly less. In consequence, the number of steal checkpoints is less than the number of regular checkpoints (see Table 4.2). Correspondingly, checkpoints are written about every r seconds only. Second, with an increasing number of graph nodes (parameter N), the task granularity increases considerably (see Table 4.2). For instance, with 144 workers, processing n tasks takes 25 seconds. Therefore, checkpoints are only written about every 25 seconds. Finally, in contrast to *UTS*$_E$ and *NQueens*, the result is not a single `long` value, but an array. Since the result is contained in checkpoints, they are larger than for *UTS*$_E$ and *NQueens* (Table 4.2). For instance, with 144 workers, a single checkpoint has 4 MB, in contrast to *UTS*$_E$ and *NQueens* checkpoints with 2 KB and 30 KB, respectively.



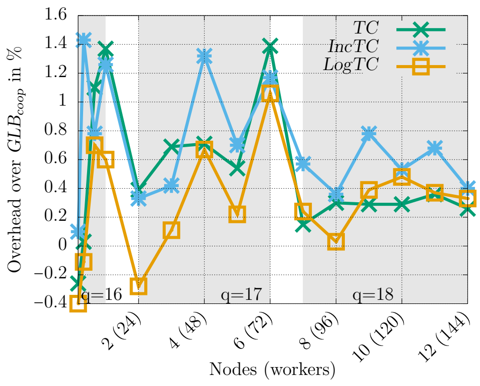
(a) Failure-free running time overheads of *TC*, *IncTC*, and *LogTC* over non-resilient *GLB_{coop}*

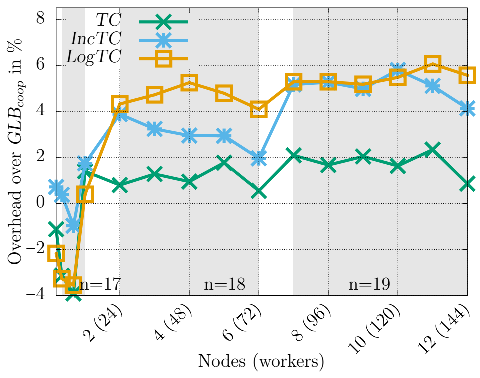




(b) Number of steals per worker and second **(c)** Number of checkpoints per worker and second

Figure 4.12: *UTS_E*: a) failure-free running time overheads, b) number of steals, and c) number of checkpoints of *TC*, *IncTC*, and *LogTC* on *Kassel*



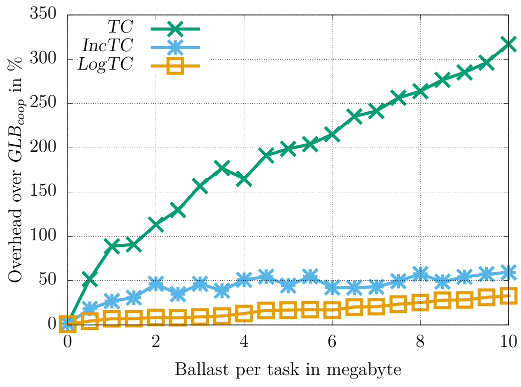


(a) *NQueens* **(b)** *BC_E*

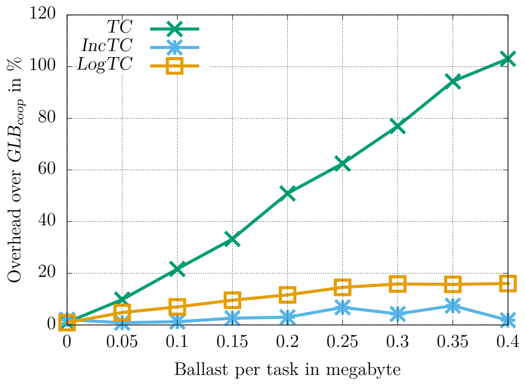
Figure 4.13: *NQueens* and *BC_E*: Failure-free running time overheads of *TC*, *IncTC*, and *LogTC* over non-resilient *GLB_{coop}* on *Kassel*

	Single Task	n Tasks	Checkpoint Size	Steal Checkpoints
***UTS*$_E$**	0.10 – 0.11 microseconds	55 – 60 microseconds	1 – 2 KB	12 – 88 %
NQueens	70 – 120 microseconds	36 – 60 milliseconds	23 – 30 KB	83 – 96 %
***BC*$_E$**	5 – 200 milliseconds	0.65 – 25 seconds	0.60 – 4 MB	3 – 16 %
VBD	0.70 – 15 milliseconds	0.08 – 2 seconds	2 KB – 235 MB	32 – 37 %
VBS	83 – 221 milliseconds	1 – 2.60 seconds	5 KB – 53 MB	60 – 96 %

Table 4.2: Average task processing time (task granularity), average checkpoint size per worker, and percentage of steal checkpoints in relation to all checkpoints on *Kassel*



(a) *VBD*



(b) *VBS*

Figure 4.14: *VBD* and *VBS*: Failure-free running time overheads of *TC*, *IncTC*, and *LogTC* over non-resilient *GLB$_{coop}$* with 144 workers on *Kassel*

Synthetic benchmarks (*VBD* and *VBS*): For the synthetic benchmarks, as expected, the overheads tend to increase with ballast, see Figure 4.14. Table 4.2 shows that the checkpoint sizes increase up to 235 MB (*VBD*) and 53 MB (*VBS*), respectively. Both values are significantly higher than those of the other benchmarks.

VBD shows a clear ranking: *LogTC* performs best with overheads of up to 32.95%, *IncTC* ranks second with overheads up to 59.27%, and *TC* loses with large overheads up to 317.12% (all with 10 MB ballast per task). Obviously, the reduced checkpoint volume of *IncTC* and *LogTC* pays off.

Results for *VBS* show a clear ranking, as well, although a different one. Here, *IncTC* performs best with overheads up to 7.47%, and *LogTC* ranks second with overheads up to 16.05%. *TC* again loses clearly with overheads up to 103.04% (with 0.35 MB or 0.4 MB ballast per task, respectively).

Overall, there is no clear ranking between our three fault tolerance techniques, but the results are benchmark-dependent. This outcome can be explained by different pros and cons of the three algorithms: *IncTC* and *LogTC* have a lower checkpoint volume than *TC*, and thus lower communication costs. The difference is, however, only noticeable for large pools, since otherwise communication costs are dominated by latency. On the backside, *IncTC* causes additional overhead for monitoring, and *LogTC* causes additional overhead for local task copying and handshaking. The latter is required to avoid task deliveries to non-empty pools.

Let us finally consider the impact of *GLB* factor r. As stated in Section 4.5.2, r should be calculated with the Daly formula, which gave us a $[10, 1000]$ seconds range. We performed most experiments with $r = 10$ seconds, but also tried $r = 500$ seconds. With this setting, the BC_E overheads of our fault tolerance techniques is only 0.16% for *TC*, 1.23% for *IncTC* and 0.98% for *LogTC* (all with 144 workers). In this sense, the reported values are a kind of conservative upper bound for the success of our techniques.

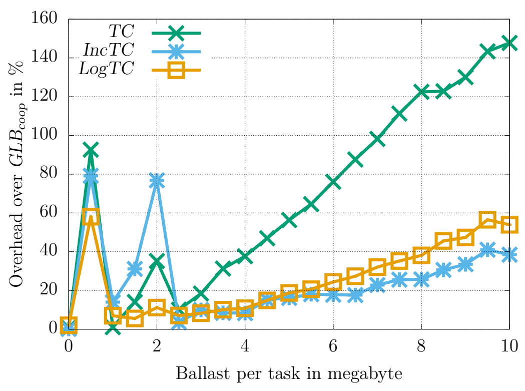
4.5.3.2 Memory Footprint

Beside running time, memory footprint is relevant, since it determines the biggest possible problem size, respectively the number of workers that can be assigned to each node. We calculated memory footprints by taking the average over the peak memory consumptions of all workers during a job's runtime. As in the previous paragraph, we report overheads of our fault tolerance techniques over GLB_{coop}, and express them as a percentage.

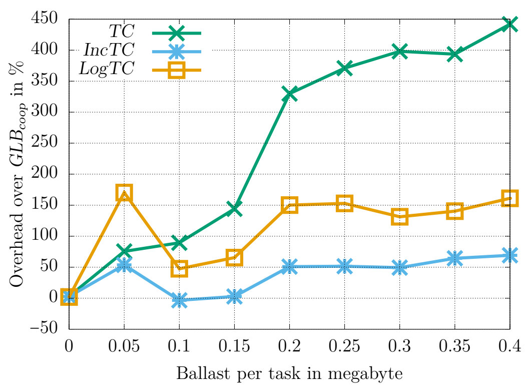
For clearer results, experiments for this section were performed with the synthetic benchmarks. The results are depicted in Figure 4.15, for the same parameters as before.

VBD results vary for small task descriptor sizes, but above 4.5 MB ballast per task the picture is clear: *IncTC* needs the least memory with overheads of at most 38.48%, followed by *LogTC* with overheads of at most 53.89%, and *TC* with overheads of at most 147.74% (all with 10 MB ballast per task). Absolute values are given in Table 4.3.

VBS yielded similar results. Here, a clear picture arises above 0.1 MB ballast: *IncTC* has the lowest memory footprint overheads with at most 69.17%, followed by *LogTC* with at most 161.14%, and *TC* with up to even 441.92%. Again, absolute values can be found in Table 4.3.



(a) *VBD*



(b) *VBS*

Figure 4.15: *VBD* and *VBS*: Failure-free memory footprint overheads of *TC*, *IncTC*, and *LogTC* over non-resilient GLB_{coop} with 144 workers on *Kassel*

	non-resilient GLB_{coop}	***TC***	***IncTC***	***LogTC***
***VBD* (10 MB)**	6,826.50 MB	16,911.80 MB	9,453.64 MB	10,505.59 MB
***VBS* (0.4 MB)**	3,278.70 MB	17,768.01 MB	5,546.52 MB	8,561.91 MB

Table 4.3: Memory footprint of one worker on *Kassel*

4.5.3.3 Recovery Overhead

In our third group of experiments, we determined the time for handling worker failures. It includes failure detection by *APGAS*, execution of the recovery procedure, and re-processing of the lost tasks. For that, we compared the running times of three UTS_E executions:

A: 144 workers without crashes,

B: 132 workers without crashes, and

C: 144 workers with 24 workers crashes after half of *A*'s running time.

Since executions *B* and *C* use on average the same total amount of computing resources (because *C* crashes 24 workers after half of *A*'s running time), the recovery overhead can be estimated by $t_C - t_B$. This approximation is somewhat rough, because UTS_E has a highly irregular workload, with more workers idle in the second half of the execution of *A* (more specifically, in the final phase of the computation when the number of tasks becomes fewer and fewer) than in the first. Thus, the reduction in resources hurts less than.

The value of 24 for the number of worker crashes is unrealistically large. We used it to obtain measurable results. To avoid a program crash despite the large value, we set

`IMap`'s number of replicas to 6. Two workers per physical node were crashed by calling `System.exit()`.

Table 4.4 depicts the measured running times in cases *A*, *B* and *C*. With the above formula, we obtain an estimated recovery overhead of 3.35% for *TC*, 3.64% for *IncTC*, and 2.96% for *LogTC* (all for 24 failures). Since the recovery overhead for a single failure case is about $\frac{1}{24}$ of the measured 24 failures overhead, the recovery overheads for a single failure are negligible in all cases.

	Workers	***TC***	***IncTC***	***LogTC***
A	**144**	904.54	909.80	902.12
B	**132**	984.79	989.87	983.84
C	**144 – 24**	1017.79	1025.93	1012.96

Table 4.4: Running times in seconds for UTS_E with $d = 19$ and `IMap`'s number of replicas = 6 on *Kassel*

4.5.3.4 Summary

All three fault tolerance techniques – *TC*, *IncTC*, *LogTC* – have low overheads in failure-free runs, typically below 6%. Furthermore, the recovery overheads after failures are quite low. Thus, all three techniques constitute an efficient alternative to system-level checkpointing for task-based applications.

The difference between the three techniques is less clear, however. In the practically relevant case of small task pools, they have a similar performance. Here, *TC* seems to be the best choice, since it is easiest to implement, and does impose the least number of constraints. In particular, a *TC* step may process multiple tasks, and thus the technique is suitable for compact task representations.

For large task pools, in contrast, *IncTC* and *LogTC* significantly outperform *TC*. The choice between the two techniques depends on a task pool scheme's compliance with the additional constraints, respectively the costs for establishing them.

4.5.4 Performance of TC and SST

In this section, we compare and analyze the running times of *TC* and *SST*, first in failure-free runs, and then under failures.

The experiments were run on *Kassel* [130] and on *Goethe* [131]. Again, we mapped the workers cyclically onto the lowest possible number of nodes, and started up to 144 workers on *Kassel* (on up to 12 nodes) and up to 640 workers on *Goethe* (on up to 16 nodes).

We deployed Java in version 13.0.2. The resilience mode of *APGAS* was switched on for *TC* and *SST* runs only.

In the *TC* implementations, we kept the regular checkpoint interval r at 10 seconds, as in Section 4.5.2. We focused on small task sizes, to obtain clearer results. The benchmark parameters and the *GLB* parameter n were set as follows:

- *UTS*$_E$: geometric tree shape, branching factor $b = 4$, initial seed $s = 19$, tree depth $d = 18$ (*Kassel*) or tree depth $d = 19$ (*Goethe*), $n = 511$.
- *NQueens*: $N = 17$, threshold $t = 11$ (*Kassel*) or $N = 18$, threshold $t = 12$ (*Goethe*), $n = 511$.
- *BC*$_E$: initial seed $s = 2$, number of graph nodes $N = 2^{18}$, $n = 511$.
- *SWSD*: $\widehat{T}(p) = 100s$, tasks per worker $t = 1,000,000$, average task fluctuation $f = 20\%$, $n = 511$.
- *SWSS*: $\widehat{T}(p) = 100s$, tasks per worker $t = 6000$, average task fluctuation $f = 20\%$, $n = 1$.

As noted in Section 2.5.8.2, both *SWSD* and *SWSS* are the synthetic benchmarks that were designed with the aim of supporting **S**mooth **W**eak **S**caling for **D**ynamic/**S**tatic tasks. These benchmarks enable an accurate analysis of the performance of *TC* and *SST*.

Table 4.5 displays the average task execution times for the above benchmark configurations on *Goethe*.

SWSS	***SWSD***	***UTS***$_E$	***BC***$_E$	***NQueens***
17 ms	100 μs	360 ns	120 ms	115 ns

Table 4.5: Average task execution times on *Goethe*

4.5.4.1 Running Times of Failure-Free Runs

Running Times of Synthetic Benchmarks Figure 4.16 depicts the running times of *SWSS* (left) and *SWSD* (right), reporting averages over 5 runs. Note that the axes do not start from zero.

It can be observed that all overheads of *TC*/*SST* over non-resilient GLB_{coop} are below 1%, which is comparable to the overheads of work stealing (the latter corresponds to the difference between the *GLB* running times and $\widehat{T}(p)$).

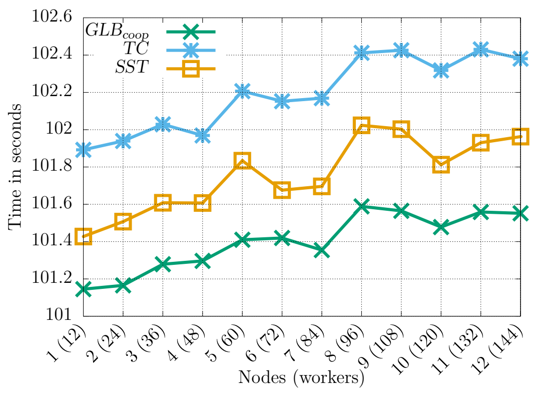
Note that in Section 4.5.3 we observed an overhead of below 6% for *TC*. The new value of less than 1% is more accurate because the deployed smooth weak scaling allows values such as *tasks per worker* to be constant, whereas in Section 4.5.3 with scaling of workers they fluctuated due to the benchmarks.

The *SST* overheads are consistently about half of the *TC* overheads. For *SWSS*, they are a maximum of 0.43 s (*SST*) vs. 0.86 s (*TC*). For *SWSD*, they are a maximum of 0.65 s (*SST*) vs. 1.10 s (*TC*). The curves run roughly in parallel. As the *GLB* overheads result from work stealing, this suggests that the resilience costs increase proportionally to the steal rate.

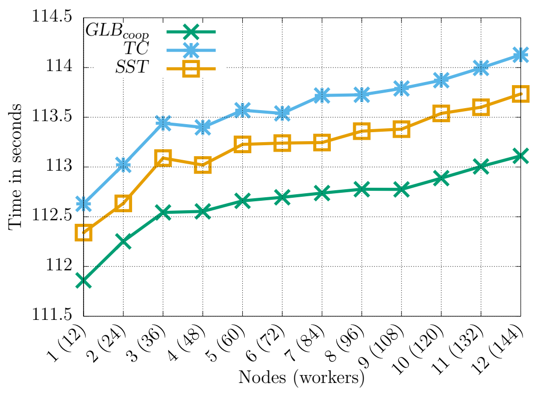
Number of Messages Figure 4.17 presents the number of messages sent for the same program runs as above. Again, the *SST* curve is clearly located beneath the *TC* one, indicating that part of the performance difference is owing to differences in the communication overheads. The *SST* curve is closer to the *GLB* one, however, suggesting that another part of the difference is owing to *SST*'s computation costs for history maintenance.

The concrete numbers in Figure 4.17 meet our expectations: while *GLB* issues two messages per steal, *SST* issues three, and *TC* issues seven (see Sections 4.3 and 4.4.3, respectively). This results in factors 1.5 and 3.5 for the respective message numbers.

UTS_E*, *BC_E*, and *NQueens Figure 4.18 depicts the running times on *Kassel* and *Goethe*. Unlike before, this figure employs strong scaling to convey an impression of the magnitudes. The figure presents two curves: a falling one describing the running times, and a rising one describing the number of processed nodes (of the respective benchmarks) per second. All *TC* and *SST* curves are close to the *GLB* ones, indicating again that the resilience overheads are low.

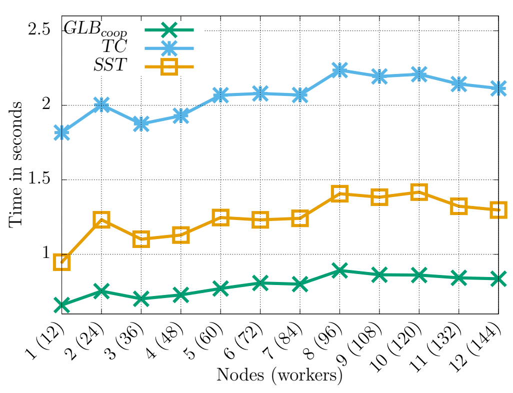


(a) *SWSS*

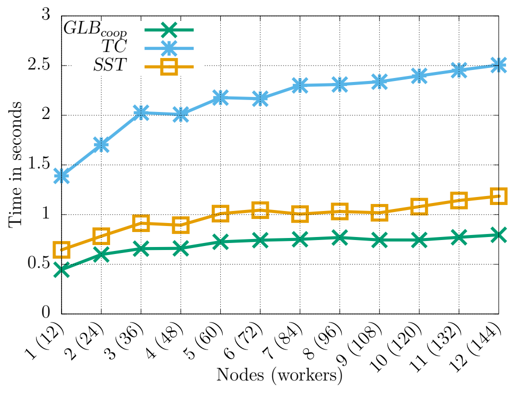


(b) *SWSD*

Figure 4.16: Weak scaling on *Kassel*: Failure-free running times of *GLB$_{coop}$*, *TC*, and *SST*

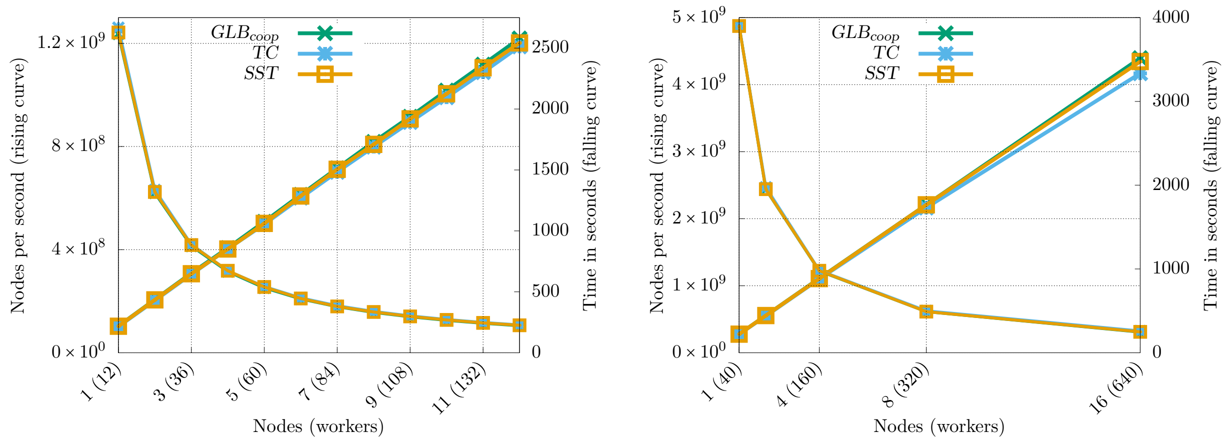


(a) *SWSS*

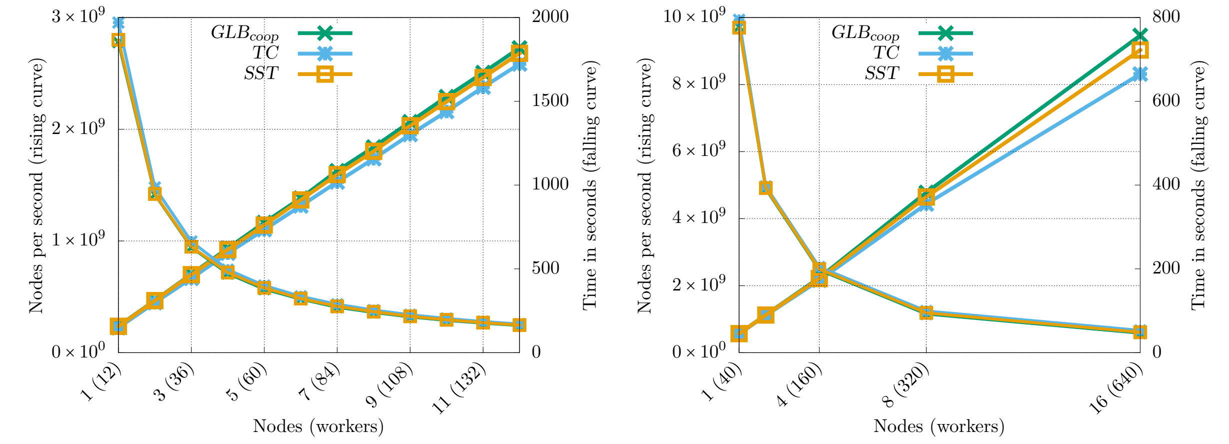


(b) *SWSD*

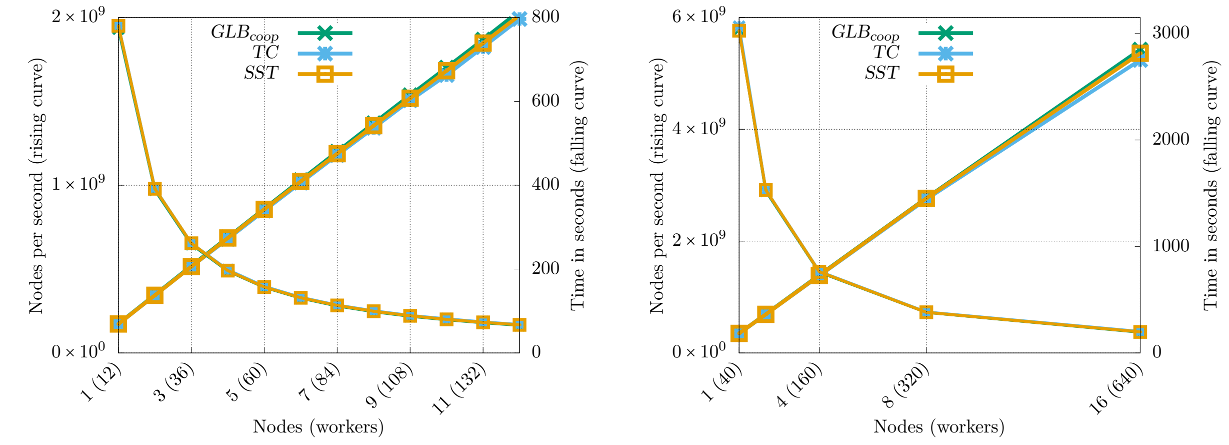
Figure 4.17: Weak scaling on *Kassel*: Messages per worker per second of *GLB$_{coop}$*, *TC*, and *SST*



(a) *UTS$_E$*: Performance on *Kassel* (left) and on *Goethe* (right)



(b) *BC$_E$*: Performance on *Kassel* (left) and on *Goethe* (right)



(c) *NQueens*: Performance on *Kassel* (left) and on *Goethe* (right)

Figure 4.18: Strong scaling: Performance of *GLB$_{coop}$*, *TC*, and *SST*

Histograms Figure 4.19 depicts histograms of the processor time usage of the workers for *SWSD* and *SWSS*. For each particular time, the histograms represent the share of workers in the following states:

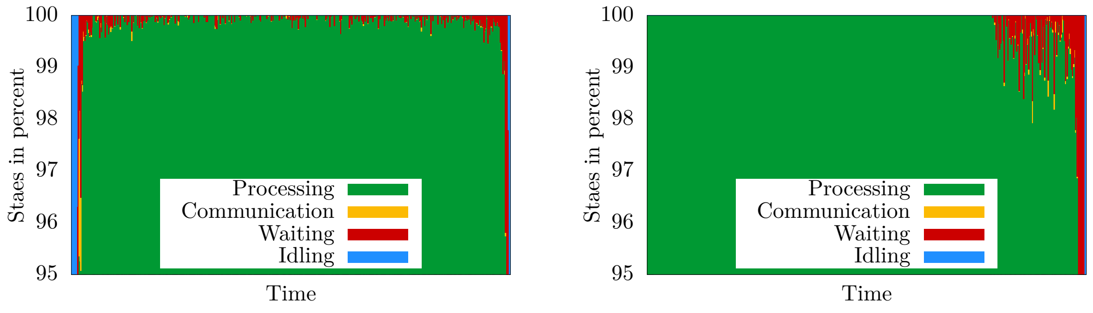
- *processing*: worker processes tasks (green),
- *communication*: worker is involved in stealing or checkpoint writing (orange),
- *waiting*: worker is waiting for a response to a steal request (red), and
- *idling*: worker is initially or finally inactive (blue).

Note that work stealing results in communication and waiting states. The histograms refer to a run with $\widehat{T}(144) = 20s$ on *Kassel*. A small $\widehat{T}(p)$ value was used to pronounce the start and end phases in the figures. Similarly, the histograms were cut at 95% to save space. The omitted parts are almost exclusively green.

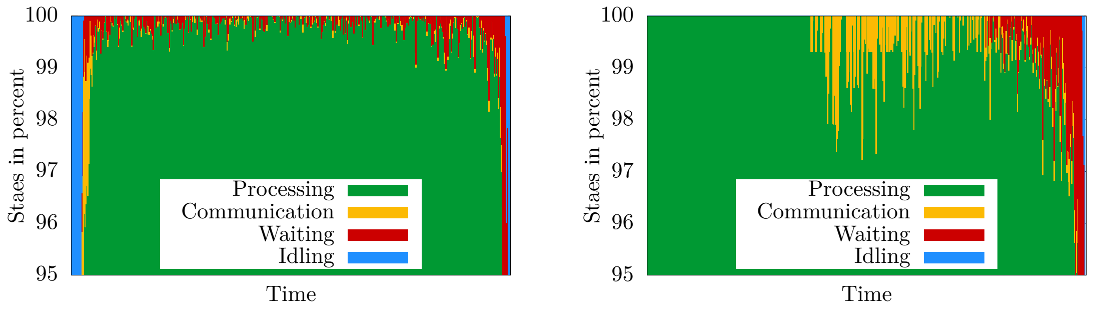
Interpreting the histograms, *SWSS* (Figure 4.19, right side) starts all workers in the processing state. As the task distribution is even, they remain in this state for most of the time. Work stealing arises only in the end phase, owing to the variations in task durations. The number of steals increases until more and more workers enter idle state.

Although all *SWSS* histograms share this same pattern, the work stealing states take more room in *TC* and *SST*. The reason can be seen in their higher numbers of messages, as discussed before. The *TC* histogram exhibits several communication spikes in the middle, which are owing to regular checkpoint writing. The reason that this is not one spike is because we have offset the checkpoints slightly to reduce competition in accessing the resilient store.

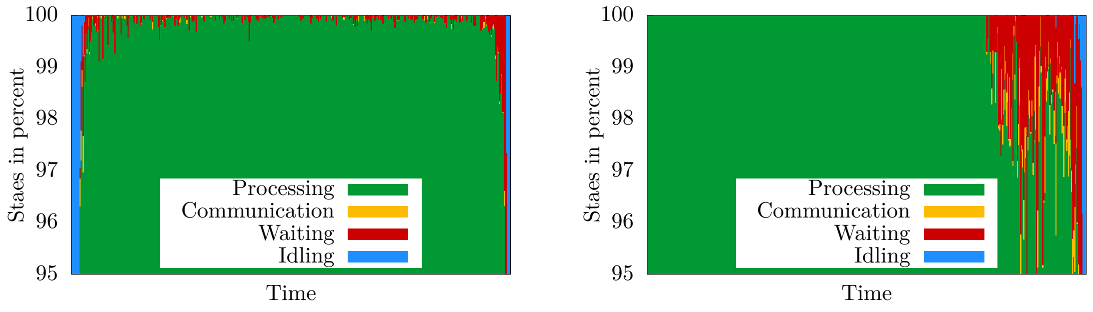
SWSD (Figure 4.19, left side) exhibits a start phase in which the initial task is transformed into initial task bags for all workers. A noticeable number of steals occurs in the main phase, as the benchmark is irregular. Again, the work stealing takes more room in *TC* and *SST*. In accordance to *SWSS*, the difference is more distinct for *TC*.



(a) *GLB*$_{coop}$



(b) *TC*



(c) *SST*

Figure 4.19: Histograms of processor time usage of *SWSD* (left) and *SWSS* (right) runs with $\hat{T}(144) = 20s$ on *Kassel*

4.5.4.2 Recovery Overheads

The recovery time consists of the time to perform the actual recovery, the time for reprocessing the lost tasks, and the surplus time owing to the loss of computing power in the subsequent computation. Table 4.6 summarizes these times for a concrete, but typical example of a *SWSD* run. The table reports averages of 100 runs, into which we injected 1 or 2 failures, such that random workers failed at random times. As indicated in the table, the major differences between *TC* and *SST* are in the time required for reprocessing lost tasks, which is much higher for *SST* than for *TC*, and in the overhead of failure-free runs, which is higher for *TC*.

	1 failure		**2 failures**	
	TC	***SST***	***TC***	***SST***
Failure-free overhead	2.96 s	1.14 s	2.96 s	1.14 s
Actual recovery	0.39 s	0.35 s	0.67 s	0.70 s
Reprocessing	0.05 s	1.36 s	0.08 s	2.84 s
Lost computation	1.51 s	1.49 s	3.11 s	3.28 s
Total running time (incl. above costs)	119.95 s	119.38 s	121.81 s	123.84 s

Table 4.6: Recovery times of *SWSD* on *Goethe* with 40 workers, injecting 1 or 2 failures

4.5.4.3 Summary

In this section, we have evaluated the performance of *TC* and *SST*. The results show for both *TC* and *SST* low overheads in failure-free runs, typically below 1% for smooth weak scaling. However, *SST* showed slightly smaller resilience overheads in failure-free runs, whereas *TC* recovery is faster.

4.6 Estimation of Running Times[2]

In this section, we derive formulas for the overall running times of *TC* and *SST*, including failure handling. We consider the case that $x \ll p$ failures occur at independent and identically distributed times. The formulas depend on Mean Time Between Failures (MTBF), number of workers, and steal rate. We use the derived formulas in Section 4.7 to predict running times in larger-scale settings than in our experiment and to determine

[2]This section is chiefly the work of the co-authors of publications [P12, P15]

general conditions under which either TC or SST is superior. We use the following notation:

- p: number of workers,
- s: steal rate (average number of steals per worker and second),
- r: regular checkpoint rate (number of regular checkpoints per worker and second),
- $T^{NO}(p)$: running time of non-resilient work stealing algorithm on a given program call (a program call is an invocation with fixed inputs),
- $T_x^{alg}(p)$: expected running time of $alg = \{SST \,|\, TC\}$ when x failures are encountered during this program call, and
- $MTBF$: Mean Time Between Failures (system-wide).

4.6.1 Running Times of TC

Backed by $x \ll p$, we assume that the x failures affect different workers. At each of them, the respective failure strikes with equal probability at any particular time during the program's execution. This is for the reason that hardware components live much longer than what the program run takes, and thus their susceptibility to failure is about constant. From general properties of uniform distributions, the expected time for the occurrence of a worker's failure is at half of its running time (and thus the overall computing power p available for our computation is reduced to $p-(1/2)$). Similarly, summing up the x uniformly distributed times, which are independent from the above assumption, implies that an expected $x/2$ of the overall computing power is lost (the expected value of a sum equals the sum of the expected values). The corresponding share of work must be taken over by the other workers, leading to a proportional increase in running time. Similarly, the other workers must repeat an expected half of the work from the last checkpoint interval of each failed worker. As the average interval length is $b = 1/(s+r)$, we obtain

$$T_x^{TC}(p) = \frac{p}{p-(x/2)} T_0^{TC}(p) + \sum_{i=1}^{x} \frac{b}{2(p-i)} + xR^{TC},$$

where R^{TC} is the cost of the actual recovery procedure. R^{TC} is essentially independent of p, as can be easily observed from the algorithm description in Section 4.3. Furthermore, $T_0^{TC}(p)$ differs from $T^{NO}(p)$ by the checkpointing overhead. Most of this overhead increases proportionally to the steal rate and regular checkpointing rates, and thus

$$T_0^{TC}(p) = (1 + c_0 s + c_1 r)\, T^{NO}(p)$$

for some constants c_0 and c_1. With $c_2 = R^{TC}$, we obtain

$$T_x^{TC}(p) = \frac{p}{p - (x/2)}\, (1 + c_0 s + c_1 r)\, T^{NO}(p) + \sum_{i=1}^{x} \frac{b}{2(p-i)} + x c_2 \,. \tag{4.1}$$

4.6.2 Running Times of SST

Upon each failure, the failed worker's share of previous work is lost, which on average is $1/p$-th of the overall previous work [21]. Similarly, the worker's $1/p$-th share of future work must be taken over by the other workers, resulting in a proportional increase in running time. We obtain

$$T_x^{SST}(p) = \frac{p}{p-x}\, T_0^{SST}(p) + \sum_{i=1}^{x} R^{SST}(p-i)\,,$$

where $R^{SST}(p)$ denotes the overhead of the recovery procedure with p workers and is analyzed below. $T_0^{SST}(p)$ differs from $T^{NO}(p)$ by the costs to maintain and communicate the history information, as well as by the costs to report back the results of task bags. Similar to TC, most of these costs increase proportionally to the steal rate, and thus

$$T_0^{SST}(p) = (1 + c_3 s)\, T^{NO}(p)\,.$$

$R^{SST}(p)$ covers the overheads of the following actions:

a) scan all locally stored frames and their histories, searching for frames to/from the failed worker (at each worker),

b) participate in the system-wide history collection (at each worker per lost frame), and

c) compress the collected histories into a replay tree (at one worker per lost frame).

To estimate the costs of actions a) to c), we make some observations about the typical size of the steal tree parameters: number of vertices $n = \Theta(p)$ (as each worker processes one frame in steady state), node degree $d = \Theta(1)$ (as random steals spread evenly across the tree), tree height $h = \Theta(\log\ p)$, and history length $l = \Theta(dh) = \Theta(\log\ p)$. On this basis, step a) requires time $\Theta(ld) = \Theta(\log\ p)$; step b) collects a maximum of $O(pld) = O(p\ \log\ p)$ data, resulting in time $O(\log\ p)$ per worker; and step c) processes the collected data for an average time of O(log p) per worker. Therefore, we estimate $R^{SST}(p-i) \approx R^{SST}(p) = c_4 \log(p)$, and obtain

$$T_x^{SST}(p) = \frac{p}{p-x}(1+c_3 s)\,T^{NO}(p) + x c_4 \log p. \tag{4.2}$$

4.6.3 Estimation of Constants

We experimentally determined approximations for the constants c_0 to c_4, based on the single-failure cases of formulas 4.1 and 4.2. They are displayed in Table 4.7. Except for c_4, the values were directly measured, averaging over 25 runs of the *SWSD* benchmark on 200 workers. For c_4, we first calculated $R^{SST}(p)$ for several p, as the difference between $T_0^{SST}(p)$ and $T_1^{SST}(p)$ in 100 runs of the *SWSD* benchmark on $20, 40, \ldots, 240$ workers. Thereafter, we approximated c_4 through a regression analysis using the least squares method. The fitted $R^{SST}(p)$ function has an R^2 value of 0.944898.

Constants	c_0	c_1	c_2	c_3	c_4
Value	38 ms	41 ms	558 ms	24 ms	219 ms

Table 4.7: Experimentally determined values of constants

4.6.4 Experimental Validation

By inserting the above constants from single-failure runs into formulas 4.1 and 4.2, we obtain predictions for multi-failure cases. To confirm these, we injected up to 12 failures into *SWSD* runs with $\widehat{T}(40) = 100s$ on *Goethe*. Again, we let random workers fail at random times. Figure 4.20 depicts our results, reporting averages of 100 runs, alongside the running time predictions. It can be observed that predictions and measurements are very close.

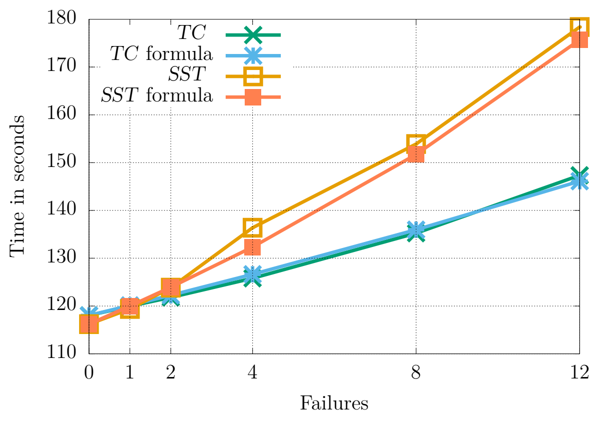


Figure 4.20: Total running times for failures for *SWSD* with $\hat{T}(40) = 100s$ on *Goethe*

4.7 Prognosis

Based on the formulas from Section 4.6, in this section we predict running times in larger-scale settings than in our experiments. First, we predict the execution times of single long-running applications under failures. Second, we perform simulations to determine the makespans of job sets, in which either all jobs are made resilient via *TC* or *SST* (protected jobs), or none of the jobs uses any resilience scheme (unprotected jobs).

4.7.1 Long-Running Applications[3]

Using our notation from Section 4.6, we consider a program call that likely experiences exactly one failure; that is, a program call with running time

$$MTBF = T^{NO}(p) \approx T_0^{TC}(p) \approx T_0^{SST}(p)\,.$$

Below, we derive conditions under which, say, *TC* outperforms *SST* for this call. The same conditions also apply to program calls with longer running times. To see this, consider the program executions as being composed of sections of length MTBF. If *TC* is superior in a single MTBF section, then *TC* is superior in every MTBF section, provided that the conditions are stable. The conditions involve $MTBF$, s, r, and p. The first three are normally stable (or quite stable) inside an application, and p only declines slightly for a reasonably low number of failures. We determine the conditions by solving the inequality

$$T_1^{TC}(p) < T_1^{SST}(p)\ \ ,$$

substituting equations 4.1 and 4.2 from Section 4.6 and setting $T^{NO}(p) = MTBF$. We obtain two solutions:

- $p > k_2/k_1$ and
  $MTBF < (p-0.5)\dfrac{(p-1)(c_4\log(p)-c_2)+(b/2)}{k_1p^2-k_2p}$, and
- $p < k_2/k_1$ and
  $MTBF > (p-0.5)\dfrac{(p-1)(c_4\log(p)-c_2)+(b/2)}{k_1p^2-k_2p}$,

where $k_0 = 1 + c_0s + c_1r + c_3s - 1$, and $k_1 = 1 + c_0s + c_1r - 0.5c_3s - 0.5$.

[3]This section is chiefly the work of the co-authors of publications [P12, P15]

Under these conditions, TC outperforms SST; otherwise, SST is superior. By inserting our estimates for c_0 to c_4 from Table 4.7, we obtain the exemplary values in Table 4.8, which are break-even points between TC and SST superiority. The table only refers to the first solution; the second one translates into $p < 28$. Note that the table displays the worker MTBF for clarity, which is calculated as $p \cdot MTBF$ [34].

We conclude that SST is usually superior, but TC takes over for an order of millions of workers.

Workers (p)	s	r	Worker MTBF
1,000	0.28	0.033	< 13.9 hours
100,000	0.28	0.033	< 122.9 days
1,000,000	0.28	0.033	< 4.2 years
1,000	0.05	0.033	< 7.1 days
100,000	0.05	0.033	< 3.0 years
1,000,000	0.05	0.033	< 17.7 years

Table 4.8: Scenarios in which TC outperforms SST

4.7.2 Sets of Jobs

When scheduling job sets on a supercomputer, one often strives for a low overall completion time, known as *makespan*. We considered sets of independent parallel jobs that were known a priori and simulated their execution in faulty environments. In each simulation run, we protected all jobs in the same manner (all unprotected, all TC, or all SST).

We adapted the job scheduling simulator from [172, 173], which was originally designed for studying silent errors. The simulator starts the jobs in priority order, we used random priorities. We randomly injected fail-stop failures into our simulation runs. For the TC/SST jobs, we simulated the running time increases with the formulas from Section 4.6. For unprotected jobs, we aborted the job and re-queued it. Note that we continued TC/SST jobs on less than p workers but restarted unprotected jobs on p workers. Two job sets on different supercomputers were considered:

- **Mira:** This computer (ranked 22 by top500.org in November 2019) has a total of $49,152$ nodes [174]. We extracted 30 job sets, namely one per day during June 2019, from the official published log data[4]. Each job set holds $66 \ldots 277$ jobs with running times of $37 \ldots 86$ s, and $p = 512 \ldots 49,152$.

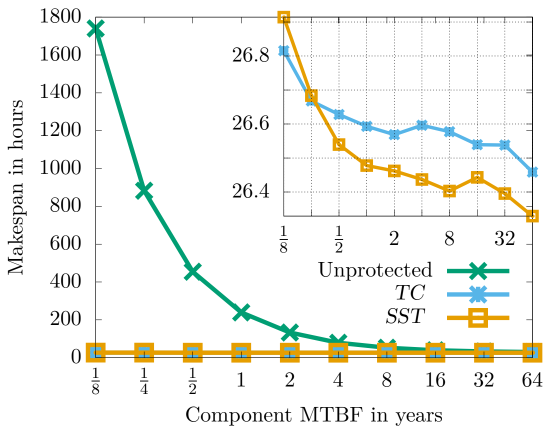
[4]This data was generated from resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357 [175].

- **Exa:** This hypothetical exascale computer has 1 million nodes. We generated 30 job sets, each with approximately 1250 jobs, with running times of $50 \ldots 10,000$ s (for a total of $p \cdot 24$ h) and $p = 500 \ldots 500,000$.

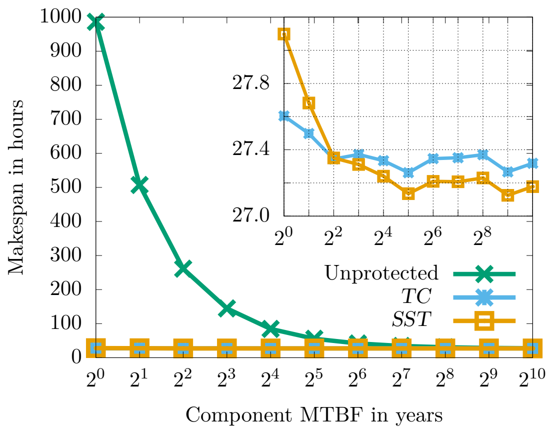
Figure 4.21 depicts the simulated makespans, averaged over all respective job sets and 1,000 runs of each. Both figures indicate a clear difference between protected and unprotected jobs. For example, in the Mira experiment with worker MTBF 0.125 years, *TC* reduces the makespan *by* as much as 98.46%, and *SST* reduces the makespan by 98.45%. The effect is smaller for a large MTBF, but at worker MTBF 64 years, the reduction is still 12.01% for *TC* and 12.44% for *SST*. Similarly, in the Exa experiment with worker MTBF 1 year, *TC* reduces the makespan by 97.20%, and *SST* reduces the makespan by 97.15%.

The difference between *TC* and *SST* is rather small. As detailed in the upper right corners, *SST* is slightly better for *worker MTBF* > 0.25 years in Mira and for > 4 years in Exa.

Overall, in both simulations *SST* caused slightly less overhead than *TC*, except for a low worker MTBF. This outcome agrees with that of Section 4.7.1. Our most striking result was a huge difference between protected and unprotected jobs, indicating that job protection by a task-level resilience method pays off.



(a) Mira Simulation



(b) Exa Simulation

Figure 4.21: Makespan simulations of unprotected jobs and jobs protected with *TC* and *SST*

4.8 Related Work

Although substantial research on fault tolerance has been conducted in recent years [33, 34, 72, 176], resilient programs are not yet state of the art. In the already cited recent survey among participants of the US Exascale Computing Project, only 2% of the respondents reported on current fault-tolerant applications, whereas 67% were planning to add this to their applications [38].

Regarding fail-stop failures, checkpoint/restart is the prevailing approach [71, 177, 178]. This method is available in a traditional variant, which writes data to a shared file system, and newer variants such as uncoordinated, in-memory, and multi-level checkpointing. The traditional variant is realized by system-level libraries such as BLCR [74] and DMTCP [73], and the newer ones by application-level libraries such as FTI [76] and SCR [75]. Application-level libraries provide users with control over aspects such as data selection [179]. All variants restart the application after failures, delays may be avoided by allocating spare nodes at job submission [180].

Other application-level approaches include naturally fault-tolerant algorithms [81] and algorithm-based fault tolerance (ABFT) [82, 83, 84, 85]. Checkpoint/restart and ABFT can be combined, such that different program sections are protected differently [181]. Chung *et al.* defined transaction-based containment domains, which encapsulate failures and recoveries in hierarchically arranged program segments [182]. Some program-level approaches rely on resilient arrays, e.g., the resilience scheme of NWChem [78]. These approaches require failure notification, as offered by programming systems such as ULFM for MPI [88, 183] and Resilient X10 [89, 184]. Other resilience support of programming systems includes in-memory checkpointing [95] and replication [86]. Fault tolerance for programming systems is ongoing research-oriented work in progress, e.g., to further increase the efficiency and programming productivity of Resilient X10 [185, 186]

Previous task-level resilience techniques were mostly aimed at static tasks, e.g., in MapReduce [187], hierarchical master/worker patterns [188], and the A* algorithm [189]. A well-known example is the lineage technique of Spark [112].

Resilience for dynamic tasks has received rather little attention to date, and the research conducted is spread across different topics. To note some examples, Kurt *et al.* [190] consider soft errors that are discovered late during the execution of a task graph and minimize the number of task re-executions. Cao *et al.* [191] detect silent data corruptions in task graphs and reduce task re-executions with the help of checkpointing. Subasi *et al.* [192] handle silent errors with a combination of task-/system-level checkpointing and message logging. Ma and Krishnamoorthy [94] consider fail-stop failures for tasks with side effects and suggest a technique to avoid updating the same data twice.

Prior to SST_{NFJ}, the Cilk-NOW system used supervision for dynamic tasks but did not integrate healthy subtasks [92]. Later, Satin improved on it by integrating finished subtasks and aborting the others [93]. Moreover, checkpointing was added to Satin [193]. In contrast to that work, SST_{NFJ} integrates *all* subtasks that are available on healthy nodes [21].

Among the precursors of *TC* is an X10 scheme ($X10\text{-}FT_{GLB}$) that explicitly saves data in other workers' main memories instead of using a resilient store [20]. We have compared TC_{GLB} and $X10\text{-}FT_{GLB}$ in Section 4.3.6. In the context of $X10\text{-}FT_{GLB}$, a precursor of *IncTC* was sketched, alongside a rudimentary implementation [194]. Recently, a *TC*-like algorithm for NFJ has been sketched [195, 196].

The official IBM repository for *APGAS* and X10 [115] contains a fault-tolerant *UTS* variant for *APGAS* and X10, respectively. Like *TC*, they utilize the `IMap` or a X10 resilient store, respectively, but introduce a restricted stealing scheme in which the lifeline graph is one-dimensional. We are not aware of any other previous work on resilient *APGAS* applications. Regarding resilient X10 applications, *TC*'s predecessors are particularly noteworthy [20, 197].

There are several resilient data structures around. For instance, Terracotta [198] and Infinispan [199] provide similar features as the Hazelcast `IMap`.

The `IMap` of Hazelcast does not appear to be widely used in research or at least the usage is not documented. Instead, Hazelcast has been mostly utilized to connect JVMs and distribute storage across them [200]. Kathiravelu *et al.* [201] introduced a framework for detecting and deleting duplicates in the data analytics context, which utilizes Hazelcast's MapReduce support. The distributed simulator Cloud2Sim [202, 203] deploys Hazelcast for simulating cloud and MapReduce algorithms.

Theoretical analyses are common in fault tolerance research. For example, these have been used to determine optimal checkpoint intervals [171, 34], and to evaluate combinations of checkpoint/restart with process replication [204]. Similar to our work, Subasi *et al.* [192] derive formulas for the overall running time of their combined scheme and use these for a comparison with checkpoint/restart. Makespan analyses of job sets have to date focused on resilient scheduling heuristics for parallel jobs [172], checkpointing strategies for optimized system I/O [205], and the impact of malleability on job scheduling [102].

4.9 Conclusions

In this chapter, we have contributed to the open research questions regarding fault tolerance. We have started by describing our new fault tolerance task-level technique *TC*. Each worker independently writes checkpoints of its task pool contents and its current result in a resilient in-memory store. Checkpoints are updated in the event of stealing. Upon failures, the failed worker's tasks are adopted by co-workers, such that the program computes the same result as in non-failure executions. We have described *TC* in a general way such that it can be applied to a spectrum of task models. Thus, we have shown that checkpointing can be provided at task-level, as asked in an open research question.

Moreover, we have described the adaptation and implementation of *TC* to *GLB*. The resulting TC_{GLB} has several advantages over the related predecessor $X10\text{-}FT_{GLB}$. In particular, TC_{GLB} is less complex, easier to maintain, configurable, robust against multiple simultaneous failures, and available for a mainstream language. TC_{GLB} requires negligible programming effort beyond that of non-resilient *GLB*, as asked in an open research question. Experiments showed a failure-free overhead below 1% and a recovery overhead after failures below 0.5 seconds, both for smooth weak scaling. In a comparison to DMTCP, *TC* has shown significantly lower failure-free running times and a significantly lower recovery overhead than DMTCP.

In addition, we have introduced three variants – *IncTC*, *LogTC* and *SST* – and formulated them as well as *TC* in a general way. *IncTC* periodically writes checkpoints in a resilient store, like *TC*, but reduces the checkpoint volume in both number of checkpoint writings and checkpoint size. For that, *IncTC* selectively saves stable tasks only, and writes the checkpoints incrementally. *LogTC* combines an existing supervision and steal tracking approach for nested fork-join programs [21] with *TC*'s checkpointing scheme but does not update checkpoints in the event of stealing. *SST* transfers the same supervision and steal tracking approach [21] from the nested fork-join context to our context, but without writing checkpoints. This way, we contributed to the open research question whether task-level resilience techniques are specific to a particular task model. Experiments showed no clear winner between our four techniques.

Furthermore, we have contributed to the open research question of how much our techniques could increase the effective throughput of supercomputers. We have started with a derivation of formulas that predict the total running times of applications protected by *TC* or *SST* but affected by failures. Thereafter, we have determined conditions under which *TC*/*SST* are superior in single application runs. Finally, we have simulated the execution of job sets on a real and on a hypothetical supercomputer and evaluated the makespans. Results consistently support the same conclusions: program protection at the level of an AMT runtime system pays off. Moreover, the choice between *TC* and *SST* is secondary. We consistently observed *SST* as being superior in typical current settings, but *TC* takes over on large machines and for frequent failures.

Chapter 5

Resource Elasticity

Contents

5.1 Introduction

In this chapter, we present our contributions to the open research questions regarding resource elasticity that were outlined in Section 1.4.3. In Section 5.2, we start by presenting our novel ***T**ask-level **R**esource **E**lasticity* technique *TRE*, thus answering one of the open research questions. *TRE* allows applications to transparently adapt to the addition or release of multiple nodes. As its major characteristic, elasticity operations are performed concurrently to task processing, which makes *TRE* efficient.

Then, in Section 5.3, we outline the implementation of *TRE*, for which we extend the multi-worker GLB variant denoted by GLB_{multi} (described in Section 2.4). Since *TRE* involves only a few complex operations, it can be implemented in a straightforward way, thus addressing the open research question for simplicity.

Another research question asked whether resource elasticity can be provided by an AMT in a way that does not require additional programming effort. We contribute to it by keeping GLB_{multi}'s programming requirements unchanged.

Afterwards, in Section 5.4, we propose formulas that estimate the overhead-free running time of work stealing programs with a changing number of resources. With these formulas, we contribute to the open research question of how to predict running times.

In Section 5.5, we introduce a heuristic to determine the minimum, maximum and preferred number of nodes for a given job. In addition, we introduce a job scheduler's strategy to assign resources to malleable jobs. This way, we contribute to the open research question of how job schedulers can support malleability in a user-friendly way.

In Section 5.6, we determine the overhead for adding or releasing nodes. This overhead could not simply be measured, since it is impossible to run a program with a particular resource scenario without causing the overhead. Therefore, we use the previously derived formulas and subtract the calculated values from the measured running times. We observe that adding/releasing up to 64 nodes takes less than 0.5 seconds.

Then, in Section 5.7, we quantify the gain of deploying malleable jobs on supercomputers. For that, we simulate the execution of job sets that contain some percentage of malleable jobs. We use the elastic job scheduler strategy and the heuristic from Section 5.5. Results show that the overall completion time can be reduced by up to 20% if most jobs are malleable. Thus, we contribute to the open research question by how much the effective throughput of supercomputers increases from using our resource elasticity technique.

Finally, we conclude this chapter with related work and conclusions in Sections 5.8 and 5.9, respectively.

This chapter was adapted from publications [P13, P14].

5.2 Task-Level Resource Elasticity (TRE)

In the following, we describe the ***T**ask-level **R**esource **E**lasticity* technique *TRE*. Although the description refers to GLB_{multi}, the main ideas of the technique should be applicable to other work stealing-based cluster AMTs, as well. Also, we describe the realization of malleability, but the same technique could be used to handle resource change requests that arise from within the program. Finally, we assume, for simplicity, that there is always one place per node, but the technique could easily deal with multiple places. Please note that we treat *TRE* independently of the fault tolerance techniques from Chapter 4 and therefore always assume failure-free runs in the following.

As its major characteristics, the elasticity technique is performed concurrently to task processing, which makes it efficient. Place 0 (denoted by $\mathtt{P}_0$) takes some additional responsibilities, and therefore it cannot be released. This is reasonable since $\mathtt{P}_0$ is also responsible for global termination detection, as described in Section 2.4.

A request for resource changes from the job scheduler triggers a *resize* operation in the GLB_{multi} runtime system. Resize operations can *shrink* the application by releasing places or *expand* the application by adding places. An operation can refer to one or multiple places, but different operations are handled sequentially. Resize operations can be triggered at any time and programmers do not need to provide any explicit synchronization point.

In the following, we describe the shrink and expand operations, as well as implementation aspects. Added places are denoted by $\mathtt{P}_{\mathtt{add}}$, released ones by $\mathtt{P}_{\mathtt{rel}}$, and staying ones by $\mathtt{P}_{\mathtt{stay}}$.

5.2.1 Shrinking

Shrink operations are parameterized by a list of nodes, a list of place IDs, or just a number of places to be released. In the latter case, the places with the highest IDs will be released. First, $\mathtt{P}_0$ broadcasts a `shrink` message to *all* places, and waits for their acknowledgments. The `shrink` message is parameterized with a list of all $\mathtt{P}_{\mathtt{rel}}$.

On receiving the `shrink` message, each $\mathtt{P}_{\mathtt{rel}}$ stops sending new inter-place steal requests and sends an acknowledgment to $\mathtt{P}_0$. However, $\mathtt{P}_{\mathtt{rel}}$ continues to wait for pending responses, if any, and continues to respond to incoming steal requests.

In contrast, each $\mathtt{P}_{\mathtt{stay}}$ performs the following actions concurrently to task processing, but *not* to ongoing inter-place work stealing (in an activity):

- A local list of all alive places is updated. This list is used to determine random victims in work stealing, and to reject any incoming steal requests from $\texttt{P}_{\texttt{rel}}$. Such requests may occur when a $\texttt{P}_{\texttt{rel}}$ receives the `shrink` message late.
- The lifeline graph is recalculated to avoid a dissection of it [109], which would result in some places not receiving further tasks. For this, l, z as well as new lifeline buddies are determined.
- All recorded lifeline requests from all $\texttt{P}_{\texttt{rel}}$ are deleted. Thus, no more tasks will be sent to any $\texttt{P}_{\texttt{rel}}$ in the future.

Afterwards $\texttt{P}_{\texttt{stay}}$ sends an acknowledgment to $\texttt{P}_0$. When $\texttt{P}_0$ has received all acknowledgments from $\texttt{P}_{\texttt{rel}}$ and $\texttt{P}_{\texttt{stay}}$, it is guaranteed that system-wide there are no open inter-place steal requests related to any $\texttt{P}_{\texttt{rel}}$ and no more will be sent.

Nevertheless, the task queues of $\texttt{P}_{\texttt{rel}}$ may still contain tasks, and the $\texttt{P}_{\texttt{rel}}$ workers may still be processing tasks. Therefore, $\texttt{P}_0$ sends a `stop` message to all $\texttt{P}_{\texttt{rel}}$, and waits for their acknowledgments. On receiving the `stop` message, each $\texttt{P}_{\texttt{rel}}$ performs the following actions (in an activity):

- It stops task processing by forcing all workers to go inactive. This is done at the beginning of the next communication phase, i.e., the workers finish processing up to n tasks, and then go inactive.
- Afterwards, it takes all tasks and intermediate results out of the worker queues, the `intra-queue` and the `inter-queue`. This contradicts the original policy that each local queue can only be accessed by its owner but is reasonable since all workers are inactive and are never restarted. The tasks and intermediate results are merged into a new queue, called `loot-release`.
- A place receiving `loot-release` is determined randomly from $\texttt{P}_{\texttt{stay}}$, and `loot-release` is sent there, where it is handled like any other loot.

After performing these actions, each $\texttt{P}_{\texttt{rel}}$ sends an acknowledgment to $\texttt{P}_0$. When $\texttt{P}_0$ has received all acknowledgments, all $\texttt{P}_{\texttt{rel}}$ do not have tasks anymore, and they are no longer involved in inter-place steals. Thus, $\texttt{P}_0$ shuts down all $\texttt{P}_{\texttt{rel}}$ to free the corresponding nodes.

5.2.2 Expanding

Expand operations are parameterized by a list of nodes on which the new places are started by $\texttt{P}_0$. The places connect automatically to the running application, which is managed by *APGAS*. Nevertheless, the $\texttt{P}_{\texttt{add}}$ still need to be included logically in the running computation, which is accomplished as follows:

When all $\texttt{P}_{\texttt{add}}$ have been started and connected, static immutable data are initialized there, if necessary. For this, $\texttt{P}_0$ invokes an activity on each $\texttt{P}_{\texttt{add}}$. When all activities have finished, $\texttt{P}_0$ broadcasts an `expand` message to *all* places, including $\texttt{P}_{\texttt{add}}$.

On receiving the `expand` message, each place updates its local list of alive places. Moreover, it recalculates the lifeline graph. This is the same as in shrinking, however the consequences are different, namely all $\texttt{P}_{\texttt{stay}}$ that determine a $\texttt{P}_{\texttt{add}}$ as a new lifeline buddy immediately send tasks to it. This way, the $\texttt{P}_{\texttt{add}}$ receive tasks and start processing them.

5.3 Implementation

We implemented *TRE* by extending $\textit{GLB}_{\textit{multi}}$ and denote the result by $\textit{TRE}_{\textit{GLB}}$. As noted in Section 2.4, $\textit{GLB}_{\textit{multi}}$ workers run as *APGAS* activities within place-level `finish` blocks contained in a global `finish` block. The remote steals are also encapsulated in these blocks.

When performing resize operations, it is essential that the global termination detection remains intact, i.e., it must still ensure that all tasks have been processed when the global reduction is performed. For this, resize operations start with one activity on P_0, which is encapsulated in the place-level `finish` block of P_0. All corresponding messages, like `shrink` and `expand`, are also encapsulated in this block. Therefore, programs can terminate only when no resize operation is in progress.

For shrink operations, the task processing of the corresponding workers is stopped, which also ends the activities. As a result, the corresponding place-level `finish` blocks are marked as ended in the global `finish` block. Thus, the global termination detection notes that all $\texttt{P}_{\texttt{rel}}$ have ended.

For expand operations, new places get tasks sent by lifeline buddies. Such sending is encapsulated in the place-level `finish` block of the lifeline buddy. On receiving, a new place-level `finish` block is started, which is encapsulated in the global `finish` block via the place-level `finish` block of the lifeline buddy. Thus, the global termination detection takes new places into account, too.

5.4 Overhead-Free Running Times

As noted in Section 5.1, we need formulas to estimate the overhead-free running times of program executions with a changing number of places. Below we derive these formulas for the case of a single resize operation. The following notation refers to a given program invocation:

- P_i: initial number of places with which the execution is started,
- P_s: number of places that are released by a **s**hrink operation at T_{op},
- P_e: number of places that are added by an **e**xpand operation at T_{op},
- T_{op}: elapsed running time when resize operation is triggered,
- $\widehat{T}(p)$: running time of a rigid execution with p places, *not* accounting for load balancing or resize overheads,
- $L(p)$: overhead for load balancing as a percentage of $\widehat{T}(p)$. For simplicity, we assume that the overhead depends on p only. In practice, it is higher at the beginning and end of a computation than in steady state. In the formulas below, $L(p)$ includes either the beginning or end, and part of the steady state, which justifies the simplification.
- $T_{est}(P_i - P_s)$: estimated overhead-free running time when the program is initialized with P_i places and P_s places are released at T_{op},
- $T_{est}(P_i + P_e)$: estimated overhead-free running time when the program is initialized with P_i places and P_e places are added at T_{op}.

In the following derivation of $T_{est}(P_i - P_s)$, we denote by $work = p \cdot t$ the amount of useful computation that is accomplished within some time interval t by p places. Obviously, for any time t,

$$t = \hat{t} \cdot (1 + L(p))$$

where $\hat{t}$ is the time that one would need without load balancing, and t is the time that one needs with load balancing, to accomplish the same amount of work. Thus,

$$T_{est}(P_i - P_s) = T_{op} + \widehat{T}_{rest} \cdot (1 + L(P_i - P_s))$$

where $\widehat{T}_{rest}$ is the duration of the second computation phase, in which the work that has not yet been accomplished by T_{op} is performed, and factor $1 + L(P_i - P_s)$ accounts for the load balancing overhead in this phase.

To determine $\widehat{T}_{rest}$, we observe that the work of the first phase would have finished by time $\widehat{T}_{op} = T_{op}/(1 + L(p_i))$ if there was no load balancing. The second phase then uses $P_i - P_s$ processors and lasts time $\widehat{T}_{rest}$. Since it must accomplish the same work as the rigid (load balancing-free) P_i-processor execution in time $\widehat{T}(P_i) - \widehat{T}_{op}$, we obtain

$$\widehat{T}_{rest} = (\widehat{T}(P_i) - \widehat{T}_{op}) \cdot \frac{P_i}{P_i - P_s}$$

which gives us

$$T_{est}(P_i - P_s) = T_{op} + \frac{\left(\widehat{T}(P_i) - \dfrac{T_{op}}{1 + L(P_i)}\right) \cdot P_i \cdot (1 + L(P_i - P_s))}{P_i - P_s} \tag{5.1}$$

Analogously, the formula for expansion is

$$T_{est}(P_i + P_e) = T_{op} + \frac{\left(\widehat{T}(P_i) - \dfrac{T_{op}}{1 + L(P_i)}\right) \cdot P_i \cdot (1 + L(P_i + P_e))}{P_i + P_e} \tag{5.2}$$

5.5 Elastic Job Scheduler

Our simulation relies on an elastic job scheduler, which requests users to specify a minimum, preferred, and maximum number of nodes for their jobs. Section 5.5.1 introduces a heuristic to do so. Thereafter, Section 5.5.2 describes the scheduler's strategy to assign resources to malleable jobs.

5.5.1 A Heuristic for Malleable Job Parameters

While many jobs can be run with any number of nodes, only a certain range of nodes makes sense from a performance point of view. To determine this range, our heuristic builds on the well-known concept of program efficiency, which is defined as:

$$\textit{efficiency} = T(1)/(p \cdot T(p))$$

Efficiency is a measure for resource utilization. In our case, $T(1)$ denotes the running time of one place with 40 worker threads, and $T(p)$ denotes the running time of p places with 40 worker threads each.

On this basis, we intuitively choose thresholds, leading to the following heuristic:

- The *minimum* number of nodes is always 1, as all tasks may be mapped to the same place.
- The *preferred* number of nodes is defined as the largest p, for which the efficiency is ≥ 0.8.
- The *maximum* number of nodes is defined as the largest p, for which the efficiency is ≥ 0.5.

5.5.2 Elastic Job Scheduling Strategy

Job schedulers rely on a strategy to assign jobs to resources, and to decide when and with how many resources to shrink or grow the jobs. We adapted an existing strategy designed to improve overall system throughput [101]. The strategy is executed each time when a job ends or a new job is submitted. The following actions are performed in sequence as long as there are idle nodes:

- Jobs that run with less than the preferred number of nodes are expanded to their preferred number.
- If there are pending jobs waiting to be started, running jobs are shrunk to do so, if possible. However, running jobs are never shrunk to less than their preferred number of nodes. Pending jobs are started with any number of nodes.
- Jobs that run with less than the maximum number of places are expanded.

This strategy must be further parameterized by a priority order to determine which malleable job to consider next. The original paper [101] uses earliest submission time first for this purpose, but we extend this and compare the following six priority orders:

- Later job submission time (called `LaterSub`),
- Earlier job submission time (called `EarlierSub`),
- Later job completion time (called `LaterComp`),
- Earlier job completion time (called `EarlierComp`),

- Later job start time (called `LaterStart`),
- Earlier job start time (called `EarlierStart`).

5.6 Experiments

In this section, we quantify the costs for adding and releasing places on-the-fly.

5.6.1 Experimental Setting

Experiments were run on *Goethe* [131]. We started one place per node on up to 128 nodes. Each place maintains 40 workers. This results in a maximum of 5120 workers. Java was used as OpenJDK in version 14.0.2.

We used *SWSS* and *SWSD* – the synthetic benchmarks that were designed with the aim of supporting **S**mooth **W**eak **S**caling for **S**tatic/**D**ynamic tasks, see Section 2.5.8.2. The combination of these benchmarks with the formulas from Section 5.4 enables an accurate analysis. In both benchmarks, users specify a base computation time $\widehat{T}(p)$, and then a *rigid* execution takes time $T_{rigid}(p) = \widehat{T}(p) \cdot (1 + L)$ with L reflecting the overhead for load balancing as a percentage of $\widehat{T}(p)$, as in Section 5.4.

We configured *SWSS* as follows:

- base computation time $\widehat{T}(p) = 100s$,
- number of tasks per worker 6,000, and
- fluctuation range for the task durations 20%.

We configured *SWSD* as follows:

- base computation time $\widehat{T}(p) = 100s$,
- approximately 10,000,000 tasks per worker, and
- fluctuation range for the task durations 20%.

We executed the benchmarks in three ways:

- *Rigid:* no resize operation was triggered.

- *Shrinking:* after $\widehat{T}(p)/2 = 50s$ a shrink operation was triggered to release half of the places.

- *Expanding:* after $\widehat{T}(p)/2 = 50s$ an expand operation was triggered to double the places.

5.6.2 Cost Analysis

Table 5.1 reports our measured running times of *SWSS* (top) and *SWSD* (bottom) in columns *measured*. All times are in seconds, and the given numbers are averages of five runs. Within each row, all *SWSS* runs are configured with the same total number of tasks, and all *SWSD* runs are configured with the same perfect m-ary task tree. The table is divided into *rigid* runs on the left, *shrinking* runs in the middle, and *expanding* runs on the right.

For the rigid runs, column *costs* presents the overhead for dynamic load balancing in seconds, which is calculated as $T_{measured}(p) - \widehat{T}(p)$.

For the shrinking and expanding runs, columns *estimated* present the estimated overhead-free running times calculated with formulas 5.1 and 5.2, respectively. Columns *costs* present the incurred costs of the resize operations and are calculated as the difference of columns *measured* and *estimated*.

In addition, Figure 5.1 visualizes the costs of dynamic load balancing in rigid runs (left), the costs of shrink operations (middle), and the costs of expand operations (right). Note that all horizontal axes are log-scaled, and the scales of the vertical axes differ.

Dynamic Load Balancing Costs: Since *SWSS* distributes all tasks evenly at the beginning, it requires fewer work sharing and work stealing events to balance the load than *SWSD*. This is reflected by lower costs for load balancing. For both *SWSS* and *SWSD*, the costs for load balancing grow with the number of places, but only gently, and are expected to converge for a large number of places [206].

Shrink Costs: For *SWSS*, shrink operations releasing up to 64 places from a running program incur costs between $0.21s$ and $0.33s$, and for *SWSD* between $0.18s$ and $0.29s$. These costs are mainly caused by sending tasks and results from the released places to staying places.

Rigid			***Shrinking***				***Expanding***			
Plac.	**Meas.**	**Costs**	**Places**	**Meas.**	**Est.**	**Costs**	**Places**	**Meas.**	**Est.**	**Costs**
1	100.34	0.34					**1 + 1**	75.41	75.20	0.21
2	100.47	0.47	**2 − 1**	151.02	150.81	0.21	**2 + 2**	75.52	75.26	0.26
4	100.59	0.59	**4 − 2**	151.36	151.05	0.31	**4 + 4**	75.63	75.33	0.30
8	100.73	0.73	**8 − 4**	151.57	151.31	0.26	**8 + 8**	75.72	75.39	0.33
16	100.85	0.85	**16 − 8**	151.78	151.57	0.21	**16 + 16**	75.79	75.45	0.34
32	100.98	0.98	**32 − 16**	152.09	151.81	0.28	**32 + 32**	75.88	75.52	0.36
64	101.12	1.12	**64 − 32**	152.41	152.08	0.33	**64 + 64**	75.92	75.59	0.33
128	101.25	1.25	**128 − 64**	152.61	152.34	0.27				

(a) Weak Scaling of *SWSS* with $\hat{T}(p) = 100s$

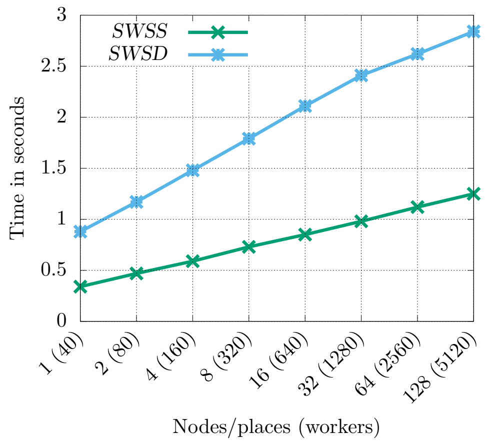
Rigid			***Shrinking***				***Expanding***			
Plac.	**Meas.**	**Costs**	**Places**	**Meas.**	**Est.**	**Costs**	**Places**	**Meas.**	**Est.**	**Costs**
1	100.88	0.88					**1 + 1**	75.75	75.51	0.24
2	101.17	1.17	**2 − 1**	152.22	152.03	0.19	**2 + 2**	75.91	75.65	0.26
4	101.48	1.48	**4 − 2**	152.85	152.61	0.24	**4 + 4**	76.09	75.81	0.28
8	101.79	1.79	**8 − 4**	153.39	153.21	0.18	**8 + 8**	76.18	75.96	0.22
16	102.11	2.11	**16 − 8**	154.02	153.82	0.20	**16 + 16**	76.34	76.11	0.23
32	102.41	2.41	**32 − 16**	154.65	154.41	0.24	**32 + 32**	76.49	76.23	0.26
64	102.62	2.62	**64 − 32**	155.19	154.90	0.29	**64 + 64**	76.60	76.33	0.27
128	102.84	2.84	**128 − 64**	155.55	155.31	0.24				

(b) Weak Scaling of *SWSD* with $\hat{T}(p) = 100s$

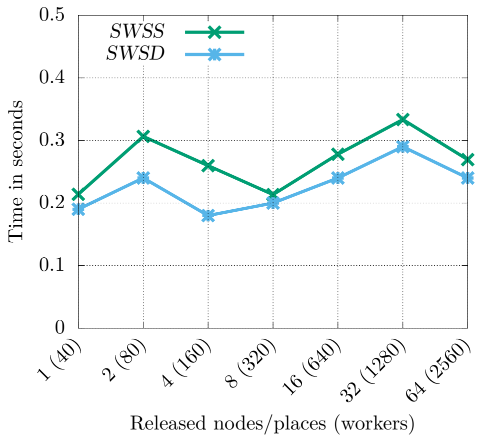
Table 5.1: Performance of *rigid*, *shrinking*, and *expanding* runs of *SWSS* and *SWSD* on *Goethe*. For each row, *SWSS* is configured with the same total number of tasks, and *SWSD* is configured with the same perfect m-ary task tree. All times are in seconds. *Left:* rigid execution. *Middle:* half of the places are released at $\widehat{T}(p)/2$ (shrinking). *Right:* number of places is doubled at $\widehat{T}(p)/2$ (expanding). Columns *Meas.* present running times from real executions, whereas columns *Est.* present estimated overhead-free running times calculated with formulas 5.1 and 5.2, respectively. Columns *Costs* present the incurred costs by dynamic load balancing (left), shrink operations (middle) and expand operations (right), and are determined using the appropriate formulas.

Expand Costs: For *SWSS*, expand operations adding up to 64 places to a running program incur costs between $0.21s$ and $0.36s$, and for *SWSD* between $0.22s$ and $0.28s$. These costs are mainly caused by the delay until added places receive their first tasks.

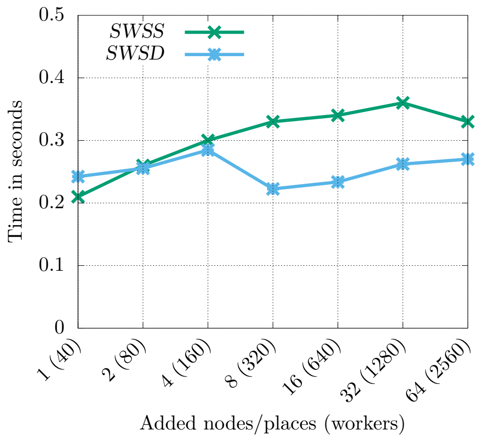
Overall, the costs for both resize operations are always less than $0.5s$ for both benchmarks. Differences between *SWSD* and *SWSS* are only in the costs for dynamic load balancing, for resize operations there are no significant differences. As the resize operations are performed in a distributed way and concurrently to task processing, the costs increase only gently with the number of places, resulting in good scalability.



(a) Load balancing costs



(b) Shrink costs



(c) Expand costs

Figure 5.1: *TRE*: Performance of rigid (top), shrinking (bottom left) and expanding (bottom right) runs of *SWSS* and *SWSD* on *Goethe*

5.7 Simulation

In this section, we analyze the impact of malleable workloads compared to rigid ones on the throughput on supercomputers. Similar to Section 4.7, we simulate the execution of sets of independent parallel jobs. The job sets are composed of real benchmarks executions. The synthetic benchmarks used in Section 5.6 would be inappropriate, as they are designed to provide a smooth weak scaling, which is unrealistic in practice.

5.7.1 Benchmarks and Job Sets

We run following benchmarks in three configurations, representing small, medium, and large runs, respectively:

- UTS_C: geometric tree shape, branching factor $b = 4$, initial seed $s = 19$, tree depth $d = 17, 18, 19$.
- *NQueens*: $N = 16, 17, 18$, threshold $t = 10, 11, 12$.
- BC_C: initial seed $s = 2$, number of graph nodes $N = 2^{18,19,20}$.
- *MatMul*: number of blocks $n = 800, 1000, 1400$, block size $m = 32$.

We again used *Goethe* [131], started one place per node, and deployed strong scaling on up to 128 nodes.

Figure 5.2 reports the measured performance of all runs expressed as efficiency, see Section 5.5.1. Note that the horizontal axis is log-scaled. Additionally, Figure 5.2 has gray horizontal lines representing the heuristic from Section 5.5.1. Table 5.2 shows the resulting malleable job configurations as well as $T(1)$ (one place with 40 workers).

The job sets were generated by randomly selecting benchmarks/configurations from Table 5.2 with a fixed-seed pseudo-random generator, until a desired *theoretical* makespan was reached. Thereby, theoretical makespan denotes the time in which all jobs could be *hypothetically* executed with their preferred number of nodes, ignoring submission times and scheduling.

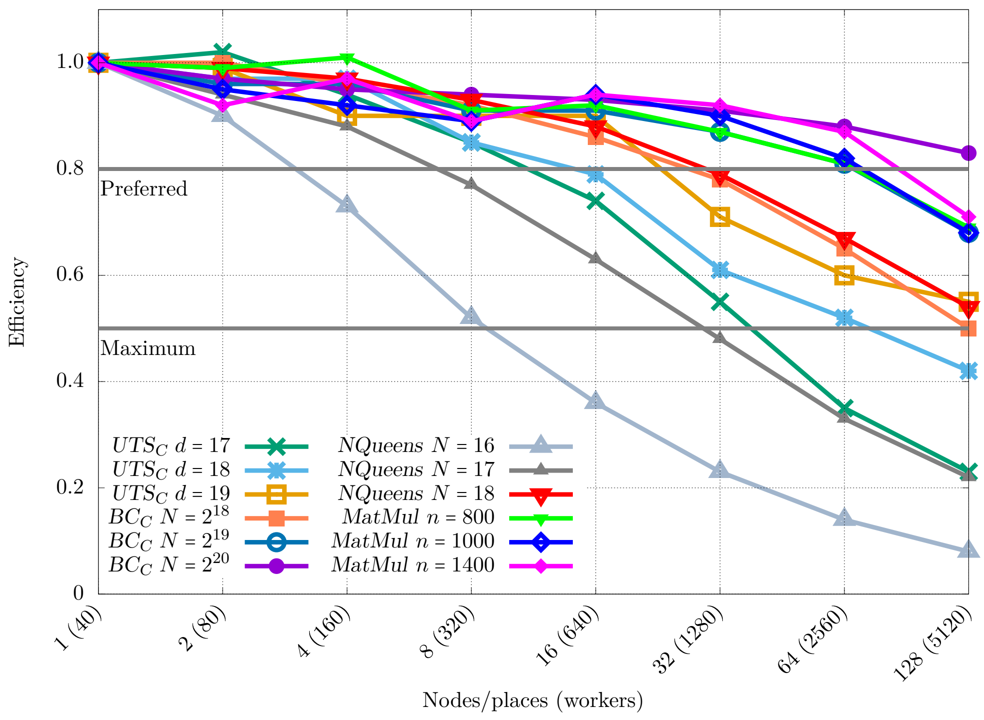


Figure 5.2: Strong scaling on *Goethe*: Program efficiencies

Benchmark	Preference	Maximum	T(1)
UTS_C $d = 17$	8	32	168.46 s
UTS_C $d = 18$	8	64	641.28 s
UTS_C $d = 19$	16	128	2554.75 s
BC_C $N = 2^{18}$	16	128	711.89 s
BC_C $N = 2^{19}$	64	128	3362.42 s
BC_C $N = 2^{20}$	128	128	15265.01 s
NQueens $N = 16$	2	8	50.19 s
NQueens $N = 17$	4	16	382.87 s
NQueens $N = 18$	16	128	3325.12 s
MatMul $n = 800$	64	128	696.44 s
MatMul $n = 1000$	64	128	1313.56 s
MatMul $n = 1400$	64	128	3588.04 s

Table 5.2: Malleable job configurations. Minimum number of places is always 1. $T(1)$ (one place with 40 workers) is in seconds.

5.7.2 Simulation Environments

We simulated the execution of the job sets in two hypothetical supercomputer settings:

- *Small*: 512 nodes, 5s average inter job arrival time, 591 total jobs on average.
- *Large:* 2048 nodes, 1s average inter job arrival time, 2362 total jobs on average.

We used 30 job sets for each setting, each with a theoretical makespan of 1h. Job submission times were determined pseudo-randomly. The total number of jobs on average (from above) have resulted from these settings.

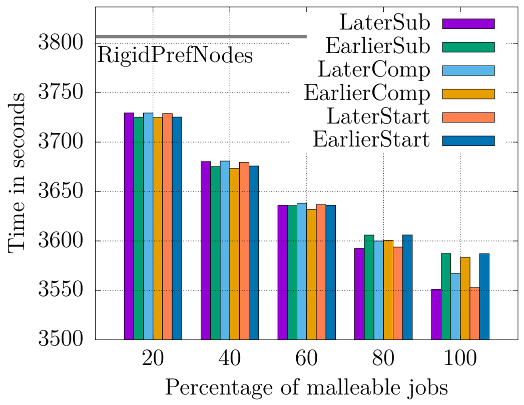
We deployed a self-written simulator that starts jobs sorted by submission time, coupled with back-filling adapted from Slurm [207]. For malleable jobs, the elastic job scheduling strategy from Section 5.5.2 was used.

Although our elasticity scheme can always be triggered, the simulator only allows one resize operation per job every five seconds. This can improve overall performance, as multiple resizing operations for a job can result in higher overhead than gain. When the simulator resizes jobs, the running time is adjusted accordingly based on the resize costs from Section 5.6.2 and the benchmark performance from Figure 5.2. We simulated all job sets 100 times and report averages.

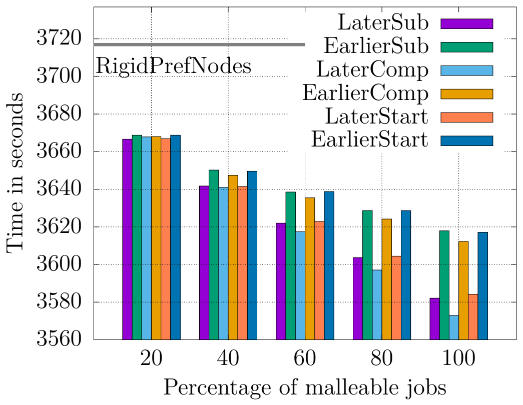
5.7.3 Impact of Malleable Workloads

Impact on Makespans As Figures 5.3a and 5.3b show, the makespan for both settings and all six priorities decreases with an increasing percentage of malleable jobs. For 100% malleable jobs, the makespan decreases by 6.71% (`LaterSub`) compared to `RigidPrefNodes` in the small setting, and by 3.87% (`LaterComp`) in the large setting. Even compared to `RigidMaxNodes`, which may get closer to practical usage scenarios, the makespan decreases by 20.24% (`LaterSub`) in the small setting, and by 19.61% (`LaterComp`) in the large setting.

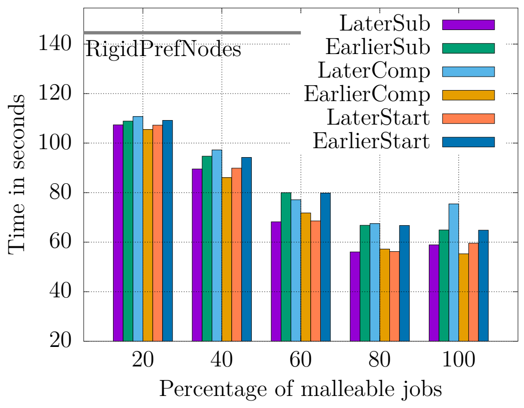
Impact on Job Waiting Times As Figure 5.3c shows, in the small setting, the average job waiting time decreases with an increasing percentage of malleable jobs. For 100% malleable jobs, `EarlierComp` has a decrease of 61.71% compared to `RigidPrefNodes`, and of 90.63% compared to `RigidMaxNodes`. As Figure 5.3d shows, the decrease is lower for the large setting than for the small setting. For 100% malleable jobs, `LaterStart` decreases the average job waiting time by 39.81% compared to `RigidPrefNodes` and by 80.21% compared to `RigidMaxNodes`.



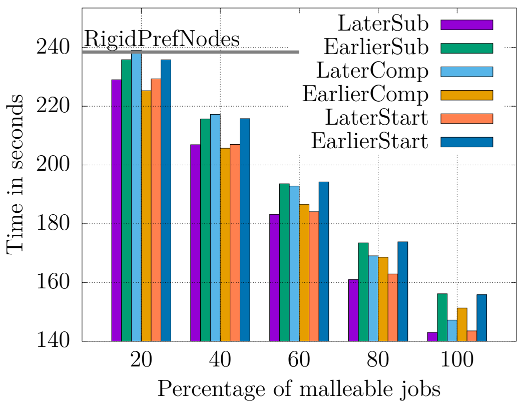
(a) Makespans: Small setting



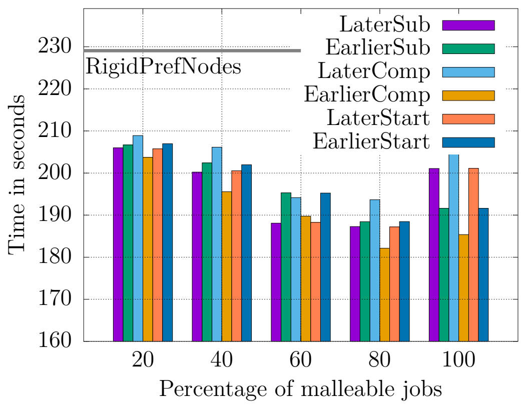
(b) Makespans: Large setting



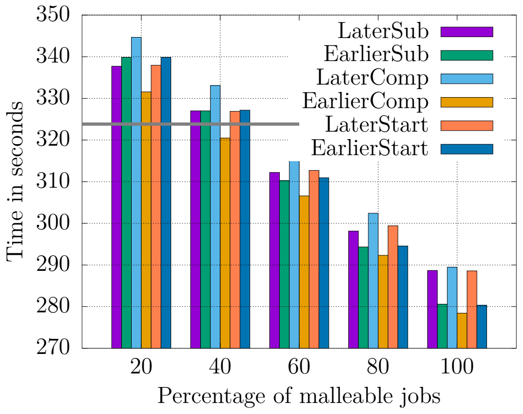
(c) Job waiting times: Small setting



(d) Job waiting times: Large setting



(e) Job response times: Small setting



(f) Job response times: Large setting

Figure 5.3: Simulations (makespan, job waiting times, job response times) of a varying number of elastic jobs. Note that the y-axes start at different times

Impact on Job Response Times As Figure 5.3e shows, in the small setting, the average response time decreases with an increasing percentage of malleable jobs, but stops at 80%, for which `EarlierComp` has a decrease of 20.49% compared to `RigidPrefNodes` and of 70.85% compared to `RigidMaxNodes`. As Figure 5.3f shows, in the large setting, the average response time decreases with an increasing percentage of malleable jobs. However, with 20% – 40% of malleable jobs, the average response time is slightly higher than for `RigidPrefNodes`. For 100% malleable jobs, `EarlierComp` decreases the average response time by 13.43% compared to `RigidPrefNodes`, and by 63.10% compared to `RigidMaxNodes`.

Overall, the simulation results show that adaptive resource management of supercomputers pays off. With an increasing percentage of malleable jobs, the improvement tends to increase. In contrast, the comparison between the six considered priorities showed no significant differences and no clear winner. Thus, the choice of priority is secondary, far more important is to have as many malleable jobs as possible.

5.8 Related Work

Malleable applications are not yet widespread in practice. The already cited recent survey among participants of the US Exascale Computing Project reports that 39% of their applications can change the number of processes via restarting from a checkpoint, and only 16% via dynamically changing the number of processes at runtime [38].

Elasticity via checkpoint/restart builds on the same mechanisms that are traditionally used for handling fail-stop failures [34]. This elasticity approach is supported, for instance, by the Charm++ runtime system [96], the SCR library [75], and the MPI extension PCM [105] as well as the MPI implementation AMPI [106]. Conventionally, checkpoints are written on-disc, but if written in-memory, the overhead of resizing can be reduced [107]. In contrast, our elasticity scheme neither writes checkpoints nor requires restarts.

Elasticity via dynamically changing the number of processes may significantly reduce the overhead [75]. However, it is harder to realize, particularly with MPI, since programmers must manually relocate computations and redistribute data, which may considerably increase the development effort. Moreover, this approach requires support from the programming system like notification of resource changes. Such support is still limited in standard MPI, although several extensions have been proposed for this purpose, e.g., [88, 208].

Most adaptive algorithms focus on iterative [101] or master/worker approaches [209] because they provide clear synchronization points that are well suited for adapting to resource changes. In contrast, our resource elasticity scheme does not require explicit synchronization points, and it does not increase the development effort.

Similar to our work, Bungart *et al.* [109] proposed a elasticity protocol for *X10-FT*$_{GLB}$ to enable the addition of processes, but they do not handle shrinking and consider single-threaded processes only. In contrast to us, they combine their approach with fault tolerance. Kehrer *et al.* [210] proposed elasticity control for task-based parallel tree search applications in the context of cost-based cloud environments. To the best of our knowledge, there are no previous elasticity schemes for dynamically adding and releasing workers in AMT cluster environments.

Scheduling strategies for malleable jobs on supercomputers have been studied both with the goal to improve the global throughput as well as with the goal to decrease global energy consumption, e.g. [100, 101, 102, 103]. This research has not yet made its way into the job schedulers in daily usage, which provide only rudimentary support for malleable jobs and elastic job scheduling. In research works, corresponding extensions have been proposed for Slurm [100, 101] and Torque [102].

5.9 Conclusions

In this chapter, we have contributed to the open research questions regarding resource elasticity. We have started by describing our new task-level resource elasticity technique *TRE*, which enables the addition and release of multiple nodes on-the-fly. *TRE* does not rely on explicit synchronization points, additional programming effort, or human input.

We have implemented *TRE* by extending *GLB*$_{multi}$ in a straightforward way, demonstrating that *TRE* is rather simple. Moreover, we have derived formulas that predict running times for work stealing programs that change their number of resources at runtime, addressing the corresponding open research question.

By combining these formulas with experiments, we have shown that costs for releasing/adding up to 64 nodes are below 0.5 seconds and that both adding and releasing nodes scales well. Thus, we have shown that efficient task-level resource elasticity is possible, as asked in an open research question.

Based on the previous results, we have simulated the execution of sets of malleable jobs. For the simulated job scheduling, we have introduced a strategy to assign resources to malleable jobs as well as a heuristic to help users to determine the minimum, maximum and preferred number of nodes for a given job. Results include a reduction of the overall completion time by up to 20% compared to rigid jobs. This way, we have contributed to the open research questions of how malleability can be used in a user-friendly way and how much impact malleability may have in practice.

Chapter 6

Conclusions and Future Work

Contents

6.1 Conclusions

In this thesis, we have outlined key issues that must be addressed to enable efficient and productive programming of exascale supercomputers and beyond: *load balancing, fault tolerance, and resource elasticity.* For each key issue, we have described the state of the art and identified open research questions.

In the main part of the thesis, we have contributed to these identified questions. While the questions have been phrased in a broad and general manner, we have restricted our answers to the specific context of *Asynchronous Many-Task (AMT)* programming for distributed-memory systems supporting dynamic independent tasks. Throughout this thesis, we have built on the *APGAS* library as a parallel programming environment and on the lifeline-based work stealing algorithm as a dynamic load balancing technique.

Regarding load balancing, we have started by experimentally comparing the performance of cooperative and coordinated lifeline-based work stealing. Only minor performance differences between them have been observed. Afterwards we have proposed a novel hybrid lifeline-based work stealing technique that achieves both intra- and inter-process load balancing. In this context, we have introduced novel tasking constructs for spawning dynamic independent tasks and computing their results. The constructs include cancelable tasks, which are useful, e.g., for search problems. In experiments, the resulting *APGAS* variant, denoted by $APGAS_{hyb}$, has shown good scalability in most cases. Then, we have compared $APGAS_{hyb}$ with other HPC and data analytics libraries, based on typical benchmarks from the two domains of HPC and data analytics. Regarding performance, $APGAS_{hyb}$ was the clear winner. Regarding programmer productivity, $APGAS_{hyb}$ was also often the winner, e.g., it required the lowest number of different constructs. Hence, $APGAS_{hyb}$ might be a good candidate for programming both HPC and data analytics applications using the same programming environment.

Regarding fault tolerance, we have proposed four techniques to protect programs transparently. All perform local recovery, continue the program execution after a failure with fewer resources, and can tolerate multiple simultaneous failures. Our first technique, *TC*, writes uncoordinated checkpoints into a resilient store. The checkpoints are written regularly at fixed time intervals as well as in the events of stealing and recovery. They only comprise task descriptors and interim results. We have described *TC* in a general way, such that it can be applied to a spectrum of task models. Experiments have shown a failure-free overhead below 1% and a recovery overhead after failures below 0.5 seconds, both for smooth weak scaling. The results clearly show that program protection at the level of an AMT runtime system has a significantly lower running time overhead and a lower recovery overhead than the well-known checkpoint/restart library DMTCP.

In addition to *TC*, we have proposed three more enhanced variants: *IncTC*, *LogTC* and *SST*. All variants aim to further reduce the overheads. *IncTC* periodically writes checkpoints in a resilient store, like *TC*, but performs the writing incrementally and for "stable" tasks only. For *LogTC* and *SST*, we have adopted a supervision and steal tracking technique for nested fork-join programs (SST_{NFJ}) in two different ways. *LogTC* combines SST_{NFJ} with *TC* and does not write checkpoints in the event of stealing. *SST* transfers SST_{NFJ} to our context and does not write checkpoints at all. This way, we have shown that task-level resilience techniques are not specific to a particular task model. Experiments showed no clear winner between the four techniques.

Thereafter, we have determined conditions under which either *TC* or *SST* is superior in single application runs. For this purpose, we have derived running time formulas depending on MTBF, number of workers, and steal rate. In addition, we have simulated the execution of job sets on a real and a hypothetical supercomputer and evaluated the makespans. All investigations consistently support the same conclusions: program protection at the level of an AMT runtime system pays off. Moreover, the choice between *TC* and *SST* is secondary. We have consistently observed *SST* as being superior in typical current settings, but *TC* takes over on large machines and for frequent errors.

Regarding resource elasticity, we have proposed a novel technique, denoted by *TRE*, that enables the transparent adaptation of programs to the addition or release of multiple nodes. Resource changes are accomplished by automatically relocating tasks to added nodes and away from released nodes. For a performance analysis, we have derived formulas that predict the running times of work stealing programs that change their number of resources at runtime.

By combining these formulas with experiments, we have demonstrated that the costs for adding/releasing up to 64 nodes are below 0.5 seconds and that both adding and releasing scales well. Based on these results, we have simulated the execution of sets of malleable jobs on two hypothetical supercomputers. For the simulated job scheduling, we have introduced a strategy to assign resources to malleable jobs as well as a heuristic to help users to determine the minimum, maximum and preferred number of nodes for a given job. Results include a reduction of the makespan by up to 20% compared to rigid jobs.

6.2 Future Work

As noted before, we have limited our contributions to the context of AMTs. Nevertheless, we had to leave some items on the table in this context as well. These should be addressed in future work.

Since we have treated each of the key issues separately, it would be beneficial to combine all of our techniques. The result would be a parallel programming environment that provides load balancing, fault tolerance and resource elasticity together. Furthermore, our implementations are prototypical, but it would be desirable to implement our techniques in a productive programming environment. In addition, our techniques could then be experimentally evaluated with real-world applications and not only with comparatively small benchmarks as in this thesis.

Next, as described in the Introduction, we have restricted our contributions to the context of clusters of homogeneous multi-core nodes, whereas supercomputers are becoming increasingly hierarchical and heterogeneous. Due to this evolution, future work should adapt our techniques to different architectures such as GPUs or FPGAs. Furthermore, an extension to other task models such as dataflow would be valuable.

Regarding load balancing, we focused on lifeline-based work stealing. Future work should evaluate more work stealing variants as well as other dynamic load balancing techniques. Furthermore, innovative technologies such as RDMA should be considered.

Regarding fault tolerance, we have restricted our techniques to the handling of fail-stop failures. Future work could investigate if and how our techniques can be extended for handling silent errors. Since we have described our techniques only textually, it would be desirable to verify their correctness formally. For this, tools such as Spin [211] could be used.

Regarding resource elasticity, we have restricted our technique to resource requests that refer to entire nodes. However, the current evolution of supercomputers may lead to "fat" nodes, which comprise multiple accelerators [212]. To efficiently utilize such systems, on-node resource elasticity will be a key issue. Thus, future work should extend our technique to include fine-grained resource changes. In addition, future work should investigate techniques that decide when shrinking/growing is appropriate within applications and take the initiative accordingly. As of today, resource elasticity is hardly used in practice on HPC systems, partly because widely used job schedulers provide it only in a rudimentary manner. Future work should address this by proposing a unified communication API to be implemented by job schedulers and programming environments.

Bibliography

[P1] Jonas Posner and Claudia Fohry. "Cooperation vs. Coordination for Lifeline-Based Global Load Balancing in APGAS". In: *Proceedings SIGPLAN Workshop on X10*. ACM, 2016, pp. 13–17. DOI: `10.1145/2931028.2931029`.

[P2] Jonas Posner and Claudia Fohry. "Fault Tolerance for Cooperative Lifeline-Based Global Load Balancing in Java with APGAS and Hazelcast". In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS) Workshops (APDCM)*. IEEE, 2017, pp. 854–863. DOI: `10.1109/ipdpsw.2017.31`.

[P3] Jonas Posner. "A Generic Reusable Java Framework for Fault-Tolerant Parallelization with the Task Pool Pattern". In: *International Parallel and Distributed Processing Symposium (IPDPS), Ph.D. Forum*. Poster. 2017.

[P4] Jonas Posner and Claudia Fohry. "A Java Task Pool Framework providing Fault-Tolerant Global Load Balancing". In: *Special Issue International Journal of Networking and Computing (IJNC)* 8.1 (2018), pp. 2–31. DOI: `10.15803/ijnc.8.1_2`.

[P5] Jonas Posner and Claudia Fohry. "A Combination of Intra- and Inter-place Work Stealing for the APGAS Library". In: *Proceedings Parallel Processing and Applied Mathematics (PPAM) Workshops (WLPP)*. Springer, 2018, pp. 234–243. DOI: `10.1007/978-3-319-78054-2_22`.

[P6] Jonas Posner and Claudia Fohry. "Hybrid Work Stealing of Locality-Flexible and Cancelable Tasks for the APGAS Library". In: *The Journal of Supercomputing* (2018), pp. 1435–1448. DOI: `10.1007/s11227-018-2234-8`.

[P7] Claudia Fohry, Jonas Posner, and Lukas Reitz. "A Selective and Incremental Backup Scheme for Task Pools". In: *Proceedings International Conference on High Performance Computing & Simulation (HPCS)*. 2018, pp. 621–628. DOI: `10.1109/HPCS.2018.00103`.

[P8] Jonas Posner, Lukas Reitz, and Claudia Fohry. "Comparison of the HPC and Big Data Java Libraries Spark, PCJ, and APGAS". In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC) Workshops (PAW-ATM)*. ACM, 2018, pp. 11–22. DOI: `10.1109/PAW-ATM.2018.00007`.

[P9] Jonas Posner, Lukas Reitz, and Claudia Fohry. “A Comparison of Application-Level Fault Tolerance Schemes for Task Pools”. In: *Future Generation Computing Systems (FGCS)* 105 (2019), pp. 119–134. DOI: `10.1016/j.future.2019.11.031`.

[P10] Jonas Posner. “System-Level vs. Application-Level Checkpointing”. In: *Proceedings International Conference on Cluster Computing (CLUSTER), Extended Abstract.* IEEE, 2020, pp. 404–405. DOI: `10.1109/CLUSTER49012.2020.00051`.

[P11] Jonas Posner. “Locality-Flexible and Cancelable Tasks for the APGAS Library”. In: *EuroHPC Summit Week, PRACEdays.* Poster. 2021.

[P12] Jonas Posner, Lukas Reitz, and Claudia Fohry. “Checkpointing vs. Supervision Resilience Approaches for Dynamic Independent Tasks”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS) Workshops (APDCM).* IEEE, 2021, pp. 556–565. DOI: `10.1109/IPDPSW52791.2021.00089`.

[P13] Jonas Posner. “Resource Elasticity at Task-Level”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS), Ph.D. Forum, Extended Abstract.* IEEE, 2021. DOI: `10.1109/IPDPSW52791.2021.00160`.

[P14] Jonas Posner and Claudia Fohry. “Transparent Resource Elasticity for Task-Based Cluster Environments with Work Stealing”. In: *Proceedings International Conference on Parallel Processing (ICPP) Workshops (P2S2).* ACM, 2021. DOI: `10.1145/3458744.3473361`.

[P15] Jonas Posner, Lukas Reitz, and Claudia Fohry. “Task-Level Resilience: Checkpointing vs. Supervision”. In: *Special Issue International Journal of Networking and Computing (IJNC)* 12.1 (2022), pp. 47–72. DOI: `10.15803/ijnc.12.1_47`.

[16] Olivier Tardieu. “The APGAS Library: Resilient Parallel and Distributed Programming in Java 8”. In: *Proceedings SIGPLAN Workshop on X10.* ACM, 2015, 25–26. DOI: `10.1145/2771774.2771780`.

[17] Vijay A. Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. “Lifeline-based Global Load Balancing”. In: *Proceedings SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP).* ACM, 2011, pp. 201–212. DOI: `10.1145/1941553.1941582`.

[18] Wei Zhang, Olivier Tardieu, David Grove, Benjamin Herta, Tomio Kamada, Vijay Saraswat, and Mikio Takeuchi. “GLB: Lifeline-based Global Load Balancing Library in X10”. In: *Proceedings Workshop on Parallel Programming for Analytics Applications (PPAA).* ACM, 2014, pp. 31–40. DOI: `10.1145/2567634.2567639`.

[19] Patrick Finnerty, Tomio Kamada, and Chikara Ohta. “Self-Adjusting Task Granularity for Global Load Balancer Library on Clusters of Many-Core Processors”. In: *Proceedings International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. ACM, 2020. DOI: 10.1145/3380536.3380539.

[20] Claudia Fohry, Marco Bungart, and Paul Plock. “Fault Tolerance for Lifeline-Based Global Load Balancing”. In: *Journal of Software Engineering and Applications (JSEA)* 10.13 (2017), pp. 925–958. DOI: 10.4236/jsea.2017.1013053.

[21] Gokcen Kestor, Sriram Krishnamoorthy, and Wenjing Ma. “Localized Fault Recovery for Nested Fork-Join Programs”. In: *Proceedings International Symposium on Parallel and Distributed Processing (IPDPS)*. IEEE, 2017, pp. 397–408. DOI: 10.1109/ipdps.2017.75.

[22] HiPEAC: European Network on High-performance Embedded Architecture and Compilation. *The HiPEAC Vision*. URL: https://www.hipeac.net/vision/2021 (visited on 12/01/2021).

[23] The European Technology Platform For High-Performance Computing ETP4HPC Association. *Strategic Research Agenda*. URL: https://www.etp4hpc.eu/pujades/files/ETP4HPC_SRA4_2020_web(1).pdf (visited on 12/01/2021).

[24] SC Conference Series. *SC20 More Than HPC Plenary: Advanced Computing and COVID-19*. URL: https://www.youtube.com/watch?v=iyd4enL_6lQ (visited on 12/01/2021).

[25] HPCwire. *PRACE Looks Back on a Year of COVID-19 Supercomputing*. URL: https://www.hpcwire.com/2021/02/04/prace-looks-back-on-a-year-of-covid-19-supercomputing (visited on 12/01/2021).

[26] Sandro Fiore, Mohamed Bakhouya, and Waleed W. Smari. “On the road to exascale: Advances in High Performance Computing and Simulations - An overview and editorial”. In: *Future Generation Computer Systems (FGCS)* 82 (2018), pp. 450–458. DOI: 10.1016/j.future.2018.01.034.

[27] Brad McCredie. *HiPEAC21 Keynote The Path to Exascale and Beyond*. URL: https://www.youtube.com/watch?v=FjwvJCkK9JY (visited on 12/01/2021).

[28] TOP500.org. *Supercomputer Fugaku*. URL: https://top500.org/system/179807 (visited on 12/01/2021).

[29] Oak Ridge National Laboratory. *Frontier*. URL: https://www.olcf.ornl.gov/frontier (visited on 12/01/2021).

[30] Thomas Sterling, Matthew Anderson, and Maciej Brodowicz. *High Performance Computing: Modern Systems and Practices.* Elsevier, 2018. ISBN: 9780124201583.

[31] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing.* Addison-Wesley, 2003. ISBN: 0201648652.

[32] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers.* CRC, 2010. ISBN: 143981192X. DOI: 10.5555/1855048.

[33] Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. "Addressing Failures in Exascale Computing". In: *The International Journal of High Performance Computing Applications (IJHPCA)* 28.2 (2014), pp. 129–173. DOI: 10.1177/1094342014522573.

[34] Thomas Herault and Yves Robert, eds. *Fault-Tolerance Techniques for High-Performance Computing.* Springer, 2015. DOI: 10.1007/978-3-319-20943-2.

[35] Al Geist. "How to kill a Supercomputer: Dirty Power, Cosmic Rays, and Bad Solder". In: *IEEE Spectrum* (2016). URL: https://spectrum.ieee.org/computing/hardware/how-to-kill-a-supercomputer-dirty-power-cosmic-rays-and-bad-solder.

[36] MPI Forum. *MPI: A Message-Passing Interface Standard Version 4.0.* URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf (visited on 12/01/2021).

[37] OpenMP Architecture Review Board. *OpenMP API 5.1 Specification.* URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf (visited on 12/01/2021).

[38] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. "A survey of MPI usage in the US Exascale Computing Project". In: *Concurrency and Computation: Practice and Experience (CCPE)* 32.3 (2020). DOI: 10.1002/cpe.4851.

[39] George Almasi. "PGAS (Partitioned Global Address Space) Languages". In: *Encyclopedia of Parallel Computing.* Springer, 2011, pp. 1539–1545. DOI: 10.1007/978-0-387-09766-4_210.

[40] Robert W. Numrich and John Reid. “Co-Arrays in the next Fortran Standard”. In: *SIGPLAN Fortran Forum* 24.2 (2005), pp. 4–17. DOI: `10.1145/1080399.1080400`.

[41] OpenSHMEM. *Application Programming Interface.* URL: `http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.5.pdf` (visited on 12/01/2021).

[42] Tarek El-Ghazawi and Lauren Smith. “UPC: Unified Parallel C”. In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC).* ACM, 2006. DOI: `10.1145/1188455.1188483`.

[43] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. “UPC++: A High-Performance Communication Framework for Asynchronous Computation”. In: *International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 2019, pp. 963–973. DOI: `10.1109/IPDPS.2019.00104`.

[44] Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. “Titanium: A High-performance Java Dialect”. In: *Concurrency: Practice and Experience* 10.11-13 (1998), pp. 825–836.

[45] Marek Nowicki and Piotr Bała. “Parallel computations in Java with PCJ library”. In: *Proceedings International Conference on High Performance Computing Simulation (HPCS).* IEEE, 2012, pp. 381–387. DOI: `10.1109/HPCSim.2012.6266941`.

[46] Intel. *oneAPI Threading Building Blocks.* URL: `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onetbb.html` (visited on 12/01/2021).

[47] Oracle. *Class ForkJoinPool.* URL: `https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/concurrent/ForkJoinPool.html` (visited on 12/01/2021).

[48] Doug Lea. “A Java Fork/Join Framework”. In: *Proceedings of the Conference on Java Grande.* ACM, 2000, pp. 36–43. DOI: `10.1145/337449.337465`.

[49] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing”. In: *SIGPLAN Notices* 40.10 (2005), pp. 519–538. DOI: `10.1145/1103845.1094852`.

[50] Bardford L. Chamberlain, David Callahan, and Hans P. Zima. “Parallel Programmability and the Chapel Language”. In: *The International Journal of High Performance Computing Applications (IJHPCA)* 21.3 (2007), pp. 91–312. DOI: 10.1177/1094342007078442.

[51] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. “Legion: Expressing Locality and Independence with Logical Regions”. In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC).* ACM, 2012, pp. 1–11.

[52] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *Concurrency and Computation: Practice and Experience (CCPE)* 23 (2 2011), pp. 187–198. DOI: 10.1002/cpe.1631.

[53] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. “HPX: A Task Based Programming Model in a Global Address Space”. In: *Proceedings International Conference on Partitioned Global Address Space Programming Models (PGAS).* ACM, 2014, pp. 1–11. DOI: 10.1145/2676870.2676883.

[54] Claudia Fohry. *An overview of task-based parallel programming models.* Tutorial at European Network on High-performance Embedded Architecture and Compilation Conference (HiPEAC). 2019.

[55] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, Thomas Fahringer, Kostas Katrinis, Erwin Laure, and Dimitrios S. Nikolopoulos. “A Taxonomy of Task-Based Parallel Programming Technologies for High-Performance Computing”. In: *The Journal of Supercomputing* 74.4 (2018), pp. 1422–1434. DOI: 10.1007/s11227-018-2238-4.

[56] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Lee, Sean Treichler, Patrick McCormick, and Alex Aiken. “Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance”. In: *Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis (SC).* ACM, 2020. DOI: 10.5555/3433701.3433783.

[57] Gurhem Jérôme and Serge G. Petiton. “A Current Task-Based Programming Paradigms Analysis”. In: *Computational Science (ICCS)*. Springer International Publishing, 2020, pp. 203–216. DOI: 10.1007/978-3-030-50426-7_16.

[58] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M. Badia. “ServiceSs: An Interoperable Programming Framework for the Cloud”. In: *Journal Grid Computing* 12.1 (2014), pp. 67–91. DOI: 10.1007/s10723-013-9272-5.

[59] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM* 51.1 (2008), p. 107. DOI: 10.1145/1327452.1327492.

[60] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. *Implementing YewPar: A Framework for Parallel Tree Search.* 2019. DOI: 10.1007/978-3-030-29400-7_14.

[61] Blair Archibald, Patrick Maier, Robert Stewart, and Phil Trinder. “YewPar: Skeletons for Exact Combinatorial Search”. In: *Proceedings SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2020, pp. 292–307. DOI: 10.1145/3332466.3374537.

[62] Peter Pirkelbauer, Amalee Wilson, Christina Peterson, and Damian Dechev. “Blaze-Tasks: A Framework for Computing Parallel Reductions over Tasks”. In: *Transactions on Architecture and Code Optimization (TACO)* 15.4 (2019). DOI: 10.1145/3293448.

[63] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. “The Implementation of the Cilk-5 Multithreaded Language”. In: *Proceedings Conference on Programming Language Design and Implementation (PLDI)* (1998), pp. 212–223. DOI: 10.1145/277650.277725.

[64] Rob V. Van Nieuwpoort, Gosia Wrzesińska, Ceriel J. H. Jacobs, and Henri E. Bal. “Satin: a High-Level and Efficient Grid Programming Model”. In: *Transactions on Programming Languages and Systems (TOPLAS)* 32.3 (2010), pp. 1–40. DOI: 10.1145/1709093.1709096.

[65] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. “Dynamic Task Discovery in PaRSEC: A Data-Flow Task-Based Runtime”. In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC) Workshops (ScalA)*. ACM, 2017, pp. 1–8. DOI: 10.1145/3148226.3148233.

[66] Vivek Kumar. “PufferFish: NUMA-Aware Work-stealing Library using Elastic Tasks”. In: *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2020, pp. 251–260. DOI: 10.1109/HiPC50609.2020.00039.

[67] Reazul Hoque and Pavel Shamis. “Distributed Task-Based Runtime Systems - Current State and Micro-Benchmark Performance”. In: *Proceedings International Conference on High Performance Computing and Communications (HPCC)*. IEEE, 2018, pp. 934–941. DOI: 10.1109/HPCC/SmartCity/DSS.2018.00155.

[68] Robert D. Blumofe and Charles E. Leiserson. “Scheduling Multithreaded Computations by Work Stealing”. In: *Journal of the ACM* 46.5 (1999), pp. 720–748. DOI: 10.1145/324133.324234.

[69] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. “Scheduling Parallel Programs by Work Stealing with Private Deques”. In: *SIGPLAN Notices* 48.8 (2013), pp. 219–228. DOI: 10.1145/2442516.2442538.

[70] Kento Yamashita and Tomio Kamada. “Introducing a Multithread and Multistage Mechanism for the Global Load Balancing Library of X10”. In: *Journal of Information Processing* 24.2 (2016), pp. 416–424. DOI: 10.2197/ipsjjip.24.416.

[71] Faisal Shahzad, Markus Wittmann, Moritz Kreutzer, Thomas Zeise, Georg Hager, and Gerhard Wellein. “A survey of checkpoint/restart techniques on distributed memory systems”. In: *Parallel Processing Letters (PPL)* 23.4 (2013), pp. 1340011–1340030. DOI: 10.1142/s0129626413400112.

[72] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. “Toward Exascale Resilience: 2014 Update”. In: *Supercomputing Frontiers and Innovations (JSFI)* 1.1 (2014), pp. 5–28. DOI: 10.14529/jsfi140101.

[73] Jason Ansel, Kapil Arya, and Gene Cooperman. “DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2009, pp. 1–12. DOI: 10.1109/ipdps.2009.5161063.

[74] Paul H. Hargrove and Jason C. Duell. “Berkeley lab checkpoint/restart (BLCR) for Linux clusters”. In: *Journal of Physics: Conference Series* 46 (2006), pp. 494–499. DOI: 10.1088/1742-6596/46/1/067.

[75] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R. de Supinski. “Design, Modeling, and Evaluation of a Scalable Multi-Level Checkpointing System”. In: *Proceedings International Conference for High Performance Computing,*

Networking, Storage and Analysis (SC). ACM, 2010, pp. 1–11. DOI: 10.1109/SC.2010.18.

[76] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. "FTI: High performance Fault Tolerance Interface for hybrid systems". In: *Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2011, pp. 1–32. DOI: 10.1145/2063384.2063427.

[77] Rajeev Jain, Klaus Weide, Saurabh Chawdhary, and Thomas Klostermann. "Checkpoint/Restart for Lagrangian particle mesh with AMR in community code FLASH-X". In: *International Symposium on Checkpointing for Supercomputing (SuperCheck)*. 2021. DOI: arXiv:2103.04267.

[78] Hubertus J. J. van Dam, Abhinav Vishnu, and Wibe A. de Jong. "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 (JCTCCE)* 7.1 (2010), pp. 66–75. DOI: 10.1021/ct100439u.

[79] Amina Guermouche, Thomas Ropars, Elisabeth Brunet, Marc Snir, and Franck Cappello. "Uncoordinated Checkpointing Without Domino Effect for Send-Deterministic MPI Applications". In: *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 989–1000. DOI: 10.1109/IPDPS.2011.95.

[80] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. "Redesigning the Message Logging Model for High Performance". In: *Concurrency and Computation: Practice and Experience (CCPE)* 22.16 (2010), pp. 2196–2211. DOI: 10.1002/cpe.1589.

[81] Christian Engelmann and Al Geist. "Super-Scalable Algorithms for Computing on 100,000 Processors". In: *Computational Science (ICCS)*. Springer, 2005, pp. 313–321.

[82] Kuang-Hua Huang and Jacob A. Abraham. "Algorithm-Based Fault Tolerance for Matrix Operations". In: *Transaction on Computers* 33.6 (1984), 518–528. DOI: 10.1109/TC.1984.1676475.

[83] Nawab Ali, Sriram Krishnamoorthy, Mahantesh Halappanavar, and Jeff Daily. "Multi-Fault Tolerance for Cartesian Data Distributions". In: *International Journal of Parallel Programming (JPDC)* 41.3 (2012), pp. 469–493. DOI: 10.1007/s10766-012-0218-5.

[84] George Bosilca, Rémi Delmas, Jack Dongarra, and Julien Langou. "Algorithm-based fault tolerance applied to high performance computing". In: *Journal of Parallel and Distributed Computing (JPDC)* 69.4 (2009), pp. 410–416. DOI: `10.1016/j.jpdc.2008.12.002`.

[85] Sihuan Li, Hongbo Li, Xin Liang, Jieyang Chen, Elisabeth Giem, Kaiming Ouyang, Kai Zhao, Sheng Di, Franck Cappello, and Zizhong Chen. "FT-ISort: Efficient Fault Tolerance for Introsort". In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2019, pp. 1–17. DOI: `10.1145/3295500.3356195`.

[86] Sri Raj Paul, Akihiro Hayashi, Nicole Slattengren, Hemanth Kolla, Matthew Whitlock, Seonmyeong Bak, Keita Teranishi, Jackson Mayo, and Vivek Sarkar. "Enabling Resilience in Asynchronous Many-Task Programming Models". In: *Proceeding Euro-Par: Parallel Processing*. Springer, 2019, pp. 346–360. DOI: `10.1007/978-3-030-29400-7_25`.

[87] Giorgis Georgakoudis, Luanzheng Guo, and Ignacio Laguna. "Reinit++: Evaluating the Performance of Global-Restart Recovery Methods For MPI Fault Tolerance". In: *Proceedings International Conference on High Performance Computing (ISC)*. 2020. DOI: `10.1007/978-3-030-50743-5_27`.

[88] Wesley Bland, Aurélien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. "Post-failure recovery of MPI communication capability: Design and rationale". In: *The International Journal of High Performance Computing Applications (IJHPCA)* 27.3 (2013), pp. 244–254. DOI: `10.1177/1094342013488238`.

[89] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. "Resilient X10". In: *SIGPLAN* 49.8 (2014), pp. 67–80. DOI: `10.1145/2692916.2555248`.

[90] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. "Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales". In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC) Workshops (FTXS)*. ACM, 2014, pp. 895–906. DOI: `10.1109/SC.2014.78`.

[91] Faisal Shahzad, Jonas Thies, Moritz Kreutzer, Thomas Zeiser, Georg Hager, and Gerhard Wellein. "CRAFT: A Library for Easier Application-Level Checkpoint/Restart and Automatic Fault Tolerance". In: *Transactions on Parallel*

and Distributed Systems (TPDS) 30.3 (2019), pp. 501–514. DOI: `10.1109/TPDS.2018.2866794`.

[92] Robert D. Blumofe and Philip A. Lisiecki. “Adaptive and Reliable Parallel Computing on Networks of Workstations”. In: *Proceedings Annual Conference on USENIX*. 1997, pp. 1–10.

[93] Gosia Wrzesińska, Rob V. van Nieuwpoort, Jason Maassen, and Henri E. Bal. “Fault-Tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2005, pp. 1–10. DOI: `10.1109/ipdps.2005.224`.

[94] Wenjing Ma and Sriram Krishnamoorthy. “Data-driven Fault Tolerance for Work Stealing Computations”. In: *Proceedings International Conference on Supercomputing (ICS)*. ACM, 2012, pp. 79–90. DOI: `10.1145/2304576.2304589`.

[95] Gengbin Zheng, Lixia Shi, and L.V. Kale. “FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI”. In: *Proceedings International Conference on Cluster Computing (CLUSTER)*. IEEE, 2004, pp. 93–103. DOI: `10.1109/CLUSTR.2004.1392606`.

[96] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. “Parallel Programming with Migratable Objects: Charm++ in Practice”. In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2014, pp. 647–658. DOI: `10.1109/SC.2014.58`.

[97] Romain Lion and Samuel Thibault. “From tasks graphs to asynchronous distributed checkpointing with local restart”. In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC) Workshops (FTXS)*. ACM, 2020, pp. 31–40. DOI: `10.1109/FTXS51974.2020.00009`.

[98] Dror G. Feitelson and Larry Rudolph. “Toward Convergence in Job Schedulers for Parallel Supercomputers”. In: *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*. Springer, 1996, pp. 1–26. DOI: `10.1007/BFb0022284`.

[99] IBM. *Elastic X10*. URL: `http://x10-lang.org/documentation/practical-x10-programming/elastic-x10.html` (visited on 12/01/2021).

[100] Mohak Chadha, Jophin John, and Michael Gerndt. “Extending Slurm for Dynamic Resource-Aware Adaptive Batch Scheduling”. In: *Proceedings International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2020. DOI: `10.1109/HiPC50609.2020.00036`.

[101] Sergio Iserte, Rafael Mayo, Enrique S. Quintana-Ortí, and Antonio J. Peña. "DMRlib Easy-coding and Efficient Resource Management for Job Malleability". In: *Transactions on Computers (TC)* (2020). DOI: `10.1109/TC.2020.3022933`.

[102] Suraj Prabhakaran, Marcel Neumann, Sebastian Rinke, Felix Wolf, Abhishek Gupta, and Laxmikant V. Kale. "A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications". In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2015, pp. 429–438. DOI: `10.1109/IPDPS.2015.34`.

[103] Rajesh Sudarsana and Calvin J.Ribbens. "Combining Performance and Priority for Scheduling Resizable Parallel Applications". In: *Journal of Parallel and Distributed Computing (JPDC)* 87 (2016), pp. 55–66. DOI: `10.1016/j.jpdc.2015.09.007`.

[104] Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Joseph Koning, Tapasya Patki, Thomas R. W. Scogland, Becky Springmeyer, and Michela Taufer. "Flux: Overcoming Scheduling Challenges for Exascale Workflows". In: *Workflows in Support of Large-Scale Science (WORKS)*. IEEE/ACM, 2018, pp. 10–19. DOI: `10.1109/WORKS.2018.00007`.

[105] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. "Dynamic Malleability in Iterative MPI Applications". In: *International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE, 2007, pp. 591–598. DOI: `10.1109/CCGRID.2007.45`.

[106] Chao Huang, Orion Lawlor, and L. V. Kalé. "Adaptive MPI". In: *Languages and Compilers for Parallel Computing (LCPC)*. Springer, 2004, pp. 306–322. DOI: `10.1007/978-3-540-24644-2_20`.

[107] Gengbin Zheng, Xiang Ni, and Laxmikant V. Kalé. "A Scalable Double In-Memory Checkpoint and Restart Scheme Towards Exascale". In: *International Conference on Dependable Systems and Networks Workshops (DSN)*. IEEE, 2012, pp. 1–6. DOI: `10.1109/DSNW.2012.6264677`.

[108] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kalé. "Towards Realizing the Potential of Malleable Jobs". In: *International Conference on High Performance Computing (HiPC)*. IEEE, 2014, pp. 1–10. DOI: `10.1109/HiPC.2014.7116905`.

[109] Marco Bungart and Claudia Fohry. "A Malleable and Fault-Tolerant Task Pool Framework for X10". In: *Proceedings International Conference on Cluster Computing (CLUSTER), Workshop on Fault Tolerant Systems*. IEEE, 2017, pp. 749–757. DOI: `10.1109/CLUSTER.2017.27`.

[110] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. “The Asynchronous Partitioned Global Address Space Model”. In: *Proceedings SIGPLAN Workshop on Advances in Message Passing (AMP).* ACM, 2010.

[111] IBM. *The APGAS Library for Fault-Tolerant Distributed Programming in Java 8.* URL: `https://github.com/x10-lang/apgas` (visited on 12/01/2021).

[112] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. “Apache Spark: A Unified Engine for Big Data Processing”. In: *Communications of the ACM (CACM)* 59.11 (2016), pp. 56–65. DOI: `10.1145/2934664`.

[113] James Dinan, D. Brian Larkins, P. Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. “Scalable Work Stealing”. In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC).* ACM, 2009. DOI: `10.1145/1654059.1654113`.

[114] Piotr Bała and Marek Nowicki. *PCJ: HPC Challenge Award at Supecomputing’14.* URL: `https://pcj.icm.edu.pl/hpcc-award` (visited on 12/01/2021).

[115] IBM. *The X10 Programming Language.* URL: `https://github.com/x10-lang` (visited on 12/01/2021).

[116] Philippe Suter, Olivier Tardieu, and Josh Milthorpe. “Distributed Programming in Scala with APGAS”. In: *Proceedings SIGPLAN Symposium on Scala.* ACM, 2015, pp. 13–7. DOI: `10.1145/2774975.2774977`.

[117] E.N. Elnozahy, Ricardo Bianchini, Tarek El-Ghazawi, Armando Fox, Forest Godf, Adolfy Hoisie, Kathryn McKinley, Rami Melhem, James Plank, Partha Ranganathan, and Josh Simons. *System Resilience at Extreme Scale.* Tech. rep. DARPA, 2008.

[118] Hazelcast. *The Leading Open Source In-Memory Data Grid.* URL: `http://hazelcast.org` (visited on 12/01/2021).

[119] Oracle. *Package java.util.concurrent.* URL: `https://docs.oracle.com/javase/8/docs/api/index.html?java/util/concurrent/package-summary.html` (visited on 12/01/2021).

[120] Hazelcast. *Hazelcast 3.12.12 API: IMap.* URL: `https://docs.hazelcast.org/docs/3.12.12/javadoc/com/hazelcast/core/IMap.html` (visited on 12/01/2021).

[121] Patrick Finnerty. *Java GLB*. URL: `https://github.com/handist/JavaGLB` (visited on 12/01/2021).

[122] Stephen Olivier, Journaln Huan, Journalnze Liu, Journaln Prins, Journalmes Dinan, P. Sadayappan, and Chau-Wen Tseng. “UTS: An Unbalanced Tree Search Benchmark”. In: *Languages and Compilers for Parallel Computing (LCPC)*. Springer, 2006, pp. 235–250. DOI: `10.1007/978-3-540-72521-3_18`.

[123] Linton C. Freeman. “A Set of Measures of Centrality Based on Betweenness”. In: *Sociometry* 40.1 (1977), p. 35. DOI: `10.2307/3033543`.

[124] Evgeni J. Gik. *Schach und Mathematik*. 1st ed. Thun, 1987. ISBN: 3-87144-987-3.

[125] Rice University. *HabaneroUPC++: a Compiler-free PGAS Library*. URL: `https://github.com/habanero-rice/habanero-upc` (visited on 12/01/2021).

[126] Malvin H. Kalos. *Monte Carlo methods*. Ed. by Paula A. Whitlock. WILEY-VCH, 2008. ISBN: 978-3-527-40760-6.

[127] Barcelona Supercomputing Center. *Component Superscalar framework and programming model for HPC (COMPSs)*. URL: `https://github.com/bsc-wdc/compss` (visited on 12/01/2021).

[128] David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William Journal Cook. *The Traveling Salesman Problem*. Princeton University Press, 2007. ISBN: 0691129932.

[129] Bernard Gendron and Teodor Gabriel Crainic. “Parallel Branch-and-Branch Algorithms: Survey and Synthesis”. In: *Operations Research* 42.6 (1994), pp. 1042–1066. DOI: `10.1287/opre.42.6.1042`.

[130] Competence Center for High Performance Computing in Hessen (HKHLR). *Linux Cluster Kassel*. URL: `https://www.hkhlr.de/en/clusters/linux-cluster-kassel` (visited on 12/01/2021).

[131] TOP500.org. *Goethe-HLR*. URL: `https://www.top500.org/system/179588` (visited on 12/01/2021).

[132] Competence Center for High Performance Computing in Hessen (HKHLR). *Lichtenberg I Cluster Darmstadt Phase I*. URL: `https://www.hkhlr.de/en/clusters/lichtenberg-cluster-darmstadt` (visited on 12/01/2021).

[133] Jonas Posner. “Global Load Balancing and Intra-Node Synchronization with the Java Framework APGAS”. Master’s thesis. University of Kassel, 2016.

[134] Vivek Kumar, Karthik Murthy, Vivek Sarkar, and Yili Zheng. “Optimized Distributed Work-Stealing”. In: *Proceedings Workshop on Irregular Applications: Architectures and Algorithms (IA^3)*. 2016, pp. 74–77. DOI: `doi:10.1109/IA3.2016.19`.

[135] Yi Guo, Journalsheng Zhao, Vincent Cave, and Vivek Sarkar. “SLAW: A Scalable Locality-Aware Adaptive Work-Stealing Scheduler for Multi-Core Systems”. In: *Proceedings SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2010, 341–342. DOI: `10.1145/1693453.1693504`.

[136] Michael Anderson, Shaden Smith, Narayanan Sundaram, Mihai Capotă, Zheguang Zhao, Subramanya Dulloor, Nadathur Satish, and Theodore L. Willke. “Bridging the Gap between HPC and Big Data Frameworks”. In: *Proceedings of the VLDB Endowment* 10.8 (2017), pp. 901–912. DOI: `10.14778/3090163.3090168`.

[137] Jorge L. Reyes-Ortiz, Luca Oneto, and Davide Anguita. “Big Data Analytics in the Cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf”. In: *Procedia Computer Science* 53 (2015), pp. 121–130. DOI: `10.1016/j.procs.2015.07.286`.

[138] Tom White. *Hadoop: The Definitive Guide*. 4st. O’Reilly Media, 2015. ISBN: 9781491901632.

[139] Max Grossman and Vivek Sarkar. “SWAT: A Programmable, In-Memory, Distributed, High-Performance Computing Platform”. In: *Proceedings International Symposium on High-Performance Parallel andDistributed Computing (HPDC)*. ACM, 2016, 81–92. DOI: `10.1145/2907294.2907307`.

[140] Alex Gittens, Kai Rothauge, Shusen Wang, Michael W. Mahoney, Jey Kottalam, Lisa Gerhardt, Prabhat, Michael Ringenburg, and Kristyn Maschhoff. “Alchemist: An Apache Spark $\leftrightarrow$ MPI Interface”. In: *Concurrency and Computation: Practice and Experience*. 2018. DOI: `10.1002/cpe.5026`.

[141] The Apache Software Foundation. *Apache Spark*. URL: `https://github.com/apache/spark` (visited on 12/01/2021).

[142] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. “Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing”. In: *Proceedings USENIX Conference on Networked Systems Design and Implementation (NSDI)*. 2012.

[143] The Apache Software Foundation. *Apache Mesos*. URL: `https://github.com/apache/mesos` (visited on 12/01/2021).

[144] Piotr Bała and Marek Nowicki. *PCJ library repository.* URL: `https://github.com/hpdcj/PCJ` (visited on 12/01/2021).

[145] Marek Nowicki, Magdalene Ryczkowska, Łukasz Górski, and Piotr Bała. "Big Data Analytics in Java with PCJ Library: Performance Comparison with Hadoop". In: *Parallel Processing and Applied Mathematics (PPAM).* Springer, 2018, pp. 318–327. DOI: `10.1007/978-3-319-78054-2_30`.

[146] Piotr Bała, Łukasz Górski, and Marek Nowicki. "Performance Evaluation of Parallel Computing and Big Data Processing with Java and PCJ Library". In: *Cray User Group (CUG).* 2018.

[147] Marek Nowicki, Łukasz Górski, and Piotr Bała. "PCJ - Java Library for Highly Scalable HPC and Big Data Processing". In: *Proceedings International Conference on High Performance Computing Simulation (HPCS).* 2018, pp. 12–20. DOI: `10.1109/HPCS.2018.00017`.

[148] Michał Szynkiewicz and Marek Nowicki. "Fault-Tolerance Mechanisms for the Java Parallel Codes Implemented with the PCJ Library". In: *Parallel Processing and Applied Mathematics (PPAM).* Springer, 2018, pp. 298–307. DOI: `10.1007/978-3-319-78054-2_28`.

[149] Leo Tolstoy. *War and Peace.* 1952. URL: `https://github.com/GITenberg/War-and-Peace_2600`.

[150] Georges de Scudéry. *Artamène ou le Grand Cyrus.* 1654. URL: `http://www.artamene.org`.

[151] Google. *Google Java Style Guide.* URL: `https://google.github.io/styleguide/javaguide.html` (visited on 12/01/2021).

[152] Matthias Korch and Thomas Rauber. "A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms". In: *Concurrency and Computation: Practice and Experience* 16.1 (2003), 1–47.

[153] Ralf Hoffmann and Thomas Rauber. "Adaptive Task Pools: Efficiently Balancing Large Number of Tasks on Shared-address Spaces". In: *International Journal of Parallel Programming* 39.5 (2011), pp. 553–581. DOI: `10.1007/s10766-010-0156-z`.

[154] Kaushik Ravicandran, Sangho Lee, and Santosh Pande. "Work Stealing for Multi-core HPC Clusters". In: *Proceedings Euro-Par Parallel Processing.* Springer, 2011, pp. 205–217. DOI: `10.1007/978-3-642-23400-2_20`.

[155] Swann Perarnau and Mitsuhisa Sato. “Victim Selection and Distributed Work Stealing Performance: A Case Study”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2014, pp. 659–668. DOI: 10.1109/IPDPS.2014.74.

[156] James Dinan, Stephen Olivier, Gerald Sabin, Jan Prins, P. Sadayappan, and Chau-Wen Tseng. “Dynamic Load Balancing of Unbalanced Computations Using Message Passing”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370581.

[157] David Chase and Yossi Lev. “Dynamic Circular Work-Stealing Deque”. In: *Proceedings Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, 2005, pp. 21–28. DOI: 10.1145/1073970.1073974.

[158] James Dinan, Sriram Krishnamoorthy, D. Brian Larkins, Jarek Nieplocha, and P. Sadayappan. “Scioto: A Framework for Global-View Task Parallelism”. In: *International Conference on Parallel Processing (ICPP)*. IEEE, 2008, pp. 586–593. DOI: 10.1109/ICPP.2008.44.

[159] Hannah Cartier, James Dinan, and D. Brian Larkins. “Optimizing Work Stealing Communication with Structured Atomic Operations”. In: *Proceedings International Conference on Parallel Processing (ICPP)*. ACM, 2021. DOI: 10.1145/3472456.3472522.

[160] Jeeva Paudel, Olivier Tardieu, and José Nelson Amaral. “On the Merits of Distributed Work-Stealing on Selective Locality-Aware Tasks”. In: *Proceedings International Conference on Parallel Processing (ICCP)*. 2013. DOI: 10.1109/icpp.2013.19.

[161] Jeeva Paudel, Olivier Tardieu, and José Nelson Amaral. “Hybrid parallel task placement in X10”. In: *Proceedings SIGPLAN Workshop on X10*. 2013. DOI: 10.1145/2481268.2481277.

[162] Lukas Reitz. “Load Balancing Policies for Nested Fork-Join”. In: *Proceedings International Conference on Cluster Computing (CLUSTER), Extended Abstract*. IEEE, 2021, pp. 817–818. DOI: 10.1109/Cluster48925.2021.00075.

[163] Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer. “How to Cancel a Task”. In: *Proceedings International Conference Multicore Software Engineering, Performance, and Tools*. Springer, 2013, pp. 61–72. ISBN: 978-3-642-39955-8. DOI: 10.1007/978-3-642-39955-8_6.

[164] HamidReza Asaadi, Dounia Khaldi, and Barbara Chapman. “A Comparative Survey of the HPC and Big Data Paradigms: Analysis and Experiments”. In: *International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 423–432. DOI: `10.1109/CLUSTER.2016.21`.

[165] Lightbend. *Akka library repository*. URL: `https://github.com/akka/akka` (visited on 12/01/2021).

[166] Jonas Scherbaum. “Comparison of HabaneroUPC++ and APGAS Library”. Bachelor’s thesis. University of Kassel, 2016.

[167] Andreas Prell. “Embracing Explicit Communication in Work-Stealing Runtime Systems”. PhD thesis. University of Bayreuth, Germany, 2016.

[168] Jonathan Lifflander, Sriram Krishnamoorthy, and V. Laxmikant Kale. “Steal Tree: low-overhead tracing of work stealing schedulers”. In: ACM, 2013, pp. 507–518. DOI: `StealTree:Low-OverheadTracingofWorkStealingSchedulers`.

[169] Lukas Reitz. “Design and Evaluation of a Work Stealing-Based Fault Tolerance Scheme for Task Pools”. Master’s thesis. University of Kassel, 2019.

[170] Kapil Arya, Gene Cooperman, Rohan Garg, Jiajun Cao, and Artem Polyakov. *DMTCP: Distributed MultiThreaded CheckPointing*. URL: `https://github.com/dmtcp/dmtcp` (visited on 12/01/2021).

[171] John T. Daly. “A higher order estimate of the optimum checkpoint interval for restart dumps”. In: *Future Generation Computer Systems (FGCS)* 22.3 (2006), pp. 303–312. DOI: `10.1016/j.future.2004.11.016`.

[172] Anne Benoit, Valentin Le Fèvre, Padma Raghavan, Yves Robert, and Hongyang Sun. “Design and Comparison of Resilient Scheduling Heuristics for Parallel Jobs”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 1–12. DOI: `10.1109/IPDPSW50202.2020.00099`.

[173] Valentin Le Fèvre. *Source Code of Job Simulator*. URL: `http://www.github.com/vlefevre/job-scheduling` (visited on 12/01/2021).

[174] TOP500.org. *Mira - BlueGene/Q, Power BQC 16C 1.60GHz*. URL: `https://www.top500.org/system/177718` (visited on 12/01/2021).

[175] Argonne Leadership Computing Facility. *Mira log traces*. URL: `https://reports.alcf.anl.gov/data/mira.html` (visited on 12/01/2021).

[176] Saurabh Hukerikar and Christian Engelmann. “Resilience Design Patterns: A Structured Approach to Resilience at Extreme Scale”. In: *Supercomputing Frontiers and Innovations (JSFI)* 4.3 (2017), pp. 4–42. DOI: `10.14529/jsfi170301`.

[177] Ifeanyi P. Egwutuoha, David Levy, Bran Selic, and Shiping Chen. “A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems”. In: *The Journal of Supercomputing* 65.3 (2013), pp. 1302–1326. DOI: 10.1007/s11227-013-0884-0.

[178] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. “A Survey of Rollback-Recovery Protocols in Message-Passing Systems”. In: *Computing Surveys (CSUR)* 34.3 (2002), pp. 375–408. DOI: 10.1145/568522.568525.

[179] Greg Bronevetsky, Keshav Pingali, and Paul Stodghill. “Experimental Evaluation of Application-Level Checkpointing for OpenMP Programs”. In: *Proceedings International Conference on Supercomputing (ICS)*. ACM, 2006, pp. 2–13. DOI: 10.1145/1183401.1183405.

[180] Atsushi Hori, Kazumi Yoshinaga, Thomas Herault, Aurélien Bouteiller, George Bosilca, and Yutaka Ishikawa. “Overhead of using spare nodes”. In: *The International Journal of High Performance Computing Applications (IJHPCA)* 34.2 (2020), pp. 208–226. DOI: 10.1177/1094342020901885.

[181] George Bosilca, Aurélien Bouteiller, Thomas Herault, Yves Robert, and Jack Dongarra. “Composing resilience techniques: ABFT, periodic and incremental checkpointing”. In: *International Journal of Networking and Computing (IJNC)* 5.1 (2015), pp. 2–25. DOI: https://doi.org/10.15803/ijnc.5.1_2.

[182] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. “Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems”. In: *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis (SC) Workshops (FTXS)*. ACM, 2012, pp. 1–11. DOI: 10.1109/SC.2012.36.

[183] Nuria Losada, Patricia González, Marìa J. Martìn, George Bosilca, Aurélien Bouteiller, and Keita Teranishi. “Fault tolerance of MPI applications in exascale systems: The ULFM solution”. In: *Future Generation Computer Systems (FGCS)* 106 (2020), pp. 467–481. DOI: https://doi.org/10.1016/j.future.2020.01.026.

[184] David Grove, Sara S. Hamouda, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Josh Milthorpe, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Olivier Tardieu. “Failure Recovery in Resilient X10”. In: *Transactions on Programming Languages and Systems (TOPLAS)* 41.3 (2019), pp. 1–40. DOI: 10.1145/3332372.

[185] Sara S. Hamouda and Josh Milthorpe. “Resilient Optimistic Termination Detection for the Async-Finish Model”. In: *High Performance Computing*. Springer, 2019, pp. 291–311. DOI: 10.1007/978-3-030-20656-7_15.

[186] Sara S. Hamouda. “Resilience in High-Level Parallel Programming Languages”. PhD thesis. Research School of Computer Science, Australian National University, 2019.

[187] Bunjamin Memishi, Shadi Ibrahim, María S. Pérez, and Gabriel Antoniu. “Fault Tolerance in MapReduce: A Survey”. In: *Computer Communications and Networks*. Springer, 2016, pp. 205–240. DOI: 10.1007/978-3-319-44881-7_11.

[188] Ahcene Bendjoudi, Nouredine Melab, and El-Ghazali Talbi. “FTH-B&B: A Fault-Tolerant Hierarchical Branch and Bound for Large Scale Unreliable Environments”. In: *Transactions on Computers (TC* 63.9 (2014), pp. 2302–2315. DOI: 10.1109/tc.2013.40.

[189] Upama Kabir and Dhrubajyoti Goswami. “Identifying Patterns Towards Algorithm Based Fault Tolerance”. In: *Proceedings International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2015, pp. 508–516. DOI: 10.1109/hpcsim.2015.7237083.

[190] Mehmet Can Kurt, Sriram Krishnamoorthy, Kunal Agrawal, and Gagan Agrawal. “Fault-Tolerant Dynamic Task Graph Scheduling”. In: *Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2014, pp. 719–730. DOI: 10.1109/SC.2014.64.

[191] Chongxiao Cao, Thomas Herault, George Bosilca, and Jack Dongarra. “Design for a Soft Error Resilient Dynamic Task-Based Runtime”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2015, pp. 765–774. DOI: 10.1109/ipdps.2015.81.

[192] Omer Subasi, Tatiana Martsinkevich, Ferad Zyulkyarov, Osman Unsal, Jesus Labarta, and Franck Cappello. “Unified fault-tolerance framework for hybrid task-parallel message-passing applications”. In: *The International Journal of High Performance Computing Applications (IJHPCA)* 32.5 (2018), pp. 641–657. DOI: 10.1177/1094342016669416.

[193] Gosia Wrzesińska, Ana-Maria Oprescu, Thilo Kielmann, and Henri E. Bal. “Persistent Fault-Tolerance for Divide-and-Conquer Applications on the Grid”. In: *Proceedings Euro-Par Parallel Processing*. Vol. 4641. 2007, pp. 425–436. DOI: 10.1007/978-3-540-74466-5_46.

[194] Claudia Fohry, Marco Bungart, and Jonas Posner. “Towards an Efficient Fault-Tolerance Scheme for GLB”. In: *Proceedings SIGPLAN Workshop on X10*. ACM, 2015, pp. 27–32. DOI: 10.1145/2771774.2771779.

[195] Claudia Fohry. “Checkpointing and Localized Recovery for Nested Fork-Join Programs”. In: *International Symposium on Checkpointing for Supercomputing (SuperCheck)*. 2021. DOI: arXiv:2102.12941.

[196] Lukas Reitz. “Task-Level Checkpointing for Nested Fork-Join Programs”. In: *Proceedings International Parallel and Distributed Processing Symposium (IPDPS), Ph.D. Forum, Extended Abstract*. IEEE, 2021. DOI: 10.1109/IPDPSW52791.2021.00160.

[197] Marco Bungart. “Fehlertoleranz und Elastizität für ein Framework zur globalen Lastenbalancierung”. PhD thesis. University of Kassel, 2018. DOI: 10.17170/kobra-2018122577.

[198] Terracotta. *Terracotta Open Source Community*. URL: https://github.com/terracotta-oss (visited on 12/01/2021).

[199] Infinispan. *Infinispan, In-Memory Distributed Data Store*. URL: https://github.com/infinispan/infinispan (visited on 12/01/2021).

[200] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. “ElastiCon: An Elastic Distributed Sdn Controller”. In: *Proceedings Symposium on Architectures for Networking and Communications Systems*. ACM, 2014, pp. 17–28.

[201] Pradeeban Kathiravelu, Helena Galhardas, and Luís Veiga. “$\partial U \partial U$ Multi-Tenanted Framework: Distributed Near Duplicate Detection for Big Data”. In: *On the Move to Meaningful Internet Systems*. Springer-Verlag New York, Inc., 2015, pp. 237–256. DOI: 10.1007/978-3-319-26148-5_14.

[202] Pradeeban Kathiravelu and Luis Veiga. “Concurrent and Distributed CloudSim Simulations”. In: *Proceedings International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE, 2014, pp. 490–493. DOI: 10.1109/MASCOTS.2014.70.

[203] Pradeeban Kathiravelu and Luis Veiga. “An Adaptive Distributed Simulator for Cloud and MapReduce Algorithms and Architectures”. In: *Proceedings International Conference on Utility and Cloud Computing*. IEEE, 2014, pp. 79–88. DOI: 10.1109/UCC.2014.16.

[204] Anne Benoit, Thomas Herault, Valentin Le Fèvre, and Yves Robert. “Replication is More Efficient than You Think”. In: *Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2019, pp. 1–14. DOI: `10.1145/3295500.3356171`.

[205] Thomas Herault, Yves Robert, Aurélien Bouteiller, Dorian Arnold, Kurt Ferreira, George Bosilca, and Jack Dongarra. “Checkpointing Strategies for Shared High-Performance Computing Platforms”. In: *International Journal of Networking and Computing (IJNC)* 9.1 (2019), pp. 28–52.

[206] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. “A Tighter Analysis of Work Stealing”. In: *Algorithms and Computation*. Springer, 2010, pp. 291–302.

[207] SchedMD LLC. *Slurm: Scheduling Configuration Guide*. URL: `https://slurm.schedmd.com/sched_config.html` (visited on 12/01/2021).

[208] Gonzalo Martín, David E. Singh, Maria-Cristina Marinescu, and Jesús Carretero. “Enhancing the performance of malleable MPI applications by using performance-aware dynamic reconfiguration”. In: *Parallel Computing* 46 (2015), pp. 60–77. DOI: `10.1016/j.parco.2015.04.003`.

[209] Sergio Iserte, Héctor Martínez, Sergio Barrachina, Maribel Castillo, Rafael Mayo, and Antonio J Peña. “Dynamic Reconfiguration of Noniterative Scientific Applications: A Case Study with HPG Aligner”. In: *The International Journal of High Performance Computing Applications (IJHPCA)* 33.5 (2019), pp. 804–816. DOI: `10.1177/1094342018802347`.

[210] Stefan Kehrer and Wolfgang Blochinger. “Equilibrium: An Elasticity Controller for Parallel Tree Search in the Cloud”. In: *The Journal of Supercomputing* (2020). DOI: `10.1007/s11227-020-03197-y`.

[211] Spin. *Formal Verification*. URL: `http://spinroot.com` (visited on 12/01/2021).

[212] Martin Schulz, Dieter Kranzlmüller, Laura Brandon Schulz, Carsten Trinitis, and Josef Weidendorfer. “On the Inevitability of Integrated HPC Systems and How They Will Change HPC System Operations”. In: *Proceedings of the 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*. ACM, 2021. DOI: `10.1145/3468044.3468046`.