IMT School for Advanced Studies, Lucca

Lucca, Italy

Programming Abstractions for Data Sharing in Distributed Spaces

PhD Program in Computer Science and Engineering

XXVIII Cycle

By

Marina Andrić

2017

The dissertation of Marina Andrić is approved.

Program Coordinator: Prof. Rocco De Nicola, IMT Institute for Advanced Studies, Lucca

Supervisor: Prof. Rocco De Nicola, IMT Institute for Advanced Studies, Lucca

Supervisor: Prof. Alberto Lluch Lafuente, Techincal University of Denmark

Tutor: Prof. Rocco De Nicola, IMT Institute for Advanced Studies, Lucca

The dissertation of Marina Andrić has been reviewed by:

Prof. Christian W. Probst, Techincal University of Denmark

Prof. Francesco Tiezzi, University of Camerino

IMT School for Advanced Studies, Lucca

2017

suavis laborum est præteritorum memoria

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

First and foremost I would like to express my deep gratitude to my advisors, Professor Alberto Lluch Lafuente and Professor Rocco De Nicola, for sharing enthusiasm and insight into the subject, as well as help and guidance throughout the entire doctorate.

I am also grateful to the reviewers: Professor Christian W. Probst and Professor Francesco Tiezzi for their valuable comments and suggestions.

I owe a debt of gratitude to the following staff of IMT and DTU: Leonardo, Andrea, Sebastian, Simon and Bernd for their unfailing support and assistance in accessing computer clusters for the experimental evaluations.

I wish to thank to my colleagues and friends for the enjoyable working atmosphere and pleasant moments spent together. It has indeed been a great privilege to spend three years at IMT in a wonderful city such as Lucca.

Last but not least, a special thanks goes to my family and to L. for their encouragement, support and understanding.

Vita

September 2, 1986	Born, Belgrade, Serbia.
2005 - 2011	Degree in Mathematics, Average grade: 9.5 (out of 10), Department of Computer Science, Faculty of Mathematics, University of Belgrade.
2011 - 2013	Designer of software systems, Power Symbol Technology.
2013 - 2017	PhD in Computer Science, IMT School for Advances Studies, Lucca.
2015 - 2016	Visiting Research Student during 3 months, DTU Compute - Section for Formal Methods, Technical University of Denmark.

Publications

1. Y. Abd Alrahman, M. Andric, A. Beggiato, A. Lluch Lafuente. Can We Efficiently Check Concurrent Programs Under Relaxed Memory Models in Maude? In *Rewriting Logic and Its Applications - 10th International Workshop, (WRLA)*, LNCS 21-41, Springer Vol. 8663, 2014.
2. M. Andric, R. De Nicola, A. Lluch Lafuente. Replica-Based High-Performance Tuple Space Computing. In *Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, (COORDINATION)*, LNCS 3-18, Springer Vol. 9037, 2015.
3. M. Andric, R. De Nicola, A. Lluch Lafuente. Replicating Data for Better Performances in X10. In *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, LNCS 236-251, Springer Vol. 9560, 2016.

Presentations

1. "Replica-Based High-Performance Tuple Space Computing", 17th International Conference on Coordination Models and Languages, *Inria Grenoble Rhne-Alpes*, Grenoble, France, June 2015.
2. "Replicating Data for Better Performances in X10", 8th Swedish Workshop on Multi-Core Computing, *Technical University of Denmark*, Copenhagen, Denmark, November 2015.

Abstract

The cost of moving data is becoming a dominant factor for performance and energy efficiency in high-performance computing systems. To minimize data movement, applications have to consider data placement in order to optimize data transfer between processing units. To address this scenario, new compiler techniques, tools, libraries and programming abstractions are necessary for harnessing data locality.

The goal of this thesis is to offer suitable solutions to the challenging problems of data distribution and locality in large-scale high-performance computing. To this end, we have developed new programming primitives for two *partitioned data space* languages, namely, Klaim and X10. Abstractions for partitions and data items are called *tuple spaces* and *tuples* in Klaim, and *places* and *objects* in X10. As a result, we designed two languages, RepliKlaim and SharedX10 which enrich Klaim and X10 with new primitives for *data sharing*.

Our approach aims at allowing programmers to specify and coordinate shared data and, specifically, to replicate shared data items while taking into account desired consistency properties. Programmers can exploit such flexible mechanisms to adapt data distribution and locality to the desired levels, e.g., to improve performance in terms of concurrency and data access. We investigate issues related to replica consistency and provide analysis of performance and programmability, including several applications from large scale graph analytics.

Chapter 1

Introduction

1.1 Context and Overview

Data locality and consistency are identified as critical issues for guaranteeing scalability, availability and good performances primarily in large-scale distributed systems such as cloud systems (SKQ13; CGB+06), and have recently gained the interest of the High-Performance Computing (HPC) community (TKD+14; UNZ+16).

Data locality in distributed systems is typically improved via replication. In early systems, the implied consistency level of replicas was *strong consistency*, while with the advent of highly scalable systems, the notion of consistency has been weakened. In fact, as captured by the CAP theorem (GL02), strong consistency of replicas, partition tolerance and availability cannot be achieved at the same time. This has motivated *weak consistency* models of replicated data, e.g., *eventual consistency* (see *optimistic data replication* (SS05; BEH14)), in which local replicas can be modified at any time, without immediate synchronization with other replicas, and it is only guaranteed that all replicas will *eventually* converge to the same state. Weak consistency levels reduce the cost of synchronization and machine latency, however, they are not always sufficient to application developers. For instance, problems considered in the cloud sys-

tems community, such as reserving a seat on an airplane or withdrawing money from a bank account, require stronger guarantees. To address this challenge, the recent work in (BFLW12), inspired by (SPBZ11), proposes linguistic support for controlling data consistency in cloud applications and thus brings replica-awareness to the programming language level.

The concept of data locality is currently gaining a good deal of attention also in the HPC domain. Indeed, with the increasing rate of parallelism, the cost of moving data has become a dominant factor for performance, in term of execution time and energy efficiency of HPC systems. Moreover, the long-held programmability assumption such as cache-coherence is no longer attainable, hence programmers, to achieve the performance goals, are expected to manually orchestrate communications between computational entities. As a result of these developments, harnessing data locality has become a concern of fundamental importance for compiler and hardware experts, and also for programming language designers who need to come up with new models for the coming era of HPC systems.

In our view, programmers should be equipped with programming abstractions to carefully specify locality of shared data, in order to be able to specify the sharing pattern, that could, e.g., take advantage of replication or rely on centralized data. In this work, we consider two levels of replica consistency, namely, *strong* and *weak* consistency, corresponding to the notions of *linearizability* and *sequential consistency*. Our approach considers replica-awareness at the programming language level. It has to be said that this is not a full novelty; for example CPU caching of shared data in multithreaded programs has led to the possibility of *data races* and, consequently, to high-level programming abstractions that can guarantee safe access to shared data (e.g., the *volatile* and *synchronized* constructs in Java).

In this thesis we present two languages, RepliKlaim and SharedX10, equipped with programming abstractions for specifying locality and consistency of shared data. They are based on Klaim (DNFP98) and X10 (CGS+05; X1017), respectively; two languages with a *partitioned data space*. Abstractions for partitions and data items are respectively called

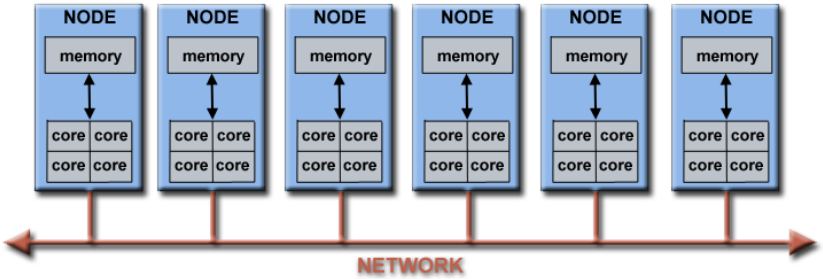


Figure 1: A schematic view of a parallel computing cluster

tuple spaces and *tuples* in Klaim, and *places* and *objects* in X10. Our results include a number of experiments aimed at providing some performance related criteria for deciding whether to employ a specific sharing strategy. These performance measures are accompanied by two case studies from large-scale graph analytics to demonstrate programmability and efficiency of our approach.

1.2 Background and Motivation

Parallel and distributed computing systems are used to solve more and more complex computational problems in HPC. Now, when more computing power is needed, one does not buy a faster uniprocessor but another processor or another thousand processors, and connects them with a high-speed communication network into the so-called High-Performance Computing Cluster (HPCC). This gives one whatever desired number of computer cycles but poses the problem of how to use those computer cycles effectively by dividing the workload into chunks that can be executed simultaneously.

The emerged HPCC structure in the present time is a *Non-Uniform Cluster Computing* (NUCC) system with nodes that consist of physically linked symmetric multiprocessor (SMP) cores with non-uniform memory hierarchies, that are interconnected by horizontally scalable cluster configurations, as shown in Figure 1. According to the current forecasts,

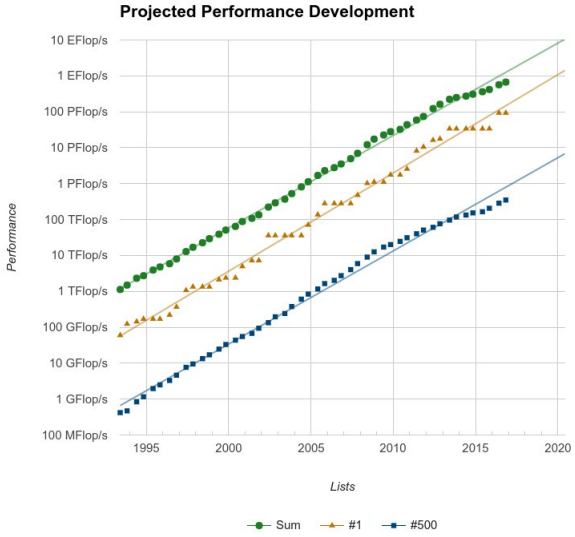


Figure 2: Projected performance development

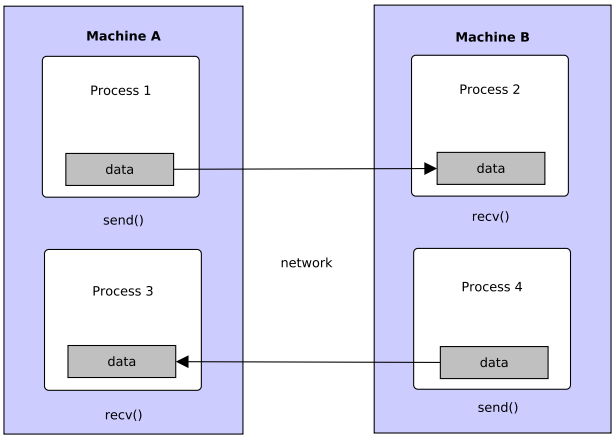
the number of cores on cutting-edge HPCC systems will soon be of the order of thousands, meaning that the order of parallelism continues to increase. Figure 2 shows the projected performance development for the top 500 supercomputers in the world growing linearly at least until 2020. The green dots represent the total performance, the orange triangles represent the performance of the current No.1 supercomputer and blue squares that of the machine on position 500 (as of November 2015). According to the forecast, by 2020 the fastest supercomputer will execute at speed of 1 EFlop/s, while the slowest will execute at speed of 1 PFlop/s, which translate respectively to 10^{18} and 10^{15} calculations per second.

These developments have lead to an increased cost of data movements in terms of energy and computation (KS13). Several studies suggested that asynchronous computation, with reduced communica-

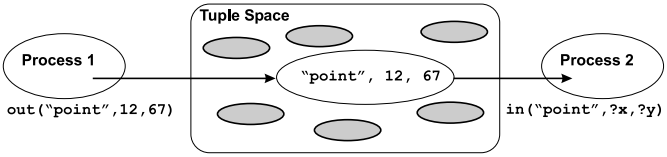
tion, is necessary to reduce those costs for large exascale cluster systems (SDM11; WPC16). Consequently, programming models for these systems are required to be data-centric, i.e., they need to provide programming abstractions that describe how data are laid down and permit to perform asynchronous computations that are aware of data location.

From the programmer's point of view, the key question is what programming model should be used to orchestrate concurrency and data sharing. The most straightforward model is the traditional Shared Memory (SM) model, e.g., as offered by POSIX Threads (PThreads) over cache-coherent shared memory hardware. However, modern HPCC systems use distributed memory for scalability, hence sacrificing the programmability advantages of SM models. In contrast to SM, Message Passing (MP) libraries allow efficient implementations of parallel programs on distributed memory systems. Message Passing Interface (MPI) (OM15) is *de facto* standard for programming cluster systems, however the design and development of MPI programs is intrinsically complex. The main drawback being that it requires the explicit management of the interaction between multiple processes and the coordination of data exchange; large data-structures that are conceptually unitary must be thought of as fragmented across different nodes.

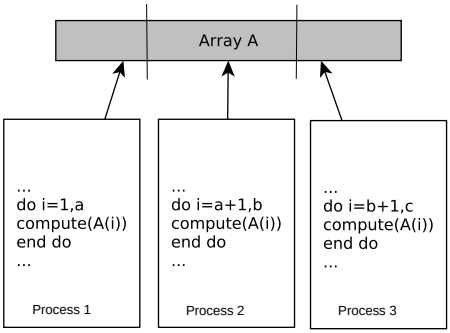
The most notable proposals of programming models which aimed to resolve the tension between programmer's desire for shared memory model and architect's need to sacrifice cache-coherency for scalability include the Partitioned Global Address Space (PGAS) model and Distributed Shared Memory (DSM). The PGAS has been proposed to permit programmers to think of a single computation running across multiple processors, sharing a global address space and relying on zone-based memory management, and it has been implemented in several languages (EGS06; YSP+98; NR98; CCZ07). A notable form of DSM is the tuple space (TS) coordination model, which gained popularity due to the simplicity of its programming primitives. TS has been accompanied by many implementations (Bet03; BDP02; FAH99; Erl16; PyL16; jRe17) for distributed/parallel computing, including TUPLEWARE (Atk08) for programming HPCC systems.



(a) MPI



(b) TS



(c) PGAS

Figure 3: Programmer's view of computation and memory. Arrows represent memory accesses.

Figure 3 shows programmer's view of computation and memory in MPI, TS and PGAS. In MPI, computation is organized around a collection of processes that communicate by sending and receiving messages. Data transfers require synchronous communication, i.e., a send operation must have a matching receive operation, and each process has to have a separate memory that is inaccessible to other processes (Figure 1.3(a)). In TS model processes are meant to interact by exchanging messages (tuples) through a data repository called *tuple space* (Figure 1.3(b)). The PGAS model rests on the idea of a *partitioned* global address space; data structures can be allocated either privately (in local partitions) or globally (shared with other processes). Global data sets are typically organized into a common structure, such as a globally distributed array, which allows processes to work collectively on the same data structure, while each process works on a different portion (Figure 1.3(c)). Differently from MPI, the communication model underlying TS and PGAS is asynchronous.

To study abstractions for data sharing we wanted a minimal model that is close enough to the SM model to ensure programmability and with features to allow good performances in a distributed memory environment. This criterion have led us to select two languages, namely X10 and Klaim. X10 has been proposed as a parallel object-oriented language and one of the first members of the second generation of PGAS languages, extending the PGAS model with notions for asynchrony and locality. Klaim is based on the idea of multiple tuple spaces (Gel89) with explicit information about the location of the nodes where each tuple space is allocated.

Both X10 and Klaim make information about data locality explicitly visible to the programmer who should control which data and processes are co-located. However, adjusting the data locality to the need of application may require significant efforts from the programmer.

X10 promotes locality awareness in the form of places which group *activities* (i.e., processes) and data local to the activities. Accesses to remote data, i.e., located at places different from the one of the executing activity, must comply with the following points: (1) an activity must

spawn a remote activity or *shift* to the place of the remote data, (2) remote data must be modified via specific cross-place references i.e., a reference to a data item at one place that can be shared with other places, and (3) the previous requirements apply to *mutable* but not necessarily to *immutable* data, which are additionally replicated by the runtime at each place that accesses the remote data.

Klaim's equivalent for X10 place is a node on which a tuple space is allocated; the data management is simpler compared to that of X10, primarily due to the fact that, unlike the latter, Klaim does not consider the cost of data accesses at the linguistic level, i.e., the linguistic mechanisms for local and remote data accesses are the same.

1.3 Contributions and Organization

In our view, programmers should be equipped with suitable primitives to deal with locality of shared data in a natural and flexible way. Moreover, efficient data management can be particularly challenging in processing applications with large volumes of data (HLH+11; LK14). To this end, in this thesis we propose two languages with programming abstractions for data sharing: the RepliKlaim language based on Klaim, and the SharedX10 language based on X10.

The thesis is organized into two parts. **Part I** introduces the main notions and tools we deal with in the thesis and it is structured into two chapters. **Part II** discusses the contributions in three chapters. In particular, the questions we address are following:

- *What is a suitable programming abstraction to share data?* We answer this question in Chapters 4 and 5 by providing extensions of Klaim and X10. In RepliKlaim, data sharing is based on replication, while SharedX10 features additional sharing strategy based on centralized data locality. Furthermore, the granularity of consistency is at the level of an operation in RepliKlaim, while in SharedX10 it is at the level of a data item.

- *What are the suitable consistency levels for replicated data?* Our ap-

proach aims at allowing programmers to specify and coordinate replication of shared data items by taking into account one of the two consistency properties, namely *linearizability* and *sequential consistency*.

– *What is the impact of proposed solutions for data sharing?* We answer this question in Chapter 6 by conducting a performance and programmability analysis on two case studies from large-scale graph analytics. We measure performance in term of execution time, an programmability in terms of lines of code.

Last but not least, the conclusions, introspections and perspectives of our work are presented in the final chapter of the thesis.

Part I

Preliminaries

Chapter 2

Klaim, X10 and XTEXT

In this chapter we introduce the programming languages Klaim and X10 in Sections 2.1 and 2.2, respectively. In particular, we detail on their programming models and main concepts that we refer to in **Part II** to present contributions of the thesis, namely, RepliKlaim and SharedX10. The sections introduce several code specifications which serve as a introduction to the real-world applications presented in Chapter 6. Final Section 2.3 gives an overview of Domain Specific Languages (DSLs) and the XTEXT framework for DSL development.

2.1 The Klaim Programming Language

Klaim (DNFP98) was proposed as a coordination language for code mobility, i.e., for specifying migratory applications in network programming. Essentially, Klaim consists of Linda (GC92) features for process communication and a set of operators for process building inspired by CCS (Mil89). As a coordination language, Klaim focuses on coordination and communication requirements of an application, while the computation should be expressed in an additional language, called the *host language* (e.g. C, Java). For simplicity, in this section we present several code snippets using Java as a host language, hence we refer to them as Java-Klaim specifications.

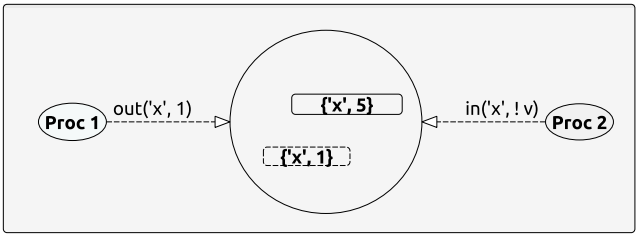


Figure 4: A simple process communication expressed in the Linda model

The Linda model is based on a shared memory store called *tuple space* in which data items are fields called *tuples*. There are four operations provided to access the tuple space:

- `out` - Inserts a tuple into the tuple space,
- `in` - Removes a tuple from the tuple space, and returns it to a process, if a matching tuple cannot be found the process is blocked,
- `rd` - Retrieves a copy of a matching tuple from a tuple space to a process, blocks if the tuple cannot be found, and
- `eval` - Inserts a tuple into the tuple space, unlike in the case of `out`, a new concurrent process is created for evaluating the tuple.

The input operations (`rd` and `in`) specify *template* to retrieve a tuple from the tuple space using the *pattern-matching*. Some fields in the template have their values defined, often referred to as *actual fields*, which are used to find a tuple with matching values for those fields. Remaining fields in the template, called *formal fields*, are variables which are bound to the values in the retrieved tuple by the input operation. In that way the information is transferred between processes.

A simple process communication can be expressed using `out` and `in` as shown in the Figure 4. In this case `('x', 1)` is the tuple being deposited in the tuple space by Process 1. The template, (`'x'`, !v) consists of one defined field (i.e., 'x') which will be used to find a matching tuple,

N	$::=$	$\mathbf{0} \mid l :: [K, P] \mid N \parallel N$	(networks)
K	$::=$	$\emptyset \mid et \mid K, K$	(repositories)
P	$::=$	$\texttt{nil}$	(null process)
		$\mid A.P$	(action prefixing)
		$\mid P + P$	(choice)
		$\mid P \mid P$	(parallel composition)
A	$::=$	$\texttt{out}(t)@\ell \mid \texttt{in}(T)@\ell$	
		$\mid \texttt{read}(T)@\ell \mid \texttt{eval}(P)@\ell$	(actions)
t	$::=$	$e \mid P \mid \ell \mid t, t'$	(tuples)
T	$::=$	$e \mid P \mid \ell \mid !x \mid T, T'$	(templates)

Figure 5: Syntax of Klaim

and a variable v denoted by a leading `!`. As a result of matching, the variable v will be nondeterministically bound to either 1 or 5 since tuple `('x', 5)` in the tuple space is also matching the template. Other forms of communication, as well as synchronization, can easily be expressed using the four operations of the Linda model, as we show in examples in this section.

Klaim extends Linda with multiple tuple spaces, which are allocated to *nodes* or *components* of a network, whose configuration may even change dynamically due to new nodes being added or existing ones removed. Definition 1 introduces a formal specification of Klaim, which differs from the original specification given in (DNFP98) in several aspects. In fact, we decided to choose a subset of Klaimwhich was minimal, yet sufficient to study the main programming abstractions we had in mind. Recursion, for example, is an orthogonal feature that would not have helped in our study.

Definition 1 (Klaim syntax) *The syntax of Klaim is defined by the grammar of Figure 5, where $\mathcal{L}$ is a set of locations (ranged over by $\ell, \ell', \ldots$), $\mathcal{U}$ is a set of basic values (ranged over by $u, v, \ldots$), $\mathcal{V}$ is a set of value variables (ranged over by $x, y, \ldots$), $!\mathcal{V}$ denotes the set binders over variables in $\mathcal{V}$ (i.e. $!x, !y, \ldots$), $\mathcal{T} \subseteq (\mathcal{U} \cup \mathcal{V})^*$ is a set of tuples (ranged over by $t, t', \ldots$), $\mathcal{ET} \subseteq \mathcal{U}^*$ is a set*

of evaluated tuples (ranged over by $et, et', \ldots$), $\mathcal{TT} \subseteq (\mathcal{U} \cup \mathcal{V} \cup !\mathcal{V})^*$ is a set of templates (ranged over by $T, T', \ldots$), and $\mathcal{EXP}$ is a set of value expressions (ranged over by e) built from values and value variables by using a set of operators.

A Klaim specification is a *network* N, i.e. a possibly empty set of components, such that a component $\ell :: [K, P]$ has a locality name ℓ, a data repository K, and parallel processes P. In the original Klaim presentation, each node is characterized by both a physical and logical locality related by the *allocation environment* which provides (partial) mapping from logical to physical localities. Such approach enables controlling visibility, i.e., by allowing processes from one node to access only nodes included in the image of its allocation environment. We simply assume that each node has a single and unique locality known at each other node of a network. Furthermore, we assume that the configuration of a network is static, hence we omit the `newloc` construct for dynamic allocation of new nodes. In fact, we present only the minimal set of Klaim features that would allow us to study abstractions for data sharing.

Klaim tuples are sequences of actual fields (i.e., expressions, processes, localities) while templates are sequences of both actual and formal fields (these are denoted by '$!v$' where v is a generic variable). Data repository is a collection of evaluated tuples according to the tuple evaluation function in Table 1. In particular, the evaluation function, $\mathcal{T}[\![]\!]$, uses evaluation mechanism, $\mathcal{E}[\![e]\!]$, which in turns applies on expressions $e \in \mathcal{EXP}$.

$$\begin{array}{lcl@{\qquad}lcl} \mathcal{T}[\![e]\!] & = & \mathcal{E}[\![e]\!] & \mathcal{T}[\![t, t']\!] & = & \mathcal{T}[\![t]\!], \mathcal{T}[\![t']\!] \\ \mathcal{T}[\![P]\!] & = & P & \mathcal{T}[\![!x]\!] & = & !x \\ \mathcal{T}[\![l]\!] & = & l & & & \end{array}$$

Table 1: Tuple evaluation function

Processes are created from the `nil` process, using the constructs for *action prefixing* ($A.P$), *non-deterministic choice* ($P + P$) and *parallel execution*

($P \mid P$). The actions of Klaim are based on standard Linda primitives for tuple spaces which are additionally located, i.e., each operation has an additional `@`l part to designate the target tuple space, where `self` is a distinguished locality which points to the local tuple space.

The operational semantics of Klaim presented in Figure 6 combines a structural operational semantics (SOS) style for collecting the process actions (rules ACTP, CHOICE and PAR) and reduction rules for the evolution of nets. The operational semantics of nets exploits an evaluation mechanism for tuples and a pattern-matching to select tuples in a tuple space. We use notation P_σ where $\sigma = match(T, et)$, to indicate the substitution of T for *et* in P. The rules for defining the patter-matching predicate are reported in Table 2.

$$match(v, v) \qquad match(P, P)$$

$$match(l, l) \qquad match(!x, v)$$

$$\frac{match(et_1, et_2)}{match(et_2, et_1)} \qquad \frac{match(et_1, et_2)\ match(et_3, et_4)}{match((et_1, et_3), (et_2, et_4))}$$

Table 2: Pattern-matching predicates

Reduction rule OUT adds a new (evaluated) tuple to the tuple space located at l'. As for the communication operations `in` and `read`, we remark that `in` (Rule IN) modifies the tuple space while `read` does not (READ), as matching tuple *et* remains at the component l'. Rule EVAL describes a case in which a process is spawned at a component l'.

$$\frac{}{A.P \xrightarrow{A} P} \text{ (ACTP)} \qquad \frac{P \xrightarrow{A} P'}{P+Q \xrightarrow{A} P'} \text{ (CHOICE)} \qquad \frac{P \xrightarrow{A} P'}{P|Q \xrightarrow{A} P'|Q} \text{ (PAR)}$$

$$\frac{P \xrightarrow{\mathsf{out}(t)@l'} P' \qquad et = \mathcal{T}[\![t]\!]}{N\|\ell::[K,P]\|\ell'::[K_{\ell'},P_{\ell'}] \longrightarrow N\|\ell::[K,P']\|\ell'::[(K_{\ell'},et,)P_{\ell'}]} \text{ (OUT)}$$

$$\frac{P \xrightarrow{\mathsf{in}(T)@\ell'} P' \qquad \sigma = match(T,et)}{N\|\ell::[K,P]\|\ell'::[(K_{\ell'},et),P_{\ell'}] \longrightarrow N\|\ell::[K,P'\sigma]\|\ell'::[K_{\ell'},P_{\ell'}]} \text{ (IN)}$$

$$\frac{P \xrightarrow{\mathsf{read}(T)@\ell'} P' \qquad \sigma = match(T,et)}{N\|\ell::[K,P]\|\ell'::[(K_{\ell'},et),P_{\ell'}] \longrightarrow N\|\ell::[K,P'\sigma]\|\ell'::[(K_{\ell'},et),P_{\ell'}]} \text{ (READ)}$$

$$\frac{P \xrightarrow{\mathsf{eval}(Q)@\ell'} P'}{N\|\ell::[K,P]\|\ell'::[K_{\ell'},P_{\ell'}] \longrightarrow N\|\ell::[K,P']\|\ell'::[K_{\ell'},P_{\ell'}|Q]} \text{ (EVAL)}$$

Figure 6: Operational semantics of Klaim

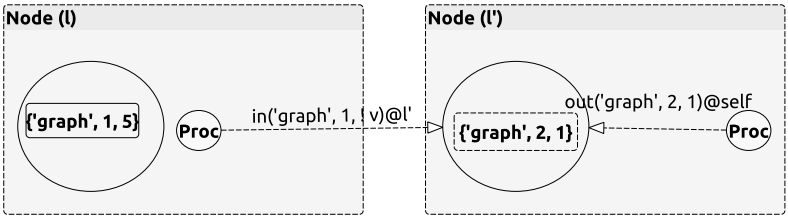


Figure 7: A network of tuple spaces in Klaim

Figure 7 illustrates a network of two nodes, characterized by localities l and l', used by processes to address the target tuple spaces.

Tuples can be used as building blocks for encoding more complex information. For example, a graph can be represented as a series of tuples of the form: `('graphA', 1, firstElement),('graphA', 2, secondElement), ..., ('graphA', m, mthElement)`. Each tuple models a graph node, the first field designates the corresponding graph name, the second stands for the node index in the graph, while the third is in this case generic, e.g. it can store some node property, such as node degree or the PageRank value, depending on the application.

Distribution of tuples across tuple spaces is controlled via the `out` operation, e.g. `out('graphA', i, ithElement)@l` places the argument tuple to the node with locality l. For example, assuming there are `m` graph nodes and `n` network nodes associated with localities $l_1,\ldots,l_n$, one can implement a *block distribution* via e.g., `out('graphA', i, ithElement)@`$l_{h(i)}$, where function `h` computes `i` modulo `n`. As a result, graph nodes will be allocated to tuple spaces in blocks of approximately the same number (`m/n`) of elements.

Listing 2.1 illustrates read/write operations over a graph node. We use terms *write* and *update* interchangeably to refer to the operation which modifies the data. The operation at `line 1` reads the element, while the update operation has to be realized in two steps, first, tuple is withdrawn via `in` (`line 2`), followed by inserting a new tuple via `out` (`line 3`).

Listing 2.1: Simple graph processing

```
1 rd('graphA', i, ?x)@l; /* reads i-th element */
2 in('graphA', i, ?oldValue)@l;
3 out('graphA', i, newValue)@l; /* updates i-th element */
```

Klaim has no specific constructs for process synchronization, however the synchronization can be implemented via set of instructions that each process should perform reaching the point of synchronization. For example, in barrier synchronization, each process within a group must wait at a specific point, called barrier, until all processes in the group reached it; then all can proceed. To implement barrier synchronization in Klaim, one process needs to execute in advance `out('barrier', n)@`l, which places a tuple `('barrier', n)`, usually called a *synchronization token*, to a specific tuple space, and where `n` is the number of processes. Coming to the point of synchronization each process performs the code fragment in Listing 2.2, which comprise the barrier function.

Listing 2.2: Barrier synchronization

```
1 in('barrier', ?val)@l;
2 out('barrier', val-1)@l;
3 rd('barrier', 0);
```

The barrier function consists of conceptually two phases. In the first phase, the process signals that it reached the barrier by updating the synchronization tuple (`lines 1-2`). If it was not the last process to reach the barrier, i.e., `val` is greater that 0, then the process remains blocked in the second phase by the `rd` operation (`line 3`), until the last process reaches the barrier.

Finally, we present two code snippets in Listings 2.3 and 2.4, which illustrate the basic idea of graph processing we present in Chapter 6. We assume that the number of graph nodes is stored in `n` and the number of iterations for the iterative step in `numIter`.

Listing 2.3: Java-Klaim graph node processing

```
1  for (i = 0; i < n; i = i + 1)
2    eval(compute(i));
3  compute(i):
4  {
5    for (j = 0; j < numIter; j++) {
6       /* graph processing code*/
7    }
8  }
```

The main idea is that `n` processes are spawned to evaluate in parallel the `compute` function for different argument, `i`, indicating the index of a graph node. The body of the function simply specifies iterative computation in which node processing is realized. Snippet in Listing 2.4 in addition implements previously introduced barrier synchronization to ensure that each process proceeds at the same speed.

Listing 2.4: Java-Klaim graph node processing with a barrier

```
1  out('barrier', n)@l;
2  for (i = 0; i < n; i = i + 1)
3    eval(compute(i));
4  compute(i):
5  {
6    for (j = 0; j < numIter; j++) {
7      in('barrier', ?val)@l;
8      out('barrier', val--)@l;
9
10       /* graph processing code*/
11
12     in('barrier', ?val)@l;
13     out('barrier', val++)@l;
14     rd('barrier', 0)@l;
15   }
16 }
```

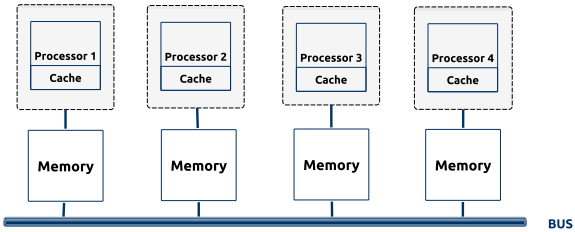


Figure 8: Programmer's view of NUMA architecture

2.2 The X10 Programming Language

X10 (CCS$^+$14) is a programming language developed at IBM, and its design philosophy is based on a belief that prevailing configuration of the future high-end systems will consist of multi-core nodes with possibly non-uniform memory access time (NUMA nodes) (illustrated in Figure 8) interconnected in scalable clusters called *Non-Uniform Cluster Computing* (NUCC) systems. Four main goals set for X10 are to 1) be more productive than current models (X10 stands for "ten times productivity boost"), 2) exploit multiple levels of parallelism and non-uniform data access, 3) be suitable for multiple architectures and 4) support high levels of abstraction. Furthermore, X10 was created to combine ease of programming of object-oriented languages and efficiency of high-performance languages, targeting high-end computers supporting $\approx 10^5$ hardware threads and $\approx 10^{15}$ operations per second (SBP$^+$14). Using the words of the designers, X10 was designed to *"to increase programmer productivity for NUCC without compromising performance"*.

X10 is built upon *asynchronous partitioned global address space* (APGAS) (SAB$^+$10) that enriches PGAS with two main concepts: *places*, which provide an explicit mechanism for data and code locality, and *asyncs* which allows forking a task, possibly at a remote place. Essentially, the PGAS model combines data locality (partitioning) of a distributed memory model and global address space of a shared memory model, thus each processor has private memory for local data and shared

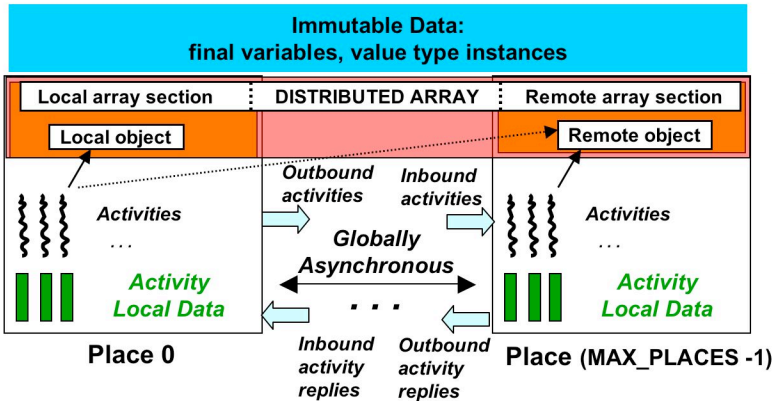


Figure 9: Overview of X10 activities, places and data distribution

memory for globally shared data.

The set of places is fixed before program execution. To set the number of places, one needs to set a value to `X10_NPLACES` program environment variable prior to the program execution. The program starts executing in `Place.places()(0)`, other places can be addressed in a similar fashion by their integer ranks. An activity's local place can be simply addressed by a keyword `here`.

Figure 9 shows a schematic overview of the X10 programming model. Each place hosts some data and runs a number of possibly dynamically created lightweight threads (i.e. *activities*). Data items in X10 can be mutable (`var`) or immutable (`val`), also called *values*. X10 supports user-defined types, standard types (`Boolean`, `Byte`, `Short`, `Char`, `Int`, `Long`, `Float`, `Double`, `Complex` and `String`), functional literals and multi-dimensional arrays (`DistArrays`). For example, `var i:Long = 0` defines a *mutable* variable `i` of type `Long` that is initialized to `0`. Similarly, `val j:Long = 0` defines an *immutable* variable `j` of type `Long` initialized to zero. Unlike mutable data, immutable data cannot be reassigned; for example, an attempt of assigning a new value, e.g., `j = 1`, would trigger a compile-time error. It is important to note that al-

though variable `j` is immutable, object it points to can be itself mutable, such as `val j = new DataType()` where `DataType` is a user-defined class possibly with mutable methods. Moreover, immutable data are copied by the X10 runtime between places via mechanism called *value copying*, while mutable data reside only at a place of allocation and need to be accessed via *global references* and *place-shifting* mechanism by remote activities. Both these concepts, i.e., *value copying* and *place-shifting via global references* are in relation to our idea of data sharing, introduced in SharedX10, and we explain them in details further below. In fact, one of the main ideas behind data sharing is to hide the low-level details of orchestrating data accesses from the programmer.

X10 features function literals that can be assigned to a variable. In general, a function literal $(x_1:T_1 \ldots x_n:T_n)c:T$ `=>` `e` creates a function of type $(x_1:T_1 \ldots x_n:T_n)c:T$, with the body `e` and condition `c` For example, a function literal `val f:(x:Long){x!= 0} => Long = (x:Long){x!= 0} => (1/x)`, stored in value `f`, computes inverse of long integer `x` if condition `x != 0` evaluates to true. We use this convenient feature to specify a function literal `cmp` that compares integer numbers inside a sorting function in Section 5.3.

`DistArray`s (distributed array) are used to spread globally shared data across places. `DistArray` relies on a `Dist` object for defining *distribution*, i.e., mapping of elements to places. `Dist` in turn uses a `Region` object that captures shape and dimensionality of the array. Furthermore, the region and distribution associated with an array are first-class constructs, i.e., they can be used independently of arrays. It is worthwhile to mention that the distribution remains unchanged throughout the program's execution.

As an example, region [0:50, 0:100] specifies a collection of two-dimensional points (i, j) where i ranges from 0 to 50 and j ranges from 0 to 100. Points are used in array to select a specific element in a possibly multi-dimensional grid, while distribution specifies a place for each point in the region. The two distributions that we use in the thesis are:

Unique distribution. Unique distribution maps elements to distinct places provided that there are sufficient places available, otherwise some points will be co-located. For example, distribution `Dist.makeUnique(Region.make(1...k))` maps every point in a 1-dim region of k points to a distinct place of those available in the program.

Block distribution. Block distribution (e.g. `Dist.makeBlock(Region.make(1...k))`) distributes elements in the region in approximately even blocks over all places available in the program.

Remaining distributions are *cyclic*, which cyclically distributes points, *constant*, which maps all points to a single place, and combinations of previous such as blockcyclic, blockblock and so on.

Each distribution may also specify an initializer function which applies to all points in the region. Listing 2.5 shows a distributed array creation with the initializer function stored in a variable.

Listing 2.5: A distributed array creation

```
1 val ident = ([i]:Point(1)) => i;
2 val data:DistArray[Long] = DistArray.make[Long](Dist.
    ↪ makeUnique(Region.make(1,10), ident))
```

The `data` variable stores a reference to an array of ten elements, such that each element is initialized to the value of its index via the initializer function stored in `ident`. We use distribution promoted in this example when we present the SharedX10 encoding, in Section 5.3.

The main X10 construct for concurrency within a place is the `async` construct. The main form of `async` is `async S` that starts a new activity to execute a statement `S` in the same place of the executing process. Remote execution is achieved by means of the `at` construct. For example, the activity that executes `at(P) S` is *place-shifted*, meaning that its execution is suspended in the current place and shifted to place `P` where `S` will be executed. After completion of `S` the control comes back to the current place, with the result of `S`.

One needs to be careful when using the `at(P) S` construct as it involves copying values and it can potentially lead to high costs as the values used in `S` (and depending objects) are copied to place `P`. Such copying is called *value coping* and it can be avoided by using global references (`GlobalRef`s) and place-shifting, as we will show in following examples.

Program in Listing 2.6 defines a variable `x` which stores an object of some generic class `ClassG` (`line 1`), which should be updated with contributions from computations performed at each place. In particular, the enumerator `for` loop in combination with the `at` construct ensures that the `localCompute()` function will be executed at each place and the obtained value `t` will be used to update `x`.

Listing 2.6: Value copying

```
1 val x = new ClassG();
2 for (p in Place.places()) at(p) {
3   var t = localCompute();
4   x.add(t);
5 }
```

One may expect that `x` defined at `line 1` and used for reference at `line 4` are pointers to the same object in memory, however, this is not the case. In particular, due to value copying that `at` entails, there will be several replicas of `x` created (one per each place), and each will be modified independently. As a result, the object stored in `x` at `line 1` will not be modified in the scope of `at(p)`.

This behavior can be altered via global references, i.e., `GlobalRef` objects, as variables reachable through `GlobalRef`s are ommitted in value copying. Therefore, to modify a value in computations that involve several places, one needs to define a global reference (`line 2`) and use the *place-shifting* operation (`line 5`) as shown in the snippet in Listing 2.7.

Listing 2.7: Place-shifting via GlobalRef

```
1 val x = new DataObject(0);
2 val xRef = GlobalRef[DataObject](x);
3 for (p in Place.places()) at(p) {
4   var t = localCompute();
5   at(xRef.home) xRef().add(t);
6 }
```

In general, `val ref = GlobalRef[T](v)` creates a reference to a variable `v` of a generic type `T` and stores it in `ref`. Retrieval of the value is done via `ref()` and it demands place-shifting to the place where `v` is allocated, i.e., `at(ref.home)`.

Parallelism across places is achieved by combining `async` and `at` to spawn a new activity at a remote place, e.g. `at(P) async S` creates a new activity at place `P` to execute statement `S`. X10 provides several primitives for synchronizing concurrently executing activities. In particular X10 features `atomic` blocks, the `finish` statement and the `clock` construct.

There are two types of atomic blocks provided, i.e., the unconditional and conditional atomic block. The unconditional atomic block, `atomic S`, is used to guarantee execution of a statement `S` as if it was a single step with respect to other concurrently executing atomic blocks in the same place. The conditional atomic block provides the same guarantees for the execution of body `S` which is additionally guarded with the condition `c`. Any executing activity of such a statement gets suspended until the guard evaluates to true, moreover the checking of the guard and execution of body `S` is done atomically. However, the guarantees given by the atomic blocks hold only if none of the three restrictions on body `S` is violated: 1) `S` must be sequential, i.e., must not spawn another activity, 2) `S` must execute at a single place and 3) `S` must not use blocking statements.

`finish` construct provides a simple mechanism to synchronize activities. An activity that executes `finish S` will execute `S` and then be suspended until all the activities spawned by `S` terminate. Moreover, `finish` is a distributed termination construct, meaning that it may be

applied to synchronize activities across multiple places.

The underlying idea of the `clock` construct is based on a notion of *phased computation*. Each phase in such computation consists of a set of memory locations which are read and written by concurrently executing activities whose termination signals the end of a phase, and each following phase starts with a new set of activities. Clocks may be explicitly created by the programmer as instances of `x10.lang.Clock` class. However, it is more common to create clocks implicitly by the `clocked finish` construct. An activity gets *registered* to the generated clock when it is created by the `clocked async` construct. Synchronization occurs when an activity executes the `Clock.advanceAll()` primitive; at that point the activity gets suspended until every registered activity have executed `Clock.advanceAll()` and then all activities are released. The clocks can be seen as a generalization of the classical barriers as activities may be registered to several clocks that work independently.

The following code listings are instrumental to introducing the case studies of Chapter 6. The main idea is parallel graph processing, in the fist code variant activities process at their speed, while in the second variant the barrier synchronization is employed to ensure phased execution. A `graph` is represented as a distributed array of nodes (`GraphNode` objects), which are block-distributed across available places.

Listing 2.8: Graph processing

```
1  val graph:DistArray[GraphNode] = DistArray.make[GraphNode](
       ↪ Dist.makeBlock(Region.make(0, size-1)), (Point) =>
       ↪ new GraphNode());
2    /* graph instantiation code, e.g. via processing input
         ↪ file */
3  for (nodeId in graph) async
4    for (var i:Long = 0; i < numIter; i++)
5      at (graph.dist(nodeId)) graph(nodeId).compute();
```

To keep the presentation simple, we commented the code for instantiating such graph (`line 2`) in Listing 2.8, which includes processing an input file that records links between nodes, and storing the information

in corresponding `GraphNode` elements of `graph`. Node computations are conceptually done in parallel by separate activities spawned by the `async` construct at `line 3`. Each activity executes an iterative `for` loop with a body that performs computation on a corresponding node object referenced via `graph(nodeId)`. We leave for now the size of the graph, `size`, the number of iterations, `numIter`, and the `compute()` function unspecified as we detail them in Chapter 6. The code variant which features `clock` constructs to synchronize activities is shown in Listing 2.9. As one may notice, accessing a node requires a place-shifting operation to the place where the node is allocated (`line 3`).

Listing 2.9: Use of clocked finish and clocked async

```
1 clocked finish for (nodeId in graph) clocked async
2   for (var i:Long = 0; i < numIter; i++) {
3     at (graph.dist(nodeId)) graph(nodeId).computation();
4     Clock.advanceAll();
5   }
```

As we illustrated in the examples in this section, the low-level mechanism for data communication, such as activity place-shifting, is transparent to X10 programmers. Moreover, the side-effect of the `at` construct, i.e., the value copying, entails hazard in terms of creating replicas which can be easily neglected by the programmer. The idea of SharedX10 is to introduce the concept of shared data as a first class primitive, so the programmer can specify how the data item is shared between places, with two options: via replication or centralized instance. Thus, the low-level details of data accesses are left to the implementation, i.e., activity place-shifting and replica consistency. We further detail the data sharing primitives in Section 5.1.

We conclude this introduction to X10 with the remark that X10 is still under development at IBM in collaboration with academia. There are two implementation available via source-to-source compilation to another language. The two provided runtime frameworks are named `Native X10` and `Managed X10` that are respectively based on C++ and Java backends. The resulting C++ or Java program is then compiled

by either a specific C++ compiler to produce an executable or compiled to class files and then executed on a Java Virtual Machine (JVM). Detailed performance model of X10 can be found in (GTC+11). The semantics of the language has been formalized in (SJ05) along with a resilient version (CCS+13). A core calculus with X10's main constructs for parallelism is presented in (LP10). Cogumbreiro et al. developed Armus (CHMY15), a verification tool that detects barrier deadlocks for Java and X10 programs. Gligoric et al. attempted to develop a model checking tool (GMM12) for X10 based on the JAVA PATH FINDER tool for model checking Java programs. A line of work focuses on compiling and porting programs to X10, specifically, (KH14) reports on compiling MATLAB to X10 for high performance computing. The work in (GN15) presents a kernel benchmark suite implementing distributed algorithms in X10. A complete list of X10 related publications can be found online at the official website (IBM17).

2.3 Domain Specific Languages

The main goal when developing a programming language is to make programming more efficient. Ideally, one programming language should provide the suitable level of abstraction which allows solutions to be expressed naturally and hides unnecessary details. Furthermore, it should be expressive enough, should provide guarantees on the properties which are important in the domain and it should also have precise semantics to enable formal reasoning about a program.

Domain Specific Languages (DSLs), according to (Bet11), are small languages dedicated to a particular aspect of a software system. Hence, DSLs are designed with an aim to improve programmers' productivity in a particular domain compared to General Purpose Languages (GPLs) such as Java or C.

Implementing a DSL at least consists of a series of procedures which would allow one machine to perform *lexical analysis*, *syntax analysis* or *parsing*, and finally interpret program or generate the code in another language.

In the lexical analysis, an input textual file representing the source code is decomposed into atomic elements or *tokens* which are of four types: *keywords*, *identifiers*, *symbols* and *literals*. For example, a variable declaration `int value = 100;` contains a keyword `int`, an identifier `value`, a literal `100` and symbols ';' and '='. Lexical analysis is followed by the syntax analysis which for an input construct checks whether it respects the syntactic structure specified by the language *grammar*. The main idea of the grammar is to describe the concrete syntax and how it is mapped to an in-memory representation, i.e., the semantic model. This model is produced during parsing and it is called the Abstract Syntax Tree (AST). Upon successful completion of the previous phases, such program is ready to be interpreted or alternatively it can be used as in input to a *code generator* to generate code in a different language.

2.3.1 The XTEXT Framework

XTEXT (Xte16) is an open source framework for developing DSLs. Implementing a DSL starts with specifying the language grammar and can include additional specifications for program verification, code generator or interpreter, type checking and scoping. XTEXT promotes a Java-like programming language, Xtend, to write parts of a DSL implementation. Additionally, XTEXT provides integration of a DSL in the Eclipse Integrated Development Environment (IDE). The main advantages that come along with an IDE support are *syntax highlighting*, that improves code visibility through coloring and formatting keywords, *background parsing*, which continuously checks the program syntax and marks errors, *content assistant*, which provides auto completing mechanism, and *hyperlinking*, which permits navigation between references in a program.

The Grammar Language

The grammar language of XTEXT is itself a DSL designed for describing other DSLs. Essentially, grammar specification of a languages establishes a connection between the language syntax and its semantic model.

The body of a grammar file is a list of grammar *rules*. For example, rule `ID`:

```
terminal ID:
('^')?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_
    ↪ '|'0'..'9')*;
```

specifies that a token `ID` can start with optional " ^ " character, followed by a letter or underscore, followed by an arbitrary number of letters, underscores and numbers. Symbols *, + and ? indicate the cardinality of expressions in the brackets; operator * means zero or greater, + indicates one or greater, while ? indicates cardinality zero or one. If no operator is used the assumed cardinality is exactly one. Rule `ID` is a *terminal* rule in that it contains only elementary symbols, such as letters, numbers and other characters.

Apart from terminal rules, two other types of rules are *data type* and *production* rules. Data type rules, like terminal rules, do not contain *feature assignments*, but may use other rules. On the other hand, production rules contain one or more feature assignments and yield an instance in the AST of the parsed program.

Listing 2.10 shows a grammar snippet which defines two rules.

Listing 2.10: Sample grammar

```
grammar org.xtext.example.Sample
  with org.eclipse.xtext.common.Terminals
Program:
('package' name = QualifiedName ';')?
classes += Class*
;
QualifiedName: ID ('.' ID)* ;
```

`QualifiedName` rule exemplifies a data type rule, while `Program` is an example of a production rule as it contains a feature assignment, i.e., `name = QualifiedName`. Moreover, the first line declares the name of the language and of the grammar, while the second line states the use of the built-in grammar `Terminals` which defines basic rules, such as the

`ID` rule, which can be used in the grammar, as in the definition of the `QualifiedName` rule.

Each rule has a *return type*. If it is not explicitly stated, it is implied that the type's name equals the rule's name. As `Program` is the starting rule, it defines the type of the root element of the syntax tree. Types of features `name` and `classes` are inferred from the right hand sides of assignments and correspond respectively to `QualifiedName` and `Class`. Moreover, operator += indicates a collection of zero or more objects of type `Class` to be stored in feature `classes`.

XTEXT allows declaration of cross-links in the grammar. Rule:

```
1 Variable:
2   'var' name = ID ':' type = [Type];
```

contains a cross-reference pointing to a `Type`, meaning that for a feature `type` only instances of `Type` are allowed. Cross-reference resolution involves *scoping*. Typically, a DSL designer implement *scopes* that define target candidates for the given cross-reference. Such references in the source code are validated during parsing. An additional feature to `name` can be used to record a boolean value which captures is variable type is specified or not, as in the following:

```
1     Variable:
2     'var' name = ID (istyped ?=':' type = [Type])?;
```

Once a language grammar is defined, a DSL designer may specify an interpreter that works on the AST or a code generator to translate the one program to another or generate a configuration file.

Chapter 3

The Memory Consistency Guarantees

Several concepts from the domain of memory models are essential for understanding our work on replica-aware programming in RepliKlaim and SharedX10. This chapter starts by providing a short introduction to the theory of the memory consistency models in Section 3.1. Section 3.2 presents a brief overview of the consistency guarantees in high-level programming languages, and provides an example that promotes the programming styles introduced in RepliKlaim and SharedX10. Section 3.3 outlines several ubiquitous consistency models for replicated data in *replicated systems* (e.g., distributed systems with replicas), and presents formalizations of the two consistency models for replicated data supported in RepliKlaim and SharedX10.

3.1 Overview

To reason on the behavior of a program one has to rely on the *memory (consistency) model* for the guarantees on results of their programs. Moreover, the memory model for a multithreaded program specifies how memory actions (i.e., reads and writes) in a program will appear to execute to the programmer (MPA05a). In particular, the memory model

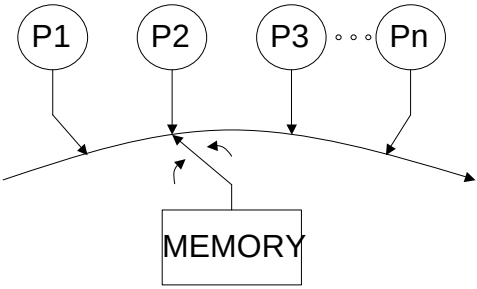


Figure 10: Programmers' view of memory and computation in a sequentially consistent system

specifies the values that a shared variable read in a multithreaded program is allowed to return.

The simplest model for reasoning is *sequential consistency* (SC), defined in (Lam97), which states that the result of any execution is the same as if operations of all processors were executed in a sequential order, and operations of each individual processor appear in this sequence in the order specified by its program. Such model imposes two requirements: (1) *write atomicity*, that is, memory operations must execute atomically with respect to each other and (2) *total program order*, which means that program order is maintained between operations from individual processors.

Sequential consistency is arguably the most intuitive consistency model as it can be seen as an extension of the uniprocessor model to multiprocessor. Figure 10 shows the programmers' view of a sequentially consistency system. One can assume that each of the concurrently running threads accesses memory one at a time, as if there would be a dynamic switch that established an exclusive and temporary connection between the shared memory and a thread.

SC is easy for programmers to understand and adopt, but hampers the system performance (e.g., see discussion in (Sut05)). Indeed, the increasing demand for performance has lead hardware designers and compiler constructors to develop sophisticated optimization techniques that

thread 1	thread 2
`count.inc(1)`	`count.getValue()`

(a)

thread 1	thread 2
`count.inc(1)`	`count.inc(1)`

(b)

Figure 11: Two programs with a data race

give up sequential consistency to accelerate memory operations. This has lead to a lattice of *relaxed memory models*, which can be categorized based on the relaxation of the requirements 1 and 2 for sequential consistency.

3.2 The Memory Consistency Guarantees in High-Level Programming Languages

The memory model of a high-level programming language should make a balance between being simple enough for programmers to use and flexible enough to be implemented efficiently. Despite the fact that sequential consistency is appealing to programmers of high-level languages, it is deemed to significantly restricts the use of many compiler and hardware transformations. To this end, *data-race-free* (DRF) models (Adv93) have been proposed to achieve both programming simplicity and implementation flexibility. The DRF approach states: sequential consistency is guaranteed only to programs that don't contain *data races*.

The memory model specification of a programming language is expected to clarify the program behavior when conflicting operations (at least one is write), in different threads, are accessing the same memory location with no specified order of operations. This scenario is called data race, and programming languages may take different approaches to giving semantics to programs with data races.

Figure 11 shows two scenarios in Java in which a data race occurs in a two-threaded program. The idea is that two threads, **thread 1** and **thread 2**, are concurrently operating on a shared data object `count` (see

Listing 3.1 for class definition). Figure 11(a) shows a scenario in which conflicting operations are a write operation `inc(1)` (increments object's attribute `counter` by 1) and a read operation `getValue()` (returns the `counter` value), while in Figure 11(b), both conflicting operations are writes. Due to the presence of data race in the first scenario, the values observed by **thread 1** and **thread 2** for `counter` may differ, while in the second scenario the result stored in `counter` may be incorrect. We detail how to avoid data races in this example in the following section, using the synchronization mechanisms.

Examples of high-level programming languages that adopt the data-race-free approach are Java and C++ (since C++11). However, the two memory model specifications differ in the approach towards giving semantics to programs containing data races. In particular, C++ memory model leaves the semantics for such programs unspecified (C++14), while Java Memory Model (JMM) also provides semantics for programs with data races (MPA05b).

To avoid data races, C++ programmers are provided with *synchronization mechanisms* (i.e., atomic variables, mutexes, lock objects) that establish ordering between conflicting actions. By default, these synchronization mechanism enforce sequential consistency, however C++ also promotes several additional memory orderings (i.e., *relaxed*, *acquire*, *release* and *consume*) that deviate from sequential consistency for producing more efficient software. Java, on the other hand, provides the *volatile* modifier and *synchronized* methods and statements for specifying programs without data races.

Klaim's semantic model, as it is presented in (DNFP98), guarantees sequential consistency. One of the first performance improvements for tuple-space implementations was the *ghost* tuple technique proposed in (RW96), and proven not to alter the sequentially consistent semantics in (DPR00). Three additional semantics for Linda-like languages have been proposed in (BGZ00), based on relaxing the atomicity of the `out` operation which is seen as composed of two phases called *emission* and *rending*. The three proposed semantics are called *instantaneous*, *ordered* and *unordered*, and to our knowledge no implementation and perfor-

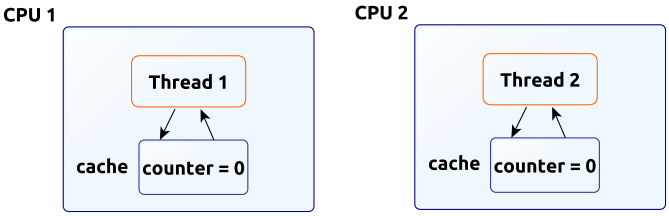


Figure 12: CPU caching of shared counter variable

mance comparison of the three semantics is yet carried out.

To the best of our knowledge, X10's memory model is at the present time only implicitly defined, by the implementation. Furthermore, the behavior of X10 programs is expected to be the same, regardless of which compiler backend is selected, C++ or Java. A recent work (Zwi16) proposed a reasoning for defining the X10 memory model under the assumption that X10 will gradually evolve into a language primarily dedicated to achieving high performance and targeting the C++ backend.

3.2.1 Data Races in SharedX10 and RepliKlaim Programs

In a multithreaded Java application each thread may create a replica of a shared variable into its CPU cache for performance reasons. Caching of the shared `counter` variable introduced in Figure 11 and below Listing 3.1 is illustrated in Figure 12. In general, the Java implementation does not provide strong guarantees for consistency of cached copies. In particular, the two cached copies in our example may be temporarily inconsistent, such that **thread 1** and **thread 2** observe different states for the same variable. Java offers the `volatile` keyword which gives guarantees that the replicas will be strongly consistent, i.e., ensures that only one consistency state may be observed.

Listing 3.1: Count class in Java (1)

```
1 public class Count {
2   private int counter = 0;
3   public void inc(value) {
4     this.counter += value;
5   }
6   public int getValue() {
7     return this.counter;
8   }
9 }
```

Write operations, such as the one that comprise the body of the `inc` method (`line 4` in Listing 3.1), do not by default exclusively access the memory, hence their interleaving during concurrent executions can lead to program errors. Indeed, if prior to execution of program in Figure 11(b), the `counter` value was 0, then as a result of the program execution, the new `counter` value can be 1 instead of 2; i.e., if **thread 1** is preempted by **thread 2** before storing the incremented value. To avoid this scenario, Java provides the `synchronized` keyword, and guarantees that two synchronized methods on the same object will not be interleaved. Listing 3.2 shows a specification that uses both `volatile` and `synchronized` modifiers to avoid both data races in programs shown in Figure 11.

Listing 3.2: Count class in Java (2)

```
1 public class Count {
2   public volatile int counter = 0;
3   public synchronized void inc(value) {
4     this.counter += value;
5   }
6   public int getValue() {
7     return this.counter;
8   }
9 }
```

As we have motivated in the Introduction in Chapter 1, caching is no longer held assumption for modern large-scale high-performance com-

RepliKlaim	SharedX10
$\texttt{out}_s\texttt{('count', 0)@}L$	**rvals**@places count:Counter

(a)

RepliKlaim	SharedX10
$\texttt{out}_w\texttt{('count', 0)@}L$	**rvalw**@places count:Counter

(b)

Figure 13: Replicating data in RepliKlaim and SharedX10 with strong (a) and (b) weak guarantees

puting systems. The approach we took in SharedX10 and RepliKlaim is to allow the programmer to use special primitives for specifying the placement of data replicas across the underlying physical units through abstractions: *nodes* in RepliKlaim, inherited from Klaim, and *places* in SharedX10, inherited from X10. Using the data race example we introduced, Figure 13 shows how such replication of the shared `count` variable can be expressed in RepliKlaim and SharedX10, with two consistency levels, namely *strong* (case (a)) and *weak* (case (b)). In general, RepliKlaim specifications rely on the $\texttt{out}_\alpha(t)@L$ operation to output tuple t on all nodes with localities in L, while in SharedX10, `rval`α`@places x` is used to replicate the variable `x` across places contained in the variable `places`, where $\alpha = s$ stands for the strong replication and $\alpha = w$ stands for weak.

The strong guarantees conceptually correspond to having a centralized data instance, while our weak guarantees correspond to sequential consistency, which is, as we outlined, typically assumed in high-level programming languages. We provide formal definitions of the two consistency models in the following section.

Specifications in Figure 11, along with the specification of the `Count` class in Listing 3.3, can as well serve to illustrate data races in SharedX10, as Java and SharedX10 are related with the object-oriented syntax. To avoid data race in those programs, two points needs to be addressed, similarly to the case in Java. Firstly, strong consistency of replicas eliminates the data race in scenario (a), which is achieved via declaring `rvals`

$\texttt{in}_w\texttt{('counter', ?x)@}l_1$ $\texttt{out}_w\texttt{('counter', x+1)@}\{l_1, l_2\}$	**thread 1**
$\texttt{rd('counter', ?x)@}l_2$	**thread 2**

(a)

$\texttt{in}_w\texttt{('counter', ?x)@}l_1$ $\texttt{out}_w\texttt{('counter', x+1)@}\{l_1, l_2\}$	**thread 1**
$\texttt{in}_w\texttt{('counter', ?x)@}l_2$ $\texttt{out}_w\texttt{('counter', x+1)@}\{l_1, l_2\}$	**thread 2**

(b)

Figure 14: Two specifications in RepliKlaim: with data race in (a) and no data race in (b)

`count`. Secondly, unlike in Java, in the SharedX10 implementation, invocation of methods on the same object are guaranteed not to interleave, except in the case when they are declared as `const` (see Section 5.3). The justification for this reasoning is following: `const` methods are not propagated to all replicas unlike regular methods; i.e., they are executed only against the local replica. Hence, to have a correct implementation of the `Count` class in Listing 3.3, `const` should be added in `getValue()` declaration, i.e., `public def getValue() const`.

Listing 3.3: Count class in SharedX10

```
1 public class Count {
2   var counter:Long = 0;
3   public def inc(value) {
4     this.counter += value;
5   }
6   public def getValue() {
7     return this.counter;
8   }
9 }
```

Unlike data items in SharedX10, in RepliKlaim the same data item could be accessed with operations of different consistency levels. Input

of a replicated tuple is typically enacted from a local node and entails removal of all replicas, while output requires the specification of all replica localities.

Figure 14 presents two RepliKlaim specifications analogue to previously shown; i.e., showing the interplay of a concurrent write and read (a) and two concurrent writes (b), where **thread 1** is executing at locality l_1, and **thread 2** is executing at locality l_2. A data race is present in the first scenario (a), due to the weak consistency which can leave localities temporarily inconsistent. To avoid the data race, **thread 1** needs to specify strong operations, i.e., $\texttt{in}_s$ and $\texttt{out}_s$. Scenario (b) shows concurrent execution of two write operations, however, there is no data race in this specification. This is due to the fact that there is always an ordering between two writes as `in` operation, in both variants $\texttt{in}_s$ and $\texttt{in}_w$, is a blocking operation.

3.3 Consistency Models for Replicated Data

In the past decades, many memory consistency models have produced and used across different research communities, i.e., distributed systems, databases, multiprocessor computer hardware (AG96; PWS+00; SS05). In particular, data replication is commonly used approach in these systems to improve availability and performance (TS06). Such systems that employ replication are referred to as replicated systems. For a programmer it is important to know what functionalities one can rely on when operating on replicated data. A replication consistency model abstracts away implementation details and identifies functionality of operations.

A particular replication consistency model is usually presented in terms of a condition that can be true or false for individual executions. If every possible execution that can occur makes the condition true, then it is said that the design satisfies the consistency model.

The most common consistency models in replicated systems according to (FR10) are:

Strong consistency. The simplest way the programmer can understand the behavior of a replicated systems is to ignore the replication; if every

execution on replicated system is the same as on an unreplicated system, with only a single site, it is said that system is strongly consistent, i.e., *linearizable* or *atomic*.

Sequential consistency. Compared to strong consistency, sequential consistency disregards the order of operations at different locations (e.g., threads in a concurrent system).

Release consistency. The idea of release consistency model is based on two operations labeled as *release* and *acquire*. Acquire operation is used to signal that a critical region (e.g., access to shared/replicated data) is about to be entered, while release operation signals that a critical region has just been exited. This consistency models is an example of a cache consistency model used in hardware architecture community, and in particular it has inspired a new memory consistency model called *regional consistency* (RRV14) with the aim of providing programmability and performance on non-cache-coherent systems.

Eventual consistency. Eventual consistency has attracted a lot of attention in large-scale distributed system, such as cloud systems (e.g., see discussion in (Vog09)). Replicating data allows any replica to be modified without remote synchronization, guaranteeing that all the replicas will *eventually* be consistent. Since updates can be executed at different replicas in different orders typically a conflict-resolution mechanism is required to implement this consistency model.

Main considerations in replicated systems can be summed up as follows:

- Replication improves performance by reducing access latency.
- Replication can lead to a potentially high network overhead of maintaining strong consistency.

  E.g., suppose an object is replicated N times. If read frequency is R, write frequency is W, and if $R \ll W$ then the outcome is likely high consistency overhead.

- The solution to decrease the synchronization cost and improve performance is to weaken the consistency guarantees.

Replication Consistency Guarantees in RepliKlaim and SharedX10

We now formulate replication consistency guarantees in SharedX10 and RepliKlaim associated with operating on replicated data. In particular, we considered two replication consistency levels in both languages that we refer to as *strong* and *weak* consistency. In the literature on consistency models, strong consistency is also known as *linearizability* (introduced by M.Herlihy in (HW90)), and it is the strictest consistency model, while our weak consistency corresponds to the notion of *sequential consistency* (introduced by L.Lamport in (Lam97)).

To describe the replication consistency guarantees we use the framework based on operation-ordering style presented in (FR10). For the purpose of completeness, we include the definitions of the two consistency models, along with the auxiliary ones, and a set of examples.

A **sequential data type** captures an aspect of a consistency model related to the semantics of operations in a replicated system. It is defined by a set of operations O, a set of states S, an initial state s_0, a set of return values R, function next-state: $O \times S \rightarrow S$ and function return-value: $O \times S \rightarrow R$. Formalization of a sequential data type is used to help the user to understand the behavior of a replicated system by relating it to a simpler, unreplicated system.

In order to make an intuitive presentation we describe a simple sequential data type with elements borrowed from the object-oriented domain. Hence, our set of operations O corresponds to $\{o.read()$, $o.write(x)$, for $o \in Obj$, $x \in Int\}$, where Obj is a finite set of data objects with one integer (Int) field, which is read by the $read()$ operation and written to by the $write()$ operation. The set of states S consists of functions $Obj \rightarrow Int$ which map objects to the values of their integer fields, the initial state has all objects mapped to zero; the return values are integers and the string OK (returned by the $write()$ operation) i.e., $R = Int \cup$ OK. The next-state function is defined by next-state($o.read()$,s) = s, next-state($o.write(x)$,s) = t, where $t : Obj \rightarrow Int$ such that $t(o) = s(o_1)$ if $o \neq o_1$ and $t(o) = x$. The

return-value function is then specified with return-value ($o.read()$, s) = $s(o)$, return-value($o.write(x)$,s) = OK. As one can observe, each operation in the model has a unique next state and a return value.

Our replicated system thus consists of parallel threads of execution which perform operations $read()$ and $write()$ on objects possibly replicated across several sites.

A **legal history** H is a sequence of pairs (operation, return value), where an operation is paired with its return value, and it is performed in the state that results from all the operations before it in the sequence, done in order.

The definition of strong consistency relies on a **real-time partial order**, $<_{E,rt}$, on operations that occur in the execution E, in which $p <_{E,rt} q$ means that the duration of operation p (time between invocation and return) occurs entirely before the duration of operation q. If the two operations overlap, they cannot be related by this order. The definition of sequential consistency uses **thread partial order**, $<_{E,t}$, in which $p <_{E,t} q$ means that p and q are executed by the same thread and that p returns before q is invoked.

As an example to illustrate the introduced notions, one can consider a parallel execution E of threads t_1 and t_2 such that thread t_1 executes $a.write(5)$ followed by $b.read()$, while t_2 executes $b.write(3)$ followed by $a.read()$. If we assume that $a.write(5) <_{E,rt} b.write(3)$, then the two legal histories whose total order is compatible with the real-time partial order in E are: H_1 = {($a.write$(5), OK), ($b.write$(3), OK), ($b.read$(),3), ($a.read$(),5)} and H_2 = {($a.write$(5), OK), ($b.write$(3), OK), ($a.read$(),5), ($b.read$(),3)}. An example of legal history whose total order is not compatible with the real-time partial order, but compatible with the thread partial order is H = {($b.write$(3), OK), ($a.write$(5), OK), ($b.read$(),3), ($a.read$(),5)}.

Strong consistency (Linearizability). Every execution on strongly replicated data is said to be *linearizable* i.e., its effect is atomic. The replica management algorithm states: each read is done on one (local) replica, each write is done on all replicas, different writes are done in the same

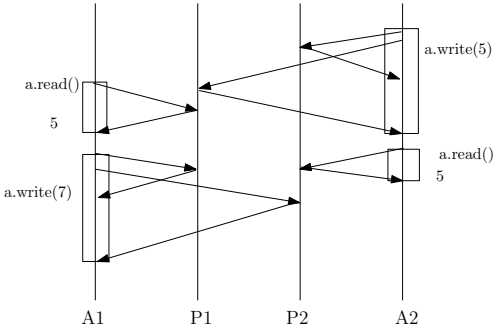


Figure 15: Linearizable execution

order at all replicas, and a write does not return until all replicas are modified.

Definition 2 (strong consistency) *Execution E is strongly consistent provided that there exists a history H such that:*

L1 H contains exactly the same operations that occur in E, each paired with the return value received in E,

L2 The total order of operations in H is compatible with the real-time partial order between operations that occur in E, $<_{E,rt}$

L3 H is a legal history of the sequential data type.

Sequential consistency. A weaker model is obtained by relaxing the condition L2 to allow reads to appear out of their real-time order. The obtained notion of consistency is sequential consistency. Unlike strong consistency, the replica management algorithm allows write to return before all replicas are modified.

Definition 3 (weak consistency) *An execution E is sequentially consistent provided that there exists a history H such that it is satisfies conditions L1 and L3 of Def. 1, and*

SC The total order of operations in H is compatible with the thread partial order, $<_{E,t}$.

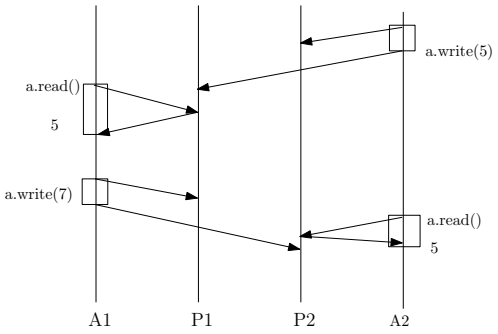


Figure 16: Sequentially consistent execution

Figures 15 and 16 show two space-time diagrams; time increases down the page, each operation is shown happening on a vertical line and messages are shown as diagonal arrows. Each rectangle illustrates the duration of operation - from it's invocation until it returns. A1 and A2 indicate a stream of operations coming from two concurrent threads, while P1 and P2 represent two sites each hosting one replica of `a`.

Figure 15 shows a linearizable execution as H' = {($a.write$(5), OK), ($a.read$(), 5), ($a.read$(), 5), ($a.write$(7), OK)} satisfies all three conditions. Figure 16 presents sequentially consistent (with the same history H'), but not linearizable execution because for any sequence H, ($a.read$(), 5) would have to be before ($a.write$(7), OK) which is not compatible with the real-time partial order on operations in E, where $a.write$(7) is ordered before $a.read$().

Strong and sequential consistency are both chosen as the basis of many concurrent programming constructs. Because of its strong constraints, linearizabiity is easier to reason about, however, it often incurs a high cost in terms of synchronization. Weaker notions of consistency are provided as a trade-off for better performance.

Part II

Contributions

Chapter 4

RepliKlaim

As we motivated in **Part I**, two key aspects in the design of distributed and parallel systems are *data locality* and *data consistency*. A proper design of those aspects can bring significant performance advantages, e.g. in terms of minimization of communication between computational entities.

In this chapter we present the coordination language RepliKlaim, a variant of Klaim, with first-class features to deal with data locality and consistency. In particular, the idea is to let the programmer specify and coordinate data replicas and operate on them with two levels of consistency, i.e., *strong* and *weak* (see Definition 2 and Definition 3 in Chapter 3). This chapter investigates issues related to replica consistency, provide an operational semantics and discuss the main synchronization mechanisms of our implementation. Finally, we provide a performance evaluation study in our prototype run-time system. Our experiments include scenarios where replica-based specifications and relaxed consistency provide significant performance gains.

Structure of the chapter

We start with the definition of the syntax in Section 4.1 and proceed then with the description of the operational semantics in Section 4.3. Section 4.2 discusses some examples aimed at providing some insights on

$$
\begin{array}{rcll}
N & ::= & \mathbf{0} \mid l :: [K, P] \mid N \parallel N & \text{(networks)} \\
K & ::= & \emptyset \mid \langle et_i, L \rangle \mid K, K & \text{(repositories)} \\
P & ::= & \mathtt{nil} \mid A.P \mid P + P \mid P \mid P & \text{(processes)} \\
A & ::= & \mathtt{out_s}(t_i)@L \mid \mathtt{in_s}(T_\iota)@\ell \mid \mathtt{read}(T_\iota)@\ell & \text{(strong actions)} \\
 & & \mathtt{out_w}(t_i)@L \mid \mathtt{in_w}(T_\iota)@\ell \mid & \text{(weak actions)} \\
 & & \mathtt{in_u}(T_\iota, L)@\ell \mid \mathtt{out_u}(et_i, L)@\ell \mid & \text{(unsafe actions)} \\
 & & \mathtt{eval}(P)@\ell & \text{(eval action)} \\
L & ::= & \epsilon \mid \ell \mid \underline{\ell} \mid L \bullet L & \text{(locations)} \\
t & ::= & e \mid P \mid \ell \mid t, t' & \text{(tuples)} \\
T & ::= & e \mid P \mid \ell \mid !x \mid T, T' & \text{(templates)}
\end{array}
$$

Figure 17: Syntax of RepliKlaim

semantics; implementation and performance aspects are detailed in Section 4.4, while Section 4.5 describes related work and provides a summary of the chapter.

4.1 Syntax

The syntax of RepliKlaim is based on Klaim's syntax (see Section 2.1 and Figure 5 in Chapter 2). The main difference is the extension of communication primitives to explicitly deal with replicas, and the absence of features to deal with recursion.

Definition 4 (RepliKlaim syntax) *The syntax of RepliKlaim is defined by the grammar of Figure 3, where $\mathcal{L}$ is a set of locations (ranged over by $\ell, \ell', \ldots$), $\mathcal{U}$ is a set of values (ranged over by $u, v, \ldots$), $\mathcal{V}$ is a set of variables (ranged over by $x, y, \ldots$), $!\mathcal{V}$ denotes the set binders over variables in $\mathcal{V}$ (i.e. $!x, !y, \ldots$), $\mathcal{I}$ is a set of tuple identifiers (ranged over by i, i', j, j'), $\mathcal{T} \subseteq (\mathcal{U} \cup \mathcal{V})^*$ is a set of $\mathcal{I}$-indexed tuples (ranged over by $t_i, t'_{i'}, \ldots$), $\mathcal{ET} \subseteq (\mathcal{U}^*$ is a set of $\mathcal{I}$-indexed evaluated tuples (ranged over by $et_i, et'_{i'}, \ldots$), and $\mathcal{TT} \subseteq (\mathcal{U} \cup \mathcal{V} \cup !\mathcal{V})^*$ is a set of templates (ranged over by $T_\iota, T'_{\iota'}, \ldots$, with $\iota \in \mathcal{I} \cup !\mathcal{V}$).*

Networks. A RepliKlaim specification is a *network* N, i.e. a possibly empty set of *components* or *nodes*.

Components. A component $\ell :: [K, P]$ has a locality name ℓ which is unique (see well-formedness in Definition 5), a *data repository* K, and parallel processes P. Components may model a data-coherent unit in a large scale system, where each unit has dedicated memory and computational resources.

Repositories. A data repository K is a set of data items, which are pairs of identifier-indexed tuples and their replication information. In particular a data item is a pair $\langle et_i, L\rangle$, where et_i is a tuple, i is a unique identifier of the tuple, and L is a list of localities where the tuple is replicated. For a data item $\langle et_i, L\rangle$ with $|L| > 1$ we say that t_i is *shared* or *replicated*. We use indexed tuples in place of ordinary anonymous tuples to better represent long-living data items such as variables and objects that can be created and updated. We require the replication information to be *consistent*, also this property is preserved by the semantics, as we show further below.

It is worth to note that a locality ℓ in L can appear as ℓ or as $\underline{\ell}$. The latter case denotes a sort of *ownership* of the tuple. Each replicated tuple is required to have exactly one owner (cf. well-formedness in Def. 5). This is fundamental to avoid inconsistencies due to concurrent *weak* operations, i.e, retrievals or updates of a replicated tuple.

Processes. Processes are the main computational units and can be executed concurrently either at the same locality or at different localities. Each process is created from the nil process, using the constructs for *action prefixing* ($A.P$), *non-deterministic choice* ($P + P$) and *parallel execution* ($P \mid P$).

Actions and targets. The actions of RepliKlaim are based on standard primitives for tuple spaces, extended to suitably enable replica-aware programming. Some actions are exactly as in Klaim. For instance, $\texttt{read}(T_\iota)@\ell$ is the standard non-destructive read of Klaim.

The standard output operation is enriched to allow a list of localities L as target. RepliKlaim features two variants of the output operation: a *strong* (i.e. atomic) one and a *weak* (i.e. *asynchronous*) one. In particular, $\texttt{out}_\alpha(t_i)@L$ is used to place the shared tuple t_i at the data repositories located on sites $l \in L$ atomically or asynchronously (resp. for $\alpha = \texttt{s}$ or $\alpha = \texttt{w}$). In this way the shared tuple is replicated on the set of sites designated with L. In RepliKlaim output operations are blocking: an operation $\texttt{out}_\alpha(t_i)@L$ cannot be enacted if an i-indexed tuple exists at L. This is necessary to avoid inconsistent versions of the same data item in the same location to co-exist. Hence, before placing a new version of a data item, the previous one needs to be removed. However, weak consistency operations still allow inconsistent versions of the same data item to co-exist but in *different* locations.

As in the case of output operations, RepliKlaim features two variants of the standard destructive operation $\texttt{in}$: a *strong* input $\texttt{in}_s$ and a *weak* input $\texttt{in}_w$. A strong input $\texttt{in}_s(T_\iota)@\ell$ retrieves a tuple et_i matching T_ι at ℓ and atomically removes all replicas of et_i. A weak input $\texttt{in}_w(T_\iota)@\ell$ tries to asynchronously remove all replicas of a tuple et_i matching T_ι residing in ℓ. This means that replicas are not removed simultaneously. Replicas in the process of being removed are called *ghost* replicas, since they are reminiscent of the *ghost* tuples of (RW96; DPR00) (cf. the discussion in Section 4.5).

RepliKlaim features two additional (possibly) *unsafe* operations: $\texttt{out}_u(et_i, L)@\ell$ puts a data item $\langle et_i, L\rangle$ at all locations in L, while $\texttt{in}_u(T_\iota, L)@\ell$ retrieves a tuple et_i matching T_ι at ℓ and does not remove the replicas of et_i. These operations are instrumental for the semantics and are not meant to appear in user specifications.

As we showed, the syntax of RepliKlaim admits some terms that we would like to rule out. We therefore define a simple notion of well-formed network.

Definition 5 (well-formedness) *Let N be a network. We say that N is well formed if:*

1. *Localities are unique, i.e. no two distinct components $\ell :: [K, P]$, $\ell :: [K', P']$ can occur in N;*

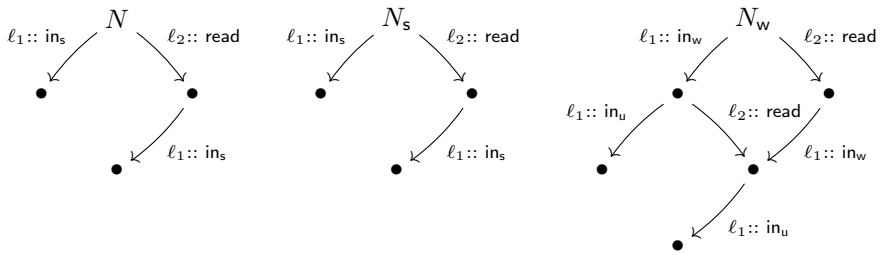


Figure 18: Concurrent reads and inputs with no replicas (left), replicas and strong input (center) and weak input (right).

2. *Replication is consistent, i.e. for every occurrence of* $\ell :: [(K, \langle et_i, L\rangle), P]$ *in a network* N *it holds that* $\ell \in L$ *and for all (and only) localities* $\ell' \in L$ *we have that component* ℓ' *is of the form* $\ell' :: [(K', \langle et'_i, L\rangle), P']$*. Note that* et' *is not required to be* et *since we allow relaxed consistency of replicas.*

3. *Each replica has exactly one owner, i.e. every occurrence of* L *has at most one owner location* $\underline{\ell}$*.*

4. *Tuple identifiers are unique, i.e. there is no* K *containing two data items* $\langle et_i, L\rangle$*,* $\langle et'_i, L'\rangle$*. Note that this guarantees local uniqueness; global uniqueness is implied by condition (2).*

Well-formedness is preserved by the semantics, but as usual we admit some intermediate bad-formed terms which ease the definition of the semantics.

We assume the standard notions of free and bound variables, respectively denoted by $fn(\cdot)$ and $bn(\cdot)$.

4.2 Examples

We provide here a couple of illustrative examples aimed at providing insights on semantics, implementation and performance aspects. In below we use notation $\doteq$ to define a RepliKlaim specification.

Concurrent reads and inputs. The following example illustrates three ways of sharing and accessing a tuple and is meant to exemplify the benefit of replicas and weak inputs. The example consists of the networks

$$\begin{array}{rcl} N & \doteq & \ell_1 :: [\langle et_i, \ell_1\rangle, \mathtt{in}_{\mathtt{s}}(T_\iota)@\ell_1] \\ & & \| \\ & & \ell_2 :: [\emptyset, \mathtt{read}(T_\iota)@\ell_1] \end{array}$$

$$\begin{array}{rcl} N_\alpha & \doteq & \ell_1 :: [\langle et_i, \{\underline{\ell_1}, \ell_2\}\rangle, \mathtt{in}_\alpha(T_\iota)@\ell_1] \\ & & \| \\ & & \ell_2 :: [\langle et_i, \{\underline{\ell_1}, \ell_2\}\rangle, \mathtt{read}(T_\iota)@\ell_2] \end{array}$$

with $\alpha \in \{\mathtt{s}, \mathtt{w}\}$. The idea is that in N a tuple that has to be accessed by both ℓ_1 and ℓ_2 is shared in the traditional Klaim way: it is only stored in one location (namely, ℓ_1) with no replicas. On the contrary, N_α models the same scenario with explicit replicas. The tuple et_i is replicated at both ℓ_1 and ℓ_2, possibly after some process executed $\mathtt{out}(t_i)@\{\underline{\ell_1}, \ell_2\}$. Networks N_{s} and N_{w} differ in the way the tuple et_i is retrieved by ℓ_1: using strong or weak input, respectively. Figure 18 depicts the transition systems for the three networks. The transition systems of N and N_{s} are similar but differ in the way the transitions are computed. In N, the input is local to ℓ_1, but the read is remote (from ℓ_2 to ℓ_1), while in N_{s} the input is global (requires a synchronization of ℓ_1 and ℓ_2 to atomically retrieve all replicas of et_i), and the read is local to ℓ_2. The main point in N_{w} is that the process in ℓ_2 can keep reading the ghost replicas of et_i even after ℓ_1 started retrieving it.

Concurrent reads and outputs. The next example illustrates (see also Fig. 19) the interplay of reads with strong and weak outputs.

$$M_\alpha \quad \doteq \quad \ell_1 :: [\emptyset, \mathtt{out}_\alpha(t_i)@\{\underline{\ell_1}, \ell_2\}] \quad \| \quad \ell_2 :: [\emptyset, \mathtt{read}(T_\iota)@\ell_1]$$

with $\alpha \in \{\mathtt{s}, \mathtt{w}\}$. The idea is that component ℓ_1 can output a tuple with replicas in ℓ_1 and ℓ_2 in a strong or weak manner, while ℓ_2 is trying to read the tuple from ℓ_1. In the strong case, the read can happen only after all replicas have been created. In the weak case, the read can be interleaved with the unsafe output.

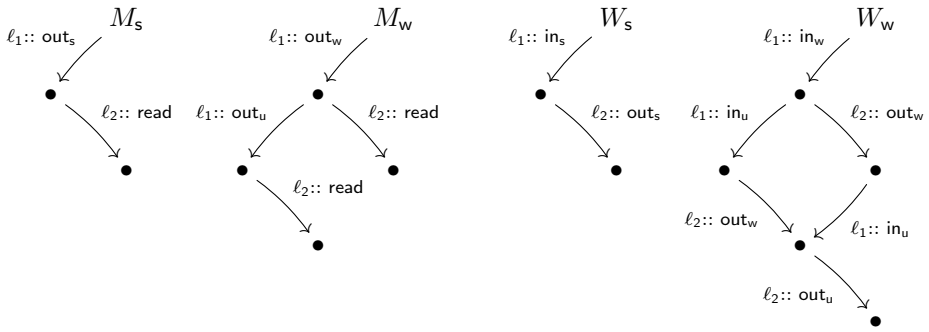


Figure 19: Transitions for M_{s} (concurrent read and strong output), M_{w} (concurrent read and weak output), W_{s} (concurrent strong input and strong output) and W_{w} (concurrent weak input and weak output).

Concurrent inputs and outputs. The last example (see also Figure 19) illustrates the update of a data item using strong and weak operations.

$$\begin{array}{rcl} W_\alpha & \doteq & \ell_1 :: [\langle et_i, \{\underline{\ell_1}, \ell_2\}\rangle, \mathtt{in}_\alpha(T_i)@\{\underline{\ell_1}, \ell_2\}.\mathtt{out}_\alpha(f(et)_i)@\{\underline{\ell_1}, \ell_2\}] \\ & \| & \ell_2 :: [\langle et_i, \{\underline{\ell_1}, \ell_2\}\rangle, \mathsf{nil}] \end{array}$$

with $\alpha \in \{\mathtt{s}, \mathtt{w}\}$. The idea is that component ℓ_1 retrieves a tuple and then outputs an updated version of it (after applying function f). Relaxing consistency from $\mathtt{s}$ to $\mathtt{w}$ increases the number of interleavings.

4.3 Structural Operational Semantics

RepliKlaim terms are to be intended up to the structural congruence induced by the axioms in Figure 20 and closed under reflexivity, transitivity and symmetry. As usual, besides axiomatising the essential structure of RepliKlaim systems, the structural congruence allows us to provide a more compact and simple semantics. The axioms of the structural congruence are standard. We just remark the presence of a *clone* axiom (bottom) which is similar to the one used in early works on Klaim. In our case, this clone axiom allows us to avoid cumbersome semantic rules for

$$
\begin{array}{rcl}
P + (Q + R) & \equiv & (P + Q) + R \\
P + \mathsf{nil} & \equiv & P \\
P + Q & \equiv & Q + P \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R \\
P \mid \mathsf{nil} & \equiv & P \\
P \mid Q & \equiv & Q \mid P \\
N \parallel (M \parallel W) & \equiv & (N \parallel M) \parallel W \\
N \parallel \mathbf{0} & \equiv & N \\
N \parallel M & \equiv & M \parallel N \\
\ell :: [K, P] & \equiv & \ell :: [K, \mathsf{nil}] \parallel \ell :: [\emptyset, P]
\end{array}
$$

Figure 20: Structural congruence for RepliKlaim

dealing with multiparty synchronisations where the subject component is also an object of the synchronisation (e.g. when a component ℓ removes a shared tuple t_i that has a replica in ℓ itself). The clone axiom allows a component to participate in those interactions, by separating the processes (the subject) from the repository (the object). It is worth to note that this axiom does not preserve well-formedness (uniqueness of localities is violated).

$$\frac{}{A.P \xrightarrow{A} P} \text{ (ACTP)} \qquad \frac{P \xrightarrow{A} P'}{P+Q \xrightarrow{A} P'} \text{ (CHOICE)} \qquad \frac{P \xrightarrow{A} P'}{P|Q \xrightarrow{A} P'|Q} \text{ (PAR)}$$

$$\frac{P \xrightarrow{\mathsf{eval}(Q)@\ell'} P'}{N\|\ell::[K,P]\|\ell'::[K_{\ell'},P_{\ell'}] \longrightarrow N\|\ell::[K,P']\|\ell'::[K_{\ell'},P_{\ell'}|Q]} \text{ (EVAL)}$$

$$\frac{P \xrightarrow{\mathsf{out_s}(t_i)@L} P' \qquad \forall \ell' \in L.\nexists et', L'.\langle et_i', L'\rangle \in K_{\ell'} \qquad et_i = \mathcal{T}[\![t_i]\!]}{N\|\ell::[K,P]\|\Pi_{\ell' \in L}\ell'::[K_{\ell'},P_{\ell'}] \longrightarrow N\|\ell::[K,P']\|\Pi_{\ell' \in L}\ell'::[(K_{\ell'},\langle et_i,L\rangle),P_{\ell'}]} \text{ (OUTS)}$$

$$\frac{P \xrightarrow{\mathsf{out_w}(t_i)@L} P' \qquad \ell'' \in L \qquad \nexists et', L'.\langle et_i', L'\rangle \in K_{\ell''} \qquad et_i = \mathcal{T}[\![t_i]\!]}{N\|\ell::[K,P]\|\ell''::[K_{\ell''},P_{\ell''}] \longrightarrow N\|\ell::[K,P']\|\ell''::[(K_{\ell''},\langle et_i,L\rangle),P_{\ell''}|\Pi_{\ell' \in (L \setminus \ell'')}\mathsf{eval}(\mathsf{out_u}(et_i,L))@\ell']} \text{ (OUTW)}$$

$$\frac{P \xrightarrow{\mathsf{out_u}(et_i,L)} P' \qquad \nexists et', L'.\langle et_i', L'\rangle \in K}{N\|\ell::[K,P] \longrightarrow N\|\ell::[(K,\langle et_i,L\rangle),P']} \text{ (OUTU)}$$

Figure 21: Operational semantics of RepliKlaim [1]

$$\frac{P \xrightarrow{\mathsf{in_s}(T_\iota)@\ell''} P' \qquad \ell'' \in L \qquad \sigma = match(T_\iota, et_i)}{N \| \ell :: [K, P] \| \Pi_{\ell' \in L} \ell' :: [(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \longrightarrow N \| \ell :: [K, P'\sigma] \| \Pi_{\ell' \in L} \ell' :: [K_{\ell'}, P_{\ell'}]} \quad (\textsc{InS})$$

$$\frac{P \xrightarrow{\mathsf{in_w}(T_\iota)@\ell''} P' \qquad \ell'' \in L \qquad \underline{\ell'} \in L \qquad \sigma = match(T_\iota, et_i)}{N \| \ell :: [K, P] \| \ell' :: [(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \longrightarrow N \| \ell :: [K, P'\sigma] \| \ell' :: [K_{\ell'}, P_{\ell'} | \prod_{\ell''' \in (L \setminus \ell')} \mathsf{eval}(\mathsf{in_u}(et_i, L))@\ell''']} \quad (\textsc{InW})$$

$$\frac{P \xrightarrow{\mathsf{in_u}(T_\iota, L)} P' \qquad \sigma = match(T_\iota, et_i)}{N \| \ell :: [(K, \langle et_i, L \rangle), P] \longrightarrow N \| \ell :: [K, P']} \quad (\textsc{InU})$$

$$\frac{P \xrightarrow{\mathsf{read}(T_\iota)@\ell'} P' \qquad \sigma = match(T_\iota, et_i)}{N \| \ell :: [K, P] \| \ell' :: [(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}] \longrightarrow N \| \ell :: [K, P'\sigma] \| \ell' :: [(K_{\ell'}, \langle et_i, L \rangle), P_{\ell'}]} \quad (\textsc{Read})$$

Figure 22: Operational semantics of RepliKlaim [2]

The operational semantics in Figures 21 and 22 mix an SOS style for collecting the process actions (cf. rules ACTP, CHOICE and PAR) and reductions for the evolution of nets. The rules for defining pattern-matching, $match(T_\iota, t_i)$, which yields a substitution for the bound variables of T_ι and the evaluation function are already given in Table 2 and Table 1 of Chapter 2. Note that ι may be a bound variable to record the identifier of the tuple. We use $\prod$ to denote parallel composition of processes and P_σ, where $\sigma = match(T, et)$, to indicate the substitution of T for *et* in *P*. The standard congruence rules are not included for simplicity.

It is worth to remark that the replicas located at the owner are used in some of the rules as tokens to avoid undesirable race conditions. The role of such tokens in inputs and outputs is dual: the replica must *not* exist for output to be enacted, while the replica *must* exist for inputs to be enacted.

Rule OUTS deals with a strong output $\texttt{out}_s(t_i)@L$ by putting the evaluated tuple et_i in all localities in L. However, the premise of the rule requires a version of data item i (i.e. a tuple et_i') to *not* exist in the repository of the owner of et_i (ℓ'').

Rule OUTW governs weak outputs of the form $\texttt{out}_w(t_i)@L$ by requiring the absence of a version of data item i. The difference with respect to the strong output is that the effect of the rule is that of creating a set of processes that will take care of placing the replicas in parallel, through the unsafe output operation. Such operation is handled by rule OUTU which is very much like a standard Klaim rule for ordinary outputs, except that the operation is blocking to avoid overwriting existing data items.

Rule INS deals with actions $\texttt{in}(T_\iota)@\ell$ by retrieving a tuple et_i matching T_ι from locality ℓ, and from all localities containing a replica of it. Rule INW retrieves a tuple et_i from an owner ℓ' of a tuple that has a replica in the target ℓ. As a result, processes are installed at ℓ' that deal with the removal of the remaining replicas in parallel (thus allowing the interleaving of `read` operations). As in the case of weak outputs, weak inputs resort to unsafe inputs. Those are handled by rule INU, which is like a standard input rule in Klaim.

Finally, rule READ is a standard rule for dealing with non-destructive reads.

4.4 Performance Evaluation

We describe in this section our implementation and present a set of experiments aimed at showing in which conditions an explicit use of replicas can provide significant performance advantages.

Implementing RepliKlaim in KLAVA.

The run-time framework is based on KLAVA, a Java package used for implementing distributed applications based on Klaim. KLAVA provides a set of process executing engines (nodes) connected in a network via one of the three communication protocols (TCP, UDP, local pipes). The implementation of RepliKlaim is based on an encoding of RepliKlaim into standard Klaim primitives. However, we do not provide here a formal encoding from RepliKlaim to Klaim, instead we take a practical approach. To this end, we present several code snippets that illustrate how RepliKlaim primitives are rendered in KLAVA. Such implementation is guided by the semantic rules presented in Figures 21 and 22. Hence, replicated tuples are indexed with integers and contain localities of replicas in a data structure which permits standard tuple space operations, and finally, the *owner* of a tuple is the first locality contained in such data structure. Listing 4.1 shows the rendering of $\texttt{in}_w(t)@\ell$.

Listing 4.1: Implementation of $\texttt{in}_w$

```
1  TupleSpaceVector vectorLoc = t.getItem(1);
2  Tuple template = new Tuple(new PhysicalLocality());
3  vectorLoc.read_nb(template);
4  PhysicalLocality ploc = template.getItem(0);
5  in(t, ploc);
6  while(vectorLoc.read_nb(template)) {
7    PhysicalLocality ploc = template.getItem(0);
8    eval(new InU(t), ploc);
9    template.resetOriginalTemplate();
10 }
```

`Lines 1–5` withdraw replica from the owner, while following lines withdraw remaining replicas asynchronously via the `eval` operation, which triggers execution of the `executeProcess` method defined in the `InU` class (see below Listing 4.2).

Listing 4.2: `InU` implementation snippet

```
1 public void executeProcess() throws KlavaException {
2   in(t, self);
3 }
```

$\texttt{out}_w$ is implemented similarly to $\texttt{in}_w$, including a difference at `line 8` such that a tuple is inserted relying on the `OutU` definition. In addition, before a new version is inserted, it is required to check that the older version is not present. Listing 4.3 show the implementation of the `executeProcess` method of `OutU` class.

Listing 4.3: `OutU` implementation snippet

```
1 public void executeProcess() throws KlavaException {
2   if(!read_nb(t, self))
3     out(t, self);
4 }
```

Implementation of strong operations requires using a lock, i.e., token, to ensure that each access to replicated data is *atomic*. Token is represented with two fields, one that specifies index of the replicated tuple and the other stating that the tuple is representing a *token*. As an example, an expression `new Tuple(new KInteger(0), new KString("token"))` creates a token for synchronizing accesses to replicated tuple indexed with `0`. Such token is stored at the owner location once the tuple is replicated for the first time. Finally, Listing 4.4 shows how $\texttt{in}_s(t)@\ell$ in rendered in KLAVA.

Listing 4.4: Implementation of $\texttt{in}_s$

```
1 TupleSpaceVector vectorLoc = t.getItem(1);
2 Tuple template = new Tuple(new PhysicalLocality());
3 vectorLoc.read_nb(template);
4 PhysicalLocality tokenLoc = template.getItem(0);
5 in(new Tuple(new KInteger(0), new KString("token"),
    ↪ tokenLoc);
```

```
6  template.resetOriginalTemplate();
7  while(vectorLoc.read_nb(template)) {
8    PhysicalLocality ploc = template.getItem(0);
9    eval(new InU(t), ploc);
10   template.resetOriginalTemplate();
11 }
12 out(new Tuple(new KInteger(0), new KString("token"),
       ↪ tokenLoc);
```

Experiments: Hypothesis

The main hypothesis of our experiments is that better performances are achieved with improved data locality and data communication minimized through the use of replicated tuples and weak operations. Indeed, maximizing data locality can be easily done by replicating data, however it comes at a cost in terms of synchronization if replicated data need to be kept consistent (e.g. when using strong inputs and outputs). Our experimental results show how the ratio between the frequencies of read and update (i.e. sequences of inputs and outputs on the same data item) operations affects the performance of three different versions of a program: a *traditional* one that does not use replicas, and two versions using replicas: one using strong operations and another one using weak operations. However, we had to deviate in one thing from the semantics: while spawning parallel processes in rules INW and OUTW to deal with the asynchronous/parallel actions on replicas seems very appealing, in practice performing such operations in sequence showed to be more efficient. Of course, in general, the choice between parallel and sequential composition of such actions depends on several aspects, like the number of available processors, the number of processes already running in the system and the size of the data being replicated.

Experiments: Configuration of the Scenario

The general idea of the scenario we have tested is that multiple nodes are concurrently working (i.e. performing inputs, reads and outputs) on a list whose elements can be scattered on various nodes. A single element

(i.e. the counter) is required to indicate the number of the next element that can be added. In order to add an element to the list, the counter is removed using an input, the value of the counter is increased and the tuple is re-inserted, and then a new list element is inserted. We call such a sequence of input and output operations on the same data item (i.e. the counter) an *update* operation. The source code and Klava library are available online at below link[1].

Each of the nodes is running processes that perform read or update operations. Both reader and updater processes run in loops. We fix the number of updates to 10, but vary the number of read accesses (20, 30, 50, 100, 150, 200). We consider two variants of the scenario. The first variant has 3 nodes: one node containing just one reader process, another node containing just one updater process and a last one containing both a reader and an updater process. The second variant has 9 nodes, each containing process as in the previous case, i.e. this scenario is just obtained by triplicating the nodes of the previous scenario. The main point for considering these two variants is that we run the experiment in a dual core machine, so that in the first case one would ideally have all processes running in parallel, while this is not the case in the second variant.

Formally, the RepliKlaim nets N we use in our experiments are specified as follows

$$N \doteq \prod_{i=1}^{n} \left\{ \ell_{i,1} :: [\emptyset, \mathsf{P}_1(\ell_{i,1})] \parallel \ell_{i,2} :: [\emptyset, \mathsf{P}_2(\ell_{i,2})] \parallel \ell_{i,3} :: [\emptyset, \mathsf{P}_1(\ell_{i,3}) \mid \mathsf{P}_2(\ell_{i,3})] \right\}$$

where P_1 is an updater process and P_2 is a reader process, both parametric with respect to the locality they reside on. P_1 is responsible for incrementing the counter and adding a new list element, while P_2 only reads the current number of list elements. For the scalability evaluation we compare results for nets obtained when $n = 1$ and $n = 3$, meaning that corresponding nets have 3 and 9 nodes respectively. Our aim is to compare the following three alternative implementations of processes

[1] `http://sysma.imtlucca.it/wp-content/uploads/2015/03/RepliKlaim-test-examples.rar`

P_1 and P_2 which offer the same functionality, but exhibit different performances:

Program **no-replicas**: this implementation follows a standard approach that does not make use of replica-based primitives. The idea here is that the shared tuple is stored only in one location, with no replicas. The consistency of such model is obviously strong, as there are no replicas. Local access to the shared tuple is granted only to processes running on the specified location, while other processes access remotely. In the begining we assume that one of the sites has executed $\mathtt{out}_s(counter_a)@\ell_1$ which places the counter tuple $counter_a$ at place ℓ_1, with a being a unique identifier. Then processes P_1 and P_2 can be expressed as follows:

$$\begin{array}{lcl} \mathsf{P}_1(self) & \equiv & \mathtt{in}_s(counter_a)@\ell_1. \\ & & \mathtt{out}_s(f(counter_a))@\ell_1. \\ & & \mathtt{out}_s(lt_{a_{counter}})@self.\mathsf{P}_1(self) \end{array}$$

$$\mathsf{P}_2(self) \equiv \mathtt{read}(T_a)@\ell_1.\mathsf{P}_2(self)$$

where $f(\cdot)$ refers to the operation of incrementing the counter and lt refers to the new list element which is added locally after the shared counter has been incremented. Note that we use a as unique identifier for the counter and $a_{counter}$ as unique identifier for the new elements being inserted.

Program **strong-replicas**: The difference between this model and the non-replicated one is the presence of replicas on each node, while this model also guarantees strong consistency. Concretely, each update of replicated data items is done via operations $\mathtt{in}_s$ and $\mathtt{out}_s$. The formalization is presented below, after the description of the weak variant of this implementation.

Program **weak-replicas**: In this variant, the replicas are present on each node, but the level of consistency is weak. This means that interleavings of actions over replicas are allowed. However, to make this program closer to the functionality offered by the above

ones, we forbid the co-existence of different versions of the same data item. Such co-existence is certainly allowed in sequences of operations like $\texttt{in}_w(t_i)@\ell.\texttt{out}_w(t_i')@L$ as we have seen in the examples of Section 4.2. To avoid such co-existence, but still allow concurrent reads we use an additional tuple that the updaters used as sort of lock to ensure that outputs (respectively, inputs) are only enacted once inputs (respectively, outputs) on the same data item are completed on all replicas. Of course, this makes this program less efficient than it could be but it seems a more fair choice for comparison and still our results show its superiority in terms of performance.

In the above two replication-based implementations we assume that the counter is replicated on all nodes by executing $\texttt{out}_\alpha(\mathit{counter}_a)@\{\underline{\ell_1}, \ell_2, \ell_3\}$ with $\alpha \in \{\texttt{s}, \texttt{w}\}$. In this case the processes are specified as:

$$\begin{array}{lcl} \mathsf{P}_1(\mathit{self}) & \equiv & \texttt{in}_\alpha(\mathit{counter}_a)@\mathit{self}.\texttt{out}_\alpha(f(\mathit{counter}_a))@\{\ell_1,\ell_2,\ell_3\}. \\ & & \texttt{out}_s(a_{\mathit{counter}})@\mathit{self}.\mathsf{P}_1(\mathit{self}) \\ \mathsf{P}_2(\mathit{self}) & \equiv & \texttt{read}(T_a)@\mathit{self}.\mathsf{P}_2(\mathit{self}) \end{array}$$

where the strong and weak variants are obtained by letting α be $\texttt{s}$ and $\texttt{w}$, respectively.

Experiments: Data and Interpretation

The results of our experiments are depicted in Figures 23 and 24. The x axis corresponds to the ratio of reads and updates performed by all processes, while the y axis corresponds to the time needed by the processes to complete their computation. We measure the relation between average running time and the ratio between access frequencies. Time is expressed in seconds and presents the average of 15 executions, while the ratio is a number (2, 3, 5, 10, 15, 20). The results obtained for programs **no-replicas**, **strong-replicas** and **weak-replicas** are respectively depicted in blue, green and red.

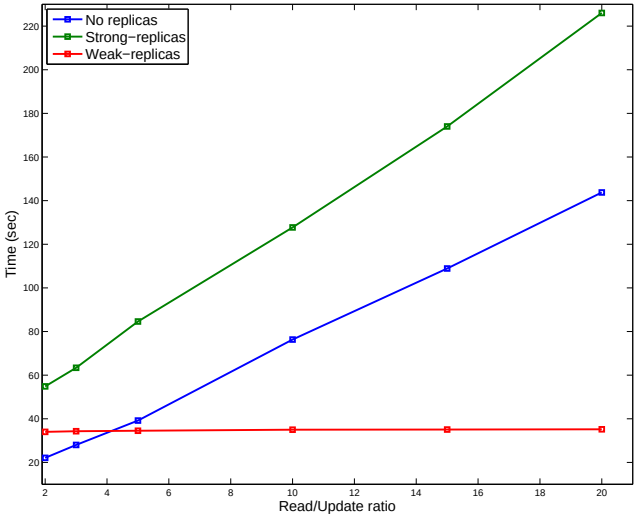


Figure 23: Comparing three strategies in a scenario with 3 nodes

It can be easily observed that when increasing the ratio the **weak-replicas** program is the most efficient. This program improves over program **no-replicas** only after the ratio of reading operations reaches a certain level that varies from the two variants used (3 and 9 nodes). Results show that **strong-replicas** offers the worst performance. Indeed, as we hinted in Section 3.3, preserving strong consistency in presence of replicas is often unfeasible in practice because it requires a great deal of synchronization.

4.5 Summary and Related Work

In this chapter we presented the RepliKlaim language, which enriches Klaim with primitives for data sharing. In particular, the new primitives allow the programmer to specify and coordinate the replication of shared data items and the desired consistency properties so to obtain better per-

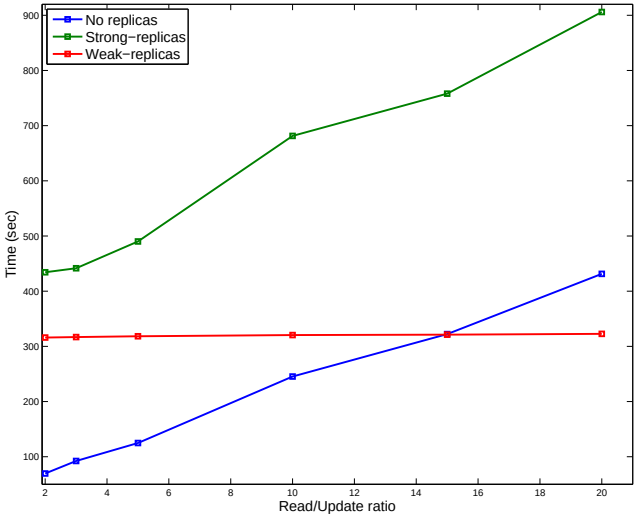


Figure 24: Comparing three strategies in a scenario with 9 nodes

formances. We provided an operational semantics to formalize our proposal as well as to guide the implementation of the language in KLAVA, a Java-based implementation of Klaim. We also discussed issues related to replica consistency and the main synchronization mechanisms of our implementation. Finally, we provided an evaluation study which compares performances of programs following different strategies for data sharing.

Many authors have investigated issues related to the performance of tuple space implementations and applications of tuple space coordination to large-scale distributed and concurrent systems (cloud computing, high-performance computing, services, etc.).

One of the first performance improvements for tuple-space implementations was the *ghost* tuple technique (RW96), which improves local operations by allowing existence of local tuple replicas under several conditions. Another seminal work considering performance issues in tu-

ple space coordination was the introduction of asynchronous tuple space primitives in Bonita (asynchronous Linda) (Row97). This work provided a practical implementation and an illustrative case study to show the performance advantages of asynchronous variants of tuple space primitives for coordinating distributed agents. A thorough theoretical study of possible variants of tuple space operations is presented in (BGZ00). In particular, the authors studied three variants for the output operation: an instantaneous output (where an output can be considered as instantaneous creation of the tuple), and ordered output (where a tuple is placed in the tuple space as one atomic action) and an unordered output (where the tuple is passed to the tuple space handler and the process will continue, the tuple space handler will then place the tuple in the tuple space, not necessarily respecting order of outputs). A clear understanding of (true) concurrency of tuple space operations was developed in (BGZ97), where the authors provide a contextual P/T nets semantics of Linda. All these works have inspired the introduction of the asynchronous weak operations in RepliKlaim.

Performance issues have been also considered in tuple space implementations. Besides Klaim implementations (BDP02; BDL06), we mention GigaSpaces (Gig17), a commercial tuple space implementation, Blossom (vdG01), a C++ high performance distributed tuple space implementation, Lime (MPR06), a tuple space implementation tailored for ad-hoc networks, TOTA (MZ09), a middleware for tuple-based coordination in multi-agent systems, and PeerSpace (BMZ04) a P2P based tuple space implementation. Moreover, tuple space coordination has been applied and optimised for a large variety of systems where large-scale distribution and concurrency are key aspects. Among other, we mention large-scale infrastructures (Cap08), cluster computing environments (Atk10), cloud computing systems (Har12), grid computing systems (LP05), context-aware applications (BCP07), multi-core Java programs (GH11), and high performance computing systems (JXLY06). As far as we know, none of the above mentioned implementations treats replicas as first-class programming citizens.

Chapter 5

SharedX10

In Chapter 4 we introduced our first contribution, namely, the RepliKlaim language which enriched Klaim with primitives for data sharing. This chapter presents the transfer of ideas from our work on Klaim to X10, which resulted in the SharedX10 language. Klaim's notions for locality and asynchronous computation, i.e., node/locality and `eval(s)@p`, are comparable with X10's place and the `async` operation, where `at(p) async s` is used to spawn a new activity (i.e., process) at locality `p` to remotely execute `s`. However, as we outlined in Chapter 2, access to data in Klaim is uniform, while in X10 there are specific mechanisms to be used depending on whether the data item is local to the accessing activity or remote. This has lead us to introduce primitives for sharing based on a centralized data location in SharedX10, in addition to sharing based on replication with *strong* and *weak* levels of consistency, that had previously been introduced in RepliKlaim.

Structure of the chapter

Section 5.1 introduces the primitives for data sharing, while the complete syntax of SharedX10 is presented in Section 5.2. Section 5.3 presents the encoding scheme for SharedX10 programs. Section 5.4 introduces the implementation of SharedX10 syntax in terms of the XTEXT grammar. Section 5.5 reports on a number of performance experiments aimed at

comparing different strategies of data sharing. Section 5.6, provides a brief summary of the chapter.

```
 i sval x:T
ii rvals@l:ArrayList[Place] x:T
iii rvalw@l:ArrayList[Place] x:T

iv sval x:DistArray[T]
 v rvals@l:DistArray[ArrayList[Place]] x:DistArray[T]
vi rvalw@l:DistArray[ArrayList[Place]] x:DistArray[T]
```

Figure 25: Syntax of SharedX10 primitives for declaring shared variable `x`

5.1 Primitives for Data Sharing

As we motivated in Section 2.2 of Chapter 2, which introduced the X10 programming model, the work on SharedX10 is motivated by our critical view of X10 mechanisms for data management. In particular, the data replication (i.e., value copying) is implicit and can lead to program errors, while the use of global references for data communication adds to the complexity of the program.

This section introduces the concept of *shared data*, which is the core of SharedX10, and primitives for specifying sharing strategies, with the following aims:

1. To allow programmers to easily express how data is shared, with several possibilities to help programmers fit shared data access patterns to the need of their applications.

2. To hide the orchestration of data communication from the programmer.

3. To allow programmers to sacrifice performance for the sake of programmability, i.e., to allow programmers to write programs as in shared-memory languages (e.g. Java), evading the problems associated with value-copying and global references.

Figure 25 shows the syntax of SharedX10 primitives for declaring shared variables. The main idea is that the SharedX10 programmer uses the primitives to specify data sharing according to the application sharing pattern. When designing the primitives we had in mind common variable access patterns, i.e., *read-mostly*, *producer-consumer*, *general read-write* and *stencil*. Accesses to a shared variable follow the

- *read-mostly* pattern if the variable is initialized once and subsequently only read, sequentially or concurrently, by activities from multiple places;
- *producer-consumer* pattern if variable is updated to by activities (writers) from a single place and read by activities (readers) from possibly multiple places;
- *general read-write* pattern if the variable is read and updated to by activities from multiple places;
- *stencil* pattern if the variable is organized as a distributed array and each array element is updated with the contributions from a subset of neighbor variables.

Shared variable refers either to an object of built-in or user-defined type denoted with `T` (primitives (*i*) - (*iii*)) or to a one-dimensional distributed array of objects of type `T` (primitives (*iv*) - (*vi*)). When declaring a shared variable, the programmer decides whether data should be stored at a single place (`sval`) or replicated (`rvals` and `rvalw`). When sharing is based on replication, the programmer also provides an array of places where replicas would be allocated, denoted in Figure 25 with `l`.
Sharing based on a centralized data location. Primitive (*i*) declares shared a `sval` variable to be stored at a single place, i.e., the place of allocation in the program. If an array is declared as `sval` (primitive (*iv*))

then each array element is to be stored at a single place. This sharing pattern fits general read-write data access pattern, as the replication-based strategy would reduce read latency but increase write latency due to the added expense of updating all data replicas.

Sharing based on the replication strategy creates data replicas at places provided in an (array) variable `l` of type `ArrayList[Place]` (primitives (*ii*) and (*iii*)). If the replicated variable refers to a distributed array (primitives (*v*) and (*vi*)), then the expected type of `l` is `DistArray[ArrayList[Place]]` such that an array element `x(i)` is replicated over the list of places contained in `l(i)` (*pair-wise* sharing). This strategy fits producer-consumer, read-mostly and stencil pattern, as localizing accesses, in the presence of infrequent writes, avoids expensive data movements from remote places. The programmer is expected to annotate read methods as `const`, while the other are considered as write (see example in Section 3.2). Data sharing based on replication in SharedX10 offers two levels of replica consistency, namely *strong* and *weak*.

Sharing based on strong data replication. Primitives (*ii*) and (*v*) declare `rvals` variable specifying that referenced data is shared via replication with strong replica consistency (see Definition 2).

Sharing based on weak data replication. Primitives (*iii*) and (*vi*) declare `rvalw` variable for data shared based on weakly consistent replicas (in the sense of Definition 3).

A SharedX10 programmer is expected to identify the sharing pattern for each variable, which is typically not a difficult task. The programmer can then use the suitable SharedX10 mechanisms to optimize the communication needs of the data and improve application performance.

5.2 Syntax

Syntactic constructs of SharedX10 are shown in Table 3. We use `v`, `w`, `l`, `r` to range over values, `x`, `y` for variables and `f` to range over field names. A SharedX10 program runs over a finite set of places ranged over by `p`.

The construct `async s` spawns an activity to execute statement `s`.

`finish s` executes `s` and waits for the termination of all the activities spawned during the execution of `s`. Statement `at(e) s` synchronously executes `s` at place that corresponds to the evaluated expression `e`. The sequence statement `{s, t}` executes `t` after executing `s`. However, if `s` is an `async`, its execution will spawn a new activity and then activate `t`, and hence statements `s` and `t` will actually be executed in parallel. The grammar of SharedX10 uses "{" and "}" for grouping of statements.

Variable declarations `val v:T = e s` and `var x:T = e s` declare a new (immutable) variable `v` and (mutable) `x`, respectively, of type `T`, bind it to the value of expression `e` and continue as `s`. A function literal `(`x_1`:`T_1`,...,`x_n `: `T_n`):T => e` creates a function of type `(`$x_1 : T_1$`,...,`$x_n : T_n$`) => T`. For example, `(x:Long):Long => x*x` is a function literal describing the squaring function on integers.

Statement `def m(`v_1`:`T_1`, ..., `v_n`:`T_n`) s` defines a standard X10 method with a name `m`, body `s` and arguments v_i of type T_i, $i = \overline{1,n}$. SharedX10's `const` methods are declared with a `const` keyword.

Atomic statements, unconditional `atomic s` and conditional `when (e) s`, execute body `s` as if in a single step with respect to atomic blocks executed by all other activities in the same place. Conditional variant suspends the execution until the guard `e` is evaluated to `true`, furthermore the execution of the test `when (e)` is atomic with the execution of the block `s`.

The expression `GlobalRef[T](e)` creates a new global reference to the evaluation of the expression `e`. When an expression `e` evaluates to a global reference, then `e()` returns the object referenced by `e`.

SharedX10 distinguishes the between two method invocations for a variable `v`, one for the standard X10 method calls i.e., `v.m(`e_1`,...,`e_n`)` and the other for the `const` method calls i.e., `v.`$m_{\texttt{const}}$`(`e_1`,...,`e_n`)`, where $e_1, \ldots, e_n$ are arguments corresponding to the method's formal parameters.

The rest of the constructs are common language constructs, typical of object-oriented languages.

The syntax of SharedX10 presented in Table 3 omits the boilerplate part, such as statements for importing packages, main function and class

s, t	::=	s'	
		| sval v : T = e s	
		| rvals@l : ArrayList[Place] v : T = e s	
		| rvalw@l : ArrayList[Place] v : T = e s	
		| sval v : DistArray[T] = e s	
		| rvals@l : DistArray[ArrayList[Place]] v : DistArray[T] = e s	
		| rvalw@l : DistArray[ArrayList[Place]] v : DistArray[T] = e s	
s'	::=	async s	(*spawn s in a different task*)
		| finish s	(*run s and wait for termination*)
		| at(e) s	(*run s at place e*)
		| while (b) s	(*while loop*)
		| if (b) s else t	(*if − else branch*)
		| for (s; s; s) s	(*for loop*)
		| for (w in *r*) s	(*enumerator loop*)
		| def m($v_1 : T_1, \ldots, v_n : T_n$) s	(*method m with a body s*)
		| def m($v_1 : T_1, \ldots, v_n : T_n$) const s	(*const method m with a body s*)
		| return s'	(*return statement*)
		| atomic s	(*atomic statement*)
		| when (e) s	(*when statement*)
		| try s catch t	(*try s on failure execute t*)
		| val v : T = e s	(*binds e to value v in s*)
		| val v : ($x_1 : T_1, \ldots, x_n : T_n$) => T = ($x_1 : T_1, \ldots, x_n : T_n$) => e s	(*function literal binds e to value v in s*)

s′	::=	| var x : T = e s	(*let bind e to variable v in s*)
		| x = e s	(*update var x to e in s*)
		| {s t}	(*run s and then t*)
		| e	(*expression*)
b	::=	e ⋈ e	(*logical expression*)
		| ! b	(*negation*)
		| true	(*tautology*)
e	::=	e.f	(*field selecton*)
		| new T()	(*object construction*)
		| e.m($e_1, \ldots, e_n$)	(*method invocation*)
		| e.m_{const}($e_1, \ldots, e_n$)	(*const method invocation*)
		| v	(*values*)
		| x	(*variables*)
		| v(i)	(*array access*)
		| GlobalRef[T](e)	(*GlobalRef construction*)
		| e()	(*GlobalRef deconstruction*)
		| e ⊕ e	(*arithmetic expression*)

Table 3: SharedX10 syntax

declaration, that is required to change an SharedX10 program into an executable one. X10 relies on a type system to ensure that any selection operation occurring at runtime is performed on an object that actually contains the selected field. We don't provide static semantic rules, nor work with exceptions in the syntax, we simply assume correctness of programs written in SharedX10.

5.3 Encoding

In this section we introduce the transformation function

$$[\ ]_{\theta,\pi} : s \rightarrow s'$$

that takes as an argument a SharedX10 program `s`, and transforms into an X10 program `s'`.

v	$\theta(v)$	*Description*
`sval`	`u`	Sharing based on a single data instance.
`rvals`	`rs`	Sharing based on strong replication.
`rvalw`	`rw`	Sharing based on weak replication.

Table 4: Function θ

The rules for transformation use auxiliary functions θ and π, compare function and synchronization variable, i.e., tokens. Function θ when applied to a shared variable returns the sharing strategy of the shared variable. Table 4 shows the possible values for θ. Function π for a replicated variable returns a variable that contain a set of places for replicas. For example, evaluation: `[rvals@l v:T = new T(); s]`$_{\theta,\pi}$ is composed of two consecutive steps; the first step evaluates `rvals` statement i.e., `[rvals@l v:T = new T();]`$_{\theta,\pi}$, and it is followed by `[s]`$_{\theta',\pi'}$, where θ and π are updated after the first step, such that θ'(`v`) = `rs`, π'(`v`) = `l`. It is assumed that initially, at the beginning of the program transformation, it holds that domains of the functions are empty i.e., Dom(θ) = $\emptyset$ and Dom(π) = $\emptyset$.

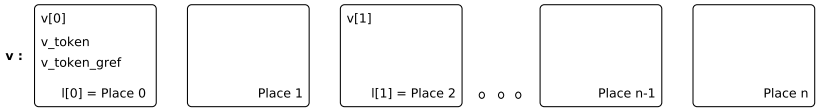


Figure 26: Schematic view of possible allocation of variables to places in Rules 2 and 3

5.3.1 Transformation Rules

The transformation rules we present below guided our prototype implementation of SharedX10 in XTEXT.

1) `[sval v:T = e; s]`$_{\theta,\pi} \rightarrow$

```
val v:T = [e]θ,π;
val v_gref:GlobalRef[T] = new GlobalRef[T](v);
[s]θ∪(v,u),π
```

Definition of a shared variable `sval v` of generic type `T`, and initialized to the value of expression `e`, is transformed into a definition of variable `val v` which has an associated global reference `v_gref`. This representation ensures that an `sval` is stored at a single place (that of allocation) with no replicas. We detail the transformation of data accesses to `sval` variable in the rule 4. In the continuation of the program transformation, θ is updated such that it maps `v` to `u`, indicating that `v` is stored at a single place, i.e., unreplicated.

2-3) `[rval(s/w)@l:ArrayList[Place] v:T = e; s]`$_{\theta,\pi} \rightarrow$

```
val v:DistArray[T] = DistArray.make[T]
(Dist.makeUnique(l), Point => [e]θ,π]);
val v_token = new Token();
val v_token_gref:GlobalRef[Token] =
new GlobalRef[Token](v_token);
[s]θ∪(v,rs(rw)),π∪(v,l)
```

To replicate (either strongly or weakly) a variable `v` of type `T` across an ar-

ray of places, `l:ArrayList[Place]`, we use X10's built-in distributed array class, `DistArray`, that represents a generic multidimensional array distributed over multiple places. There are various strategies available for initializing such array (e.g. see Section 2.2). We choose the *unique* distribution, which stores one data element (`Point`), initialized to the value of evaluated expression `e`, per place contained in array `l`. Furthermore, token `v_token` and its global reference `v_token_gref` are created to be used for synchronizing accesses to `v`.
Figure 26 illustrates allocation of variables to places for the code snippet in Listing 5.1. `Lines 1-3` are dedicated to creating list of places `l`, which in this case contains two places out of possible `n`. At `line 5` `rvals` is initialized to a new instance of a generic class `ClassG`. Consequently, `v_token` and `v_token_gref` are allocated at the same place where `rvals` statement is executed.

Listing 5.1: Example use of rvals statement

```
1  val l:ArrayList[Place] = new ArrayList[Place]();
2  l.add(Place.places(0));
3  l.add(Place.places(2));
4  rvals@l v:ClassG = new ClassG();
```

4) $[\![\texttt{v}]\!]_{\theta,\pi}$

If $\texttt{v} \notin \text{Dom}(\texttt{v})$ then apply 4a) $[\![\texttt{v}]\!]_{\theta,\pi} \rightarrow$

```
v
```

If $\theta(\texttt{v}) = \texttt{u}$ then apply 4b) $[\![\texttt{v}]\!]_{\theta,\pi} \rightarrow$

```
at(v_gref.home) v_gref()
```

If $\theta(\texttt{v}) \in \{\texttt{rs}, \texttt{rw}\}$ then apply 4c) $[\![\texttt{v}]\!]_{\theta,\pi} \rightarrow$

```
v(v.dist.get(here).maxPoint())
```

Rule 4 describes transformation of a variable reference. There are tree cases: if $\texttt{v} \notin \text{Dom}(\theta)$ then no transformation is needed since `v` is not a shared variable (Rule 4a); if $\theta(\texttt{v}) = \texttt{u}$ the reference is resolved by using the global reference `v_gref` created in rule 1 (Rule 4b); and if $\theta(\texttt{v}) \in \{\texttt{s},\texttt{w}\}$ the transformed expression targets the local replica of `v` (Rule 4c).

5) $[\texttt{v.m(}\ e_1,\ldots,e_n\texttt{)}]_{\theta,\pi}$

If $\texttt{v} \notin \text{Dom}(\texttt{v}) \vee \theta\texttt{(v)} = \texttt{u}$
then apply 5a) $[\texttt{v.m(}\ e_1,\ldots,e_n\texttt{)}]_{\theta,\pi} \rightarrow$

```
[v]θ,π.m( [e1]θ,π,...,[en]θ,π)
```

If $\theta\texttt{(v)} \in \{\texttt{rs, rw}\}$ then apply 5b) $[\texttt{v.m(}\ e_1,\ldots,e_n\texttt{)}]_{\theta,\pi} \rightarrow$

```
at(v_token_gref.home) v_token_gref().acquire();
finish for (p in π(v)) at(p) async
[v]θ,π.m( [e1]θ,π,...,[en]θ,π);
at(v_token_gref.home) v_token_gref().release();
```

Rule 5 is applicable when transforming method invocations, where `m` is a method name, e_1 to e_n are arguments and the method is invoked on variable `v`. There are two cases depending on the type of `v`. Rule 5a is applicable if `v` is not shared, or defined as `sval` (see Rule 1), while Rule 5b is applicable if `v` is shared via replication, in which case the method `m` needs to be invoked on all replicas, and such code needs to be synchronized by acquiring and releasing the synchronization token `v_token` (see Rule 2 and Listing 5.3). There are several strategies one can employ to make replicas consistent. We chose `finish` and `async` constructs to synchronize several parallel activities (one activity per replica).

6) $[\texttt{v.m}_{\texttt{const}}\texttt{(}\ e_1,\ldots,e_n\texttt{)}]_{\theta,\pi}$

If $\theta\texttt{(v)} = \texttt{rw}$ then apply 6a) $[\texttt{v.m}_{\texttt{const}}\texttt{(}\ e_1,\ldots,e_n\texttt{)}]_{\theta,\pi} \rightarrow$

```
[v]θ,π.m( [e1]θ,π,...,[en]θ,π)
```

If $\theta\texttt{(v)} = \texttt{rs}$ then apply 6b) $[\texttt{v.m}_{\texttt{const}}\texttt{(}\ e_1,\ldots,e_n\texttt{)}]_{\theta,\pi} \rightarrow$

```
at(v_token_gref.home) v_token_gref().acquire();
[v]θ,π.m( [e1]θ,π,...,[en]θ,π);
at(v_token_gref.home) v_token_gref().release();
```

Rule 6 is dedicated to `const` methods which play important role for replicated variables. According to the replica management algorithms

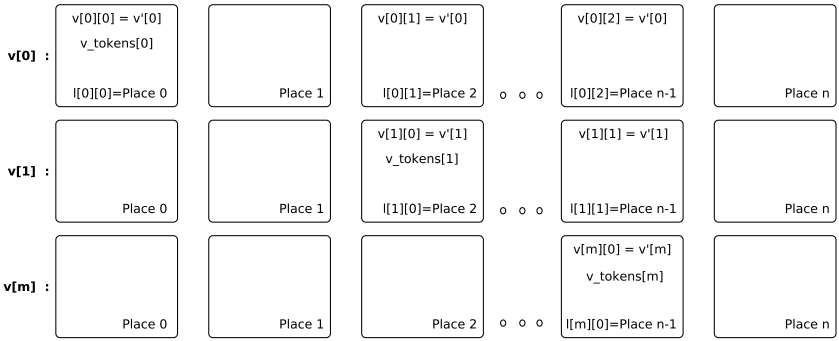


Figure 27: Generic example illustrating Rules 8-9

(see Section 3.3), when variable `v` is weakly replicated (Rule 6a) then no synchronization is required, while when `v` is strongly replicated then then the method invocation happens between token acquisition and release. In both cases, `const` method is invoked only on the local replica, obtained by transforming $[\![\mathtt{v}]\!]_{\theta,\pi}$ via Rule 4.

7) $[\![\mathtt{sval\ v:DistArray[T]\ =\ e;\ s}]\!]_{\theta,\pi} \rightarrow$

$$\left[\begin{array}{l} \mathtt{val\ v:DistArray[T]\ =}\ [\![e]\!]_{\theta,\pi}\mathtt{;} \\ [\![\mathtt{s}]\!]_{\theta\cup(\mathtt{v},\mathtt{u}),\pi} \end{array}\right]$$

Rule 7 applies when variable `sval v` is a distributed array of elements of generic type `T`. Each array element is allocated and accessed at a single place (see Rule 10b).

Figure 27 illustrates on a generic example an allocation of variables to places as a result of transformation specified in following Rules 8 and 9. From the figure one can note that each element of `v` is a distributed array `v[i]`, such that `v[i][j]` = `v'[i]` located at `l[i][j]`.

8-9) $[\![\mathtt{rval(s/w)@l:DistArray[ArrayList[Place]]\ v:DistArray[T]\ =\ v';\ s}]\!]_{\theta,\pi} \rightarrow$

```
val v:DistArray[DistArray[T]] = DistArray.make[DistArray
    ↪ [T]](v'.dist);
val v_tokens:DistArray[Token] = DistArray.make[Token](v
    ↪ '.dist);
for (p in v') at(v'.dist(p)) {
l(p).sort(cmp);
val s:Long = l(p).size();
val temp:Rail[Place] = new Rail[Place](s);
for (var j:Long = 0; j<s; i++)
temp(j) = l(p)(j);
val replicaPlaces:PlaceGroup = new SparsePlaceGroup(temp
    ↪ );
val replicaDist = Dist.makeUnique(replicaPlaces);
v_tokens(p) = new Token();
v(p) = DistArray.make[T](replicaDist, ([i]:Point(1)) =>
    ↪ [v'(i)]θ,π(p));
}
[s]θ∪(v,rs(rw)),π∪(v[0],l[0])∪···∪(v[n],l[n])
```

Rules 8 and 9 describe transformations applied when sharing of distributed array elements is based on replication. Such replication is pair-wise, i.e., each element `v(i)` is replicated across places in the corresponding array `l(i)`. Auxiliary variables `v_tokens` and `v_distributions`, both being distributed arrays with the same element distribution as `v`, are used to store synchronization tokens and distributions of replicas for each element in `v`. It is required that the type of variable `v'` used for initialization corresponds to the type of `v`, i.e., `DistArray[T]`.

Rules 10, 11 and 12 describe transformations involving elements `v(i)` and are respectively analogue to rules 4, 5 and 6. Expression `v.dist(i)` is used to refer to the place where `v(i)` is stored.

10) $[\texttt{v(i)}]_{\theta,\pi}$

If `v(i)` $\notin$ Dom(`v(i)`) then apply 10a) $[\texttt{v(i)}]_{\theta,\pi} \rightarrow$

```
v(i)
```

If θ(`v(i)`) = `u` then apply 10b) $[\texttt{v(i)}]_{\theta,\pi} \rightarrow$

```
at(v.dist(i)) v(i)
```

If θ(v(i)) $\in$ {rs, rw} then apply 10c) [v(i)]$_{\theta,\pi}$ $\rightarrow$

```
(at(v.dist(i)) v(i))(v(i).dist.get(here).maxPoint())
```

11) [v(i).m (e_1,...,e_n)]$_{\theta,\pi}$

If θ(v(i)) = u then apply 11a) [v(i).m (e_1,...,e_n)]$_{\theta,\pi}$ $\rightarrow$

```
[v(i)]θ,π.m( [e1]θ,π,...,[en]θ,π)
```

If θ(v(i)) $\in$ {rs, rw} then apply 11b) [v(i).m (e_1,...,e_n)]$_{\theta,\pi}$ $\rightarrow$

```
at(v.dist(i)) v_tokens(i).acquire();
finish for (p in π(i)) at(p) asyns
[v(i)]θ,π.m( [e1]θ,π,...,[en]θ,π);
at(v.dist(i)) v_tokens(i).release();
```

12) [v(i).m_{const} (e_1,...,e_n)]$_{\theta,\pi}$

If θ(v(i)) = rs then apply
12a) [v(i).m_{const} (e_1,...,e_n)]$_{\theta,\pi}$ $\rightarrow$

```
[v(i)]θ,π.m( [e1]θ,π,...,[en]θ,π)
```

If θ(v(i)) = rw then apply 12b) [v(i).m_{const} (e_1,...,e_n)]$_{\theta,\pi}$ $\rightarrow$

```
at(v.dist(i)) v_tokens(i).acquire();
[v(i)]θ,π.m( [e1]θ,π,...,[en]θ,π);
at(v.dist(i)) v_tokens(i).release();
```

The remaining rules, starting from the 13th until the 39th, just propagate the transformation as follows:

13) [async s]$_{\theta,\pi}$ $\rightarrow$ async [s]$_{\theta,\pi}$

14) [finish s]$_{\theta,\pi}$ $\rightarrow$ finish [s]$_{\theta,\pi}$

15) [at(e) s]$_{\theta,\pi}$ $\rightarrow$ at([e]$_{\theta,\pi}$) [e]$_{\theta,\pi}$

16) $[\texttt{while (b) s}]_{\theta,\pi} \rightarrow \texttt{while } [\texttt{b}]_{\theta,\pi}\ [\texttt{s}]_{\theta,\pi}$

17) $[\texttt{if (b) s else t}]_{\theta,\pi} \rightarrow \texttt{while } [\texttt{b}]_{\theta,\pi}\ [\texttt{s}]_{\theta,\pi}$

18) $[\texttt{for (s; s; s) s}]_{\theta,\pi} \rightarrow \texttt{for } ([\texttt{s}]_{\theta,\pi}\texttt{;}[\texttt{s}]_{\theta,\pi}\texttt{;}[\texttt{s}]_{\theta,\pi})[\texttt{s}]_{\theta,\pi}$

19) $[\texttt{for (w in r) s}]_{\theta,\pi} \rightarrow \texttt{for (w in r) } [\texttt{s}]_{\theta,\pi}$

20) $[\texttt{def m(}\ \texttt{v}_1\texttt{:T}_1\texttt{,...,v}_n\texttt{:T}_n\texttt{)}\ \texttt{s}]_{\theta,\pi} \rightarrow$

$\texttt{def m(}\ \texttt{v}_1\texttt{:T}_1\texttt{,...,v}_n\texttt{:T}_n\texttt{)}\ [\texttt{s}]_{\theta,\pi}$

21) $[\texttt{def m(}\ \texttt{v}_1\texttt{:T}_1\texttt{,...,v}_n\texttt{:T}_n\texttt{)}\ \texttt{const s}]_{\theta,\pi} \rightarrow$

$\texttt{def m(}\ \texttt{v}_1\texttt{:T}_1\texttt{,...,v}_n\texttt{:T}_n\texttt{)}\ \texttt{const}\ [\texttt{s}]_{\theta,\pi}$

22) $\texttt{return } [\texttt{s}]_{\theta,\pi} \rightarrow \texttt{return } [\texttt{s}]_{\theta,\pi}$

23) $\texttt{atomic } [\texttt{s}]_{\theta,\pi} \rightarrow \texttt{atomic } [\texttt{s}]_{\theta,\pi}$

24) $[\texttt{when (e) s}]_{\theta,\pi} \rightarrow \texttt{when (}\ [\texttt{e}]_{\theta,\pi}\texttt{)}\ [\texttt{s}]_{\theta,\pi}$

25) $[\texttt{try s catch t}]_{\theta,\pi} \rightarrow \texttt{try } [\texttt{s}]_{\theta,\pi}\ \texttt{catch } [\texttt{t}]_{\theta,\pi}$

26) $[\texttt{val v:T = e; s}]_{\theta,\pi} \rightarrow \texttt{val v:T = } [\texttt{e}]_{\theta,\pi}\texttt{;}[\texttt{s}]_{\theta,\pi}$

27) $[\texttt{val v:(}\texttt{x}_1\texttt{:T}_1\texttt{,...,x}_n\texttt{:T}_n\texttt{) => T}$
$\texttt{= (}\texttt{x}_1\texttt{:T}_1\texttt{,...,x}_n\texttt{:T}_n\texttt{) => e s}]_{\theta,\pi} \rightarrow$
$\texttt{val v:(}\texttt{x}_1\texttt{:T}_1\texttt{,...,x}_n\texttt{:T}_n\texttt{) => T}$
$\texttt{= (}\texttt{x}_1\texttt{:T}_1\texttt{,...,x}_n\texttt{:T}_n\texttt{) => } [\texttt{e}]_{\theta,\pi}[\texttt{s}]_{\theta,\pi}$

28) $[\texttt{var x:T = e; s}]_{\theta,\pi} \rightarrow \texttt{var x:T = } [\texttt{e}]_{\theta,\pi}\texttt{;}[\texttt{s}]_{\theta,\pi}$

29) $[\texttt{x = e; s}]_{\theta,\pi} \rightarrow \texttt{x = } [\texttt{e}]_{\theta,\pi}\texttt{;}[\texttt{s}]_{\theta,\pi}$

30) $[\{\texttt{s; t}\}]_{\theta,\pi} \rightarrow \{[\texttt{s}]_{\theta,\pi}\texttt{;}[\texttt{t}]_{\theta',\pi'}\}$

31) $[\texttt{e} \bowtie \texttt{e}]_{\theta,\pi} \rightarrow [\texttt{e}]_{\theta,\pi} \bowtie [\texttt{e}]_{\theta,\pi}$

32) $[\texttt{!b}]_{\theta,\pi} \rightarrow \texttt{![b]}_{\theta,\pi}$

33) $[\texttt{true}]_{\theta,\pi} \rightarrow \texttt{true}$

34) $[\texttt{e.f}]_{\theta,\pi} \rightarrow [\texttt{e}]_{\theta,\pi}\texttt{.f}$

35) $[\texttt{new T()}]_{\theta,\pi} \rightarrow \texttt{new T()}$

36) $[\texttt{x}]_{\theta,\pi} \rightarrow \texttt{x}$

37) $[\texttt{GlobalRef[T](e)}]_{\theta,\pi} \rightarrow \texttt{GlobalRef[T}([\texttt{e}]_{\theta,\pi})$

38) $[\texttt{e()}]_{\theta,\pi} \rightarrow [\texttt{e}]_{\theta,\pi}\texttt{()}$

39) $[\texttt{e} \oplus \texttt{e}]_{\theta,\pi} \rightarrow [\texttt{e}]_{\theta,\pi} \oplus [\texttt{e}]_{\theta,\pi}$

The function literal `cmp` is used in the rule number 8 for sorting an array of places, its specification is shown in the Listing 5.3.

Listing 5.2: Compare function

```
val cmp : (p1:Place, p2:Place) => Int
  = (p1:Place, p2:Place) => {
  if (p1.id() > p2.id())
    return Int.operator_as(1);
  else if (p1.id() < p2.id())
    return Int.operator_as(-1);
  else return Int.operator_as(0);
};
```

Accesses to replicated variables is guarded by the synchronization variables which are instances of the Token class. The definition of the Token class is shown in the Listing 5.3. The class contains one integer (`Long`) field named `token` and two methods for acquiring (`acquire()`) and releasing (`release()`) hold of the synchronization token.

Listing 5.3: Token class

```
public class Token {
  var token:Long;
  def this() {
    token = 0;
  }
```

```
  def acquire() {
    when(token == 0) token = 1;
  }
  def release() {
    atomic token = 0;
  }
}
```

Example

Listing 5.4 shows a simple SharedX10 program snippet which will serve to illustrate how some of the transformation rules work. The main idea of the snippet is to define a distributed object which would represent a graph and, based on it, to define a shared graph that replicates its nodes such that neighboring nodes are co-located.

`Lines 1-2` define a variable `graph` as a distributed array whose elements will be block distributed across available places (see block distribution in Section 2.2). `Lines 3-5` define a distributed array `allPlaces` which will be instrumental in defining shared graph `sharedGraph` (`Line 7`), hence it should store localities (places) in arrays of places (`ArrayList[Place]`). For instance, places of neighbors of node `graph(p)` would be stored at `allPlaces(p)`. Furthermore, its distribution should correspond to that of `graph`, i.e., `dist`. For conciseness, we leave out code that stores information in `graph` and `allPlaces` based on some input data (`Line 6`). Final line simply stores degree of node at position `0` to a local variable `degree`.

Listing 5.4: SharedX10 program snippet

```
1 val dist = Dist.makeBlock(Region.make(0, size-1));
2 val graph:DistArray[GraphNode] = DistArray.make[GraphNode](
    ↪ dist, (Point) => new GraphNode());
3 val allPlaces:DistArray[ArrayList[Place]] = DistArray.make[
    ↪ ArrayList[Place]](dist);
4 for (nodeId in allPlaces) at(dist(nodeId))
5   allPlaces(nodeId) = new ArrayList[Place]();
6 /* Code that instantiate graph and allPlaces */
7 rvalw@allPlaces sharedGraph:DistArray[GraphNode] = graph;
8 val degree:Long = sharedGraph(0).getDegree();
```

Listing 5.5 shows an X10 program obtained after applying the transformation rules (`Rules 8` and `10`) on program snippet in Listing 5.4. `Lines` in red `7-19` and `Line 20` show replacements for `Lines 7` and `8` in the SharedX10 program snippet.

Listing 5.5: Transformed program snippet

```
1  val dist = Dist.makeBlock(Region.make(0, size-1));
2  val graph:DistArray[GraphNode] = DistArray.make[GraphNode](
       ↪ dist, (Point) => new GraphNode());
3  val allPlaces:DistArray[ArrayList[Place]] = DistArray.make[
       ↪ ArrayList[Place]](dist);
4  for (nodeId in allPlaces) at(dist(nodeId))
5    allPlaces(nodeId) = new ArrayList[Place]();
6  /* Code that instantiate graph and allPlaces */
7  val sharedGraph:DistArray[DistArray[GraphNode]] =
       DistArray.make[DistArray[GraphNode]](graph.dist);
8  val sharedGraph_tokens:DistArray[Token] =
       DistArray.make[Token](graph.dist);
9  for (p in graph) at (graph.dist(p)) {
10   allPlaces(p).sort(cmp);
11   val s:Long = allPlaces(p).size();
12   val temp:Rail[Place] = new Rail[Place](s);
13   for (var j:Long = 0; j < s; j++)
14     temp(j) = allPlaces(p)(j);
15   val replicaPlaces:PlaceGroup = new
         SparsePlaceGroup(temp);
16   val replicaDist = Dist.makeUnique(replicaPlaces);
17   sharedGraph_tokens(p) = new Token();
18   sharedGraph(p) = DistArray.make[GraphNode](replicaDist,
         ([i]:Point(1)) => graph(p));
19 }
20 val degree:Long = (sharedGraph(0)(sharedGraph(0).dist
       ↪ .get(here).maxPoint())).getDegree();
```

In the transformed program, `sharedGraph` is represented as a distributed array, such that each array element is yet another distributed array (`Line 7`). The distribution of the main distributed array corresponds to that of `graph`. `Line 8` creates a distributed array which stores tokens for synchronization.

`for` loop in `Lines 9-19` is responsible for creating a distributed array of replicas for each graph node. Essentially, a distributed array of replicas of node `graph(p)` is stored at `sharedGraph(p)`, according to distribution recorded in `replicaDist`, at `line 18`. In particular, at

each place contained in `replicaDist`, one replica of the current node is created. Leading `Lines 10–16` perform several necessary and preparatory computations. First, the set of places for replicas is sorted (`line 10`), then copied to an auxiliary data structure (`Rail`), at `lines 12–14`, which is used to form a `PlaceGroup` (`line 15`), and finally used to create a distribution `replicaDist`. `Line 17` creates an instance of `Token` class for synchronizing accesses to replicated node `graph(p)` and assigns it to the dedicated array.

As a result, `sharedGraph` is a two-dimensional distributed array, such that element at position `p`, `sharedGraph(p)`, is a distributed array of replicas of `graph(p)`. Finally, at `line 20`, a local replica of node at position `0` is accessed to retrieve its degree. Expression `sharedGraph(0).dist.get(here).maxPoint()` computes the position of (local) replica at the place of execution (`here`) as it first finds indexes of all points in replica distribution mapped to `here`, and then takes the maximum one, which is also the unique, as distribution maps only one point (replica) per place (`line 16`).

5.4 Implementation

It is worth to mention that our initial goal was to implement primitives for data sharing as X10 annotations. Program transformation associated with annotations would then be applied by a pre-processor during the pre-processing phase of program compilation. However, we were unable to apply this approach as X10 compiler infrastructure was under development at the time and no documentation for annotations was available. Our following choice was to implement SharedX10 as a DSL using the XTEXT framework. The implementation encompasses grammar specification, routines for scoping and type checking and a specification of a code generator that produces X10 code for an input SharedX10 program.

Listing 5.6 shows a snippet of main SharedX10 syntactic rules in terms of XTEXT grammar.

Listing 5.6: SharedX10 grammar snippet

```
1 Program:
2 class += Class*
3 ;
4 Class:
5 'public' 'class' name = ID
```

```
6  ('extends' superclass=[Class])?
7  '{' members += Member* '}'
8  ;
9  Member:
10 Method | VariableDef | SharedVariableDef
11 ;
12 Method:
13 'def' name = (ID | 'this') '(' (params += Parameter
14 (',' params += Parameter)*)? ')' (isconst ?= 'const')?
15 body = Body
16 ;
17 Parameter:
18 name = ID (istyped ?= ':' type = VariableType)?
19 ;
20 Body:
21 '{' statements += Statement* '}'
22 ;
23 VariableType:
24 type = [Class]
25 (isarray ?= '[' innerType = VariableType ']' )?
26 ;
27 VariableDef:
28 vartype = ('var' | 'val')
29 name = ID (istyped ?= ':' type = VariableType)?
30 (isinit ?= '=' expression = Expression)? ';'
31 ;
32 SharedVariableDef:
33 'sval' name = ID ':' type = VariableType '=' expression =
34 Expression ';' |
35 'rvals' name = ID '@' places = [VariableDef] ':' type =
36 VariableType '=' expression = Expression ';' |
37 'rvalw' name = ID '@' places = [VariableDef] ':' type =
38 VariableType '=' expression = Expression ';'
39 ;
```

Starting rule, `Program`, indicates that the root element of program syntax tree is a collection of `Class` objects, which are in turn collections of `Member` objects. Class members are methods, standard X10 variable definitions and shared variable definitions, modeled with rules `Method`, `VariableDef` and `SharedVariableDef`. Methods begin with a keyword `'def'` followed by a method name (`this` for a constructor), fol-

lowed by a list of method parameters and a body. If the method is a `const`, then the value of feature `isconst` in corresponding node of the syntax tree will be *true*, otherwise *false*. Rule `Parameter` indicates that a parameter has a name and that it can be typed. `VariableType` defines variable type through cross-reference mechanism. Standard variable definition contains a keyword `val` or `var`, a name, and it is possibly typed and initialized to a expression. Shared variable definition contains cross-references for the feature `places` and it is expected to be initialized to an expression.

The grammar snippet omits the standard rules for expressions and statements. The complete grammar specification can be found in Appendix B.

5.5 Performance Evaluation

In this section we describe the practical experiments that were performed in X10 on a synthetic benchmark application, in order to measure the impact on performance that shared data accesses can have.

The model behind the benchmark application is based on intensive parallel data accesses. Figure 5.7 shows a pseudo-code specification of the data access function. Each access to data is done by a separate activity, that is spawned in a loop (`line 1`). The number of concurrently running activities can be up to some pre-defined `NUM_AC` number. Furthermore, each activity can perform either an update or a read access on a shared variable `v`, with a pre-defined probability `p`.

We tuned parameters `p` and `NUM_AC` to compare program performances with respect to different ratios of read/update access, levels of concurrency, as well as size of accessed data.

Listing 5.7: Data access function

```
1 for (var i:Long = 0; i < NUM_AC; i++) async {
2   with probability p { // update
3     v.update();
4   }
5   with probability 1-p { // read
6     val temp = v.getData();
7   }
8 }
```

Hypothesis. The main motivation behind the experiments is to demonstrate that better data locality and minimized communication can be achieved by replicating data in X10. In a classical, non-replicated scenario, local read access is granted only to activities residing at the same place of the data. Remote read access to data involves network data transfer cost, which is not negligible, and increases with the size of accessed data, as we experimentally confirm. Data replication can be seen as an optimization that can remedy this problem. However, replications calls for consistency protocols, that introduce the costs of performing the same update access on each replica. We performed a set of experiments that provide indications about the situations when such optimization is beneficial and the level of impact it can have on performance. Our experimental results show how the ratio between frequencies of updates and reads, the degree of concurrent data accesses and the size of data affects the performance of two different versions of a program: a *standard* one that does not use replicas (program **no-replicas**) and the one with replicas (program **replicas**).

Program **replicas**: In this variant, the shared data is replicated at each place. Presence of replicas call for the use of consistency protocols. In these implementations the level of consistency for replicated data is weak i.e., sequential consistency. According to the definition of SC presented in the Section 3.3, this means that the interleaving of actions is allowed as update of replicas does not happen instantaneously across all the places. Particularly, when one replica is updated at a certain place, multiple activities update in parallel remaining replicas in non-atomic way. During this process, local reads can occur at remote places, before all replicas have the same values. Interleaving of two or more update operations is prevented by the synchronization operations.

Listing 5.8: Program replicas in SharedX10

```
1  val places = Place.places();
2  rvalw@places a = new A();
3  for (place in places) at(place) async {
4    for (var i:Long = 0; i < NUM_AC; i++) async {
5      with probability p { // update
6        a.update();
7      }
8      with probability 1-p { // read
9        val temp = a.getData();
10     }
```

```
11    }
12  }
```

The SharedX10 code snippet (Listing 5.8) presents a pseudo-code specification of the program **replicas** in SharedX10. Variable `a`, an instance of class `A`, is shared via weak replication across all available places (`lines 1-2`). All places perform the same kind of access to the data in parallel (specified in Listing 5.7) (`lines 3-12`).

Following Listing 5.9 shows the same program specified in X10, obtained by applying the program transformation described in Section 5.3. The same code snippet also demonstrates the programmer's effort in implementing replicated shared variable in X10.

Listing 5.9: Program replicas in X10

```
1  val places = Place.places();
2  val a:DistArray[A] = DistArray.make[A](Dist.makeUnique(
       ↪ places), Point => new A());
3  val a_token = new Token();
4  val a_token_gref:GlobalRef[Token] = new GlobalRef[Token](
       ↪ a_token);
5  for (place in places) at(place) async {
6    for (var i:Long = 0; i < NUM_AC; i++) async {
7      with probability p { // update
8        at(a_token_gref.home) a_token_gref().acquire();
9        finish for (p in places) at(p) async
10         a(a.dist.get(here).maxPoint()).update();
11       at(a_token_gref.home) a_token_gref().release();
12     }
13     with probability 1-p { // read
14       val temp = a(a.dist.get(here).maxPoint()).getData();
15     }
16   }
17 }
```

Program **no-replicas**: the implementations of these programs are based on the standard approach that does not involve replication of shared data. The basic idea is that shared data is stored at a single place, with no replicas. Local access to the shared data is granted only to activities running at that place, while other accesses are done remotely, via global-references.

Listing 5.10 shows a specification of pseudo-code of program **no-replicas** in SharedX10. The only difference to the program **replicas** shown in the Listing 5.8 is that variable `a` is declared as `sval`.

Listing 5.10: Program no-replicas in SharedX10

```
1  val places = Place.places();
2  sval@places a = new A();
3  for (place in places) at(place) async {
4    for (var i:Long = 0; i < NUM_AC; i++) async {
5      with probability p { // update
6        v.update();
7      }
8      with probability 1-p { // read
9        val temp = v.getData();
10     }
11   }
12 }
```

Listing 5.11 shows the same program specification in X10 obtained by the program transformation.

Listing 5.11: Program no-replicas in X10

```
1  val places = Place.places();
2  val a:A = new A();
3  val a_gref:GlobalRef[A] = new GlobalRef[A](a);
4  for (place in places) at(place) async {
5    for (var i:Long = 0; i < NUM_AC; i++) async {
6      with probability p { // update
7        at(a_gref.home) a_gref().update();
8      }
9      with probability 1-p { // read
10       val temp = at(a_gref.home) a_gref().getData();
11     }
12   }
13 }
```

The source code of the program specifications is available for download at below link[1].

[1] http://sysma.imtlucca.it/wp-content/uploads/2015/05/Source_X10_example.rar

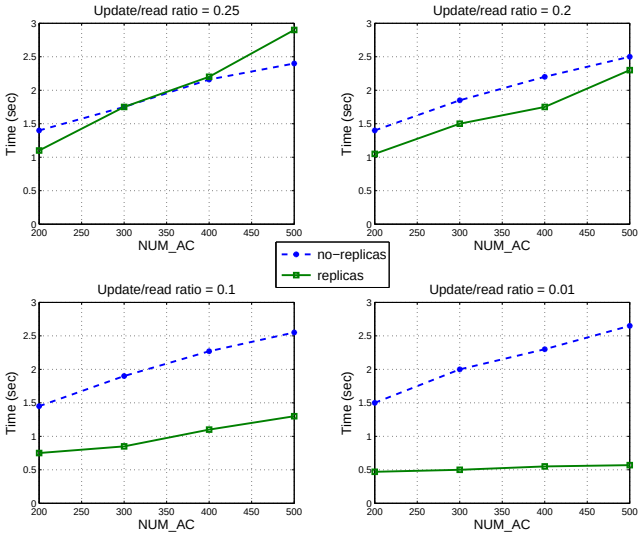


Figure 28: (`Ratio`): The two strategies with shared data of size ≈ 0.4`MB`

Experiments: Configuration of the Scenario. We compare performances of two variants which we refer to as **no-replicas** and **replicas**. The essence of the program with replicas has been introduced through Listings 5.8 and 5.9. In contrast to the replicated variant, the non-replicated one excludes creation of replicas, hence every access is directed towards a single centralized data variable, as promoted in Listings 5.10 and 5.11.

To obtain more elaborate results we tune three parameters in our implementations:

- The ratio of update/read rates;
- The number of shared data accesses per place `NUM_AC`; and
- The size of shared data.

Update and read rates are used to compute the probability `p` with which update can happen inside our `dataAccess` function, and it is

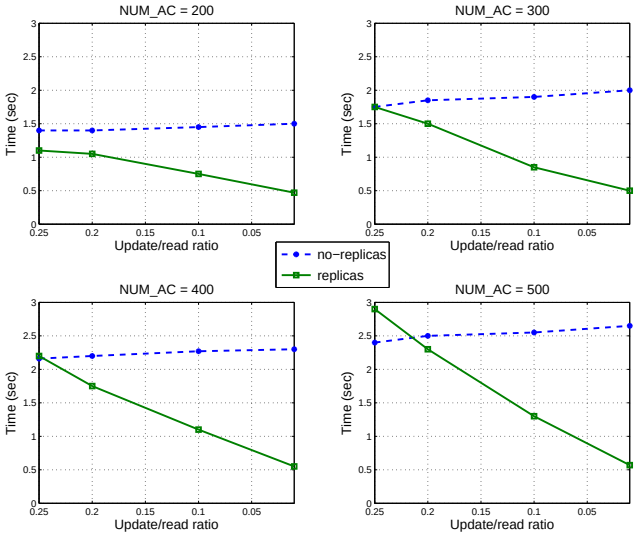


Figure 29: (`Access number`): The two strategies with shared data of size $\approx$ 0.4MB

calculated by the formula:

```
p = update_rate/(update_rate + read_rate)
```

For calculating `p` we use the following pairs of `update` and `read` rates: {(1, 100), (1, 10), (1, 5), (1, 4)}. The number `NUM_AC` is a number of data accesses/concurrently spawned activities per place and takes values 200, 300, 400 and 500. As an example, if the update/read ratio is $1/5$ and `NUM_AC` is 400, it means that there are approximately 80 update and 320 read accesses to shared data per place.

Finally, the size of shared data in one case of our experiments is $\approx$0.4MB and $\approx$4MB in the other.

For evaluating our test examples, we used the X10 compiler targeting the Java backend (a.k.a. the Managed X10), version X10-2.5.0-linux/x86_64 on OS Ubuntu 14.4.

All results are obtained on hardware with 2 processors `Intel(R) Xeon(R) CPU E5620` @ 2.40GHz, each one with 4 cores and 2 threads per core, with 40GB of `RAM`.

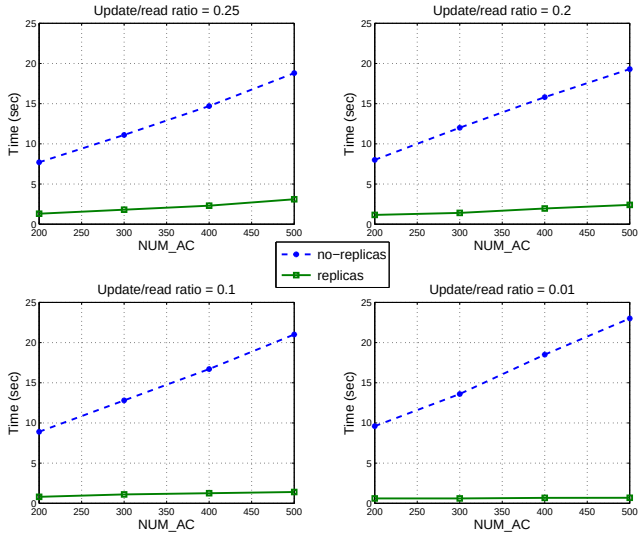


Figure 30: (Ratio): The two strategies with shared data of size ≈ 4MB

Experiments: Data and Interpretation. The results of our experiments are given in terms of dependencies between the ratio of updates and reads performed by all activities (Fig. 28, 30) or the number of accesses NUM_AC (Fig. 29, 31), represented on x axis, and time taken by activities to complete their computations, on y axis. Time is expressed in seconds and it is obtained as the average of 10 executions. Fig. 28 and 29 correspond to results obtained for size of shared data of ≈ 0.4MB, while Fig. 30 and 31 correspond to results obtained for the size of ≈ 4MB.

Figs. 28 to 31 are obtained for 4 places. The results obtained for 8 places can be found in the Appendix A.

From the presented results we can conclude that the performance benefit of replication tends to grow with the increasing number of total accesses and decreasing update/read ratio. Furthermore, the greater the size of shared data, the more desirable it is to replicate it.

As it can be seen from the figures, preserving consistency across many replicas can be expensive. However, replication still brings good pays off when the size of data is either large enough (Fig A.38(c)) or the up-

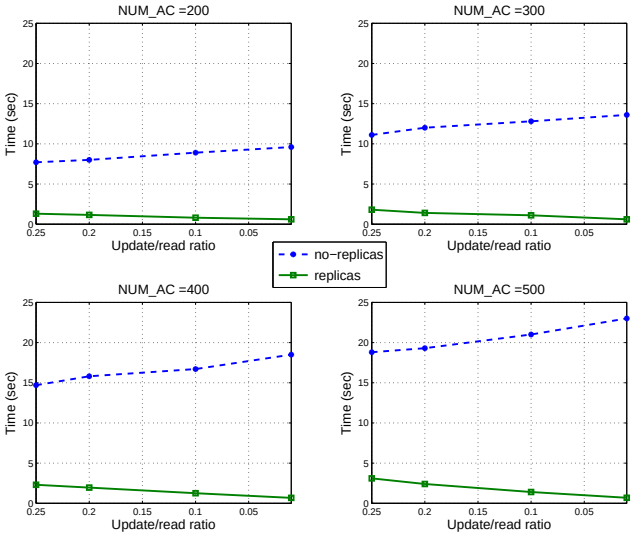


Figure 31: (`Access number`): The two strategies with shared data of size ≈ 4MB

date/read ration is small enough (Fig. A.38(a)).

Our initial attempts to scale the experiments to 16 places failed at runtime with a "`Place(0):TOO MANY THREADS`" error. We found that a similar issue with X10 was reported in (IS14). By reconsidering our experiments, we came to the conclusion that the problem was mainly due to a centralized lock variable and to the large number (more than a thousand) of activities competing simultaneously for it. This should have created congestion at home place of the lock, i.e. at place 0. Initially, we did aim for high parallelism and implemented each access to shared data as a separate activity, by using the `async` feature (see listing 5.7 `line 1`). Alternatively, one could dedicate a smaller number of activities to handle those accesses. Indeed, update accesses are atomic and hence could be sequentialized rather than parallelized. Conversely, read accesses can be interleaved and therefore should be parallelized in order to achieve high performance. To reach this goal, one has to take into account certain limitations posed on the maximum number of activities and the amount of memory dedicated to the program.

5.6 Summary and Related Work

In this chapter we presented the SharedX10 language which comprises of a subset of X10 enriched with primitives to allow the programmer to specify data sharing following the application sharing pattern. Such sharing is based on two strategies: 1) replication, with two levels of replica consistency, and 2) centralized data instance. In particular, the chapter presented language syntax, formalization of encoding of SharedX10 into X10, the implementation in terms of XTEXT grammar and experimental results. The evaluations were based on a synthetic benchmark application, which show how the data sharing strategy impacts the application performance in term of the size of shared body of data, frequency and ration of read and write accesses.

To our knowledge, the most closely related work on X10 to ours is presented in (PTA14). The main idea is improving performance of X10 programs by employing compiler and runtime in deciding which sharing strategy should be used for each variable. In this approach, no hints from the programmer are required, however, profiling activities by compiler and runtime as reported incur significant performance cost in some cases. We believe that using primitives for data sharing, such as the ones we propose, supported by the X10 compiler, could lead to optimal performances.

Chapter 6

Case Studies

The focus of this chapter is on evaluation of the proposed approach in terms of performance and programmability. To this end, we experimented with X10, SharedX10, Klaim and RepliKlaim using two case studies from the large-scale graph analytics. In particular, we considered two graph algorithms; one which computes the maximum node degree and the other that computes PageRank values. Both algorithms are iterative, in which computations are realized conceptually in parallel, however, they differ in terms of required level of synchronization between parallel processes. This chapter presents several program specifications and experimental results.

Structure of the chapter

Section 6.1 is a preliminary, while Sections 6.2 and 6.3 introduce the two case studies with specifications in X10, SharedX10, Klaim and RepliKlaim. Section 6.4 describes the performance evaluation setup and the discusses results. Finally, in Section 6.5 provides a brief summary with a reference to related work.

6.1 Preliminaries

Algorithms presented in the subsequent sections operate on data representing graph input from an external resource (i.e., a textual file) and distributed across a number of localities (see Figure 32). In terms of X10 and

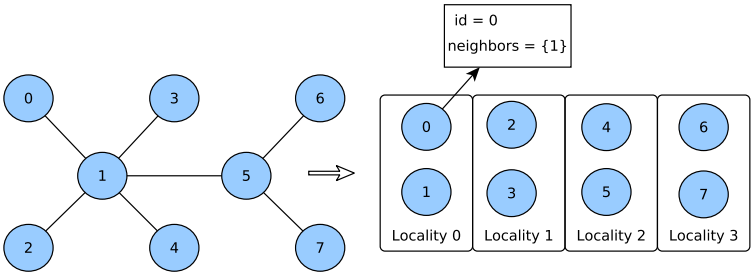


Figure 32: A graph representation based on block distribution across four localities

SharedX10 one such locality corresponds to the notion of place, while in Klaim and RepliKlaim's terms a locality corresponds to a network node. Each node in a graph is at least characterized by an integer id and a list of ids of neighboring nodes. Distribution of nodes across localities is organized into blocks of approximately the same size, which would be deployed at run-time on homogeneous cluster nodes, equal in terms of storage and computational power. X10 and SharedX10 feature a distributed array (`DistArray`, see Section 2.2) which facilitates spreading blocks of globally shared data across places. In the corresponding algorithm specifications, the `graph` variable refers to such distributed array, while the expression `graph.dist(nodeId)` is used to obtain the place of allocation of the node with id `nodeId`. In Klaim and RepliKlaim specifications, a graph is represented as a collection of tuples whose block distribution is achieved by placing a tuple t, associated to node with id `nodeId`, to a network component whose locality corresponds to `nodeId%n`, where `n` is the total number of components (see Section 2.1). Moreover, we use notation $l_{h(nodeId)}$ to refer to locality of such tuple t.

For each case study we consider two approaches in specifying locality of graph data; the traditional one with no replicas and one based on replicas. Figure 32 provides an illustration of an undirected graph distributed across four localities. Figure 33 shows the same graph distributed across four localities such that some nodes are replicated so that neighboring nodes are co-located. The choice of distribution may have a significant impact on application performance, as it is experimentally shown in this chapter. Moreover, both case studies employ weak level of replica consistency i.e., the sequential consistency, which provides sufficient guar-

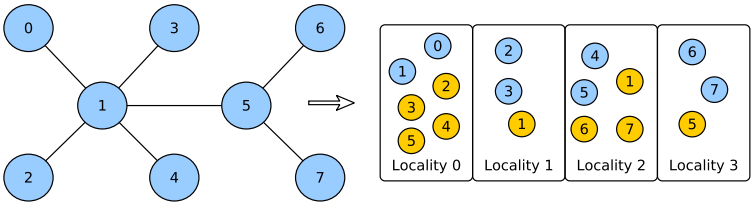


Figure 33: A graph representation based on block distribution with replicas across four localities

antees in both cases.

6.2 Maximum Graph Degree

The algorithm presented here computes the maximum degree of an undirected graph in an iterative computation. At each step of the algorithm, each node degree is updated to the maximum value of degrees of its neighbor nodes. The algorithm finishes when the number of iterations is equal to graph diameter, at which point the maximum degree is propagated to each node. The algorithm requires all processes to proceed at the same speed, meaning that the synchronization step is required at each iteration.

6.2.1 X10 and SharedX10 Specifications

The following code listings use constructs: `clocked finish`, `clocked async` and `Clock.advanceAll()` to achieve synchronous execution (e.g., as exemplified in Section 2.2). Methods `getDegree()`, `setDegree()` and `getNeighbors()` are invoked on a node object to respectively retrieve/set node degree and retrieve a list of neighbor node ids.

The first code snippet shown in Listing 6.1 presents a *naive* implementation of the algorithm in X10.

Listing 6.1: Case study I in X10 (1)

```
1 clocked finish for (nodeId in graph) clocked async
      ↪ at(graph.dist(nodeId))
2   for (var i:Long = 0; i < diameter; i = i + 1) {
```

```
3     for (neighId in graph(nodeId).getNeighbors()) {
4       if (graph(nodeId).getDegree() <
            ↪ at(graph.dist(neighId))
            ↪ graph(neighId).getDegree())
5         graph(nodeId).setDegree(at(graph.dist(neighId))
              ↪ graph(neighId).getDegree());
6     }
7     Clock.advanceAll();
8   }
```

Line 1 spawns an activity for each node in the graph, to execute at the place the node is allocated to, which computes the main iterative computation (lines 2–8) such that all spawned activities synchronize at the end of each iterative step (line 7). One could rightfully remark that spawning an activity per graph node could be impractical in real implementations, in fact, we address this point below in section dedicated to the experimental evaluations. In each iteration (lines 3–6), an activity iterates through neighbor nodes of the local node, compares their degrees with the local node degree, and updates its value if greater is found.

The highlighted code performs the place-shifting operation which temporarily suspends the executing activity, until the body of the operations is executed at a remote place; in this case the activity is suspended until remote data is retrieved. We remark that omitting `at(graph.dist(neighId))` would cause a run-time error when an activity attempts to access array data which is not co-located with it. Realization of the highlighted operation requires the undesirable inter-place communication. In fact, in X10, such communication is made transparent to the programmer, to prompt the programmer to optimize his code by reducing remote communication. We hence refer to the implementation as naive as obviously the same data is retrieved twice by each activity.

Listing 6.2 shows a more carefully written specification in which a local variable `neighDegree` (line 4) is used to store the value of neighbor node degree and hence reduce the number of times each activity is place-shifted to half as many.

Listing 6.2: Case study I in X10 (2)

```
1 clocked finish for (nodeId in graph) clocked async
      ↪ at(graph.dist(nodeId))
2   for (var i:Long = 0; i < diameter; i = i + 1) {
```

```
3     for (neighId in graph(nodeId).getNeighbors()) {
4       val neighDegree = at(graph.dist(neighId))
            graph(neighId).getDegree();
5       if (graph(nodeId).getDegree() < neighDegree)
6         graph(nodeId).setDegree(neighDegree);
7     }
8     Clock.advanceAll();
9   }
```

Since each node object is involved in computations which realize at two or more places, it can be considered as shared data object. Furthermore, it is easy to conclude that the shared data access pattern corresponds to stencil (see Section 5.1).

We further present two specifications in SharedX10, of the same algorithm, representing two different approaches to specifying shared data locality.

The first approach, shown in Listing 6.3, is based on the idea that each node object is to be stored at a single place, or it may not be even relevant to the programmer, who simply wishes to declared data as shared and not having to consider data communication in the program.

`sharedGraph` in below listing is defined as

```
sval sharedGraph:DistArray[GraphNode] = graph
```

where `graph` object is constructed from an input file. `sharedGraph` shares the same node distribution as `graph`, illustrated in Figure 32.

While place-shifting is not visible in this specification, it is implied by the above `sharedGraph` definition that all remote data accesses will be realized by inter-place communication (e.g. see Rule 10b in Section 5.3).

Listing 6.3: SharedX10 implementation (1)

```
1  finish for (nodeId in sharedGraph) clocked async
       ↪ at(sharedGraph.dist(nodeId))
2    for (var i:Long = 0; i < diameter; i = i + 1) {
3      for (neighId in nodes(nodeId).neighbors) {
4        if (sharedGraph(nodeId).getDegree() < sharedGraph(
             ↪ neighId).getDegree())
5          sharedGraph(nodeId).setDegree(sharedGraph(neighId).
               ↪ getDegree());
6      }
7      Clock.advanceAll();
8    }
```

While it is clear that this approach will not lead to an efficient program, for the same arguments used for Listing 6.1, the proportionate advantage is gained on the side of programmability. As one may observe, data communication is not visible, which led to a simpler and more straightforward specification.

An alternative approach, based on the bulk synchronous parallel (BSP) model (Val90), is presented in Listing 6.4. The BSP model organizes parallel computation in a sequence of steps separated by a synchronization barrier. Each step consists of *computation phase* and *communication phase*. In computation phase only *local variables* can be accessed (and locally held copies of remote variables), while in communication phase data are exchanged between parallel processes.

To make the computation part of the main iterative step execute locally, it is required that neighbor nodes are collocated, as illustrated in Figure 33. Hence, implementing the BSP-style of computation calls for node replication. The style of the computation described requires that each processor must complete its iterations and communicate the results to other processors before a new iteration can begin. This can be achieved with sequential consistency of replicas. In fact, sequential consistency of replicated variable enables the programmer to apply the same programming logic used for programming common variable accesses (see Section 3.2), additionally having in mind that frequent write accesses should be optimized, i.e., intermediate results should be saved locally and only final results should be propagated to all replicas. Furthermore, methods `getDegree()` and `getNeighbors()`, invoked on `GraphNode` objects, need to be annotated as `const` as they do not modify the state of replicated node objects.

As a result of the above analysis, `sharedGraph` used in following code listing is defined as

```
rvalw@l sharedGraph:DistArray[GraphNode] = graph
```

where `l` is a distributed array of lists of places which shares the same distribution with `graph`, and it is constructed during of the graph inputing phase (complete program specification is available in Appendix C.1.2).

Listing 6.4: SharedX10 implementation (2)

```
1  clocked finish for (nodeId in sharedGraph) clocked async
       ↪ at(sharedGraph.dist(nodeId))
2    for (var i:Long = 0; i < diameter; i = i + 1) {
3      val maxDegree:Long = sharedGraph(nodeId).getDegree();
4      for (neighId in sharedGraph(nodeId).neighbors)
5        if(sharedGraph(neighId).getDegree() > maxDegree)
6          maxDegree = sharedGraph(neighId).getDegree();
7      if(sharedGraph(nodeId).getDegree() != maxDegree)
8          sharedGraph(nodeId).setDegree(maxDegree);
9      Clock.advanceAll();
10   }
```

`Line 1` spawns `clocked` activities and distributes them across places where graph nodes are allocated. The computation phase of the algorithm is realized at `lines 3-6`. `line 3` defines a local variable `maxDegree` to store the computed maximum value of neighbor node degrees. In communication phase, `lines 7-8`, `maxDegree` is propagated to node replicas if it is greater than the current degree value.

Localizing array accesses reduces data movements from remote places and can hence have a beneficial affect on the application performance. In fact, we carried out experiments based on above specifications and real-world large graph datasets and report the results in Section 6.4.

The complete programs are available in Appendix C; X10 program C.1.1 and SharedX10 program C.1.2 correspond to specifications in Listings 6.2 and 6.4, while the final X10 program C.1.3 is the SharedX10 program encoded in X10.

6.2.2 Klaim and RepliKlaim Specifications

A graph is a collection of tuples of the form `(nodeId, nodeDegree)` and `(nodeId, i, neighId)` where `neighId` is an id of the i-th neighbor node, whose total number corresponds to `nodeDegree`.

Listing 6.5 shows a specification in Java-Klaim which relies on traditional approach, with no replicas.

Listing 6.5: Klaim specification

```
1  out('token', size)@l_token;
2  for (i = 0; i < size; i = i + 1)
3    eval(compute(i))@l_h(i);
4  compute(nodeId):
5  {
6    for (i = 0; i < diameter; i = i + 1) {
7      in('token', ?value)@l_token;
8      out('token', value--)@l_token;
9      read(nodeId, ?nodeDegree)@self;
10     for (j = 0; j < nodeDegree; j++) {
11       read(nodeId, j, ?neighId)@self;
12       read(neighId, ?neighDegree)@l_h(neighId);
13       if (neighDegree > nodeDegree) {
14         in(nodeId, nodeDegree)@self;
15         out(nodeId, neighDegree)@self;
16       }
17     }
18     in('token', ?value)@l_token;
19     out('token', value++)@l_token;
20     read('token', size)@l_token;
21   }
22 }
```

At the beginning, the synchronization token is inserted to tuple space with locality l_{token} (`line 1`), which is followed by a `for` loop (`lines 2–3`) that spawns a tread to evaluate `compute` function (defined at `line 4`) per each graph node and at its home location.

Instructions at `lines 7–8` and `18–20` implement barrier synchronization via the synchronization token, which ensures that a process spawned at `line 3`, can proceed to a next iteration (`line 6–21`), only after each such process completed the previous one. The approach shows a standard implementation of the barrier synchronization in tuple space languages (see Section 2.1).

The essential steps of the algorithm are realized at `lines 9–17`. Firstly, node degree is stored in a local `nodeDegree` variable (`line 9`), used to realize iterations through neighbor nodes (`lines 10–17`). If a greater value for degree is found, then node degree is updated with a new value (`lines 14–15`).

As one may observe, obtaining neighbor node degree is a remote operation ($@l_{h(neighId)}$), while operations over the current node are local

(@self).

Listing 6.6 shows a specification in RepliKlaim, based on the idea of replicas illustrated in Figure 33. The specification in Klava is given in Appendix C.1.4.

Listing 6.6: RepliKlaim specification

```
1  out_w('token', size)@l_token;
2  for (i = 0; i < size; i = i + 1)
3    eval(compute(i))@l_h(i);
4  compute(nodeId):
5    for (i = 0; i < diameter; i = i + 1) {
6      in_w('token', ?value)@l_token;
7      out_w('token', value--)@l_token;
8      read(nodeId, ?nodeDegree);
9      out_w('copy', nodeId, nodeDegree)@self;
10     for (j = 0; j < nodeDegree; j++) {
11       in_w(nodeId, j, ?neighId)@self;
12       read(neighId, ?neighDegree)@self;
13       read(nodeId, ?copyDegree)@self;
14       if (neighDegree > copyDegree) {
15         in_w('copy', nodeId, copyDegree)@self;
16         out_w('copy', nodeId, neighDegree)@self;
17       }
18     }
19     in_w('copy', nodeId, ?copyDegree)@self;
20     if (nodeDegree != copyDegree) {
21       in_w(nodeId, nodeDegree)@self;
22       out_w(nodeId, copyDegree)@L_nodeId;
23     }
24     in_w('token', ?value)@l_token;
25     out_w('token', value++)@l_token;
26     read('token', size)@l_token;
27 }
```

The essential characteristic of the RepliKlaim specification is that operations over neighbor node are local, i.e., realized `@self`, the introduction of an auxiliary tuple to store a copy of the current node, and the use of weak operations (`lines 21–22`) to propagate updates to all replicas.

The corresponding specification in Klava is given in Appendix C.1.5.

6.3 PageRank

The PageRank (BP98) algorithm was reported to be a fundamental component of the early versions of Google search engines and it remained the best-known approach to ranking web pages since its announcement. Essentially, the PageRank value of a web page A, or $PR(A)$, represents an approximation of the importance of the page A obtained by examining the importance of the pages linked to it. It is computed in a simple iterative algorithm that employs the hyperlinks between pages. In particular, the PageRank of a page A is given as follows:

$$PR(A) = (1 - d) + d((PR(T_1)/C(T_1) + \cdots + PR(T_n)/C(T_n))$$

where

- $PR(T_i)$ is the PageRank of a *backlink* page T_i, which contains a link to page A,
- $C(T_i)$ is the number of outbound links on page T_i, i.e., the number of links going out of page T_i, and
- d is a *damping factor* which can be set between 0 and 1, and is usually set to 0.85.

The algorithm operates on a graph, such that nodes represent web pages and edges represent hyperlinks between them. Initially, each node is assigned some starting PageRank value, which is further updated in iterations. It is considered that it takes approximately around hundred iterations to get good PageRank approximations for the entire web.

Parallel processing techniques are widely adopted to improve efficiency of large-scale PageRank computations. There are several possibilities for specifying the PageRank algorithm. The aforementioned bulk synchronous parallel model requires that the computation and communication phases of an iteration should be completed for each processor, before the new iteration can begin. This approach enhances well ordering of operations of the algorithm and simplifies the convergence analysis, albeit it has some disadvantages such as the need for synchronization. In fact, given a distributed algorithm it is natural to determine the minimum degree of synchronization which is necessary for the algorithm to work correctly. As the PageRank algorithm computes the approximative values, we thus consider a variant that offer a more flexible ordering of computation and communication between processors and allow processors to proceed at different speed.

6.3.1 X10 and SharedX10 Specifications

In the code listings presented below it is assumed that a `GraphNode` object contains information of its page rank value (`pageRank`), the number of outbound links (`c`) and a list of ids of nodes that link to it (`backlinkNodes`). Methods `getBacklinkNodes()`, `getContribution()` and `setPageRank()` are used respectively to retrieve `backlinkNodes`, ration `pageRank/c` and set node's `pageRank` value.

Listing 6.7 shows an X10 implementation of the algorithm. Like in the previous case study, an activity is spawned per each node to execute the main iterative computation at `lines 2–7`. We assume that the desirable number of iterations `numIterations` is specified in advance. `Line 3` defines a local variable `contribution` that stores summed contributions from the backlink nodes. Finally, the new PageRank value is stored at `line 6`.

Listing 6.7: X10 implementation

```
1 finish for (nodeId in graph) async at (graph.dist(nodeId))
2   for (var i:Long = 0; i<numIterations; i = i + 1) {
3     var contribution:Double = 0.0;
4     for (neighId in graph(nodeId).getBacklinkNodes())
5       contribution += at (graph.dist(neighId))
            graph(neighId).getContribution();
6     graph(nodeId).setPageRank((1 - d) + d*contribution);
7   }
```

The highlighted code retrieves a contribution from a backlink node, and it is executed at the place where a backlink node belongs to. An alternative specification, in which the computation of `contribution` is local, is shown in Listing 6.8. The idea is apply the stencil pattern, based on the replication strategy, i.e.,

```
rvalw@1 sharedGraph:DistArray[GraphNode] = graph
```

Listing 6.8: SharedX10 implementation

```
1 finish for (nodeId in sharedGraph) async at (sharedGraph.
    ↪ dist(nodeId))
2   for (var i:Long = 0; i<numIterations; i = i + 1) async {
```

```
3     var contribution:Double = 0.0;
4     for (neighId in sharedGraph(nodeId).getBacklinkNodes())
5       contribution+=sharedGraph(neighId).getContribution();
6     sharedGraph(nodeId).setPageRank((1 - d) + d*sum);
7   }
```

The full programs are available in Appendix C; X10 program C.2.1 and SharedX10 program C.2.2 correspond to specifications in Listings 6.7 and 6.8, while the final X10 program C.2.3 is obtained after transforming the SharedX10 program.

6.3.2 Klaim and RepliKlaim specifications

Similarly to the specification of the maximum graph degree, a graph is represented as a collection of tuples of form `(nodeId, pageRank, c)`,`(nodeId, i, neighId)` where `neighId` is id of the i-th neighbor node. Listing 6.9 shows an implementation in Java-Klaim.

Listing 6.9: Klaim specification

```
1  for (i = 0; i < size; i = i + 1)
2    eval(compute(i))@l_h(i);
3  compute(nodeId):
4  {
5    for (i = 0; i < numIter; i = i + 1) {
6      in(nodeId, pageRank, ?c);
7      out('contribution', 0);
8      for (j = 0; j < c; j ++) {
9        read(nodeId, j, ?neighId)@self;
10       read(neighId, ?neighPageRank, ?neighC)@l_h(neighId);
11       in('contribution', ?val);
12       out('contribution', val += neighPageRank/neighC)@self;
13     }
14     in('contribution', ?val)@self;
15     out(nodeId, (1-d) + d*val, c)@self;
16   }
17 }
```

Specification in Klava in given in Appendix C.2.4.

Listing 6.10 shows a specification in RepliKlaim. Differently from the above specification, access to the pagerank value of the neighbor node is a local operation.

Listing 6.10: RepliKlaim specification

```
1  for (i = 0; i < size; i = i + 1) {
2    eval(compute(i))@l_{h(i)}
3  }
4  compute(nodeId):
5  {
6    in_w(nodeId, pageRank, ?c)@self;
7    out_w('contribution', 0)@self;
8    for (i = 0; i < numIter; i = i + 1) {
9      for (j = 0; j < c; j ++) {
10       read(nodeId, j, ?neighId)@self;
11       read(neighId, ?pageRank, ?c)@self;
12       in_w('contribution', ?val)@self;
13       out_w('contribution', val += pageRank/c)@self;
14     }
15     in_w('contribution', ?val)@self;
16     in_w(nodeId, pageRank, c)@self;
17     out_w(nodeId, pageRank, (1-d) + d*val)@L_{nodeId};
18   }
19 }
```

Specification in Klava are given in Appendix C.2.5

6.4 Evaluation

This section presents experimental results obtained by running the case study programs on a high-performance computing cluster. However, we could carry out experiments only using the X10 framework, as we came across several highly severe issues in Klaim implementation (Klava). In particular, those included excessive memory consumption that caused a program failures in some cases, as well as high inefficiency of tuple-matching operations and communication protocols.

The results reported below show the performance comparison between two approaches in specifying data locality, namely, the standard one that relies on a centralized data locality (no replicas) and the other based on the replication strategy.

	no-replicas	replicas
maximum graph degree (mgd)	Listing 6.2	Listing 6.4
pageRank (pr)	Listing 6.7	Listing 6.8

Table 5: Evaluated programs

Table 5 associates the presented code listings with the program names and strategy applied for data sharing. The evaluated programs are based on specifications in the table, we refer to them as **mgd_no-replicas**, **mgd_replicas**, **pr_no-replicas** and **pr_replicas**. We make two remarks below in reference to adjustment we undertook to overcome a couple of limiting factors.

6.4.1 Performance Evaluation

The performance evaluation was carried out on a cluster that features 64 HP ProLiant SL2x170z G6 nodes. Each node is configured with 2x Intel Xeon Processor X5550 (quad-core, 2.66 GHz) and 24GB of main memory. The cluster is interconnected with QDR Infiniband interconnect.

X10 runtime relies on X10RT library which provides a communication layer between places. The X10RT is responsible for sending and receiving messages and data between places. There are 3 X10RT implementations available for inter-place communication on multiple hosts, based on TCP/IP, MPI or PAMI protocols. According to the X10 specification, the best performances are achieved with PAMI implementation, while MPI-based takes second place and the TCP/IP sockets third.

Our cluster setup supports only MPI implementation with serialized threading level allowing multiple threads to make MPI calls, but only one thread at a time. Consequently, all evaluated programs had to be compiled to C++ backend, which was done via gcc v.5.2.0, openmpi v.2.0.0 and current X10 runtime version 2.6.0.

The X10 runtime executes activities by scheduling them on a pool of worker threads within each place. To fully exploit physical capacities of our cluster, we set the environment variable X10_NTHREADS to 1, while the value of X10_NPLACES is set to the number of hardware threads, thus collocating 8 X10 places per one SMP node of the cluster. Such configuration enables the workload for our graph computations to be divided uniformly across physical machines.

Methodology. Each program is run ten times to account for potential variances in the X10 runtime.

Evaluation dataset. Experiments are performed using real world graph dataset from Stanford Large Network Dataset Collection (SNAP) repository (SNA16). Each graph is given in a textual format, such that the first line contains the number of nodes and the number of edges, and each of the remaining lines contains two integer numbers representing the two node ids that form an edge.

Nodes	*Edges*	*Diameter*	*Max Degree*	*Description*
5 242	14 496	17	94	Collaboration network
9 877	25 998	17	65	Collaboration network
36 692	183 831	11	1 383	Email comm. network
196 591	950 327	14	14 730	Loc. based soc. netw.
317 080	1 049 866	21	306	DBLP collab. netw.

Table 6: Evaluation datasets

Limitations. Our initial goal was to compare performance of X10 and SharedX10 implementations of specifications as shown in Table 5. However, the evaluation revealed that SharedX10 implementation based on the encoding scheme into (see Section 5.3), was not sufficiently efficient, hence we manually modified implementations of SharedX10 specifications to overcome the issue we detail below.

The analysis showed that the problem in SharedX10 implementation lies the encoding of distributed arrays, `rvals` and `rvalw`, which are shared via replication (see Rules 8 and 9). In particular, one such object, `sharedGraph`, used in Listings 6.4 and 6.8, is encoded as a 2-dim distributed array, e.g. `sharedGraphX10`, such that `sharedGrahX10(nodeId)` is a distributed array of replicas of graph node whose id is `nodeId`. As captured by the Rule 10, the translation of `sharedGraph(neighId)`, which targets the local replica of neighbor node `neighId`, is

```
(at(sharedGraphX10.dist(neighId))sharedGraphX10(neighId)) (
    ↪ sharedGraphX10(neighId).dist.get(here).maxPoint())
```

Essentially, evaluating the element location in a 2-dim distributed array requires remote communication, i.e., involves the place-shifting `at` construct, even when the data is collocated with the activity that accesses it. On the contrary, it is desirable and expectable that one can write

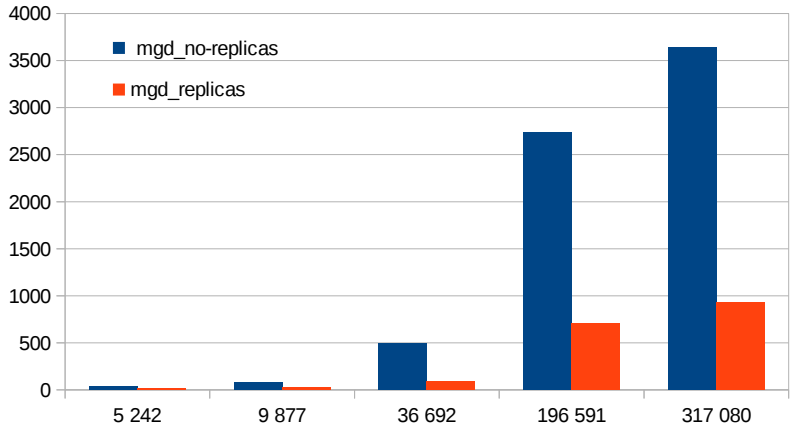


Figure 34: Performance of **mgd_no-replicas** and **mgd_replicas** at different sizes of graph (shown along x-axis) on 8 hardware threads (1 cluster node).

directly `array2(nodeId1)(nodeId2)` to access the local element on position `(nodeId1, nodeId2)` in a 2-dim distributed array `array2`. This seems to be a limitation of the current X10 implementation for 2-dim distributed arrays, as in the case of 1-dim distributed arrays, local array elements can be accessed directly, without the need for activity place-shifting. Furthermore, we found no alternative possibility to encode such shared distributed arrays. Therefore, we concluded that an efficient SharedX10 implementation would have to include optimizations on the compiler and/or runtime level, primarily concerning the memory management, as e.g. done in (SU15). Further investigation of these optimizations is a possible direction for the future work.

As a consequence of the aforementioned problem, the performance gain which would come from localizing data accesses by replication was entirely canceled by the costly place-shifting operation. In order to obtain performance measures for programs which rely on replicas, we modified programs **mgd_replicas** and **pr_replicas** by manually programming data replication and consistency such that no place-shifting is required when accessing local replicas.

The second remark concerns the organization of graph processing.

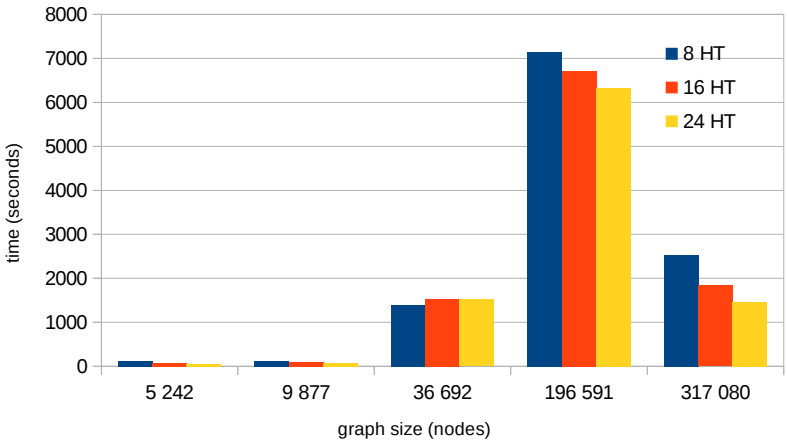


Figure 35: Performances of **pr_no-replicas** at different sizes of graph (shown along x-axis) on 8, 16 and 24 hardware threads (HT).

For the purpose of presentation, in the code listings each graph node is processed by a separate activity, hence all graph nodes are conceptually processed in parallel. In practice, this approach causes congestion of resources due to scheduling a large number of activities on incomparably fewer number of physical threads. Instead, in our implementations, graph nodes are divided into regions equal to the number of places and processed sequentially by one activity, scheduled on single hardware thread.

Case Study I - Results

Figure 34 shows the performance comparison between **mgd_no-replicas** and **mgd_replicas**. The evaluation showed significant performance gain attainable with replication-strategy on all datasets given in Table 6. The speedup mainly comes from localizing accesses to neighbor nodes for retrieving their degrees. However, we discovered that performances did not improve with increased number of hardware threads, due to the increased communication cost.

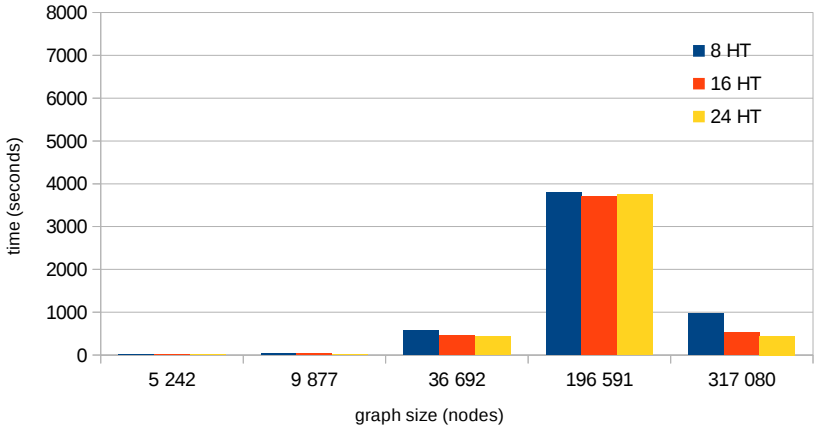


Figure 36: Performances of **pr_replicas** at different sizes of graph (shown along x-axis) on 8, 16 and 24 hardware threads (HT).

Case Study II - Results

Figures 35 and 36 show respectively performances of **pr_no-replicas** and **pr_replicas**. Each figure shows 3 bars for each graph dataset representing results obtained on 1, 2 and 3 cluster nodes each featuring 8 hardware threads each. Table 7 shows the average speedup in percentages that **pr_replicas** achieved over **pr_no-replicas**.

Dataset size	5 242	9 877	36 692	196 591	317 080
Speedup	683%	417%	301%	179%	299%

Table 7: Average performance speedup of **pr_replicas** over **pr_no-replicas**

The results showed that performances depend heavily on the ratio between the number of edges and nodes. Increasing the number of hardware threads did not significantly impact performances on the case of graph with 196 591 nodes. Furthermore, it took more than twice as much time for that dataset than for a larger one of 317 080 nodes, even though the number of edges is almost similar for the two graphs. We conclude

that this result is due to the high communication cost which increased with the number of hardware threads.

6.4.2 Programmability Evaluation

This section addresses the impact of the proposed approach on the programmability aspect using the two case studies presented in this chapter. Therefore, we measured the programmer's effort, in terms of lines of code, required to express carefully hand-written solutions that would lead to optimal performances. We drew comparison between SharedX10 and RepliKlaim programs and their encoded variants in, respectively, X10 and Klaim.

Table 8 reports on the number of code lines taken to expresses algorithms in SharedX10 programs and the encoded variants in X10 .

	Maximum Degree	PageRank
X10	33	29
SharedX10	13	11

Table 8: Programmability comparison between X10 and SharedX10

Table 9 reports on a similar comparison between RepliKlaim and Klaim programs.

	Maximum Degree	PageRank
Klaim	48	32
RepliKlaim	27	19

Table 9: Programmability comparison between Klaim and RepliKlaim

The results show that programming abstractions for data sharing can significantly reduce the programmer's effort in expressing *performant* solutions.

6.5 Summary and Related Work

In this chapter we considered two case studies to make a performance and programmability comparison between SharedX10 and X10, as well

as RepliKlaim and Klaim. The results pointed to the benefits of programming abstractions for data sharing, however, a few limitations and challenges remained for the future work. In particular, a more efficient implementation of Klaim is required, as well as a more efficient implementation of SharedX10 that could potentially include compiler and runtime optimizations.

Numerous frameworks and libraries have been proposed particularly for large-scale graph analytics, including an open-source library SCALEGRAPH (SU15) built on top of X10. Its model is based on the bulk synchronous parallel model, and the implementation includes optimizations of X10 runtime in terms of communication and memory management.

Chapter 7

Conclusions

The multiprocessor structures that are currently emerging are built out of many multi-core SMP nodes with non-uniform memory hierarchies which are interconnected in scalable cluster configurations. It is argued that with increasing rate of parallelism the cost of moving data has become a dominant factor for performance, in term of application execution time and energy efficiency. Consequently, this shift in hardware evolution, towards highly parallel systems, has posed challenges for new solutions for programming those systems efficiently.

In the realm of distributed computing, one can find several high-performance computing languages that offer support for designing applications on the emerging hardware architectures. The de-facto standard, the message-passing model (MPI), is being challenged by new languages and programming models that try to address concerns such as the *memory address to physical location* problem. In fact, as cache-coherence is no longer attainable on emerging cluster systems, achieving good data locality without compromising programmability has become the primary goal. As a result, programmers are expected to properly place data and processes operating on them and to adequately orchestrate data exchange among the different locations containing memory and processors.

In this work, we tackled the problem of providing appropriate linguistic abstractions to handle data sharing that, in presence of multiple computational contexts, can rely either on replication or on centralized locality of data. Moreover, the granularity of consistency for replicated data can be at the level of a single operation or at the level of a data item.

The choice may depend on the application domain; for example, in reactive applications, allowing the same data to be accessed with different consistency levels, could provide opportunities to sacrifice strong consistency for performance in the presence of system overload or opportunities to meet the real-time needs of an application. The main goal of this thesis was to show the benefits of using specific linguistic abstractions for data sharing in terms of programmability and performance, when compared to the standard approaches based on orchestrating data communication.

We designed programming abstractions for data sharing for two *partitioned space* languages, namely Klaim and X10, where data items and partitions are *tuples* and *tuple spaces* in Klaim, and *objects* and *places* in X10. Both languages rely on asynchronous computations and make use of explicit localities, and these features make them suitable candidates for programming scalable cluster systems. However, they offer different features which highly influenced the proposed data sharing abstractions. In particular, Klaim is more suitable for programming applications with reactive behavior, while X10 is better suited for imperative programming. Moreover, X10 features different mechanisms for data access, depending on whether the data is local or remote to the process trying to access, while Klaim offers uniform approach in accessing data item.

In the thesis, we introduced extensions of Klaim and X10: RepliKlaim enriches Klaim with primitives for sharing based on replication, while SharedX10 extends X10 and in addition to replication strategy provides sharing based on centralized locality of data which guarantees uniform access to data by both local and remote processes. We considered two levels of consistency for replicated data, namely *strong* and *sequential* consistency, and adjusted the granularity of consistency to the individual language. In RepliKlaim consistency is guaranteed at the level of a single operation, thus permitting the same data item throughout its lifetime to be accessed with operations of different consistency levels. In SharedX10, the granularity of consistency is suited to the imperative paradigm and is thus provided at the level of the data item.

7.1 Directions for Future Work

We would like to conclude by simply listing the challenges that we see in front of us and that could be the topics of future research.

- To consider other forms of consistency beyond strong and sequential consistency (see (FR10; VV16) for an overview), as advocated, e.g., in (Ter13; Bre12), or offered in C++ though ordering annotations (C++16).
- To understand if there are automatic ways to support the programmer in deciding when and which form of consistency to use; e.g., by following the approach described in (LPC$^+$12).
- To investigate compiler and runtime optimizations techniques for guaranteeing more efficient implementations, such as the one presented in (SU15; PTA14).
- To apply our approach to other paradigms and languages. In particular, suitable models would be those that come closer to the traditional shared memory model where the idea of sharing is already present.

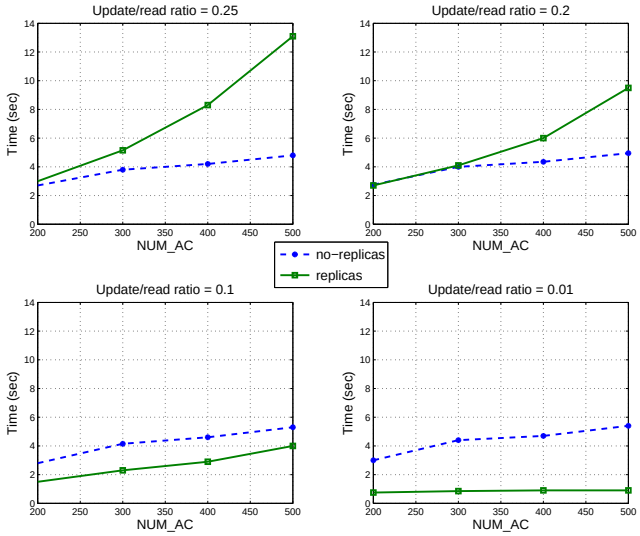
Relatively to the last item, we would like to stress that we see it difficult to apply our approach to models such as agent-based and actor-based that adhere to the set of rules defined by the message passing. In these models, the idea of data sharing is not natural and forcing it would violate their essence. An interesting candidate for our extension is the SCEL language (DLPT14). One specific difference between SCEL and Klaim is that in the former the target of tuple operations can be specified by a predicate on the attributes of components. This provides a great flexibility as it allows to use group-cast operations without explicitly creating groups (*ensembles* in SCEL). In many applications creating replicas is a convenient mechanism to share information among groups. However, the dynamicity of ensembles (components may change attributes at run-time and thus join and leave ensembles arbitrarily) poses additional challenges when defining the semantics and providing implementation of shared data items that are worth further investigations.

Appendices

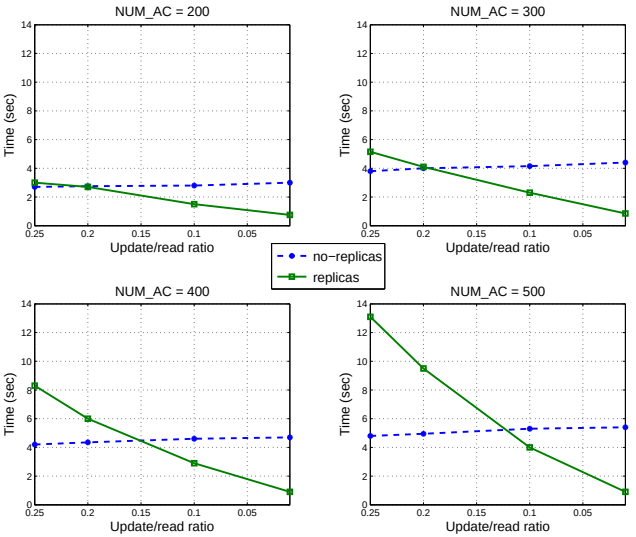
Appendix A

Results for eight places

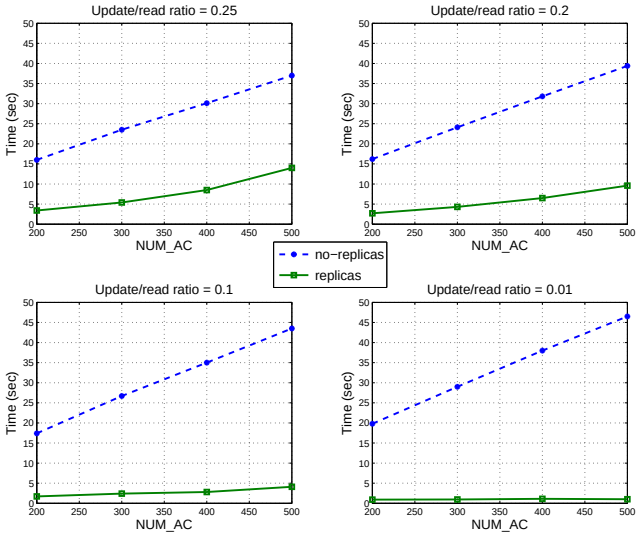
Figure 37: X10 Experiments: Scenario with 8 places



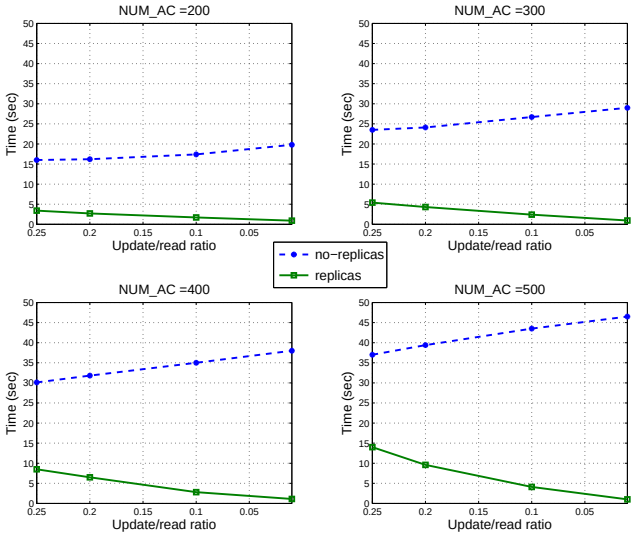
(a) (`Ratio`): The two strategies with shared data of size ≈ 0.4`MB`



(b) (`Access number`): The two strategies with shared data of size ≈ 0.4MB



(c) (`Ratio`): The two strategies with shared data of size $\approx$ 4MB



(d) (`Access number`): The two strategies with shared data of size $\approx$ 4MB

Appendix B

SharedX10 Grammar

Appendix B presents the complete SharedX10 grammar specification in the XTEXT framework.

```
1  grammar org.xtext.SharedX10
2    with org.eclise.xtext.common.Terminals
3  Program:
4    ('package' name = QualifiedName ';')?
5    importElements += AbstractElements*
6    class += Class*
7  ;
8  AbstractElements:
9    'import' importedNamespace = QualifiedNameWithWildcard';'
10 ;
11 QualifiedNameWithWildcard:
12   QualifiedName '.*'?
13 ;
14 QualifiedName:
15   ID ('.' ID)*
16 ;
17 Class:
18   'public' 'class' name = ID
19   ('extends' superclass=[Class|QualifiedName])?
20   '{' members += Member* '}'
21 ;
22 Member:
23   MainMethod |
24   Method |
```

```
25    Print |
26    Operator |
27    FuncVariableDef |
28    VariableDef |
29    SharedVariableDef
30  ;
31  MainMethod:
32    'public' 'static' 'def' 'main'
33    '(' type = VariableType ')'
34    body = Body
35  ;
36  Method:
37    'def' name = (ID | 'this') '(' (params += Parameter
38    (',' params += Parameter)*)? ')' (isconst ?= 'const')?
39    body = Body
40  ;
41  Print:
42    'Console' ('.' ID)*
43    '(' expression = Expression ')' ';'
44  ;
45  Operator:
46    'public' 'operator' name = (ID | 'this')
47    '(' (params += Parameter (',' params += Parameter)*)? ')'
48    '=' expression = Expression ';'
49  ;
50  FuncVariableDef:
51    'val' name = ID ':'
52    '(' (params += Parameter (',' params += Parameter)*)? ')'
53    '=>' type = VariableType '='
54    '(' (params += Parameter (',' params += Parameter)*)? ')'
55    '=>' body = Body ';'
56  ;
57  VariableDef:
58     vartype = ('var' | 'val')
59    name = ID (istyped ?= ':' type = VariableType)?
60    (isinit ?= '=' expression = ArithExpression)? ';'
61  ;
62  SharedVariableDef:
63    'sval' name = ID ':' type = VariableType '=' expression =
64    Expression ';' |
65    'rvals' name = ID '@' places = [VariableDef] ':' type =
66    VariableType '=' expression = Expression ';' |
67    'rvalw' name = ID '@' places = [VariableDef] ':' type =
```

```
68    VariableType '=' expression = Expression ';'
69  ;
70  Parameter:
71    name = ID (istyped ?= ':' type = VariableType)?
72  ;
73  Body:
74    '{' statements += Statement* '}'
75  ;
76  Block:
77    statements += Statement |
78    ispar ?= '{' statements += Statement* '}'
79  ;
80  VariableType:
81    type = [Class|QualifiedName]
82    (isarray ?= '[' innerType = VariableType ']' )?
83  ;
84  Statement:
85    Async |
86    Finish |
87    At |
88    Atomic |
89    WhenAtomic |
90    For |
91    ForEnum |
92    If |
93    While |
94    Return |
95    Print |
96    TryCatch |
97    SharedVariableDef |
98    VariableDef |
99    FuncVariableDef |
100   Expression ';'
101 ;
102 Async returns Statement:
103   'async' body = Block
104 ;
105 Finish returns Statement:
106   'finish' body = Block
107 ;
108 At returns Statement:
109   'at' '(' exppression = SelectionExpression ')'
110   body = Block
```

```
;
Atomic:
  'atomic' statement = Statement
;
WhenAtomic:
  'when' '(' expression = Equality ')'
  statement = Statement
;
For returns Statement:
  'for' '(' init = VariableDef
  condition = ArithExpression ';' finalexp = Expression ')'
  body = Block
;
ForEnum returns Statement:
  'for'
  '(' par = Parameter 'in' data = TerminalExpression ')'
  body = Block
;
If returns Statement:
  'if' '(' expression = ArithExpression ')'
  thenBlock = Block
  (=> iselse ?= 'else' elseBlock = Block)?
;
While returns Statement:
  'while' '(' expression = Expression ')'
  body = Body
;
Return returns Statement:
  'return' expression = Expression ';'
;
TryCatch returns Statement:
  'try' bodyTry = Body
  'catch' '(' name = ID ')' bodyCatch = Body
;
Expression:
  Assignment
;
ArithExpression returns Expression:
  Or
;
Assignment returns Expression:
  SelectionExpression
  ({Assignment.left = current} '='
```

```
154   right = ArithExpression)?
155 ;
156 SelectionExpression returns Expression:
157   TerminalExpression
158   (
159   {MemberSelection.receiver = current}
160   '.' member = [Member]
161   (ispar ?= '[' par = [Class] ']')?
162     (methodinvocation ?=
163     '(' (args += Expression (',' args += Expression)*)? ')'
164     )?
165   )*
166 ;
167 Or returns Expression:
168   And ({Or.left = current} '||' right = And)*
169 ;
170 And returns Expression:
171   Equality ({And.left = current} '&&' right = Equality)*
172 ;
173 Equality returns Expression:
174   Comparison
175   (
176   {Equality.left = current} op = ('==' | '!=')
177   right = Comparison
178   )*
179 ;
180 Comparison returns Expression:
181   PlusOrMinus
182   (
183   {Comparison.left = current} op = ('<=' | '>=' | '<' |
184    '>')
185   right = PlusOrMinus
186   )*
187 ;
188 PlusOrMinus returns Expression:
189   MulOrDiv
190   (
191   ({Plus.left = current} '+' | {Minus.left = current} '-')
192   right = MulOrDiv
193   )*
194 ;
195 MulOrDiv returns Expression:
196   Primary
```

```
  (
  {MulOrDiv.left = current} op = ('*' | '/')
  right = Primary
  )*
;
Primary returns Expression:
  '(' Expression ')' |
  {Not} '!' expression = Primary |
  SelectionExpression |
  'at' '(' expression = SelectionExpression ')'
  body = SelectionExpression
;
TerminalExpression returns Expression:
  {StringConstant} value = STRING |
  {IntConstant} value = IntegerNegative |
  {BoolConstant} value = ('true' | 'false') |
  {DeRef} ref = [VariableDef] '(' ')' |
  {This} 'this' |
  {Null} 'null' |
  {Here} 'here' |
  {Reference} base = [Base] (isarray ?= '('
  params += SelectionExpression ')')? |
  {New} 'new' type = VariableType '(' (args += Expression
  (',' args += Expression)*)? ')' |
  {Init} '('type = VariableType ')' '=>'
  expression = TerminalExpression
;
IntegerNegative:
  (isneg ?= '-')? value = INT
;
Base:
  SharedVariableDef |
  FuncVariableDef |
  Parameter |
  Class
;
```

Appendix C

Specifications of Case Studies

C.1 Case Study I: Maximum Graph Degree

C.1.1 Specification in X10

```
1  /* Algorithm computing maximum graph degree */
2  clocked finish for (reg in dist.regions()) clocked async
3    for (nodeId:Point in reg) at(dist(nodeId)){
4      for (var i:Long = 0; i < diameter ; i++) {
5        for (neighId in graph(nodeId).neighbors) {
6          val neighDegree:Long = at(dist(neighId)) graph(
               ↪ neighId).getDegree();
7          if (neighDegree > graph(nodeId).getDegree())
8            graph(nodeId).setDegree(neighDegree);
9        }
10     }
11   Clock.advanceAll();
```

C.1.2 Specification in SharedX10

```
/* Graph sharing */
rvalw@l sharedGraph:DistArray[GraphNode] = graph;
/* Compute the maximum graph degree */
clocked finish for (reg in dist.regions()) clocked async
  for (nodeId:Point in reg) at(dist(nodeId)) {
    for (var i:Long = 0; i < diameter ; i++) {
      val maxDegree:Long = sharedGraph(nodeId).getDegree();
      for (neighId in sharedGraph(nodeId).neighbors)
        if (sharedGraph(neighId).getDegree() > maxDegree)
          maxDegree = sharedGraph(neighId).getDegree();
          if (sharedGraph(nodeId).getDegree() != maxDegree)
            sharedGraph(nodeId).setDegree(maxDegree);
        Clock.advanceAll();
    }
  }
```

C.1.3 SharedX10 Specification Encoded in X10

```
1  /* Creation of sharedGraph */
2  val sharedGraph:DistArray[DistArray[GraphNode]] = DistArray
       ↪ .make[DistArray[GraphNode]](graph.dist);
3  val sharedGraph\_tokens:DistArray[Token] = DistArray.make[
       ↪ Token](graph.dist);
4  val sharedGraph\_distributions:DistArray[Dist] = DistArray.
       ↪ make[Dist](graph.dist);
5  for (p in graph) at (graph.dist(p)) {
6    allPlaces(p).sort(cmp);
7    val s:Long = allPlaces(p).size();
8    val temp:Rail[Place] = new Rail[Place](s);
9    for (var j:Long = 0; j < s; j++)
10     temp(j) = allPlaces(p)(j);
11   val replicaPlaces:PlaceGroup = new SparsePlaceGroup(temp)
         ↪ ;
12   val replicaDist = Dist.makeUnique(replicaPlaces);
13   sharedGraph\_tokens(p) = new Token();
14   sharedGraph\_distributions(p) = replicaDist;
15   sharedGraph(nodeId) = DistArray.make[GraphNode](
         ↪ replicaDist, ([i]:Point(1)) => graph(p));
16 }
17 /* Computation of the maximum graph degree */
18 clocked finish for (reg in dist.regions()) clocked async {
19   for (nodeId:Point in reg) {
20     for (var i:Long = 0; i < diameter ; i++) {
21       var maxDegree:Long = (sharedGraph(nodeId)(
             ↪ distributions(nodeId).get(here).maxPoint())).
             ↪ getDegree();
22       for (neighId in (sharedGraph(nodeId)(distributions(
             ↪ nodeId).get(here).maxPoint())).neighbors)
23         if ((sharedGraph(nodeId)(distributions(nodeId).get(
               ↪ here).maxPoint())).getDegree() > maxDegree)
24           maxDegree = at (sharedGraph.dist(neighId)) (
                 ↪ sharedGraph(neighId)(distributions(neighId).
                 ↪ get(here).maxPoint())).getDegree();
25       }
26       if ((sharedGraph(nodeId)(distributions(nodeId).get(
             ↪ here).maxPoint())).getDegree() != maxDegree)
27         val newDegree_graph = newDegree;
28         sharedGraph_tokens(nodeId).acquire();
```

```
29        finish for (p in sharedGraph(nodeId)) at (
              ↪ distributions(nodeId)(r))
30          sharedGraph(nodeId)(r).setDegree(newDegree_graph);
31        sharedGraph_tokens(nodeId).release();
32       }
33     }
34   Clock.advanceAll();
35   }
```

C.1.4 Specification in Klaim

```
1  for (int i = 0; i < diameter.integer; i++) {
2    /* Synchronization I */
3    tokenValue = new KInteger();
4    in(new Tuple(new KString("token"), tokenValue), server);
5    out(new Tuple(new KString("token"), new KInteger(
         ↪ tokenValue.integer-1)), server);
6    KInteger nodeId = new KInteger();
7    KInteger nodeDeg = new KInteger();
8    Tuple node = new Tuple(nodeId, nodeDeg);
9    while(read_nb(node, self)) {
10     System.out.println(nodeId);
11     neighbors = new TupleSpaceVector();
12     read(new Tuple(node.getItem(0), neighbors), self);
13     /* Iterate through neighbors */
14     Tuple neighborId = new Tuple(new KInteger());
15     while(neighbors.read_nb(neighborId)) {
16       neighId = (KInteger) neighborId.getItem(0);
17       neighL = (PhysicalLocality) vectorClients.getTuple(
             ↪ neighId.integer%numClients).getItem(1);
18       neighborDeg = new KInteger();
19       read(new Tuple(neighborId.getItem(0), neighborDeg),
             ↪ neighL);
20       if(neighborDeg.integer > nodeDeg.integer) {
21         in(new Tuple(node.getItem(0), node.getItem(1)), self)
               ↪ ;
22         out(new Tuple(node.getItem(0), neighborDeg), self);
23       }
24       neighborId.resetOriginalTemplate();
25     }
26     node.resetOriginalTemplate();
27   }
28   /* Synchronization II */
29   tokenValue = new KInteger();
30   in(new Tuple(new KString("token"), tokenValue), server);
31   out(new Tuple(new KString("token"), new KInteger(
         ↪ tokenValue.integer+1)), server);
32   read(new Tuple(new KString("token"), new KInteger(
         ↪ numClients)), server);
33 }
```

C.1.5 Specification in RepliKlaim

```
1  for (int i = 0; i < diameter.integer; i++) {
2    /* Synchronization part I */
3    tokenValue = new KInteger();
4    /* Token is stored at a distinguished locality - server
         ↪ */
5    in(new Tuple(new KString("token"), tokenValue), server);
6    out(new Tuple(new KString("token"), new KInteger(
         ↪ tokenValue.integer-1)), server);
7    KInteger nodeId = new KInteger();
8    TupleSpaceVector neighbors = new TupleSpaceVector();
9    Tuple node = new Tuple(nodeId, new TupleSpaceVector());
10   KInteger copyDegree = new KInteger();
11   while(read_nb(node, self)) {
12     KInteger nodeDeg = new KInteger();
13     read(new Tuple(node.getItem(0), nodeDeg), self);
14     /* Create a copy of the current node */
15     out(new Tuple(new KString("copy"), node.getItem(0), node
           ↪ .getItem(1)), self);
16     /* Iterate through neighbors */
17     Tuple neighborId = new Tuple(new KInteger());
18     while(neighbors.read_nb(neighborId)) {
19       neighId = (KInteger) neighborId.getItem(0);
20       neighborDeg = new KInteger();
21       read(new Tuple(neighborId.getItem(0), neighborDeg),
             ↪ self);
22       read(new Tuple(new KString("copy"), node.getItem(0),
             ↪ copyDegree), self);
23       if(neighborDeg.integer > copyDegree.integer) {
24         /* Update the local copy */
25         in(new KString("copy"), node.getItem(0), new KInteger
               ↪ ()), self);
26         out(new KString("copy"), node.getItem(0), neighborDeg
               ↪ ), self);
27       }
28       neighborId.resetOriginalTemplate();
29     }
30     /* Update the current node if necessary */
31     copyDegree = new KInteger();
32     in(new Tuple(new KString("copy"), node.getItem(0),
           ↪ copyDegree), self);
33     if(copyDegree.integer > nodeDeg.integer) {
```

```
34        update(new Tuple(node.getItem(0), copyDegree));
35      }
36      node.resetOriginalTemplate();
37    }
38    /* Synchronization part II */
39    tokenValue = new KInteger();
40    in(new Tuple(new KString("token"), tokenValue), server);
41    out(new Tuple(new KString("token"), new KInteger(
          ↪ tokenValue.integer+1)), server);
42    read(new Tuple(new KString("token"), new KInteger(
          ↪ numClients)), server);
43  }
```

C.1.6 Update function

```
1  public void update(Tuple t) throws KlavaException {
2    PhysicalLocality ploc;
3    Tuple template = new Tuple (new LogicalLocality(), new
         ↪ PhysicalLocality());
4    while(vectorClients.read_nb(template)) {
5      ploc = (PhysicalLocality) template.getItem(1);
6      if(read_nb(new Tuple(t.getItem(0), new KInteger()), ploc
           ↪ )) {
7        in(new Tuple(t.getItem(0), new KInteger()), ploc);
8        out(new Tuple(t.getItem(0), t.getItem(1)), ploc);
9      }
10   template.resetOriginalTemplate();
11   }
12 }
```

C.2 Case Study II: PageRank

C.2.1 Specification in X10

```
1  /* Algorithm computing node pageRank values */
2  finish for (reg in dist.regions()) async
3    for (nodeId:Point in reg) at(dist(nodeId)){
4      for (var i:Long = 0; i < numIterations ; i++) {
5        var sum:Double = 0.0;
6        for (neighId in graph(nodeId).backlinkNodes) {
7          sum += at (graph.dist(neighId)) graph(neighId).
               ↪ getContribution();
8          graph(nodeId).setPageRank((1-dfactor) + dfactor*sum);
9        }
10     }
11   }
```

C.2.2 Specification in SharedX10

```
1  /* Graph sharing */
2  rvalw@l sharedGraph:DistArray[GraphNode] = graph;
3  /* Compute PageRank values */
4  finish for (reg in dist.regions()) async
5    for (nodeId:Point in reg) {
6      for (var i:Long = 0; i < numIterations ; i++) {
7        var sum:Double = 0.0;
8        for (neighId in sharedGraph(nodeId).backlinkNodes)
9          sum += sharedGraph(neighId).getContribution();
10         sharedGraph(nodeId).setPageRank((1-dfactor) + dfactor
               ↪ *sum);
11       }
12     }
13   }
```

C.2.3 SharedX10 Specification Encoded in X10

```
1  /* Creation of sharedGraph */
2  val sharedGraph:DistArray[DistArray[GraphNode]] = DistArray
       ↪ .make[DistArray[GraphNode]](graph.dist);
3  val sharedGraph_tokens:DistArray[Token] = DistArray.make[
       ↪ Token](graph.dist);
4  for (p in graph) at (graph.dist(p)) {
5    allPlaces(p).sort(cmp);
6    val s:Long = allPlaces(p).size();
7    val temp:Rail[Place] = new Rail[Place](s);
8    for (var j:Long = 0; j < s; j++)
9      temp(j) = allPlaces(p)(j);
10   val replicaPlaces:PlaceGroup = new SparsePlaceGroup(temp)
         ↪ ;
11   val replicaDist = Dist.makeUnique(replicaPlaces);
12   sharedGraph_tokens(p) = new Token();
13   sharedGraph(nodeId) = DistArray.make[GraphNode](
         ↪ replicaDist, ([i]:Point(1)) => graph(p));
14 }
15 /* Computation of PageRank values */
16 finish for (reg in dist.regions()) async
17   for (nodeId:Point in reg) at(dist(nodeId)) {
18     for (var i:Long = 0; i < numIterations ; i++) {
19       var pr:Double;
20       var sum:Double = 0.0;
21       for (neighId in graph(nodeId).backlinkNodes)
22         sum += at(dist(neighId))
23           (sharedGraph(neighId)(((distributions(neighId)).get
                 ↪ (here)).maxPoint())).getRatio();
24       pr = (1-dfactor) + dfactor*sum;
25       val nPr = pr;
26       tokens_graph(nodeId).acquire();
27       async for (r in sharedGraph(nodeId)) at(distributions(
             ↪ nodeId)(r))
28         sharedGraph(nodeId)(r).setPr(nPr);
29       tokens_graph(nodeId).release();
30     }
31   }
```

C.2.4 Specification in Klaim

```
1  for (int i = 0; i < numIter.integer; i++) {
2    KInteger tid = new KInteger(); TupleSpaceVector
         ↪ backlinkNodes = new TupleSpaceVector();
3    Tuple template = new Tuple(tid, backlinkNodes);
4    /* Iterates through local graph nodes */
5    while(read_nb(template, self)) {
6      KInteger cid = new KInteger();
7      read(new Tuple(tid, new KDouble(), cid), self);
8      double contribution = 0.0;
9      /* Iterates through backlink nodes and collect
           ↪ contributions */
10     Tuple blNodeId = new Tuple(new KInteger());
11     while(backlinkNodes.read_nb(blNodeId)) {
12       KInteger nodeId = (KInteger) blNodeId.getItem(0);
13       PhysicalLocality blNodeL = (PhysicalLocality)
             ↪ vectorClients.getTuple(nodeId.integer%numClients
             ↪ ).getItem(1);
14       KInteger blnodeC = new KInteger(); KDouble blnodePR =
             ↪ new KDouble();
15       read(new Tuple(blNodeId.getItem(0), blnodePR, blnodeC)
             ↪ , blNodeL);
16       contribution += blnodePR.d/blnodeC.integer;
17       blNodeId.resetOriginalTemplate();
18     }
19     /* Updates the pagerank value of the current graph node
           ↪ */
20     in(new Tuple(tid, new KDouble(), new KInteger()), self);
21     out(new Tuple(tid, new KDouble(1-d + d*contribution),
           ↪ cid), self);
22     template.resetOriginalTemplate();
23   }
24 }
```

C.2.5 Specification in RepliKlaim

```
for (int i = 0; i < numIter.integer; i++) {
  KInteger tid = new KInteger();
  TupleSpaceVector backlinkNodes = new TupleSpaceVector();
  Tuple template = new Tuple(tid, backlinkNodes);
  /* Iterates through local graph nodes */
  while(read_nb(template, self)) {
    KInteger cid = new KInteger();
    read(new Tuple(tid, new KDouble(), cid), self);
    double contribution = 0.0;
    /* Iterates through backlink nodes and collect
        ↪ contributions */
    Tuple blNodeId = new Tuple(new KInteger());
    while(backlinkNodes.read_nb(blNodeId)) {
      KInteger nodeId = (KInteger) blNodeId.getItem(0);
      KInteger blnodeC = new KInteger();
      KDouble blnodePR = new KDouble();
      read(new Tuple(blNodeId.getItem(0), blnodePR, blnodeC)
          ↪ , self);
      contribution += blnodePR.d/blnodeC.integer;
      blNodeId.resetOriginalTemplate();
    }
    /* Updates the pagerank value of the current graph node
        ↪ */
    update(new Tuple(tid, new KDouble(1-d + d*contribution),
        ↪  cid));
    template.resetOriginalTemplate();
  }
}
```

C.2.6 Update function

```
1  public void update(Tuple newTuple) throws KlavaException {
2    /* Tuple *template* will be used to store localities of
         ↪ each client */
3    Tuple template = new Tuple (new LogicalLocality(), new
         ↪ PhysicalLocality());
4    /* Iterate through vector of clients and replaces the old
         ↪  tuple with the new one */
5    while(vectorClients.read_nb(template)) {
6      PhysicalLocality loc = (PhysicalLocality) template.
           ↪ getItem(1);
7      if(read_nb(new Tuple(newTuple.getItem(0), new KDouble(),
           ↪  new KInteger()), loc)) {
8        in(new Tuple(newTuple.getItem(0), new KDouble(), new
             ↪ KInteger()), loc);
9        out(new Tuple(newTuple.getItem(0), newTuple.getItem(1)
             ↪ , newTuple.getItem(2)), loc);
10     }
11     template.resetOriginalTemplate();
12   }
13 }
```

References

[Adv93] Sarita Vikram Adve. *Designing Memory Consistency Models for Shared-memory Multiprocessors*. PhD thesis, Madison, WI, USA, 1993. UMI Order No. GAX94-07354. 34

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996. 40

[Atk08] A. K. Atkinson. Tupleware: A distributed tuple space for cluster computing. In *2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 121–126, Dec 2008. 5

[Atk10] Alistair Kenneth Atkinson. *Development and Execution of Array-based Applications in a Cluster Computing Environment*. PhD thesis, University of Tasmania, 2010. 66

[BCP07] Davide Balzarotti, Paolo Costa, and Gian Pietro Picco. The lights tuple space framework and its customization for context-aware applications. *Web Intelligence and Agent Systems*, 5(2):215–231, 2007. 66

[BDL06] Lorenzo Bettini, Rocco De Nicola, and Michele Loreti. Implementing mobile and distributed applications in x-klaim. *Scalable Computing: Practice and Experience*, 7(4), 2006. 66

[BDP02] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. Klava: a java package for distributed and mobile applications. *Softw., Pract. Exper.*, 32(14):1365–1394, 2002. 5, 66

[BEH14] Ahmed Bouajjani, Constantin Enea, and Jad Hamza. Verifying eventual consistency of optimistic replication systems. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*

'14, San Diego, CA, USA, January 20-21, 2014, pages 285–296. ACM, 2014. 1

[Bet03] Lorenzo Bettini. PhD thesis - Linguistic Constructs for Object-Oriented Mobile Code Programming and their Implementations. `http://klava.sourceforge.net/`, 2003. 5

[Bet11] Lorenzo Bettini. A DSL for writing type systems for xtext languages. In Probst and Wimmer (PW11), pages 31–40. 28

[BFLW12] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, pages 283–307, 2012. 2

[BGZ97] N. Busi, R. Gorrieri, and G. Zavattaro. A truly concurrent view of linda interprocess communication. Technical report, University of Bologna, 1997. 66

[BGZ00] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. Comparing three semantics for linda-like languages. *Theor. Comput. Sci.*, 240(1):49–90, 2000. 35, 66

[BMZ04] Nadia Busi, Alberto Montresor, and Gianluigi Zavattaro. Data-driven coordination in peer-to-peer information systems. *Int. J. Cooperative Inf. Syst.*, 13(1):63–89, 2004. 66

[BP98] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Seventh International World-Wide Web Conference (WWW 1998)*, 1998. 105

[Bre12] E. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012. 118

[C++14] ISO/IEC 14882:2014(E) Programming language C++. International Organization for Standardization ISO, Geneva, CH, Nov.2014., 2014. 35

[C++16] C++ memory orderings. Reference website: `http://en.cppreference.com/w/c/atomic/memory_order`, 2016. 118

[Cap08] Sirio Capizzi. *A tuple space implementation for large-scale infrastructures*. PhD thesis, University of Bologna, 2008. 66

[CCS+13] Silvia Crafa, David Cunningham, Vijay A. Saraswat, Avraham Shinnar, and Olivier Tardieu. Semantics of (resilient) X10. *CoRR*, abs/1312.3739, 2013. 28

[CCS+14] Silvia Crafa, David Cunningham, Vijay A. Saraswat, Avraham Shinnar, and Olivier Tardieu. Semantics of (resilient) X10. In Richard Jones, editor, *ECOOP 2014 - Object-Oriented Programming - 28th European Conference. Proceedings*, volume 8586 of *Lecture Notes in Computer Science*, pages 670–696. Springer, 2014. 20

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007. 5

[CGB+06] Fan R. K. Chung, Ronald L. Graham, Ranjita Bhagwan, Stefan Savage, and Geoffrey M. Voelker. Maximizing data locality in distributed systems. *J. Comput. Syst. Sci.*, 72(8):1309–1316, 2006. 1

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005. 2

[CHMY15] Tiago Cogumbreiro, Raymond Hu, Francisco Martins, and Nobuko Yoshida. Dynamic deadlock verification for general barrier synchronisation. In Albert Cohen and David Grove, editors, *Proc. 20th ACM Symp. on Principles and Practice of Parallel Programming, PPoPP 2015*, pages 150–160. ACM, 2015. 28

[DLPT14] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *TAAS*, 9(2):7, 2014. 118

[DNFP98] R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *Software Engineering, IEEE Transactions on*, 24(5):315–330, May 1998. 2, 11, 13, 35

[DPR00] Rocco De Nicola, Rosario Pugliese, and Antony I. T. Rowstron. Proving the correctness of optimising destructive and non-destructive reads over tuple spaces. In António Porto and Gruia-Catalin Roman, editors, *Coordination Languages and Models, 4th International Conference, COORDINATION 2000, Limassol, Cyprus, September 11-13, 2000, Proceedings*, volume 1906 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2000. 35, 50

[EGS06] Tarek El-Ghazawi and Lauren Smith. UPC: Unified parallel c. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. 5

[Erl16] Erlinda. Website for Erlinda: `https://code.google.com/archive/p/erlinda/`, 2016. 5

[FAH99] Eric Freeman, Ken Arnold, and Susanne Hupfer. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1st edition, 1999. 5

[FR10] Alan David Fekete and Krithi Ramamritham. Consistency models for replicated data. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, Lecture Notes in Computer Science, pages 1–17. Springer, 2010. 40, 42, 118

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, February 1992. 11

[Gel89] David Gelernter. Multiple tuple spaces in linda. In *Proceedings of the Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, PARLE '89, pages 20–27, London, UK, UK, 1989. Springer-Verlag. 7

[GH11] Stefan Gudenkauf and Wilhelm Hasselbring. Space-based multi-core programming in java. In Probst and Wimmer (PW11), pages 41–50. 66

[Gig17] Gigaspaces technologies ltd, `www.gigaspaces.com`, 2017. 66

[GL02] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. 1

[GMM12] Milos Gligoric, Peter C. Mehlitz, and Darko Marinov. X10X: model checking a new programming language with an "old" model checker. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *2012 IEEE Fifth International Conference*, pages 11–20. IEEE Computer Society, 2012. 28

[GN15] Suyash Gupta and V. Krishna Nandivada. Imsuite: A benchmark suite for simulating distributed algorithms. *J. Parallel Distrib. Comput.*, 75:1–19, 2015. 28

[GTC+11] David Grove, Olivier Tardieu, David Cunningham, Ben Herta, Igor Peshansky, and Vijay Saraswat. A performance model for x10 applications: What's going on under the hood? In *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*, X10 '11, pages 1:1–1:8, New York, NY, USA, 2011. ACM. 28

[Har12] Hariprasad Hari. *Tuple Space in the Cloud*. PhD thesis, Uppsala Universitet, 2012. 66

[HLH+11] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1199–1208, 2011. 8

[HW90] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. 42

[IBM17] IBM. Web site for X10: http://x10-lang.org/x10-community/publications-using-x10.html, 2017. 28

[IS14] Shams Imam and Vivek Sarkar. A Case for Cooperative Scheduling in X10s Managed Runtime. In *The 2014 X10 Workshop*, X10'14, June 2014. 94

[jRe17] jResp: Java Runtime Environment for SCEL programs. jResp in a nutshell: `http://jresp.sourceforge.net/?page_id=30`, 2017. 5

[JXLY06] Yi Jiang, Guangtao Xue, Minglu Li, and Jinyuan You. Dtupleshpc: Distributed tuple space for desktop high performance computing. In Chris R. Jesshope and Colin Egan, editors, *Advances in Computer Systems Architecture, 11th Asia-Pacific Conference, ACSAC 2006*, volume 4186 of *Lecture Notes in Computer Science*, pages 394–400. Springer, 2006. 66

[KH14] Vineet Kumar and Laurie J. Hendren. MIX10: compiling MATLAB to X10 for high performance. In Andrew P. Black and Todd D. Millstein, editors, *Proc. 2014 ACM International Conference, OOPSLA 2014*, pages 617–636. ACM, 2014. 28

[KS13] Peter Kogge and John Shalf. Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture. *Computing in Science and Engg.*, 15(6):16–26, November 2013. 4

[Lam97] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Computers*, 46(7):779–782, 1997. 33, 42

[LK14] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014. 8

[LP05] Zhen Li and Manish Parashar. Comet: a scalable coordination space for decentralized distributed environments. In *Second International Workshop on Hot Topics in Peer-to-Peer Systems, HOT-P2P 2005*, pages 104–111. IEEE Computer Society, 2005. 66

[LP10] Jonathan K. Lee and Jens Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN*, pages 25–36. ACM, 2010. 28

[LPC+12] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012)*, pages 265–278. USENIX Association, 2012. 118

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. 11

[MPA05a] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM. 32

[MPA05b] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 378–391, 2005. 35

[MPR06] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328, 2006. 66

[MZ09] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4), 2009. 66

[NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. 5

[OM15] Open-MPI. Web site for MPI: http://www.open-mpi.org/, 2015. 5

[PTA14] Jeeva Paudel, Olivier Tardieu, and José Nelson Amaral. Optimizing shared data accesses in distributed-memory X10 systems. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*, pages 1–10, 2014. 95, 118

[PW11] Christian W. Probst and Christian Wimmer, editors. *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*. ACM, 2011. 142, 144

[PWS+00] Fernando Pedone, Matthias Wiesmann, Andr Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, pages 464–474. IEEE Computer Society, 2000. 40

[PyL16] PyLinda. Website for PyLinda: `http://freecode.com/projects/pylinda`, 2016. 5

[Row97] Antony I. T. Rowstron. Using asynchronous tuple-space access primitives (BONITA primitives) for process co-ordination. In David Garlan and Daniel Le Métayer, editors, *Coordination Languages and Models, Second International Conference, COORDINATION '97*, volume 1282 of *Lecture Notes in Computer Science*, pages 426–429. Springer, 1997. 66

[RRV14] Bharath Ramesh, Calvin J. Ribbens, and Srinidhi Varadarajan. Regional consistency: Programmability and performance for non-cache-coherent systems. *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 00:941–948, 2014. 41

[RW96] Antony I. T. Rowstron and Alan Wood. An efficient distributed tuple space implementation for networks of workstations. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par '96 Parallel Processing, Second International Euro-Par Conference, Volume I*, volume 1123 of *Lecture Notes in Computer Science*, pages 510–513. Springer, 1996. 35, 50, 65

[SAB+10] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The asynchronous partitioned global address space model. Technical report, Toronto, Canada, June 2010. 20

[SBP+14] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification version 2.5, 2014. 20

[SDM11] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 1–25, Berlin, Heidelberg, 2011. Springer-Verlag. 5

[SJ05] Vijay A. Saraswat and Radha Jagadeesan. Concurrent clustered programming. In Martín Abadi and Luca de Alfaro, editors, *Proc. CONCUR 2005*, volume 3653 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2005. 28

[SKQ13] Jawwad Shamsi, Muhammad Ali Khojaye, and Mohammad Ali Qasmi. Data-intensive cloud computing: Requirements, expectations, challenges, and solutions. *J. Grid Comput.*, 11(2):281–310, 2013. 1

[SNA16] Snap datasets. Reference website: `https://snap.stanford.edu/data/`, 2016. 110

[SPBZ11] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, pages 386–400, 2011. 2

[SS05] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005. 1, 40

[SU15] T. Suzumura and K. Ueno. Scalegraph: A high-performance library for billion-scale graph analytics. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 76–84, Oct 2015. 111, 115, 118

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005. 33

[Ter13] Doug Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12):82–89, 2013. 118

[TKD+14] Adrian Tate, Amir Kamil, Anshu Dubey, Armin Größlinger, Brad Chamberlain, Brice Goglin, Carter Edwards, Chris J. Newburn, David Padua, Didem Unat, Emmanuel Jeannot, Frank Hannig, Tobias Gysi, Hatem Ltaief, James Sexton, Jesus Labarta, John Shalf, Karl Fürlinger, Kathryn O'Brien, Leonidas Linardakis, Maciej Besta, Marie-Christine Sawley, Mark Abraham, Mauro Bianco, Miquel Pericàs, Naoya Maruyama, Paul H. J. Kelly, Peter Messmer, Robert B. Ross, Romain Cledat, Satoshi Matsuoka, Thomas Schulthess, Torsten Hoefler, and Vitus J. Leung. Programming Abstractions for Data Locality. Research report, PADAL Workshop 2014, April 28–29, Swiss National Supercomputing Center (CSCS), Lugano, Switzerland, November 2014. 1

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. 40

[UNZ+16] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, and John Shalf. *TiDA: High-Level Programming Abstractions for Data Locality Management*, pages 116–135. Springer International Publishing, Cham, 2016. 1

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. 101

[vdG01] R. van der Goot. *High Performance Linda using a Class Library*. PhD thesis, Erasmus University Rotterdam, 2001. 66

[Vog09] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. 41

[VV16] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1):19:1–19:34, June 2016. 118

[WPC16] Jordi Wolfson-Pou and Edmond Chow. Reducing communication in distributed asynchronous iterative methods. *Procedia Computer Science*, 80:1906 – 1916, 2016. International Conference on Computational Science 2016, {ICCS} 2016, 6-8 June 2016, San Diego, California, {USA}. 5

[X1017] The X10 programming language website. X10 - Performance and Productivity at Scale: `x10-lang.org`, 2017. 2

[Xte16] Xtext. Website for Xtext: `http://www.eclipse.org/Xtext/`, 2016. 29

[YSP+98] Katherine A. Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul N. Hilfinger, Susan L. Graham, David Gay, Phillip Colella, and Alexander Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998. 5

[Zwi16] Andreas Zwinkau. A memory model for X10, To appear in X10 workshop procedings, 2016. 36