Replicating Data for Better Performances in X10

Marina Andrić[1], Rocco De Nicola[1], and Alberto Lluch Lafuente[2]

[1] IMT Institute for Advanced Studies Lucca, Italy
[2] DTU Compute, Technical University of Denmark, Denmark

Abstract. Linguistic primitives for replica-aware coordination offer suitable solutions to the challenging problems of data distribution and locality in large-scale high performance computing. The data replication mechanisms that had previously been designed to extend KLAIM with replicated tuples are now used to experiment with X10, a parallel programming language primarily targeting clusters of multi-core processors linked in a large-scale system via high-performance networks. Our approach aims at allowing the programmer to specify and coordinate the replication of shared data items by taking into account the desired consistency properties. The programmer can hence exploit such flexible mechanisms to adapt data distribution and locality to the needs of the application, in order to improve performance in terms of concurrency and data access. We investigate issues related to replica consistency and provide a performance analysis, which includes scenarios where replica-based specifications and relaxed consistency provide significant performance gains.

1 Introduction

Parallel and distributed computing systems are more and more frequently used to solve complex computational problems. Now, when more computing power is needed, one does not buy a faster uniprocessor but another processor or another million processors, and connects them with a high-speed communication network. Or, perhaps, one rents them instead, by resorting to cloud computing services. This gives one whatever number of computer cycles he can desire but poses the problem of how to use those computer cycles effectively by dividing the available work into chunks that can be executed simultaneously without introducing undesirable indeterminacy or waiting for conditions that may never materialize.

One of the key issues in parallel and distributed computing is the partitioning and exchange of data between computational entities. Better performances are achieved with increased data locality and minimized data communication.

Increasing data locality can be achieved by replicating data, but this comes at a high price in terms of synchronization in case replicated data need to be kept consistent. As a matter of fact the trade-off between consistency and performance is one of the big dilemmas in distributed and parallel computing and is one of the main topics of research of the High Performance Computing (HPC) community.

The recent years have seen the advent of technologies that provide software engineers and programmers with flexible mechanisms to conveniently specify data locality, communication and consistency to the benefit of their applications.

A pragmatical example for large-scale distributed services is the Google Cloud Storage [10] service, that allows users to geographically specify data locality (to reduce cost and speed up access) and provides different consistency levels (e.g. strong and eventual consistency) for different operations (e.g. single data and list operations). Indeed, many modern distributed systems are based on *optimistic data replication* techniques for achieving high availability and performance (see e.g. the discussion in [2]). In such systems it is vital for the programmer to know when consistency can be sacrificed for the sake of performance without compromising the application's expected functionality. One guidance for common weak memory models in distributed computing can be found in [7].

One response to this problem has been to move to a fragmented memory model. Multiple processors are programmed largely as if they were uniprocessors, but are meant to interact via message-passing middlewares such as MPI [15]. One disadvantage is that programmers must explicitly manage the interaction between multiple processes and coordinate their data exchange; large data-structures that are conceptually unitary must be thought of as fragmented across different nodes. The Partitioned Global Address Space (PGAS) model has then been proposed, see. e.g. Titanium [17], to permit the programmer to think of a single computation running across multiple processors, sharing a global address space and relying on zone-based memory management. All data resides at some processor, which is said to have affinity to the data. Each processor may operate directly on the data it contains but must use some indirect mechanism to access or update data at other processors. Some kind of global barriers are used to ensure that processors remain synchronized. More recently a new language, X10, has been proposed that can be considered as one of the first member of the second generation of PGAS languages. It extends the PGAS model with asynchrony (yielding the APGAS programming model) by introducing the notion of *places* as an abstraction for a computational context with a locally synchronous view of shared memory. An X10 computation runs over a large collection of places. Each place hosts some data and runs one or more activities. Activities can be dynamically created. Activities are lightweight threads of execution. An activity may synchronously (and atomically) use one or more memory locations in the place in which it resides.

This programming model facilitates the development of distributed applications having a body of data which is shared between a few or all components. Such data can range from simple variables to large arrays, structured types or multimedia objects. To reduce the number of accesses to a single point in the system, programmers often do decompose such large objects into sub-parts, which are then distributed and processed in parallel, or move a copy of the shared data to the sites that use it, thus forming local replicas at each site.

Data locality and data consistency are indeed two key aspects in the design of distributed and parallel systems and software. A proper design of those aspects

can bring significant performance advantages, e.g. in terms of minimization of communication between computational entities. In our view, data locality and data consistency issues cannot be fully hidden to the programmer of the high performance applications of the future. Programmers should be equipped with suitable primitives to deal with those aspects in a natural and flexible way. Early works presented in [6] and [9] pointed to the importance of this aspect and developed a theory of sharing which captures the behavior of programs with respect to shared data into the framework of process algebra. The core theory can describe programs performing read and write access to unitary pieces of shared data. Extensions allow shared data to be decomposed and atomic copies to be made, reflecting the common operations of parallel programs. The authors tackled the problem of decomposition strategies of shared data, from the performance perspective, and replication of commonly-read state.

Our contribution to this approach, applied to the distributed tuple space coordination paradigm, was recently presented in [1]. We introduced RepliKlaim, a tuple-based coordination language which enriches the Klaim language [5] with primitives for replica-aware coordination, in order to offer suitable solutions to the challenging problems of data distribution and locality in large-scale high performance computing. In particular, RepliKlaim allows the programmer to specify and coordinate replication of shared data items and the desired consistency properties. The programmer can hence exploit such flexible mechanisms to adapt data distribution and locality to the needs of the application, in order to improve performance in terms of concurrency and data access. We provided also a performance analysis, which includes scenarios where replica-based specifications and relaxed consistency provide significant performance gains.

In this work we describe our initial attempt at exporting our approach to X10, a general purpose object-oriented, scale-out programming language. The main motivation for turning our attention to X10 are its similarities with Klaim. Indeed, both languages consider localities as a first-class citizen and offer primitives for asynchronous parallel computations and code mobility.

We hope that the results of our preliminary work are sufficiently interesting to stimulate research on X10 aiming at adding to the language specific primitives or libraries would enable programmers to easily manipulate replicated data while choosing the appropriate level of consistency.

Structure of the paper. The rest of the paper is organised as follows. Section 2 introduces X10, by providing an overview of the underlying programming model and by presenting the basic features of the language through small examples. Section 3 reports on a number of performance experiments by making different assumptions on the size of the data and on the number of available processing units. In section 4 we draw conclusions and sketch possible directions for future work.

2 X10 in a nutshell

X10 is a programming language primarily targeting clusters of multi-core processors linked in a large-scale system via a high-performance network, consequently concurrency and distribution are the main focus of the language design. The design philosophy of X10 is based on a belief that future server systems will consist of multi-core SMP nodes with non-uniform memory hierarchies interconnected in scalable clusters referred to as *Non-Uniform Cluster Computing* (NUCC) systems. The goal of the designers of X10 was to create a language that would combine ease of programming of object-oriented languages and efficiency of high-performance languages. Using the words of the designers, their goal was *"to increase programmer productivity for NUCC without compromising performance"*.

The programming model of X10 is called *(asynchronous) partitioned global address space*, i.e. (A)PGAS. The PGAS model combines data locality (partitioning) of a distributed memory model and global address space of a shared memory model. In PGAS each processor has private memory for local data and shared memory for globally shared data. APGAS enriches the PGAS model with two additional concepts: *places*, which provide an explicit mechanism for data and code locality, and *asynchronous invocation*, which allows forking a task, possibly at a remote place. These two notions are reminiscent of the locality/node concept and of the eval operation in Klaim [5], where the command `eval(S)@l` is used to spawn a new process at locality `l` to remotely execute `S`. The Klaim command `eval(S)@self` is instead used to execute `S` locally. As a matter of fact, these similarities between X10 and Klaim have inspired our interest in investigating the transfer of our approach from replica-aware programming [1] to X10.

The code snippet written in X10 below (Listing 1.1) presents a slight simplification and adaptation of the case study used in our experimental evaluation. We shall use this simplified version of our case study in the rest of the section to introduce some key ingredients of X10, necessary to understand our work.

Listing 1.1. GlobalRef usage

```
1  var a:A = new A();
2  val y = GlobalRef[A](a);
3  val places = Place.places();
4
5  at(places(0)) async {
6      atomic y().update();
7  }
8
9  at(places(1)) async {
10     val temp = at(y.home) y().getData();
11 }
12
13 at(places(2)) async {
14     at(y.home) atomic y().update();
15 }
```

A variable `a`, actually an object of class `A`, is going to be created in a place (`0`) and shared with two other places (`1` and `2`), through a global reference `y`. Parallel computations at the three places perform different operations on `a` (through the reference `y`): an (atomic) update (local in place `0` and remote in place `2`) and a remote read (in place `1`).

As one can observe in the example, data items in X10 can be mutable (`var`, e.g `a`) or immutable (`val`, e.g. the reference `y`). The set of places is fixed before program execution. Places cannot be dynamically created in the current version of X10. 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 numbers. Each X10 place is indented to map to a hardware data-coherent unit, such as an SMP node in a multi-core machine. Functions are first-class data and as such they can be stored, passed between activities and so on. X10 provides several primitives for coordinating access to shared mutable data. Among the others we would like to mention `atomic` blocks. Specifically, `atomic S` is used to guarantee execution of a statement `S`, following certain restrictions, as if it was a single step, with respect to other concurrently executing atomic blocks in the same place. It is used for the update in the example above to avoid race situations.

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` control comes back to the current place, with the result of `S`. One needs to be careful when using the `at` construct as it can potentially lead to high costs as the objects used in `S` (and depending objects) are copied to place `P`. This behavior can be altered by using global references (`GlobalRefs`) as we do in our example, which we will explain further below.

Parallelism across places can be 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`. This is used in our example to spawn the parallel remote activities on places `1` and `2`. To synchronize activities one of the mechanisms offered by X10 is a `finish S` construct. An activity that executes `finish S` will execute `S` and then wait for all the activities spawned by `S` to terminate.

It is worth mentioning that the activities running in a place may access (read, modify) data items located at that place with the efficiency of on-chip access. Accesses to remote places can be significantly longer, sometimes even orders of magnitude longer, as we will see in Section 3.

As we have already mentioned, careless use of `at` can result in copying and transmitting very large data structures. In order to avoid this copying, one has

to create and use global value references `GlobalRefs`. In particular, `val ref = GlobalRef[T](v)` creates a reference to a value `v` of type `T` and stores it in `ref`. Retrieval of a value is done by operation `ref()`. In such a way, manipulating data with references across different places will not involve copying, however operating on referenced values requires a place-shift to the home place of the reference, that is obtained with `ref.home`.

To illustrate this important programming concepts, we present in the code snippet below (Listing 1.2) a "wrong" variant of our previous example (i.e. the one on Listing 1.1):

Listing 1.2. Value copying

```
1  val a:A = new A();
2  val places = Place.places();
3
4  at(places(0)) async {
5      atomic a.update();
6  }
7
8  at(places(1)) async {
9      val temp = a.getData();
10 }
11
12 at(places(2)) async {
13     atomic a.update();
14 }
```

Contrary to the previous example, no global reference is used to operate on variable `a`. The effect is that all places will operate on *local* copies of `a`, possibly introducing unwanted inconsistencies. This is due to the already explained data copying that the `at` construct entails.

As a final example, consider the following semi-formal X10 specification, which permits implementing, in a programmed manner, the kind of data replication we promote in our work:

Listing 1.3. Program replicas

```
1  val places = Place.places();
2
3  val lock = new Lock();
4  val lockRef = GlobalRef[Lock](lock);
5
6  val replicas : DistArray[A] =
7          DistArray.make[A](Dist.makeUnique(places), (Point)=> new A());
8
9  for (q in places) at(q) async {
10         dataAccess();
11 }
```

To replicate an object of class `A` we use X10's inbuilt distributed array class, `DistArray`, that represents a generic multidimensional array distributed over multiple places. There are various strategies available for initializing such array. In this case we choose the *unique* distribution, which stores one data element

(`Point`) per place in a designated region. In order to replicate an instance of class `A` across all the available places we initialized each `Point` to an instance of class `A` and region to the set of all available places in the execution (`line 6`).

All places perform the same kind of access to the data in parallel, specified by function `dataAccess`:

Listing 1.4. Data access function

```
 1  for (var i:Long = 0; i < NUM_AC; i++) async {
 2          with probability p update{
 3                  at(lock.home) lock().take();
 4                  for (r in places) at (r) async {
 5                          replicas(r.id()).update();
 6                  }
 7                  at(lock.home) lock().release();
 8          }
 9          with probability 1-p read{
10                  val temp = replicas(q.id()).getData();
11          }
12  }
```

In this model, each actual 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, with a pre-defined probability `p`. Update access is performed in a way that *all* replicas are updated to ensure consistency (`lines 4-6`). Of course, such an update of all replicas can follow different strategies. For instance, one can aim at strong consistency or weak consistency with the use of appropriate locks (as shown above) or, else, one can execute the updates in sequential order or as parallel activities (as we do above). A `lock` variable is used to *synchronize* data accesses. The read access is simply performed against a local replica (`line 10`).

As we will show in Section 3, we tune 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. The example above is instrumental to convey our main idea: if updates are infrequent with respect to reads, then replicating data in X10 specifications yields more performant applications.

Due to the limited space, we have focused here on the main X10 constructs and concepts that are relevant to understanding the experiments we performed. X10 is still under development at IBM in collaboration with academia. There are two runtime frameworks available, `Native X10` and `Managed X10` that are respectively based on `C++` and `Java` backends. The semantics of the language has been formalized in [16] along with a resilient version [4]. A core calculus with X10's main constructs for parallelism async and finish is presented in [14]. Cogumbreiro et al. developed Armus [3], a verification tool that detects barrier deadlocks for Java and X10 programs. Gligoric et al. attempted to develop a model checking tool [8] for X10 based on the Java Path Finder tool for model checking Java programs. A line of work focuses on compiling and porting

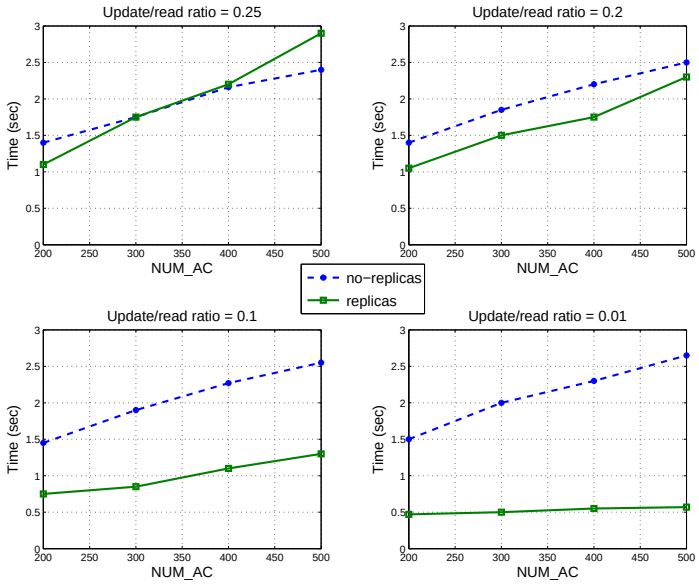


Fig. 1. (`Ratio`): The two strategies with shared data of size $\approx$ 3`MB`

programs to X10, specifically, [13] reports on compiling Matlab to X10 for high performance computing. The work in [11] 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 [12].

3 Experiments

In this section we describe the practical experiments that we have performed in X10 in order to support the claim that explicit use of replicas can provide significant performance improvements. We present a number of examples, discuss the implemented replica consistency protocol, and conclude by analyzing the obtained results.

Hypothesis. As we have already stated in the Introduction (Section 1) the main motivation behind our experiments is to show 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 will 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

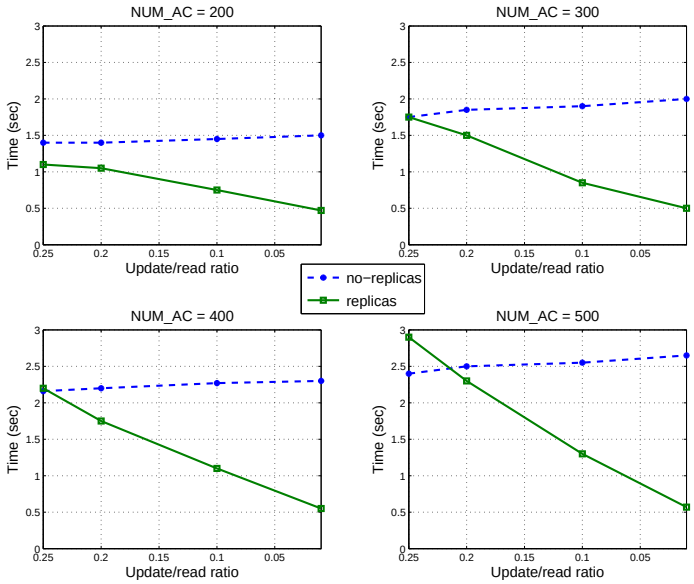


Fig. 2. (`Access number`): The two strategies with shared data of size ≈ 3`MB`

replica. We have 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 and the one with *replicas*.

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`. The full implementation of our case studies is available for download at `http://sysma.imtlucca.it/wp-content/uploads/2015/05/Source_X10_example.rar`.

Experiments: Configuration of the Scenario. The main idea of the scenario we have tested is that concurrent activities running across multiple places are operating (performing read and update accesses) on the same piece of data, which is considered to be *shared* data between a number of places. We compare performances of two variants which we refer to as no-replicas and replicas. The essence of the program with replicas has been already introduced through examples 1.3 and 1.4. In contrast to the replicated variant, the non-replicated one excludes

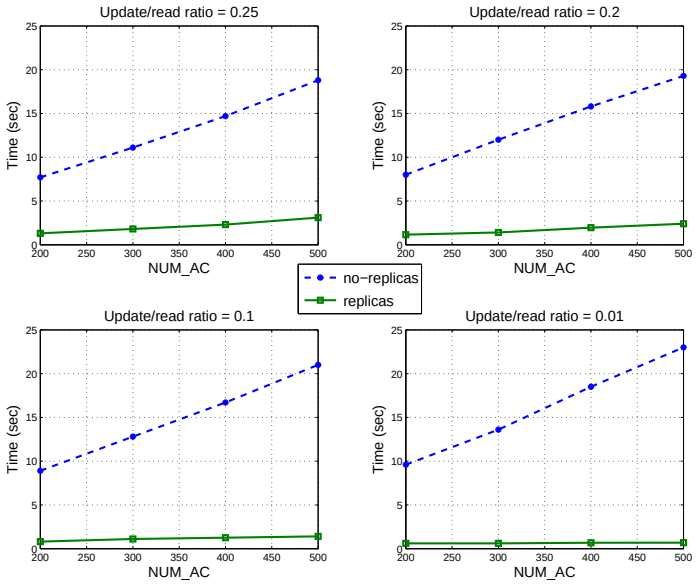


Fig. 3. (`Ratio`): The two strategies with shared data of size ≈ 30`MB`

creation of replicas, hence every access is directed towards a single centralized data variable, as promoted in example 1.1.

As we have already mentioned, to give 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 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 ≈3`MB` and ≈30`MB` in the other.

The two strategies (programs no-replicas and replicas) that we compare are described as follows.

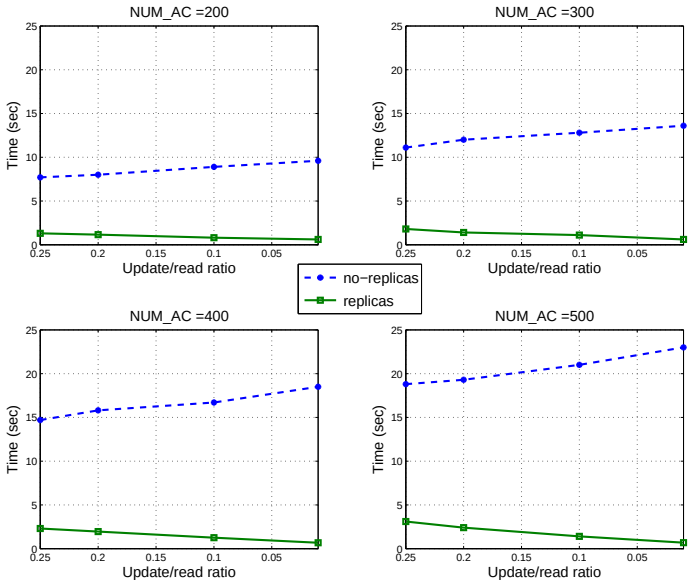


Fig. 4. (`Access number`): The two strategies with shared data of size $\approx$ 30MB

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 place-shifting.

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*. 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 not allowed, as this would lead to undesirable and unpredictable results. We forbid such behavior by means of a synchronization lock, that is acquired before performing the update operations, and released at the end (see Listings 1.3 and 1.4).

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. 1, 3) or the number of accesses `NUM_AC` (Fig. 2, 4), 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. 1 and 2 correspond to results obtained for size of shared data of $\approx$ 3MB, while Fig. 3 and 4 correspond to results obtained for the size of $\approx$ 30MB.

Here we present initial results obtained for a 4 and 8 places scenario, we plan to extend the experiments in the future to 16 and larger number of places.

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.

The results obtained for 8 places can be found in Appendix. 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 5(c)) or the update/read ration is small enough (Fig. 5(a)).

We have to add that our initial attempts to scale the experiments to 16 places failed at runtime with a "`Place(0): TOO MANY THREADS`" error. After a first, superficial, analysis, we come to the conclusion that the problem is mainly due to a centralized lock variable and to the large number (more than a thousand) of activities competing for it. This should have created congestion at home place of the lock i.e. place 0. In the future, we shall devise different models and experimental settings that take this issue into account.

4 Conclusions

Performance-vs-consistency is an inherent and classical dilemma in distributed and parallel computing, from local highly parallel systems (e.g. a multi-core machine) to widely distributed concurrent systems (e.g. a world-spread data center). The resolution of such a dilemma is often delegated to run-time frameworks and middlewares and is hidden to programmers. For some applications, however, programmers would benefit from having some control on such design decisions which significantly define the user-perception on the application's Quality-of-Service.

We are investigating programming abstractions for dealing with a key instrument in the performance-vs-consistency dilemma, namely replication of data. In a first stage of our investigation [1], we focused on a language mainly targeted at largely distributed systems, and we proposed RepliKlaim, an extension of the Klaim language [5], with the notion of replicated tuples and with specific communication operations for dealing with them.

In this paper we have tried to apply the lessons we learned when considering Klaim to highly parallel systems. In particular, we have focused on X10, a language for high-performance computing that shares with Klaim a couple of important features such as importance assigned to explicit localities and to code mobility. Like Klaim, the language X10 follows the APGAS programming model which allows for remote operation on shared data, possibly involving transfer and local replication of data. We have performed experiments similar in spirit to those we presented in [1], comparing different strategies for operating on shared

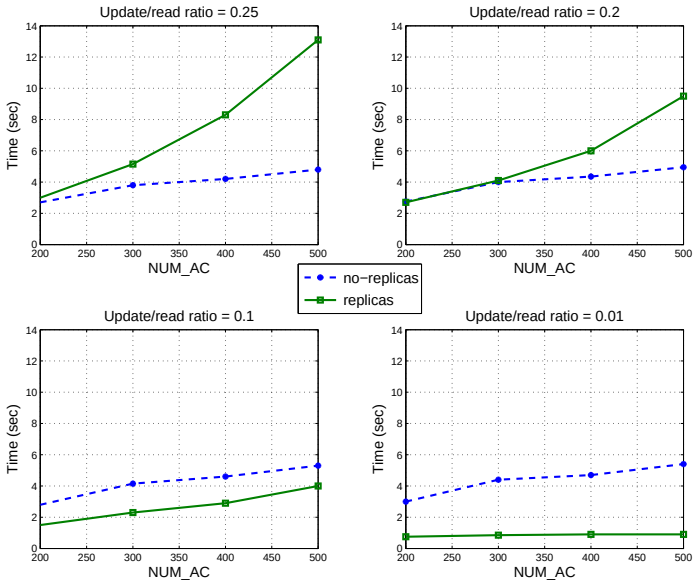
data. The results we have obtained show the benefit of replicating data in specific scenarios, especially when the size of shared data is very large.

Our main goal with these investigations is to identify suitable programming abstractions for dealing with replicated data in high-performance applications. In [1], for example, we proposed a primitive $\texttt{out}_\alpha\texttt{(t)@P}$, to specify an operation that places replicas of a tuple t in all places P using the consistency level α (either weak or strong). Similarly, one could conceive convenient X10 constructs like e.g. `share X with P` to specify that the data item `X` is meant to be shared with the set of places P, and additional features to specify the level of consistency (e.g. weak, strong) desired when invoking methods on `X`. We do not necessarily advocate that programming languages like X10 should be equipped with first-class primitives supporting those abstractions. In many cases, suitable macros or libraries can be sufficient to provide programmers with mechanisms to specify and control data replication in a natural and disciplined manner.

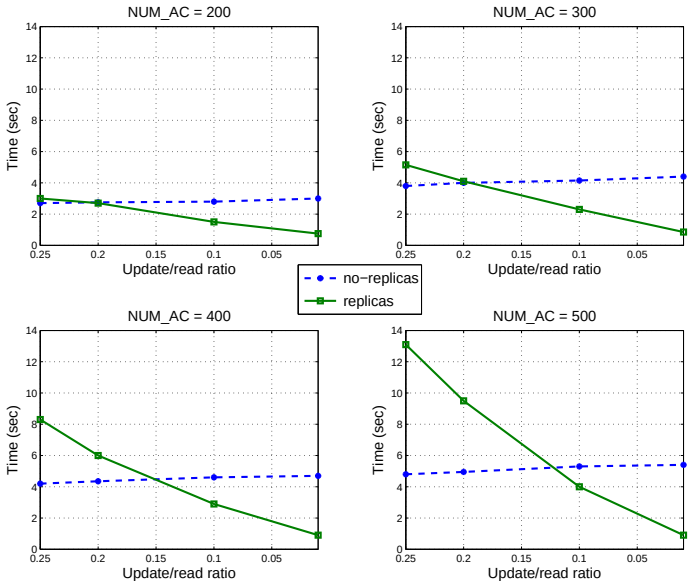
As future work, we plan to introduce scalability tests, by extending the current framework to consider a larger number of places and of CPU cores. We shall consider both situations where more than one place is hosted on each `CPU` core and the issues of lock variables on shared data that prevented us from carrying out the planned experiments for 16 cores.

We are particularly interested in considering different (i.e. weaker) consistency models and implementations for obtaining such models. As we stated in the Introduction (Section 1) one of the most popular consistency models used in practical applications is *eventual consistency*. It is argued that this model is the weakest that can be accepted. In the present model, we restrict concurrent updates to replicas by means of a lock, while this is not the case when looking only for eventual consistency.

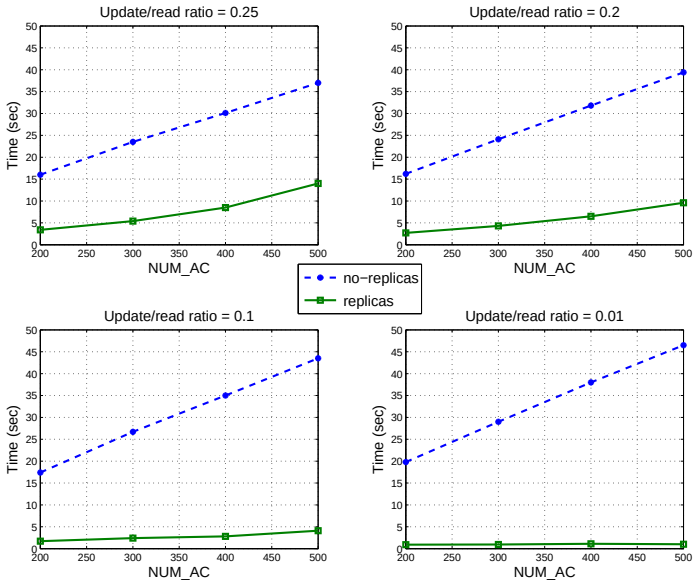
A Results for eight-places scenario



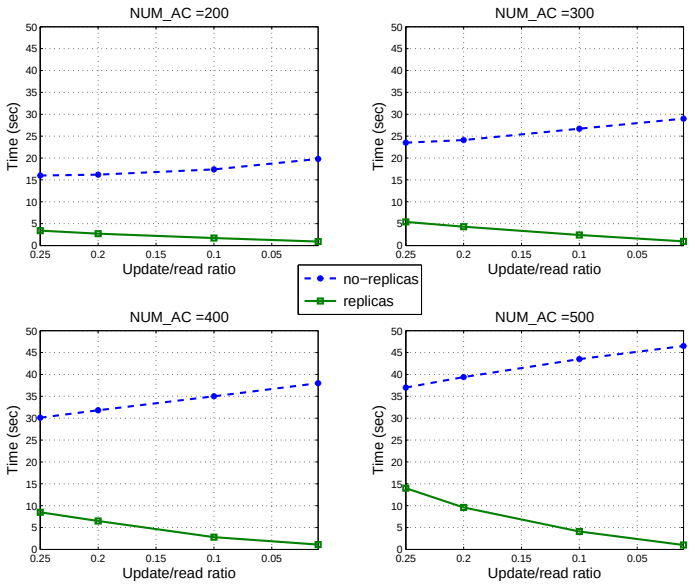
(a) (`Ratio`): The two strategies with shared data of size ≈ 3MB



(b) (`Access number`): The two strategies with shared data of size ≈ 3MB



(c) (`Ratio`): The two strategies with shared data of size $\approx$ 30`MB`



(d) (`Access number`): The two strategies with shared data of size $\approx$ 30`MB`

Fig. 5. Scenario with 8 places

References

1. M. Andric, R. De Nicola, and A. Lluch-Lafuente. Replica-based high-performance tuple space computing. In T. Holvoet and M. Viroli, editors, *Proc. COORDINATION 2015 - Coordination Models and Languages*, volume 9037 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2015.
2. A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *Proc. 41st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '14*, pages 285–296, 2014.
3. T. Cogumbreiro, R. Hu, F. Martins, and N. Yoshida. Dynamic deadlock verification for general barrier synchronisation. In *Proc. 20th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP 2015*, pages 150–160, 2015.
4. S. Crafa, D. Cunningham, V. A. Saraswat, A. Shinnar, and O. Tardieu. Semantics of (resilient) X10. *CoRR*, abs/1312.3739, 2013.
5. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *Transactions on Software Engineering*, 24(5):315–330, 1998.
6. S. A. Dobson and C. P. Wadsworth. Towards a theory of shared data in distributed systems. In I. Jelly, I. Gorton, and P. R. Croll, editors, *Software Engineering for Parallel and Distributed Systems*, volume 50 of *IFIP Conference Proceedings*, pages 170–182. Chapman & Hall, 1996.
7. A. D. Fekete and K. Ramamritham. Consistency models for replicated data. In *Replication: Theory and Practice*, pages 1–17, 2010.
8. M. Gligoric, P. C. Mehlitz, and D. Marinov. X10X: model checking a new programming language with an "old" model checker. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, Montreal, QC, Canada, April 17-21, 2012*, pages 11–20, 2012.
9. D. Goodeve, S. Dobson, J. Nash, J. Davy, P. Dew, M. Kara, and C. P. Wadsworth. Towards a model for shared data abstraction with performance. *Journal of Parallel and Distributed Computing*, 49, 1998.
10. Google-Storage. Web site for Google Storage: https://cloud.google.com/storage/, 2015.
11. S. Gupta and V. K. Nandivada. Imsuite: A benchmark suite for simulating distributed algorithms. *J. Parallel Distrib. Comput.*, 75:1–19, 2015.
12. IBM. Web site for X10: http://x10-lang.org/x10-community/publications-using-x10.html, 2015.
13. V. Kumar and L. J. Hendren. MIX10: compiling MATLAB to X10 for high performance. In *Proc. 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014*, pages 617–636, 2014.
14. J. K. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPOPP 2010*, pages 25–36, 2010.
15. Open-MPI. Web site for MPI: http://www.open-mpi.org/, 2015.
16. V. A. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *Proc. CONCUR 2005 - Concurrency Theory*, pages 353–367, 2005.
17. K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.