UNORDERED ASSOCIATIVE CONTAINERS IN STAPL

An Undergraduate Research Scholars Thesis

by

TYLER J. BIEHLE

Submitted to Honors and Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Research Advisor: Nancy Amato

May 2014

Major: Computer Engineering

TABLE OF CONTENTS

Page

ABSTRACT 1

ACKNOWLEDGMENTS 2

I INTRODUCTION 3

II RELATED WORK 6

III STAPL PCONTAINERS 8

STAPL Overview 8

pContainer Framework 9

IV UNORDERED ASSOCIATIVE PCONTAINERS 12

The Unordered pSet 12

Hash Directory 14

The Unordered Multi pContainers 14

The Multikey Class 15

The Unordered pMultiset 15

The Unordered pMultimap 16

V PERFORMANCE EVALUATION 17

Unordered Associative pContainer Method Evaluation 17

Unordered Associative pContaner Map-Reduce Evaluation 19

VI CONCLUSION 21

REFERENCES 22

ABSTRACT

Unordered Associative Containers in STAPL. (May 2014)

Tyler J. Biehle
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Nancy Amato
Department of Computer Science and Engineering

The Standard Template Adaptive Parallel Library (STAPL) is a parallel programming framework for C++ that provides parallel algorithms and containers similar to those found in the Standard Template Library (STL) [1]. Currently STAPL is lacking implementations for three unordered associative containers: unordered set, unordered multiset, and unordered multimap. These are commonly used containers in the field of computer science [2]; therefore, their implementations are a necessity for STAPL. The similarity of operations and structure between each container will allow a large portion of code to be reused. The goal of this work is to design and create a parallel implementation of these containers that provides the same user-level facilities as their STL equivalents and displays a high level of scalability when executed on a large number of processors.

ACKNOWLEDGMENTS

I would like to acknowledge Dr. Nancy Amato for her continued support and guidance. Thank you so much for allowing me to be a part of such an amazing research group. I would also like to thank Adam Fidel, Timmie Smith, and Nathan Thomas for their ideas and insight with this work.

CHAPTER I

INTRODUCTION

Parallel programming has become a necessity in application development due to the availability of multiprocessor and multicore architectures and the need to solve large, complex problems. The purpose of the Standard Template Adaptive Parallel Library (STAPL) [1] is to allow users to write parallel programs at a high level and avoid many low-level details specific to parallel programming. STAPL is a parallel C++ library with similar functionality as the C++ Standard Template Library (STL) [3]. The STL provides the user with common containers, iterators, and algorithms to use as the building blocks of sequential applications. Similarly, STAPL provides the user with various distributed containers (pContainers) and parallel algorithms (pAlgorithms) with which to write parallel applications [4]. The primary goal of STAPL is to provide a high productivity environment for application development on a variety of parallel architectures including both shared and distributed memory.

This work presents implementations for three unordered associative pContainers inside of STAPL that are parallel versions of their STL equivalents. Unordered associative containers provide methods to store and locate elements in amortized constant time. The term **associative** refers to the manner by which the containers reference their elements. An associative container uses a key to identify and locate each element rather than referencing the element by its absolute position in the container. The **unordered** property indicates that the containers do not enforce a global ordering on their elements; instead, the elements are grouped together through the use of a hash function. The hash function uses an element's key to compute a value that identifies which group the element belongs to.

There are four additional properties that define the structure and capabilities of the unordered associative containers: *simple*, *pair*, *unique*, and *multiple*. Simple elements require that an element's key be equivalent to its value; therefore, rather than store two separate

values, only the key is stored in the container. Pair elements do not enforce the aforementioned requirement and must store the standard (*key*, *value*) pair. Uniqueness requires that all of the keys stored in the container be unique while the multiple property allows different elements to have equivalent keys. Using the definitions of each of the four properties we can properly define each of the unordered associative containers: the unordered set is simple and unique, the unordered map is pair and unique, the unordered multiset is simple and multiple, and the unordered multimap is pair and multiple.

The primary advantage of unordered associative containers is that the user is able to quickly store and access any arbitrary element by computing the hash value of its key rather than searching through the entire container for it. This makes insert, find, and erase methods amortized constant-time operations, or asymptotically $O(1)$. The STAPL unordered associative containers are thread-safe, concurrent objects that provide interfaces to access and manipulate their elements concurrently. The methods of the STAPL unordered associaitve pContainers include parallel counterparts to the methods provided by the STL unordered associative containers: insert, erase, find, and equal_range; the equal_range method is not currently implemented.

We will present the design and implementation of three STAPL unordered associaitve pContainers: unordered set, unordered multiset, and unordered multimap. We will also discuss the STAPL pContainer framework (PCF), which provides the facilities necessary to construct thread-safe, distributed pContainers from a few basic building blocks, as well as a new structure created to optimize the performance of the STAPL unordered associative pContainers. By developing scalable, parallel implementations for the three unordered associative containers, we were able to extend the facilities of STAPL and show that the STAPL framework limits the overhead cost of performing operations in parallel. In summation, our work has made the following contributions:

- An extension to the STAPL pContainer facilities

- A hash-based directory to efficiently map elements to locations.

This paper will first discuss projects with goals similar to this work (Chapter 2). Afterwards we will discuss the STAPL pContainers and the library components used to implement our containers (Chapter 3) and the discuss in detail our implementations for the unordered set, unordered multiset, and unordered multimap (Chapter 4). We will conclude with an analysis of the results and discuss our conclusions and future work (Chapters 5 & 6).

CHAPTER II

RELATED WORK

There has been a large amount of research in the area of distributed and concurrent data structure development. Most of the work has focused on utilizing either different locking primitives or lock-free data structures to implement concurrent data structures [5]. Work has been done with concurrent unordered associative containers to develop efficient storage methods and various locking implementations and strategies for shared memory architectures. However STAPL unordered associative pContainers are designed for both shared and distributed memory.

The Intel Thread Building Blocks library (TBB) [6, 7] segments shared memory systems and provides parallel implementations for all four unordered associative containers. TBB provides the user with an interface that resembles the STL interface. This library achieves high levels of concurrency through fine-grained locking and lock-free techniques. In fine-grain locking, threads are able to lock a specific part of the container to allow multiple threads to operate on the same container at once. The drawback to this strategy is the high overhead cost. In order for the concurrent containers to outperform their sequential counter-parts, there must be a large amount of available parallelism. Also, the containers do not support a safe concurrent erasure operation. STAPL differs from TBB in that it functions on both shared and distributed memory hierarchies while TBB operates only on shared-memory.

The Parallel Patterns Library (PPL) [8] provides task parallelism, parallel algorithms, and parallel containers that follow similar conventions as the STL. Included in this library are concurrent implementations for the four unordered associative containers. As with the TBB library's implementations, the PPL does not provide a concurrent version of the erasure operation.

There are several parallel libraries and languages that share similar goals with STAPL [9, 10, 11, 12, 13, 14]. The Parallel Standard Template Library (PSTL) [15, 16] followed the same principle of STAPL by extending the STL for parallel programming. The PSTL provides concurrent containers with both global and local iterators through which elements may be accessed; the local iterators traverse a container's elements located on a single location while the global iterators traverse all of the container's elements across the entire machine. The library has implementations for several concurrent containers but it does not provide implementations for concurrent unordered associative containers and the project is no longer active.

There is also work [17] that shares many concepts with STAPL. The most notable similarity is the use of sub-containers, or base containers, to distribute the container's elements across the machine. The primary difference between the two is that STAPL can be ported to both shared memory and distributed systems rather than being limited to only one type of system. Also, the STAPL pContainers are extensible; this gives users the abililty to define and implement their either a new custom container from our pContainer framework [4] or as an extension of one of our containers.

In addition to the aforementioned libraries there are several languages [18] that aim to reduce the complexity of parallel programming; one such example is Chapel [19] by Cray. The language describes a formal approach for containers and data distributions. Chapel provides default data distributions and specifications for creating new ones. Currently Chapel does not support unordered associative containers. STAPL differs from languages such as Chapel in that it is a C++ library and thus can work directly with standard C++ compilers rather than requiring custom compilers to build applications.

CHAPTER III

STAPL PCONTAINERS

STAPL Overview

STAPL, whose components are shown in Figure III.1, consists of the following: pContainers, pAlgorithms, pViews, PARAGRAPHS, and a runtime system. A pContainer is the thread-safe, concurrent equivalent of the STL container; its methods are parallel, meaning they can be invoked concurrently. Every pContainer provides a parallel implementation for the methods found in the sequential interface of the STL; in addition, some pContainers also provide methods specifically to take advantage of available parallelism. The user is able to ignore the physical location of elements by interacting with a shared object view which provides unifom access through a distribution manager. STAPL provides mechanisms to ensure that all operations leave the pContainers in a consistent state after execution; this guarantees thread safety. STAPL also supports nested parallelism by allowing pContainers to be constructed from other pContainers.

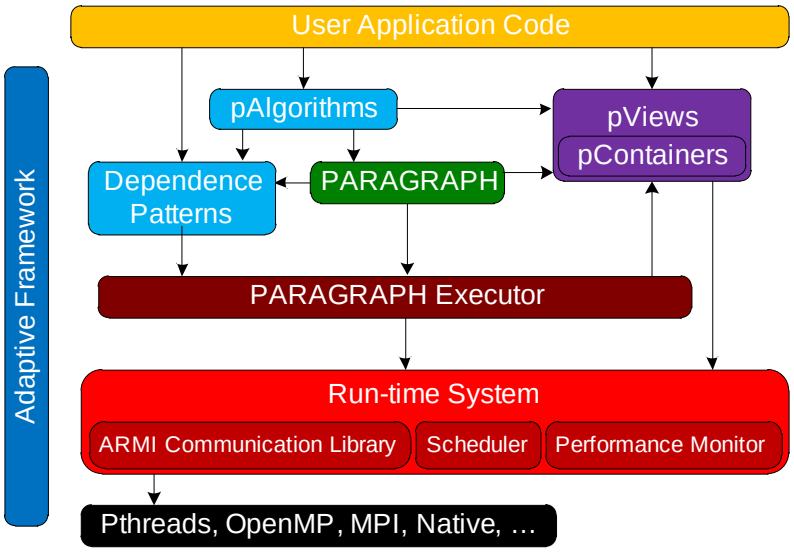

Fig. III.1. STAPL Components

The user is able to access data inside of a pContainer through a pView. In STAPL, a pView is logically equivalent to the iterators of the STL. The physical location of each element is abstracted away and instead the elements are viewed as a collective group. The user can directly access individual elements using iterators from within a pView. STAPL uses pViews to access data in generic parallel algorithms (pAlgorithms), which follows the STL convention of writing algorithm data access in terms of iterators instead of directly on the container. The PARAGRAPH is used to represent the task graph of an algorithm in parallel. Essentially each node in the graph is a task composed of a higher order function (denoted work function) and a view. The PARAGRAPH allows the user to specify and enforce data dependencies between individual tasks.

The runtime system (RTS) and its communication library Adaptive Remote Method Invocation (ARMI) [20, 21] provide the interface to the underlying operating system, naive communication library, and hardware architecture. In order to hide the low-level implementations of communication, ARMI uses the remote method invocation (RMI) communication abstraction. In STAPL, the remote invocation of a method can be either blocking or non-blocking. A blocking invocation will cause a location to block until the method finishes its execution on a remote location and returns the results. A non-blocking invocation will only cause the a location to initialize the specified method; no return type is specified. The RTS will handle the non-blocking invocation once it has completed its execution. ARMI also provides a mechanism to guarantee the completion of any previous RMI calls; this mechanism is refered to as a fence. In order to minimize remote communication, the RTS can aggregated asynchronous calls with an internal buffer.

pContainer Framework

The pContainer Framework (PCF)[4] was created to simplify the task of implementing a new pContainer. The PCF allows the programmer to derive specialized containers from existing

containers and classes and avoid low-level concurrency issues. The PCF is designed as a hierarchy of classes from which pContainers may derive.

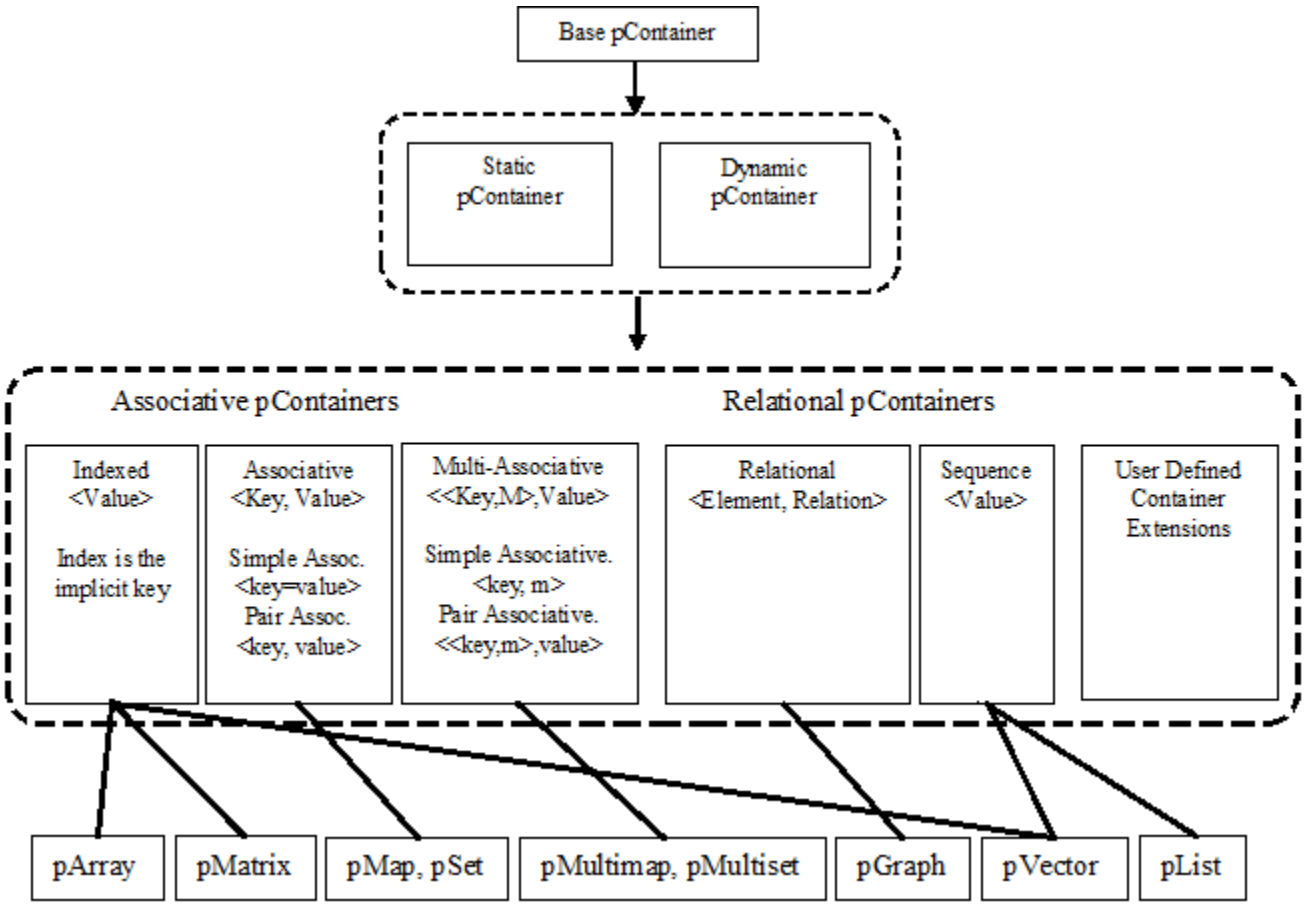

Fig. III.2. The PCF Hierarchy of Classes.

The first level is the base pContainer which stores the pContainer's elements and distribution strategy; all pContainers derive from this base. The next level contains two classes, static and dynamic, which dictate whether elements may be added and removed once the container is constructed. The third level discriminates between relational and associative containers, both of which have implicit and explicit variations. Relational pContainers store elements that hold some sort of relation to one another. An implicit relation would be found in a list or vector where a sequential relation is enforced on the elements. An explicit

relation would be found in a graph or tree where the relations can be created or removed and assigned values. Associative relations store elements and associate each one with a key. An implicit association would be found in an array where elements are associated with the index of their position in the container. An explicit association would be found in a type of map where each element must be assigned a key value. Once the programmer has selected one class from each level, they may customize the pContainer's interface to match their needs.

Previously there was no class from which multi-associative pContainers could derive. The first stage of this work involved implementing a multi-associative base class. The new multi-associative class was then used to implement the unordered multiset and unordered multimap pContainers.

The PCF also supports the shared object view of a pContainer. In order to determine where an element is or should be, the PCF provides an address translation mechanism. The address translation is comprised of three different components: a domain, a partition directory, and a partition mapper. The domain is the set of each element's unique global identifier (GID). The domain is split into non-intersecting sub-domains. The partition directory then maps each GID to its sub-domain and the parititon mapper maps each sub-domain to a location. Access is a specific element is achieved by determining its GID, finding the sub-domain the GID belongs to, and determining the location which houses the specified sub-domain.

CHAPTER IV

UNORDERED ASSOCIATIVE PCONTAINERS

In this chapter we will discuss the Unordered Associative pContainers and explain their implementations within STAPL. First we will give a detailed explanation of the implementation for the Unordered pSet which shares many similarities with the Unordered pMap. We will conclude this chapter by explaining how the Unordered pSet was extended to implement the Unordered pMultiset which again shares many similarities with the Unordered pMultimap.

The Unordered pSet

The Unordered pSet was designed to emulate the STL's interface as closely as possible by providing standard functions such as insert, erase, find, and functions which return an iterator to an element in the pContainer. The primary difference between the two is the returned values for the functions in the interface. The functions in the STL implementation return either an iterator or a reference to elements which have been inserted or that follow erased elements. In STAPL, function calls can be invoked remotely so returning a value would require a synchronization which would negatively impact performance; therefore, our functions do not return a value unless required by their definition, such as find. This allows us to invoke methods such as insert and erase asynchronously.

The STAPL pContainer Framework provides base classes to assist in implementing pContainers that are both extendable and composable. The components derived from the PCF are the *globally unique identifier*, the *domain*, the *distribution*, and the *base container*.

STAPL requires that elements within a container be given a globally unique identifier called the GID. Requiring every element to be uniquely identifiable prevents ambiguity when accessing elements across the global space. Similarly, the unordered set requires that each element be paired with a unique key. The commonality between STAPL's uniqueness re-

quirement and the definition of the unordered set allows an element's GID and key to be equivalent. This allows the user to access any arbitrary element just by knowing its key which aligns perfectly with the STL interface.

The domain represents all possible GIDs for a container and stores the GIDs of elements currently in the pContainer. For example, if our Unordered pSet contained the values 2, 3, and 5, the domain would represent all integer values and the current instance of the domain would be {2, 3, 5}. This is implemented by the *iterator domain* from the PCF. The iterator domain extracts GID information from values stored in the container using the container's iterators to the first and last value as the domain bounds. Global iteration uses a GID-based iterator class.

The distribution determines the location of the pContainer's elements on the machine and breaks the domain down into sub-domains. This task is broken into two subtasks; the distribution first determines which sub-domain a particulair GID belongs to and then determines which location a sub-domain belongs to. This two-tiered approach allows work to distribute evenly across the available locations. This is implemented in the Unordered pSet through the use of a hash function. The user may define their own customized hash function but the default function is defined as a modulus operation of an element's key over the number of processors available. The result of the hash function determines the element's location and then control is passed over to that location.

The final structure is the base container. The base containers actually store and maintain the elements of the pContainer. For the Unordered pSet, the base containers are sequential unordered sets. By definition, the unordered set maintains several buckets in which it stores elements. This definition aligns quite nicely with the concept of base containers in the sense that our buckets are actually sequential unordered sets. Each location is given a single base container; therefore, when the distribution maps an element to a location, it is

actually mapping it to a specific, sequential unordered set. This allows us to distibute the elements evenly and access them by performing two hash operations; the first to determine the correct base container and the second to determine the correct bucket within the base container. This allows us perform operations on elements in constant time.

Hash Directory

Prior to this work, STAPL did not provide a mechanism within the distribution to determine where elements in a dynamic container should be stored through a hash function. The alternative was to explicitly store a pair of values (*key*, *location*) for each element where *key* represents an arbitrary element and *location* represents where *key* is stored. The drawback of this procedure was the high cost of memory. We decided to design the missing mechanism in order to address this issue of high memory consumption.

The hash directory allows for efficient look-up of elements based on their key through the use of the same hash function used by the pContainer. The specific difference between the two hash functions is the pContainer uses it to map elements to base containers while the directory uses it to map elements to locations; however, since each location is given only one base container in the general case, the two hash functions are identical and their performance correlates directly. Also, because we can easily compute where an element is located, we do not need to explicitly store the location of each element which significantly reduces our memory consumption.

The Unordered Multi pContainers

Theoretically, the Unordered pMultiset is identical to the Unordered pSet except that it allows for different elements to have equivalent keys. The same can be said regarding the Unordered pMultimap and Unordered pMap. This small difference presented a large problem within the STAPL framework. Both the Unordered pSet and Unordered pMap had used their elements' keys as the GIDs because the keys were required to be unique. Since the Unordered

pMultiset and Unordered pMultimap do not require unique keys, there was no longer a way to uniquely identify elements inside the pContainers. In order to correct this issue we had to implement a class, called *multikey*, to act as a wrapper around the user-defined key.

The Multikey Class

The *multikey* class takes an element's key and pairs it with a nonnegative integer; this nonnegative integer represents the multiplicity of a given key. Assigning the multiplicity value to each key allows the pContainer to store duplicate keys while simultaneously adhering to the unique GID requirement enforced by the underlying classes.

Changing the structure of the elements changes the user interface of the pContainer. To correct this issue the *multikey* class is implemented to receive the lone *key* value from the user to construct or access an element, as with insert and find, and strip away the *multiplicity* when returning an element to the user. The erase function erases all elements with a specific key so the *multiplicity* does not impact the function's implementation. By implementing the *multikey* class to modify what the user sees when interacting with elements we are able to provide the expected interface and hide our pContainer's implementation details from the user.

The Unordered pMultiset

The usage of the *multikey* class as the key of Unordered pMultiset's element changes their structure from (*key*) to (*key*, *multiplicity*) which allows the pContainer to satisfy the requirement of unique GIDs. Similar to the Unordered pSet, the Unordered pMultiset uses the *iterator domain* to initialize its domain; however, the iterators handle elements with *multikey* keys. The underlying base container of the pContainer is a sequential unordered multiset. The distribution follows the same strategy as the Unordered pSet.

The Unordered pMultimap

The usage of the *multikey* class as the key of Unordered pMultimap's element changes their structure from (*key*, *value*) to ((*key*, *multiplicity*), *value*) which allows the pContainer to satisfy the requirement of unique GIDs. Similar to the previous pContainers, the Unordered pMultimap uses the *iterator domain* to initialize its domain; however, the iterators handle elements with *multikey* keys. The underlying base container of the pContainer is a sequential unordered multimap. The distribution follows the same strategy as the previous two pContainers.

CHAPTER V

PERFORMANCE EVALUATION

In this chapter, we evaluate the scalability of the parallel methods described in the previous chapter. We will also show the performance of the container with the **map_reduce** algorithm; the map operation is an identity function and the reduction is a sum function. The experiments were conducted at Texas A&M University on a Cray XE6m-200 with 24 compute nodes; 12 nodes have a single processor and an NVIDIA Kepler GPU and the other 12 nodes have two processors. All processors are AMD Opteron 6272 'Interlagos' 16-core processors running at 2.1 GHz. Each node has RAM that amounts to 2GB per core. Interconnect between nodes is a Cray Gemini interconnect with nodes arranged in a 2D torus. In all experiments a location contains a single processor and the terms are equivalent.

Unordered Associative pContainer Method Evaluation

In this section we discuss the scalability performance of the interface methods for the Unordered pSet, the Unordered pMultiset, and the Unordered pMultimap. Scalability, S, between two processor counts is defined as the ratio between execution time on the base processor count, T_b, and execution time on a higher processor count, T_p.

$$S = \frac{T_b}{T_p}$$

To evaluate each method's scalability we invoked each method on $\frac{N}{P}$ local elements, where N is the total number of elements in the pContainer and P is the number of processors the test ran on; we began our timer immediately before the first method invocation and stopped the timer once all of the method invocations finished. The time taken to invoke a given method globally N times was then used to compute the scalability values.

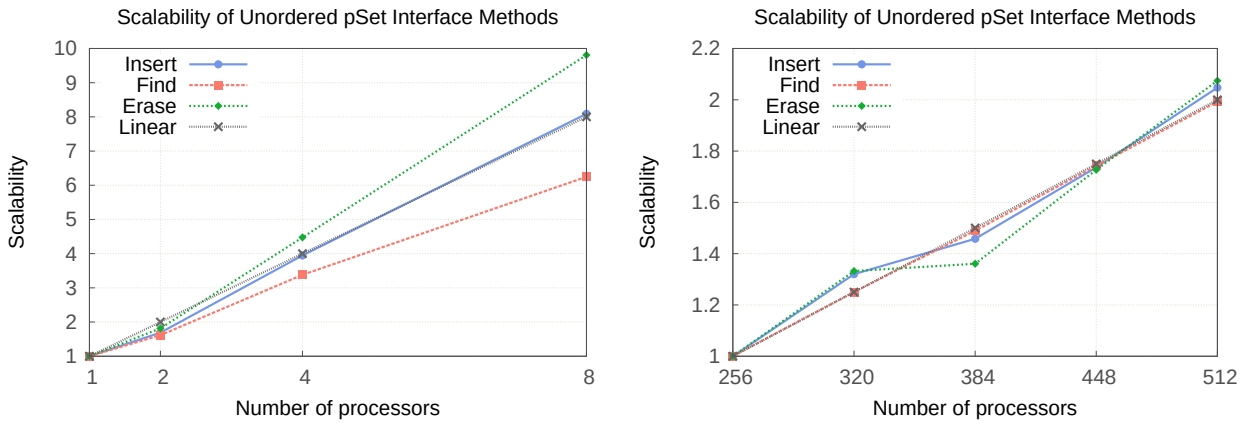

Fig. V.1. Method comparison for Unordered pSet. All operations are executed locally. For 1 to 8 processors N=100,000 and for 256 to 512 processors the N=180,000,000.

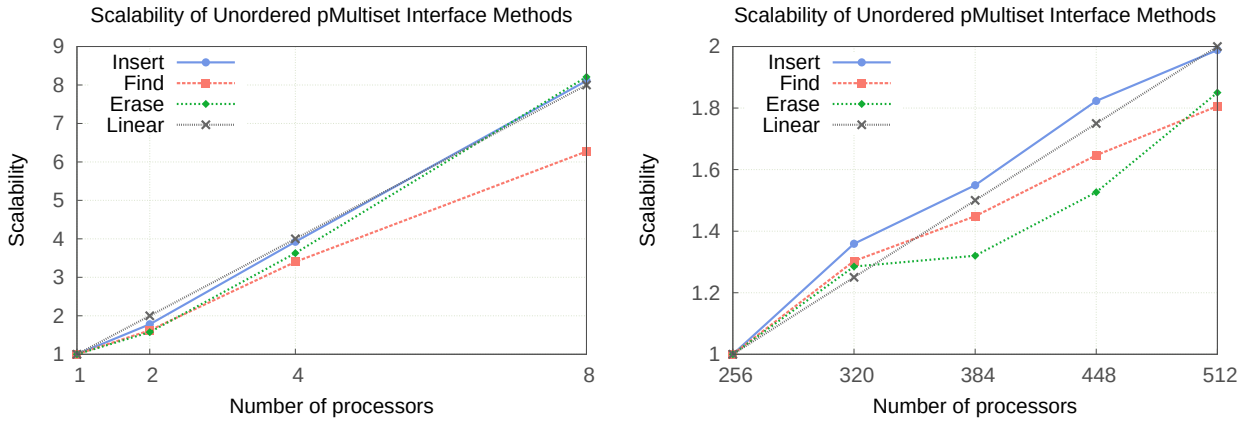

Fig. V.2. Method comparison for Unordered pMultiset. All operations are executed locally. For 1 to 8 processors N=100,000 and for 256 to 512 processors N=180,000,000.

Figure V.1 shows the scalability results for the Unordered pSet. For the Unordered pSet we see that all three methods scale well on the higher core counts. On the lower core counts insert and erase scale well but find does not; this is because find must return a value which makes it a synchronous operation. The find method did scale well on the higher core counts;

this is likely because the work was distributed enough to negate the cost of the synchronous behaviour.

Figure V.2 shows the scalability results for the Unordered pMultiset, whose methods scale as expected on the low core counts. On the high core counts we lose some scalability in the erase function; however, the lost scalability is less than 10%.

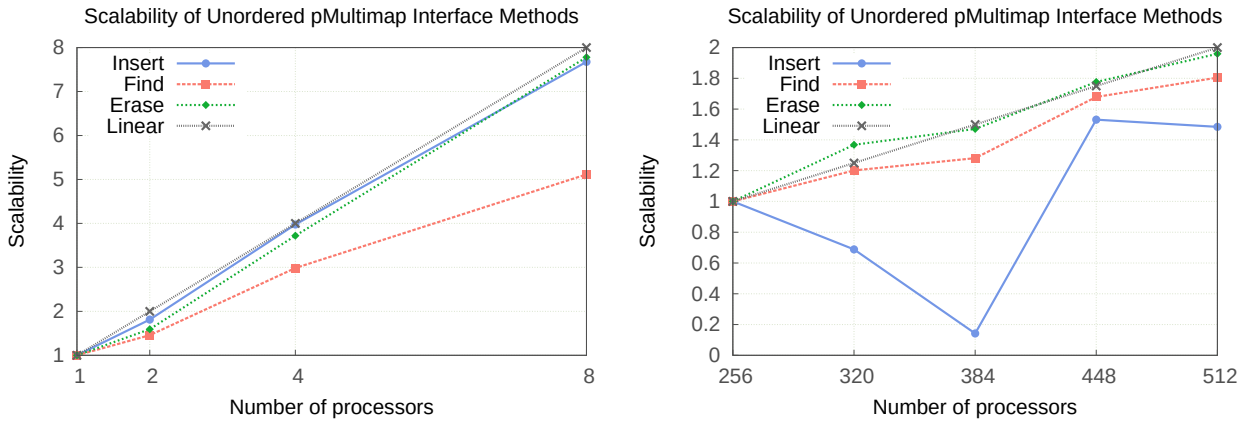

Fig. V.3. Method comparison for Unordered pMultimap. All operations are executed locally. For 1 to 8 processors N =75,000 and for 256 to 512 processors N=90,000,000.

Figure V.3 shows the scalability results for the Unordered pMultimap whose methods on the lower core counts again scale well. The only issue on the higher core counts was the erratic behavior of the insert function. One potential explanation for this erratic behaviour is poor hashing performance in the base container. This issue will be investigated further.

Unordered Associative pContaner Map-Reduce Evaluation

In this section we discuss the scalability performance of the **map_reduce** algorithm for all three pContainers. The scalability performance was measured in the same manner as in the previous section. The map operation of the algorithm was a simple identity operation and

the reduction segment was a summation of the element keys. In Figure V.4 we see sub-linear scalability on the lower core counts; this is due to the fact that the smaller division of work was not enough to overcome increase in communication introduced by adding additional processors. To resolve this issue we will need to reduce the communication costs. On the higher core counts the algorithm scaled well with the Unordered Associative pContainers.

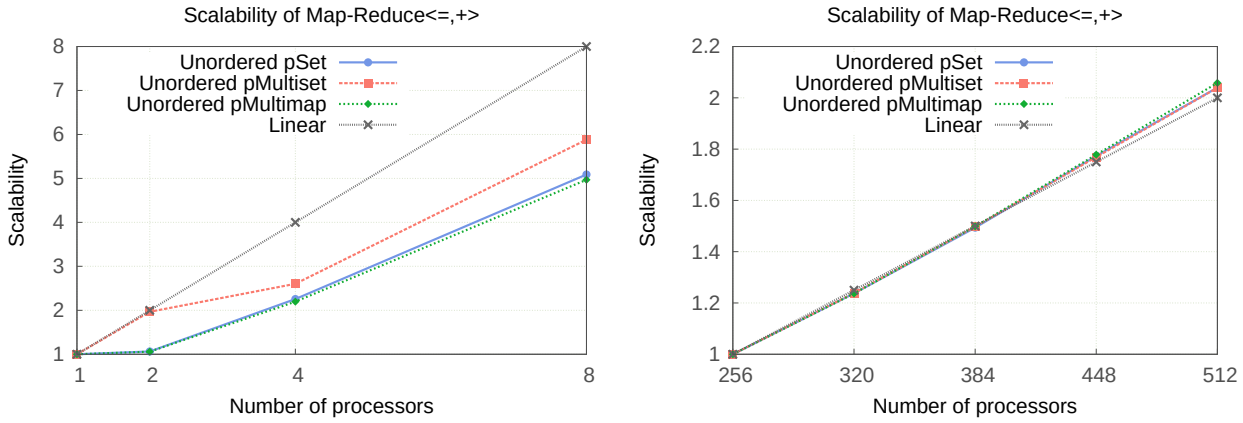

Fig. V.4. Map-Reduce comparison for Unordered Associative pContainers. For 1 to 8 processors, the Unordered pSet and Unordered pMultiset have N=100,000 and the Unordered pMultimap has N=75,000. For 256 to 512 processors, the Unordered pSet and Unordered pMultiset have N=180,000,000 and the Unordered pMultimap has N=90,000,000.

CHAPTER VI

CONCLUSION

In this paper we presented the STAPL Unordered pSet, Unordered pMultiset, and Unordered pMultimap; three distributed data structures optimized for fast, dynamic operations such as insert, find, and erase. We described the design and implementation of the three containers whose interfaces include counterparts of the STL unordered associative container methods. For the most part, the experimental results show that the containers provide good scalability; however, there are some performance issues that must be addressed.

There are several issues that will be addressed in future work. We will need to implement the equal_range method for the two multi containers. This method is included in the STL interface and our implementation is incomplete without it. In terms of performance, we will need to resolve the scalability issue for the Unordered pMultimap's insert method and reduce the communication overhead in the PARAGRAPH.

In summary, we have designed and implemented three parallel containers that each have properties identical to their sequential counterparts but allow for scalable concurrent access when used in a parallel program.

REFERENCES

[1] A. Buss et. al. STAPL: Standard Template Adaptive Parallel Library. In *Haifa Experimental Systems Conference*, Haifa, Israel, 2010.

[2] D. Mount M. Goodrich and R. Tamassia. *Data Structures ans Algorithms in C++*. Wiley, second edition, 2011.

[3] D. Musser P. Plauger, M. Lee and A. Stepanov. *C++ Standard Template Library*. Prentice-Hall, 1st edition, 2000.

[4] G. Tanase et. al. STAPL: The STAPL parallel container framework. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, 2011.

[5] M. Herlihy. A methodology for implementing highly concurrent data objects. In *ACM Trans. Prog. Lang. Sys.*, volume 15(5), pages 745–770.

[6] J. Reinders. *Intel Thread Building Blocks: Outfitting C++ for Multicore Processor Parallelism*. O'Reilly, first edition, 2011.

[7] Intel. *Reference for Intel Thread Building Blocks, version 1.0*. 2006.

[8] K. Kerr. *Visual C++ 2010 and the Parallel Patterns Library*. 2009.

[9] G. Blelloch. Vector models for data-parallel computing. In *MIT Press*.

[10] G. Blelloch. Nesl: A nested data-parallel language. In *Technical Report CMU-CS-93-129, Carnegie Mellon University*.

[11] A. Chan and F. K. H. A. Dehne. Cgmgraph/cgmlib: Implementing and testing cgm graph algorithms on pc clusters. In *In Euro PVM/MPI*, pages 117–125.

[12] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *SIGPLAN*, volume 40(10), pages 423–437.

[13] L. V. Kale and S. Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *SIGPLAN*, volume 28(10), pages 91–108.

[14] J. C. Cummings S. R. Atlas S. Banerjee W. F. Humphrey S. R. Karmesin K. Keahey M. Srikant J. V. W. Reynders, P. J. Hinker and M. D. Tholburn. Pooma: A framework for scientific simulations for parallel architectures. In *In G. V. Wilson and P. Lu, editors, Parallel Programming in C++*, volume Chapter 14, pages 547–588, MIT Press.

[15] D. Gannon E. Johnson and P. Beckman. Hpc++: Experiments with the parallel standard template library. In *Proc. of the 11th Int. Conference on Supercomputing (ICS)*, pages 124–131, Vienna, Austria.

[16] E. Johnson and D. Gannon. Programming with the HPC++ parallel standard template library. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[17] D. Andrade A. De Vega and B. Fraguela. An efficient parallel set container for multicore architectures. In *Advances in Parallel Computing*, 2011.

[18] V. Saraswat C. Donawa A. Kielstra K. Ebcioglu C. von Praun P. Charles, C. Grothoff and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In

OOPSLA '05: Proc. of the 20h ACM SIGPLAN conf. on Object oriented programming, systems, languages, and applications, pages 519–538, New York, NY, USA.

[19] B.L. Chamberlain D. Callahan and H. Zima. The cascade high productivity language. In *The 9th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments*, volume 26, pages 52–60, Los Alamitos, CA, USA.

[20] S. Saunders and L. Rauchwerger. ARMI: an adaptive, platform independent communication library. In *Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 230–241, San Diego, CA, USA, 2003.

[21] T. Smith G. Tanase N. Thomas, S. Saunders and L. Rauchwerger. ARMI: a high level communication library for stapl. In *Parallel Processing Letters*, volume 16(2), pages 261–280, 2006.