MULTIMAP AND MULTISET DATA STRUCTURES IN STAPL

An Undergraduate Research Scholars Thesis

by

JUNJIE SHEN

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 M. Amato

December 2014

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

MULTIMAP AND MULTISET DATA STRUCTURES IN STAPL. (December 2014)

JUNJIE SHEN
Department of Computer Science and Engineering
Texas A&M University

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

The Standard Template Adaptive Parallel Library (STAPL) is an efficient programming framework whose components make it easier to implement parallel applications that can utilize multiple processors to solve large problems concurrently [1]. STAPL is developed using the C++ programming language and provides parallel equivalents of many algorithms and data structures (containers) found in its Standard Library (STL). Although STAPL contains a large collection of parallel data structures and algorithms, there are still many algorithms and containers that are not yet implemented in STAPL. Multimap and multiset are two associative containers that are included in STL but not yet implemented in STAPL. The goal of this work is to design and implement the parallel multimap and parallel multiset containers that provide the same functionality as their STL counterparts while enabling parallel computation on large scale data.

ACKNOWLEDGMENTS

I would like to acknowledge Dr. Nancy Amato for her tremendous support and guidance on my udergraduate research. I am grateful to her for giving me the opportunity to participate in this fantastic research project.

I would also like to thank Dr. Tmmie Smith, Dielli Hoxha, Mani Zandifar, Adam Fidel, Harshvardhan and Tyler Biehle for their help and suggestions.

CHAPTER I

INTRODUCTION

In recent years, the rapid development of networking technology and advancing work in application domains where large data sets are the norm has resulted in the scale and availability of data growing rapidly. Increasing the performance of the processor is not efficient enough to solve large problems, but the low cost and high availability of the processors make parallel programming a feasible and effective way to shorten the running time of large problems. The time needed to solve large programs in parallel can be reduced by increasing the number of processors if the application is written using efficient parallel algorithms and data structures.

Standard Template Adaptive Parallel Library(STAPL)

Standard Template Adaptive Parallel Libray (STAPL) [1] is such a high-productivity parallel programming framework that encapsulates the details of how the data is distributed across and accessed by different processors and provides user-friendly interface similar to the C++ Standard Library (STL) [2]. The main components of STAPL are pContainers and pAlgorithms, which are convenient and practical building blocks that can be used to compose parallel applications efficiently. STAPL pContainers are concurrent objects that apply parallel methods on the data they store [3]. pAlgorithms are the parallel counterparts of STL algorithms. They are expressed using dependence patterns, and are provided access to the data they process by pViews [4]. pViews are abstract data types (ADTs) that abstract the details of data stored by a pContainer, providing only data access operations needed by a pAlgorithm. The automatic distribution of data in the pContainers along with the abstraction of inter-processor communication provided by the STAPL runtime system [5] allows the applications to run in parallel without the programmer explicitly specifying the details of the parallelization. The main objective of developing STAPL is to provide portable parallel performance for applications developed using it. Currently, STAPL includes a large

collection of parallel data structures (e.g., pArray[6], pVector, pList[7], pMap[8], pGraph[9], etc.) and parallel algorithms (e.g., prefix sums, the Euler tour technique), but the multimap and multiset containers do not have their parallel equivalents in STAPL.

Multimap and Multiset

Multimap and multiset are two important data structures in the C++ Standard Library. They are associative containers that keep track of the elements by their key values, not by the sequence of insertion or the relative position in a container [10]. The element value in multiset is its key value, and an element in multimap consists of a pair of one key value and one mapped value. The multiset is good at storing numeric data. A simple example is using the multiset to arrange prime factors of a number N from smallest to largest. For instance, the prime factorization $60 = 2 \times 2 \times 3 \times 5$ gives the multiset $\{2, 2, 3, 5\}$. The multimap is frequently used on indexed data such as inventory information or ranked webpages. In a multimap, the mapped value may have different data type from that of the key value. For example, the multimap can be used in a hotel inquiry system to store the name of each guest in a mapped value and use his or her room number as the key value. Identical room numbers are allowed if two guests share a room.

Several associative pContainers have already been completed or are being developed in STAPL, such as the map [8], set [8], unordered map, unordered set [11], unordered multimap [11] and unordered multiset containers [11]. The multimap and multiset differ from the map and set containers in that they may have some elements with the same keys while the keys in the map and set containers must be unique; but they are all ordered containers that keep their elements sorted by the key values. The unordered map, unordered multimap, unordered set and unordered multiset are similar to their ordered counterparts in the elements they store. However, the elements in unordered containers have no explicit order. Although STAPL has already implemented map and set containers to stored ordered elements, the key values that are used to sort elements in the containers must be distinct. Therefore, it is

necessary to implement multimap and multiset in STAPL to allow users to handle elements with equivalent key values.

The parallel multimap and multiset will be implemented using many facilities that are already available in STAPL. For example, my work will use the partition and mapping mechanisms provided by STAPL to organize the data between different processors. The STAPL pContainer Framework (PCF) [3] provides base classes that contain the essential functionality of pContainers to allow a developer conveniently assemble the framework of a new container. My work will derive from the associative pContainer base classes, and then add the functionality that is unique to multimap and multiset.

The implementation of parallel multimap and multiset containers in STAPL will offer the same functionality as the STL equivalent containers. The implementation of the methods will scale as the number of processing elements increase. The completion of this work will fill the gap of multiple-key associative containers in STAPL. By expanding its functionality, STAPL can have a widespread use in various fields, and shorten the time needed to solve large problems.

In summation, this work will make the following contributions:

- High-performance parallel implementation of multiple associative containers that provide alternatives for STL multimap and multiset
- An extension of STAPL features and functionality

This thesis will first discuss the previous work of parallel associative containers (Chapter 2) as well as the structures and some components of STAPL related with the multimap and multiset containers (Chapter 3). Afterwards, I will describe the implementation (Chapter 4) and present the results of tests with a large constant input and varying number of processors (Chapter 5). Finally, I will discuss the conclusions based on the analysis of the results and outline the future work on the topic (Chapters 6).

CHAPTER II

RELATED WORK

Due to the demand for high-performance parallel frameworks, a considerable amount of research has been done to develop distributed data structures and other concurrent building blocks for large-scale computing on clusters. As an important set of data structures in STL, multiple associative containers have been implemented in some parallel libraries. In STAPL, unordered multimap [11] and unordered multiset [11] are two existing containers whose basic functionality is close to that of the multimap and multiset, processing elements based on their key value, but the position of an element in unordered multimap and unordered multiset is determined by a hash function instead of a comparator used by multimap and multiset. All four of these containers are built using the STAPL pContainer Framework [3], which will be described in the next chapter. Other parallel libraries, such as PSTL [12, 13], TBB [14] and PPL [15], which pursue similar objectives to STAPL, also have distributed multiple associative containers or other equivalents, but not all of these libraries have implemented multimap and multiset containers. Some libraries only implemented unordered multimap and unordered multiset. Some libraries limit their implementations to shared memory systems.

The HC++ Parallel Standard Template Library (PSTL) [12, 13] is an extension of C++ STL that provides distributed containers, parallel iterators and parallel algorithms. Four distributed associative containers have been implemented in PSTL; they are distributed set, distributed map, distributed multiset and distributed multimap. The PSTL is focused on its portability and only supports Single Program Multiple Data (SPMD) model, while STAPL supports both SPMD and MPMD (Multiple Program Multiple Data) model and provide distributed containers, parallel iterators and parallel algorithms as well. Moreover, STAPL supports nested parallelism that is not provided in PSTL. Although PSTL includes the distributed version of both multimap and multiset, this project is no longer active.

The Intel Thread Building Blocks library (TBB) [14] is an efficient, parallel library that maximizes the concurrency and core utilization by using task stealing. Task stealing is a scheduling model that balances the workload between parallel processing units. If one core completes its work, TBB can reassign some work from other busy cores to it. STAPL also has a scheduler in its run-time system (RTS) [5] to determine which task to execute; and the scheduler also can be customized by the users to use work stealing if desired. TBB includes the concurrent equivalents for all four unordered associative containers, but it does not provide the implementations for the map, set, multimap and multiset containers. Another difference is that TBB can only run its application on a shared memory system, while STAPL supports both shared memory system and distributed memory system.

The Parallel Patterns Library (PPL) [15, 16] provides a programming model that resembles the C++ Standard Library (STL) [2] and the Intel Thread Building Blocks library (TBB) [14]. The main features of PPL are Task Parallelism, Parallel algorithms and Parallel containers that provide concurrent building blocks for fine-grained parallelism. PPL has a cooperative task scheduler that is similar as the task-stealing of TBB to distribute work between processing units. Although PPL provides concurrent unordered associative containers, the concurrent multimap and multiset have not been implemented in it.

Besides these parallel libraries, several languages have the same goals as STAPL to support parallel programming. Unfortunately most of them lack the implementation of ordered multiple associative containers. STAPL multimap and multiset can provide users more convenience and useful features to deal with indexed data.

CHAPTER III

STAPL OVERVIEW

Superficially, STAPL is represented by its core library, which includes the pContainers and pAlgorithms that can be directly used to compose parallel applications. Indeed, other components that contain the mechanisms and strategies for data distribution and task scheduling are kept away from the users. Our work is built on this infrastructure.

The Parallel Container Framework

To simplify the process of developing concurrent data structures, the Parallel Container Framework provides a set of predefined concepts and a common methodology that can be used to construct a new parallel container through inheriting features from the appropriate concepts [3]. Moreover, the PCF allows programmers to easily specialize the features according to their unique needs. The main concepts defined in the Parallel Container Framework are the Global Identifier, Domain, Distribution, Partition, Partition Mapper and PFC Base Classes [1]. The basic structure of the PCF and the interaction between these modules are shown in Fig. III.1.

Global Identifier (GID): Each element stored in a STAPL parallel container (pContainer) has a Global Identifier that distinguishes it from other elements in the same container. The GID supports data distribution and is a significant factor that allows us to provide a shared object view of the data space [1]. Some pContainers (e.g., map, set, unordered map) can use their unique key value to be the GID, but other multiple associative pContainers (e.g., multimap, multiset) have to use a (key, m) pair as the GID, where m is an integer used to manage the multiplicity of duplicated key values [8].

Domain: Each STAPL container has a domain that is a union of GIDs; associative containers in STAPL have a domain that is the entire set of possible GIDs. The first GID in the domain

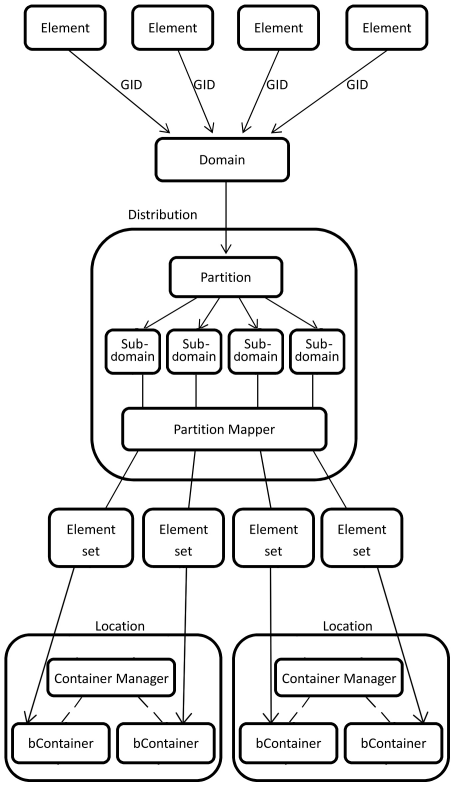

Fig. III.1. PCF structure

is the domain's lower bound, while the last GID is the upper bound. The gap between the lower bound and the upper bound represents the range of the domain.

Distribution: Data Distribution is the component that takes charge of the allocation of each element. An element will be sent to different location according to its GID. A location is a parallel execution unit that has a contiguous memory address space and has associated execution capabilities (e.g., threads) [1]. The Data Distribution uses a Partition to separate the domain into disjoint subdomains, and then employs a partition mapper to determine the location on which a subdomain should be stored (Fig. III.1).

Partition: The partition is a strategy that specifies how a domain is split into subdomains. It provides a surjective mapping function from GID to subdomains. Usually the partition

equally distributes the GID into each subdomain to keep the balance of workload for each processor.

Partition Mapper: the partition mapper is responsible for mapping the partitioned subdomains to locations. This is not a one to one mapping function; several subdomains may be allocated in one location depending on the number of subdomains and locations. There are two partition mappers currently available in STAPL: cyclic mapper, where sub-domains are distributed cyclically among locations, and blocked mapper, where m/L consecutive subdomains are mapped to a single location [8].

PCF Base Classes: The PCF Base Classes are implemented as the cornerstones for programmers to waive the tedious and repetitive work by building the PCF components deriving from these base classes. They are generic classes that use template parameters to meet users needs and provide the basic functionality of data distribution and container management such as associative distribution base class and container manager base class, which will be described later. Each of the components mentioned above can be customized by a developer implementing a new container.

Run-time System

The STAPL Runtime System contains ARMI (Adaptive Remote Method Invocation (RMI)) [5], an executor, schedulers and performance monitors. ARMI is a communication library that provides an abstraction of the inter-location communication for the higher level STAPL components. It is a platform dependent component in that its running characteristics are relevant to the operating system and the computer architecture, but it has the ability to adjust its features automatically according to different platforms or conditions to improve the performance and resource usage. ARMI provides primitives for communication that are Remote Method Invocations (RMI). RMIs have the flexibility of passing data or calling methods between processors, and thus can be more easily adapted to the needs of the ap-

plication. The executor is responsible for processing the tasks that are ready for execution. The scheduler arranges the set of tasks to determine which tasks are to be executed first [1].

pView

Another STAPL concept that provides a generic data access to the parallel algorithms is pView [4]. It works like the iterator in the C++ Standard Library, but a pView represents a set of elements while an iterator corresponds to a single element. A pView defines an abstract data type (ADT) that abstracts the details of the underlying pContainer and provides a uniform collection of data access operations for the elements it represents. It also has reference semantics, which means that a pView does not own the actual elements of the collection but simply references them [4].

CHAPTER IV

METHOD

STAPL multimap and multiset are designed to be the parallel version of STL multimap and multiset, therefore, they will provide the same user-level facilities as their STL equivalents, but with higher level of performance and scalability. Superficially, STAPL multimap and multiset are similar to their STL equivalents, but hide the detail of the concurrency and distribution management in the implementation of the STAPL containers. The components of the STAPL pContainer Framework that are specialized for multimap and multiset are base container, base container traits, distribution, container manager, the container class and container traits. They were implemented using the STAPL map and STAPL set as references, as they are already implemented in STAPL and are structurally similar.

The Traits are the template classes that can be passed to their corresponding components to specialize the properties according to users needs. It abstracts the types of primary components for the user to access them rapidly and conveniently, while hiding the other private or subordinate component types away from the user.

The container classes of STAPL multimap and multiset have the interfaces that are similar to their STL counterparts; most member functions in STL multimap and multiset are provided in STAPL equivalents with the same function names, return types and parameters. Furthermore, the STAPL containers also provide extra functions for users to manipulate the data, such as applying asynchronous methods to a given element. In the definition of these member functions, a key will be transformed to a unique GID to support the data distribution, but the key of the element in a multimap or a multiset is not necessarily exclusive. Thus, when an element with a key is passed into a function, the multiplicity of the key in this container will be calculated first, and then the multiplicity will be paired with the key to compose an instance of the multiKey structure (Fig.IV.1a) as the GID of this element. The multiKey contains the type cast operator to allow implicit conversion from multiKey

```
template<typename Key>
class multiKey : public std::pair<Key, size_t> {
  typedef Key key_type;

public:
   multi_key() : std::pair<key_type, size_t>() {}
   multi_key(key_type const& k, size_t const& m)
      : std::pair<key_type, size_t>(k, m) {}
   operator key_type() const {
      return this->first;
   }
   void define_type(stapl::typer& t) const {
      t.member(this->first);
      t.member(this->second);
   }
};
```

(a) multiKey

```
template<typename multiKey, typename Comp>
class multi_comp {
   typedef multiKey    key_type;
   typedef Comp        comp_type;
   comp_type           m_comp;
public:
   multi_comp(): m_comp() {}
   multi_comp(comp_type c): m_comp(c) {}
   bool operator() (const key_type& fst_key,
                    const key_type& snd_key) const{
      if (!m_comp(fst_key.first, snd_key.first) &&
         !m_comp(snd_key.first, fst_key.first))
         return fst_key.second < snd_key.second;
      else
         return m_comp(fst_key.first, snd_key.first);
   }
};
```

(b) multiComp

Fig. IV.1. Implementation of multiKey and multiComp

type to Key type. This allows the use of the container to be unaware of the multiplicity. Unordered multimap and unordered multiset also use the same multiKey concept as their GID type. In the ordered associative containers, the GID is sorted in the domain, so we also design a templated multiComp class (Fig.IV.1b) as the comparator of the multiKey to keep GIDs in order.

The distribution classes of STAPL multimap and multiset are derived from the base distribution and associative distribution classes. The former is the common base class that provides essential functionality for all distributions of STAPL parallel containers. The latter is a subclass of the former that provides functionality exclusively for the distribution of STAPL associative containers. The distribution of STAPL multimap and multiset uses the balanced partition as the default partition if the user does not specify it. The balanced partition evenly divides the domain into subdomains that contain contiguous ranges of GIDs. It guarantees that the difference between the sizes of any two subdomains is at most one.

When an element is inserted into a STAPL multimap or a STAPL multiset, its GID will be registered into the vector directory instance of the container. The vector directory is a class built for dynamic-sized containers to manage the distributed GIDs. It can determine on which location a GID is located and allow user to invoke the methods on the location of GIDs without using external entities to know exact locality information.

The container manager of the multimap and multiset containers store the elements of the container mapped to a location by the partition and mapping. After mapping the subdomains to the locations, the elements with the GIDs in one subdomain will be sent to a base container, which is contained in the corresponding location that the subdomain should belong to. A Container Manager is the component that stores base containers and provides the access of the base containers on a location. The associative container manager is a base class implemented for associative container in the STAPL Parallel Container Framework. Deriving from the base class, the container manager classes of STAPL multimap and multiset inherit the basic functionality of supervising the base containers. It knows which base containers elements reside locally. The container manager provides functions that can invoke base container functions on an element without specifying in which base container the element is located.

The base container, also called bContainer, is a subunit that holds a part of the elements to share the workload of STAPL container. The base containers of current STAPL containers are implemented based on their STL counterparts, whose properties are suitable for the requirement of STAPL containers. For example, the base containers of STAPL multimap consist of STL multimap and an integer class member, CID, which is the identifier of this base container. However, other libraries containers or a customized container can also be used to build the base container according to programmer's need. Except directly invoking the function of its STL counterpart, the base containers of STAPL multimap and multiset provide additional functions to support the interaction of PCF components, such as retrieving the identifier of a base container or returning a domain of all the GIDs in this base container. Moreover, to make the GIDs fit in with the features of STAPL multimap and multiset, a

unique function is added to decrease the multiplicity of all the elements with a certain key by one each time an element with that key is erased. It is unique for multiple associative containers, whose GID is a key-multiplicity pair.

To apply a method of STAPL multimap or multiset on a given element, the element's key will be first translated to the GID type, multiKey, and then invoke the related function of the container distribution. If the GID exists in the domain, the distribution will find which location the element is located and invoke the function of the location's container manager. The container manager will find in which base container the element is contained. Finally, the function of base container is called to access the element.

CHAPTER V

PERFORMANCE EVALUATION

This chapter presents an experimental evaluation of our work. We tested the STAPL multimap and multiset containers and evaluated the scalability of these data structures and their methods. Scalability is a characteristic that indicates the ability of an application to efficiently deal with the data using increasing numbers of parallel processing units (processes, cores, threads etc.). We did both strong and weak scaling tests on three functions: insert, find, and erase. They are the most frequently used functions that greatly affect the speed of data processing when we are using the STAPL multimap and multiset containers in a parallel program.

These experiments were conducted on a Cray XE6m [16] with 24 compute nodes; 12 nodes have 2 AMD 16-core 64-bit 'Interlagos' 2.1 GHz CPUs; the other 12 nodes that contain accelerators have 1 16-core 64-bit 'Interlagos' 2.1 GHz CPU. Our system uses Gemini ASICs for inter-processor communication.

Strong Scalability Evaluation on STAPL Multimap and Multiset

Strong scaling refers to a test of the application running time using an increasing number of processors and a fixed input size. In strong scaling tests, an application is considered to scale linearly if the ratio of the performance improvement is equal to the ratio of the increased number of processors. The scale of running the application with k processors, S_k, is defined as the ratio between t_1 and t_k, where t_1 is the amount of time to complete a work unit with 1 processor and t_k is the amount of time to finish the same unit of work with K processors.

$$S_k = \frac{t_1}{t_k}$$

There is run-to-run variability in the execution time when we run the same program with a fixed number of processors. To minimize the impact of that variability, we collect the execution time for a given processor count 32 times, and then report the mean and 95% confidence interval using a normal distribution. The pseudocode of how we collect the execution time is listed below:

```
N = number of input elements;
P = number of processors;
timer.start();
{
  for N/P local elements:
    apply methods on each element;
}
exection_time = timer.stop();
```

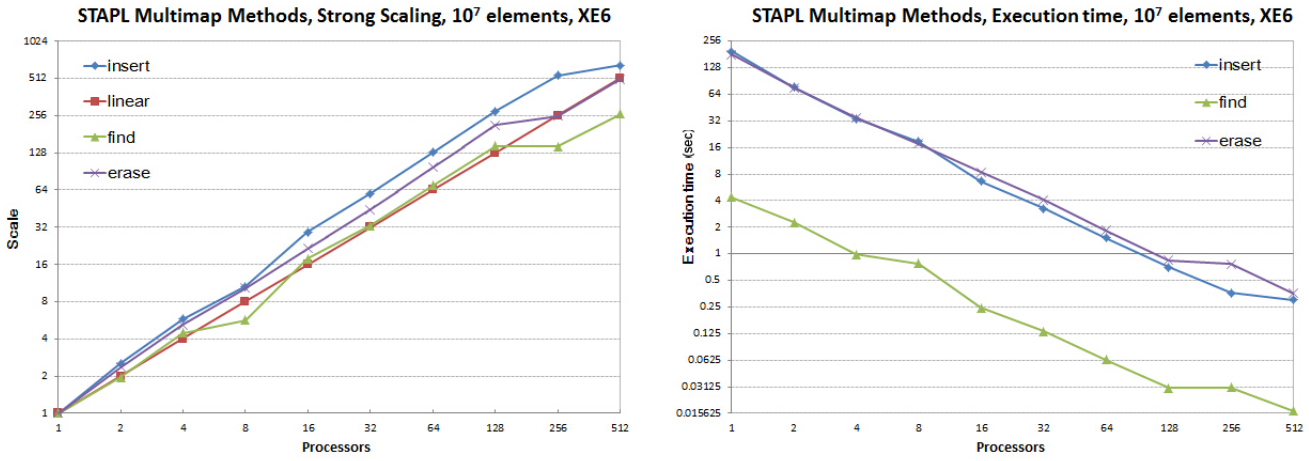

Fig. V.1. Strong Scaling Test for STAPL multimap using 10,000,000 elements

From Fig.V.1 and Fig.V.2, we can see that STAPL multimap and multiset show a good scalability for all of three functions. It is normal that the scales of these methods are larger than their linear values. The multimap and multiset base containers are implemented as a

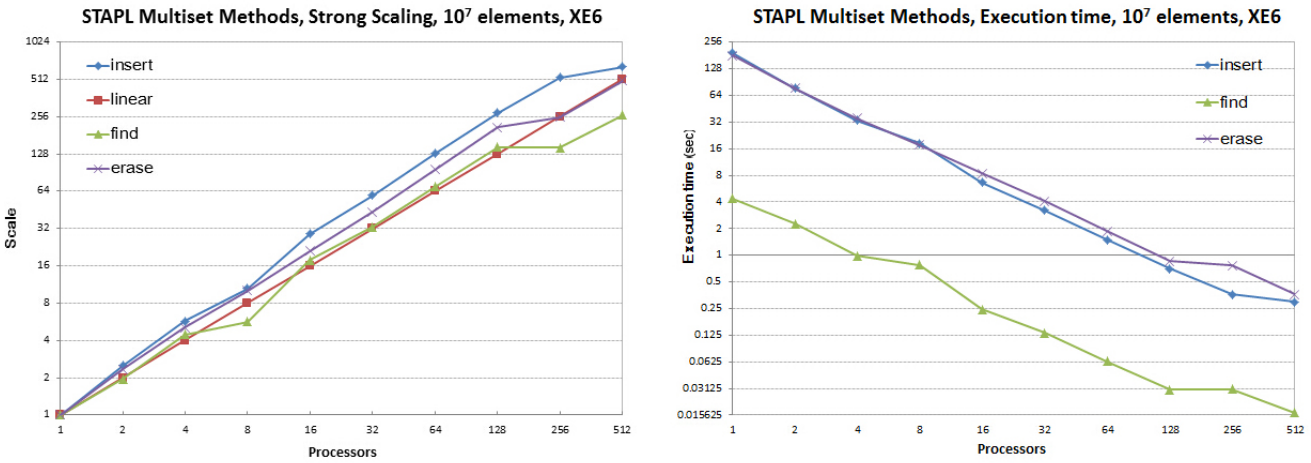

Fig. V.2. Strong Scaling Test for STAPL multiset using 10,000,000 elements

red-black tree to arrange the order of the elements they contain according to the key value. The average time of insert, find and erase function is $O(logn)$. Thus, if we apply one of three functions on n elements, the total amount of time will be $O(nlogn)$. Because in the strong scaling test, the total number of elements n is constant, when we double the number of processors, each processor will handle $n/2$ elements, which take $O(\frac{n}{2}log\frac{n}{2})$ time. The scale between these two levels can be represented as:

$$\frac{O(nlogn)}{O(\frac{n}{2}log\frac{n}{2})} = \frac{nlogn}{\frac{n}{2}(logn-1)} = \frac{2logn}{logn-1} > 2$$

It means theoretically, we can cut more than half of the execution time by using twice the number of processors. The results we showed in Fig.V.1 and Fig.V.2 are closely matching the expected values. The decrease at 8 processors is because that the application fully uses all 8 cores that are located on one die of the processor, the competition and bandwidth limits the processing speed. After retesting the application using 8 cores that 4 of them are from a different die, we got nearly 10% improvement on the scalability. Another issue is that we lost some scalability on the higher processor counts. One reason is that when the number of elements on each processor is small, the weight of the overhead to use additional processors

becomes heavier on the performance. We will evaluate the performance with a larger amount of data later to check if there are other factors that cause this decrease.

Weak Scalability Evaluation

Weak scaling is another basic method of evaluation that measures the parallel performance of a given application. In this test, the number of elements stored on each location is constant no matter how many processors we use. Weak scaling differs from the strong scaling, which focuses on the measurements of CPU limitation (CPU-bound), by emphasizing the presentation of the memory effect (memory-bound) on the parallel application. If the amount of time to complete a work unit with 1 processing element is t_1, and the amount of time to complete K of the same work units with K processors is t_k, the weak scaling efficiency is given as:

$$S_k = \frac{t_1}{t_k}$$

We report the normalized execution time, T_k, which is defined as:

$$T_k = \frac{t_k}{t_1} \times 100\%$$

This will show us the percent increase in execution time as processor count increases. Because the workload of each processor will not change, in theory, the execution time should be fixed as well. Thus, when we double the number of processors, the smaller variation of execution time, the better efficiency we have. From the graphical view, an ideal scaling should be a horizontal line.

Fig.V.3 shows the scalability of multimap and multiset in the weak scaling test. We can see that the scales of the execution time on each case are steady; the connecting line is approximately horizontal especially when number of running processors is larger than 2. The increase from 1 to 2 processors is because of the communication overhead between processors. The problems that cause the unexpected increase when using 8 cores and 512

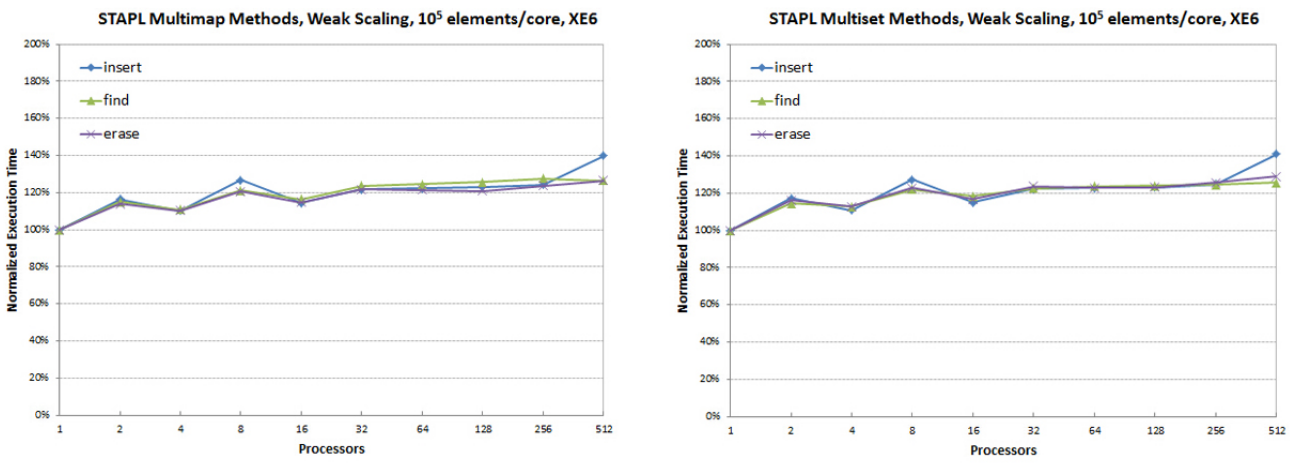

Fig. V.3. Weak Scaling Test for STAPL multimap and multiset using 100,000 elements on each processor

cores are the same as those we discussed in the strong scaling test. Basically, the variation of the execution time on the remaining processor counts is very small. We can infer that our application also has steady weak scalability since the weak scaling efficiency is the reciprocal of the normalized execution time.

CHAPTER VI

CONCLUSION

In this paper, we presented the STAPL multimap and multiset, two distributed multiple associative containers. These two containers are widely used in many retrieval applications that need to deal with big data efficiently. The implementation of multimap and multiset in STAPL provides ordered distributed containers for data processing based on the keys, which are not unique. We presented the design and implementation of STAPL multimap and multiset and evaluated the performance of our implementation with a large constant input and different number of processors. The results of the experiments show a good scalability of our implementation.

Future research on STAPL multimap and multiset containers will investigate the scalability issues identified and add more functionality that helps users to compose parallel applications easily. We will evaluate the scalability of our implementation with a larger input and test the find function with varying amounts of remote access. Another design issue is that the domain ranges in the default constructors of these two containers are currently fixed. We are trying to find a more intelligent way to precisely determine the domain range according to the specific use cases to improve the distribution of the data. Overall, we will focus on reducing the overhead of communication and increasing the concurrency of our implementation in STAPL.

REFERENCES

[1] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Tkachyshyn, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger. *STAPL: Standard template adaptive parallel library*. In Haifa Experimental Systems Conference, Haifa, Israel, May 2010.

[2] Musser, David R, Gilmer Derge, and Atul Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Second Edition, 2001.

[3] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedhal Mourad, Jeremy Vu, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger. *The STAPL Parallel Container Framework*. In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP), Feb 2011.

[4] Antal Buss, Adam Fidel, Harshvardhan, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger. *The STAPL pView*. InWkshp. on Lang. and Comp. for Par. Comp. (LCPC), Oct 2010. Also, Technical Report, TR10-001, Parasol Laboratory, Department of Computer Science, Texas A&M University, Jul 2010.

[5] Nathan Thomas, Steven Saunders, Tim Smith, Gabriel Tanase, Lawrence Rauchwerger. *ARMI: A High Level Communication Library for STAPL*. Parallel Processing Letters, 16(2):261-280, Jun 2006.

[6] Gabriel Tanase, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. *The STAPL pArray*. In Proc. of Workshop MEDEA, pp. 81-88, Brasov, Romania, Sep 2007.

[7] Gabriel Tanase, Xiabing Xu, Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Tkachyshyn, Timmie Smith, Nathan Thomas, Mauro Bianco, Nancy M. Amato, Lawrence Rauchwerger. *The STAPL pList*. In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Newark, Delaware, Oct 2009.

[8] Gabriel Tanase, C. Raman, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. *Associative Parallel Containers In STAPL*. Lecture Notes in Computer Science, 5234/2008:156-171, 2008. Also, In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Oct 2007.

[9] Harshvardhan, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger. *The STAPL Parallel Graph Library*. In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Tokyo, Japan, Sep 2012.

[10] Bjarne Stroustrup. *Programming Principles and Practice Using C++*. Addison-Wesley, second edition, 2009.

[11] Tyler J. Biehle, Nancy M. Amato, Timmie Smith. *Unordered Associative Containers in STAPL*. Undergraduate Research Program, Texas A&M University, May 2014.

[12] 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.

[13] E. Johnson. *Support for Parallel Generic Programming*. PhD thesis, Indiana University, Indianapolis, IN, 1998.

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

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

[16] Cray: System description *Cray XE6m (XK7m-200)*. 2014. [Online]. Available: https://parasol.tamu.edu/pwiki/index.php/Cray:_System_description.