

Experiences with Implementing Task Pools in Chapel and X10

Claudia Fohry¹(✉) and Jens Breitbart²

¹ Research Group Programming Languages/Methodologies, University of Kassel,
Kassel, Germany

`fohry@uni-kassel.de`

² Engineering Mathematics and Computing Lab (EMCL), Heidelberg University,
Heidelberg, Germany

`jens.breitbart@iwr.uni-heidelberg.de`

Abstract. The Partitioned Global Address Space (PGAS) model is a promising approach to combine programmability and performance in an architecture-independent way. Well-known representatives of PGAS languages include Chapel and X10. Both languages incorporate object orientation, but fundamentally differ in their way of accessing remote memory as well as in synchronization constructs and other issues of language design.

This paper reports on and compares experiences in using the languages. We concentrate on the interplay between object orientation and parallelism/distribution, and other issues of coding task parallelism. In particular, we discuss the realization of patterns such as objects that internally contain distributed arrays, and suggest improvements such as support for activity-local and place-local data, as well as scalar variable-based reduction. Our study is based on Unbalanced Tree Search (UTS), a well-known benchmark that uses task pools.

Keywords: Chapel · X10 · PGAS · UTS · Task pool

1 Introduction

The goal of high-productivity parallel programming has led to the Partitioned Global Address Space (PGAS) programming model and its concretization in a number of languages/systems such as Chapel and X10. PGAS languages expose to the programmer a shared memory that is split into disjoint partitions. Each partition comprises distinct computing resources that have faster access to local than to remote memory.

Chapel and X10 have similar goals and a similar level of maturity, but differ in many other aspects. Chapel was introduced by Cray, and X10 by IBM, both at the beginning of this century and with funding from the DARPA “High Productivity Computing Systems” project. The paper refers to Chapel version 1.8.0, and X10 version 2.4. A brief survey of the languages is given in Sect. 2.

We base our comparison on Unbalanced Tree Search (UTS), a benchmark for studying issues such as load balancing or characteristics of the implementation language [6]. Unlike previous Chapel and X10 implementations of UTS [3, 13], our programs use both multiple nodes and multiple threads per node. UTS is implemented with task pools, deploying a fixed number of long-running workers and a distributed data structure for mutual access. The benchmark and this setting were selected to study communication, synchronization, and the interplay between object-orientation and parallelism/distribution in a simple framework. They may set Chapel at some disadvantage by not appreciating its rich set of built-in data parallel features.

Our focus is expressiveness of the languages, moreover some preliminary performance numbers are included. We report on our experiences in deploying the available language constructs, discuss variants for coding common patterns such as objects that internally contain distributed arrays, and suggest improvements such as support for activity-local and place-local data as well as scalar variable-based reduction. Moreover, we touch on diverse issues such as objects vs. records and locality optimization.

The paper starts with background on Chapel, X10, task pools and UTS in Sect. 2. That section also details the task pool variant that we selected. Section 3 gives an overview of our implementations, and then organizes language assessment along various topics. Thereafter, Sects. 4, 5 and 6 are devoted to performance, related work and conclusions, respectively.

2 Background and Benchmark

2.1 Chapel

The following introduction is by necessity brief, for further information see [2].

A Chapel program runs on some number of *locales*, each of which comprises processors and a memory partition. The `on` statement places a code block on a particular locale. Within the block, all variables in scope may be read and written, although accesses to remote locales are more expensive.

Parallel tasks are created with, e.g., `begin` or `coforall`. Task creation can be combined with code placement, for an example see Sect. 3.4. Chapel does not expose threads, but tasks are transparently mapped to threads by a configurable tasking layer.

In Chapel, synchronization is almost exclusively based on synchronization variables, which are declared with type qualifier `sync` or `single`. The former hold a value of some primitive type, and additionally have state full or empty. A write to a full `sync` variable blocks the calling task, as does a read from an empty one. When the variable changes state, one of the waiting tasks may proceed.

The base language has C-like syntax. Arrays are defined over multidimensional domains and may be distributed, e.g. blockwise or cyclic. Constants may be marked as configurable, in this case they may be overwritten at the command line.

Chapel programs are composed of modules, which contain data, functions, classes etc. Classes have the usual functionality, including constructors, inheritance, nesting, and generics. Records resemble classes, but variables of this type directly hold the values of all fields.

2.2 X10

Many Chapel concepts have an analogue in X10 [17], except for using different terminology:

Chapel	locale	domain	on	task	begin	record	sync statement
X10	place	region	at	activity	async	struct	finish

The concrete realization differs, e.g. X10 structs are less flexible than Chapel records. Unlike in Chapel, variables may only be accessed if they are stored at the current place, are globally accessible through a `GlobalRef`, or have been copied with `at`.

`at` copying rules are complex, yet in general single-assignment variables (specified with `val`) are copied, whereas normal variables (specified with `var`) are not. The X10 standard library supports place-local data, which are accessed through a `PlaceLocalHandle` that may be communicated and resolved at any particular place.

The major synchronization construct, `atomic`, encloses a critical section and operates intra-place. All read and write accesses to shared variables must be protected, and all critical sections at a place are mutually exclusive.

In many respects, the base language resembles Java. We used the most elementary type of arrays, called rails. There is no equivalent of a Chapel module, but classes may have static fields, and support inheritance, nesting, and generics.

2.3 Task Pools

Many irregular applications are composed of sub-computations (tasks) that vary in size. Task pools are a well-known pattern to map these tasks to execution resources at runtime, and thereby achieve load balancing. Task Pools may be implemented in either the user program or the runtime system, and come in various forms. In the following, we only describe the variant that we implemented, which resembles the one in [10]. Note that the task pool literature uses the term task different from PGAS languages. To avoid confusion, we denote the execution resources, which will correspond to X10 activities or Chapel tasks, as workers.

A task pool is a data structure from which idle workers repeatedly take a task, compute it, possibly insert new tasks, take the next task etc., until the pool is empty. The data structure is distributed. Each worker maintains a split queue [10], which is a kind of circular buffer that comprises a private and a public

portion. It is double-ended, such that **head** denotes the first free position of the private part, and **tail** the last filled position of the public part. The elements in-between **head** and **tail** are divided into **nprivate** elements in the private pool, followed by **npublic** elements in the public pool.

The task pool is accessed by **push** and **pop** operations: **push** inserts a task at position **head**, and **pop** takes a task out from the same end. If the private pool holds $2k$ elements, for some constant k , **push** additionally **releases** k elements to the public pool. Analogously, if **pop** discovers an empty private pool, it **acquires** k elements from the public pool. Operations **acquire** and **release** do not move tasks, but shift the division line between the private and public portions. Synchronization is required for the public pool only.

When **acquire** fails, the worker tries to steal k tasks from some other worker. Therefore, it first cycles through all workers of its own place, and then through those of the others. When no victim is found after one global cycle, the worker terminates. Termination detection is actually more complex [11], but we rely on the simple scheme for brevity.

2.4 UTS

A task pool may either be provided as a reusable component, e.g. by a library, or be used as a pattern to implement a particular algorithm. We considered the second scenario with the Unbalanced Tree Search (UTS) benchmark [6].

UTS consists in extracting a tree and counting the number of nodes. For given tree shape parameters, a node holds all information about the subtree rooted in it, and thus may be deleted after having been expanded. The information is encoded in a 20-byte node descriptor, using some cryptographic method. Naturally, a task corresponds to the expansion of one node, and is represented by this node descriptor.

Open-source implementations of UTS are available for various systems, among them Chapel and X10 [3, 6, 13–15]. They will be discussed in Sect. 5. We reused parts of the implementations [3, 15], chiefly the respective native interfaces to the C cryptographic tools, and the deployment of place local handles from [15]. Our own code can be obtained from the first author’s homepage.

3 Language Assessment

3.1 Overview of Implementations

In both languages, we implemented UTS with the task pool variant described in Sect. 2.3. Our implementation uses both multiple places and multiple activities per place. Workers are realized by long-running activities/Chapel tasks that are started after the task pool has been filled with initial tasks. The functional components of all program variants are similar:

- setting parameters of the tree
- generating and initializing the distributed data structure for the task pool

- expanding the root and inserting initial tasks into the task pool
- managing the workers that process the tasks
- managing the split queues with **push**, **pop**, **release** and **acquire**
- realizing the **steal** operation, including definition of the cyclic order and moving tasks to another place
- computing the result number of nodes by reduction
- invoking C functions for initializing the root and decoding node descriptors.

The X10 standard library includes basic local data structures, but neither Chapel nor X10 provide split queues, which therefore had to be implemented manually. For simplicity, we assume that the public pools never overflow.

3.2 Object-Orientation and Parallelism

The Encapsulation Problem. In object-oriented programming, data structures are often coded as classes, such that each instance of the class represents an instance of the data structure. The reference to this instance is stored in a variable, and operations are invoked by method calls on this variable. Thus, the variable provides a single access point to the data structure, abstracting away all details of the internal representation at the caller site:

```
var s: Stack = new Stack();
s.push(elem);
```

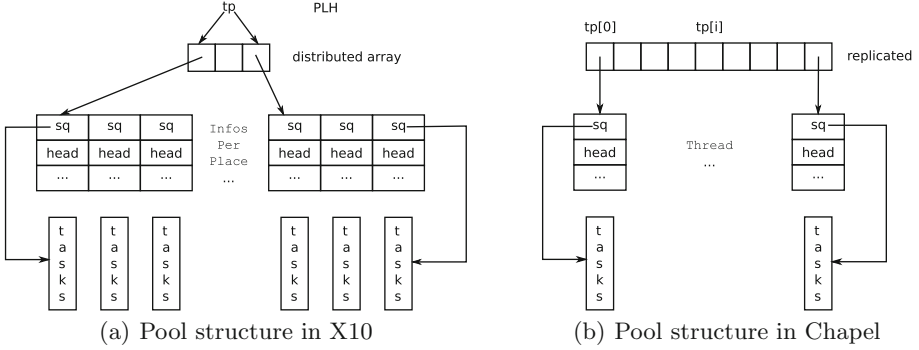
When the data structure is distributed and accessed from different places, such as our task pool, there is currently no equivalent for this convenient notation, since a single access point `s`, located in a single place, hurts performance.

To solve the problem, we distributed the data structure across places and addressed the local portions. We implemented several variants of this “distributed-first” approach, which all share the drawback that they provide less encapsulation than the above “objects-first” approach. In particular, they do not hide, at the caller site, the fact that the task pool is distributed. Thus, a programmer must decide between either the well-structured but inefficient “objects-first”, or the more efficient but less modular “distributed-first” approach.

X10 Implementation. We start the discussion of “distributed-first” variants with X10, since X10 provides some library support with its `PlaceLocalHandle` (PLH). As explained in Sect. 2.2, a PLH supports access to place-local data, and thus simplifies addressing the local portion when invoking task pool operations. Nevertheless, declaration and initialization of the distributed structure remain the responsibility of the user:

```
val tp = PlaceLocalHandle.make[InfosPerPlace](...)
```

In this code fragment, `InfosPerPlace` is a user-defined class which, as illustrated in Fig. 1(a), chiefly contains a rail of split queues. As can be seen in the figure,

**Fig. 1.** Implementation variants

use of the PLH requires a two-level addressing scheme, in which the PLH resolves to place-local information, and then a particular split queue is selected.

A split queue object encapsulates all data of a worker and the respective methods, which has the advantage that split queue methods can directly access the local data through instance fields. Access to remote queues goes through variable `tp`, which is stored in the split queue object.

Chapel Variant. As Chapel does not support a PLH-like construct, we had to explicitly work with a distributed (or replicated) array. We use a one-level addressing scheme, i.e., we have one entry per worker (as opposed to per locale). The Chapel variant is illustrated in Fig. 1(b). Access to remote queues is enabled by declaring the distributed array at module scope.

Place-Local and Activity-Local Data. Comparing the two variants, the PLH concept is appealing since it eliminates the need for explicit indexing. When place-internal indexing is needed instead, the advantage is, however, lost. For our application, PLH-like support for activity-local data would have been most useful. It would have enabled a similar program structure as in Fig. 1(b), but with simpler addressing. While our use of activity-local data is to some degree application-specific, long-running activities reduce the overhead, e.g. for initializing data structures, and may therefore be valuable beyond our application. As an application may mix place-local and activity-local data, we suggest to support both in both languages.

3.3 References, Values, and Copying

In both languages, a tree node may be represented by either an object, or a record (struct). From a performance point of view, records may be cheaper as they do not incur the typical indirection and method resolution costs of objects. Moreover, memory allocation is cheaper if one large block is allocated for all

records in the task pool, as compared to allocating pieces of memory for each node object. At the backside, record assignment and parameter passing by value, e.g. in `push`, involve expensive copying and should therefore be avoided.

In addition to class-based versions, we implemented record-based versions that avoid copying by expanding children directly into their task pool entry. A problem arises during the generation of initial tasks, when a child is to be expanded into a remote queue. Chapel calls a C function to decode a node descriptor, which takes as arguments pointers to both the parent and future child descriptors, but it is not possible to pass a pointer to remote memory to this function. Therefore, the parent descriptor is first copied to a remote variable, and then the C function is called. This shows up limits of native code integration with Chapel.

In the X10 program, remote access to native code was easier. Since the reused code from [15] represents a node by a C++ object, that object is automatically copied to the remote place when its native code is required. The approach appears easier but less efficient, especially as X10 generates a deep copy and inclusion of fields can not be controlled at the C++ side.

3.4 Worker Management and Initialization

The base functionality for starting worker tasks is obvious, e.g.:

```
coforall loc in Locales do on loc
  coforall tid in 0..#numWorkersPerLoc do runWorker(tid);
```

The corresponding X10 code is slightly longer, as there is no equivalent of `coforall`, and `async` must be ended by `finish`. Activity-local data would help managing the task identifier `tid`.

To allow stealing, the above `coforall` loop may only be entered after the task pool has been initialized, especially references to the remote split queues must have been set. Task pool initialization is by itself distributed, but the pool must not be used before initialization has finished. This can be achieved, e.g., by a barrier. There are various opportunities to implement this barrier, e.g. in Chapel pairwise synchronization before a worker's first steal access to a victim locale can be implemented with synchronization variables.

3.5 Reduction

Reduction is a well-known pattern to combine multiple values. It can be efficiently parallelized, in our setting by first combining the values within each worker, then within each place, and finally within the overall program. UTS uses reduction to compute the result number of nodes.

Chapel and X10 provide language support for reduction only on the assumption that the values are stored in an array, which has several drawbacks:

- The values must be kept in an array even if it does not match the application's structure. In our Chapel variant, e.g., a local value would logically belong to

the **Worker** class, and thus the array needs to be defined and filled just for the purpose of reduction.

- In the array, false sharing between neighbored values is likely.

OpenMP defines reduction differently [7]: A user program declares a scalar variable with some keyword for reduction, and specifies the operator. The system transparently defines local copies, collects values locally, and synchronizes the update of the global result. At least in Chapel, a similar scheme should be possible and would be desirable from a user’s point of view.

3.6 Diverse Language Issues

Split Queue Synchronization. Synchronization is required for the public pool. The **steal** critical section, e.g., includes checking **npublic**, modifying **npublic/tail**, and copying the tasks out of the pool. It is kept short by first making a local copy of the tasks, and sending it to the remote place after the critical section. X10 critical sections are coded with **atomic**, whereas our Chapel programs use a synchronization variable that holds **npublic**.

Remote Access. A Chapel programmer may inadvertently access remote variables. Tool support might help and, unlike X10’s **at**, not impose any restrictions. When using a PLH, a similar problem occurs when the user forgets **at** as in **for** `<allPlaces> { tp().init(); }`, where **tp** is always evaluated at the origin. An X10 **at** only copies **val**’s. When they need to be computed before being sent, both a **var** and a **val** variable for the same purpose are needed, which blows up the code and requires copying.

Constants and Parameters. In Chapel, tree parameters are naturally stored in configurable variables that can be easily overwritten on the command line. Parameter passing in X10 is more complicated.

In Chapel, **val**-like variables may be declared with **single**, i.e., there is some redundancy between **const** and **single**. Possibly, **const** may be removed if the **config** label is extended to **single**, and **single** values are replicated across places.

Language vs. Library. By releasing central functionality to the library, chances for integration may be dismissed. In X10, for instance, a language construct such as **at(allPlaces)** would appear elegant.

4 Performance

We run experiments on a cluster of 8-core Intel Xeon E5-2670 processors, with 2 processors per node and Infiniband network. For X10, compiler option **-O** was used, and for Chapel **gasnet/ibv** for multi-node and **none** for single-node execution.

Table 1 shows running times for the T1L sample of UTS [14], which is a geometric tree with branching factor 4 and maximum depth 13. The results

Table 1. Running times of different program versions (averaged over 3 runs).

	Chapel class	Chapel struct	X10 class	X10 struct
1 Place 1 Thread	72.8	365.5	44.8	36.4
1 Place 4 Threads	26.6	138.8	16.9	12.6
1 Place 16 Threads	23.5	65.7	7.3	4.5
4 Places 4 Threads	143.0	1286.0	6.1	5.4

suggest a performance advantage of X10 over Chapel. In X10, the struct-based variant was slightly faster than the class-based one, while in Chapel the record-based variant were inferior.

The results can only be considered a snapshot, as the current versions of the languages do not yet exhaust their performance potential. For instance, the release notes state that Chapel 1.8.0 is not suitable for in-depth performance comparisons, and the X10 atomic sections induce unneeded serialization. Most of all, we did not tune the performance, and therefore there is likely much room for improvements in all versions.

5 Related Work

As mentioned in Sect. 2.4, UTS has already been implemented with Chapel and X10. The previous Chapel implementation [3] starts with one task queue. As soon as it has reached a certain size, it is split into two queues, and a new Chapel task is started to process the second queue. The program runs within a single place only. The previous X10 implementation [13] focuses on termination detection, and performance tuning includes low-level functionality such as `IndexedMemoryChunk`. This way, it achieves excellent and scalable performance. In contrast, we took the position of a high productivity programmer and did not tune the performance. The previous X10 implementation deploys only one activity per place and a cooperative work stealing algorithm that does not require synchronization. Unlike these implementations, we closely followed the traditional task pool pattern.

Beyond UTS, several experience reports on coding applications with Chapel and X10 have been published. Referring to older language versions, Shet et al. [12] discuss experiences with a quantum chemistry kernel. Their work includes a central task pool, which is simpler than ours. Weiland [16] presents a nice comparative survey of language features in earlier language versions. Khaldi et al. [4] compare six parallel languages and discuss aspects of expressiveness such as synchronization constructs with the Mandelbrot example. Several recent papers report on experiences in coding applications such as constraint-based local search and the fast multipole method in X10 [9].

While we have used task pools as a benchmark for language design, Chapel and X10 also deploy task pools in the runtime system, to map activities/tasks to threads, see e.g. [5]. Problems of combining object orientation and parallelism

have been discussed since a long time [1, 8]. This paper focused on encapsulation, and was specific to the PGAS setting.

6 Conclusions

This paper has evaluated Chapel and X10 from a user's perspective, working out both differences and common grounds such as difficulties in integrating object orientation and parallelism. We suggested several modifications to strengthen the languages such as support for place-local and activity-local data, scalar variable-based reduction, and the omission of `const`.

Our work was based on a single benchmark, with focus on task parallelism and object orientation. Before drawing conclusions on the usability of the languages in general, one needs to consider more benchmarks and put a stronger emphasis on performance.

References

1. Agha, G., Wegner, P., Yonezawa, A. (eds.): Research Directions in Concurrent Object-Oriented Programming. MIT Press, Cambridge (1993)
2. Chapel Language Specification, Version 0.94. <http://chapel.cray.com/papers.html> (2013)
3. Dinan, J., et al.: Unbalanced Tree Search (UTS) benchmark in Chapel. Program source <https://chapel.svn.sourceforge.net/svnroot/chapel/trunk/test/studies/uts/> (2007)
4. Khaldi, D., Jouvelot, P., Ancourt, C., Irigoin, F.: Task parallelism and data distribution: an overview of explicit parallel programming languages. In: Kasahara, H., Kimura, K. (eds.) LCPC 2012. LNCS, vol. 7760, pp. 174–189. Springer, Heidelberg (2013)
5. Kumar, V., et al.: Work-stealing by stealing states from live stack frames of a running application. In: Proceedings of the ACM SIGPLAN X10 Workshop (2011)
6. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.-W.: UTS: an unbalanced tree search benchmark. In: Almási, G., Caşcaval, C., Wu, P. (eds.) LCPC 2006. LNCS, vol. 4382, pp. 235–250. Springer, Heidelberg (2007)
7. OpenMP Application Program Interface, Version 3.1. <http://www.openmp.org> (2011)
8. Philippsen, M.: A survey on concurrent object-oriented languages. *Concurr. Pract. Exp.* **12**(10), 917–980 (2000)
9. Publications Using X10. <http://x10-lang.org> (2013)
10. Ravichandran, K., Lee, S., Pande, S.: Work stealing for multi-core HPC clusters. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 205–217. Springer, Heidelberg (2011)
11. Saraswat, V., et al.: Lifeline-based global load balancing. In: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, pp. 201–212 (2011)
12. Shet, A.G., et al.: Programmability of the HPCL languages: a case study with a quantum chemistry kernel. In: Proceedings of the International Parallel and Distributed Processing Symposium. IEEE (2007)

13. Tardieu, O., et al.: X10 for productivity and performance at scale: a submission to the 2012 HPC Class II challenge. In: Proceedings of the SC Conference on High Performance Computing, Networking, Storage and Analysis. <http://x10-lang.org> (2012)
14. UTS. <http://hpcrl.cse.ohio-state.edu/wiki/index.php/UTS>
15. X10 Code for UTS. <http://x10.svn.sourceforge.net/viewvc/x10/benchmarks/trunk/UTS/>
16. Weiland, M.: Chapel, Fortress and X10: novel languages for HPC. Technical report, HPCx Consortium (2007)
17. X10 Language Specification, Version 2.4. <http://x10-lang.org> (2013)