UNIKASSEL
VERSITÄT

TasGPI: A Global Load Balancing Framework for C++

MASTERTHESIS

Department of Electrical Engineering and Computer Science
Universität Kassel

Submitted by:	Adrian Steinitz
Matriculation number:	35673745
E-Mail:	uk077035@student.uni-kassel.de
Presented to:	Research Group Programming Languages/Methodologies
First examiner:	Prof. Dr. Claudia Fohry
Second examiner:	Prof. Dr. Oliver Hohlfeld
Supervisor:	Dr. Jonas Posner
Filed on:	October 4, 2023

Statutory Declaration

I herewith declare that I have composed the present thesis myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The thesis in the same or similar form has not been submitted to any examination body and has not been published. This thesis was not yet, even in part, used in another examination or as a course performance. Furthermore I declare that the submitted written (bound) copies of the present thesis and the versions submitted per e-mail and disc are consistent with each other in contents.

Kassel, October 4, 2023

Adrian Steinitz

Contents

Index of abbreviations

TasGPI	Tasks for GPI
SPMD	Single Program Multiple Data
MPMD	Multiple Program Multiple Data
MPI	Message Passing Interface
IB	InfiniBand
PGAS	Partitioned Global Address Space
APGAS	Asynchronous Partitioned Global Address Space
RDMA	Remote Direct Memory Access
RoCE	RDMA over Converged Ethernet
WR	Work Request
NIC	Network Interface Card
RNIC	RDMA enabled NIC
GLB	Global Load Balancing
DIT	Dynamic Independent Task
GPI	Global address Programming Interface
GASPI	Global Address Space Programming Interface
UTS	Unbalanced Tree Search

1 Introduction

Modern compute clusters are used to tackle problems from various fields of research like physics, medicine and artificial intelligence. Many of these problems are highly irregular and do not allow for predetermined scheduling. Instead, the workloads have to be balanced at runtime to ensure optimal usage of the available compute resources.

The above-mentioned clusters usually utilize a hybrid approach, where each compute node in the distributed system is in itself a shared memory machine. In addition, specialized accelerators and networking technologies have been introduced. Among them are high-speed interconnects like InfiniBand (IB) or RDMA over Converged Ethernet (RoCE). These technologies allow for Remote Direct Memory Access (RDMA) and one-sided communication, i.e. data transfers managed purely by the sender without involvement of the target CPU. While these technologies require careful design to fully utilize their capabilities, they allow for an overlap of communication and computation and vastly reduced latency. In such clusters, programmers can leverage both shared and distributed memory programming to accelerate their computations. The resulting complexities, however, also put a great burden on programmers.

A programming model that alleviates this burden is the Partitioned Global Address Space (PGAS) model. It provides programmers with a global address space shared between all processes (called places) of a distributed system, while still distinguishing remote and local memory operations. An extension of PGAS called Asynchronous Partitioned Global Address Space (APGAS) enhances the model by introducing activities, i.e. lightweight threads of execution that may be executed asynchronously on any place [50].

One common way to fully utilize the above-mentioned architectures is to partition problems into smaller sub-problems, which are expressed as *tasks*. Here, a task denotes a discrete transferable unit of work that can be executed in parallel to other tasks.

In shared memory systems, work stealing is a popular technique to schedule tasks and balance the load between processors. Unfortunately, this approach can be inefficient in a distributed memory setting. Primary reasons for inefficiencies are the increased cost of communication via an interconnect, as well as the larger number of processors prevalent in distributed systems.

Saraswat et al. propose a suitable extension of work stealing to distributed systems with their Global Load Balancing (GLB) framework [44]. An initial implementation of GLB is provided by the GLB library written for X10 [54]. Since then, several variants of the GLB technique were introduced to improve upon the initial implementation. Recently, one variant with significantly increased performance was proposed by Reitz et al. [42]. So far, implementations of the GLB technique utilize active messages provided by the APGAS model. However, research shows that the use of data-driven RDMA communication instead of active messages might be able to reduce the latency of such communications [7].

This thesis makes the following contributions: A variant of the GLB technique using exclusively RDMA and one-sided communication was conceptualized. Then, a new prototypical framework for GLB named Tasks for GPI (TasGPI) was implemented using the RDMA focused PGAS library GPI-2 [18]. Other than the utilization of RDMA for communication, TasGPI attempts to stay as faithful to the reference variant by Reitz et al. as possible. The framework was then benchmarked against the reference variant in order to evaluate, if RDMA and one-sided communication can be used to further enhance the performance of the GLB technique.

Given GPI-2's lack of support of APGAS's activity concept, several elements of the GLB technique had to be redesigned. This encompasses critical elements such as termination detection. Further, to facilitate extensibility, system-specific optimiza-

tions, and future evaluations, the framework was designed in a modular manner. This was achieved by extracting core parts (e.g. steal-behaviour) of the worker logic into strategies, i.e. replaceable classes implementing the required behaviour.

TasGPI's performance for regular and irregular workloads was experimentally evaluated via two benchmarks. Unfortunately, technical challenges prevented the utilization of RDMA in the experiments. Therefore, a suitable workaround was found in the form of an ethernet wrapper provided by GPI. It simulates RDMA over TCP, suggesting the issues lie with the Global address Programming Interface (GPI) library or RDMA hardware. This way, the operational functionality and correctness of the implementation could be established and the benchmarks executed. However, at the time of writing, we are unable to definitely attribute the achieved results to the quality of the implementation, the emulation of RDMA over TCP, or the compatibility of RDMA with GLB.

The experiments show near-linear speedups akin to the reference variant, particularly when dealing with larger problem sizes. However, it's worth noting that benchmarks involving smaller problem sizes yielded less favorable outcomes.

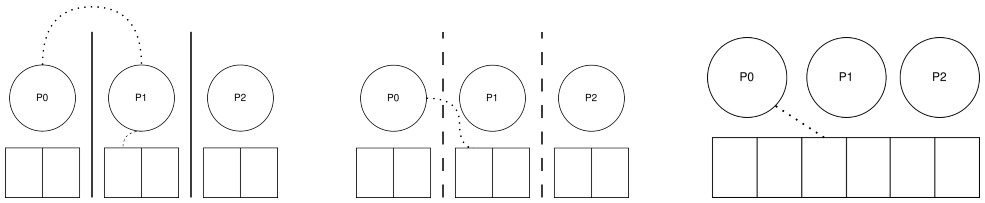
This thesis commences with the essential background knowledge in Chapter 2. Following this, Chapter 3 outlines the fundamental ideas and design decisions of TasGPI. Chapter 4 then details the actual implementation of TasGPI. The thesis continues by providing a brief overview of the experiments conducted to validate the correctness and assess the performance of TasGPI, and discusses the obtained results in Chapter 5. After that, Chapter 6 gives a brief overview of related work. The thesis concludes by summarizing the key findings and prospective future work in Chapter 7.

2 Background

This chapter contains the technical background necessary to understand this thesis. First, the PGAS model and its extension APGAS are described in Section 2.1 and Section 2.2, respectively. Following that, Section 2.3 describes the principles of RDMA. Then, a detailed description of the Global address Programming Interface (GPI) is given in Section 2.4. Finally, the GLB technique itself is described in Section 2.5.

2.1 The PGAS model

The Partitioned Global Address Space (PGAS) model allows programmers to access memory via a shared global address space similar to shared memory programming. The physical memory, however, is not unified as is the case in shared memory.
Instead, every process (called *place* or *rank*) has some local memory, that is interconnected with other places memory. Usually, one place is mapped to one compute node in the Single Program Multiple Data (SPMD) model, but in most implementations other configurations (e.g. one place per core or diverging code paths) are possible [13].



(a) Message Passing (b) PGAS (c) Shared Memory

Figure 2.1: Simplified illustration of the memory models. The PGAS model may be semantically put between the message passing and shared memory models.
Adapted from [13].

As shown in Figure 2.1, the PGAS model may be put conceptually between the *message passing* and *shared memory* models. The bars between processes and memory may be interpreted as their *distance*, i.e. their cost of communication.
Message passing (Figure 2.1(a)) only allows explicit messages between processes, represented by the solid bars. Direct access to the targets memory is impossible, instead the receiving place has to process a message and modify its memory itself.
The shared memory approach (Figure 2.1(c)), on the other hand, has no bars, representing the free and equal access of the processes to the shared memory.
In PGAS, processes may directly access another place's memory as depicted in Figure 2.1(b). This still incurs the increased cost caused by the network interconnect, but does not necessarily involve the target process. In fact, modern communication techniques like RDMA (compare Section 2.3) may be used to bypass the process completely, thus alleviating the need for unnecessary synchronization between sender and receiver [9]. Thus, the details of communication are hidden from the programmer whilst keeping the varying costs for local and remote memory access transparent to the developer.

2.2 APGAS

The APGAS model extends the PGAS model described in Section 2.1 by introducing the concept of *activities* and the needed constructs to manage those [45]. For the sake of brevity, the specifics of constructs are omitted from this description.
The APGAS model was first introduced in the X10 programming language [51]. An implementation of the APGAS model for Java 8 is also available as part of the X10 project [2].
An `activity` is a piece of code, that may be executed on a place. Every program in the APGAS model has at least one activity, called root activity. Once the root activity terminates, the program terminates as well.
Each activity may launch new activities locally or remotely on other places. Activities spawned that way, may access immutable variables of the enclosing scope. To that

end, developers may use the `async` construct. The construct spawns the desired activity and immediately returns. Once launched that way, the activity remains on the place for its lifetime.

Different implementations diverge in the access capabilities of activities. The X10 implementation, for example, only allows local memory access for activities. However, programmers may emulate global access behaviour in X10 by launching activities at the target place that write to the desired location, thus not violating the restriction.

To enable developers control over the flow of execution, the APGAS model also introduces the `finish` construct. The construct allows programmers to wait until an activity (and activities spawned by it recursively) has terminated. The construct thus can be used for *distributed termination detection*, i.e. it may be used to detect that no work, i.e. activities, remains in the system [45].

To further enhance the ability of programmers to synchronize activities, the APGAS model also introduces the `atomic` construct. Atomic blocks are executed as if they were only one step and no other activity was running. In addition, the APGAS model also supports guarded atomics. Here, the atomic block is associated with an atomic expression. The runtime blocks, until the expression evaluates to true, in which case the atomic block is executed. As with traditional mutual exclusion, the critical section should remain as small as possible.

In summary, the APGAS model provides programmers with powerful tools to express concurrency in addition to the convenient memory view provided by PGAS.

2.3 RDMA

Remote Direct Memory Access (RDMA) is a networking technique, that allows direct memory access between remote systems in a network, without involving the CPU or operating system of the recipient. This so-called *kernel bypass* allows computation and communication to overlap. In addition, RDMA allows zero-copy data transfer, i.e. the need for intermediary communication buffers is eliminated [30, 15].

Since RDMA is a very complex topic, this section gives only a shallow introduction to the relevant aspects of the technique. A comprehensive overview of RDMA and the surrounding concepts and technologies may be found at "RDMA protocol verbs specification" [23].

To use RDMA, nodes must be equipped with special Network Interface Cards (NICs) called RDMA enabled NIC (RNIC). To facilitate communication, the application registers memory buffers with involved RNICs. Data can then be transferred between such buffers without additional copies [15, 30].

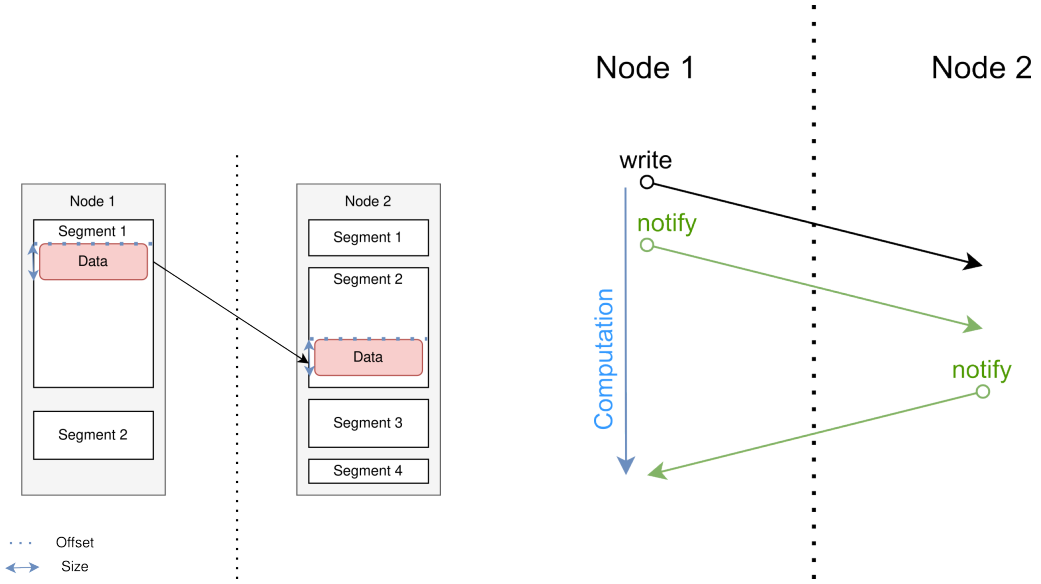
To transfer data, the application creates a so-called *Work Request (WR)* [23, p.126ff]. The WR is posted to a *Work Queue* on the RNIC. At this point, control returns to the application and the transfer is handled entirely by the RNICs of sender and receiver, allowing for the aforementioned overlap of computation and communication [15]. This kind of communication is called *one-sided*, since the application layer of only one side, i.e. the sender, is involved in the exchange [15].

In addition, RDMA has very low network-latency. Mellanox IB adapters, for instance, can achieve latencies under 1 µs [25]. This makes RDMA a very suitable technology for latency sensitive messages.

While the advantages described above can be a great benefit for distributed programs, RDMA communication can be quite complex. Programmers are presented with a variety of options, that may lead to degraded performance, if used incorrectly [15, 30]. In addition, the one-sided nature of communication means that the remote side is not notified once a data exchange completes. Instead, the application must implement a suitable synchronization mechanism [15].

2.4 GPI-2

GPI-2 is an implementation of the PGAS standard Global Address Space Programming Interface (GASPI) [12, 20]. For the sake of readability, GPI-2 will be referred to as GPI for the rest of this thesis. GPI is a fully thread-safe library available for



(a) A data exchange using *place id*, *segment id* and *offset*. See Section 2.4.1.

(b) A complete message exchange using weak synchronization. See Section 2.4.3.

Figure 2.2: An exemplary data exchange using `gaspi_write`. Adapted from [46].

C/C++ and Fortran and was developed at the Fraunhofer Institute for Industrial Mathematics (ITWM). It is open source and available on GitHub [17].

The main focus of the library lies on efficient one-sided communication using RDMA. This way, GPI can provide true asynchronous and one-sided data exchanges as described in Section 2.3. It aims to replace the synchronous two-sided approach implemented by message passing, as implemented by the Message Passing Interface (MPI) [20]. Like MPI, GPI supports both SPMD and Multiple Program Multiple Data (MPMD) style executions [12, p.18].

The following subsections give a brief but detailed overview over the most important concepts of the library.

2.4.1 Segments

GPI implements the PGAS model by allowing programmers to register partitions of memory called *segments*. Every segment is assigned a unique id, that can be used to identify it.

The memory of a segment is pinned as a requirement for the RDMA and does not follow any specific memory model. Instead, the programmer registering the segment decides the purpose, size and (if desired) physical location of the underlying memory. Accordingly, a segment is a contiguous block of memory with a specified address and size [20].

Access to the segments may be made by direct memory manipulation locally, or via the communication routines provided by GPI remotely. As depicted in Figure 2.2(a), the latter uses a triple of the *place id*, *segment id* and an *offset* as well as the size of the data to be read/written, to identify the local and remote memory locations [12, p.57].

2.4.2 One-sided communication

Most communication routines of GPI utilize the segments described in Section 2.4.1. The most common variant of communication provided by GPI are one-sided reads and writes. Those are provided in their most basic form via `gaspi_read` and `gaspi_write`, which can be used to directly read from or write to a remote segment. In case of a read, the results are written directly to a local segment [12, p.57ff].

The requests are asynchronous and non-blocking, meaning the call may return before the communication actually finished. This way, the process may continue computations, while the underlying infrastructure handles the communication [12, p.57] as seen in Figure 2.2(b). The weak synchronization via notifications also depicted here is described in Subsection 2.4.3.

GASPI also provides the `gaspi_read_list` and `gaspi_write_list` routines, which are semantically equivalent to multiple calls to the respective singular counterparts, but may be more efficient, depending on the implementation [12, p.79, p.85].

Internally, GPI inserts such requests into a *message queue* specified by the programmer. Multiple such queues may be created and can be differentiated by their ID. They may be used by the programmer to distinguish messages by their purpose to implement separation of concerns. All entries in such a queue are synchronized with

each other, but multiple queues are independent of each other. While this guarantees that messages in a queue are posted to the network in order, there is no general guarantee, that the messages also are processed by the network or arrive at the target in order [12, p.11, p.57f]. An exception are notifications, which will be described in Section 2.4.3.

GASPI guarantees fairness between queues, meaning no queue will be delayed infinitely. However, queues are limited in size [46]. Once a queue is full, the process must wait for it to be emptied again. To that end, GPI introduces the `gaspi_wait` routine, which blocks, until a given message queue has been processed. Once this routine returns, all requests in the queue have been processed by the network. Importantly, this does not mean that the communication has finished. It rather means, that the request has been put on the network and may not have been received by the recipient yet [12, p.65].

Accordingly, one-sided communication is purely sender-initiated and managed by the local place. While this means, that the communication does not disturb the computation of the recipient, it also means, that the receiver (as well as the sender) have no knowledge of the state of the communication. For instance, the receiver has no knowledge, if relevant data has been received yet [20].

2.4.3 Weak synchronization

To resolve the issue addressed in Section 2.4.2, i.e. that the receiver has no knowledge of incoming communications, GPI introduces *weak synchronization primitives* in the form of *notifications*. The notification procedure is also a form of one-sided communication and as such managed only by the local place. Different to read and write operations however, a notification is guaranteed to be non-overtaking, i.e. the notification is guaranteed to complete only after all previous writes were completed. However, to guarantee this ordering, a notification has to be sent via the same queue the corresponding one-sided communication requests have been posted to [12, p.69].

Notifications in GPI are bound to specific segments and are consequently identified by the triple of *place id*, *segment id* and *notification id*[1]. In GPI, each segment provides up to 65536 (16 Bit) notification ids. A notification itself is a 64 Bit unsigned integer.

The value 0, however, is reserved and may not be used as a notification value [12, p.70]. Instead, it signifies that no notification has been received for the given notification id. A process may send an isolated notification via the `gaspi_notify` procedure or as part of an extended communication call in form of the `gaspi_write_notify` or `gaspi_write_list_notify` procedures [12, p.78f].

The receiver of a notification may wait for any number of consecutive notification IDs via the `gaspi_notify_waitsome` routine [12, p.71]. The routine waits, until at least one of the specified notifications have been received and provides the first such ID encountered. Multiple threads may wait for the same notification. To prevent multiple threads from erroneously processing the same notification ID, the value of the notification is retrieved via the `gaspi_notify_reset` routine. The routine atomically resets the notification to 0 and returns the notifications value. Consequently, only one thread retrieves the correct value, while later threads retrieve the value 0, thus knowing that the notification is already being processed [12, p.73f].

Figure 2.2(b) depicts an exemplary data exchange between two places. Node1 sends data via `gaspi_write`. Node2, on the other hand, is expecting some data from Node1. Since it has no way to detect if Node1 has already sent the required data, it instead uses the routine `gaspi_wait_some` to wait for a pre-specified notification. Once the notification has been received, the process can be sure that the data is written and valid. It then sends a notification back to Node1 to notify it of the receipt.

In contrast to, e.g., MPI, Node1 does not need to wait for the communication to finish. Instead, it sends the data and following notification and resumes the computation. It later checks, if the acknowledgement has been sent by Node2. This way, the latency of the communication can be hidden. While other technologies also allow for this

[1] Here, GPI differs from GASPI, which identifies notifications by the tuple of *place id* and *notification id* [12, p. 70].

overlap, RDMA has the distinct advantage of circumventing the receiver's CPU entirely in this process.

2.4.4 Collective communication

In addition to the one-sided communication procedures described in Section 2.4.2, GPI also supports *collective* communication. Collective procedures involve a subset of all places called *group*. All places are part of *GASPI_GROUP_ALL*, but additional groups may be added by the programmer [12, p.11].

GPI provides a global barrier in form of `gaspi_barrier` and a group-wide reduction via `gaspi_allreduce`. The latter provides implementations for the minimum, maximum and sum, but user provided reductions are possible via `gaspi_allreduce_user` [12, p.102ff].

In contrast to one-sided communication, the memory buffers used by the collective reduction procedures are not required to be part of the global address space. Instead, they are copied to internal buffers [12, p.107].

Consequently, only one collective operation of the same kind may be active per group, i.e. a group may invoke a barrier and a reduce procedure, but not two reductions at the same time [12, p.102]. Notably, collective operations are not assigned to a communication queue and thus synchronized independently of other communications [12, p.14].

2.4.5 Atomic operations

GPI further provides two atomic operations. Those are guaranteed to be fair, meaning no atomic operation will be delayed infinitely. They are also guaranteed to be executed free of interference by other processes, thus preventing data corruption by concurrent access [12, p.96]. Both atomic operations are restricted to operate on `gaspi_atomic_value_t`[2].

[2]GPI defines this as unsigned long. The prevalent RDMA fabric InfiniBand, for instance, guarantees this to be a 64bit unsigned integer [3, p.263].

`gaspi_atomic_fetch_add` allows programmers to atomically add a value to an existing value, identified as usual by the triple of *place id*, *segment id* and *offset*. After completion, the procedure also returns the previous value [12, p.96].
`gaspi_atomic_compare_swap`, on the other hand, atomically compares an existing value with a programmer specified value called *comparator*. If the values are equal, a new programmer specified value is written to the location. Otherwise, no further action is taken. Like `gaspi_atomic_fetch_add`, the procedure returns the value before the procedure was executed [12, p.98].
The atomic operations allow data-driven synchronization between processes. The *compare-and-swap* operation, for example, is an intuitive fit for the implementation of a global lock [12, p.100].

2.4.6 Fault tolerance

To allow for fault-tolerant code, GPI utilizes timeouts for all of its potentially blocking operations. This way, no procedure may block forever in case of an error [12, p.13]. Timeouts are specified in milliseconds, but two special values are available.
Calls using `GASPI_BLOCK` block until the execution has completed with a success or error. Since not all errors are detectable by all operations, procedures called with `GASPI_BLOCK` may block indefinitely. The other special constant is provided with `GASPI_TEST`, which blocks for the shortest possible amount of time before returning [12, p.13].
All GASPI procedures consequently return one of the following values [12, p.14]:

GASPI_SUCCESS The procedure completed successfully.

GASPI_TIMEOUT The procedure could not be completed in the given timeframe.

GASPI_ERROR The procedure completed with an error.

GASPI_QUEUE_FULL The communication request could not be posted due to a full communication queue and must be submitted again.

In addition GPI provides the concept of a *health vector*. The health vector is a collection of states for all processes. Each process may be either *healthy* or *corrupt*. Each process maintains its own health vector, which is updated when a local remote operation detects an error or consequent recovery [12, p.33f]. Since the state vector is a purely local concept, other processes may not recognize an error until communication is attempted with a failed process.

2.5 Global Load Balancing

Global Load Balancing is a technique for dynamic load balancing using a lifeline-based cooperative work stealing approach [44]. It is able to handle both static and irregular workloads efficiently while also hiding the intricacies of distributed programming from the programmers [54].
This section first characterizes the task model supported by GLB in Subsection 2.5.1. Following that, the lifeline graph underlying the algorithm is detailed in Subsection 2.5.2. The chapter continues with a description of the GLB technique in Subsection 2.5.3 and concludes with an exploration of a recent multi-worker variant in Subsection 2.5.4.

2.5.1 Dynamic Independent Tasks

A task is a discrete transferable unit of work that can be executed in parallel to other tasks. In general, this work may depend on the results of other tasks or some external state. However, the GLB technique imposes some restrictions on tasks in the system [54]:

- A task may access an immutable reference state common to all tasks.
- A task may be processed by any compute unit.
- Tasks are side effect free.
- A task may generate zero or more child tasks.

- Tasks generate a result of a specific, unchanging type.
- Tasks are deterministic, as they only depend on immutable state.
- The results of tasks can be reduced by a user-specified commutative and associative reduction operator.[3]

Tasks adhering to those restrictions are called Dynamic Independent Task (DIT) in accordance with other literature [36, 42].
While they may be more restricted than arbitrary tasks, they are simpler to schedule, since no dependency graph or similar is needed. This way, the load balancing algorithm can be kept simpler while taking burden off the programmers that would otherwise need to keep possible side effects and dependencies in mind.
Variants supporting other task models like the nested fork-join model exist, but are not further explored in this thesis [42].

2.5.2 Lifeline Graphs

In their paper "Lifeline-based Global Load Balancing" Saraswat et al. introduce the concept of a lifeline graph [44].
A lifeline graph is a directed and connected graph with N nodes representing the workers in the system. The graph has a low diameter and the out-degree of each vertex is bounded by a parameter z defined by the user. In such a graph the edge between two vertices is called *lifeline*, making the connected vertices *lifelines buddies*. The low diameter of the lifeline graph ensures, that work can be disseminated quickly in the system, even if little work was initially available. Likewise, the low degree of each vertex bounds the outgoing lifeline steals (described in section 2.5.3) of the respective worker. Finally, the properties of the lifeline graph ensure, that an idle worker can always receive new work, as long as work is left in the system.
A class of graphs satisfying those restrictions are *cyclical hypercubes*. Such a graph can be constructed by enumerating the nodes in the system with z-digit integers of

[3] This restriction allows tasks to be processed independently in any order.

base h with arbitrary h and z, such that $h^{z-1} < N \leq h^z$. A lifeline exists between two nodes if the manhattan distance of their indices (in modulo h arithmetic) equals 1.

2.5.3 The GLB algorithm

This section describes the initial X10 implementation of the GLB approach proposed by Zhang et al. [54].

The library requires the user to provide two sequential pieces of code: A `TaskQueue` implementing the actual computation and a reduction operation, as well as a `TaskBag` implementing a container for pending tasks. On each place exactly one worker activity may exist with its own `Queue` and `Bag`.

At the beginning of the computation, work is distributed among the workers. If only one task is available or no direct mapping to the places in the system is possible, a dynamic start is performed, meaning that the initial tasks are given to the worker at place 0. Otherwise, the tasks are distributed statically among all workers in the system. The decision regarding the initial distribution of tasks is made by the user.

Following initialization, each worker processes n tasks until it runs out of work. Once a worker runs out of tasks to process, it starts stealing. In contrast to traditional work stealing approaches, where idle workers continuously attempt to steal from random victims, stealing in GLB happens in two stages.

In the first stage the worker attempts to steal from up to w random victims. If no work could be obtained that way, the worker attempts to steal from its *lifeline buddies* in the second round of stealing. In contrast to random steals, a victim of a lifeline steal stores failed steal attempts. Once such a victim obtains new work, it will also send tasks to lifeline buddies stored in that way. In both cases, thieves wait for the result of the steal request before proceeding.

Between each round of processing, workers respond to pending steal requests by splitting off work from their Bag and sending the resulting *loot* to the thief until either no tasks, or no more steal requests are left. If unsatisfiable steal requests remain, thieves are notified of the fact with a failure message.

If no work could be obtained in either stealing stage, the worker's activity ends and can only be reactivated by receiving work from a lifleline buddy. Once all worker activities on all places have ceased, the algorithm terminates and reduces the result on place 0. GLB detects termination by utilizing the `finish` construct described in Section 2.2.

The initial proposal operates under the assumption, that at most one activity can run per place, thereby restricting the presence of multiple workers at a single place. While this approach greatly simplifies synchronization requirements, it also limits the potential for leveraging locality to enhance the algorithms performance and scalability.

2.5.4 Multi-Worker GLB

As described in Section 2.5.3, the original GLB implementation is limited to one worker per place. *Multi-Worker GLB* extends GLB by removing this restriction. Different variants of multi-worker GLB exist. One of them is described below and serves as the reference implementation for this thesis [42]. For the sake of brevity, this section concentrates on the general workflow instead of the intricacies of the implementation.

Reitz et al. propose a straightforward extension of GLB by removing the worker limitation. Their approach initially made no adjustments to the logic of the workers. Instead, workers are spawned as threads rather than processes. In addition, the lifeline graph is extended to include each worker as a node, rather than the places. Each worker is then treated as a separate potential victim for random and lifeline steals [42].

Since local workers (i.e. workers residing on the same place) are aware of the other workers at their place, they may directly communicate with each other. This way, the workers can exploit locality by forgoing network communication and preventing unnecessary work like serialization.

The approach further introduces the concept of a *coordinator*. The coordinator serves to simplify the implementation by handling the communication between workers. For remote workers, the exchange is serialized and put on the network, while local communication is able to forgo the network entirely [42].

To further exploit locality, the approach proposes two optimizations. First, idle workers initially steal from the most busy local worker, if possible. Second, the coordinator redirects incoming steal requests to the most busy local worker [42].

Another multi-worker variant called hybrid multi-worker GLB combining work stealing and work sharing (briefly noted in Chapter 6) exists [53]. Reitz et al. did a comprehensive comparison between their and the hybrid variant and found their own variant to be up to 25.2 % more performant [42].

3 Concepts of TasGPI

As mentioned in Chapter 1, this thesis proposes a new GLB variant exclusively using RDMA and one-sided communication instead of active messages. The concept was then implemented as the prototypical Tasks for GPI (TasGPI) framework. TasGPI utilizes the PGAS library GPI, that provides an interface for one-sided communication, primarily focused on efficient RDMA. The GLB variant proposed by Reitz et al. (compare Subsection 2.5.4) served as a reference implementation.

In general, TasGPI relies exclusively on RDMA and one-sided communication for communication. In contrast, existing implementations use message handlers and active messages provided by the APGAS model to communicate. While also asynchronous, active messages still require the target's CPU to execute the messages.

As described in Section 2.3, RDMA not only has very low latency and zero-copy data transfers, it also circumvents the targets CPU. Thus, RDMA can result in lower latencies and less overhead for communications [7] and consequently shorter times of inactivity for idle workers.

Most communication of the GLB technique occurs at the exchange of steal requests. In TasGPI, each worker maintains a queue of incoming steal requests that may be accessed by all workers concurrently. The implementation exploits one-sided RDMA operations to minimize the network cost and provide fast exchanges of such requests. The queues are implicitly synchronized by using atomic operations on the hardware level, further exploiting the RDMA hardware and one-sided nature of communication.

However, the queue must be carefully designed, since RDMA is a complex technology that presents the programmer with many performance critical decisions [15, 30]. TasGPI's design exploits lower-level design decisions already made by the GPI library.

Notably, GPI aggressively inlines[1] messages if possible, and polls for local completion of Work Requests (WRs) (compare Section 2.3). Both aspects are critical to improve latency on RDMA hardware [5, 30].

Since inlining is only possible for small messages, TasGPI keeps latency-sensitive control messages (e.g. rejection of steal attempts) as small as possible. The transfer of tasks, on the other hand, is bundled into big messages. This way, TasGPI can profit from the low latency for control messages, while still exploiting the high bandwidth provided by RDMA for big data packages.

Furthermore, to utilize the RDMA hardware to capacity, the RNIC must be kept busy [5, 30]. To facilitate this, TasGPI does not utilize the coordinator concept introduced by Reitz et al. (compare Subsection 2.5.4). Instead, every worker may directly communicate with any other worker in the system.

In addition, RDMA reads are more expensive than RDMA writes, as the former requires a full round-trip. As such, TasGPI uses RDMA writes in favour of reads, if possible.

Where further synchronization between workers is required, e.g. the rejection of a steal request, TasGPI utilizes the notification concept provided by GPI (compare Subsection 2.4.3). Considering the factors described above, a notification is a single 64 Bit RDMA write and as such can be inlined.

Other than described above, TasGPI attempts to stay as faithful to the reference implementation as possible (compare Subsection 2.5.4). However, the change from APGAS to PGAS and the choice of C++ necessitate further changes to the technique which will be described in the following.

Most notably, the PGAS model does not inherently provide the means to detect distributed termination. While existing variants rely on the `finish` construct provided by APGAS as mentioned in Subsection 2.5.3, TasGPI requires an explicit implementation of termination detection. To that end, TasGPI introduces the `TerminationStrategy`, which is required to implement the desired behaviour. Sim-

[1] Small messages are stored directly in the RNIC instead of the host memory, resulting in lower latency [30].

ilar to the `finish` construct of X10, TasGPI uses a counter-based termination scheme [48].

In shared memory settings, the `TerminationStrategy` simply counts the number of idle workers, signaling termination, once all workers are idle. For distributed memory settings on the other hand, the `TerminationStrategy` instead counts idle places, consequently signaling termination once all places are idle. To keep communication low at this point, TasGPI differentiates between local (per place) and global termination, i.e. a place only communicates with other places, if all local workers are currently idle.

Furthermore, C++ does not provide a standardized approach to data serialization. Instead, the user's `TaskQueue` assumes the responsibility of implementing procedures to serialize data into and deserialize data from a given byte buffer. To prevent redundant copies of the tasks, TasGPI ensures that the serialized data is directly written into a GPI segment. Since this approach eliminates the need for direct access to the tasks, the framework doesn't require the user to define a `TaskBag`. Consequently, users may store their tasks according to their preferences, e.g. by directly storing them in the `TaskQueue`.

Last but not least, TasGPI introduces more flexibility and possibilities for system-specific optimizations. To that effect, TasGPI extracts termination detection, steal behavior and the creation of the lifeline graph into separate strategies (compare Figure 4.2), i.e. exchangeable implementations of the respective algorithms [19]. This way, programmers can exchange sections of the algorithm tailored to their specific needs, without changing the worker logic or other components of TasGPI.

4 Implementation of TasGPI

This chapter describes the actual implementation of TasGPI following the concepts discussed in Chapter 3. First, the usage of TasGPI is described in Section 4.1. Following, Section 4.2 gives a brief overview of TasGPI's most important classes. Section 4.3 then describes the logic of a TasGPI worker in more detail.
Subsequently, Sections 4.4 and 4.5 describe the steal behaviour implemented by TasGPI for distributed and shared memory models respectively. Section 4.6 then briefly describes TasGPI's termination detection algorithms. The chapter closes with a detailed description of the RDMA-based queue datastructure used by TasGPI to facilitate steals in Section 4.7.

4.1 Using TasGPI

TasGPI is a prototypical header-only framework for GLB written in C++ 20. The accompanying source code contains a detailed description of TasGPI's configuration options and compilation instructions in the sub-folder documentation. To configure and compile the project, the well-known build system CMake is used.
Building TasGPI requires the following software:

- libibverbs (v1.1.6 or higher), given TasGPI is configured for InfiniBand
- autoconf (v2.63 or higher), libtool (v2.2 or higher) and automake (v.1.11 or higher)
- GNU Awk and SED utilities
- GNU compiler collection or a compatible compiler[1]

[1]While not documented, GPI relies on some compiler intrinsics like `__sync_fetch_and_add` which necessitates this restriction.

In the scope of this thesis, TasGPI was built with GCC 11 or newer. Older versions supporting C++ 20 or higher should work, but were not tested as of yet. The requirements stated above are directly adopted from the GPI library [17], which is described in Section 2.4.

TasGPI relies on GPI for synchronization and one-sided communication. Should the user require no support for distributed memory, TasGPI can be configured to exclude GPI and network communication entirely. In this case, the dependency on GPI is also removed.

Given TasGPI is build with support for distributed memory, version 1.5.1 (at time of writing, the most recent version) of the GPI library is automatically added to the project. The library is then build, and linked to TasGPI without the need for interaction by the user. In this context, the user is able to configure the utilized network fabric, i.e. ethernet or InfiniBand (default), and the desired job-launcher, i.e. SSH or Slurm (default).

To use TasGPI in applications, it is recommended to add the code into its own sub-folder in the apps directory, and adding an entry into the corresponding CMakeLists.txt. This way, the application has full access to the CMake project and its configuration. TasGPI can also be externally added to existing CMake projects in the usual manner.

The application itself must implement a `TaskQueue` as described in task_queue.md of the documentation, and provide the strategies to be used. Currently, TasGPI provides strategies for both distributed and shared memory systems. The technique is then launched by simply instancing the `Runtime` and calling its `run` method as shown in Listing 4.1.

The `run` method expects a single boolean argument. Given a truthy value, TasGPI performs a dynamic start, i.e. all initial tasks are given to the worker with ID 0. In addition, all other workers start out in the idle state with their lifelines activated. Otherwise, the work is distributed evenly between workers and all workers are active at the start of the computation.

```
1 tasgpi::Runtime<
2     ExamplePiQueue,
3     MultiWorkerHypercubeStrategy,
4     FIFOStealStrategy,
5     LocalTerminationDetectionStrategy
6 > runtime{};
7 runtime.initialize();
8
9 const auto [inside, thrown] = runtime.run(true);
```

Listing 4.1: An exerpt of the MonteCarlo example provided with the accompanying code.

Finally, the compiled application can be run. In shared memory mode, i.e. configuration without GPI, the compiled programs can be directly executed on the host machine. Otherwise, the `gaspi_run` utility [43] must be used, similar to MPI's `mpi_run`, to ensure proper initialization of GPI.

Example scripts to build TasGPI and run experiments are provided in the scripts directory in the accompanying code. Their usage and requirements are detailed in the corresponding readme.md.

4.2 Overview

Broadly speaking, TasGPI consists of three core components: The `Runtime`, `Worker`, and the user-provided `TaskQueue` classes. As described in Chapter 3, these components utilize the `TerminationStrategy`, `LifelineStrategy`, and `StealStrategy` to facilitate user defined algorithms without the need for modification of the framework itself. The following gives a high-level overview over the responsibilities and key differences to the original GLB of those components as depicted by Figure 4.1.

Runtime

The central component of TasGPI is the `Runtime`. It is the only class the user is required to instantiate manually and serves as an entry point to the GLB technique. As seen in Figure 4.1, the `Runtime` expects the types of the user-defined `TaskQueue`,

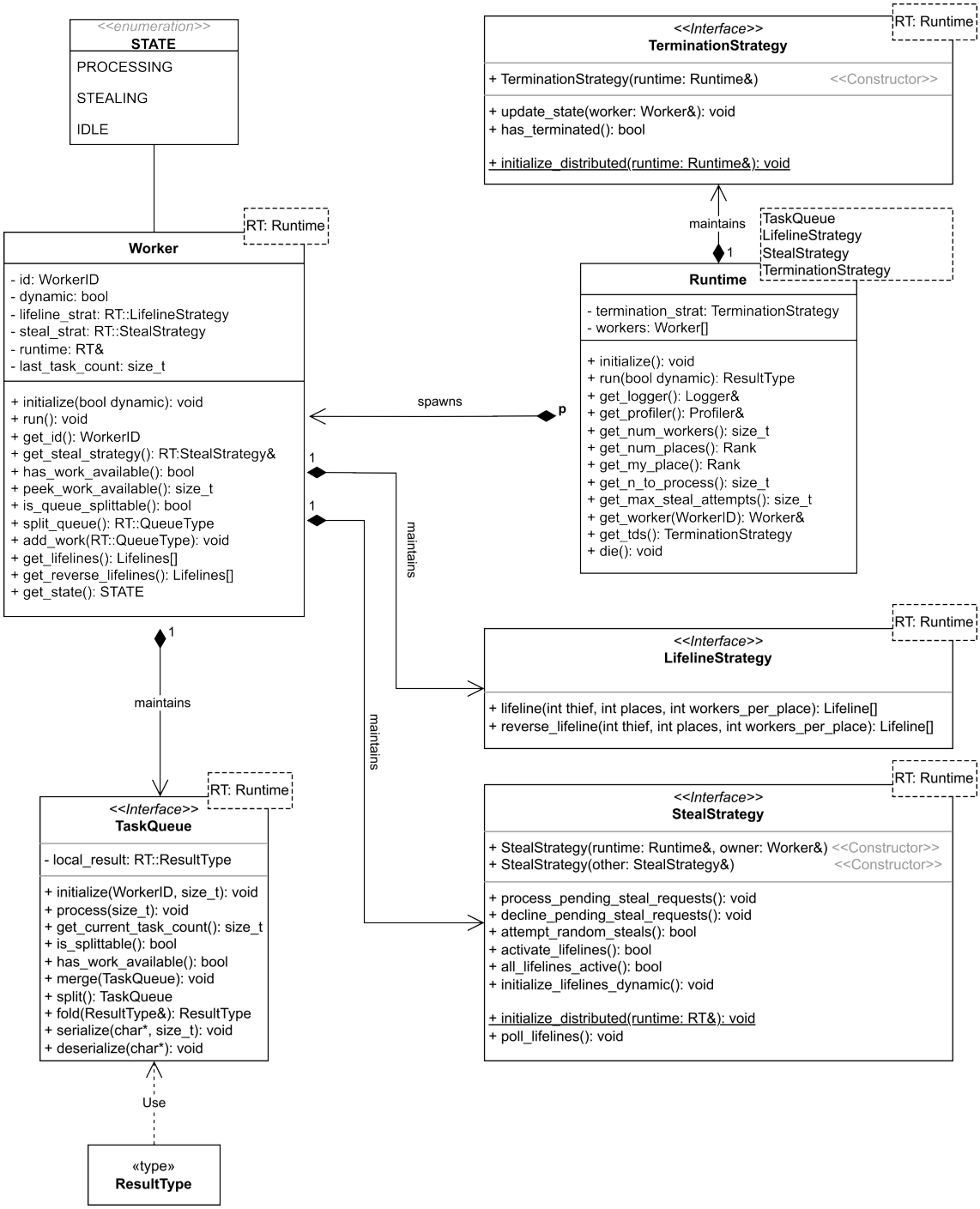


Figure 4.1: Simplified class diagram of TasGPI.

as well as the three strategies, as template parameters. The resulting `Runtime` type is then passed to the other components as their respective template parameters. This way, all components can access the required types via the `Runtime`. This procedure also allows for additional compile time optimizations, for instance, the removal of pointer indirections or the need for virtual tables, that would not be possible by using traditional polymorphism over templates.

The main functions of the `Runtime` are the initialization, coordination, and final teardown of the other core components. In the initialization phase of the technique, the `Runtime` first configures and initializes GPI. At this point, the `Runtime` also is assigned its *place_id* and receives the number of total places from GPI, similar to MPI's `MPI_Comm_size` and `MPI_Comm_rank`. It then utilizes a barrier to ensure that the communication between all places is established and all places are in a valid state before continuing. Once all places entered the barrier, the `Runtime` calls the static `initialize_distributed` methods of the `TerminationStrategy` and `StealStrategy`, which are discussed later in this section. Given TasGPI is configured for shared memory only, the initialization of GPI is skipped and the place ID is always set to 0, while the number of places is set to 1.

The `Runtime` then continues by creating p `Worker` objects and storing them in a `std::vector` for later use. Each worker is identified by a unique ID calculated by $place\_id * p + local\_id$, where *local_id* is the workers index in the local vector of workers.

After the initialization phase, the user can start the GLB technique by calling the `run` method as described in Section 4.1. At this point, the `Runtime` spawns one thread per worker, which runs the `run` method of each worker. The `Runtime` then suspends the main thread until all worker threads terminated, signaling the end of the computation.

Once the computation has finished, each `Runtime` collects the results of all local workers and reduces them via the `TaskQueue's fold` method. The actual result of the computation is then collected via `gaspi_allreduce_user` as described in

Subsection 2.4.4, again using the `fold` method to reduce the partial results. Before returning the final result, GPI is terminated, finalizing the communication between places and freeing all resources, similar to `MPI_Finalize`.
Last but not least, `Runtime` facilitates the direct communication between local components by providing a central interface to query their respective objects. For instance, a `Worker` attempting to steal from another local `Worker` may directly access the `Worker` object via the `Runtime`, forgoing any network communication.

TaskQueue

Every computation using GLB requires an implementation of a `TaskQueue` provided by the user. The class simultaneously serves as the task pool for the computation as well as describing the actual processing of tasks in a sequential manner. As described in Chapter 3, TasGPI attempts to stay as faithful to the reference variant as possible. However, a notable change to the `TaskQueue` is the addition of the `serialize` and `deserialize` methods. Those methods are necessary to transfer tasks over the network, since C++ does not support out-of-the-box serialization of non-trivially-copyable data. Whenever the `StealStrategy` attempts to transfer data via the network, it first splits a portion of the current `TaskQueue` by calling its `split` method. The resulting temporary `TaskQueue` instance contains the split tasks. Usually, half of the current tasks are taken in this process, but TasGPI allows arbitrary proportions.
Since it is usually unsafe to convert arbitrary task representations into bytes, the tasks are instead serialized into a byte buffer. To that end, the `serialize` method is called with a pointer to a memory buffer. This memory buffer usually resides directly inside of a GPI segment to prevent additional copies of the data, but `StealStrategies` are free to use intermediary buffers as needed.
To prevent buffer overflows, the method is also provided with the size of the memory buffer. The implementation is responsible for ensuring that all written tasks fit into the provided buffer. Once the data is serialized, the method returns the number of written bytes. The bytes are then transferred via the network and deserialized

from the received bytes into a temporary `TaskQueue` that can be merged into the thief's `TaskQueue`. While this approach offloads the responsibility for correct and efficient serialization on to the user, it also provides the most flexibility and imposes no further restrictions on the implementation of the `TaskQueue`.

In addition, each `TaskQueue` is required to define a public type alias named `ResultType`. This type is then automatically used by all components of the GLB framework as the type for the intermediate and final results. For instance, the `run` method of the `Runtime` class automatically returns an object of type `ResultType`.

Since results are reduced at the end of computation, the type is currently required to be trivially copyable. Other than that, the user is free to use arbitrary data. The ExamplePiQueue for instance, returns a struct containing the number of simulated points and the number of points contained in the semicircle as exemplified in Listing 4.2.

```
1  struct ExamplePiQueue
2  {
3      struct SimulationResult {
4          std::size_t inside{0};
5          std::size_t thrown{0};
6      };
7      using ResultType = SimulationResult;
8
9      [...]
10 }
```

Listing 4.2: Complex ResultType and type alias defined by the ExamplePiQueue.

Analogous to the original GLB, multiple instances of the `ResultType` must be mergeable. This functionality is provided by the `fold` method of the `TaskQueue`. Consequently, the method must be commutative and associative to guarantee deterministic results, as described in Section 2.5.

LifelineStrategy

The `LifelineStrategy` is responsible for the computation of lifelines and reverse-lifelines, as described in Subsection 2.5.2. To that end, each strategy has to implement

two methods: `lifeline` and `reverse_lifeline`. Both methods accept the ID of the worker, the number of places and the number of workers per place as arguments and return a list of worker IDs in form of a `std::vector`.

The strategy is encouraged to use the user-provided configuration for the lifeline-graph (most notable GLB's `z` parameter), but TasGPI does not enforce this. For instance, the provided implementation `MultiWorkerHypercubeStrategy`, ignores the parameter. It instead calculates the minimum amount of lifeline buddies to span a valid lifeline graph and uses this value.

It is worth noting, that TasGPI does not check the validity of the generated lifeline graph. Instead, the user is responsible to ensure correctness of the used implementation.

Since TasGPI currently supports no malleability or otherwise mutable set of compute nodes, the `LifelineStrategy` currently is designed to work without network communication. This way, the lifeline strategies can be kept simple and easy to implement.

StealStrategy

The `StealStrategy` is an integral part of the TasGPI framework. As discussed in Chapter 3, the `StealStrategy` implements the actual behaviour behind the exchange of steal requests and the exchange of tasks.

As seen in Figure 4.1, the strategy is required to implement methods corresponding to specific points of the GLB technique. For instance, `attempt_random_steals` is required to attempt up to `w` random steals, while `process_pending_steal_requests` must implement the exchange of tasks or rejection of pending steal requests.

In addition, the strategy must implement `initialize_lifelines_dynamic`. The method is only called, if the technique is run with a dynamic start, as discussed earlier. If called, the strategy should initialize all incoming lifelines as if they were previously rejected. An important exception is the worker with ID 0, since this worker initially receives all the work. This way, the initial wave of (probably failing)

steal requests can be prevented, saving CPU time and lowering network congestion. In contrast, the original GLB technique facilitates a similar behaviour by initially only spawning a single worker activity on place 0.

Given the strategy supports distributed memory, it is also required to implement the static `initialize_distributed` method. The method is called after GPI is initialized and is supposed to create any needed components (e.g. segments or communication queues, as described in Section 2.4) or negotiate needed information with remote places. Developers are free to use the whole spectrum of the GPI library. In this context it may be noted that TasGPI strongly encourages the use of GPI, but developers are free to initialize a different communication library and use this one instead, as long as it does not interfere with GPI. As described earlier, a barrier ensures, that all strategies are initialized on all places prior to finalizing the initialization phase.

TasGPI provides two implementations of the `StealStrategy`. For distributed memory, TasGPI implements the `PGASFifoStealStrategy`, which utilizes RDMA and one-sided communication provided by GPI. Its implementation is discussed in Section 4.4.

The `FIFOStealStrategy`, on the other hand, targets shared memory machines and can only be used in shared memory configurations. It facilitates the GLB technique by direct communication between the local `StealStrategies` and is briefly described in Section 4.5.

TerminationStrategy

TasGPI can not rely on the `finish` construct of APGAS, as decsribed in Chapter 3. Instead, the user is required to specify the desired termination behaviour in the form of a `TerminationStrategy`.

In contrast to the other strategies, only one `TerminationStrategy` is instantiated and maintained per place and shared between all local workers. As such, developers must ensure proper synchronization of the implementation via mutual exclu-

sion or other appropriate synchronization primitives. Like the `StealStrategy`, the `TerminationStrategy` is initialized at startup via its static `initialize_distributed` method. Both methods share the same purpose and restrictions described above.

The termination strategy has a simple functional interface consisting of only two methods, as seen in Figure 4.1. The first, `update_state`, is called by the `Worker` class whenever the state of the worker changes or the `add_work` method is called. A reference to the responsible worker is passed to the method via reference.

When called, the `TerminationStrategy` is responsible to collect any necessary data and update its internal state. The other method, `has_terminated`, is periodically called by idle workers to detect global termination (compare Section 4.3). The method must only return true, once no work remains in the system.

Otherwise, the user has total freedom on how the strategy implements the detection. For instance, a simple (yet poorly scaleable) termination detection scheme could utilize a shared global counter of the number of tasks in the system. Once the number reaches 0, termination is achieved. More complex schemes could involve voting trees, token passing and similar mechanisms [31, 32].

While developers are not restricted in their use of network communication in either method, TasGPI encourages to avoid its use in the `has_terminated` method. Instead, network operations should only be performed if the system state actually changed, making `update_state` the better choice.

TasGPI provides two implementations of the `TerminationStrategy`. For distributed termination detection, the `CounterTerminationStrategy` facilitates termination detection by counting idle places. The `LocalTerminationDetectionStrategy`, on the other hand, implements a scheme for shared memory systems and utilizes a similar approach by counting idle workers. Both implementations are briefly discussed in Section 4.6.

Worker

The `Worker` class implements the scaffolding of the GLB algorithm. It implements the workflow of GLB, as described in Section 4.3, but relies on the `Strategies` to implement the actual behaviour of steals and termination detection. Due to its importance, this workflow is explained in detail in Section 4.3.

While APGAS based implementations of GLB realize workers as transient activities (compare Section 2.2), TasGPI creates the desired number of workers at initialization and maintains them until the computation finishes, as described above. As described previously, each worker resides on its own thread until termination. At initialization, each worker creates its own copies of the user-provided `TaskQueue`, `StealStrategy` and `LifelineStrategy`.

First, each worker calculates its own lifelines (and respective reverse-lifelines) as described in Subsection 2.5.2, using the `LifelineStrategy`. To prevent computational overhead, the results are stored in `std::vector` for later retrieval.

Then, the `TaskQueue` is initialized by calling its `initialize` method. Given a dynamic start is performed, i.e. all tasks are initially given to worker 0, each worker also calls the previously discussed `initialize_lifelines_dynamic` method on their `StealStrategy`. Finally, each worker updates its own state at the shared `TerminationStrategy`. At this point, the worker is fully initialized.

During the computation, any worker may be in one of three states: `PROCESSING`, `STEALING`, or `IDLE`. While processing, the worker repeatedly calls the `process` method on its `TaskQueue`, while intermittently processing steal requests, analogous to the original GLB technique. After each iteration of processing, the worker stores the new number of tasks in its `TaskQueue` in an atomic variable that may be accessed by other workers or `Strategies`. This value can then be used as a heuristic for further optimizations, without requiring synchronization of the `TaskQueue`. For instance, a feasible optimization using this heuristic may be implemented by first attempting to steal from the most busy local worker before attempting to steal from random workers as described in Subsection 2.5.4.

Once a worker runs out of tasks, it enters the `STEALING` state. Here, the worker attempts to acquire new work analogous to the original GLB technique described in Section 2.5. Given new tasks were acquired that way, the worker starts processing again.

Otherwise, the worker enters the `IDLE` state. In this state, the worker waits for work from its lifelines or termination of the overall computation. To reduce CPU overhead, workers utilize condition variables to passively wait for the arrival of work. Since this work may never arrive, the worker periodically wakes up to reject any pending steal requests and check if global termination occurred. Should the `StealStrategy` implement the `poll_lifelines` method, it is also called to process received lifelines. The presence of the method is again detected at compile time via the new `concepts` feature of C++.

Other than the scaffolding of the GLB technique, the `Worker` provides synchronized access to its own `TaskQueue`, state and variables. For instance, the `merge` method may be called by the `StealStrategy` to enqueue stolen tasks without fear of corruption due to concurrent access. This way, the `Strategies` and other workers can access any required state without the need for explicit synchronization. Since TasGPI has no knowledge about the kind of access (i.e. read or write access), the implementation utilizes mutual exclusion via `std::unique_lock`. For instance, the worker's `split_queue` method is implemented as shown in Listing 4.3.

```
1 QueueType split_queue() {
2     std::unique_lock lck(m_queue_mutex);
3     return m_queue.split();
4 }
```

Listing 4.3: Mutual exclusion via `std::unique_lock`, exemplified by `split_queue()`

4.3 Worker logic

As mentioned in Chapter 3, TasGPI tries to implement the original worker logic described in Section 2.5 as faithfully as possible. Figure 4.2 depicts the main loop

of a TasGPI worker in the form of a flow chart. In addition, a formulation of the implemented logic in pseudocode can be found in the appendix as Listing A.1.
Similar to the original GLB implementation, the worker's main loop can be roughly separated into three phases corresponding to the worker's current state: The work phase (green), the steal phase (blue) and the termination phase (red). The work and steal phases are largely unchanged. The only difference consists of the extraction of the steal logic into a `StealStrategy`. The termination phase, on the other hand, had to be adapted to work with the PGAS model.
In contrast to APGAS based implementations, the worker thread does not terminate once it activates its lifelines. Due to the lack of the `activity` construct in PGAS, the ability to restart a worker on a remote place would have to be explicitly implemented by the runtime. Whilst not impossible, this would introduce a new layer of communication and as such was deemed unappealing.
Instead, the worker remains active, but reduces idle CPU cycles. This is achieved by passively waiting for the arrival of new work, in which case the worker wakes up and resumes operation as depicted in the termination phase of Figure 4.2. In addition, the worker intermittently wakes to check, if new work has been received via a lifeline buddy. If the steal strategy is not able to directly enqueue work into the workers Queue, it may implement a `poll_lifelines` method. Provided such a method exists, it is automatically called by the worker to allow the strategy to check for progress without necessitating a background thread or similar mechanism. If on the other hand, the `StealStrategy` is able to directly enqueue work into the worker's Queue (e.g. in the case of a shared memory or coordinated implementation), the worker simply checks, if work is already available.
Given work has been received, the worker starts processing again. Otherwise, the worker checks if the `TerminationStrategy` detected quiescence. If the latter is the case, the worker terminates, as no work is left in the system. Once all local workers have terminated in this fashion, the places reduce their local results using `gather_allreduce` and return the final result to the user.

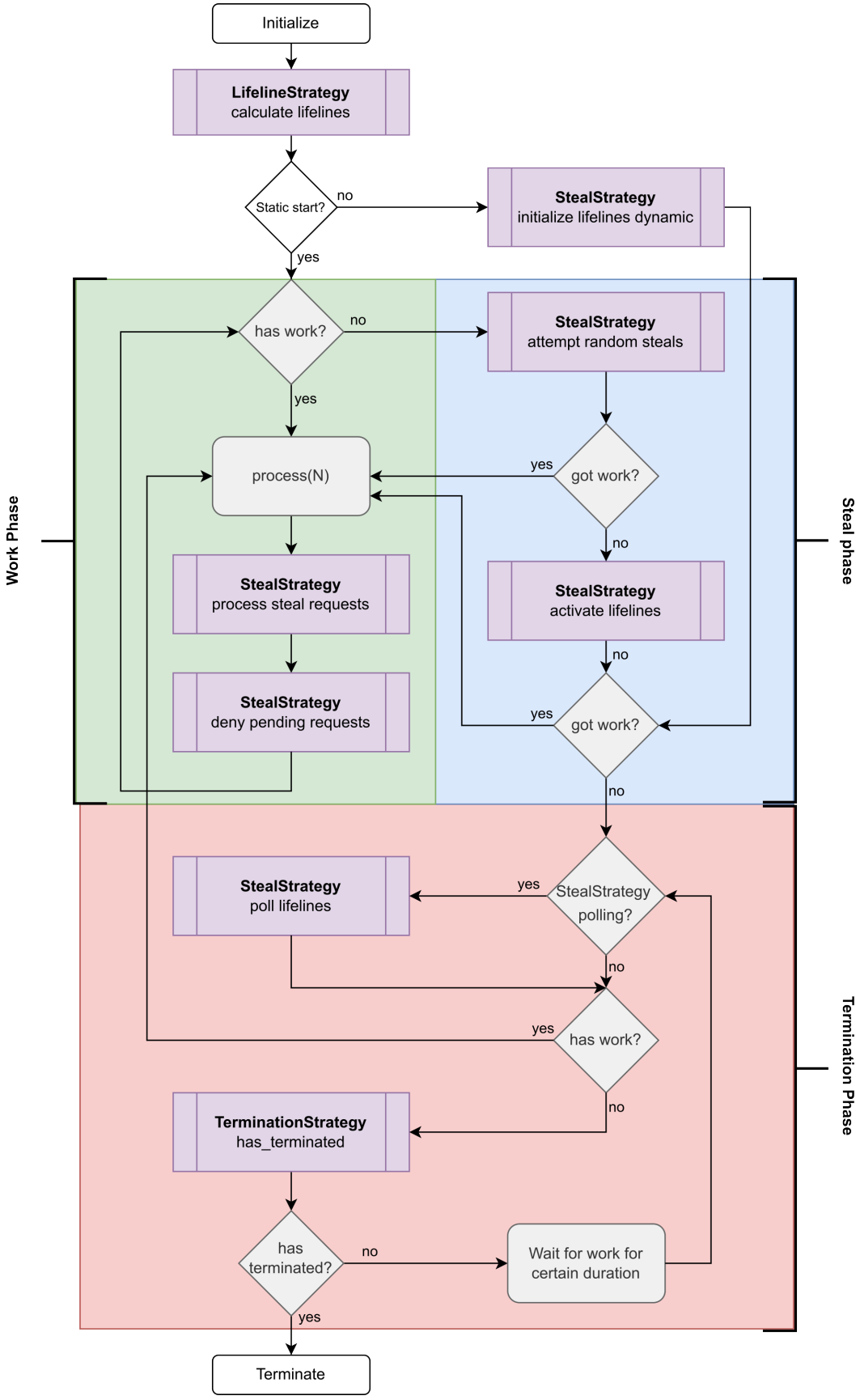


Figure 4.2: Flow diagram of the TasGPI workers main loop. The main loop may be seperated into three phases: Work (green), Stealing (blue) and Termination (red). Calls to strategies are highlighted in purple.

4.4 Steal behaviour in distributed memory

This section initially presents the general procedure implemented by TasGPI for distributed memory systems, which is discussed in Subsection 4.4.1. This general description is subsequently supplemented by further detailing the procedures required by the `StealStrategy` as depicted by Figure 4.2. First, Subsection 4.4.2 describes the process underlying random steals, followed by Subsection 4.4.3 elucidating the processing of steal requests. Subsection 4.4.4 subsequently outlines the rejection of unprocessed steal requests. The section concludes by describing the process of lifeline management in Subsection 4.4.5, with Subsection 4.4.6 characterizing how polling is used to process work received via stored lifelines.

4.4.1 General procedure

Since TasGPI utilizes purely one-sided communication, instead of spawning remote activities, the steal procedure of TasGPI is slightly modified in comparison to APGAS based implementations of GLB, as noted in Chapter 3. To facilitate the GLB technique, TasGPI workers utilize different data structures residing on local or PGAS memory as depicted in Figure 4.3.

Similar to the original GLB technique, TasGPI differentiates between random and lifeline steals, as described in Section 2.5.3. Whenever a worker runs out of tasks, it enters the steal phase (compare Figure 4.2, marked in blue).

Instead of spawning an activity on the victim's place, the thief writes a steal request directly into the victim's memory. To that end, each worker maintains a globally accessible queue on a PGAS memory partition. Due to the one-sided nature of RDMA, thieves directly place their steal requests into this queue, without involving the victim. As seen in Figure 4.3, this queue will be called inquiry queue. Due to its importance, the implementation of this queue is described in detail in Section 4.7.

Each steal request contains the thief's ID, as well as a marker distinguishing random from lifeline steals. This way, only one such queue has to be maintained. Since the inquiry queue can be accessed concurrently by multiple workers, the implementation

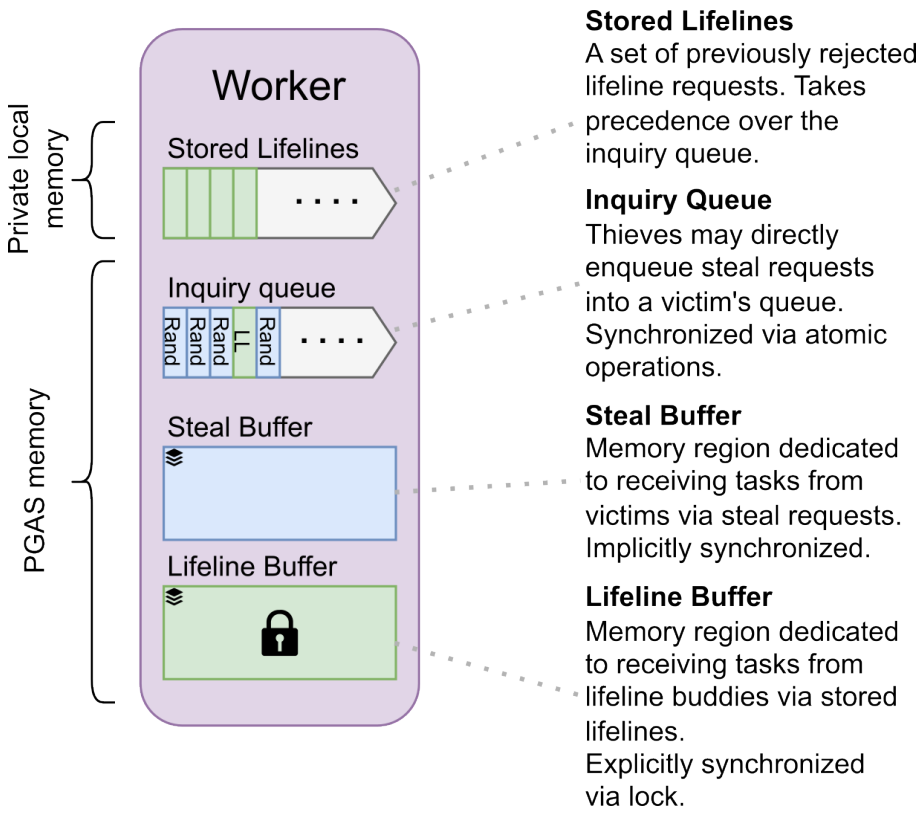


Figure 4.3: Components involved in TasGPIs distributed stealing procedure (simplified).

must be properly synchronized in order to prevent data corruption or loss, as described in Section 4.7.

As in the original GLB, busy workers periodically answer their incoming steal requests. To that end, a victim pops requests from its inquiry queue and transfers a portion of its tasks to the thief, until it has no more work left to share. To transfer tasks between workers, each worker maintains a steal buffer. Each steal buffer is a region of memory on a PGAS partition as depicted in Figure 4.3.

While a worker waits for a steal response (compare Section 2.5.3), it grants temporary ownership of its steal buffer to the victim, i.e. the victim may write to the buffer without fear of interference by other workers. Accordingly, no further synchronization is needed to transfer the tasks to the thief, thus reducing network communication and simplifying the process. A victim then shares work with a thief by simply serializing a portion of its tasks and writing them directly into the thiefs' steal buffer. Once the remote write finished, the waiting thief is notified (compare Subsection 2.4.3) of

the successful steal. By utilizing `gaspi_write_notify`, this process takes only one network operation. The thief then reclaims ownership of its steal buffer, deserializes the contained work from the steal buffer and adds it to its bag.

Once a worker processed all pending steal requests or runs out of work to share, it rejects any remaining steal requests. To that end, it sends rejection messages to all pending thieves. However, the victim also stores rejected lifeline steals, as described in Section 2.5.3. For this purpose, each worker maintains a set of lifelines in stored lifelines, as depicted in Figure 4.3. Since the owning worker manages the set of lifelines exclusively, the data structure resides on local memory. As such, well-known data structures of the shared memory domain may be used and no further synchronization is needed.

Once a worker acquires more tasks, it also shares work with stored lifelines in addition to queued steal requests, as described in Section 2.5.3. In contrast to random and lifeline steals, where steals are processed sequentially, stored lifelines require additional synchronization, since multiple workers may attempt to serve the same stored lifeline concurrently. Each worker therefore also maintains a lifeline buffer in addition to its steal buffer, as depicted in Figure 4.3. However, ownership of the buffer is not transferred implicitly as described above, instead workers attempting to serve work via a stored lifeline need to acquire ownership explicitly. To that end, each lifeline buffer is secured via a lock.

Initially, this lock is acquired by the owning worker. Once the worker runs out of tasks and attempts its lifeline steals, it also releases the lock, potentially granting access to its lifeline buddies. Whenever a worker attempts to serve a stored lifeline, it initially attempts to acquire the thief's lock. Given the lock is acquired, the serialized tasks are transferred via the thief's lifeline buffer and subsequent notification, analogous to random steals. If the lock could not be obtained, however, the worker may safely assume that the lifeline has already been served, i.e. the worker was granted new tasks already. In this case, the victim evicts the stored lifeline until a new lifeline steal was rejected. Finally, an idle worker who is granted new tasks also reacquires

its own lock, signalling its lifeline buddies that no further work is needed at this time. This way, unnecessary transfers of tasks can be prevented.

4.4.2 Attempt random steals

Whenever a worker runs out of work, it initially attempts to steal from w random victims as described in Section 2.5.3. The general workflow of this procedure is depicted by Figure 4.4.

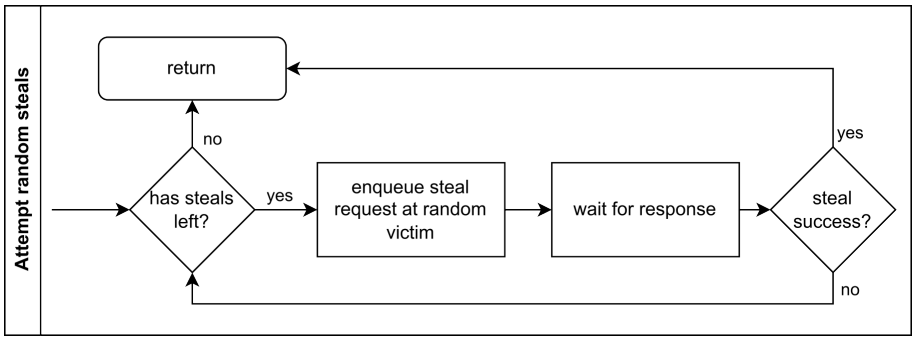


Figure 4.4: Flow chart of TasGPIs random steal procedure.

Initially, the worker generates a random victim ID. It then directly enqueues a random steal request into the victim's inquiry queue. As in the original approach, the worker now waits for an answer in the form of work or a rejection message.
Given work was received, the tasks are deserialized from the steal buffer and added to the worker's queue before returning. If, on the other hand, a rejection message was received, the worker attempts to steal from a new random victim. This process is repeated until work has been received or the worker runs out of steal attempts.

4.4.3 Process pending steal requests

As seen in Figure 4.2, each worker intermittently processes steal requests received from other workers as described in Subsection 4.4.2. The general behaviour of the procedure is depicted in Figure 4.5.
Initially, the worker checks if it has enough tasks to share. Should this not be the case, the routine returns without further processing. Otherwise, the worker processes stored lifeline requests (compare Subsection 4.4.4) and pending steal requests. Since

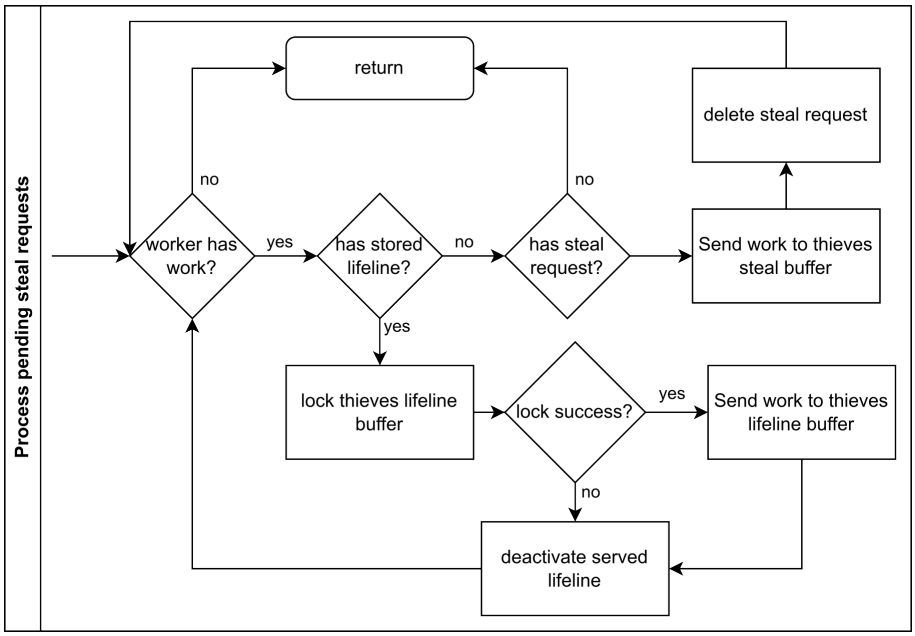


Figure 4.5: Flow chart of TasGPIs procession of pending steal requests.

stored lifelines already exhausted their random steal attempts, such requests are given precedence over random steal requests in an attempt to keep as many workers busy as possible.

To process a stored lifeline, the worker first attempts to lock the thief's lifeline buffer (compare Section 4.4.1). If the lock could not be acquired, the worker has already received work or is currently in the process to do so. In this case, the lifeline request is considered as completed and not further processed. On the one hand, this procedure optimistically assumes that more tasks will be generated by the tasks that were already received by the thief. On the other hand, network communication caused by lifeline requests can be minimized that way, since work is only sent if the thief actually has no tasks.

Alternatively, i.e. if the lock was acquired, the worker splits off a portion of its work and transfers it to the thief via the locked lifeline buffer. In both cases, the worker evicts the stored lifeline from its stored lifelines before returning to the beginning of the procedure.

Provided the worker has no more stored lifelines, it instead handles pending steal requests. While those steal requests may be either lifeline steals or random steals, they are handled effectively in the same way. Analogous to the stored lifelines

described above, the worker splits off a portion of its tasks and sends them to the thief. However, instead of sending work to the lifeline buffer, pending steal requests utilize the steal buffer, which does not require locking, since the thief grants exclusive access to the victim, until it received an answer.

The procedure is repeated until the worker runs out of work to share, or neither stored lifelines nor pending steal requests remain in the worker's inquiry queue, in which case the routine returns.

4.4.4 Reject pending steal requests

After processing its steal requests, a worker rejects any remaining inquiries as depicted by Figure 4.6.

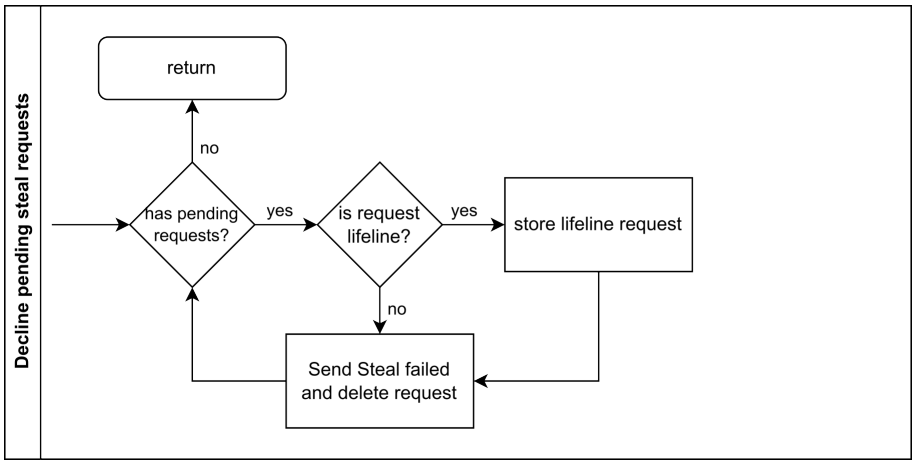


Figure 4.6: Flow chart of TasGPIs rejection of pending steal requests.

To that end, the worker loops over the pending steal requests, sending each thief a rejection message as described in Section 2.5.3. As in the original GLB technique, the worker stores rejected lifeline steals in its stored lifelines. Should the worker receive tasks in the future, it then shares work with such lifeline thieves as described in Subsection 4.4.3. Once all pending requests have been rejected, the routine returns.

4.4.5 Activate lifelines

If an idle worker failed to steal from random victims as described in Subsection 4.4.2, it instead activates its lifelines. To that end, the worker sends lifeline steal requests, analogous to the process described in Section 2.5.3.

As depicted in Figure 4.7, the worker initially unlocks its lifeline buffer.

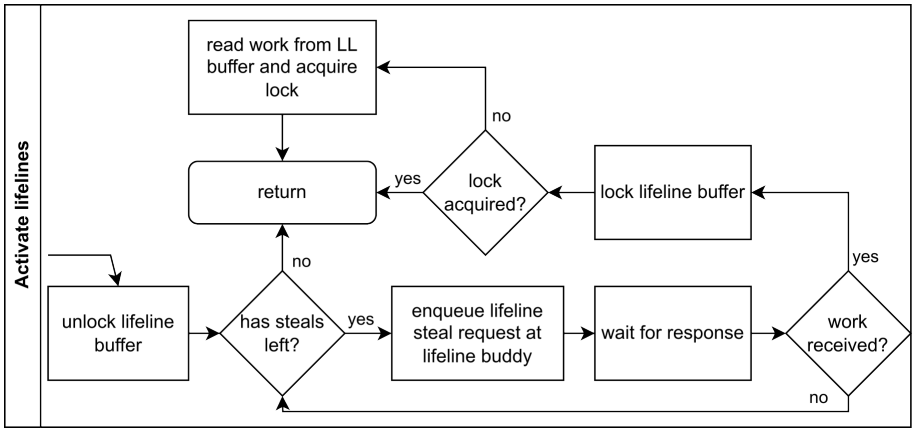


Figure 4.7: Flow chart of TasGPIs lifeline activation procedure.

By doing so, the worker ensures that it can receive work in its lifeline buffer. While the lifeline buffer is only used for stored lifeline steals, since these need additional synchronization as described in Subsection 4.4.1, it is necessary to unlock the lifeline buffer previous to attempting the lifelines steals, to prevent early eviction of stored lifelines. A possible scenario exemplifying this is depicted by Figure 4.8.

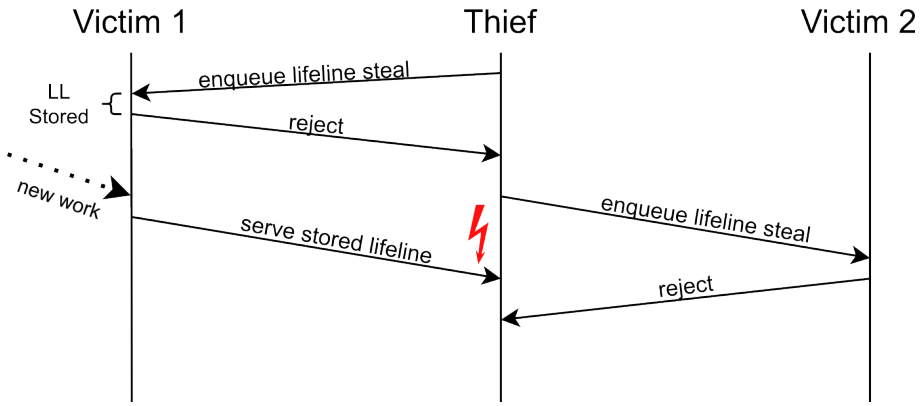


Figure 4.8: Precocious rejection of stored lifeline may occur, if buffer is not unlocked.

Here, the worker Thief has two lifeline buddies, namely Victim 1 and Victim2, and failed to acquire work via random steals. Subsequently, it activates its lifelines by

sequentially requesting work from its lifeline buddies. In this scenario, both Victim 1 and Victim 2 are initially out of work themselves, leading to both victims rejecting the lifeline requests. The issue, marked via a red bolt, arises, once Victim 1 received work from another worker and attempts to serve the stored lifeline to the thief.
Would Thief only unlock its lifeline buffer, once all lifeline steals are finished, Victim 1 would erroneously assume that Thief already received work since the lock could not be acquired, as described in Subsection 4.4.3. Subsequently, Victim 1 would evict the lifeline early, unnecessarily increasing the delay of Thief to acquire new work. In the worst case, all lifelines of a worker could be evicted in that manner, effectively cutting the worker off from new work.
By releasing the lock previous to the lifeline steals, the worker prevents this scenario from happening. However, this also means that a thief may simultaneously receive work from a stored lifeline, as well as from a lifeline steal. Accordingly, a worker who received tasks via a lifeline steal must also check whether work was received via a stored lifeline. Fortunately, this check is implicitly done when the thief reacquires its lock, as described in Subsection 4.4.1.
If the lock could not be reacquired, it means the lock was acquired by another worker, thus a served lifeline was processed. The thief accordingly needs to deserialize the work from the lifeline buffer and add it to its TaskQueue, before proceeding.
Consequently, the worker sequentially attempts lifeline steals to its lifeline buddies only after releasing the lock on its lifeline buffer. As in the original GLB approach, lifeline steals are otherwise functionally equivalent to random steals, except that the victim stores failed lifeline steals as described in Subsection 4.4.4.
Given the worker received new tasks via a lifeline steal, it reacquires the lock of its own lifeline buffer, respecting the exception described above. Otherwise, the lifeline buffer stays unlocked, signaling lifeline buddies that work is still required. The procedure then returns once new tasks were acquired or all lifeline buddies rejected the inquiry.

4.4.6 Poll lifelines

Due to the one-sided communication utilized by TasGPI, an idle worker in the termination phase (See Figure 4.2, marked in red) has to poll for the arrival of new tasks via its lifelines. To that end, TasGPI utilizes notifications (compare 2.4.3), making the check comparatively cheap. The process is quite simple, as depicted by Figure 4.9.

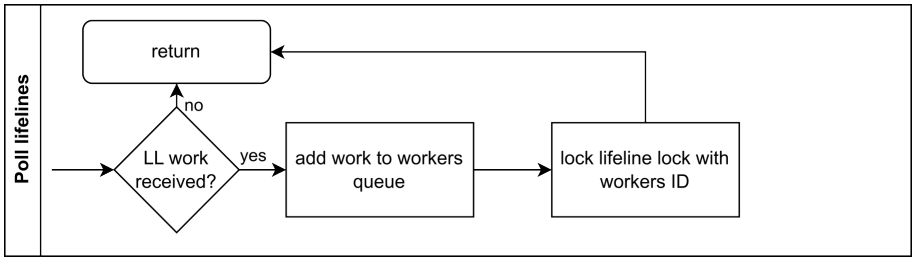


Figure 4.9: Idle workers need to periodically poll the state of their lifeline buffers.

First, the worker has to actively check if lifeline work has been received, as described in Section 4.4.1. To that end, it simply checks whether another worker acquired its lifeline lock. Given the lock was acquired, the worker waits for confirmation that the transfer of tasks was finished. It then deserializes the tasks written to its lifeline buffer and adds them to its TaskQueue. Finally, the routine returns after the worker reacquired its own lock as described in Subsection 4.4.1.

4.4.7 Initialize lifelines dynamic

As mentioned in Section 4.2, this method is used to initialize the `StealStrategy` if a dynamic start is requested by the user. The strategy consequently assumes a state, as if all incoming lifelines had been previously rejected. TasGPI utilizes the pre-calculated reverse lifelines, i.e. the incoming edges of the worker in the lifeline graph, to efficiently store the lifelines without the need for network communication as depicted in Figure 4.10.

Since a dynamic start assigns all initial tasks to the worker with ID 0, lifelines of this worker are consequently ignored.

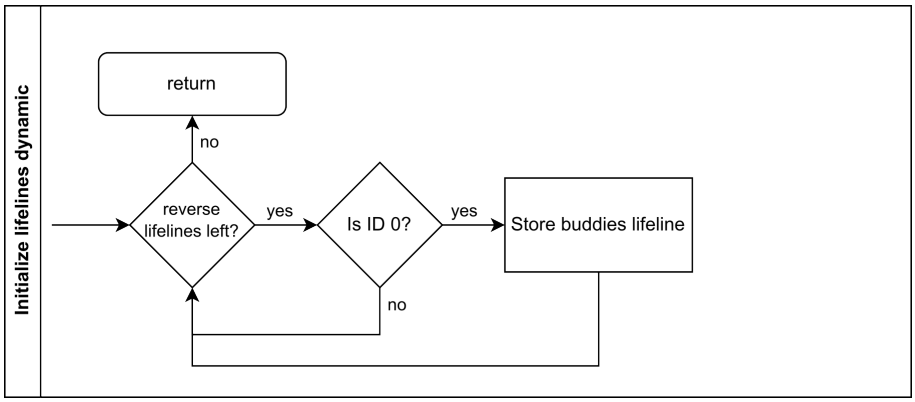


Figure 4.10: Given a dynamic start, initializes the incoming lifelines as if they were previously rejected.

4.5 Steal behaviour in shared memory

This section briefly describes the steal behaviour implemented by TasGPI for shared memory machines. Since the focus of this thesis lies on distributed memory, this section only presents the general approach of the `StealStrategy` implemented for shared memory.

In contrast to the distributed steal behaviour described in Section 4.4, workers in shared memory environments operate without communication via an interconnect. Instead, thieves query their victims from the runtime and directly call appropriate methods on the victim or its strategies, as exemplified by Listing 4.4.

```
1 auto &thief = c_runtime.get_worker(request.thief);
2 thief.get_steal_strategy().satisfy_lifeline(split_work);
```

Listing 4.4: Direct communication between workers in shared memory environments. (Modified excerpt of `fifo_steal_strategy.hpp`)

Here, a victim of a random steal directly queries the thief's worker object from the runtime and calls the handler method `satisfy_steal` on it. This approach also eliminates the need for explicit communication buffers as described in Subsection 4.4.1. Instead, any required data is explicitly passed as arguments to the appropriate handler methods (compare Listing 4.4, the tasks are directly passed to the handler as split_work). Additionally, weak synchronization (compare Subsection 2.4.3) is not

needed in shared memory systems and replaced in favour of well-known mechanisms like atomics, mutual exclusion, and condition variables.
Beyond that, shared memory stealing works mostly analogous to distributed memory stealing described in Section 4.4. However, in contrast to the behaviour described in Subsection 4.4.3, a worker receiving work via a stored lifeline actively closes its lifelines by removing itself from the stored lifelines of its lifeline buddies.

4.6 Termination detection

As described in Chapter 3, TasGPI does not have access to the `finish` construct which is usually used to detect distributed termination in GLB. Instead, TasGPI requires an explicit implementation of a termination detection scheme in the form of a `TerminationStrategy`. As mentioned in Section 4.2, TasGPI provides example implementations for distributed and shared memory. Both variants utilize a counter-based termination scheme, similar to X10's implementation of the `finish` construct [48], to detect termination as described below.
The shared memory variant simply counts idle workers. Since all workers are local, no network communication is needed and synchronization of the shared counter is guaranteed via the usage of `std::atomic<std::size_t>`. Once all workers are in the idle state, no work remains in the system and the strategy signals termination.
In the case of distributed memory, TasGPI distinguishes between local (i.e. per place) termination and global termination in order to minimize network communication. To that end, TasGPI maintains a shared counter in a PGAS partition and a place local counter. The local counter tracks the number of idle local workers via an `std::atomic`, analogous to the shared memory variant described above.
The global counter, on the other hand, tracks the number of idle places. Accordingly, the last worker on an active place entering the `IDLE` state, increments the shared counter by one. Respectively, the first worker on an idle place receiving new work via a shared lifeline decrements the counter.

To minimize congestion on the shared counter, the shared counter is only read when it is actually updated. Accordingly, only the last place incrementing the counter can actually detect termination. Since the implementation utilizes atomic `fetch_add` operations to increment the counter, the worker automatically receives the current count. Given, the post-increment counter equals the number of places in the system, termination is detected. The counter consists of a single `gaspi_atomic_value_t` on a shared memory partition of place 0. Corruption of the counter is prevented by allowing only atomic operations on the shared memory.

Since only the incrementing worker detects termination, the result is then disseminated among the other places via `gaspi_notify`. This way, the `has_terminated` method of the `TerminationStrategy` has no need to poll the shared counter. Instead, a single check against the termination-notification is made, requiring only local memory access. This way, network communication and congestion on the shared counter can be minimized.

However, for distributed systems, the counter alone is not enough to detect global termination, due to the one-sided nature of communication and the additional latency introduced by the interconnect. Accordingly, TasGPI has to also ensure distributed *quiescence*, i.e. the absence of ongoing network communication, in addition to termination of all workers. Multiple options exist to that effect, and the topic is subject to ongoing research. An in-depth discussion of the available algorithms would unfortunately go beyond the scope of this thesis. However, a good overview may be found at [31, 32].

Due to time constraints, TasGPI implements a rather simple solution for the problem. By preventing workers with outstanding communications from going idle, the problem can be circumvented in its entirety. Since only active messages, i.e. messages containing actual work, can reactivate a worker, other messages (e.g. rejections of steal requests) need not be considered in this context.

Accordingly, the current implementation of the distributed `TerminationStrategy` requires workers to remain in the `PROCESSING` state until all active messages have

been acknowledged. An acknowledgement in this context, is a single notification signaling receipt of the tasks. To prevent racing conditions, the acknowledgment must be posted only after the recipient updated its state with the `TerminationStrategy`. This way, termination can only be detected if no active messages are in flight. Combined with the global counter, this approach can be used to correctly detect distributed termination.

It is important to note that this approach introduces additional latency due to the additional acknowledgement messages. As described in Section 2.3, each additional message takes roughly 1 µs. In addition, the current approach requires the `StealStrategy` to implement the acknowledgements, resulting in undesirable coupling between the strategies. While these tradeoffs are acceptable for this prototype, future iterations of the `TerminationStrategy` should implement a more sophisticated and scalable termination scheme.

4.7 Steal queue using RDMA

As described in Chapter 3, TasGPI uses a queue exclusively utilizing RDMA and one-sided communication primitives to facilitate the exchange of steal requests between workers. The inquiry queue (see Figure 4.3) proposed by TasGPI allows the retrieval of elements from the queue using only cheap local memory operations. Simultaneously, the remote addition of new elements requires only two network operations for most cases. The following describes the design and mechanisms utilized by the proposed datastructure.

Since multiple workers may enqueue requests, but only one worker processes them, the queue is a multi-producer single-consumer queue. In general, two approaches exist to implement such concurrent queues and can be adapted to the PGAS model: Linked lists and circular buffers. Since the number of possible concurrent steal requests to a single worker is bounded by the number of workers and the features of a linked list (e.g. dynamic size and fast insertion at random indices) are not needed, TasGPI uses circular buffers. In contrast to linked lists, this approach does not

require pointers or dynamic memory operations, making circular buffers a good fit for the PGAS model.

In their simplest form, a circular buffer consists of a contiguous segment of memory and metadata in the form of two indices, as depicted by Figure 4.11. The indices, often called head- and tail-pointer, denote the current start and end indices of the queue. New elements are inserted at the head index, while elements are consumed from the tail index. Once such an index exceeds the bounds of the underlying contiguous memory, it instead wraps around using modulo arithmetic.

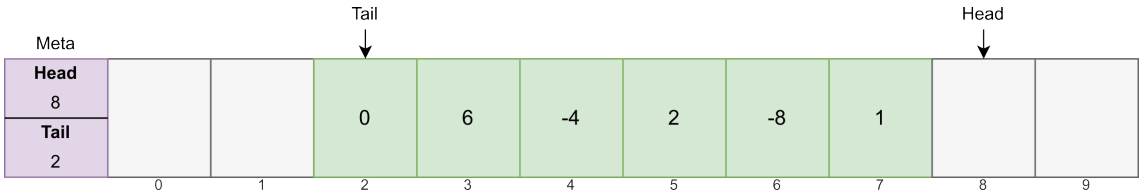


Figure 4.11: Circular buffer with capacity of 10 and current size of 6. Meta data is marked in purple, occupied slots in green.

Since a steal request only requires the ID of the worker and request type, i.e. if it is a random steal or a lifeline steal, TasGPI is able to store each request in a single 64 Bit integer. Here, the most significant bit denotes the type, resulting in a lifeline steal if it is set or a random steal otherwise[2]. This approach leaves the remaining 63 Bit for the worker ID, allowing for more than nine quintillion possible worker IDs. The metadata is also stored in a single 64 Bit integer, using 32 Bit for the head and tail indices respectively. Hence, TasGPI only requires the queues' capacity plus one 64 Bit integers per queue. The capacity of the queue can be configured in pgas_fifo_steal_strategy.hpp and is currently set to 256.

Using this approach, the state of the queue, i.e. the metadata describing it, can be queried and manipulated in a single atomic operation (compare Subsection 2.4.5). This allows for efficient concurrent insertion of new elements into the queue, as described by Listing 4.5.

First, a worker performs an atomic fetch-and-add operation on the target queues metadata (line 3), incrementing the head index by one and receiving the previous

[2]While `gaspi_atomic_value_t` are unsigned, the most significant Bit can be interpreted as the signed bit as in Figure 4.11

```
1  push(victim_id, is_lifeline) {
2      // Increase head by one and retrieve previous value
3      meta = atomic_fetch_add_meta(victim_id)
4      write_slot = read_write_slot(meta)
5
6      if (queue_full(meta)) {
7          poll_until_free(victim_id, write_slot)
8      }
9
10     // If lifeline, set most significant bit
11     if (is_lifeline) {
12         to_enqueue = set_msb(my_worker_id)
13     } else {
14         to_enqueue = my_worker_id;
15     }
16
17     write_notify(victim_id, write_slot, to_enqueue)
18 }
```

Listing 4.5: Insertion of an element into the circular buffer in pseudocode.

metadata. By doing so, the worker reserves the pre-increment head index for exclusive access. Here, the head index may overtake the tail index, indicating that the queue is currently full. Should this be the case, the worker has to poll the metadata of the target queue until the reserved slot is free and the data can be written (lines 6 to 8). A cancellation of the reservation is not possible, since it would require expensive synchronization due to the concurrent manipulation of the metadata. In practice, even with small capacities, this scenario did not occur in the performed experiments. Once the targeted index is free, the worker writes its ID (including the most significant bit as appropriate, lines 10 to 15) directly into the target memory (line 17). Since this communication is completely one-sided, the consumer of the steal requests can not be sure if a reserved slot has already been written to, or if the data is still in transit. Therefore, TasGPI utilizes `gaspi_write_notify` to instantly notify the queue once the transfer finished.

To avoid further synchronization requirements, TasGPI utilizies continuously increasing head and tail indices. This way, collisions of index reservations due to modular arithmetic can be prevented. On the other hand, this approach limits the number of

possible steal requests due to the risk of overflows in the metadata. This problem is solved by performing an atomic compare-and-swap operation to reset the head and tail indices to zero, whenever a worker rejects remaining steal requests. This way, TasGPI is able to efficiently circumvent the arising issue by exploiting the nature of the GLB technique.

Complementary, the retrieval of stored requests from the queue must also be as efficient as possible. The nature of the queue means that the tail index is exclusively manipulated by the owning worker. Exploiting that, the owning worker may pop an element from the queue as described in Listing 4.6.

```
1  pop() {
2      meta = volatile_read_meta()
3      if (queue_is_empty()) {
4          return null
5      }
6
7      // Wait for a notification on the current tail
8      wait_for_notification(meta)
9
10     // Read the current tail value
11     retrieved = read_from_tail(meta)
12
13     volatile_increment_tail()
14
15     return retrieved
16 }
```

Listing 4.6: Retrieval of a value from the circular buffer in pseudocode.

First, the worker performs a volatile read of its own queues metadata (line 2). This ensures that other workers read no corrupt data due to concurrent push operations. Next, the worker compares the head and tail indices and immediately returns if the queue is empty (lines 3-5). Otherwise, it waits (if necessary) until the slot currently pointed to by the tail pointer is ready using `gaspi_notification_wait_some` (and subsequent `gaspi_notify_reset`) as described in Subection 2.4.3 (line 8). Once ready, the steal request is read from the buffer (line 11) and returned after incrementing the tail index by one (lines 13 to 15). Since the metadata may be concurrently

accessed by another worker, this write again is performed in a volatile manner. Notably, this process relies solely on local data, allowing for fast retrieval of requests in all cases.

5 Experiments

This chapter describes the experimental evaluation of TasGPI. It commences with descriptions of the Unbalanced Tree Search (UTS) benchmark in Section 5.1, followed by the Monte-Carlo benchmark in Section 5.2. Then, the used hardware environment is described in Section 5.3. The acquired measurements are subsequently presented in Section 5.4. The chapter concludes with a discussion of the obtained results in Section 5.5.

5.1 Unbalanced Tree Search

The UTS benchmark deterministically generates a highly unbalanced tree using the SHA1 hashing algorithm [34]. Due to the unpredictable and highly irregular nature of the generated tree, UTS is well suited to evaluate the performance of dynamic load balancing schemes. The tree starts with a single node, and each node in the tree is represented by a single hash. The initial hash value is generated from a user provided seed.

From the root node, new nodes are recursively generated on the fly, using the parents hash value as a basis. Due to the unpredictable nature of the hashing algorithm, the number of generated nodes is initially unknown and can not be calculated. Instead, the generated tree structure has to be searched.

In addition to the seed of the tree, the user can configure a cut-off depth, i.e. the maximum depth to be searched, a branching factor, i.e. the upmost number of children per node, and the shape of the generated tree, i.e. binomial or geometric. The benchmark returns a single `long` containing the size of the generated tree for the given configuration.

This benchmark was ported from [37]. Their approach stores unvisited nodes in three arrays, rather than self-contained tasks. The first array *hash*, stores the parents hash value, while the *upper* and *lower* arrays store the child indices to be generated. Thus, this triple represents exactly $upper - lower$ tasks for each index of the arrays. UTS is provided by the reference variant and was ported to TasGPI.

5.2 Monte-Carlo Simulation

This benchmark approximates the value of PI using the well-known Monte-Carlo technique. To that end, the benchmark creates n random coordinates in the unit square. By counting the number of such points falling inside the unit circle, according to $x^2 + y^2 \leq 1$, PI can then be approximated by the following formula:

$$\pi = 4 * \frac{points\_inside}{points\_simulated}$$

The number of simulated points are configured by the user at runtime, by specifying the number of points to be simulated per worker. Each task simulates a single point, allowing the benchmark to be run with varying granularity by configuring GLB's N parameter. Consequently, no new tasks are generated at runtime and the workload is evenly distributed between all workers at the start of the computation. The benchmark results in a `double`, containing the approximated value of PI.
The benchmark has been implemented from scratch for both TasGPI and the reference variant.

5.3 Benchmark environments

The benchmarks presented in Section 5.4 were executed on the Linux Cluster of the University of Kassel [22]. The used partition public2023 provides up to 36 nodes, each equipped with 256 GB of main memory and two AMD EPYC 7443 24-Core processors. Each node is connected via a Mellanox InfiniBand EDR interconnect.

Initial tests were run on the public-1 partition of the Goethe-HLR [52]. Unfortunately, at the time of writing, the cluster was not available for further benchmarks.

5.4 Results

The experiments were run on up to 32 nodes, where each node spawned one worker per physical core for a maximum of 1536 workers. As noted in Chapter 1, the experiments were run without true RDMA due to technical difficulties. Instead, an ethernet-based wrapper (provided by GPI) simulating the desired behaviour over TCP was used.

The UTS benchmark adapts the configuration of the reference variant with an initial seed of 19, branching factor of 4, and geometric tree shape [42]. Likewise, GLB was configured to process chunks of 511 tasks with three steals per round (N=511, W=3). Parameter Z was calculated at runtime to fit the number of workers (compare Section 4.2). Data was collected for TasGPI and the reference variant. Since Reitz et al. support different levels of optimization (compare Section 2.5.4), the following distinction is made:

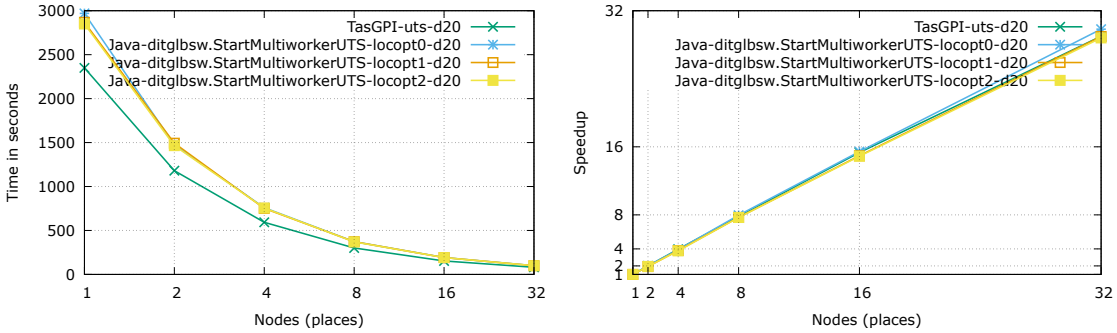
locopt0 The reference variant, without local optimization.

locopt1 The reference variant, only attempt local steals first.

locopt2 The reference variant, all local optimizations enabled.

Each experiment was run ten times with tree depths of 17 through 20. The collected data can be found in Appendix A.2.

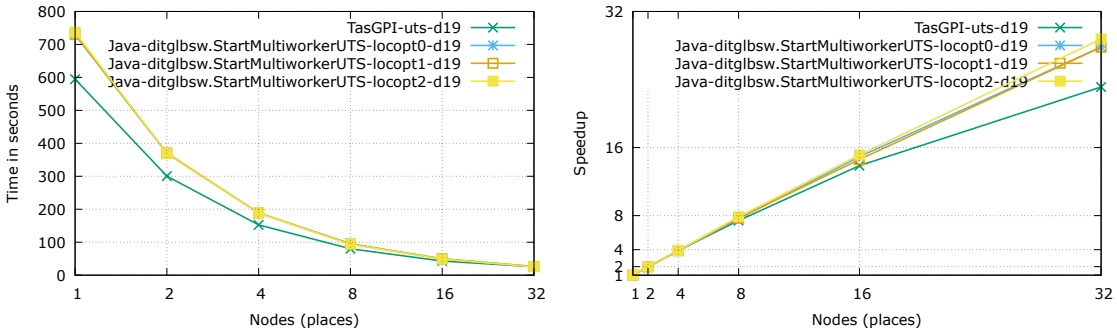
As seen in Figure 5.1(b), TasGPI achieves very similar speedups to the reference variant. On 32 places *locopt0* has a slight advantage (of roughly 2.8 %) over the other variants. In terms of runtime, TasGPI outperforms all variants of the reference implementation as seen in Figure 5.1(a). On average, TasGPI is 19.15 % faster than the best run of the reference versions.



(a) Runtime (b) Speedup

Figure 5.1: UTS Experiment with tree-depth of 20

A similar result is achieved for a tree depth of 19. As seen in Figure 5.2(a), TasGPI performs the calculation significantly faster up to 16 places.



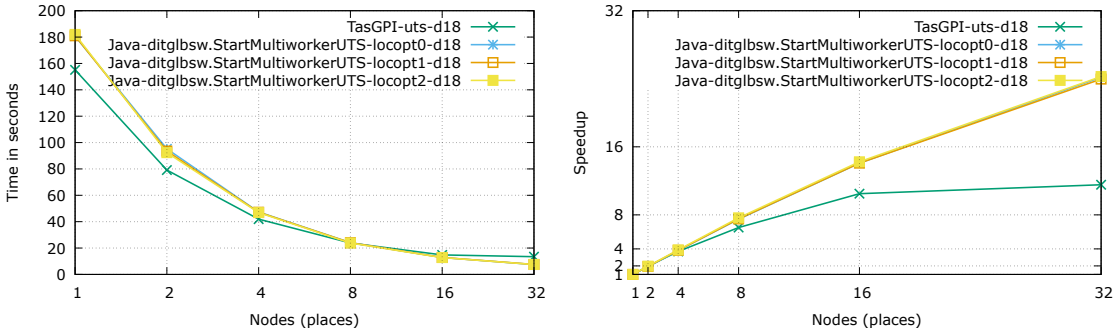
(a) Runtime (b) Speedup

Figure 5.2: UTS Experiment with tree-depth of 19

However, the total runtimes are nearly identical between all versions when run on 32 places. Consequently, TasGPI's speedup scales similar to the reference variant up to 16 places, degrading noticeable for 32 places.

The trend continues for a tree depth of 18, as seen in Figure 5.3. Here, TasGPI scales well up to eight places, but stagnates on 16 or more places.

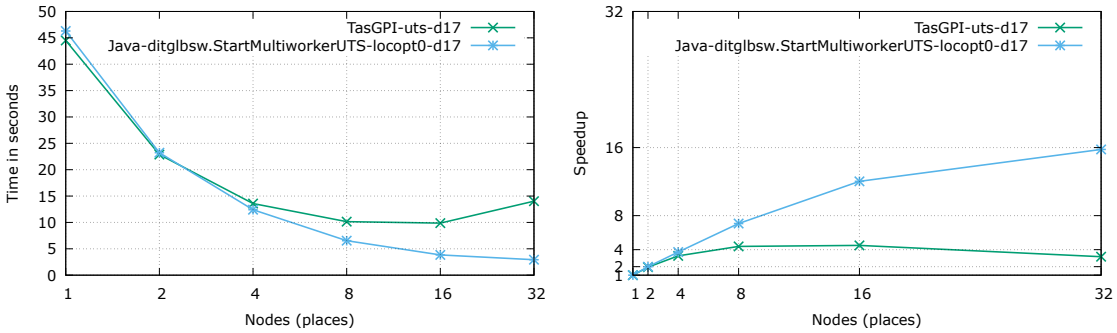
Consequently, TasGPI outperforms the reference variants only up to four places. At eight places, runtimes are practically identical, while on 16 or more places the reference variants slightly outperform TasGPI. For a tree depth of 17, both variants exhibit degrading speedups, as seen in Figure 5.4(b). It is important to note that even on one place TasGPI performs the calculations only 4 % faster than the



(a) Runtime (b) Speedup

Figure 5.3: UTS Experiment with tree-depth of 18

reference version, which is a sharp decline from the roughly 20 % achieved in the other experiments.



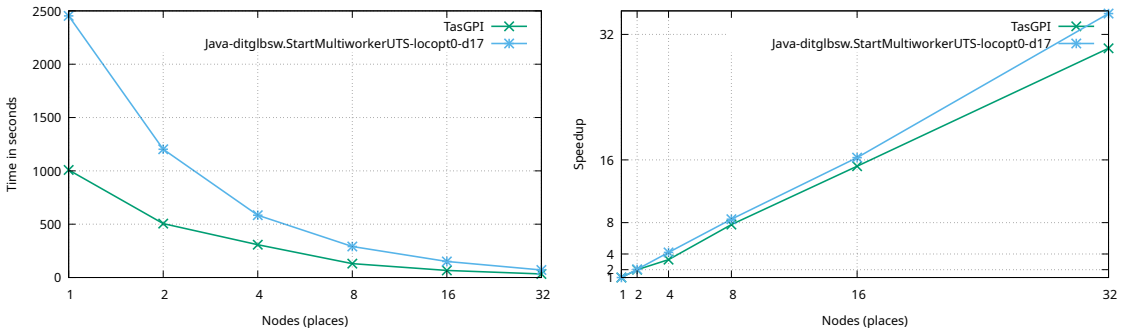
(a) Runtime (b) Speedup

Figure 5.4: UTS Experiment with tree-depth of 17

From there, TasGPI only scales well up to four places, exhibiting stagnating speedups up to 16 places. When run on 32 places, TasGPI's speedups actually regress, resulting in a higher runtime on 32 places than on 16 places. Consequently, the reference variant significantly outperforms TasGPI for eight places and more.

In addition to the UTS benchmark, TasGPI was evaluated using a Monte-Carlo simulation, as described in Section 5.2. Similar to the UTS benchmark, the experiments were run on up to 32 nodes with one worker per physical core. The GLB configuration was adapted from the UTS benchmark. Steal optimizations of the reference variant were disabled, as a lower number of steals are expected due to the static scheduling of tasks. Measurements were taken for simulations with $1.649\,267\,4 \times 10^{12}$ (large) and 25769803776 (small) points, distributed evenly between all workers.

The pattern observed for the UTS benchmark continues with this static workload. As seen in Figure 5.5, TasGPI exhibits near linear speedups for the large configuration. The reference variant, on the other hand, actually exhibits super-linear speedups.

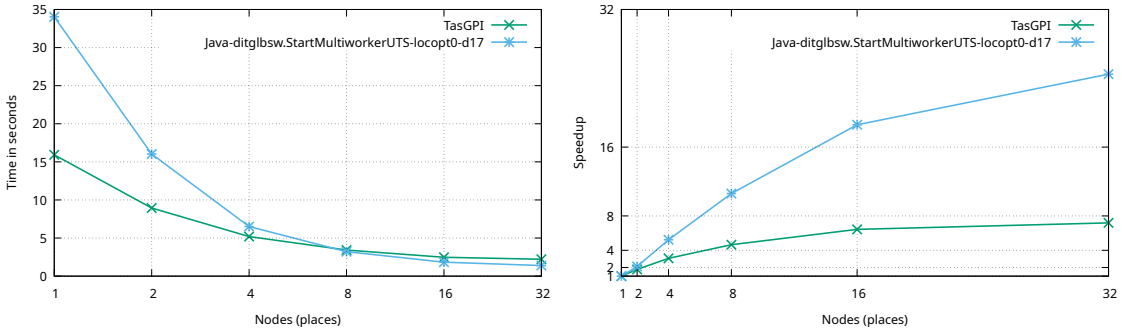


(a) Runtime (b) Speedup

Figure 5.5: Strong scaling (large): Monte-Carlo simulation with 1.6492674e+12 simulated points.

However, TasGPI still has the edge when it comes to raw runtime, significantly outperforming the reference variant for all tested place counts.

For the small configuration seen in Figure 5.6, speedups degrade for both variants, with the reference variant performing significantly better than TasGPI. In terms



(a) Runtime (b) Speedup

Figure 5.6: Strong scaling (small): Monte-Carlo simulation with 25769803776 simulated points.

of raw runtime, TasGPI outperforms the reference variant up to four places. After that, runtimes are nearly identical with a slight edge for the reference variant.

5.5 Discussion

In the scope of this thesis, a new variant of the GLB technique was proposed and implemented in the form of the prototypical GLB framework TasGPI. The extraction of key behaviours (termination detection, steal behaviour and creation of the lifeline graph) works well and without overhead due to static polymorphism, as discussed in Chapter 3.

TasGPI was benchmarked against the reference variant by Reitz et al. with promising results, as described in Section 5.4. Both variants handle both static and dynamic workloads well, as can be seen in Figure 5.5 and Figure 5.1 respectively. In terms of raw runtime, TasGPI significantly outperforms the reference variant given sufficiently large workloads. However, the reference variant scales much more consistently with smaller workloads. TasGPI, on the other hand, exhibits stagnating speedups for decreasing work loads.

The effect is probably caused by a combination of factors. With less work in the system, workers are forced to steal more frequently since the local tasks themselves do not generate a self-sustaining amount of new work. Similarly, an overcommitment of workers, i.e. too many workers for a given problem size, can result in stagnating speedups on larger processor counts. Here, the causes are increasing communication and coordination costs, while the parallelizable portion of the application stays constant.

Considering TasGPI's performance in the UTS benchmark, overcommitment may be a factor. Here, TasGPI scales well on all node-counts for a tree depth of 20 (compare Figure 5.1(b)). From there, the number of nodes for which TasGPI exhibits near-linear scaling roughly halves with each decrease in tree depth (compare Figures 5.2(b), 5.3(b), and 5.4(b)). For a tree depth of 17 in particular, TasGPI loses its runtime advantage (compare Section 5.4) over the reference variant even if executed on one place, further suggesting that the problem size may be the culprit. However, further investigation is required to precisely identify the cause for the observed stagnation in speedups for smaller workloads. Two potential approaches

include examining the actual count of steal requests made by both variants, and incorporating a synthetic benchmark (e.g. smooth weak scaling [36, p.45]) in the evaluation.

Furthermore, the current termination detection scheme employed by TasGPI should be replaced by a more sophisticated algorithm. As discussed in Section 4.6, the current implementation is rather simple and requires additional acknowledgements for each transfer of tasks. In addition, the shared counter requires synchronized access to the underlying memory, which may cause congestion, particularly in scenarios with many places and little work. Both factors are likely to cause declining performance, especially for irregular workloads on high place counts.

In terms of raw computation time, TasGPI outperforms the reference variant for sufficiently large problems. The primary reason for this disparity likely stems from the choice of programming languages, with TasGPI being implemented in C++ and the reference variant being developed in Java. While C++ is considered to be faster than Java in most scenarios, it is difficult to quantify the effect of the language. Accordingly, no definite assertion can be made about the quality of TasGPI's implementation.

Regardless, C++ may be the superior choice for particularly memory intensive or long-running computations, given its manual memory management. For such workloads, C++ is expected to perform at least as well as Java (given adequate programming of the `TaskQueue`), while avoiding the need for garbage collection, which in Java's case may cause unpredictable delays. However, this problem can be mitigated by utilizing off-heap memory (e.g. direct memory buffers) and choosing an appropriate garbage collector [4, 33].

Furthermore, the choice of libraries and benchmark specific implementations can influence the runtime and scaling of the experiments. A good example is the UTS benchmark implemented by TasGPI. The initial implementation relied on the recommended OpenSSL implementation of message digests [39]. Even with one context per worker (eliminating state-sharing between workers) and following the documented recommendations, the original implementation of UTS performance was

rapidly deteriorating with increasing worker counts. Via extensive profiling, the issue could be traced back to lock contention in the EVP interface. By switching to the deprecated SHA1 [40] implementation of OpenSSL, performance was significantly increased. For instance, 16 local workers on a 16 core machine were able to perform the UTS benchmark for a tree depth of 15 in roughly half the time the original implementation required. As stated above, the relative performance gain increases with growing worker counts. The old version can be still enabled for comparisons by setting SHA1_PROVIDER to OSSL via CMake.

Furthermore, TasGPI in its current iteration does not pin workers to specific cores of the CPU. Accordingly, TasGPI's results may suffer from context switches and cache effects. In contrast, the super-linear speedup of the reference variant shown in Figure 5.5(b) suggests that the reference variant is able to exploit cache effects.

Last but not least, the experiments could not be run using RDMA, instead relying on a TCP wrapper provided by GPI, as noted in Section 5.4. While extensive debugging and research was undertaken, TasGPI in its current iteration still fails in unpredictable manners when executed with RDMA on high worker counts. At the time of writing, no definite conclusion can be made about the root cause of the issues. However, many causes could be ruled out:

First, the implementation was checked against the GASPI documentation [12] and guidelines to ensure GPI is used in the correct manner, with no result. Subsequently, the algorithms and offset calculations were checked for correctness, both manually and via GPI's debug library. So far, no issue with TasGPI's implementation could be found, and successful runs using the TCP wrapper suggest that neither the used algorithms nor offset calculations are at fault.

In addition, TasGPI was tested on different RDMA enabled clusters. Experiments run on the Goethe cluster (compare Section 5.3) using InfiniBand, occasionally resulted in an error message (See A.2) which provided additional information. Since some of the information contained in the error message is not documented publicly, the NVIDIA support was contacted [47]. However, the suggested causes were already

researched (i.e. buffer sizes and offsets) or out of control of the developer and handled by GPI internals (i.e. buffer access rights and access keys).

Furthermore, a similar issue is documented by another user of the GPI library [27]. While the issue contains a proposed solution, it could not be tested yet, since the proposal requires the change of system settings and no root access to an RDMA enabled cluster is available. Unfortunately, GPI's developers have not reacted to the issue at the time of writing. Accordingly, the issue likely lies with the RDMA hardware or the GPI library, not TasGPI's implementation, but further investigation is needed.

Given future iterations of TasGPI are able to utilize RDMA, improved results can be expected. Initial runs of the experiments on low place counts but with RDMA were promising, but the issues described above prevented the collection of sufficient data for further analysis.

Considering TasGPI's focus on RDMA, another important aspect of the technology that is not fully utilized in the current iteration is the concept of zero-copy transfers (compare Section 2.3). Since the current implementation allows users to store arbitrary task representations (see Chapter 3), tasks must be explicitly serialized before transferring them over the network. By enforcing a trivially copyable [8] task representation, tasks can be stored and transferred directly between segments. This way, both the serialization and the additional copying of tasks to a transfer buffer can be prevented.

Considering the relatively high cost of serialization and copying [15], the potential for performance and scalability gains could outweigh the additional restrictions imposed on developers. At the same time, this would enable the implementation of coordinated work stealing, i.e. thieves could steal tasks directly from their victim without the need for steal requests. Given coordinated work stealing eliminates the need for active participation of the victim, further research in this direction is warranted.

6 Related work

Dynamic load balancing can be realized in the form of work stealing or work sharing [6]. Work stealing, popularized by the Cilk parallel programming system [16], has idle workers actively steal tasks from busy workers. In contrast, work sharing achieves load balancing by disseminating the work of busy workers to idle ones. While both approaches have been shown to perform roughly equivalent [1], work stealing has fewer task exchanges between workers, since exchanges only occur if workers are idle [6].

Both approaches can be implemented in a coordinated fashion, i.e. thieves may directly access the task pools of their victims, or in a cooperative fashion, i.e. thieves notify their victims that work is needed, and the victim sends a portion of their tasks from a private task pool. Similar to work stealing and work sharing, both variants have been shown to exhibit comparable performance [38, 1].

Many solutions for scheduling tasks via work stealing exist [e.g. 35, 11, 29]. However, early iterations of dynamic load balancers were unable to perform well on distributed multi-core clusters that are prevalent today [41]. A variant of work stealing with good performance on such supercomputers is the lifeline scheme of the GLB technique [44]. While initially restricted to one worker per place, the technique has since been improved upon. One of the first variants to allow for multiple workers per place combines intra-place work sharing with the lifeline-based work stealing approach of GLB [53]. However, another variant of GLB by Reitz et al. [42] has since shown that the GLB technique can be extended to multi-worker GLB with better performance and without the additional complexity introduced by the hybrid variant. Consequently, the newer variant served as the reference variant for this thesis.

Other research on work stealing concentrates on improving the performance of the underlying task pools [26, 24, 41, 14]. One popular approach is the utilization of `split queues`, specialized task pools that allow for greater levels of concurrency with less synchronization and shorter critical paths [14, 28, 10]. In this context, special attention is given to new and emerging network technologies that allow for even better performance [10, 28].

By compacting the state of the `split queue` into a compact format that can be manipulated via singular atomic operations, the communication between workers to acquire tasks from their victims can be reduced. By utilizing modern networking technologies like RDMA that allow for one-sided communication, low latency, and atomic operations with kernel-bypass [21], `split queues` can further enhance the performance of work stealing by allowing thieves to assess availability and steal tasks without interrupting the active computations of their victims [10, 28]. Accordingly, this thesis finds itself between the research on lifeline-based work stealing and improved task pools by combining the GLB technique with modern networking technologies.

7 Conclusion

This thesis presents a novel multi-worker variant of the GLB technique, with a primary focus on RDMA and one-sided communication. One of the central goals of this thesis was to investigate if existing GLB variants could be enhanced through the integration of RDMA capabilities. The implementation of this approach in the form of the prototypical framework TasGPI was realized in C++ using the PGAS library GPI. Given previous implementations usually rely on the APGAS model, the transition to the PGAS model necessitated significant changes, including the development of an explicit termination detection scheme, and a shift towards data-driven communication in favour of active messages.

An empirical evaluation of TasGPI through the UTS and Monte-Carlo benchmarks yielded promising results. Although technical difficulties prevented the utilization of RDMA in the experiments, TasGPI displayed near-linear scaling and superior runtimes when compared to the reference variant, particularly for large workloads. However, it is worth noting that TasGPI scaled less stable for small workloads, resulting in worse performance for very small problem sizes, highlighting the need for further research and optimizations. It can be anticipated that TasGPI's performance can be further improved by fixing the issues regarding RDMA and replacing the current termination detection mechanism with a more sophisticated scheme, both in runtime and scalability.

In summary, TasGPI is a promising prototype of the new approach to GLB, demonstrating good results with considerable potential for future improvements. In light of the encouraging outcomes, it becomes evident that further exploration into GLB leveraging RDMA communication is both warranted and promising.

Future iterations of TasGPI should first focus on resolving the issues relating to RDMA, possibly moving away from the GPI library in favour of more stable RDMA libraries. In addition, the current termination detection scheme should be replaced by a more sophisticated scheme to allow for good scaling with high place counts.

Given TasGPI aims to eliminate the interruption of the victim's computation in steal processes, a transition towards coordinated workstealing should be considered. In this context, current research suggests the application of a SplitQueue data structure [10]. Combined with RDMA the approach allows for truly asynchronous steal operations without any active involvement of the victim.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Umut A Acar, Arthur Charguéraud, and Mike Rainey. "Scheduling parallel programs by work stealing with private deques". In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 2013, pp. 219–228.

[2] *APGAS Git repository*. URL: `https://github.com/x10-lang/x10/tree/master/apgas` (visited on 06/26/2023).

[3] InfiniBand Trade Association. *InfiniBand Architecture Specification*. Tech. rep. Version 1.2.1. InfiniBand Trade Association, 2007.

[4] Kinan Al-Attar et al. "Towards Java-based HPC using the MVAPICH2 Library: Early Experiences". In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2022, pp. 510–519.

[5] Dotan Barak. *Tips and tricks to optimize your RDMA code*. June 8, 2013. URL: `https://www.rdmamojo.com/2013/06/08/tips-and-tricks-to-optimize-your-rdma-code/` (visited on 08/09/2023).

[6] Robert D Blumofe and Charles E Leiserson. "Scheduling multithreaded computations by work stealing". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[7] Benjamin A Brock et al. "RDMA vs. RPC for implementing distributed data structures". In: *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE. 2019, pp. 17–22.

[8] *C++ named requirements: TriviallyCopyable*. URL: `https://en.cppreference.com/w/cpp/named_req/TriviallyCopyable` (visited on 09/23/2023). Last modified 08/11/2022.

[9] Georgel Calin et al. “A theory of partitioned global address spaces”. In: *arXiv preprint arXiv:1307.6590* (2013).

[10] Hannah Cartier, James Dinan, and D Brian Larkins. “Optimizing work stealing communication with structured atomic operations”. In: *Proceedings of the 50th International Conference on Parallel Processing*. 2021, pp. 1–10.

[11] Guojing Cong et al. “Solving large, irregular graph problems using adaptive work-stealing”. In: *2008 37th International Conference on Parallel Processing*. IEEE. 2008, pp. 536–545.

[12] GASPI Consortium. *GASPI: Global Address Space Programming Interface. Specification of a PGAS API for communication*. Standard. Version 17.1. Fraunhofer ITWM, 2017.

[13] Mattias De Wael et al. “Partitioned global address space languages”. In: *ACM Computing Surveys (CSUR)* 47.4 (2015), pp. 1–27.

[14] James Dinan et al. “Scalable Work Stealing”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC ’09. New York, NY, USA: Association for Computing Machinery, 2009.

[15] Philip Werner Frey and Gustavo Alonso. “Minimizing the hidden cost of RDMA”. In: *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE. 2009, pp. 553–560.

[16] Matteo Frigo, Charles E Leiserson, and Keith H Randall. “The implementation of the Cilk-5 multithreaded language”. In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, pp. 212–223.

[17] *GPI-2 Repository*. URL: `https://github.com/cc-hpc-itwm/GPI-2` (visited on 06/27/2023).

[18] *GPI2 Homepage*. URL: `http://www.gpi-site.com/docs/` (visited on 06/27/2023).

[19] Rainer Grimm. *The Strategy Pattern*. Dec. 11, 2022. URL: https://www.modernescpp.com/index.php/the-strategy-pattern (visited on 07/15/2023).

[20] Daniel Grünewald and Christian Simmendinger. "The GASPI API specification and its implementation GPI 2.0". In: *7th International Conference on PGAS Programming Models*. Vol. 243. 2013, p. 52.

[21] Chuanxiong Guo et al. "RDMA over commodity ethernet at scale". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, pp. 202–215.

[22] Competence Center for High Performance Computing in Hessen. *Linux Cluster Kassel*. URL: https://www.hkhlr.de/en/clusters/linux-cluster-kassel (visited on 08/21/2023).

[23] Jeff Hilland. "RDMA protocol verbs specification". Version 1.0. In: (2003). URL: http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf.

[24] 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 (2011), pp. 553–581.

[25] *HP and Mellanox Benchmarking Report for Ultra Low Latency 10 and 40Gb/s Ethernet Interconnect. Benchmark Report*. Tech. rep. Mellanox Technologies, July 2012. URL: https://network.nvidia.com/related-docs/whitepapers/HP_Mellanox_FSI%20Benchmarking%20Report%20for%2010%20%26%2040GbE.pdf (visited on 07/25/2023).

[26] 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 (2004), pp. 1–47.

[27] krzikalla. *Runtime failures on larger process counts*. July 29, 2021. URL: https://github.com/cc-hpc-itwm/GPI-2/issues/66 (visited on 09/23/2023).

[28] D Brian Larkins, John Snyder, and James Dinan. “Accelerated work stealing”. In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.

[29] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. “The design of a task parallel library”. In: *Acm Sigplan Notices* 44.10 (2009), pp. 227–242.

[30] Patrick MacArthur and Robert D Russell. “A performance study to guide RDMA programming decisions”. In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE. 2012, pp. 778–785.

[31] Jeff Matocha and Tracy Camp. “A taxonomy of distributed termination detection algorithms”. In: *Journal of Systems and Software* 43.3 (1998), pp. 207–221.

[32] Friedemann Mattern. “Algorithms for distributed termination detection”. In: *Distributed computing* 2 (1987), pp. 161–175.

[33] Reintech Media. *Java high-performance computing: Optimizing Java code for supercomputers*. Apr. 18, 2023. URL: `https://reintech.io/blog/java-high-performance-computing-optimizing-java-code-for-supercomputers` (visited on 09/23/2023).

[34] Stephen Olivier et al. “UTS: An unbalanced tree search benchmark”. In: *Languages and Compilers for Parallel Computing: 19th International Workshop, LCPC 2006, New Orleans, LA, USA, November 2-4, 2006. Revised Papers 19*. Springer. 2007, pp. 235–250.

[35] Chuck Pheatt. “Intel® threading building blocks”. In: *Journal of Computing Sciences in Colleges* 23.4 (2008), pp. 298–298.

[36] Jonas Posner. “Load balancing, fault tolerance, and resource elasticity for Asynchronous Many-Task systems”. PhD thesis. University of Kassel, Germany, 2022. URL: `https://kobra.uni-kassel.de/handle/123456789/14032`.

[37] Jonas Posner. *PLM-APGAS-Examples*. URL: https://github.com/posnerj/PLM-APGAS-Applications/tree/master (visited on 08/25/2023).

[38] Jonas Posner and Claudia Fohry. "Cooperation vs. coordination for lifeline-based global load balancing in APGAS". In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10*. 2016, pp. 13–17.

[39] OpenSSL Project. *evp*. Tech. rep. OpenSSL Project. URL: https://www.openssl.org/docs/man1.1.1/man7/evp.html (visited on 09/23/2023).

[40] OpenSSL Project. *SHA1*. Tech. rep. OpenSSL Project. URL: https://www.openssl.org/docs/man3.0/man3/SHA1.html (visited on 09/23/2023).

[41] Kaushik Ravichandran, Sangho Lee, and Santosh Pande. "Work Stealing for Multi-core HPC Clusters". In: *Euro-Par 2011 Parallel Processing*. Springer Berlin Heidelberg, 2011, pp. 205–217.

[42] Lukas Reitz et al. "Lifeline-based load balancing schemes for Asynchronous Many-Task runtimes in clusters". In: *Parallel Computing* 116 (2023).

[43] rumach. *GPI-2/bin*. URL: https://github.com/cc-hpc-itwm/GPI-2/tree/next/bin (visited on 08/23/2023).

[44] Vijay Saraswat et al. "Lifeline-based Global Load Balancing". In: vol. 46. Sept. 2011, pp. 201–212. DOI: 10.1145/2038037.1941582.

[45] Vijay Saraswat et al. "The asynchronous partitioned global address space model". In: *The First Workshop on Advances in Message Passing*. 2010, pp. 1–8.

[46] Christian Simmendinger, Mirko Rahn, and Daniel Grünewald. *GASPI Tutorial*. URL: http://gpi-site.com/tutorial/ (visited on 06/28/2023).

[47] Adrian Steinitz. *MLX Completion with error*. June 20, 2023. URL: https://forums.developer.nvidia.com/t/mlx-completion-with-error/257118 (visited on 09/23/2023).

[48] *Synchronization. Finish.* URL: http://x10-lang.org/documentation/intro/latest/html/node5.html#SECTION00533010000000000000 (visited on 07/24/2023).

[49] Mellanox Technologies. *Mellanox Adapters Programmer's Reference Manual (PRM). Supporting ConnectX®-4 and ConnectX®-4 Lx.* Tech. rep. Mellanox Technologies, 2016.

[50] *The APGAS model.* URL: https://x10.sourceforge.net/documentation/intro/latest/html/node4.html (visited on 07/24/2023).

[51] *The X10 Parallel Programming Language.* URL: http://x10-lang.org/ (visited on 06/26/2023).

[52] Top500.org. *GOETHE-HLR.* URL: https://www.top500.org/system/179588/ (visited on 08/21/2023).

[53] 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 (Mar. 2016), pp. 416–424. DOI: 10.2197/ipsjjip.24.416.

[54] Wei Zhang et al. "GLB: Lifeline-Based Global Load Balancing Library in X10". In: *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*. PPAA '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 31–40. URL: https://doi.org/10.1145/2567634.2567639.

A Appendix

A.1 TasGPI worker workflow

```
while(true) {
    while(has_work_available()) {
        set_state(PROCESSING)
        process(get_N())
        StealStrategy.process_steal_requests()
        StealStrategy.decline_pending_steal_requests()
    }

    set_state(STEALING)
    if (StealStrategy.attempt_random_steals()) {
        resume_processing()
    }
    if (StealStrategy.activate_lifelines()) {
        resume_processing()
    }

    set_state(IDLE)
    while(true) {
        if (StealStrategy.implements_lifeline_poll()) {
            StealStrategy.poll_lifelines()
        }
        if (has_work_available()) {
            resume_processing()
        }

        StealStrategy.decline_pending_steal_requests()

        if (TerminationStrategy.has_terminated()) {
            return_to_runtime();
        }
        wait_for_work(MAX_POLL_TIME)
    }
}
```

Listing A.1: TasGPI workers work loop in pseudocode.

A.2 UTS benchmark raw data

Places	TasGPI	Ditglbsw-Locopt0	Ditglbsw-Locopt1	Ditglbsw-Locopt2
1	2349.32	2969.52	2868.13	2849.31
2	1180.06	1473.33	1493.04	1464.94
4	592.88	759.31	751.34	755.24
8	300.72	372.76	372.28	370.1
16	153.16	192.27	191.76	191.3
32	80.99	99.51	98.99	98.9

Table A.1: UTS: Raw runtime data for d=20 (in seconds)

Places	TasGPI	Ditglbsw-Locopt0	Ditglbsw-Locopt1	Ditglbsw-Locopt2
1	549.65	729.71	730.67	737.11
2	300.44	368.79	370.60	368.49
4	152.51	189.41	188.38	188.66
8	80.11	95.53	94.58	93.68
16	42.85	48.90	50.05	48.73
32	25.71	26.22	26.26	25.61

Table A.2: UTS: Raw runtime data for d=19 (in seconds)

Places	TasGPI	Ditglbsw-Locopt0	Ditglbsw-Locopt1	Ditglbsw-Locopt2
1	155.13	180.68	180.97	182.03
2	79.14	95.24	93.58	92.34
4	41.92	47.33	47.19	46.88
8	23.81	24.09	24.05	23.77
16	14.78	12.82	12.84	12.77
32	13.44	7.46	7.55	7.5

Table A.3: UTS: Raw runtime data for d=18 (in seconds)

Places	**TasGPI**	**Ditglbsw-Locopt0**
1	44.50	46.3
2	22.87	23.22
4	13.58	12.41
8	10.15	6.54
16	9.87	3.84
32	14.0	32.93

Table A.4: UTS: Raw runtime data for d=17 (in seconds)

A.3 Monte-Carlo benchmark raw data

Places	TasGPI	Ditglb
1	1008.55	2453.0397
2	505.466	1202.66986
4	307.62	584.15674
8	130.136	290.760836
16	66.384	150.592266666667
32	33.364	70.77674

Table A.5: Monte-Carlo: Raw runtume data for large configuration ($1.649\,267\,4\times10^{12}$) in seconds

Places	TasGPI	Ditglb
1	15.922	34.03
2	8.94	16.03
4	5.192	6.51
8	3.422	3.21
16	2.474	1.83
32	2.214	1.39

Table A.6: Monte-Carlo: Raw runtume data for small configuration ($2\,576\,980\,377$) in seconds

A.4 Completion with error (Mellanox)

```
mlx5: node45-001.cm.cluster: got completion with error:
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
0000001e 00000000 00000000 00000000
00000000 00008813 1201e912 011a89d3
```

Listing A.2: A Completion With Error message obtained on the Goethe cluster.

The errors structure is described at [49, p. 117ff]. More information can be found at [3, p. 526ff].