UNI KASSEL
VERSITÄT

Universität Kassel

Master Thesis

# Global Load Balancing and Intra-Node Synchronization with the Java Framework APGAS

**presented to**
**Department for Electric Engineering/Computer Science**
**Research Group Programming Languages/Methodologies**

Jonas Posner
30203660

Kassel, January 22, 2016

Examiners:
Prof. Dr. Claudia Fohry
Prof. Dr. Gerd Stumme

# Contents

# List of Figures

# List of Tables

# List of Listings

# List of Abbreviations

| | |
|---|---|
| **APGAS** | Asynchronous Partitioned Global Address Space. This thesis means the framework for Java. |
| **APGAS_GLB** | Adopted variant of X10_GLB in APGAS. |
| **APGAS_Split_GLB** | Enables intra-place synchronization in APGAS_GLB. |
| **BC** | Betweenness Centrality. A benchmark. |
| **GLB** | Global Load Balancing |
| **UTS** | Unbalanced Tree Search. A benchmark. |
| **X10** | Parallel programming language. |
| **X10_GLB** | Official GLB implementation in X10. |

# Statutory Declaration

I declare on oath that I completed this work on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this, nor a similar work, has been published or presented to an examination committee.

Kassel, January 22, 2016

Jonas Posner

# 1. Introduction

Parallel programming is becoming more and more important in software development. When using multiple processors, an initial problem should be split into several smaller ones, called *tasks*, to accelerate the execution. These are then distributed to multiple inter-connected computing units, which can be cores, processors or cluster nodes. In this way, large problems can be solved cooperatively, which can reduce the execution time significantly compared to a serial solution.

Unfortunately, parallel programming entails along difficulties for the programmer. For example, shared variables have to be synchronized to guarantee data integrity. Furthermore, programming errors can cause deadlocks, so that the program will never terminate. Because of this complexity, development and maintainability of parallel programs is demanding.

There are two fundamentally different classes of parallel architectures: systems with shared memory, such as multi-core systems, and systems with distributed memory, such as clusters. In case of shared memory, every processor has access to the same memory, so that data can be exchanged easily. Open Multi-Processing (OpenMP) is an established parallel programming system whose aim is to realize this approach. OpenMP enables parallelization through compiler directives and some library functions, and is available for C, C++ and Fortran.

In architectures with distributed memory, in contrast, the nodes have to communicate across a network to exchange their data because a node cannot gain access to remote memories. A frequently used programming system in this class is the Message Passing Interface (MPI). MPI is a standardized application programming interface and is provided for C, C++ and Fortran, as well.

The Partitioned Global Address Space (PGAS) model unifies the two approaches and tries to reduce complexity. A *place* in PGAS represents a local computation unit. Every place can access every part of memory, but local accesses are faster than remote one. This way, PGAS hides the complexities of network communication from the programmer. An asynchronous variant of PGAS introduces an *activity* which is comparable with a thread in Java, and can be started at runtime. The relatively young parallel programming language X10 is based on this asynchronous variant of PGAS.

In June 2015, a member of the X10 development team released Version 1.0 of a framework called *APGAS* [27], which is written in Java 8. This framework tries to bring as many parallelization features of X10 to Java as possible. Since Java is an established language, the APGAS framework provides access to X10's parallel features for a much wider audience.

A big challenge for the efficient use of parallel systems is load balancing, i.e. distributing tasks fairly among the available processors. An interesting technique, called *lifeline-based global load balancing*, has quite recently been introduced by Saraswat et al. [21]. The technique has been implemented in X10 in the form of a framework, called *GLB* [30]. The framework is part of the standard library of X10. It has good performance and scalability, but also a significant restriction: only one activity per place can be executed.

This thesis had two goals. First, the GLB framework was ported from X10 to Java, using the APGAS framework. Second, a data structure, called *split queue* [6], was incorporated, which allows a limited form of concurrent access. Thus, multiple actives per place can be executed concurrently.

In experiments, we ran the two APGAS GLB variants at the Lichtenberg high performance computer at TU Darmstadt [24]. We referred to two benchmarks: Unbalanced Tree Search (UTS) and Between Centrality (BC). Both are available in the official X10 project. We used an implementation variant with explicit storage of individual tasks [8], and ported them to APGAS. Moreover, we ran the benchmarks in X10, utilizing the original GLB framework, and compared them to the new APGAS variants with respect to performance and scalability. The results of the experiments show no noticeable performance difference between both APGAS variants. A comparison between the APGAS variants and the X10 variant shows that neither X10 nor APGAS is superior overall.

This thesis starts with the programming background in Chapter 2. First, Chapter 2 describes the asynchronous variant of PGAS. Then the realizations of this programming model in X10 and APGAS are explained and compared. After that, Chapter 3 describes the concept and the existing X10 implementation of lifeline-based global load balancing. In Chapter 4, the new APGAS implementation of GLB is presented, and differences to the GLB implementation in X10 are explained. The split queue concept and some details of its implementation are explained in Chapter 5. After this, Chapter 6 describes the experiments and discusses their results. The conclusion in Chapter 7 summarizes the thesis and gives an outlook to possible future work.

# 2. Programming Background

This chapter provides the programming background that is required for the understanding of the rest of this thesis. It starts with principles of the used parallel programming model in Section 2.1. Next, Sections 2.2 and 2.3 give an overview of the programming language X10, and the APGAS framework for Java, respectively. Afterwards, their common constructs are explained in Section 2.4. Finally, Section 2.5 outlines several runtime options.

## 2.1. X10 Programming Model

Both the programming language X10 and the Java framework APGAS are based on the *Asynchronous Partitioned Global Address Space* parallel programming model [22]. Usually, this term is abbreviated *APGAS*. Since the Java framework is also called *APGAS*, we instead use the term *X10 programming model* throughout the thesis, whereas *APGAS* always denotes the APGAS framework for Java.

The X10 programming model assumes a global partitioned address space as mentioned in Chapter 1. Every place has its own part of memory, but can directly access every other part of memory, as well. Thereby it is not necessary to send messages to other places as in MPI. Local access is faster than remote access, but occasionally remote access is necessary. The programmer controls how the program data is mapped to the places, and thus how often remote access arises.

The X10 programming model expands the older and much better known PGAS programming model with activity-asynchronism. It was developed specifically for the programming language X10. PGAS is also used by other parallel languages, such as Fortress and UPC. In addition to features of the original PGAS model, the X10 variant offers the opportunity to create new activities at runtime asynchronously. Therefore, programmers are able to design parallel algorithms quite flexibly.

## 2.2. Programming Language X10

X10 is a parallel programming language, which is object-oriented and class-based. It offers single inheritance and a garbage collector. The basic syntax of X10 is very similar to that of Java. The X10 programming model forms the basis for asynchronous parallel programming. X10 was designed with the goal to boost the programmer's effectiveness in parallel programming by a factor of 10, as compared to conventional parallel languages, hence the name X10 has arisen.

X10 is being developed by IBM since 2004. It was sponsored by the High Productivity Computing Systems (HPCS) project from the Defense Advanced Research Projects Agency (DARPA). It is open source and licensed under the Eclipse Public License 1.0[1]. While working on this thesis, the latest official Version was 2.5.3, but the language is still under active development. Therefore, new features are added occasionally and older versions may no longer be maintained. The complete code, including recent updates outside an official release, is accessible to everyone from the official git repository [14].

X10 offers two compilers. One compiler uses C++ (called *Native X10*), and the other uses Java (called *Managed X10*). An X10 program is compiled in two steps, first to C++ or Java, and then to byte code.

## 2.3. APGAS Framework for Java

The APGAS framework is written in Java 8 and brings the main functionalities and features of the X10 programming model to Java. The framework "supports resilient, elastic, parallel, distributed programming on clusters of JVMs" [26]. It is provided as a library in a compiled jar-file for Java, and can also be used with the

[1] `http://opensource.org/licenses/EPL-1.0`

functional programming language Scala [23]. One main motivation for developing the framework was to make the X10 programming model accessible to a wider audience, whereas X10 is so far primarily used for research. Since the APGAS framework requires lambdas, Java Version 8 is necessary.

APGAS is developed by the X10 team and is part of the X10 project. Just like X10, it is open source and licensed under the Eclipse Public License 1.0. In June 2015, Version 1.0 was released. Like X10, APGAS is still under active development and the current unreleased version is available in the official X10 git repository [14]. The master thesis at hand uses the commit status in the official X10 git repository from December 15, 2015. It was necessary to use a recent unofficial release because Version 1.0 has some starting issues on clusters, which were fixed after our bug report [28].

APGAS utilizes three third-party libraries. The first is the open source framework *Hazelcast* [11] in Version 3.5.2. Hazelcast is based on Java and realizes an in-memory data grid. APGAS uses Hazelcast for cross-JVM communication and shared storage, on an elastic, resilient and distributed collection of JVMs.

Since October 2015, APGAS uses *Kryo* [7] in Version 3.0.3, which is an open source framework for optimized serialization. With that, Java objects can be copied more efficiently from place to place than with the conventional Java serialization. Using Kryo is optional and has to be enabled with a program argument. The support for Kyro is currently experimental and was not used for this thesis.

A third library is *Objenesis* in Version 1.0, which is required by Kryo. It is a small open source Java library to dynamically instantiate new objects by various reflection species, which Java does not support. For example, Objenesis can instantiate objects from classes which do not have a public constructor.

The APGAS framework has some ambitious goals for the future, which may increase popularity and practical applicability. In the short term, a new launcher should allow starting programs with *Apache Hadoop Yarn* [1]. Yarn stands for

*Yet Another Resource Negotiator* and is an application management framework for *Apache Hadoop*, which is an open source project for scalable distributed computing in clusters. Hadoop uses the established MapReduce-algorithm [5] and is specialized in big data, especially in the petabyte range. Yarn allows other engines, such as APGAS, to run on a cluster. A first experimental Yarn launcher implementation for APGAS is available.

A long term goal of APGAS is its integration into *Apache Spark* [2]. Spark is a modern open source framework for cluster computing. It can be used with different programming languages and file systems. Programs that utilize Spark can be executed in memory, and thus 100× faster than with Hadoop MapReduce.

Another long-term goal is an improvement of resiliency and elasticity of APGAS. Currently, these features rely on Hazelcast, which has some disadvantages. For example, a crash of two or more places at the same time cannot be handled. X10 does not have these disadvantages.

A last goal is a deeper integration into the Eclipse IDE. There is already a plugin available for Eclipse called *APGAS Development Tools for Eclipse.* It includes the APGAS Runtime, APGAS Compiler Warnings and APGAS Development Tools. After installing the plugin, programmers can instantly use APGAS in Java and do not have to worry about dependencies. Additionally, warnings and suggestions for improvements are shown live in the written code. This increases significantly the comfort during development. These features will be further improved, to show even more and better warnings and suggestions. However, the tools are currently available for APGAS Version 1.0 only.

## 2.4. Constructs and Keywords

This section describes some key constructs of X10 and APGAS and compares them with each other. The descriptions refer to the commit status in the official X10 git repository from December 15, 2015. Java is used in official Version 8 with Update 66.

The syntax of APGAS is similar to the syntax of X10 because the APGAS constructs have been inspired by the X10 constructs, and the basic syntax of X10 has been inspired by Java. However, APGAS uses lambdas for realizing parallel constructs, whereas X10 uses language constructs. Other differences will be presented as needed in the following paragraphs. Finally, a parallel HelloWorld-program is presented in both systems.

### Place

Places exist in X10 and in APGAS. A place is typically a set of local computational units, such as a multicore processor or cluster node, together with a finite amount of shared memory. Alternatively, multiple places can be simulated on a single machine, to simplify program development. Programmers may switch between places by using the keyword `at`. On the operating system level, a place corresponds to a system process.

In X10, the number of places can be configured by setting an environment variable, named `X10_NPLACES`. APGAS provides the JVM option `-Dapgas.places`, which can be set when starting an APGAS program.

In order to transmit objects over the network to other places, they have to be serializable. In X10, all objects are automatically serializable. APGAS is based on Java, and Java objects are not automatically serializable. Therefore, APGAS programmers have to implement Java's interface `java.io.Serializable` in the corresponding classes. The APGAS Development Tools support them in doing so.

### Activity

In both X10 and APGAS, a place can run several activities. Activities can be described as lightweight threads. They have no names, in contrast to Java threads, and execute a specified block of code. Each program has at least one activity, called *root activity*, which starts executing the main method. When the root activity terminates, the whole program finishes.

On the operating system level, an activity is mapped to a system thread, which belongs to the process of the respective place. A minimum number of concurrently running system threads can be set in X10 with the environment variable `X10_NTHREADS`, and in APGAS with the JVM option `-Dapgas.threads`. By default, the number is initialized with the number of available processor cores. The X10 Runtime starts the corresponding number of threads at its invocation and holds them in a pool.

The thread count on operating system level can differ from the value of `X10_NTHREADS` or `-Dapgas.threads`. Atomic constructs in X10 can suspend activities. In such situations, the X10 Runtime spawns a new thread, which can execute an activity. Moreover, an APGAS program starts many threads for administration, e.g. for the garbage collector or for destroying the JVM. This is not a characteristic of APGAS, is but done automatically by Java. An X10 program starts significantly fewer administrative threads, but, for example, resiliency requires one.

### async

The keyword `async`, followed by a code block, starts an activity. If there is an idle system thread available in the pool, the block is executed instantly, otherwise it has to wait until the local scheduler assigns it to a thread. The `async` construct is available in both systems. In X10, activities can be interrupted with

`Runtime.probe()`, so that the calling thread can execute all pending activities. APGAS does not provide a similar function to `Runtime.probe()`.

### finish

The `finish` construct can be used in X10 and APGAS to wait for all activities, spawned in a block. Only when all activities, including recursively spawned activities, have been terminated, the code after the finish block will be executed.

### Uncounted

The `uncounted` parameter can be combined with an `async`. It indicates that a surrounding `finish` should ignore the corresponding activity, including any exceptions that the activity may raise. Both X10 and APGAS, provide this parameter.

### Array

X10 supports the `rail` construct, which corresponds to an `array` in Java, except that the index type is `long` in X10, and `int` in Java.

### Exceptions

X10 exceptions work the same way as Java exceptions, for the purpose of this thesis. X10 and APGAS implement some new exception types, for example a `DeadPlaceException`, which is raised after a place crash.

## Variables

Listing 2.1 and Listing 2.2 show examples of variable declarations.

```
var j : Long = 1;
val i = 2;
```

Listing 2.1: X10: variables

```
long j = 1;
final int i = 2;
```

Listing 2.2: APGAS: variables

A `var`-variable in X10 resembles a usual variable in Java. A type of this variable must always be provided.

Besides syntax, X10's `val` differs by type inference from Java's `final`, i.e. the type may be omitted if clear from context. In place changes, a deep copy is automatically generated for `val`-variables. This copying is called *autoboxing*. It implies that if the variable is used at the new place, changes are not visible at the original place. This feature is helpful and often used. In contrast, `var`-variables are not copied. An APGAS programmer has to box a variable manually using an array, to make deep copying work. Examples are shown in Listings 2.3 and 2.4.

```
val i:int = 5;
finish {
  at (p) {
      async Console.OUT.println(i);
  }
}
```

Listing 2.3: X10: deep copy

```
int[] i = new int[]{5};
finish(() -> {
  asyncAt(p , () -> {
      System.out.println(i[0]);
    });
});
```

Listing 2.4: APGAS: deep copy

## Concurrency control

An `atomic`-block in X10 creates a critical section, i.e. the code is executed atomically and exclusively. APGAS does not provide an `atomic` construct and relies on the concurrency constructs of Java instead. Java provides several options to realize critical sections. This thesis uses the Java keyword `synchronized` to protect code blocks and the Java classes `AtomicBooolean` and `ConcurrentLinkedQueue` as data types for variables.

For locking, the Java keyword `synchronized` works with an object, see line 1 in Listing 2.5. The Java class `Object` provides the methods `wait()` and `notifyAll()`, which enable communication between Java threads in locking situations. With the help of an application sample with two Java threads, this communication is described below. Listing 2.5 shows a Java thread that waits until `condition` becomes `true`. Listing 2.6 shows another Java thread, which sets the `condition` to `true` and wakes up the Java thread from Listing 2.5. Both listings use the shared variable `condition`.

Let the left Java thread start first. Then it takes the lock of the object `lockObject` in line 1 and enters the critical section. Thus, while the variable `condition` in line 2 has the value `false`, the method `wait()` is called on the object `lockObject`. This method releases the lock, and thus another thread can access a critical section, which is marked by `lockObject`. We assumes that the Java thread in Listing 2.6 takes the lock of `lockObject` and accesses the critical section in line 1. Then, in line 2 the variable `condition` is set to `true` and the method `notifyAll()` is called on `lockObject`. This method wakes up all the threads that called `wait()` on the same object. When the thread from Listing 2.5 wakes up, the `while` loop in line 2 finishes.

```
1 synchronized(lockObject) {
2   while(condition == false) {
3     lockObject.wait();
4   }
5 }
```

Listing 2.5: APGAS: wait()

```
1 synchronized(lockObject) {
2   condition = true;
3   lockObject.notifyAll();
4 }
```

Listing 2.6: APGAS: notifyAll()

The Java class `AtomicBooolean` provides a thread-safe `boolean` variable. This variable can be set with the method `set()`, and read with the method `get()`. In addition, the method `compareAndSet()` sets the value to a passed value if the current value equals another passed value. Furthermore, the method `getAndSet()` sets the passed value and returns the old value.

The Java class `ConcurrentLinkedQueue` is a thread-safe collection of a generic type. It stores its elements in *FIFO order (first-in-first-out)*. The method `add()` inserts an element, whereas the method `poll()` removes the last element and returns it. These method calls are thread-safe and guarantee data integrity.

## Global Heaps

X10 and APGAS support global heap references in different forms. In X10, the classes `GlobalRef[T]` and `PlaceLocalHandle[T]` exist, where `T` represents the type of the object being referred to. A `GlobalRef[T]` is a global reference to a single object. Remote places can access this object by moving to its home place and modifying its value there. A `PlaceLocalHandle[T]` is an abstract reference to place-local information of type `T`. For this, an object of type `T` has to be created for each place. It can be resolved by different activities to obtain access to local information stored in different places.

In APGAS, the class `GlobalRef<T>` exists, which is a union of X10's `GlobalRef` and `PlaceLocalHandle`. APGAS provides other global heap constructs, for example, `GlobalID` and `PlaceLocalObject`, but only `GlobalRef<T>` is used in this thesis.

## Hello World

Listings 2.7 and 2.8 depict two analogous parallel HelloWorld-programs, written in X10 and in APGAS, respectively. The programs iterate over all available places, starting an asynchronous activity at each place. The activities write out a message including their place number.

```
class HelloWholeWorld {
  public static def main(args:Rail[String]):void {
     finish for (p in Place.places()) {
      at (p) async Console.OUT.println("Hello␣from␣" + here);
     }
  }
}
```

Listing 2.7: X10: Hello whole world

```
class HelloWholeWorld {
  public static void main(String[] args) {
    finish(() -> {
      for (final Place place : places()) {
        asyncAt(place, () -> System.out.println("Hello␣from␣" + here()));
      }
    });
  }
}
```

Listing 2.8: APGAS: Hello whole world

## 2.5. Execution

X10 and APGAS programs can be executed with different options for the communication between places and remote starting. In X10, these options have to be already set when the compiler is built. Moreover, they also have to be set again when compiling the user program. In APGAS, the options have to be set when starting the user program, using the JVM options.

X10 uses a runtime library, called *X10RT*, which is responsible for the communication between places. X10RT offers four options: *sockets*, *standalone*, *MPI* and *PAMI*. When compiling a user program, an option can be selected with the parameter `-x10rt <sockets | standalone | mpi | pami>`. *Sockets* is the default value. It uses TCP/IP sockets for communication between places and the communication protocol SSH for processing the startup. The environment variable `X10_HOSTLIST` contains a list of hosts for the places. The host list is cyclically repeated up to the value of the environment variable `X10_NPLACES`.

*Standalone* starts all places on the local machine. The number of places is taken from `X10_NPLACES`. With *MPI*, X10 utilizes MPI functionalities for communicating and distribution. In addition to the compiling option, the X10 compiler has to be built with `-DX10RT_MPI=true` to enable MPI. After compiling the program, it can be executed with the command `mpirun`. MPI has to be installed on the system. This option is recommended for execution in a cluster. The last option *PAMI* is a communication API from IBM. The X10 compiler has to be built with `-DX10RT_PAMI=true`. PAMI supports high-end networks and has to be installed on the system.

APGAS provides different launchers to start a program in various ways. The launcher has to be set with the JVM option `-Dapgas.launcher`. Valid values refer to an available launcher class in APGAS. Current options are `apgas.impl.LocalLauncher`, `apgas.impl.NoLauncher` and `apgas.impl.SshLauncher`. The `LocalLauncher` is set as default and spawns places only on the localhost. In contrast, the `NoLauncher` starts the program with only one place. Additional places have to be started and connected manually.

The `SSHLauncher` requires a host file, which contains all available hosts. It has to be set with the JVM option `-Dapgas.hostfile`. At first, the program is started with one place on localhost, called *master*. Then the other places are launched on all other hosts using SSH. After all places have connected, the parallel part in the user program starts.

# 3. Global Load Balancing

In order to use parallel systems effectively, tasks have to be mapped fairly to all places. This is particularly important for irregular applications, for which the number and size of tasks is not known at the start. This chapter handles global load balancing [21], which enables automatic inter-place load balancing. The X10 project includes an implementation of global load balancing in its GLB framework [30].

Section 3.1 describes GLB, referring to the X10 implementation, and illustrates its workflow with a flow chart. Afterwards, Section 3.2 presents two well-known benchmarks for parallel systems, and their implementations in X10.

## 3.1. Concept of GLB and Implementation in X10

Many parallel algorithms split an initial problem into multiple smaller tasks. If the tasks do not depend on each other, they can be processed concurrently at different places. Especially in recursive algorithms, the calculation of a task may give rise to several new tasks, which have to be distributed at runtime. Often, the final result is calculated from partial results of the tasks. The aggregation of these partial results to one result is called *reducing*. If the reduce operator is commutative and associative, the tasks can be processed in any order.

If places are running out of work at runtime, tasks have to be distributed dynamically. There are two established techniques for dynamic task distribution: *work sharing* and *work stealing* [4]. Both techniques utilize *workers* for processing tasks. In the GLB implementation of X10, a worker corresponds to a place. Each worker holds its tasks in a data structure, called *task pool*. Workers process off their tasks and calculate, after each processed task, their partial local results.

When all workers have finished, the partial results are reduced to one final result. Afterwards, the program finishes.

In a variant of *work sharing*, each time a worker generates new tasks, the scheduler sends some of them to other workers. The current workload of the other workers is not considered, assuming that the work has been uniformly distributed. In *work stealing*, in contrast, a worker with an empty task pool, called *thief*, tries to steal tasks from other workers, called *victims*. If a victim has too few tasks, nothing can be stolen and the steal request fails.

In parallel systems with distributed memory, non-successful work-stealing requests cause a lot of network traffic, resulting in a significant decrease in performance. This happens particularly in the final phase, when only a few tasks are left. Termination detection, i.e. recognizing when all workers have finished their tasks, is another difficulty in work stealing and in work sharing programs.

The GLB implementation from the X10 project [30] employs a cooperative variant of work stealing [21], in combination with a lifeline scheme. This variant was designed for parallel systems with distributed memory. Here, thieves send steal requests to selected victims. A victim occasionally interrupts processing tasks and works off these requests. Requests are answered by sending tasks or a reject message. In the X10 implementation, this cooperative variant was accomplished with a significant restriction: only one running activity per place is allowed. Thus, there is no synchronization necessary within a place.

The lifeline scheme is a central part of the GLB concept and is based on a lifeline graph. The graph is typically a $w$-dimensional hyper-cube, which is used for victim selection and termination detection. A hyper-cube is a directed graph. if $w$ equals to 2, the hyper-cube is an analogue of a square, if $w$ equals 3 the hyper-cube is an analogue of a cube and so on. Nodes represent workers, and edges are called *lifelines*. Each worker is connected by lifelines to other workers, called *lifeline buddies*. In GLB, if a lifeline is used, it is enabled, otherwise it is disabled.

If a thief has no more tasks in its task pool, it sends steal requests to random victims. If all of these requests are rejected, the thief gradually activates its lifelines. After an activation, the thief sends a steal request to the corresponding lifeline buddy. The lifeline buddy either sends tasks back, or rejects the request. When rejecting the request, the requesting worker is saved if it is a lifeline thief. A thief always waits for an answer. If a steal request was successful, the thief disables the lifeline again and continuous working on its tasks. If a steal request was unsuccessful, it activates the next lifeline and sends a corresponding steal request. Only if all lifeline buddies of a thief have rejected the steal requests, the worker activity ends. In case of a lifeline buddy getting new tasks and having queued lifeline requests, it shares tasks with the corresponding workers. However, if they have already finished, the lifeline buddy restarts the workers by invoking a new activity on their place.

The overall computation is completed when all worker activities have ended. This is noted by an surrounding `finish`. In that case, the root activity computes the final result by collecting and reducing all worker results.

GLB is implemented as a framework, which hides from a GLB user the dynamic load balancing between places, termination detection and result collection. The framework deploys tasks and their results as generic types. Programmers can use it easily because they only have to implement some required classes, interfaces and their methods.

## Flow Chart

Figure 3.1 illustrates a simplified workflow of GLB from the perspective of a worker (adapted from Perez [19]). Position $*1$ is triggered by a lifeline buddy, not by the original worker. Position $*2$ is managed by the root activity of the framework.

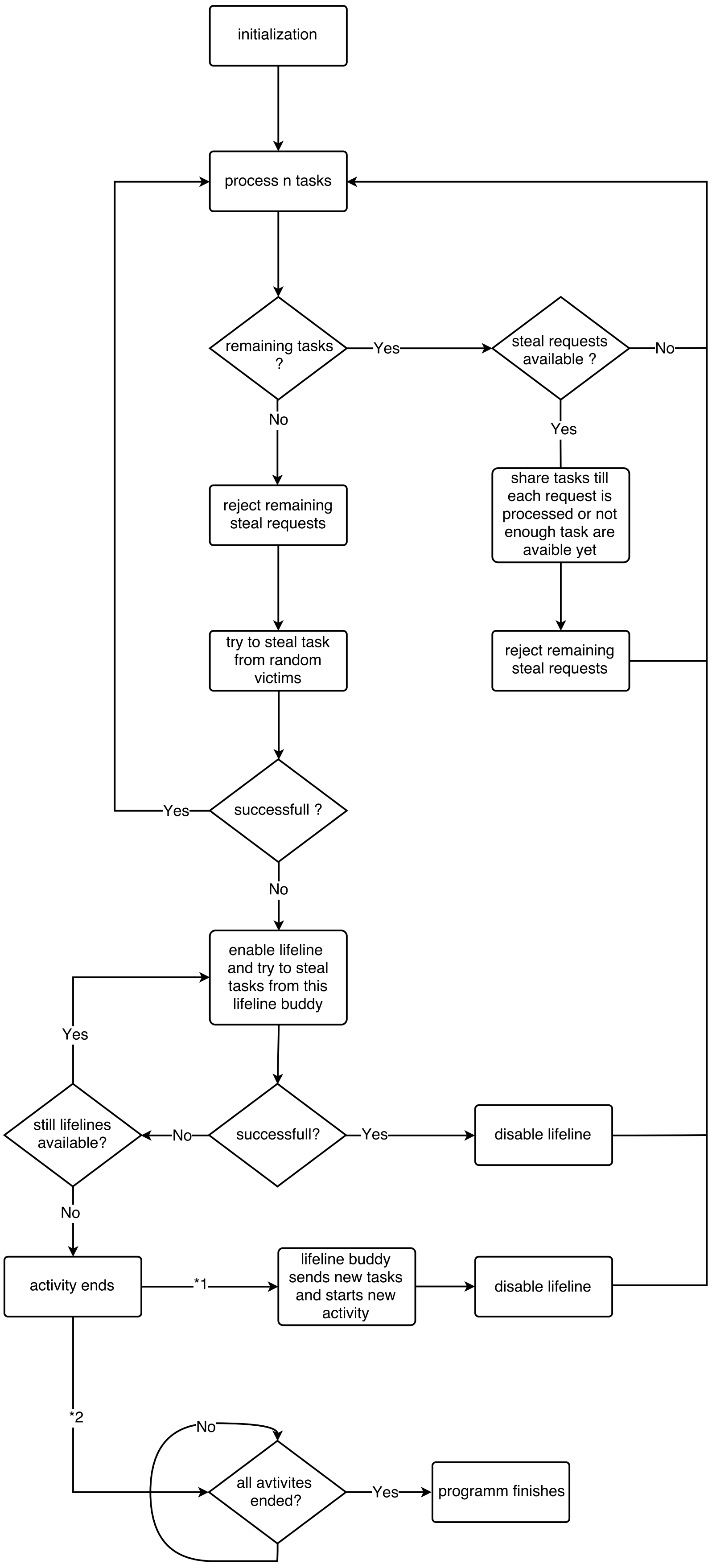
initialization
process n tasks
remaining tasks ?
Yes
No
steal requests available ?
No
Yes
share tasks till each request is processed or not enough task are avaible yet
reject remaining steal requests
reject remaining steal requests
try to steal task from random victims
successfull ?
Yes
No
enable lifeline and try to steal tasks from this lifeline buddy
Yes
still lifelines available?
No
successfull?
No
Yes
disable lifeline
activity ends
*1
lifeline buddy sends new tasks and starts new activity
disable lifeline
*2
No
all avtivites ended?
Yes
programm finishes

Figure 3.1.: Simplified flow chart of a GLB workflow

## 3.2. Benchmarks

Two well known benchmarks for parallel systems are *Unbalanced Tree Search (UTS)* and *Betweenness Centrality (BC)*. The X10 project provides implementations for both in X10. Two of the implementations utilize GLB and demonstrate the usage of GLB from a programmer's perspective. Moreover, they can be used for benchmarking parallel systems with respect to performance and scalability.

We did not deploy the official implementations, but slightly modified variants of them [8]. These variants store tasks explicitly instead of using multiple arrays as in the official version. The modified variants are well suited for our adjustments, as explained in Chapter 5.

The following Sections 3.2.1 and 3.2.2 describe the benchmarks and their implementations in X10.

### 3.2.1. Unbalanced Tree Search

The UTS benchmark was introduced in 2006 by Olivier et al. [17]. It calculates all nodes in an unbalanced tree, starting from a root value. Tree properties can be configured with program arguments, which include the branching factor $b$, the root node $s$ and the tree depth $d$. These values strongly affect the execution time. The final result of a program execution is the number of nodes in the tree. The calculation utilizes the *secure hash algorithm (SHA1)*, which is deterministic. This means, if the initial values and parameters are the same, the generated tree and, thus, the result is always the same. A calculation of a node generates new nodes recursively. Therefore, at the start of the program, the total number of nodes is unknown, and every node has to be calculated to obtain the final result. A tree node is represented by a task. UTS is well-suited to simulate load imbalance and evaluate dynamic load balancers such as GLB.

### 3.2.2. Betweenness Centrality

The BC benchmark was first described by Freeman in 1977 [9]. It calculates a BC score for each node in a graph, which rates the centrality of the corresponding node. A high BC score indicates that the corresponding node is part of many shortest paths.

The official X10 variant implements a variant from Bader et al. [3]. It provides some graph configuration opportunities, for example an initial seed $s$ and an exponent $n$ for generating a graph with $2^n$ nodes. Since, all graph nodes are known in advance, GLB is statically initialized and each worker is assigned about the same number of graph nodes. Each worker calculates the BC score for its graph nodes.

# 4. Implementation of GLB in APGAS

This chapter deals with our implementation of GLB in APGAS. From now on, it will be called *APGAS_GLB* and the official X10 implementation will be called *X10_GLB*.

The design of APGAS_GLB was adopted from X10_GLB. We tried to implement APGAS_GLB as similarly to X10_GLB as possible. However, one essential implementation detail had to be adjusted: X10_GLB allows only one thread per place and interrupts cyclically a running activity so that pending activities can be executed. Since APGAS does not provide a corresponding functionality (see Section 2.4), we allowed multiple threads per place, and synchronize access to shared variables via synchronized blocks which are described in Section 2.4. Thus, multiple activities are executed, but only one can be in a synchronized code block. Other activities have to pend until this activity leaves and releases the lock.

This Chapter starts with an overview of APGAS_GLB's basic structure, which is the same as the of X10_GLB, in Section 4.1. Afterwards, Section 4.2 explains the workflow of a worker. Section 4.3 specifies some essential implementation details of APGAS_GLB on the basis of selected code blocks. In Sections 4.2 and 4.3, differences to X10_GLB will be pointed out. Afterwards, Section 4.4 describes two newly added components. One component extends the original class `Logger` with collecting and evaluating execution times of fundamental GLB sections. The other prints optional outputs for finding bugs during development. Finally, Section 4.5 explains the porting of two benchmarks from X10 to APGAS.

## 4.1. Overview

The structure of APGAS_GLB is the same as that of X10_GLB. Figure 4.1 illustrates this structure in a simplified class diagram. Methods and fields that are not important for understanding this thesis have been omitted. Major classes are `GLB` and `Worker`. The class `GLB` is the entry point into the framework, and the class `Worker` represents a worker as introduced in Section 3.1. Hence, the class `Worker` contains all methods for processing and stealing tasks. The following paragraphs describe all individual classes and interfaces of APGAS_GLB. Changes to X10_GLB are explained when necessary.

### `GLBParameters`

The class `GLBParameters` stores parameters for the framework:

- `n`: number of tasks in an execution batch,
- `w`: number of random steal requests,
- `l`: dimension of the lifeline hypercube,
- `z`: maximum count of lifeline buddies per worker,
- `m`: maximum count of random victims per worker,
- `v`: verbose level,
- `timestamps`: fineness of logging component, described in Section 4.4.1.

### `TaskBag`

`TaskBag` is an interface, and a GLB user has to implement it in an individual class. Objects of this class are used for transferring stolen tasks between places. The interface `TaskBag` only contains the abstract method `getSize()`, which returns the number of tasks in `TaskBag`.

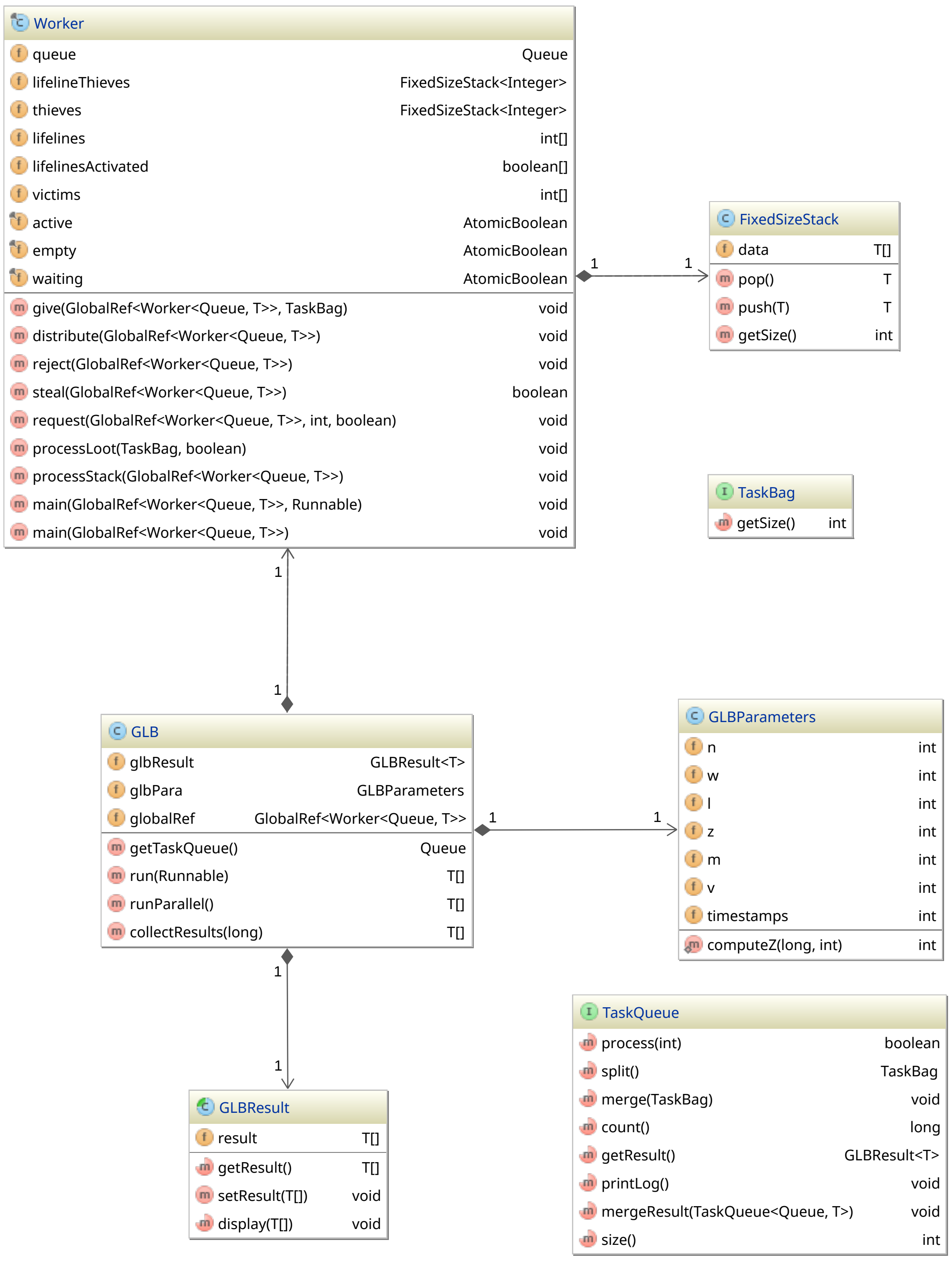

Figure 4.1.: Simplified class diagram of GLB

### `GLBResult`

`GLBResult` is an abstract class of a generic type. It has a field `result`, which is an array of the generic type. Often the result contains only one element, but the benchmark BC, for example, has multiple final results. It is imaginable that results could have different types, but neither X10_GLB nor APGAS_GLB support that. A GLB user has to implement an individual class, which extends `GLBResult`, and instantiates the generic type with a concrete data type.

### `TaskQueue`

A GLB user needs to extend the interface `TaskQueue` with an individual class, which is utilized as a type for the local task pool. This design decision was fully taken from X10_GLB and brings with it advantages and disadvantages for GLB users. On the one hand, they can design their task pools according to their own vision. On the other hand, they are obliged to implement the task pool without support from the framework.

The interface `TaskQueue` has two generic types. The first generic type instantiates the type of the data structure for storing tasks and the second instantiates the type of `GLBResult`. As we will see below, each worker will maintain one object of type `TaskQueue`. The interface contains essential abstract methods, for example, `process()`, which works off $n$ tasks. Moreover, the result can be queried by calling the method `getResult()`, which returns an object of type `GLBResult`. Each worker has one result, which is updated after processing $n$ tasks. The method `merge()` merges stolen tasks in the local task pool.

### `FixedSizeStack`

A worker provides two fields of type `FixedSizeStack`, which contain thieves and lifeline thieves. At a steal requests, a thief saves itself in one of these fields. The class `FixedSizeStack` represents a primitive data structure and stores a fixed number of objects of a generic type in an array. APGAS_GLB initializes the generic type always with `Integer`, but the class `FixedSizeStack` could also be used for other purposes. The size of `elements` does not need to be resized because the number of thieves and lifeline thieves are known at program start and does not change in runtime. The class `FixedSizeStack` simply provides the methods `pop()`, `push()` and `getSize()` with their expected functionalities. Possible exceptions do not get caught.

### `GLB`

This class is the entry point to the framework, and a GLB user has to initialize an object of this class. The corresponding constructor instantiates and initializes one object of the class `Worker` on each place. A constructor call requires an object of `GLBParameters` and an object of `SerializableCallable`. The second one is responsible for initializing an instance of `TaskQueue` on each `Worker` object. The class `Worker` is described below. Details about starting these workers are given in Section 4.3.

In X10_GLB, the class `GLB` contains a field `plh` of type `PlaceLocalHandle` which is responsible for saving one worker on each place. As explained in Section 2.4, APGAS does not provide the class `PlaceLocalHandle`. Instead, we renamed the field `plh` in the class `GLB` to `globalRef` and changed the type of it to `GlobaRef`. As stated in Section 2.4, `GlobalRef` is a union of X10's `GlobalRef` and `PlaceLocalHandle`.

Moreover, the parallel calculation can be started with the method `run(Runnable)` or `runParallel()`. If only a few tasks are known at program start and the remaining tasks can only be generated at runtime, the method `run(Runnable)` should be used. Otherwise, `runParallel()` should be used. The method `run(Runnable)` dereferences `globaRef` to get the initialized `Worker` object. Then, it calls the method `main(Runnable)` on this object. The method `main(Runnable)` only starts the worker on the first place. The other workers on the other places are started via the lifeline scheme. For details see below.

In contrast, the method `runParallel()` starts a worker on each place because the tasks have already been distributed. Starting a worker is realized with a call of the method `processStack()` on a `Worker` object. Details of this method are explained in Section 4.3.

Furthermore, the class `GLB` provides methods for collecting and reducing the partial results and statistics. The reduce operator for the partial results has to be specified from the GLB user in the method `mergeResult()` from the class which implements the interface `TaskQueue`. Depending on the verbose level, statistics are printed out. These methods are called automatically after the calculation has finished.

### `Worker`

An instance of the class `Worker` represents a worker according to the description in Section 3.1. At program start, on each place starts only one activity, which executes the worker. This activity is also called *worker activity*. The class `Worker` provides all methods for processing and stealing tasks.

A worker has three statuses, which are represented by the fields `active`, `empty` and `waiting` in the class `Worker`. Each of these fields has the type `AtomicBoolean`, see Section 2.4. In contrast, X10_GLB uses `boolean` as type for them. The reason for this adjustment will be explained below.

The field `active` indicates whether the worker is running. It is initialized with `false`. The field `empty` is `true` if there are no tasks left in the local task pool and otherwise `false`. It is initialized with `true`. If the worker waits for an answer to its steal request, the field `waiting` is `true`, otherwise `false`. It is initialized with `false`.

We adjusted the type of the field `empty` to `AtomicBoolean` because we allow multiple activities per places. The field `empty` is accessed in the method `processStack()` and also when one or more lifeline buddies send tasks. Since it is possible that a lifeline thief executes its worker activity while it gets tasks from one or more lifeline buddies, the field `empty` can be accessed concurrently.

Moreover, X10_GLB adds X10's keyword `volatile` to the fields `active`, `empty` and `waiting`. In this way, wrong compiler optimizations can be prevented. An object of the class `AtomicBoolean` is automatically `volatile`.

Furthermore, the field `waiting` needs to be an object because we used it as a lock object for synchronized blocks, as explained in Section 2.4.

The class `Worker` contains fields to save incoming steal requests, its own random victims and lifelines to the corresponding lifeline buddies. The field `thieves` has the type `FixedSizeStack` and saves thieves that send stealing requests. If a lifeline thief is rejected, it will be saved in the field `lifelineThieves`. The field `lifelineThieves` has the type `FixedSizeStack` and contains lifeline thieves which may have already finished. The constructor of the class `Worker` calculates its random victims and lifeline buddies and stores them in the corresponding fields `victims` and `lifelines`.

If tasks are only generated at runtime from an initial task, the first generated tasks have to be distributed to all workers. To realize this initial task mapping, the constructor of the class `Worker` generates lifeline steal requests. Each started worker answers these requests by sending tasks to the requesting worker and starts it, if necessary. Thus, one worker after another is started at the beginning of the program.

In X10_GLB, the constructor of the class `Worker` creates for each worker only three lifeline steals for the initial task mapping. With a high number of places, some places could never get tasks because the recursive task stealing stops when the count of tasks becomes too low. To prevent this, we modified this in APGAS_GLB, and generate the maximum count of lifeline steal requests.

The methods `run()` and `run(Runnable)` are the entry points to the class `Worker`. They call `processStack()`, which controls the workflow of a worker. This workflow is described below in the Section 4.2. Implementation details are given in Section 4.3.

## 4.2. Workflow of a Worker

A typical workflow of a worker is illustrated in Figure 3.1 in Section 3.1 on page 20. This section describes the workflow on the base of the implemented methods from APGAS_GLB. In this process, differences to X10_GLB are shown when necessary. Implementation details of these methods are given in Section 4.3.

The method `processStack()` is responsible for the workflow of a worker. Each worker is started with a new activity, which calls the method `processStack()`. These activities are started with an `asyncAt`. All activities which start a new worker, are surrounded by one `finish`. The surrounding `finish` belongs to the first started place and detects if all workers have finished. After all workers have finished, the partial results are merged to one final result. During steal processes, activities are started with an `uncountedAsyncAt`, except for one essential situation, which uses an `asyncAt`. By starting activities with `uncountedAsyncAt`, the surrounding `finish` from above does not wait for those uncounted activities because they are not relevant for detecting if all workers have finished. The aforementioned `async` is located in the method `give()` and is explained below. Moreover, Listing 4.7 on page 41 shows its implementation.

The fields `active`, `empty` and `waiting` are essential for the workflow. When the workflow starts, `active` has the value `true`, `empty` has the value `false` and `waiting` has the value `false`.

First, $n$ tasks are processed by calling the method `process()` repeatedly as long as there are tasks left in the local task pool. A GLB user has to implement the method `process()` in its task pool class, which will be called `Queue` from now on. Between each call of the method `process()`, stealing requests are answered.

In X10_GLB, the method `Runtime.probe()` is called to execute pending activities. Several of these pending activities want to send steal requests. APGAS_GLB uses synchronized blocks instead of `Runtime.probe()`. Thereby, pending activities are executed automatically as soon as the corresponding lock object is available. The synchronized code blocks are shown in the listings in Section 4.3.

Then the method `distribute()` is called. This method is responsible for successively sending tasks to all saved steal requests, until all of these requests are answered or there are no tasks left to be sent. Afterwards, the method `reject()` is called to reject any leftover requests. Details of both methods `distribute()` and `reject()` are explained below.

If there are no tasks left for processing, the repeated calls of `process()` end. In APGAS_GLB, the field `empty` is set to `true`. In contrast, in X10_GLB this is done by the method `steal()`. The reasons for this adjustment are given in Section 4.3.

Afterwards, the method `steal()` is called. The method `steal()` repeatedly sends steal requests to victims, first to random victims and, when all of these fail, to lifeline buddies. Before each sending, the field `waiting` is set to `true`. The sending is realized by starting a new activity on the victim, which then calls the method `request()`. Thereafter, the thief waits until the victim sets the field `waiting` to `false`.

If the whole method `steal()` has no success at stealing, the field `active` is set to `false`. Afterwards, the worker activity ends.

The method `request()` checks first if the victim is empty or waiting. If at least one is `true`, a new activity on the thief is started, which sets `waiting` to `false`. In this way, the thief continues with sending steal requests. Moreover, lifeline thieves are saved, so eventual tasks can be sent to them later. If the victim has tasks left and is not waiting, the request is saved. Those saved requests are answered later by the method `distribute()`, details are described below.

The method `reject()` rejects one saved steal request after the other. A rejection starts a new activity on a thief. This activity sets the field `waiting` to `false`. Thereby, the thief continues with sending steal requests.

The method `distribute()` repeatedly calls the method `split()` from the class `Queue` as long as there are saved steal requests. The method `split()` has to be implemented from the GLB user in its class `Queue`. The method `split()` extracts an object of `TaskBag` from the task pool if there are tasks left, and returns them. Those tasks are sent to the victim and are called *loot*. If the splitting was successful the method `give()` is called, passing the loot.

The method `give()` starts a new activity on the thief and then calls the user-defined method `deal()`, also passing the loot. If the thief is a lifeline thief, the activity is started with an `asyncAt` and not with an `uncountedAsyncAt`, see essential situation above. This `asyncAt` is necessary because a lifeline thief could have already finished. Then the corresponding worker has to be restarted and in the surrounding `finish`, see above.

The method `deal()` sets the field `active` to `true` and calls the method `processLoot()`, also passing the loot. In turn, the method `processLoot()` calls the method `merge()` from the class `Queue`, also passing the loot. The GLB user has to implement the method `merge()`, which adds the loot to the local task pool. Afterwards, the method `processLoot()` sets the field `empty` to `false` and returns. After calling the method `processLoot()`, the method `deal()` calls the method `processStack()` if the worker has already finished. Thus, lifeline buddies can restart lifeline thieves when they send tasks to them.

## 4.3. Specific Details

This section explains some specific details of the APGAS_GLB implementation. For a better understanding some code blocks are shown in the listings. Adjustments to the X10_GLB implementation are pointed out, when necessary.

As mentioned in Section 2.4, in X10, all objects are automatically serializable. In APGAS, in contrast, respective classes have to implement the official Java interface `Serializable`. Each class/interface from APGAS_GLB implements this interface, see Section 4.1. Therefore, objects of these classes can be written to the file system, but also copied over the network and between places. An implementation of `Serializable` requires an implementation of the constant field `serialVersionUID`. We declared this constant in each class and initialized each with the default value *1L*.

X10_GLB allows only one thread per place and therefore only one activity can be executed per place at a time. As a positive result, there is no place internal synchronization necessary. However, to realize the cooperative work stealing technique, sometimes a worker has to write values into fields at other places, for example when sending a steal request. This kind of actions are executed by starting new activities at remote places. Those activities have to pend until the worker activity calls X10's method `Runtime.probe()`, see Section 2.4. This call is performed periodically. The method interrupts its calling activity (in X10_GLB, it is always the worker activity) and executes all pending activities successively. Afterwards, the worker activity continues.

APGAS does not provide a similar method as `Runtime.probe()`, but realizes activities with Java threads. Java offers the method `yield()`, which gives the scheduler the hint to pause the calling Java thread. However, APGAS does not support an interruption of an activity with `yield()`.

Therefore, we decided to allow multiple threads and activities per place. However, we synchronized code blocks via a lock to control the access to shared variables. Therefore, the outcome remains the same as in X10_GLB. Likewise, only the worker activity processes tasks.

For locking the accesses to the local task pool, the implementation utilizes the Java keyword `synchronized`, see Section 2.4. All code blocks which need access to the local task pool are marked with `synchronized` and use the field `waiting` for locking. The field `waiting` is predestined for being the lock object because it specifies if a worker is waiting. Thereby, the local task pool cannot be accessed concurrently. The synchronized blocks are described in the following paragraphs. Moreover, they are shown in the corresponding listings below.

### `processStack()`

Listing 4.2 shows the method `processStack()` from APGAS_GLB in a simplified form. For comparison, Listing 4.1 shows the original method `processStack()` from X10_GLB. It is a major method of the framework because it controls the complete flow of processing and sharing tasks. Our implementation of the method differs significantly from X10_GLB because we use synchronized blocks instead of a thread interruption like X10's `Runtime.probe()`.

In X10_GLB, the method `processStack()` starts with a `do-while` loop, see line 2 in Listing 4.1. Line 9 contains the condition for the loop: it continues while the method `steal()` returns `true`. When the method `processStack()` ends, its method called `main()` sets the field `active` to `false`. In contrast, in APGAS_GLB, the similar loop continues while the field `active` has the value `true`, see line 18 in Listing 4.2. This condition is logically similar to the condition in X10_GLB. However, this adjustment was necessary because the method `steal()` needs access to the task pool and therefore a call of it had to be synchronized, see lines 15-17 in Listing 4.2. This implies the disadvantage that the method `reject()` has be to called again in line 18 in Listing 4.2 because since the last call, some new steal requests could have been queued.

In X10_GLB, a `while` loop starts in line 3 in Listing 4.1. The loop head contains a call of the method `process()` and thus it runs while there are tasks left for processing. In APGAS_GLB, the loop had to be adjusted because the method `process()` needs access to the local task pool and thereby, a call of it has to be synchronized. The method `process()` cannot synchronize its code by itself, because it has to be implemented in a class of a GLB user. Thus, it cannot access the field `waiting`, which is always used as lock object. So, line 6 in Listing 4.2 shows our solution: The return value of the method `process()` reflects still the condition for the loop, but is used via the additional variable `process`. Moreover, the method `distribute()` could synchronize its containing code itself, but if the synchronization is done in line 7 in Listing 4.2 it is similar to X10_GLB and it improves the comprehensibility.

After this loop ends, the APGAS_GLB variant sets the field `empty` to `true`, see line 12 in Listing 4.2. In contrast, in X10_GLB, this is done with the method `steal()`, which is called in line 9 in Listing 4.1. This had to be adjusted by setting the field directly in the method `processStack()` because other activities can send tasks during the call of `reject()` in line 13 in Listing 4.2.

```
 1  final def processStack(st:PlaceLocalHandle[Worker[Queue, R]]) {
 2    do {
 3      while (queue.process(n, context)) {
 4        Runtime.probe();
 5        distribute(st);
 6        reject(st);
 7      }
 8      reject(st);
 9    } while (steal(st));
10  }
```

Listing 4.1: X10_GLB: Method `processStack()` from the class `Worker`

```java
public void processStack(GlobalRef<Worker<Queue, T>> globalRef) {
  do {
    boolean process;
    do {
      synchronized (waiting) {
        process = queue.process(n);
        distribute(globalRef);
      }
      reject(globalRef);
    } while (process);

    empty.set(true);
    reject(globalRef);
    synchronized (waiting) {
      this.active.set(steal(globalRef) || 0 < queue.size());
    }
  } while (this.active.get());
  reject(globalRef);
}
```

Listing 4.2: APGAS_GLB: Method `processStack()` from the class `Worker`

### `steal()`

A simplified variant of the method `steal()` is shown in Listing 4.3. In line 2, a `for` loop iterates over all random victims, unless the field `empty` gets the value `true`. Before sending a steal request, the field `waiting` is set to `true` in line 3. Then a new activity is started on a chosen victim in line 5. This activity calls the method `request()` in line 6. Afterwards, the origin activity from the method `steal()` waits until the field `waiting` is set to `true`, see lines 8-12. This shown approach is always applied when an activity waits for an answer. In contrast, X10_GLB utilizes the method `Runtime.probe` instead of the synchronized block.

If all random steal requests fail, a comparable `for` loop starts afterwards. It has the same functionalities as the first loop, but sends requests to the lifeline buddies.

After this loop, the method returns `true` if the local task pool is not empty, otherwise `false`.

```
1  public boolean steal(GlobalRef<Worker<Queue, T>> globalRef) {
2    for (int i = 0; i < w && empty.get(); ++i) {
3      waiting.set(true);
4      int v = victims[random.nextInt(m)];
5      uncountedAsyncAt(places().get(v), () -> {
6        globalRef.get().request(globalRef, p, false);
7      });
8      synchronized (waiting) {
9        while (waiting.get()) {
10         waiting.wait();
11       }
12     }
13   }
14   //lifeline~steals
15   return !empty.get();
16 }
```

Listing 4.3: APGAS_GLB: Method `steal()` from the class `Worker`

### `request()`

Listing 4.4 shows the method `request()` in a simplified form. An `if` statement in line 2 checks if at least one of the fields `empty` or `waiting` has the value `true`. If this is fulfilled, the incoming request is rejected with starting an activity on the thief, see line 4. This activity sets the field `waiting` to `false` and calls the method `notifyAll()` on the field `waiting`, see line 6 and 7. With these two operations, the waiting thief can continue (the synchronized block in line 8 in Listing 4.3 finishes). This shown approach is always applied for waking up a waiting activity, which is using `waiting.wait()`.

However, if the request is a lifeline steal, the thief is saved in the field `lifelineThieves`, see line 3. This allows the lifeline thief to get tasks later on. In this way, the lifeline thief can be restarted. Additionally, if the condition in line 2 is `false`, the thief is saved in the field `thieves`, see lines 11 and 12. A random thief is saved as a negative value, see line 12. That way it is possible to distinguished between random and lifeline thieves, if they are rejected later in the method `reject()`. The method `reject()` checks the thief before the rejection. If the thief is a lifeline thief, it is saved to the field `lifelineThieves` in order to send tasks to it later, see method `reject()`.

```
 1 public void request(GlobalRef<Worker<Queue, T>> globalRef, int thief,
       boolean lifeline) {
 2   if (empty.get() || waiting.get()) {
 3     if (lifeline) lifelineThieves.push(thief);
 4     uncountedAsyncAt(places().get(thief), () -> {
 5       synchronized (globalRef.get().waiting) {
 6         globalRef.get().waiting.set(false);
 7         globalRef.get().waiting.notifyAll();
 8       }
 9     });
10   } else {
11     if (lifeline) thieves.push(thief);
12     else thieves.push(-thief - 1);
13   }
14 }
```

Listing 4.4: APGAS_GLB: Method `request()` from the class `Worker`

### `reject()`

Listing 4.5 shows the method `reject()` in a simplified form. In line 2, a `while` loop runs while there are saved thieves left. If the thief is a lifeline thief, it is saved in the field `lifelineThieves`, see line 3. Thus, this thief can be restarted later, when necessary. On each of these thieves an activity is started, see line 5. This activity sets the field `waiting` to `false` and calls `notifyAll()` on the field `waiting` for waking up a waiting activity.

```java
public void reject(GlobalRef<Worker<Queue, T>> globalRef) {
  while (thieves.getSize() > 0) {
    final int thief = thieves.pop();
    if (thief >= 0) lifelineThieves.push(thief);
    uncountedAsyncAt(places().get(thief), () -> {
      synchronized (globalRef.get().waiting) {
        globalRef.get().waiting.set(false);
        globalRef.get().waiting.notifyAll();
      }
    }
  }
}
```

Listing 4.5: APGAS_GLB: Method `reject()` from the class `Worker`

## `distribute()`

The method `distribute()` is shown simplified in Listing 4.6. In line 3, a `while` loop starts. It runs while there are queued steal requests and the method `split()` returns an object of type `TaskBag`. The method `split()` has to be implemented from the GLB user in its task pool class. It takes tasks from the task pool, if there are tasks left, stores them in an object of `TaskBag` and returns this object. If there are no tasks left, it returns `null`. The returned object is assigned to `loot` from line 2. The body of the loop contains only one method call of `give()`, passing `loot`, see line 4.

```java
1 public void distribute(GlobalRef<Worker<Queue, T>> globalRef) {
2   TaskBag loot;
3   while (((thieves.getSize() > 0) || (lifelineThieves.getSize() > 0)) &&
        (loot = queue.split()) != null) {
4     give(globalRef, loot);
5   }
6 }
```

Listing 4.6: APGAS_GLB: Method `distribute()` from the class `Worker`

### `give()`

Listing 4.7 shows the method `give()` in a simplified form. It gets past an object `loot` of type `Taskbag`, which contains stolen tasks. The method starts a new activity on the thief in line 5. This activity enters a critical section in line 6, and calls therein the method `deal()` with passing the variable `loot`, see line 7. The method `deal()` is responsible for merging the loot tasks into the local task pool. Afterwards, the field `waiting` is set to `false` and `notifyAll()` is called on it in lines 8 and 9 for waking up a waiting activity.

If the worker is a lifeline buddy, the `else`-case in line 13 is executed. During stealing, this is the only case in which an `asyncAt` is needed. The corresponding worker has to be restarted if the lifeline thief has already finished. Because of the `asyncAt`, it is handled by the surrounding `finish`, which is responsible for termination detection, see Section 4.2.Moreover, the started activity does not need to set the field `waiting` to `false` because it has already been set in the method `reject()`.

### `deal()`

Listing 4.8 shows the method `deal()` in a simplified form. In line 3, an eventual lifeline is disabled. Afterwards, in line 5, a synchronized block starts. It is a block because both included operations belong together and line 6 is only executed due to the fact that line 7 is executed. Moreover, the call of the method `processLoot()` has to be synchronized because it needs to access the task pool. The method `processLoot()` calls the method `merge()` from the GLB user's task pool class, which in turn adds the passed loot tasks to the task pool.

If the field thief was not active, it is restarted with calling the method `processStack()`, see line 11.

```java
public void give(GlobalRef<Worker<Queue, T>> globalRef, TaskBag loot) {
  int victim = here().id;
  if (thieves.getSize() > 0) {
    final int thief = thieves.pop();
    uncountedAsyncAt(places().get(thief), () -> {
      synchronized (globalRef.get().waiting) {
        globalRef.get().deal(globalRef, loot, victim);
        globalRef.get().waiting.set(false);
        globalRef.get().waiting.notifyAll();
      }
    });
  } else {
    int thief = lifelineThieves.pop();
    asyncAt(places().get(thief), () -> {
      globalRef.get().deal(globalRef, loot, victim);
    });
  }
}
```

Listing 4.7: APGAS_GLB: Method `give()` from the class `Worker`

```java
private void deal(GlobalRef<Worker<Queue, T>> st, TaskBag loot, int source) {
  boolean lifeline~= source >= 0;
  if (lifeline) lifelinesActivated[source] = false;
  boolean oldActive;
  synchronized (waiting) {
    oldActive = this.active.getAndSet(true);
    processLoot(loot, lifeline);
  }

  if (!oldActive) {
    processStack(st);
  }
}
```

Listing 4.8: APGAS_GLB: Method `deal()` from the class `Worker`

## 4.4. Additional new Components

We added two additional new components to GLB. They are described in the following Sections 4.4.1 and 4.4.2.

### 4.4.1. Logger

The original class `Logger` in X10_GLB logs for each place statistics at runtime. Depending on the configured verbose level, the output contains the following data for each worker:

- number of calculated tasks,
- number of given and received tasks,
- number of sent steal requests to random victims,
- number of sent steal requests to lifeline buddies,
- number of received random steal requests,
- number of received lifeline steal requests.

Moreover, the following data about the whole program are printed out:

- wall-clock time for the setup,
- wall-clock time for processing, inclusive stealing and waiting,
- wall-clock time to combine the partial results to one final result,
- number of stolen tasks grouped into random and lifeline,
- number of successful random steal requests.

Additionally, we added a new feature to log the execution time of relevant worker states in detail. In this way, we can determine which state consumes how much time. This information is useful to find possible causes of poor performance.

This feature is disabled by default. In our benchmark implementations, it can be enabled with the program argument `-timestamps` *`number_of_timestamps`*. The number of timestamps has to be over zero and defines the levels of fineness in the output data.

The four relevant states in GLB are *stealing*, *computing*, *distributing* and *dead*. At any given time, a worker is located in one of these states. It was necessary to add some method calls in the classes `GLB` and `Worker`. They are responsible for starting and stopping the execution time for the sections. Each worker logs its own states.

When a state begins, the method `startStoppingTimeWithAutomaticEnd()` has to be called, passing a state. The states are defined as `int` constants in the class `Logger`. First, the method checks its last logged state. If the new state equals to the last state, an incorrect call is assumed and the method returns immediately. Otherwise, the last logged state will be stopped and a new logged state is started.

If a new state is started in an `uncountedAsyncAt`, it has to be stopped manually. An uncounted activity is ignored by a surrounded `finish`. Thereby, it can happen, that the worker activity has already finished, but an uncounted activity is still active. In this case, the state in the `uncountedAsyncAt` can not be finished automatically. Hence, in an `uncountedAsyncAt` the method `startStoppingTime()` has to be called, passing a state. The method `startStoppingTime()` creates a new logged state and returns a specified generated ID of the new logged state. This kind of logged state has no direct following logged state and can only be stopped with the method `endStoppingTime()`, passing the corresponding generated ID. So the last statement in such an `uncountedAsync` has to be a call to stop the logged stated.

After all tasks have been worked off, each worker has a different number of logged states. Therefore, the execution time of the program is divided into the specified count of timestamps, which are set by a program argument, see above. The period of a timestamp is calculated with *execution time/count of timestamps*. Each logged state is organized in its respective timestamp. Afterwards, for each timestamp four values are calculated. Each value represents the percentaged proportion of a state in this timestamp.

Then, the individual worker statistics are reduced to one program statistic. This program statistic is written to the text file *data.csv*, which is saved in the folder *gnuplot*. The folder also contains a script, called *diagram.gp*. This script can be executed by the tool *gnuplot* [10] and generates a histogram of the program life cycle. A generated histogram illustrates how much time the program spends in each section. Thereby, the effectiveness and the weakness of the used algorithm can be analyzed. Section 6.4 provides a generated histogram and an explanation.

### 4.4.2. Debugging Support

During development, programming errors occurred. This is usually expressed by an uncaught exception, a wrong final result at the end of the program, or the program hangs in a deadlock. Some of these bugs are based on serial errors. These kinds of bugs can be found by starting the program with one place and debugging it with a Java debugger. However, a couple of bugs only occur when using multiple places. These bugs are caused by parallel problems, for example race conditions. They arise only sometimes in special situation and not in every execution. To find these bugs, the developed programs were executed in a loop with hundreds of full runs. Sometimes a bug appeared only after hours of execution. This made it difficult to analyze the problem with a debugger.

To simplify troubleshooting, we implemented a simple singleton class `ConsolePrinter`. It has a boolean constant `PRINT` and a method `print(String)`. `PRINT` represents whether the program is in debug mode or not. If it has the value `true`, the method prints out the passed text.

The programmer has to set the constant before compiling the project. Thus, if it is set to `false`, the compiler can optimize the code and the execution time is not affected negatively.

The constructors of the classes `Logger`, `GLB` and `Worker` initialize an object of `ConsolePrinter`. At every critical position, the `print()`-method is called with an appropriate text which contains the executing place. The texts are printed out and can be saved as a file. Thus, a transparent log can be generated. This facilitates the bug finding substantially. `ConsolePrinter` and all associated usages stay in the code to help possible future advancements.

## 4.5. Benchmarks

We ported the benchmarks UTS and BC from X10 to APGAS, keeping them as close to the X10 variants as possible.

Both benchmarks can be parameterized to configure their properties. The program arguments are parsed with the open source library *Apache Commons CLI* [29]. It was used in Version 1.3.1 and as compiled jar-file. Its functionalities are only used in the main classes of the benchmarks.

Sections 4.5.1 and 4.5.2 describe some implementation details of both benchmarks.

### 4.5.1. Unbalanced Tree Search

We implemented this benchmark with an exclusive storage for the tasks, as noted in Section 3.2. X10_GLB has a class `RingBuffer`, which is used as data structure for the local task pool. Java provides the class `ArrayDeque`, which has similar characteristics. However, some specified methods are lacking. We added these methods and used it as data structure for the local task pools. In a steal request, the victim pops tasks from the back, whereas the thief merges them to the front.

`MyArrayDeque` is designed serially because APGAS_GLB allows no concurrent access to it.

The class `UTS` extends the class `MyArrayDeque` and implements a serial variant of the benchmark. It can be started autonomously running on one place. In turn, the class `Queue` extends `UTS` and implements the APGAS_GLB interface `TaskQueue`. Additionally, it contains the inner class `UTSResult`, which extends the APGAS_GLB class `GLBResult` and instantiates its generic type as `Long`. The class `Bag` extends the APGAS_GLB class `TaskBag` and specifies a data structure for saving tasks. Objects of the class `Bag` are transferred between places.

The `main` method for starting the benchmark with the APGAS_GLB framework is implemented in the class `UTSG`. This class initializes an object of the class `GLB` and calls the method `run(Runnable)` on the object because the tasks are generated at runtime. When the run has ended, it prints out the result and the benchmark finishes.

### 4.5.2. Betweenness Centrality

The basic structure of the BC benchmark is similar to the structure of the UTS benchmark. However, the class `BC` stores the tasks in an array instead of using the class `MyArrayDeque` for that. An array can be used because all tasks are known at program start. When starting the benchmark, it will generate a graph from a seed. This is done with the help of generating random numbers. X10_GLB generates the number with the help of the official X10 class `Random`. We ported this class to APGAS to guarantee the same results with the same input values. The benchmark can be started with the class `BCG`. It initializes an object of `GLB` and calls `runParallel()` on it because all tasks are known at program start.

# 5. Intra-Node Synchronization

X10_GLB introduces a significant restriction: only one activity per place can be executed. In APGAS_GLB, multiple activities per place can be executed. However, critical sections are synchronized via a lock, therefore the outcome remains the same as in X10_GLB. For this section, we implemented a data structure for the local task pools, called *split queue* [6] which allows a limited form of concurrent access. The split queue has to be used in a user application, not directly in the framework. Additionally, we modified the APGAS_GLB implementation to support the split queue. This achieves intra-node synchronization and, thus, random steal requests no longer need to be queued. The modified implementation is called *APGAS_Split_GLB*.

APGAS_Split_GLB is an independent framework. GLB users has to decide whether they will employ APGAS_GLB or APGAS_Split_GLB. If they employ APGAS_Split_GLB, they have to utilize the class `SplitQueue` in their own class that implements the task pool. Integrating the class `SplitQueue` directly into APGAS_Split_GLB would have required more radical changes to GLB than our variant did.

In the following, Section 5.1 describes the concept and our implementation of a split queue. Then Section 5.2 explains the differences between APGAS_GLB and APGAS_Split_GLB, including corresponding benchmarks.

## 5.1. Split Queue

First, Section 5.1.1 explains the concept of the split queue. Then Section 5.1.2 gives some details about its implementation.

### 5.1.1. Concept

The concept of the split queue was taken from Dinan et al. [6]. Figure 5.1 illustrates the structure. The split queue is based on the concept of a serial double-ended queue, called *deque*, and enables a limited form of concurrent access. A deque provides the fields `tail` and `head`. The field `tail` indicates the position of the first element, and the end of the deque. The field `head` indicates the first free position, and the front of the deque. The content of the deque is located in between. Additionally, a deque provides the following operations:

- *push*: adds one element to the front of the deque,
- *pop*: removes one element from the front of the deque and returns it,
- *put*: adds one element to the end of the deque,
- *get*: removes one element from the end of the deque and returns it.

Of course, these operations can also be executed with several elements at once. The split queue splits a deque into a *private* portion and a *public* portion. Hence, the field `split` was added. Only the worker activity is allowed to access the private portion. Therefore, the private portion is not locked. The field `split` indicates the first element in the private portion, and `head` indicates the first free position in the split queue. Any activity is allowed to access the public portion, in contrast to the private portion. Hence, accesses to the public portion are locked. However, nobody is allowed to add elements to the public portion. Therefore, the split queue does not provide a method for this. The field `tail` indicates the first element in the public portion, and `split - 1` indicates the last element in the public portion.

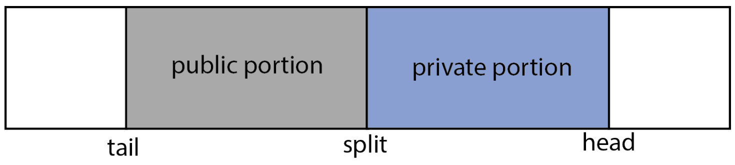

Figure 5.1.: Structure of the split queue

If the private portion is empty after removing an element from it, `split` has to be shifted into the direction of `tail`. This process is called *reacquire*, and only the worker activity is allowed to execute it. *Reacquire* needs access to the public portion and therefore the worker activity needs to take the lock.

Additionally, the worker activity has to check periodically if the public portion still contains enough tasks. If not, `split` has to be shifted into the direction of `head`. This process is called *release* and is lockless because it needs no access to the public portion. Our implemented split queue does not perform this periodic check itself, it has to be performed in the application. APGAS_Split_GLB realizes the periodic check in the method `processStack()`, after processing $n$ tasks. Implementation details are shown in Section 5.2.

### 5.1.2. Implementation Details

The class `SplitQueue` realizes the concept of the split queue. The class is utilized as data structure for a local task pool. The concept of GLB contains no concrete task pool and thereby, a GLB user has to implement it, see paragraph `TaskQueue` in Section 4.1 on page 26. Therefore, we integrated `SplitQueue` into the benchmarks, see Section 5.2.1. Additionally, we created APGAS_Split_GLB which is a modified variant of APGAS_GLB. In APGAS_Split_GLB, random steals can be performed directly and without waiting. This is made possible by the split queue. APGAS_Split_GLB only works correctly if a GLB user utilizes the class `SplitQueue` as type for the task pool. Implementation details of APGAS_Split_GLB are given in Section 5.2.

The following paragraphs describe implementation details of the new class `SplitQueue`.

### Fields

The class `SplitQueue` has a field named `elements` which is an array of a generic type. It represents a circular queue for storing the tasks. Moreover, there are the essential fields `head`, `tail` and `split` of type `int` with the functionalities described in the concept, see above. Furthermore, there is the static field `maxSplitPercent` of type `double` which is initialized with the value 0.5. The value specifies how the tasks should be distributed to the private and public portions, with default value 0.5 it is 50:50. After performing *reacquire* or *release*, the specified ratio is reestablished. It is conceivable that another ratio enables a better performance in the benchmarks. However, this could not be tested in the limited time.

### `pushPrivate()`

The method `pushPrivate()` pushes a passed element into the private portion of `elements` and recalculates the field `head`. Only the worker activity is allowed to push elements to the private portion. If `elements` has reached its capacity after a push, the method `doubleCapacity()` is called.

### `popPrivate()`

The method `popPrivate()` pops one element of the private portion and returns it. Afterwards the field `head` is recalculated. Only the worker activity is allowed to call this method. If the private portion is empty after popping, the method `reacquire()` is called.

### `getPublic()`

The method `getPublic()` pops a passed number of elements of the public portion and returns them in an array. In APGAS_Split_GLB, the returned array is stored in an object of `TaskBag`. Afterwards the field `tail` is recalculated. Any activity is allowed to call this method. Hence, the method is marked with `synchronized`.

### `reacquire()`

The method `reacquire()` shifts the field `split` towards the field `tail`. Only the worker activity can call it. However, this method is synchronized because it needs access to the public portion. After calling the method, the size of the private portion equals the size of the public portion, see above.

### `release()`

The method `release()` shifts the field `split` towards the field `head`. Only the worker activity is allowed to call it. In contrast to the method `reacquire()`, this method does not need access to the public portion. The method `release()` changes the value of the field `split` and the field `split` can be read by other stealing activities at the same moment.

At worst, the stealing activities would read a value that is too small because the field `split` can only be increased. In this case, a stealing activity steals fewer tasks than with the correct value of `split`. However, no failures would occur. Moreover, in Java memory accesses to a variable of type `int` are atomic [18]. Therefore, no mixed values can occur.

Furthermore, contrary to expectations the field `split` does not need the Java keyword `volatile` to eliminate the risk of memory consistency errors. This kind of error only occurs if a variable is cached and not refreshed before reading. However, each access from a foreign place to the field `split` is realized with a new activity. Therefore, the field `split` will not be cached at foreign places and the Java keyword `volatile` is not necessary for the field `split`. Because of these facts, the method `release()` is not synchronized.

After calling the method, the size of the private portion equals the size of the public portion, see above.

#### `doubleCapacity()`

The method `doubleCapacity()` doubles the length of the array `elements`. As a result the field `tail` has the value 0. This method is marked with `synchronized` because both portions needs to be accessed.

#### `privateSize()`

The method `privateSize()` calculates the size of the private portion with `head - split` and returns the result of the calculation. The method `privateSize()` is not synchronized because only the worker activity calls.

#### `publicSize()`

The method `publicSize()` calculates the size of the public portion with `split - tail` and returns the result of the calculation. Although the method `publicSize()` is accessed concurrently, it is not synchronized. At worth, it returns a size which is too small. But this does not lead to errors, see above.

### 5.1.3. Testing

Before we integrated the class `SplitQueue` into the benchmarks, it had to pass some tests. In Java, tests are typically handled with unit testing. One well-known unit testing framework is *JUnit* [16]. It is open source and simple to use, but it does not provide concurrency testing. In a first step, the data structure was serial tested successfully with a unit test utilizing JUnit 4.7. Afterwards, the data structure was tested with *MultithreadedTC 1.01* [20]. This is an open source framework which enables concurrency testing. So, any number of threads can work concurrently on one instance of a data structure. Additionally, a test can be configured easily to run multiple times. Only after the class `SplitQueue` passed these tests was it integrated into the benchmark implementations as type for the task pool.

## 5.2. Changes in the GLB Implementation

This section explains some implementation details of APGAS_Split_GLB to enable intra-node synchronization. Along the way, differences to APGAS_GLB are pointed out.

The interface `TaskQueue` got two additional abstract methods: `release()` and `publicSize()`. They are called in the class `Worker`; details are described below. The essential adjustments were performed in the class `Worker`. Especially the method `steal()` has been adjusted extensively and is shown in Listing 5.1 in a simplified form. Instead of sending requests to random victims as in APGAS_GLB (see lines 2-13 in Listing 4.3 on page 37), an activity on a victim calls the method `split()` directly. Thus, a steal can be performed instantly. The implementation of `split()` had to be adjusted and was marked with the `synchronized` keyword because it calculates a number of tasks and pops them from the public area. The popped tasks are returned and the victim merges them to its local task pool. If all random steals fail, the lifeline steal requests are sent like in APGAS_GLB.

Furthermore, the lifeline task distributions are adjusted in APGAS_Split_GLB. If a lifeline buddy sends a loot to a thief, it starts a new activity on the thief place. In contrast to APGAS_GLB, the received loot cannot be merged directly into the thief's local task pool though. The reason is that the loot is merged into the private portion and only the worker activity is allowed to access the private portion. Thus, we added the new field `lootMerges` of type `ConcurrentLinkedQueue`. It caches all received loots from lifeline buddies. Each worker activity calls the method `processLoot()` periodically in the method `processStack()`, see line 5 in Listing 5.2. The method `processLoot()` merges the cached loots from `lootMerges` into the private portion of the split queue.

The method `processStack()` also includes more adjustments. These are shown in Listing 5.2. The calls of the methods `process()` and `distribute()` no longer need to be synchronized (see line 3 and 10). Thereby, the method `process()` is called again in the loop head. The method `distribute()` accesses the public portion with a call of the method `split()`, see line 3 in Listing 4.6 on page 39, but the method `split()` is marked with `synchronized`, see above. Furthermore, if after processing the public portion is empty, the method `release()` is called in line 8. This realizes the required periodical check of the split queue concept which was introduced in Section 5.1.1. The method `release()` moves tasks from the private portion into the public portion.

In line 16, the method `steal()` is called. In contrast to APGAS_GLB, it changes the value of the field `empty`, see line 11 in Listing 5.1 which is made possible by direct stealing. Thus, the field `active` can be set with the value of the field `empty` instead of the return value of the method `steal()` like in APGAS_GLB, see line 16 in Listing 4.2 on page 36.

In line 16, the method `processLoot()` is called again to merge possible newly received loot into the local task pool. Afterwards, if the field `empty` has the value `false` and the public portion is empty, the method `release()` is called again to shift tasks from the private to the public portion.

```
public boolean steal(GlobalRef<WorkerSplit<Queue, T>> globalRef) {
  for (int i = 0; i < w && empty.get(); ++i) {
    int v = victims[random.nextInt(m)];
    final TaskBag[] taskBag = new TaskBag[1];
    taskBag[0] = at(place(v), () -> {
      final TaskBag split = globalRef.get().queue.split();
      return split;
    });
    if (taskBag[0] != null) {
      this.queue.merge(taskBag[0]);
      empty.set(false);
    }
  }
  //lifeline~requests
  return !empty.get();
}
```

Listing 5.1: APGAS_Split_GLB: Method `steal()` from the class `Worker`

```
public void processStack(GlobalRef<WorkerSplit<Queue, T>> globalRef) {
 do {
   while (queue.process(n)) {
     synchronized (waiting) {
       processLoot();
     }
     if (this.queue.publicSize() == 0) {
       this.queue.release();
     }
     distribute(globalRef);
   }
   empty.set(true);
   reject(globalRef);
   synchronized (waiting) {
     steal(globalRef);
     processLoot();
     if (!empty.get() && this.queue.publicSize() == 0) {
       this.queue.release();
     }
     this.active.set(!empty.get());
   }
 } while (this.active.get());
 reject(globalRef);
}
```

Listing 5.2: APGAS_Split_GLB: Method `processStack()` from the class `Worker`

### 5.2.1. Benchmarks

Both benchmark implementations from Section 4.5 had to be adjusted slightly to utilize the class `SplitQueue` as type for the local task pools: The class `UTS` extends the class `SplitQueue` instead of the class `MyArrayDeque`. Moreover, the class `BC` also extends the class `SplitQueue` instead of using an array of type `int`.

Therefore, some method calls had to be adjusted. The respective class `Queue` still extends `UTS`, respectively `BC`, and implements the adjusted interface `TaskQueue`, see Section 5.2. However, one essential adjustment had to be made in the class `Queue`: the method `split()` is now marked with the `synchronized` keyword because it calls the methods `publicSize()` for calculating a number of tasks and pops this number from the public portion. The method `split()` could be moved directly into the class `SplitQueue`. However, the method `split()` remains in the class `Queue` because it contains the logic concerning how many tasks are supposed to be stolen. A GLB user has to be able to adjust this. Our benchmark implementations steal all available tasks from the public portion. It is conceivable that another logic enables a better performance in the benchmarks. Especially, the logic has to be compatible with the field `maxSplitPercent`, see paragraph *Fields* on page 50. However, this could not be tested in the limited time.

# 6. Experiments

We employed the UTS and BC benchmarks for the experiments with the GLB framework. They have already been discussed in Section 3.2 for the X10 implementations and in Sections 4.5 and 5.2.1 for the APGAS implementations. Both benchmarks were executed with the three GLB variants, introduced in Chapters 4 and 5:

- *X10_GLB*: official X10 implementation.
- *APGAS_GLB*: adopted variant of X10_GLB in APGAS.
- *APGAS_Split_GLB*: APGAS_GLB with intra-place synchronization.

Section 6.1 will provide information about the used cluster and the software versions. Afterwards, Section 6.2 documents our ways of execution. Section 6.3 explains the configurations of the benchmarks. Then the results of the experiments are illustrated and described in Section 6.4. Moreover, a life cycle of a benchmark execution is shown. Finally, the results are discussed with regards to performance and scalability.

## 6.1. Setup

The experiments were conducted on the Lichtenberg high performance computer at TU Darmstadt [24]. This cluster is free to use for scientific work and has currently over 800 nodes. The nodes are connected with each other via infiniband. We used nodes with two 8-core Intel Xeon E5-2760 CPUs and 32 GB main memory. The cluster provides a batch system and is thereby suitable for benchmarking parallel programs.

We deployed X10 and APGAS from the official git repository (status on December 15, 2015) [14]. X10 was used with gcc in Version 5.2.0 and Open MPI in Version 1.8.8. Moreover, we used the *Native X10* compiler, which is based on C++, see Section 2.2. APGAS was compiled with Java in Version 8 with Update 66.

## 6.2. Execution

As mentioned in Section 2.5, X10 and APGAS offer different ways of starting places and for the communication between them. This section describes our ways to commit the benchmarks to the provided batch system. The X10 programs were executed with Open MPI. Therefore, after allocating nodes, they have to be written into a hostfile. After compiling an X10 program with matching parameters for using Open MPI, it can be started with:

```
mpirun -npernode PlacesPerNode -host file program
```

For *PlacesPerNode* the user has to insert the number of places which shall run on one node, for *file* the path to the hostfile, and for *program* the name of the application.

APGAS provides some launcher classes to start programs in different ways. It is recommended to use the Hadoop Yarn launcher in distributed systems. However, the Lichtenberg high performance computer currently does not support Yarn. Therefore, we had to select a different launcher.

The SSH-launcher starts the main program on the current host and launches the remaining places on the other hosts via the network protocol SSH. During first tests with this launcher, problems occurred and places could not connect with each other. After we wrote the problems to the official X10-mailinglist [28], they were fixed by an X10 developer. Since then, the SSH-launcher worked well fine and we could continue working with it.

To start an APGAS program with the SSH-launcher on a cluster with batch system, the allocated nodes have to be written into a text file, one per line. Then an APGAS program can be started with:

```
java -Dapgas.launcher=apgas.impl.SshLauncher
     -Dapgas.places=places -Dapgas.hostfile=file program
```

For *places* the user has to insert the number of places, for *file* the path to the hostfile and for *program* the name of the application.

## 6.3. Configuration

Both benchmarks were executed with two configurations: *small* and *large*. The configurations were set with program arguments, which are displayed in Table 6.1.

| **Benchmark** | **Configuration** | **Parameters** |
|---|---|---|
| UTS | small | $d = 13,\ b = 4$ |
| | large | $d = 17,\ b = 4$ |
| BC | small | $N = 2^{14},\ s = 2$ |
| | large | $N = 2^{16},\ s = 2$ |

Table 6.1.: Benchmark configurations

We modified the GLB starting routine and changed the initial task mapping, see Section 4.1. Thereby, a better load balancing can be achieved, but the dimension of the lifeline hypercube has to be set properly via the program argument `l`. The variable `z` specifies the maximum number of lifeline buddies and is calculated automatically with $\mathtt{l}^{\mathtt{z}}$, whereby the result has to be greater or equal than the number of places. Table 6.2 shows our values for `l`.

| **Number of places** | **Parameter `l`** |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 4 | 2 |
| 8 | 3 |
| 16 | 3 |
| 32 | 3 |
| 64 | 4 |
| 128 | 5 |
| 256 | 4 |

Table 6.2.: Matching parameter `l` for the corresponding number of places

During our experimental period, the Lichtenberg high performance computer had many other jobs queued in its batch system and worked at full capacity. Thus, our jobs had to wait a long time till they were processed. The waiting time increased with the rising number of allocated nodes. To perform all planned experiments in the limited time, we had to compromise. Ideally, we would start with one place on each node. However, to reduce the waiting time, we started with a rising number of place several places per node. Thus, we reduced the number of allocated nodes and hence the waiting time. Table 6.3 presents our configurations. The maximum count of places per node is 16 because each node has 16 processor cores. The compromise makes it possible to increase the number of places by the powers of 2 up to 256 places, which are distributed cyclically to the allocated nodes.

Each experiment was executed five times because the execution times varied slightly. An average value was formed from these five executions and represents the result of one experimental construct.

| Places overall | Places per Node |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 4 | 1 |
| 8 | 1 |
| 16 | 2 |
| 32 | 4 |
| 64 | 8 |
| 128 | 16 |
| 256 | 16 |

Table 6.3.: Number of places overall and their allocation to the nodes

## 6.4. Results

This section presents the experimental results in diagrams. They are discussed in Section 6.5.

Each diagram depicts a configured benchmark (*UTS small*, *UTS large*, *BC small* or *BC large*), and contains three curves. Each curve represents a GLB variant of X10_GLB, APGAS_GLB and APGAS_Split_GLB. The $x$-axis represents the number of places and the $y$-axis represents the execution time in seconds. Note that the $x$-axis is divided logarithmically.

### Unbalanced Tree Search

Figures 6.1 and 6.2 illustrate the results of the UTS benchmark. With the small UTS configuration all GLB variants scale up to 16 places. Up to 32 places the execution times of the APGAS programs improve slightly, but stagnate with 64 places. With more places, the execution time increases in all variants. The X10 variant is faster than the APGAS programs when using up to 64 places. Using

16 places, it is 94% faster. Moreover, the execution times of the APGAS variants differ very little from each other at any number of places.

The large UTS configuration is depicted in Figure 6.2. In contrast to the small UTS configuration, all GLB variants scale up to 256 places. Moreover, X10 is faster than APGAS, and both APGAS variants have similar execution times. Using 256 places, the X10 variant is 25% faster than the APGAS variants. This is significantly less than the small configuration. Compared to X10, longer execution times seem to work in favor of APGAS.

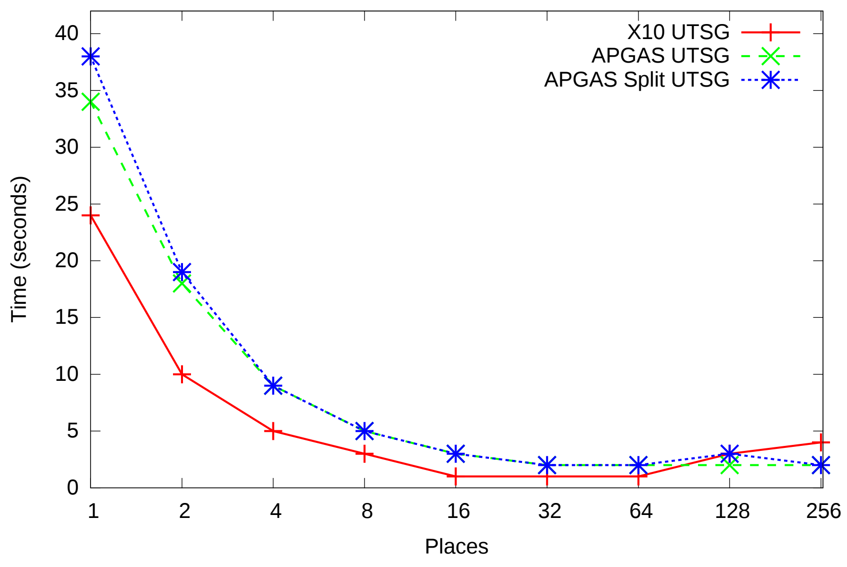

Figure 6.1.: Experimental results for the small UTS configuration

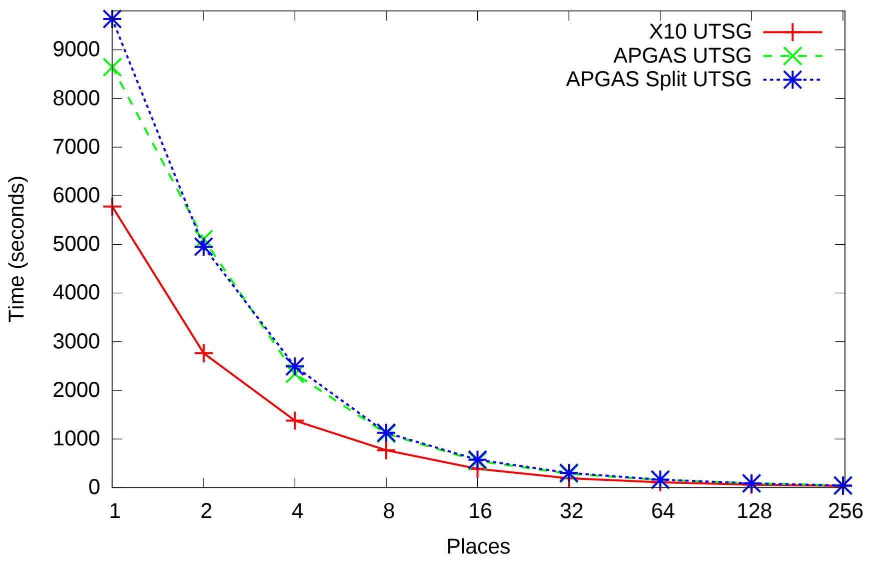

Figure 6.2.: Experimental results for the large UTS configuration

## Betweenness-Centrality

The experimental results of the BC benchmarks are illustrated in Figures 6.3 and 6.4. With the small configuration, the X10 variant has in parts slightly higher execution times than the APGAS variants. The higher execution times have a surprisingly high peak at two places, but is overall small. Apart from that, the small configuration scales well up to 64 places and the large configuration scales well up to 256 places. Both APGAS variants have similar execution times.

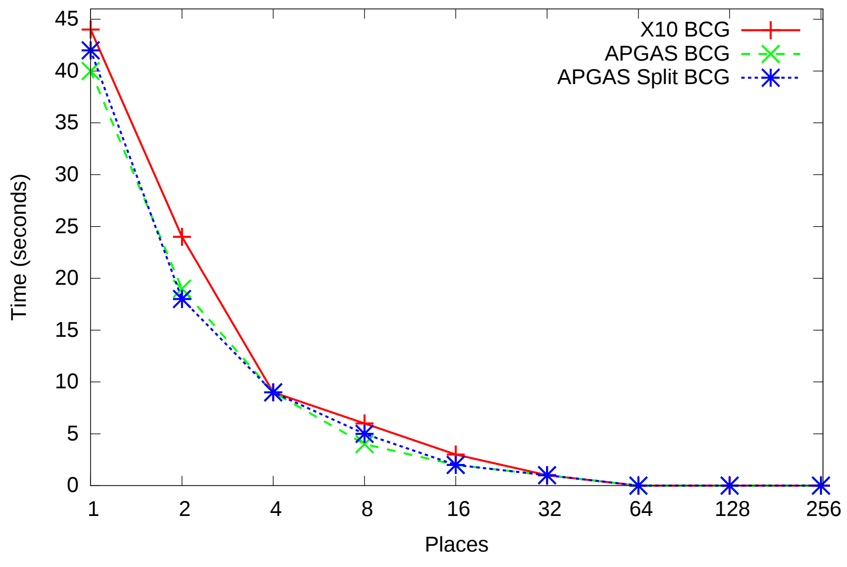

Figure 6.3.: Experimental results for the small BC configuration

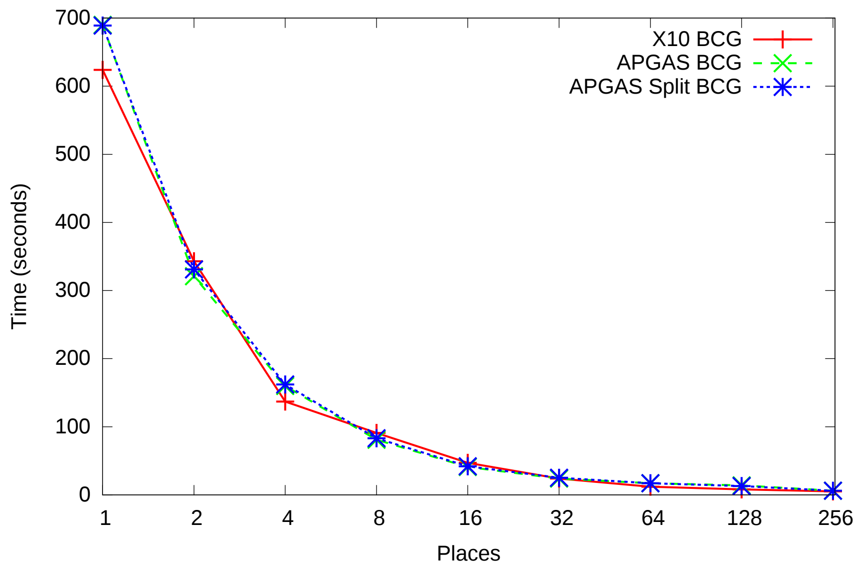

Figure 6.4.: Experimental results for the large BC configuration

## Life Cycle

Figure 6.5 shows a program life cycle in the form of a histogram. Its data originate from the added functionalities in the class `Logger`, see Section 4.4.1. We executed the small configuration of the UTS benchmark with the APGAS_GLB framework, four places and 500 timestamps. The histogram shows how much processor time is used by each of the states *computing*, *stealing*, *distributing* and *dead* at runtime

At the beginning, there is a remarkable percentage of *dead* caused by the initial task distribution. Each worker gets its first tasks successively and starts processing tasks. In the essential operation time, every worker is located mostly in *computing*, and sometimes tasks are shared with *stealing* and *distributing*. Shortly before the program finishes, *stealing* increases because the count of tasks drops. Altogether, the program uses comparatively little time for organization and computes most of the time. Therefore, the framework works effectively.

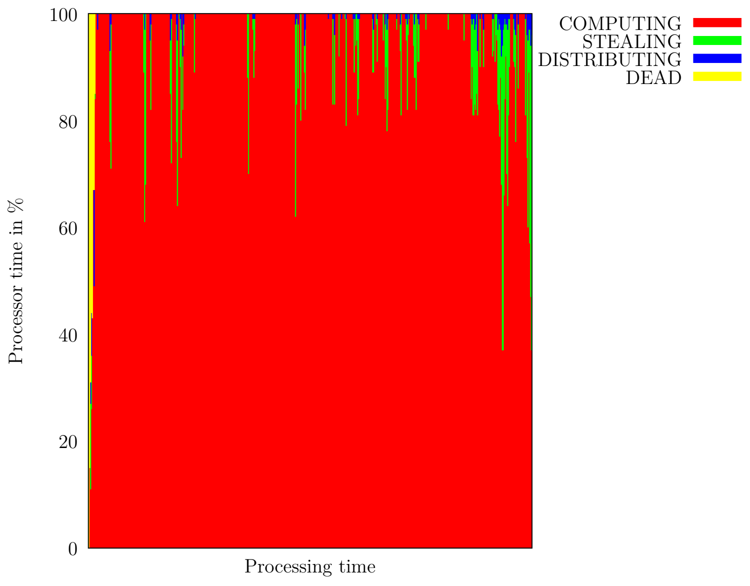

Figure 6.5.: Life cycle of UTS with APGAS_GLB

## 6.5. Performance and Discussion

This section discusses the results shown in Section 6.4. First, we can say that APGAS_GLB and APGAS_Split_GLB have similar execution times. Their performance differs in both directions, but is mostly minimal. Altogether, APGAS_GLB is a little faster, but APGAS_Split_GLB might be more appropriate for further developments.

A comparison between APGAS and X10 is more interesting. The following comparison refers to APGAS_GLB only. According to Tardieu [27] the "performance delta between APGAS and X10 is less than 0.5% at scale". Unfortunately, the used compiler of X10 is not stated. As introduced in Section 2.2, X10 provides two compilers: Native X10, which uses C++, and Managed X10, which uses Java. For compiling the X10 benchmarks, we used Native X10 because the benchmark implementations are not compatible with Managed X10.

Our results somewhat differ from Tardieu's observations. In the UTS benchmark with small configuration, X10 was up to 94% faster than APGAS, when using 16 places. One possible explanation is that the synchronized blocks in APGAS_GLB do not work as effectively as `Runtime.probe()` in X10_GLB. Another reason could be seen in the fact that X10 is using C++, whereas APGAS is using Java. Moreover, in the small UTS configuration, the execution time of all GLB variants increases, when using more than 64 places. This is caused by a lot of communication between the places, which there is little work.

The results of the large UTS configuration also show a large performance difference, when using a small number of places. However, this difference decreases with an increasing number of places. With our maximum count of places, 256, the difference is only 22%, i.e. much smaller than in the small UTS configuration. In the large configuration, more work is caused and thereby the entire communication costs descends.

The BC benchmark with the small configuration has performance differences with variations in both directions. Most of the time, APGAS is faster, with a high peak of 24% when using 16 places. In contrast, X10 is 28% faster when using 128 places. When using 64 or more places, the execution times are under one second, therefore, inaccuracies in measurement are possible and the differences should not be overrated. The large configuration has a varying execution time differences in both directions, as well. When using 64 or more places, APGAS is mostly faster, with a high peak in 128 places and a difference of 69%.

All in all, neither X10 nor APGAS is superior. The performance difference between them may be smaller with more places and larger configurations. However, we could not validate this with the available resources and the limited time.

# 7. Conclusion

The most important result of this thesis is the implementation of two frameworks for global load balancing in APGAS. The first framework reimplements the lifeline-based global load balancing concept of SARASWAT et al. [21] and the corresponding official implementation in X10 [30]. Only one significant adjustment had to be made: The official implementation in X10 allows only one running activity per place and interrupts it occasionally, if necessary. Due to this restriction, no place internal synchronizations are required. In contrast, our implementation allowed multiple activities per place because APGAS provides no similar interrupting functionality. However, with the help of synchronized code blocks the outcome remains the same as in X10_GLB.

The second framework modified the cooperative stealing technique of the first framework by enabling direct random stealing of tasks instead of queuing these requests first. For this purpose, a split queue [6] was implemented. It should be utilized as type for the local task pool and therefore, it has to be integrated in a user application. Thus, multiple activities can access the local task pool concurrently in a limited form.

Two benchmarks were run on the Lichtenberg high performance computer at TU Darmstadt [24]. Results demonstrate that the implemented frameworks have good scalability, and achieve similar performance. A comparison to the original GLB framework in X10 showed some differences in execution times, without a clear winner.

The utilized APGAS framework for Java was released shortly before starting this thesis. It is still sparsely documented and rarely used. Thus, this thesis evaluated as an aside the functionalities and usability of APGAS. In this process some starting problems were discovered, which were fixed after a bug report [28].

Altogether, it can be said that APGAS works as expected and can mostly be used intuitively. Especially with previous experience in Java and X10, the training period is short. Compared to X10, APGAS offers distinct advantages for the programmer, for example, an intuitive auto completion and a debugger. From our experience, it is a promising candidate for a parallel programming system to be used outside of science.

In future research, it would be interesting to allow multiple workers per place. Thus, one place could utilize a multi core system to its capacity. Our class `SplitQueue` could be a base for those further developments.

Moreover, an extension for fault tolerance would be attractive. Currently, if a place crashes, its results and tasks are lost. This problem could be solved with cyclic backups of data. Then, if a place crashes, its data can be restored. An implementation of this approach already exists in X10 [8].

# Bibliography

[1] Apache Software Foundation. *Apache Hadoop NextGen MapReduce (YARN)*. Available online: `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`. 2015.

[2] Apache Software Foundation. *Apache Spark - Lightning-fast cluster computing*. Available online: `http://spark.apache.org`. 2015.

[3] D. A. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse. *HPCS Scalable Synthetic Compact Applications 2: Graph Analysis*. Available online: `http://www.graphanalysis.org/benchmark/HPCS-SSCA2_Graph-Theory_v2.0.pdf`. 2015.

[4] R. D. Blumofe and C. E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing." In: *J. ACM* 46.5 (Sept. 1999), pp. 720–748.

[5] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113.

[6] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. "Scalable Work Stealing." In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, 53:1–53:11.

[7] EsotericSoftware. *kryo - Java serialization and cloning: fast, efficient, automatic.* Available online: `https://github.com/EsotericSoftware/kryo`. 2015.

[8] C. Fohry, M. Bungart, and J. Posner. “Towards an Efficient Fault-tolerance Scheme for GLB.” In: *Proceedings of the ACM SIGPLAN Workshop on X10.* ACM, 2015, pp. 27–32.

[9] L. C. Freeman. “A Set of Measures of Centrality Based on Betweenness.” In: *Sociometry* 40.1 (1977), pp. 35–41.

[10] Geeknet, Inc. *Gnuplot.* Available online: `http://www.gnuplot.info/`. 2015.

[11] Hazelcast, Inc. *The Leading Open Source In-Memory Data Grid: Distributed Computing, Simplified.* Available online: `http://hazelcast.org/`. 2015.

[12] Henri Tremblay. *A library for instantiating Java objects.* Available online: `https://github.com/easymock/objenesis`. 2015.

[13] IBM. *APGAS Release 1.0.0.* Available online: `http://x10-lang.org/articles/276.html`. 2015.

[14] IBM. *Core implementation of X10 programming language including compiler, runtime, class libraries, sample programs and test suite.* Available online: `https://github.com/x10-lang/x10`. 2015.

[15] IBM. *X10: Performance and Productivity at Scale.* Available online: `http://x10-lang.org`. 2015.

[16] Junit. *Unit Testing with JUnit.* Available online: `http://junit.org/`. 2015.

[17] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. “UTS: An Unbalanced Tree Search Benchmark.” In: *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing.* Springer-Verlag, 2007, pp. 235–250.

[18] Oracle Corporation. *Java Language Specification - Programs and Program Order*. Available online: `https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html`. 2016.

[19] M. P. Perez. "Ein Framework für globale Lastbalancierung." Masterarbeit. Universität Kassel.

[20] W. Pugh and N. Ayewah. "Unit Testing Concurrent Software." In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2007, pp. 513–516.

[21] V. A. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. "Lifeline-based Global Load Balancing." In: *SIGPLAN Not.* 46.8 (Feb. 2011), pp. 201–212.

[22] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. *The Asynchronous Partitioned Global Address Space Model*. Tech. rep. IBM, 2010.

[23] P. Suter, O. Tardieu, and J. Milthorpe. "Distributed Programming in Scala with APGAS." In: *Proceedings of the 6th ACM SIGPLAN Symposium on Scala*. ACM, 2015, pp. 13–17.

[24] TU Darmstadt. *Der Lichtenberg-Hochleistungsrechner an der TU Darmstadt*. Available online: `http://www.hhlr.tu-darmstadt.de/hhlr/index.de.jsp`. 2015.

[25] O. Tardieu. *Introduction to X10*. Available online: `http://x10.sourceforge.net/documentation/papers/X10Workshop2015/slides/x10intro.pdf`. 2015.

[26] O. Tardieu. *Presentation: The APGAS Library*. Available online: `http://x10.sourceforge.net/documentation/papers/X10Workshop2015/slides/tardieu.pdf`. 2015.

[27] O. Tardieu. *The APGAS Library: Resilient Parallel and Distributed Programming in Java 8*. Available online: `http://x10.sourceforge.net/documentation/papers/X10Workshop2015/tardieu.pdf`. 2015.

[28] O. Tardieu and J. Posner. *[X10-users] APGAS lib: Adding Places across different mashines*. Available online: `http://sourceforge.net/p/x10/mailman/message/34506003`. 2015.

[29] The Apache Software Foundation. *Apache Commons CLI*. Available online: `https://commons.apache.org/proper/commons-cli`. 2015.

[30] W. Zhang, O. Tardieu, D. Grove, B. Herta, T. Kamada, V. Saraswat, and M. Takeuchi. "GLB: Lifeline-based Global Load Balancing Library in X10." In: *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*. ACM, 2014, pp. 31–40.

# A. Appendix

## Code on CD