THE STAPL PLIST

A Thesis

by

XIABING XU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2010

Major Subject: Computer Science

THE STAPL PLIST

A Thesis

by

XIABING XU

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2010

Major Subject: Computer Science

ABSTRACT

The STAPL pList. (December 2010)

Xiabing Xu, B.S., Jilin University, P.R.China

Chair of Advisory Committee: Dr. Nancy M. Amato

We present the design and implementation of the Standard Template Adaptive Parallel Library (STAPL) `pList`, a parallel container that has the properties of a sequential list, but allows for scalable concurrent access when used in a parallel program. The STAPL is a parallel programming library that extends C++ with support for parallelism. STAPL provides a collection of distributed data structures (`pContainers`) and parallel algorithms (`pAlgorithms`) and a generic methodology for extending them to provide customized functionality. STAPL `pContainers` are thread-safe, concurrent objects, providing appropriate interfaces (`pViews`) that can be used by generic `pAlgorithms`.

The `pList` provides Standard Template Library (STL) equivalent methods, such as insert, erase, and splice, additional methods such as split, and efficient asynchronous (non-blocking) variants of some methods for improved parallel performance. List related algorithms such as list ranking, Euler Tour (ET), and its applications to compute tree based functions can be computed efficiently and expressed naturally using the `pList`.

Lists are not usually considered useful in parallel algorithms because they do not allow random access to its elements. Instead, they access elements through a serializing traversal of the list. Our design of the `pList`, which consists of a collection of distributed lists (base containers), provides almost random access to its base containers. The degree of parallelism supported can be tuned by setting the number of base containers. Thus, a key feature of the `pList` is that it offers the advantages of a classical list while enabling scalable parallelism.

We evaluate the performance of the STAPL `pList` on an IBM Power 5 cluster and on a CRAY XT4 massively parallel processing system. Although lists are generally not considered good data structures for parallel processing, we show that `pList` methods and `pAlgorithms`, and list related algorithms such as list ranking and ET technique operating on `pList`s provide good scalability on more than $16,000$ processors. We also show that the `pList` compares favorably with other dynamic data structures such as the `pVector` that explicitly support random access.

To my wife and my parents

ACKNOWLEDGMENTS

I want to thank my advisor, Dr. Nancy Amato, for her supervision and support over the last three years. I joined the group in 2007, and later Dr. Nancy Amato suggested that I work in the research area related to parallel computing. I found this to be an interesting subject, full of challenge, and I enjoy it a lot. Dr. Amato was very supportive when I decided to change from a Ph.D. program to a master's program in order to stay close with my wife. I appreciate Dr. Amato very much for supporting my decision.

I also want to thank Dr. Lawrence Rauchwerger, Dr. Jennifer Welch and Dr. Marvin Adams for suggestions and help on research. Their comments and feedback on the thesis were helpful.

Gabriel Tanase, who is a senior student in our group, has given me lots of help since I joined the group. He is very nice and patient with all my questions, and behaves like a teacher and a close friend all the time.

I also want to thank every member in the STAPL group who also helped me. I could not have finished this work without their help.

I would like to thank my parents for their consistent support. Finally, I want to thank my wife for being an assistant in my life and always supporting whatever I want to do.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

INTRODUCTION

Parallel programming is becoming mainstream due to the increased availability of multiprocessor and multicore architectures and the need to solve larger and more complex problems. To help programmers address the difficulties of parallel programming, we are developing the Standard Template Adaptive Parallel Library (STAPL) [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11].

STAPL is a parallel C++ library with functionality similar to STL, the C++ Standard Template Library. STL is a collection of basic algorithms, containers and iterators that can be used as high-level building blocks for sequential applications. STAPL provides a collection of parallel algorithms (`pAlgorithms`), parallel and distributed containers (`pContainers`), and `pViews` to abstract the data access in `pContainers`. These are the building blocks for writing parallel programs using STAPL. An important goal of STAPL is to provide a high productivity development environment for applications that can execute efficiently on a wide spectrum of parallel and distributed systems.

`pContainers` are collections of elements that are distributed across a parallel machine and support concurrent access. STAPL provides a unified approach for developing `pContainers`. It uses object-oriented technology to build distributed thread-safe containers that can easily be extended and customized. This approach allows us to provide a large variety of `pContainers` such as `pArray` [10], `pMatrix`[4], associative containers such as `pMap` and `pSet` [8], `pVector` and `pGraph`.

Distributed and concurrent data structures are an active research area that has

The journal model is *IEEE Transactions on Automatic Control.*

received significant attention. However, most of the research focuses on array-like static data structures. Due to their intuitively sequential structure, linked list data structures have presented a challenge to be used efficiently in parallel, and considerable effort has been devoted to designing efficient concurrent linked list data structures [12, 13, 14, 15, 16, 17].

A. Our Contribution

In this thesis, we present the design and implementation of the STAPL `pList`, a parallel container that has the properties of a sequential list, but allows for scalable concurrent access when used in a parallel program. In particular, the `pList` is a distributed doubly linked list data structure that is the STAPL parallel equivalent of the STL list container. The `pList` interface includes threadsafe methods that are counterparts of the STL list such as insert, erase, and splice, additional methods such as split, and asynchronous (non-blocking) variants of some methods for improved performance in a parallel and concurrent environment. As described in more detail before, a summary of our contributions includes:

- a novel design that supports an arbitrary and tunable degree of random access to match desired degree of parallelism;
- interfaces that enable the natural expression of list related algorithms that are efficient to execute.

Lists are not usually considered useful in parallel algorithms because they do not allow random access to its elements. Instead, they access elements through a serializing traversal of the list. Our design of the `pList`, which consists of a collection of distributed lists (base containers), provides almost random access to its base containers. The degree of parallelism supported can be tuned by setting the number of base containers. Thus, a key feature of the `pList` is that it offers the advantages of a classical list while enabling scalable parallelism.

List related algorithms are shown to be expressed naturally using `pList`. Under the STAPL programming model, an algorithm is expressed as a combination of work function and data. Algorithms such as list ranking, Euler Tour (ET) and its applications all can be naturally expressed using this model. In this thesis, we examine various list ranking algorithms including a classic pointer jumping algorithm, a point-to-point synchronized version of the pointer jumping algorithm and an implementation of later in STAPL. We also describe an implementation of the classic ET technique in STAPL, that uses a list ranking algorithm internally. We study the Euler Tour and four of its applications including rooting a tree, postorder numbering, computing vertex level and computing the number of descendants.

We evaluate the performance of the `pList` on an IBM Power5 cluster and an Opteron-based CRAY XT supercomputer. We analyze the running time and scalability of different `pList` methods as well as the performance of different algorithms using `pList` as data storage. We also compare the `pList` to the `pArray` and `pVector` to understand the relative trade-offs of the various data structures. Our results show that the `pList` outperforms the `pVector` when there are a significant number of insertions or deletions. List ranking, Euler Tour (ET) computation and use of the ET technique to compute tree based functions such as rooting a tree, postorder numbering, computing vertex level and computing the number of descendants are are evaluated to show scalable performance when the `pList` is used as the intermediate data structure.

B. Outline of Thesis

Chapter II describes related work to this research area. Chapter III gives an overview of STAPL and its components. Chapter IV describes the design and implementation of the `pList`, and its interface. Chapter V illustrates how list ranking can be implemented using our `pList`, and the application of the `pList` in the Euler Tour technique and its applications to compute tree based functions. Chapter VI shows the performance of the pList methods and of applying different parallel algorithms on the `pList`. Chapter VII concludes this work.

Portion of this work has been published as "STAPL: Standard Template Adaptive Parallel Library" in May 2010 [3] and "The STAPL `pList`" in October 2009 [9].

CHAPTER II

PRELIMINARIES AND RELATED WORK

This chapter presents preliminaries and related work. Preliminaries first describes in general the linked list data structure and its characteristics. The Standard Template Library (STL) List which is a generic linked list data structure is then introduced. Related work discusses previous work on concurrent data structures, parallel programming languages and parallel libraries, and list related algorithms such as list ranking and Euler Tour.

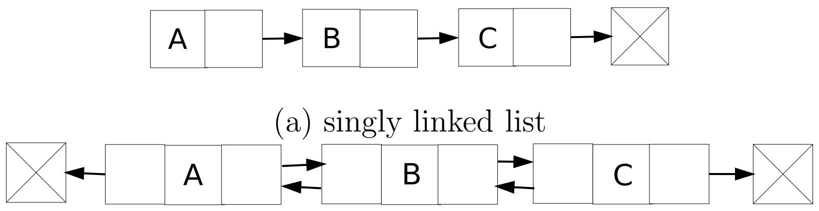
A. Preliminaries

This section presents the basics of the linked list data structure and the STL (Standard Template Library) list.

1. Linked List Data Structure

The linked list in its most basic form is a collection of elements that together form a linear ordering. There are different variants such as a singly linked list, doubly linked list and circularly linked list [18]. Each element of the linked list is called a node which stores the data and pointers. Singly linked list only stores pointers to next node (Figure 1(a)) and doubly linked list stores pointers to previous and next node (Figure 1(b)). The first and last node of a linked list are called the head and tail of the list, respectively. The elements of the list can be traversed by following the pointers to next node.

The linked list is one of the fundamental data structures which is developed as early as 1960s [19, 20]. It plays a very important role in real world applications and is used by many areas such as the file system in many operating systems, genetic



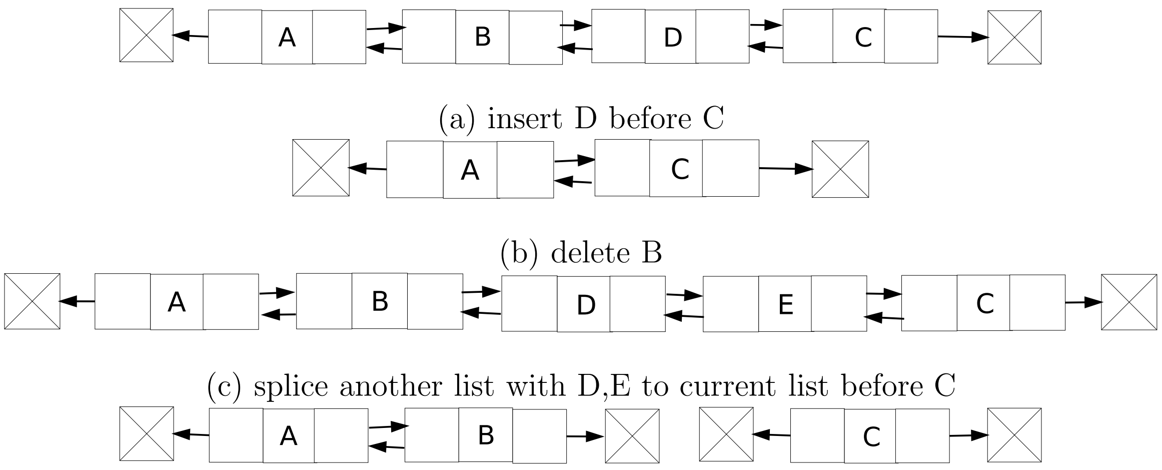
(a) singly linked list

(b) doubly linked list

Fig. 1. Example of a singly (a) and a doubly (b) linked list. A, B and C represent the data of each node. Each arrow represents a pointer to the previous or next node.

algorithms, built in data structures in languages such as Lisp and Scheme, and serves as the basic data structure for other data structures such as stacks. Also many variants have been proposed and implemented to obtain better performance.

Given a doubly linked list (Figure 1(b)) the previous node of head and the next node of tail point to a sentinel node (NULL in most implementations). Such a common doubly linked list has many characteristics (same for singly linked list) :



(a) insert D before C

(b) delete B

(c) splice another list with D,E to current list before C

(d) split current list to two lists at C

Fig. 2. Result lists of applying methods insert, delete, splice and split to the doubly linked list in Figure 1(b).

- Given a pointer to a specific node, insertion of an element at a specific node is a constant time operation. (Given a pointer to C, insertion of D results in Figure 2(a). Four operations to perform: update next pointer of B to point D; set previous pointer of D to B; update previous pointer of C to point D; set next pointer of D to C.)
- Given a pointer to a specific node, deletion of an element at a specific node is a constant time operation. (Given a pointer to B, deletion of B results in Figure 2(b). Three operations to perform: update next pointer of A to point c; update previous pointer of C to point A; delete B.)
- Given a pointer to a specific node of list1, splice list2 before that node is a constant time operation. (Given a pointer to C, splice another list with D,E results in Figure 2(c). Four operations to perform: update next pointer of B to point D; set previous pointer of D to B; update previous pointer of C to point E; set next pointer of E to C.)
- Given a pointer to a specific node, split one list to two lists at that node is a constant time operation. (Given a pointer to C, split current list results in two lists Figure 2(d). Four operations to perform: update next pointer of B to the sentinel node; set previous pointer of D to B; update previous pointer of C to point D; set next pointer of D to C.)
- Does not support random access. (In order to access C, you have to start from A, traverse the list by following pointer to next node until reach C)
- Maybe slow to traverse because elements maybe scattered in memory.

2. STL List

The STL list is one of the many generic containers provided by the Standard Template Library (STL). It is a doubly linked list which supports both forward and backward traversal and special methods such as splice which are constant time operations. Also splice does not invalidate the iterators (iterators are a generalization of pointers: they are objects that point to other objects) to list elements and deletion only invalidates iterators to elements that are deleted.

Compared to other STL containers such as the vector and deque, the STL list has several differences: Insertions and deletions at any position in a list are constant time operations, not just at one or both ends; Insertions never invalidate any iterators, and deletions invalidate only iterators that refer to the deleted element. The lack of random access means that many of the STL generic algorithms cannot operate on lists, including `random shuffle, random sample, nth element, make heap, is heap, sort heap, sort, stable sort, partial sort`.

When many insertions or deletions are required in interior positions and when random access is not needed, then the list data structure may be a good choice.

B. Related Work

1. Concurrent Data Structures

Significant research has been done in the area of distributed and concurrent data structures. Most of this effort focuses on how to implement concurrent objects using different locking primitives or how to implement concurrent data structures without locking, namely lock free data structures [12, 13, 14, 15, 16, 17, 21, 22]. Table I summarizes the previous work, shows the operations the data structures provide, the architectures they support, and the primitives they use.

Valois [12] was the first to present a non-blocking singly-linked list data structure by using the Compare&Swap (CAS) synchronization primitives rather than locks. The basic idea is to use auxiliary nodes between each ordinary node to solve the

Table I. Concurrent data structure related work comparison. CAS: compare and swap operation. $\overline{n}_E$: the average number of elements in the list during any execution E. $\overline{c}_E$: the average operation contention during E. m_E: the total number of operations invoked during E.

Paper	**Operations**	**Architecture**	**Primitives**	**Complexity**
Paul [22] (Lock-based concurrent list)	Insert front end, traverse	shared	Lock-based	Worst case amortized cost is linear in length of list plus contention
Valois [12] (First lock free algo. Uses auxiliary nodes between each node)	Insert delete anywhere, traverse	shared	Using Single CAS synchronization primitive	$\Omega(m_E)$ [16]
Harris [14] (Two CAS operations used, one to mark and the other one to delete)	Insert delete anywhere, traverse	shared	Using Double CAS synchronization primitive	$\Omega(\overline{n}_E\overline{c}_E)$ [16]
Michael [15] (First CAS-based lock-free list-based set algo)	Insert delete anywhere, traverse	shared	CAS-based, compatible with all lock-free memory management methods	Worst case amortized cost is linear in length of list plus contention
Fomitchev [16] (Combination of the technique of marking node and backlink pointers)	Insert delete anywhere, traverse	shared	Using Single CAS synchronization primitive	$\Omega(\overline{n}_E + \overline{c}_E)$
Tanase [9] (STAPL `pList`)	Insert, delete, traverse, etc.	shared & distributed	Lock-based	Table III

concurrent issues. Subsequently Michael and Scott [13] fixed a known bug in [12].

Later, Harris [14] proposed another lock-free implementation. Two CAS operations are used in this implementation to deal with the problem of concurrent insertion and deletion. The first one is used to mark the next field of the deleted node and the second one is used to delete the node. If during the deletion a concurrent insertion accesses the marked field of the deleted node, it should halt the insertion and retry later. It will proceed successfully if it tries after the second CAS and before any other list operation. The correctness is proved and also a comparison with Valois [12]'s algorithm and the lock-based implementation is done. The result shows that Harris's algorithm has better performance compared to previous algorithms.

Another lock-free linked list was proposed by Michael [15]. It points out that previous work for non-blocking data structures shows many drawbacks including size inflexibility, dependence on some primitives not supported in hardware, and dependence on some inefficient memory management techniques. In this paper, the author first proposes a CAS-based lock-free list-based set algorithm which is compatible with all lock-free memory management methods. Experimental results shows that in all lock-free cases, the new algorithm has better performance by a factor of 2.5 or more compared to [14].

Fomitchev [16] showed a linked list based on the techniques of node marking [14] and using backlink pointers [17]. A marking bit is used the same way as in [14] to mark the next field of a deleted node. A backlink pointer is used to track the predecessor of the deleted node so that other operations can proceed efficiently. A flag bit is used to indicate the next node in the list that is going to be deleted. Improved performance in [16] derives from the coordinated use of the marking bit, backlink pointer and flag bits.

In summary, previous work focuses on different concurrent list implementations for shared memory architectures, emphasizing the benefits of non-blocking implementations in comparison with lock based solutions. In contrast, the `pList` and the other STAPL `pContainers` are designed to be used in both shared and distributed memory environments. By default `pList` and other `pContainers` use lock based solutions, but non-blocking concurrent data structures discussed can be integrated into STAPL as customization for current design.

2. List Algorithms

List ranking is a fundamental technique used in many parallel algorithms. The list ranking problem is to compute the distance of each list element from the head or tail of the list. Sequentially, it can be done by simply traversing the list from beginning to the end. Much research effort has been devoted to solving this problem efficiently in parallel [23, 24, 25, 26, 27, 28, 29]. Wyllie [27] proposed the first parallel list ranking algorithm which is based on the Pointer Jumping technique. When P is equal or larger than N where P is the number of processors and N denotes the number of elements stored in the list, it takes $O(\log N)$ time using $O(N \log N)$ operations running on EREW PRAM. When P is less than N and each processor with $\frac{N}{P}$ elements, it takes $O(\frac{N}{P} + \log P)$ time and $O(N + P \log P)$ work. After that, many parallel list ranking algorithms were proposed to improve the work or time. The first optimal $O(\log N)$ time and $O(N)$ work algorithm using $\frac{N}{\log N}$ processors running on EREW PRAM model was proposed in [28]. And another optimal $O(\log N)$ time and $O(N)$ work algorithm using $\frac{N}{\log N}$ processors under EREW PRAM model is described by [23] later. Another $O(\frac{N}{P})$ time and (N) work optimal algorithm where P is equal or less than $\frac{N}{\log N}$ for CRCW PRAM model is presented in [29].

The Euler Tour (ET) is an important representation of a graph for parallel processing, since the ET represents a depth-first-search (DFS) traversal and no other efficient parallel DFS exists so far. The ET of a tree can be used to compute a number of tree functions such as rooting the tree, postorder numbering, vertex levels, and number of descendants [30]. Tarjan and Vishkin [31] first proposed a constant time and linear work algorithm under EREW PRAM model to compute an ET that needs an additional pointer from an edge to its reverse edge in the tree's adjacency lists. Later, Cong and Bader [32] described another algorithm to compute the Euler Tour when the pointer is not given in the adjacency list.

CHAPTER III

STAPL

STAPL [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] is a framework for parallel C++ code development; see Fig. 3. Its core is a library of C++ components implementing parallel algorithms (`pAlgorithms`) and distributed data structures (`pContainers`) that have interfaces similar to the (sequential) C++ standard library (STL) [33]. Analogous to STL algorithms that use *iterators*, STAPL `pAlgorithms` are written in terms of `pViews` so that the same algorithm can operate on multiple `pContainers`. The `pRange` is the STAPL concept used to represent a parallel computation which is essentially a task dependence graph. The runtime system (RTS) and its communication library ARMI (Adaptive Remote Method Invocation [7, 34]) provide the interface to the underlying operating system, native communication library and hardware architecture.

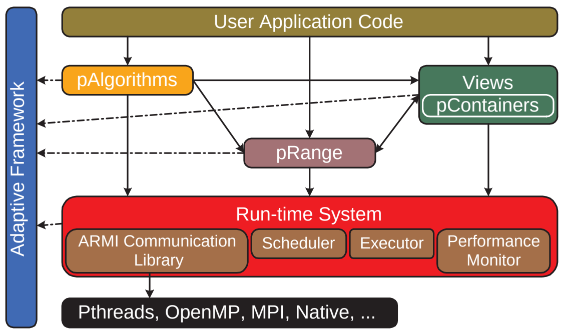


Fig. 3. STAPL overview.

A. Parallel Languages and Libraries

There are several parallel languages and libraries that have similar goals as STAPL [35, 36, 37, 38, 39, 40]. While a large amount of effort has been put into making array-based data structures suitable for parallel programming, more dynamic data structures that allow insertion and deletion of elements have not received as much attention. The PSTL (Parallel Standard Template Library) project [41] explored the same underlying philosophy as STAPL of extending the C++ STL for parallel programming. They planned to provide a distributed list, but the project is no longer active. Intel Threading Building Blocks (TBB) [42] provide thread-safe containers such as vectors, queues and hashmaps for shared memory architectures, but they do not provide a parallel list implementation. Parallel languages such as Chapel [43] (developed by CRAY), X10 [44] (developed by IBM), and many others are all aiming to ease parallel programming and to improve productivity for parallel application development. However, most of these languages only provide high level constructs such as multi-dimensional arrays and facilities to specify the distribution of the arrays. A major difference between STAPL and all these new programing languages is that STAPL is a parallel programming library that is written in standard C++ thus making it compatible with existing applications.

B. STAPL Overview

STAPL `pContainers` are distributed, thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. They are composable and extendible by users via inheritance. Currently, STAPL provides counterparts of all STL containers (e.g., `pArray`, `pVector`, `pList`, `pMap`, etc.), and two `pContainers` that do not have STL equivalents: parallel matrix (`pMatrix`) and parallel graph (`pGraph`). `pContainers` are made of a set of `bContainers`, that are the basic storage components for the elements, and distribution information that manages the distribution of the elements across the parallel machine.

pContainers provide methods corresponding to those provided by the STL containers, and some additional methods specifically designed for parallel use. For example, STAPL provides an insert_async method that can return control to the caller before its execution completes, or an insert_anywhere that does not specify where an element is going to be inserted and is executed asynchronously. While a pContainer's data may be distributed, pContainers offer the programmer a *shared object view*, i.e., they are shared data structures with a global address space. This is supported by assigning each pContainer element a unique global identifier (GID) and by providing each pContainer an internal translation mechanism which can locate, transparently, both local and remote elements. The physical distribution of pContainer data can be determined automatically by STAPL or it can be user-specified.

STAPL algorithms are written in terms of pViews [1], which provide a generic access interface to pContainer data by abstracting common data structure concepts. Briefly, pViews allow the same pContainer to present multiple interfaces, e.g., enabling the same pMatrix to be 'viewed' (or used) as a row-major or column-major matrix, or even as a vector. STAPL pViews generalize the iterator concept — a pView corresponds to a collection of elements and provides an ADT for the data it represents. STAPL pViews enable parallelism by providing random access to the elements, and support for managing the trade-off between the expressivity of the pViews and the performance of the parallel execution. pViews trade additional parallelism enabling information for reduced genericity.

The pRange is the STAPL concept used to represent a parallel computation. Intuitively, a pRange is a task graph, where each task consists of a work function and a view representing the data on which the work function will be applied. The pRange provides support for specifying data dependencies between tasks that will be enforced during execution.

The runtime system (RTS) and its communication library ARMI (Adaptive Remote Method Invocation) provide the interface to the underlying operating system, native communication library and hardware architecture [7]. ARMI uses the remote method invocation (RMI) communication abstraction to hide the lower level implementations (e.g., MPI, OpenMP, etc.). A remote method invocation in STAPL can be blocking (`sync_rmi`) or non-blocking (`async_rmi`). ARMI provides the fence mechanism (`rmi_fence`) to ensure the completion of all previous RMI calls. The asynchronous calls can be aggregated by the RTS in an internal buffer to minimize communication overhead.

The RTS provides *locations* as an abstraction of processing elements in a system. A *location* is a component of a parallel machine that has a contiguous memory address space and has associated execution capabilities (e.g., threads). A location can be identified with a process address space. Different locations can communicate to each other only through RMIs. Internal STAPL mechanisms assure an automatic translation from one space to another, presenting to the less experienced user a unified data space. For more experienced users, the local/remote distinction of accesses can be exposed and performance enhanced for a specific application or application domain. STAPL allows for (recursive) nested parallelism.

C. Parallel Container Framework (PCF) [45]

The objective of the STAPL *Parallel Container Framework (PCF)* is to simplify the process of developing generic parallel containers. It is a collection of classes that can be used to construct new `pContainers` through inheritance and specialization that are customized for the programmer's needs while preserving the properties of the base container. In particular, the `PCF` can generate a wrapper for any standard data structure, sequential or parallel, that has the meta information necessary to use the data structure in a distributed, concurrent environment. The `PCF` provides a shared object view to allow the programmer to ignore the distributed aspect of the `pContainer` if they so desire. And thread safety is also enabled internally by design. All these allow the programmer to concentrate on the semantics of the container instead of its concurrency and distribution management. Thus, the `PCF` makes developing a `pContainer` almost as easy as developing its sequential counterpart. More details are discussed in [45].

STAPL provides a library of `pContainers` constructed using the `PCF`. These include counterparts of STL containers (e.g., `pVector`, `pList` [9], and associative containers [8] such as `pSet`, `pMap`) and additional containers such as `pArray` [10], `pMatrix`, and `pGraph`.

D. PCF Design

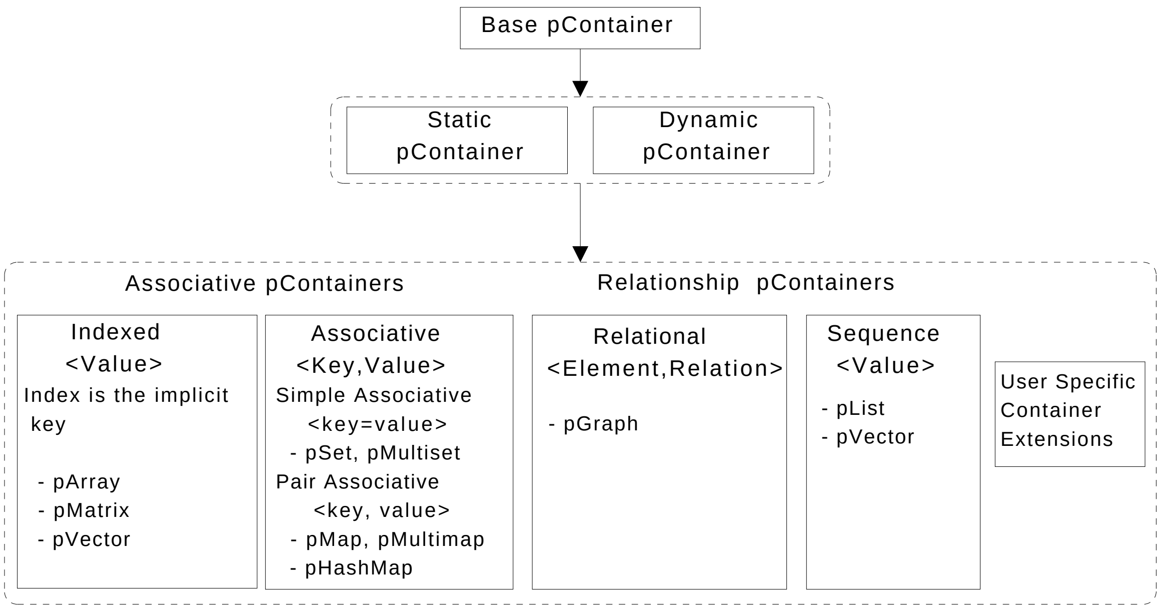


Fig. 4. PCF design.

The `PCF` is designed to allow users to easily build `pContainers` by inheriting from appropriate modules. It includes a set of base classes representing common data structure features and rules for how to use them to build `pContainers`. Figure 4 shows the main concepts and the derivation relations between them; also shown are the STAPL `pContainers` that are defined using those concepts. All STAPL `pContainers` are derived from the `pContainer` base class which is in charge of storing the data and distribution information. The remaining classes in the `PCF` provide minimal interfaces and specify different requirements about `bContainers`. First, the `static` and `dynamic` `pContainers` are tag classes that indicate if elements can be added to or removed from the `pContainer`. The next discrimination is between *associative* and *relational* `pContainers`. In associative containers, there is an implicit or explicit association between a key and a value. For example, in an array there is an implicit association between the index and the element corresponding to that index; we refer to such (multi-dimensional) arrays as indexed `pContainers`. In other cases, such as a hashmap, keys must be stored explicitly. The `PCF` provides an `associative base pContainer` for such cases. The `relational pContainers` include data structures that can be expressed as a collection of elements and relations between them. This includes graphs and trees, where the relations are explicit and may have values associated with them (e.g., weights on the edges of a graph, `pList` in chapter IV), and lists where the relations between elements are implicit.

All classes of the `PCF` have default implementations that can be customized for each `pContainer` instance using template arguments called *traits*. This allows users to specialize various aspects, e.g., the `bContainer` or the data distribution, to improve the performance of their data structures.

1. Shared Object View

Shared object view is an important concept in STAPL which is defined as that each `pContainer` instance is globally addressable. It is provided to relieve the programmer from managing and dealing with the distribution explicitly, unless he desires to do so. The fundamental concept required to provide a shared object view is that each pContainer element has a unique global identifier (GID). The GID provides the shared object abstraction since all references to a given element will use the same GID. Examples of GIDs are indexes for pArrays, keys for pMaps, vertex identifiers for a pGraph, and complex types for `pList` (defined in chapter IV).

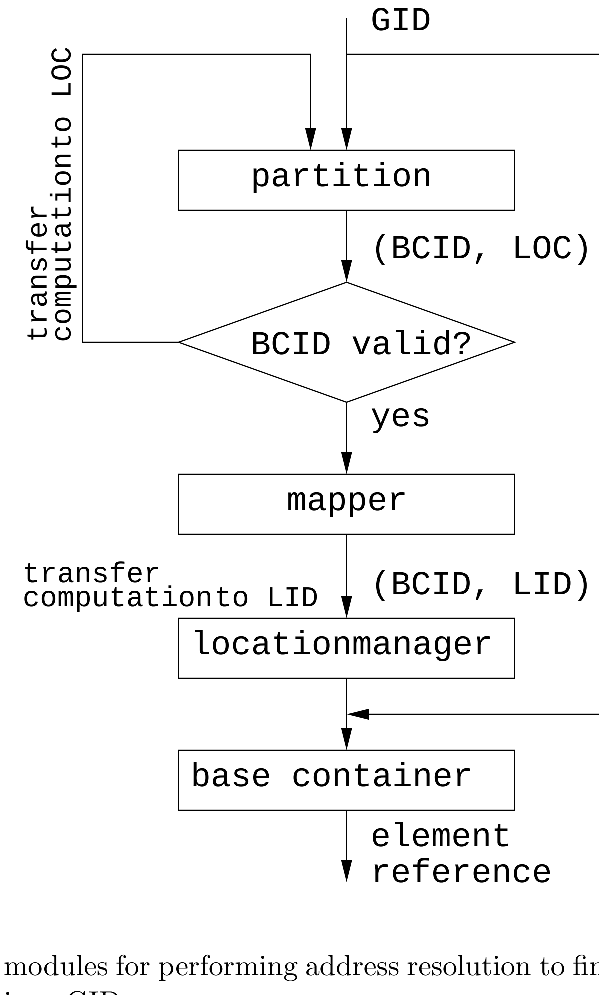


Fig. 5. `pContainer` modules for performing address resolution to find the element reference corresponding to a given GID.

The PCF supports the shared object view by providing an address translation mechanism that determines where an element with a particular GID is stored (or should be stored if it does not already exist). We now briefly mention the PCF components involved in this address translation. The set of GIDs of a pContainer is called a domain. For example, the domain of a pArray is a finite set of indices while it is a set of keys for an associative pContainer. A pContainer's domain is partitioned into a set of non-intersecting sub-domains by a partition class, itself a distributed object that also provides a map from a GID to the sub-domain that contains it, i.e., a directory. There is a one-to-one correspondence between a sub-domain and a base container. In general, there can be multiple base containers allocated in a location, where a location denotes a unit of a parallel machine that has a contiguous memory address space and associated execution capabilities (e.g., threads); a location may, but does not have to, be identified with a process address space. Finally, a class called a partition-mapper maps a sub-domain (and its corresponding base container) to the location where it resides, and a location-manager manages the base containers of a pContainer mapped to a given location.

We now describe how a pContainer method is executed using the above concepts. In Figure 5 we show a flowchart of the address resolution procedure. Given the unique GID identifying a pContainer element, the data-distribution-manager queries the partition class about the sub-domain associated with the requested GID. If the base container (specified by a base container identifier, or BCID) is not available, the partition provides information about the location (LOC) where it might be retrieved, and the process is restarted on that location. If the BCID is available and valid, then the partition-mapper gets information about the location where the base container resides (LID); if the operation is not local, the method is re-evaluated on that location, otherwise the location-manager provides the proper base container address and the operation is performed.

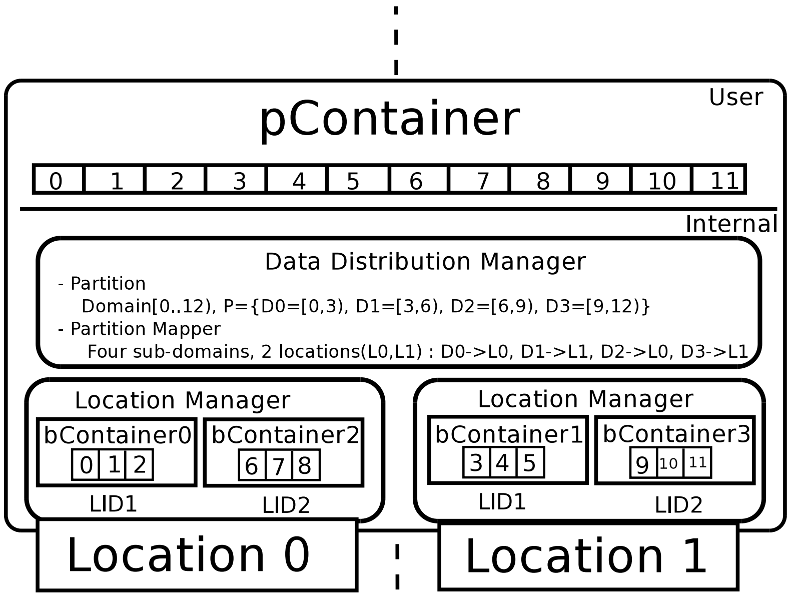


Fig. 6. Example of `pContainer` modules for performing address resolution to find the element reference corresponding to a given GID.

Figure 6 shows an example how the translation mechanism works. Here we have a `pContainer` of 12 elements with GIDS integers in the range [0, 11]. The partition specifies a block size of 3 which results in 4 sub-domains. By using a cyclic mapping we have sub-domains 0 and 2 mapped to location 0 and sub-domains 1 and 3 mapped to location 1. The location manager manages two base containers with LID 0 and 1 on each location. Suppose we want to find element with GID 9. The partition indicates it is in sub-domain D3 and the partition mapper indicates it is in location 1. The location manager in location 1 can tell the LID2 corresponds to sub-domain D3 and it is base container 3. Then 9 is the first element in base container 3.

More details are provided in the context of the `pList` in chapter IV.

CHAPTER IV

STAPL PLIST

The STAPL `pList` is a parallel linked list data structure developed using the `pContainer` framework. This chapter describes the `pList` interface and its design and implementation in STAPL.

A. `pList` Interface

The linked list is a fundamental data structure that plays an important role in many areas of computer science and engineering such as operating systems, algorithm design, and programming languages. A large number of languages and libraries provide different variants of lists with C++ STL being a representative example. The STL list is a generic dynamic data structure that organizes the elements as a sequence and allows fast insertions and deletions of elements at any point in the sequence.

The STAPL `pList` is a parallel equivalent of the STL list with an interface for efficient insertion and deletion of elements in parallel. Other interfaces such as size, testing whether the list is empty or not, accessing front and back elements, pushing elements to the front or back of the list, popping an element from the front and the back of list, etc., are also provided for convenient usage as STL lists. Analogous to STL lists, elements in a `pList` can be accessed through iterators. The STL equivalent methods are shown in the Table II.

Table II. STL equivalent methods of the `pList`. P: the number of locations; M_i: the number of base containers in one location; N: the total number of elements in the list; $T_B(N)$: the time of a broadcast communication of N elements; $T_R(N)$: the time of a reduction communication of N elements; T_ϵ: the time of a single message communication time.

pList Interface	**Description**	**Complexity**
p_list(size_t N, const T& value = T())	Creates a `pList` with N elements, each of which is a copy of `value`. Collective Operation.	$O(N/P + T_B(P))$
size_t size() const	Returns the size of the `pList`.	$O(N + T_R(P))$
bool empty() const	True if the `pList`'s size is 0.	$O(MAX(M_i) + T_R(P))$
T& [front|back]()	Access the first/last element of the sequence.	$O(T_\epsilon + 1)$
void push_[front|back](const T& val)	Insert a new element at the beginning/end of the sequence.	$O(T_\epsilon + 1)$
void pop_[front|back]()	Remove the first element from the beginning/end of the sequence.	$O(T_\epsilon + 1)$
iterator insert(iterator pos, const T& val)	Insert val before position pos and return the iterator to the new inserted element.	$O(T_\epsilon + 1)$
iterator erase(iterator pos)	Erases the element at position pos and returns the iterator pointing to the new location of the element that followed the element erased.	$O(T_\epsilon + 1)$
void splice(iter pos, pList& pl);	Splice the elements of `pList pl` into the current list before the position `pos`. Collective Operation.	$O(T_B(P) + MAX(M_i))$.

All STL equivalent methods require a return value, which in general translates into a blocking call. For this reason, we provide a set of asynchronous (non-blocking) methods, e.g., `insert_async` and `erase_async`. They are generally provided in STAPL and used by `pVector`, `pMap`, `pSet`, etc. Non-blocking methods will return immediately and they do not wait for the communication event to complete, thus allowing for better communication/computation overlap and enabling the STAPL RTS to aggregate messages to reduce the communication overhead [7].

Since there is no data replication, operations such as `push_back` and `push_front`, if invoked concurrently, may produce serialization in the locations managing the head and the tail of the list. For this reason, we added two new methods to the `pList` interface, `push_anywhere` and `push_anywhere_async` (also used by other `pContainers` such as `pVector`), that allow the `pList` to insert the element in an unspecified location in order to minimize communication and improve concurrency. The Non-STL equivalent new methods are shown in Table III.

Table III. Non-STL equivalent methods of the `pList`. P: the number of locations; M: the total number of base containers in `pList`; N: the total number of elements in the list; $T_B(N)$: the time of a broadcast communication of N elements; T_ϵ: the time of a single message communication time in the following sections.

pList Interface	**Description**	**Complexity**
p_list(size_t N, partition_type& ps)	Creates a `pList` with N elements based on the given partition strategy. Collective Operation.	$O(N/P + T_B(P))$
void split(iter pos, pList& pl);	The elements from pos to the end in current `pList` will be removed from current `pList` and inserted to `pList` pl. pl is supposed to be default constructed, if not, old data will be removed before new data is inserted. Collective Operation.	$O(T_B(P) + M)$.
void insert_async(iterator pos, const T& val)	Insert val before pos with no return value.	$O(T_\epsilon + 1)$
void erase_async(iterator pos)	Erases the element at position pos with no return value.	$O(T_\epsilon + 1)$
iterator push_anywhere(const value_type& val)	Push val into one of base containers in the location the method is invoked and return the iterator pointing to the new inserted element.	$O(1)$
void push_anywhere_async(const T& _val)	Push val into one of base containers in the location the method is invoked with no return value.	$O(1)$

The `pList` has both collective and non-collective operations. Non-collective operations are those operations which can be invoked in only one location, and different locations may invoke different number of operations concurrently. Collective operations have to be invoked by all locations and communication may happen to accomplish the work. Each collective operation ends with a global barrier to guarantee all the communication is finished.

1. Complexity of Methods

We define P as the number of locations, M_i as the number of base containers in one location, M as the total number of base containers in `pList`, N as the total number of elements in the `pList`, $T_B(N)$ as the time of a broadcast communication of N elements. $T_R(N)$ as the time of a reduction communication of N elements and T_ϵ as the time of a single message communication time in the following discussion.

The complexity of non-collective operations such as the size method is proportional to the total number of elements in the `pList` and $T_R(P)$. It needs to collect the size information of each base container and then perform a reduction to compute the total size. The empty method is similar to the size method except that the empty method of each base container is constant time. So the complexity of `pList` empty method is proportional to the max number of base containers over all locations and $T_R(P)$. Methods such as back, front, insert and erase may be invoked locally or remotely in a synchronous way. When invoked locally, it is a constant time operation. When invoked remotely, the complexity depends on the time T_ϵ to send a request and receive an answer back. The asynchronous methods such as insert_async and erase_async may be invoked locally or remotely. When invoked locally, they are constant time operations. When invoked remotely, the complexity depends on the time T_ϵ to send a request and $O(1)$ to finish remotely.

The complexity of collective operations such as different constructors take time proportional to the number of elements over locations and $T_B(P)$. By default the constructor will construct an equal number of elements in each location. Broadcast communication is used to broadcast the head and tail base container information in order to be able to traverse from the beginning to the end of `pList`. The complexity of advanced methods such as splice and split is discussed in later sections.

B. `pList` Design and Implementation

The STAPL `pList` is designed using the `PCF` Figure 4. It is a dynamic `pContainer` and derives from the sequence `pContainer`. Specialized *Global Unique Identifier (GID)* and *base container identifier* (BCID) are designed and different domains and data distributions are used for the `pList`. This section discusses the design and implementation of the various modules.

1. Base Container

The `pContainer` base container is the basic storage unit for data. For the STAPL `pList`, we use the STL `list` as the base container. Most `pContainer` methods will ultimately be executed at the base container level using the corresponding method of the base container. For example, `pList` insert will end up invoking the STL list insert method. The `pList` base container can also be provided to the user so long as insertions and deletions never invalidate iterators, and so that base containers provide the *domain* interface (see below). Additional requirements are relative to the expected performance of the methods (e.g., insertions and deletions should be constant time operations).

Within each location of a parallel machine, a `pList` may store several base containers and the `pList` employs a `location-manager` module to allocate and handle them. The `pList` has the global view of all of the base containers and knows the order between them in order to provide a unique traversal of all its data. For this reason each base container is identified by a globally unique *base container identifier* (BCID). The `pList`, needs a base container identifier that allows for fast dynamic operations. For example during the splice operation, base containers from a `pList` instance need to be integrated efficiently into another `pList` instance while maintaining the uniqueness of their BCIDs. For these reasons the BCID for the `pList` base containers is currently defined as follows:

```
typedef std::pair<plist_bcontainer*, location_identifier> BCID
```

This BCID will provide the location information and a pointer to the base container. It is a unique identifier and the pointer can only be used when the current location is the same as the location specified in the BCID.

2. Global Identifiers (`GID`)

In the STAPL `pContainer` framework, each element is uniquely identified by its `GID`. This is an important requirement that allows STAPL to provide the user with a shared object view. Performance and uniqueness considerations similar to those of the base container identifier, and the list guarantee that iterators are not invalidated when elements are added or deleted lead us to use the following definition for the `pList` `GID`.

```
typedef std::pair<std::list<>::iterator, BCID> GID;
```

This GID contains an iterator pointing to the element in the base container and the base container ID. It is unique since the BCID is unique. With this definition the `pList` can uniquely identify each of its elements.

3. Domain

In the STAPL `pContainer` framework the domain is the universe of `GIDs` that identifies the elements of a particular `pContainer`. A domain also specifies an order that defines how elements are traversed by the iterators of the `pList`. This order is specified by two methods: `get_first_gid()` which returns the first GID of the domain and `get_next_gid(GID)` which returns the next `GID` in the domain of the `GID` provided as argument. The domain can also be specified as the union of sub-domains. The `pList` derives a specialized domain from the `PCF` to support `pList` specific operations. The `pList` base container serves as the storage of data and also the sub-domain by providing domain-specific interfaces.

4. Data Distribution

Data distribution is essential to the performance of the various parallel algorithms. The data distribution of the `pList` is managed by the `PCF` as discussed in chapter III. The `pList` has its own `distribution-manager` which derives from the base `distribution-manager`. In addition to the functionality provided by the base, it is able to manage the distribution information of its advanced methods such as the splice and split operations.

When default constructing a STAPL `pList` with a known size, the balanced distribution is used and each location is allocated an equal number of elements. The balanced distribution will guarantee the best performance if every processing unit of the machine is the same. A constructor to build a `pList` with multiple base containers per location is also provided and these base containers can be linked together in a blocked or cyclic manner. We use N to refer to the total number of elements in `pList` and P to refer to the number of locations. Blocked means to distribute N/P

base containers per location and all the base containers in one location will be linked together before linking to the base containers in the next location. Cyclic means to distribute all base containers in a cyclic fashion over the locations. The choice of which constructor to use is decided by the `pList` users since they have the knowledge of the algorithms they will use and different algorithms benefit from different distributions.

Even if it was initially balanced, the `pList` may reach an imbalance distribution after insertions, deletions, splices or splits. For this reason, the `pList` provides a redistribution method to evenly redistribute the data. The order between the elements is preserved by this operation.

5. pView

In the STAPL framework, `pViews` are the means of accessing data elements stored in the `pContainer` within generic algorithms. STAPL `pAlgorithms` are written in terms of `pViews`, similar to how STL generic algorithms are written in terms of iterators. The `pList` currently supports sequence `pViews` that provide an iterator type and `begin()` and `end()` methods. A `pView` can be partitioned into sub-views. By default the native `pView` of a `pList` matches the subdivision of the list in base containers, thus allowing random access to portions of the `pList`. This allows parallel algorithms to achieve good scalability (see Section VI). The native `pList` `pView` also provides similar interfaces as those provided by the `pList` to support dynamic operations such as inert and delete.

The `pList` `pView` has to perform the validity test for the rank of each base container in order to be used with certain `pAlgorithms` such as those need the reduction with a non-commutative operator, and the rank of each base container of the `pList` has to be computed in this case. The computation of rank will increase the overall

```
1 void plist::insert_async(iterator it, value_type val)
2    Location loc;
3    dist_manager.lookup(gid(it))
4    BCID = part_strategy.map(gid(it))
5    loc = part_mapper.map(BCID)
6    if loc is local
7       location_manager.bcontainer(BCID).insert(gid(it))
8    else
9       async_rmi (loc, insert_async(it, val));
```

Fig. 7. `pList` method implementation.

execution time of the algorithm. When a `pList` is instantiated it initializes the rank of all of its base containers. However, dynamic operations such as splice and split modify the number of base containers of a `pList` and invalidate the ranking of the `pList`'s base containers.

6. Implementation of pList Methods

The `pList` methods are implemented using the `distribution-manager` and the `location-manager`. A typical implementation of a `pList` method that operates at the element level is included in Figure 7 to illustrate how the `pContainer` modules interact. The run-time cost of the method has three constituents: the time to decide the location and the base container where the element will be added (Figure 7, lines 3-5), the communication time to get/send the required information (Figure 7, line 9), and the time to perform the operation on a base container (Figure 7, line 7).

C. pList Advanced Methods

The `pList` provides methods to rearrange data in bulk. These methods are `splice` and `split`, which merge multiple lists and split lists into multiple pieces, respectively.

1. Splice

`splice` is a `pList` method and it is an overloaded name. There are two versions of splice (Figures 8 and 9) whose signatures are

```
void pList::splice(iter pos, pList& pl, iter it1, iter it2);

void pList::splice(iter pos, pList& pl);
```

```
void splice (iterator pos, PLIST& pL) {
  //find gid of the element pointed by pos.
  GID gid = gid_of(pos);
  //find bcontainer id by asking partition strategy
  PS* ps = this->m_dist->get_partition_strategy();
  BCID cid1 = ps->get_info(gid);
  //split bcontainer associated with bcontainer id cid1 to two
  //bcontainers (having id cid1 and cid2) from pos.
  BCID cid2 = split_bcontainer(cid1);

  //update the corresponding link of pL
  cid1 next bcontainer = pL first bcontainer;
  pL first bcontainer previous bcontainer = cid1;
  cid2 previous bcontainer = pL last bcontainer;
  p2 last bcontainer next  bcontainer = cid2;

  //add all bcontainers to this pList in parallel
  for all bcontainers in each location
   this->add_bcontainer(bcontainer)
  //release all bcontainers from pL.
  pL.release();
  //update pL to default constructed state
  pL.reset_cidsinfo();
  pL.add_bcontainer(empty bcontainer);
}
```

Fig. 8. `pList` splice `pList` implementation.

where `iter` stands for an iterator type, `pos` is an iterator of the calling `pList`, `pl` is another `pList`, and the iterators `it1` and `it2` are iterators pointing to elements of `pl`. `splice` removes from `pl` the portion enclosed by `it1` and `it2` and inserts it at `pos`. When `it1` and `it2` is not specified, the `pList` pl will be spliced at `pos`.

We use the same notation from Table II and III for complexity analysis. The complexity of `splice` depends on the number of base containers included within `it1` and `it2`. If `it1` or `it2` points to elements between base containers, then new base

```
void splice (iterator pos, PLIST& pL, iterator f, iterator l) {
  //find gid of the element pointed by pos.
  GID gid = gid_of(pos);
  //find bcontainer id by asking partition strategy
  PS* ps = this->m_dist->get_partition_strategy();
  BCID cid1 = ps->get_info(gid);
  //split bcontainer associated with bcontainer id cid1 to two
  //bcontainers (having id cid1 and cid2) from pos.
  BCID cid2 = split_bcontainer(cid1);

  same process as above to split from iterator f to cidf_prev
  and cidf split from iterator l to cidl_prev and cidl

  //splice [cidf, cidl) between cid1 and cid2
  while (cidf != cidl) {
    BCID cid = cidf;
    cidf = get_next(cidf);
    release cid from pL;
    add cid to *this
    link cid between cid1 and cid2;
  }
 }
```

Fig. 9. `pList` splice range implementation.

containers are generated in constant time using a sequential list splice. When the entire `pList` needs to be spliced, the complexity depends on the maximum M_i among all locations since base containers on every location have to be removed from the old `pList` and inserted into another `pList`. The re-construction of the old `pList` takes $T_B(P)$ since the head and tail BCID needs to be broadcast. So the total time is $O(T_B(P) + MAX(M_i))$.

2. Split

`split` is also a member method of `pList` that splits one `pList` into two. The algorithm for this method is described in Figure 10. It is a `pList` specific parallel method that is implemented based on `splice` with the following signature:

`void pList::split(iterator pos, pList& other_plist).`

When `pList.split(pos, other_plist)` is invoked, the part of `pList` starting at `pos` and ending at `pList.end()` is removed from current `pList` and appended at

```
void split (iterator pos, PLIST& pL) {
  BCID cidf = first cid of pL;
  BCID cidl = last cid of pL;

  iterator end = one past end iterator of pL;
  iterator this_end = one past end iterator of *this pList;
  pL.splice(end, *this, pos,  this_end);

  //traverse first P bcontainers, remove empty bcontainer if more
  //than one bcontainer in that location.
  cidl = next cid of cidl;
  while (cidf != cidl) {
    BCID cid = cidf;
    cidf = get_next(cidf);
    delete bcontainer having BCID cid from pL if more than one
    bcontainer in that location;
  }
}
```

Fig. 10. `pList` split implementation.

the end of the `other_plist`. If any data was stored in the other_plist before split operation, they will be removed and the data inside other_plist will only be the data from current `pList`.

We use the same notation from Table II and III for complexity analysis. In the worst case scenario, the whole `pList` will be splitted to an empty `pList` and another `pList` contains all the data from original `pList`. It takes $O(M)$ to split all base containers and $T_B(P)$ to rebuild the `pList`. So the total time is $T_B(P) + O(M)$.

CHAPTER V

LIST RELATED ALGORITHMS

The linked list is an important data structure and there are algorithms specific to it. In this chapter, we consider two such algorithms, list ranking, which computes the distance of each element from the head or the tail, and the Euler Tour (ET) technique which can be used to compute interesting functions for trees such as rooting a tree and postorder numbering. This chapter first describes the design and implementation of different list ranking algorithms. Next, applications such as Euler Tour and computing tree functions including rooting a tree, postorder numbering, computing vertex level and computing number of descendants using Euler Tour and list ranking are shown to demonstrate the usefulness of STAPL `pList` in parallel algorithms.

A. List Ranking

The classic list ranking problem is to identify the distance for each element in the list from either end of the list. Here we will consider the distance from the beginning. List ranking is a special case of prefix scan in which each element is initialized with a value of 1 and the operation to be performed is addition.

The sequential algorithm simply traverses the entire list and sets the rank of each element if reaches. It takes $O(N)$ for list with N elements. As mentioned in Section $II.B2$, many algorithms have been proposed for parallel list ranking, with the most efficient requiring $O(\log N)$ time and $O(N)$ work using $\frac{N}{\log N}$ processors in EREW PRAM model [23].

The best known algorithm for parallel list ranking is based on the pointer jumping or recursive doubling technique. We first implemented this algorithm with barriers between each step. Next, we designed and implemented a point-to-point synchroniza-

```
Input:
A linked list of N nodes such that
(1) the predecessor of each node i is given by pred(i)
(2) the successor of each node i is given by succ(i)
(3) the pred value of the first node is equal to sentinel value
(4) the succ value of the last node is equal to sentinel value

Output:
For each 0 <= i < N, the distance Rank(i) of node i from the
beginning of the list

Algorithm:
P(i) denotes the current predecessor
S(i) denotes the current successor

begin

for i from 0 to N - 1 pardo
  Rank(i) = 1    //initialize the rank of each element to be 1
  P(i) = pred(i) //initialize the current predecessor to be initial predecessor
  S(i) = succ(i) //initialize the current successor to be initial successor

for i from 0 to N - 1 pardo
  for j from 0 to logN - 1
    send P(i) and Rank(i) to S(i) //send current predecessor and rank to current successor
    send S(i) to P(i)             //send current successor to current predecessor
    global synchronization        //guarantee all the communication complete
    R(i) = R(i) + R_received      //update current rank
    P(i) = P_received             //update current predecessor
    S(i) = S_received             //update current successor

end
```

Fig. 11. LR-glob-sync pointer jumping algorithm for list ranking.

tion version of the pointer jumping algorithm. Also we redesigned and implemented this new algorithm under the STAPL programming model.

1. List Ranking via Pointer Jumping with Global Synchronization (LR-glob-sync)

Pointer Jumping is a very simple technique to compute rank of all elements of a list in parallel. Figure 11 shows pseudo code for the algorithm.

The algorithm first initializes the rank of each element to 1 in parallel. The rank of E_i denotes the number of elements known to be before E_i and is also the number of nodes current node will skip in the next step. When P is equal or larger than N where P is the number of processors and N is the number of nodes. The algorithm takes $O(\log N)$ steps to jump from one end to another end. And the total time is $O(\log N) * (1 + T_B)$ (T_B represents time for a single global barrier) since the algorithm takes $O(\log N)$ steps and each step performs the computation in constant time and a barrier, the total work is $O(N \log N)$. When P is less than N, each processor with $\frac{N}{P}$ nodes. The algorithm takes $O(\frac{N}{P} + \log P * (1 + T_B))$ time and $O(N + P \log P)$ work.

It is important to notice that every step has to be synchronized before proceeding to line 10 in Figure 11. This implies a barrier to synchronize all processing units which is highly inefficient, especially for large scale machines.

Figure 12 shows an example execution of the LR-glob-sync algorithm. Figure 12(a) shows a list with eight elements. The algorithm takes three steps to finish. Figure 12(b) is the result after the first jump. Figure 12(c) refers to the result of the second jump. Figure 12(d) shows the conclusion of the algorithm with the correct rank associated with each element.

2. List Ranking via Pointer Jumping with Point-to-point Synchronization (LR-pt2pt-sync)

In an effort to improve the scalability of the list ranking via Pointer Jumping algorithm, we designed a version that replaced the global barrier of each step with point-to-point synchronization between the communications of the list elements. The point-to-point synchronization version uses an auxiliary data structure associated with each node to store the rank of the different steps. This is a typical trade off between time and space. We are trading space for time since this algorithm will be used to compute ranks based on the base containers and the number of base containers is proportional to the number of locations, which is typically not as large as the number of elements. After the rank is computed, the auxiliary data structure is cleared immediately.

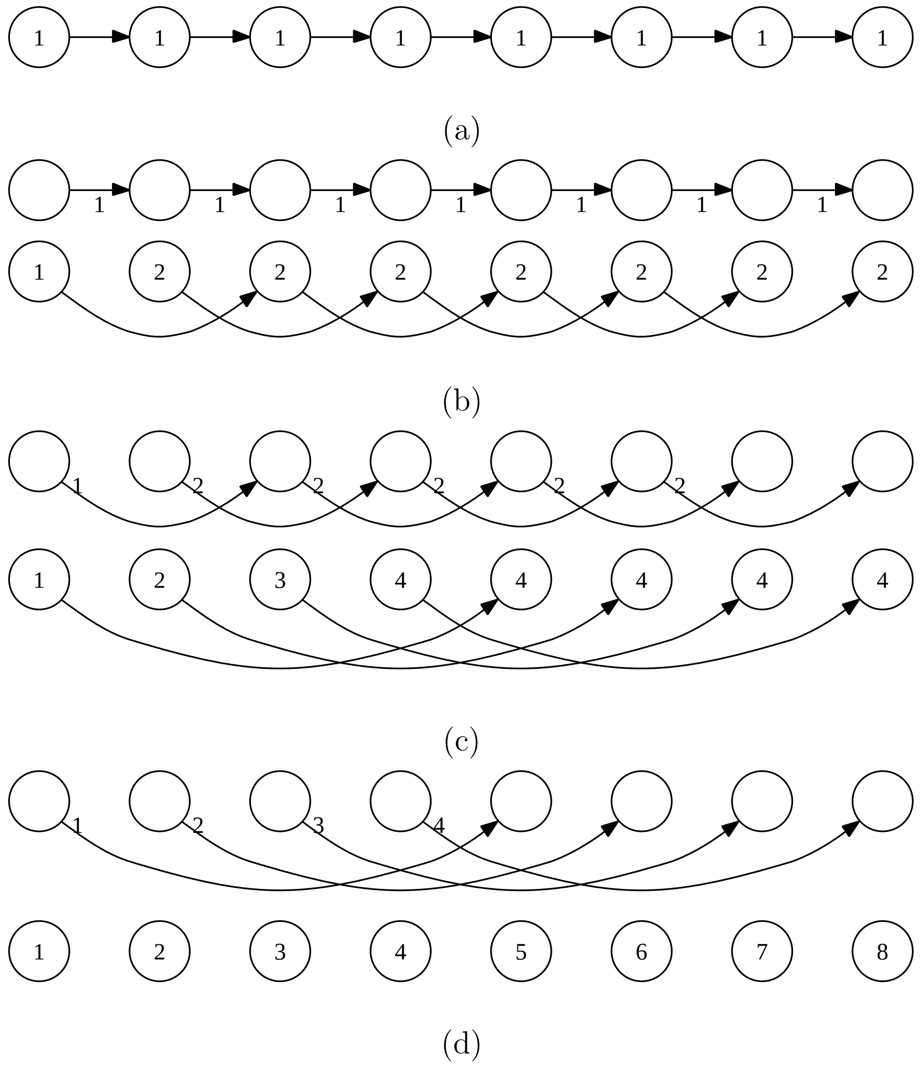


Fig. 12. Illustration of the LR-glob-sync algorithm. (a) A list with eight elements. (b) The result after the first jump. (c) The result after the second jump. (d) The result after the third jump.

Figure 13 illustrates the LR-pt2pt-sync algorithm. For each node of the list, the rank field records the final rank, the vector R stores the rank for different steps, and vectors P and S store the predecessors and successors for different steps. Each node will send its information to its current predecessor and successor and wait to proceed until all the information it needs has been received. There is no explicit global synchronization required among all the processing units, thus we expect this

```
Auxiliary Data Structure:
vector R,P,S store different level rank, predecessor and successor

Algorithm:

begin

for i from 0 to N - 1 pardo
 R(i, 0) = 1       //initialize the rank of each element at first round to be 1
 Rank(i) = 1       //initialize the rank of each element to be 1
 P(i, 0) = pred(i) //initialize the predecessor at first round to be initial predecessor
 S(i, 0) = succ(i) //initialize the successor at first round to be initial successor

for i from 0 to N - 1 pardo
 for j from 0 to logN - 1 do
   send P(i, j) and R(i, j) to S(i, j) //send predecessor and rank to successor at round j
   send S(i, j) to P(i, j)             //send successor to predecessor at round j
   while not receive all
     wait                  //wait until receive the predecessor,successor and rank at round j
   R(i, j+1) = R_received        //update rank at round j+1
   P(i, j+1) = P_received        //update predecessor at round j+1
   S(i, j+1) = S_received        //update successor at round j+1
   Rank(i) = Rank(i) + R(i, j+1) //update rank

end
```

Fig. 13. LR-pt2pt-sync algorithm.

version to scale better than the LR-glob-sync.

The complexity of LR-pt2pt-sync is analogous to LR-glob-sync. When P is equal or larger than N where P is the number of processors and N is the number of nodes. The LR-pt2pt-sync takes $O(\log N)$ steps to jump form one end to another end. And the total time is $O(\log N)$ which takes $O(\log N)$ steps and each step performs only the computation in constant time without any barrier, the total work is $O(N \log N)$. When P is less than N, each processor with $\frac{N}{P}$ nodes. The LR-pt2pt-sync takes $O(\frac{N}{P} + \log P)$ time and $O(N + P \log P)$ work.

3. List Ranking in STAPL (LR-STAPL)

The list ranking algorithm implemented in STAPL follows the LR-pt2pt-sync. Under the STAPL programming model, parallel algorithms are expressed as task graphs called pRanges. Dependences between tasks are noted in task graph and are enforced by STAPL internally - no explicit communication primitives are used in STAPL program.

Figure 14 shows the STAPL version of the algorithm. Tasks in STAPL are specified as a combination of data and the work function which should be applied to the data. Data can be accessed by views. Tasks can be dynamically added to the task graph by using the add_task function, which must specify the work function and the view. Locality information is encapsulated inside the view, so users do not need to be concerned about or be aware of where the task is going to be executed.

The algorithm starts by constructing a LISTRANKING_WF work function for each node with its rank, prev or next and round information. The computation begins by adding initial tasks with views over the previous node and the next node with the proper work function. Whenever a task is executed over a node, it checks whether the task at the previous round has been fired. If not, it does nothing, otherwise, it continues to fire all the tasks from the current round to the final round as long as prev, next and rank at that round have been updated. Whenever reaching a round where not all of the three fields have been updated, the task terminates. Or when reaching the final round, it updates the rank field and terminates the task. When no new tasks are created, and every node reaches the final round with rank field updated, the algorithm ends.

B. Euler Tour and Its Applications

The Euler Tour (ET) is an important representation of a graph for parallel processing. Since the ET represents a depth-first-search traversal, when it is applied to a tree it can be used to compute a number of tree functions such as rooting a tree, postorder numbering, vertex levels, and number of descendants [30]. Good scalability of ET

```
vector A stores accumulated rank up to level i and also serves
as a mark indicating whether the task at this level fired or not

Algorithm:

begin

 1: for i from 0 to N - 1 pardo
 2:   R(i, 0) = 1       //initialize the rank of each element at first round to be 1
 3:   Rank(i) = 1       //initialize the rank of each element to be 1
 4:   A(i, 0) = 1       //initialize the successor at first round to be initial successor
 5:   P(i, 0) = pred(i) //initialize the predecessor at first round to be initial predecessor
 6:   S(i, 0) = succ(i) //initialize the successor at first round to be initial successor

 7: for i from 0 to N - 1 pardo
 8:   LISTRANK_WF wf_prev(S(i, 0), norank,  1)//create a functor with successor at first round
 9:   add_task(wf_prev, View(P(i, 0)))        //add task on the predecessor at first round
10:   LISTRANK_WF wf_next(P(i, 0), A(i, 0), 1)//create a functor with predecessor,rank at first round
11:   add_task(wf_next, View(S(i, 0)))        //add task on the successor at first round

end

//LISTRANK_WF constructed with round, rank and prev or next information.
//When norank, next is chosen, otherwise prev is chosen.
Class LISTRANK_WF
{
 Data:
  prev | next     //data member to hold predecessor or successor
  rank            //data member to hold rank
  round           //data member to hold round

 Member:
  void operator()(Node i)
  {
    if (contain rank)
      R(i, round) = rank                        //update rank at round round
      P(i, round) = prev                        //update predecessor at round round
    else
      S(i, round) = next                        //update successor at round round

    if (A(i, round-1) valid)                    //fire new task if previous task has been fired
      for j from round to final_round           //fire new task from current round to final round
        if (P(i, j), S(i, j) and R(i, j) all set)//if receive predecessor,successor and rank
          A(i, j) = A(i, j-1) + R(i, j)         //update to mark task at round j has been fired
          if (j not final_round)                //fire new task if not reach final round
            LISTRANK_WF wf_prev( S(i, j), norank,  j+1 )
            add_task( wf_prev, View(P(i, j)) )  //fire task on predecessor at round j
            LISTRANK_WF wf_next( P(i, j), A(i, j), j+1 )
            add_task( wf_next, View(S(i, j)) )  //fire task on successor at round j
          else Rank(i) = A(i, j)
        else
          break
  }
}
```

Fig. 14. LR-STAPL algorithm.

algorithm depends on an efficient list ranking algorithm. We will discuss how the `pList` is used in our implementation of ET technique [30] and how the algorithm is implemented in this section.

1. ET Technique Overview

There are many versions of Euler Tour problem such as whether an Euler Tour exists for an arbitrary graph or how to find the Euler Tour if it exists. Here we consider the following Euler Tour problem: Given an undirected tree $T = (V, E)$, let $T' = (V, E')$ be the directed graph obtained from T where each edge $(u, v) \in E$ is replaced by two edges (u, v) and (v, u). Thus T' is an Eulerian graph because each vertex has even degree of edges, and the problem is to find the Euler Tour of T'.

a. ET Construction

Existing sequential algorithms take $O(N)$ time to build an Euler Tour since each edge has to be traversed [46] where N refers to total number of edges in the tree.

The parallel Euler Tour algorithm presented in [30] can be specified by using the successor function which is defined as a mapping function from each edge $e \in E'$ to $s(e) \in E'$ that follows e on the tour. Suppose adjacency list of v is $adj(v) = (u_0, u_1, ...u_{d-1})$ where d is the degree of v. $s(u, v) = (v, u_{(i+1) mod d})$ for $0 \leq i \leq d - 1$. Then it takes constant time for each edge to find the next edge in the tour using successor function when the input tree is represented by the circular adjacency lists with additional pointers. The entire algorithm takes constant time and $O(N)$ work where N is the number of vertices by using EREW PRAM model.

Figure 15 shows an example of a tree and its adjacency list. By applying the successor function specified above, we can get the traversal easily as follows supposing the start edge is $(2, 6)$.

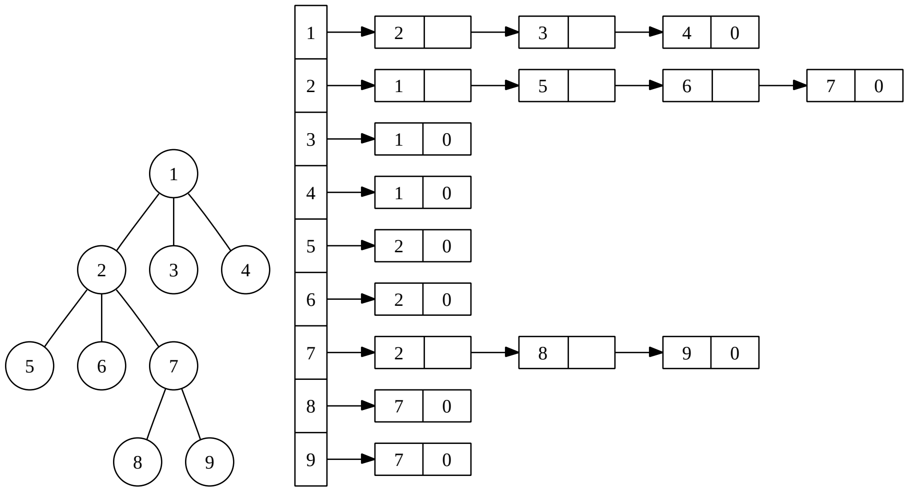


Fig. 15. Euler Tour example.

$(2,6) \rightarrow (6,2) \rightarrow (2,7) \rightarrow (7,8) \rightarrow (8,7) \rightarrow (7,9) \rightarrow (9,7) \rightarrow (7,2) \rightarrow (2,1) \rightarrow (1,3) \rightarrow (3,1) \rightarrow (1,4) \rightarrow (4,1) \rightarrow (1,2) \rightarrow (2,5) \rightarrow (5,2)$

b. ET Technique

The Euler Tour technique can be used to compute some tree functions [30] such as rooting a tree, post order numbering, computing the vertex level and computing the number of descendants.

These algorithms are similar in terms of the procedures performed. And Figure 16 shows a generic algorithm which first initializes each edge in the list with a corresponding weight, and then performs the prefix sum algorithm. The desired result can be computed with the prefix sum value of each edge.

When P is equal or larger than N where P denotes the number of processors and N is the number of vertices, the line 1-2 and 4-5 in Figure 16 take constant time and $O(N)$ work, and the line 3 takes $O(\log N)$ time and $O(N \log N)$ work. Thus

```
Input:
(1) A tree T stored in a list L
(2) An Euler Tour defined by the successor function s

Output:
For each vertex, the parent, postorder number, level or the number of descendants.

Algorithm:
begin
1: for each edge (x,y) in L pardo
2:  weight(x,y) = m               //initialized to different weight accordingly
3: parallel prefix sum            //perfrom prefix sum
4: for each edge (x,y) in L pardo
5:  computation(x,y)              //perform the corresponding computation
end
```

Fig. 16. Generic algorithm of Euler Tour applications.

the complexity of the generic algorithm is $O(\log N)$ time and $O(N \log N)$ work by using the EREW PRAM model [30]. When P is less than N and each location has $\frac{N}{P}$ elements. The line 1-2 and 4-5 take constant time and $O(N)$ work, and the line 3 takes $O(\frac{N}{P} + \log P)$ time and $O(N + P \log P)$ work. Thus the complexity of the generic algorithm is $O(\frac{N}{P} + \log P)$ time and $O(N + P \log P)$ work.

2. ET in STAPL

The parallel Euler Tour constructor algorithm [30] implemented in STAPL uses a STAPL `pGraph` to represent the tree and a `pList` to store the final Euler Tour. In the first phase, the algorithm executes traversals on the `pGraph` view and generates Euler Tour segments that are stored in a temporary `pList`. Then, the segments are linked together to form the final `pList` containing the Euler Tour in the second phase.

Figure 17 describes the first phase of the algorithm. We use an undirected `pGraph` to represent the tree and a `pList` to store the final Euler Tour. Each local traversal is stored as a list of edges (L) and then inserted as an element to a `pList` of list of edges (PLE). The next edge for a given edge can be found by using the successor

```
Input:
(1) Undirected pGraph PG which is a tree
(2) start edge Start_Edge of the Euler Tour
Output:
pList with edges stored in Euler Tour order

Algorithm:
list<edge> L stores a local traversal for each cross cutting edge
pList<list<edge> > PLE stores all L

begin

for each incoming cutting edge e(i) pardo
  L(i).insert(e(i))                        //store cross cutting edge
  next_edge(i) = e(i);
  do {
    next_edge(i) = succ_func(next_edge(i)) //find next edge
    L(i).insert(next_edge(i))              //insert to L
  } while (nex_edge(i) is not cutting edge)//until reach another cross cutting edge
  PLE.push_anywhere_async(L(i))            //store L to the PLE

//special pList constructor takes PLE, PG and
//Start_Edge, and return the final pList with Euler Tour
pList<edge> pL(PLE, PG, Start_Edge)

end

next_edge(i) succ_func(edge e(i))   //successor function
{
  vertex source(i) = e(i).source()  //source vertex of edge e(i)
  vertex target(i) = e(i).target()  //target vertex of edge e(i)
  list<vertex> adjl(i) = target’s adjacency list
  find source(i) in adjl(i)
  if source(i) is the end of adjl(i)//if source(i) is the last vertex, v(i) is the first vertex
    v(i) = adjl(i).begin()          //in the adjacency list
  else
    v(i) = adjl(i).next(source(i))  //v(i) is the next vertex of source(i) in the adjacency list
  return edge(target(i), v(i))      //return the next edge of input edge
}
```

Fig. 17. Euler Tour [30] construction.

function introduced in [30].

The second phase of the algorithm in Figure 18 is performed using a special `pList` constructor which takes the `pList` of list of edges, `pGraph` and start edge as the arguments. `pGraph` is used to find where a given remote edge lives. An internal phashmap is used to support fast look up of the base container for a given edge. The start edge is used to identify the global first and last `bContainer` of the final `pList`. The base container which starts with the start edge is the global first base container

```
pList(pList<list<edge> > PLE, pGraph PG, edge Start_edge)
{
  //internal phashmap built for fast loopup from given edge to base container
  phashmap<edge, base container> pm(i)
  for each element l(i) of PLE
    create base container bc(i) over l(i)
    insert(l(i).front(), bc(i)) to pm(i) //add cross cutting edge and its base cotainer to phashmap
    if (bc(i).front() == start_edge)
      broadcast to set bc(i) to the global first base container //broadcast head base cotainer
    if (bc(i).back() == start_edge)
      broadcast to set bc(i) to the global last base container //broadcast tail base cotainer
  for each base container bc(i)
    edge next(i) = bc(i).back()
    location loc = PG.find(next(i)) //ask pgraph to find location information
    send bc(i) to location
    find base container bc_next which begins with next at location loc using phashmap
    set bc(i) to be bc_next's previous base container //set up link
    send bc_next back to where bc(i) lives
    set bc_next to be bc(i)'s next base container     //set up link
    remove last element in bc(i)                      //remove extra edgs
}
```

Fig. 18. Euler Tour [30] linking.

and the one that ends with the start edge is the global last one. Communications happen explicitly to link two base containers, one from current base container to next base container to set next base container's previous to be current base container, and another one from next base container to current base container to set current base container's next to be next base container. These are accomplished by asking `pGraph` to decide the location of remote edge and phashmap to find the base container that starts with remote edge on that location. The last edge in each base container is used to specify the next relationship, and it can be removed once two base containers are connected. After all the links are constructed, the algorithm has constructed a `pList` of edges with the Euler Tour stored in order.

a. Complexity Analysis

We use $O(E_i)$ and $O(V_i)$ represent separately the number of edges and vertices in location i. $O(C_i)$ is the number of cross cutting edges in location i. $T_B(N)$ is the time of a broadcast communication of N elements. T_ϵ is the time of a single message communication time. P is the total number of locations.

- find next edge for each edge:
  - find next edge: $\frac{E_i}{V_i}$ (Average size of adjacency list)
  - Total phase 1 time: $O(E_i) * O(\frac{E_i}{V_i})$
- Build `pList`, for each base container, set previous and next
  - Number of base containers : $O(C_i)$
  - Communication time to set all links: $O(C_i) * O(T_\epsilon)$
  - Broad cast first and last base container of `pList`: $T_B(P)$
  - Total phase 2 time: $O(C_i) * (1 + O(T_\epsilon)) + T_B(P)$

Total Euler Tour time: $O(\frac{(E_i)^2}{V_i}) + O(C_i) * (1 + O(T_\epsilon)) + T_B(P)$

3. ET Application in STAPL

By using the Euler Tour technique, we implemented four applications in STAPL, including rooting a tree, postorder numbering, compute the vertex level and the number of descendants. These four algorithms are implemented using the generic algorithm described in Figure 16. We use the STAPL `pGraph` to represent the tree and the `pList` to store the Euler Tour. The algorithms differ in the initialization weight of each edge and the computation performed.

a. Rooting a Tree

This algorithm (Figure 19) is frequently used as initial step in other algorithms. A STAPL undirected `pGraph` is used to represent the tree, and Euler Tour is assumed to

```
Input:
(1) A tree T defined as STAPL undirected pgraph
(2) An Euler Tour defined by the successor function s stored in pList pl
(3) Root vertex r

Output:
For each vertex v != r, the parent p(v) of v in the tree rooted at r.

Algorithm:

begin

for each edge in pl pardo
  weight = 1

parallel prefix sum

for each edge (x,y) pardo
  if (prefix(x,y) < prefix(y,x))
    set x=parent(y)

end
```

Fig. 19. Rooting a tree [30].

be computed already and stored inside the `pList`. The algorithm starts by assigning a weight 1 to each edge in `pList`, and the prefix sum algorithm is run over `pList`. For each edge from vertex x to vertex y, if the prefix sum of xy is smaller than prefix sum of yx, which means xy is traversed first and then yx in the Euler Tour, then x is the parent of y. The root has no parent.

b. Postorder Numbering

This algorithm (Figure 20) requires the parent information of each vertex, so the rooting tree algorithm is used. For each edge in the `pList`, if it is an edge from a vertex to this vertex's parent, the weight is set to be 1, and otherwise is set to 0. The prefix sum algorithm runs over the `pList`. Since only the edge from a vertex to its parent has any weight, the prefix sum of the edge from a vertex to its parent after the prefix sum algorithm is the postorder number. The postorder number of the root is the number of vertices in the tree.

```
Input:
(1) A tree T defined as STAPL undirected pgraph
(2) An Euler Tour defined by the successor function s stored in pList pl
(3) Root vertex r

Output:
For each vertex v, the post order traversal number post(v) of v in the tree rooted at r.

Algorithm:

begin

rooting the tree

for each vertex v != r pardo
  weight(v, p(v)) = 1
  weight(p(v), v) = 0

parallel prefix sum

for each vertex v pardo
  if (v == r)
    set post(v) = n(the num of vertices of tree)
  else
    set post(v) = prefix(v, p(v))

end
```

Fig. 20. Postorder numbering [30].

c. Computing Vertex Level

This algorithm (Figure 21) also requires the parent information of each vertex, so the rooting tree algorithm is used. For each edge in the `pList`, if it is an edge from a vertex to this vertex's parent, the weight is set to be -1, otherwise 1. The prefix sum algorithm runs over the `pList`. Whenever an edge is from parent to a child, the vertex level increases by 1, and decreases by 1 otherwise. Thus after the prefix sum algorithm, the prefix sum of a parent to a child records the level information of each vertex. The level of the root is 0.

```
Input:
(1) A tree T defined as STAPL undirected pgraph
(2) An Euler Tour defined by the successor function s stored in pList pl
(3) Root vertex r

Output:
For each vertex v, the level level(v) of v in the tree rooted at r.

Algorithm:

begin

rooting the tree

for each vertex v != r pardo
  weight(v, p(v)) = -1
  weight(p(v), v) =  1

parallel prefix sum

for each vertex v pardo
  if (v == r)
    set level(v) = 0
  else
    set level(v) = prefix(p(v), v)

end
```

Fig. 21. Computing vertex level [30].

d. Computing Number of Descendants

This algorithm (Figure 22) also requires the parent information of each vertex, so the rooting tree algorithm is used. For each edge in the `pList`, if it is an edge from a vertex to this vertex's parent, the weight is set to be 1, and otherwise is set to 0. The prefix sum algorithm runs over the `pList`. Whenever an edge is from a child to its parent, the number of descendants increases by 1, and an edge from a parent to a child has no influence on final result. After the prefix sum, the difference of the prefix sum between children to parent and parent to children equals to the number of descendants at vertex itself. The number of descendants of the root is the number of vertices in the tree minus 1.

```
Input:
(1) A tree T defined as STAPL undirected pgraph
(2) An Euler Tour defined by the successor function s stored in pList pl
(3) Root vertex r

Output:
For each vertex v, the number of descendants size(v) of v in the tree rooted at r.

Algorithm:

begin

rooting the tree

for each vertex v != r pardo
  weight(v, p(v)) = 1
  weight(p(v), v) = 0

parallel prefix sum

for each vertex v pardo
  if (v == r)
    set size(v) = n-1(n is the num of vertices of tree)
  else
    set size(v) = prefix(v, p(v)) - prefix(p(v), v)

end
```

Fig. 22. Computing number of descendants [30].

e. Complexity Analysis

These applications are implemented using the generic algorithm under the STAPL programming model. We add an extra phase to store the prefix sum value back to the graph so that the value of an edge (u,v) can be accessed in constant time by the edge (v,u) and vice versa. We use the same notation as in the complexity analysis section of Euler Tour in STAPL. B refers to the total number of base containers in `pList`.

- Initiate weight : $O(E_i)$
- Prefix sum: $O(\log B) + O(E_i)$
- Copy back to `pGraph`: $O(E_i) + O(C_i) * O(T_\epsilon)$
- Computation: $O(E_i) + O(C_i) * O(T_\epsilon)$

Total Euler Tour Application Time: $O(\log B) + O(E_i) + O(C_i) * O(T_\epsilon)$.

CHAPTER VI

PERFORMANCE EVALUATION

Previous chapters show the `pList` interface, its design, and implementation and describe parallel algorithms using the `pList`. This chapter examines performance of `pList` methods, generic algorithms (`p_generate`, `p_foreach` and `p_accumulate`) using `pList`, and specialized algorithms such as list ranking (LR-glob-sync, LR-pt2pt-sync and LR-STAPL), and Euler Tour and four of its applications (rooting a tree, postorder numbering, computing vertex level and number of descendants).

A. Machine Specification

We conducted our experimental studies on two architectures: an IBM cluster with p575 SMP nodes available at Texas A&M University (P5-CLUSTER), and a Cray XT4 with quad core Opteron processors available at NERSC (CRAY). Table IV shows the details of the configuration of each machine. In all experiments, a location contains a single processor, and the terms can be used interchangeably.

B. `pList` Constructor and Memory Usage

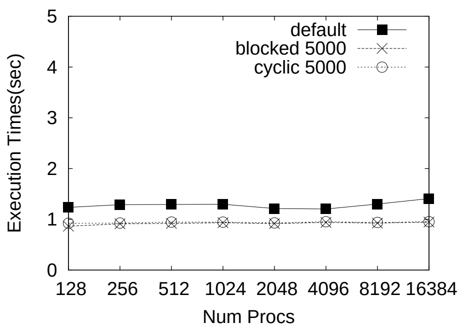
In this section we discuss the performance of the different `pList` constructors. The default `pList` constructor will be built with one base container per location and data will be evenly distributed across all processors (locations). Constructors are also provided to build the `pList` with multiple base containers per location and these base containers can be linked together in a blocked or cyclic manner.

Figure 23 shows three different constructors with different numbers of base containers per location. The more base containers are specified, the more work it takes

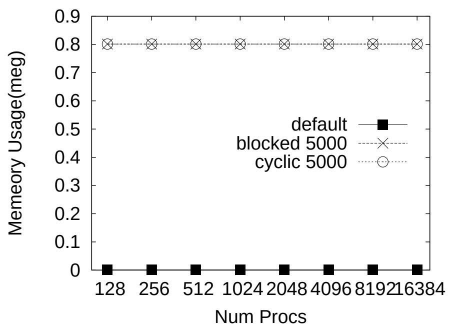
Table IV. Machine specifications about P5-CLUSTER and CRAY.

configuration	CRAY	P5-CLUSTER
Number of compute nodes	9,572	52
Processor cores per node	4	16
Number of compute processor cores	38,288	832
Processor Core type	Opteron 2.3 GHz Quad Core	1.9GHz Power5+ processor
System theoretical peak (compute nodes only)	352 TFlop/sec	6.3 TFlop/sec
Physical memory per compute node	8 GB	32 GB
Memory usable by applications per node	7.38 GB	25 GB
Switch Interconnect	SeaStar2	2-plane HPS (IBM's High Performance Switch)
Operating System	SuSE SLES10 SP1 Linux	AIX 5.3

to link all the base containers compared to the default constructor. And for the same number of base containers specified per location, the cyclic distributed constructor takes more communication time to link all of the base containers compared to a blocked distributed constructor. The reason that the cyclic distributed constructor and the blocked distributed constructor are similar is because the only difference is the linkage structure of the blocks. The default constructor is slower because the



(a) CRAY: Constructor



(b) CRAY: Metadata

Fig. 23. `pList` constructor. (a) Execution times of constructors using different number of base containers per location in a default, blocked and cyclic distributed fashion (weak scaling each processing unit has 20 millions). (b) Metadata size for the default, blocked and cyclic constructor used in (a).

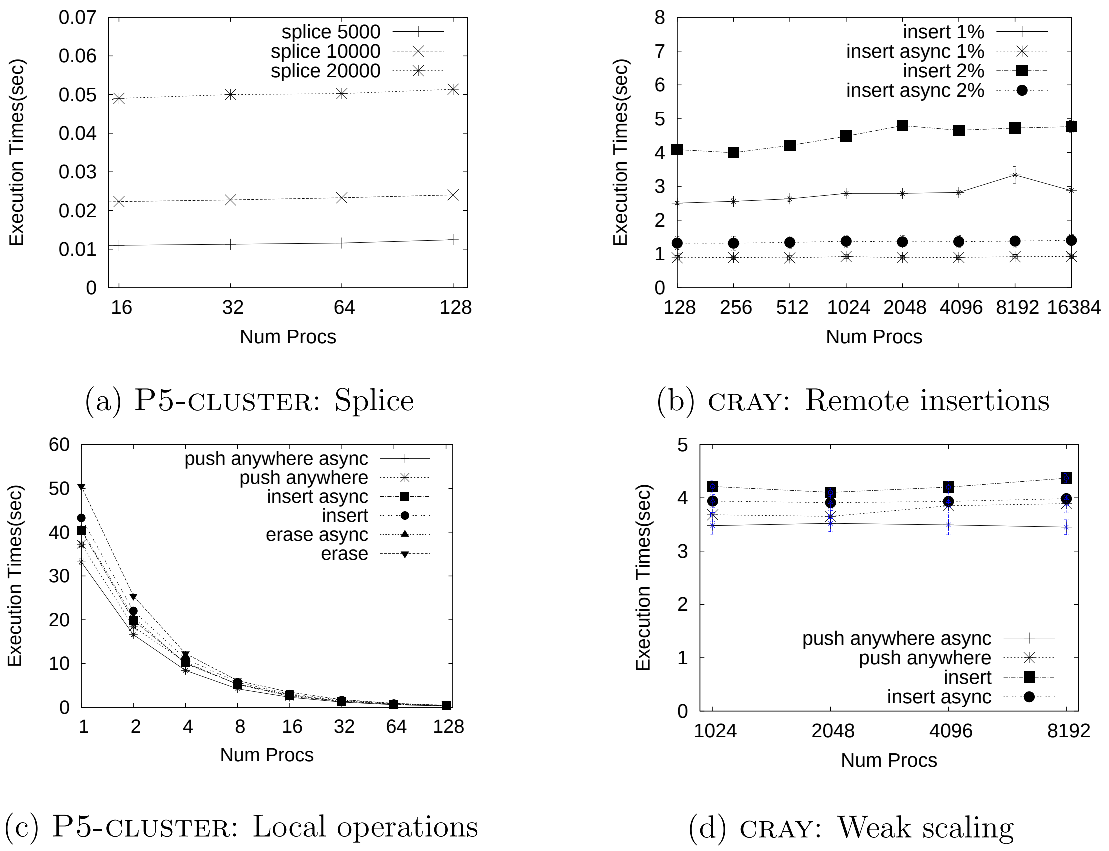
memory allocator takes more time to allocate a large chunk of memory than a number of small chunks of memory, which dominates the time to link all base containers. The meta data is proportional to the number of base containers in one location. Since each location contains 1 or 5000 base containers, thus the amount of meta data is the same across all locations for each scenario.

C. `pList` Methods

```
evaluate_performance(N,P)
   tm = stapl::start_timer();   //start timer
   //insert N/P elements concurrently
   for(i=0; i<N/P; ++i)
      plist.push_anywhere(v[i]); //pre-generated value stored in vector v
   rmi_fence(); //ensure all inserts are finished
   elapsed = stapl::stop_timer(tm); //stop the timer
   - Reduce elapsed times, getting the max time from all processors.
   - Report the max time
```

Fig. 24. Kernel used to evaluate the performance of `pList` methods.

In this section we discuss the performance of the `pList` methods and the factors



(a) P5-CLUSTER: Splice

(b) CRAY: Remote insertions

(c) P5-CLUSTER: Local operations

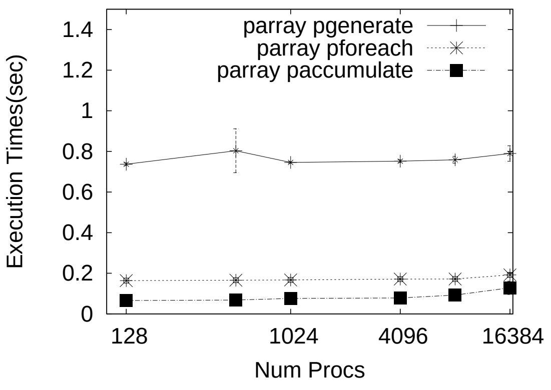
(d) CRAY: Weak scaling

Fig. 25. `pList` methods comparison. (a) `Splice` for 5000 to 20000 base containers per location. (b) Execution times for `insert` and `insert_async` when 1% and 2% of operations are remote. (c) Execution times when all operations are executed locally (N=64 millions). (d) Weak scaling for `pList` methods on large number of processors using 25 million elements per processor.

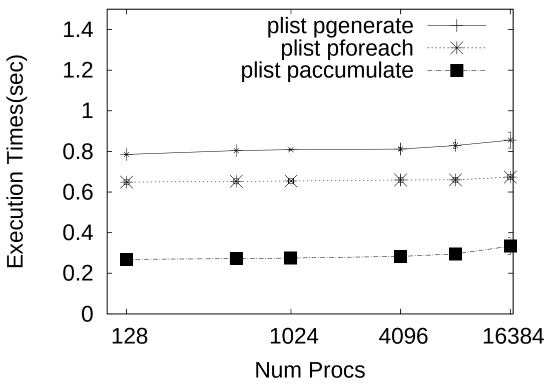
influencing their running time. To evaluate the scalability of individual methods we designed the kernel shown in Figure 24. The figure shows `push_anywhere`, but the same kernel is used to evaluate all methods. For a given number of elements N, all P available processors (locations) concurrently insert N/P elements. We report the time taken to insert all N elements globally. The measured time includes the cost of a fence call which, as stated in Section III, is more than a simple barrier. Figure 25 shows the execution time of different `pList` methods. To evaluate the `splice` method, we splice a `pList` with a fixed number of base containers per location into another `pList`. Figure 25(a) shows the execution time on P5-CLUSTER for the splice operation for different numbers of base containers per location and for various numbers of locations. As expected, the time increases linearly with the number of spliced base containers, but increases only slowly with the number of processors (almost constant), ensuring good scalability. In Figure 25(b) we show the execution time for a mix of local and remote method invocations to highlight the advantages of the asynchronous methods over the synchronous ones. When all methods are invoked with the same percentage of remote operations, the `insert` method that returns an iterator to the newly inserted element is on average slower than the `insert_async` method, which does not return a value and can exploit message aggregation. Also for the same method, the time increases as the percentage of remote operations increases. In another study, all methods are executed locally and we observe in Figure 25(c) that both synchronous and asynchronous methods exhibit scalable performance as they do not incur any communication. In Figure 25(d) we show a weak scaling experiment we performed on CRAY using 25 million elements per processor and up to 8192 processors (104.8 billion method invocations performed globally). The `push_anywhere` methods are faster than `insert` as they do not perform any additional searches to find the position to add the element. The asynchronous versions are faster as they don't return an iterator to the newly added element.

D. pAlgorithms Comparison

In this section we present the performance of three generic non-mutating `pAlgorithms`, `p_generate`, `p_foreach`, and `p_accumulate`, which store their data in two different STAPL `pContainers`: `pList` and `pArray`. Generic `pAlgorithms` can operate transparently on different data structures.



(a) `pArray`; 20M/Proc



(b) `pList`; 20M/Proc

Fig. 26. Execution times for `p_foreach`, `p_generate`, `p_accumulate` algorithms on CRAY for different data structures.

The `p_generate` algorithm produces random values and assigns them to the elements in the container, the `p_foreach` increments the elements in the container with a given constant. The `p_accumulate` accumulates all the elements in the container using a generic map reduce operation available in STAPL.

For all the algorithms considered in this section, for both P5-CLUSTER and CRAY (Figure A-1 in Appendix A shows results on P5-CLUSTER), we conducted weak scaling experiments. Strong scaling would be difficult to evaluate due to the short execution times of the algorithms even when run on very large input sizes.

In Figure 26, we show the execution times for the `pAlgorithms` on `pArray` and `pList`. In Figure 26(a) we show a weak scaling experiment for `pArray`, a simple but efficient static container for accessing data based on indices [10]. The experiment is run with 20M elements per processors and all algorithms show good scalability. As we scale from 128 to 8192 processors there is less than 5% performance degradation for `p_generate` and `p_foreach`.

The `pList` is a dynamic `pContainer` optimized for fast insert and delete oper-

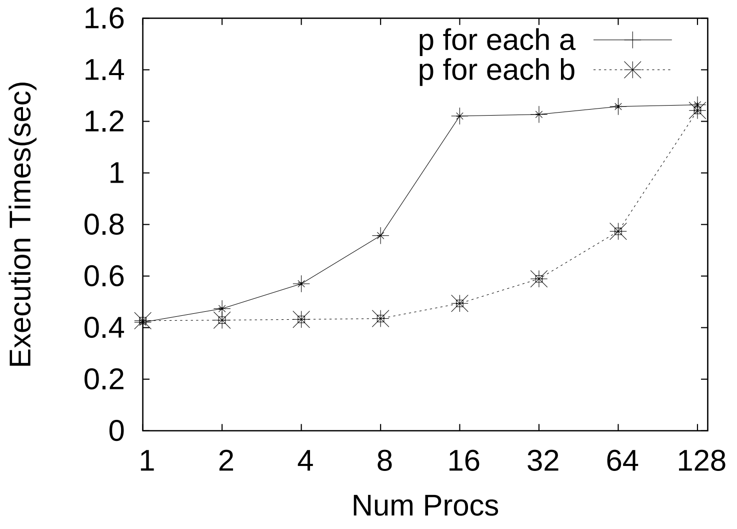


Fig. 27. P5-CLUSTER: weak scaling for `p_foreach` allocating processes on the same nodes when possible (curve a) or in different nodes (curve b). Experiments are for 20 million elements/processor.

ations at the cost of a slower access time relative to static data structures such as `pArray`. As seen in Figure 26(b), all three algorithms on a `pList` with 20M elements per location provide good scaling with less than 10% performance degradation as we scale from 128 to 8192 processors.

Figure 27 shows two weak scaling experiments on P5-CLUSTER for two different processor allocation strategies. Each node of P5-CLUSTER has 16 processors. In the figure, `p_foreach`-a represents the case where all processors are allocated on a single node (possible for 1-16 processors). `p_foreach`-b represents the case where we use cyclic allocation across 128 processors, e.g., 16 processors would be allocated one per node, and in general, there will be $P/8$ processors allocated on each node for $P < 128$. The reason why the two curves do not match is related to memory bandwidth saturation within a node. In `p_foreach`-b, the nodes are fully utilized only when running on 128 processors, while for `p_foreach`-a we use all processors in a node when running on 16 processors or more. These experiments emphasize the importance of a good task placement policy on the physical processors.

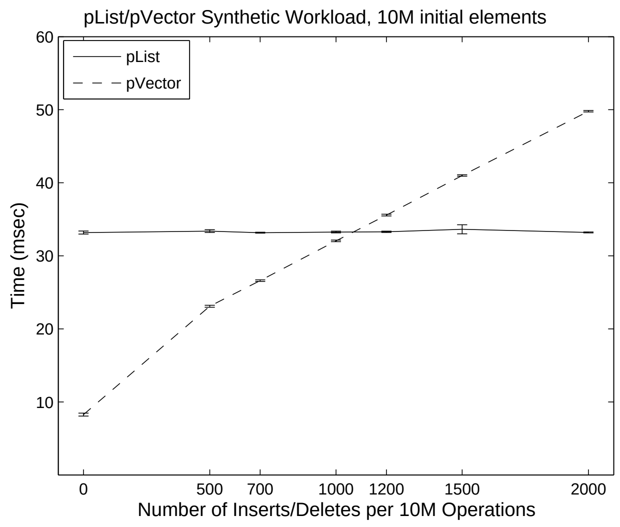


Fig. 28. Comparison `pList` and `pVector` dynamic data structures using a mix of 10M operations (read/write/insert/delete).

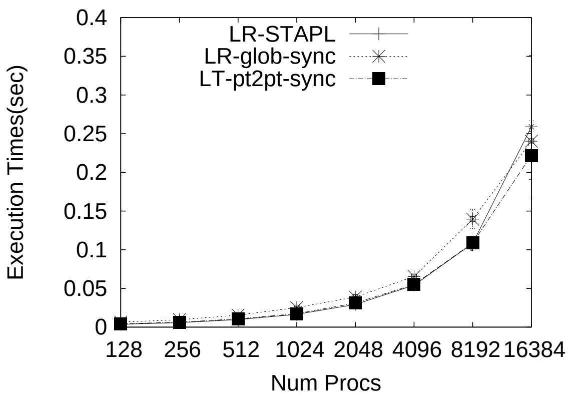
E. Comparison of Dynamic Data Structures in STAPL

In this section, we compare the performance of the `pList` and `pVector` for various mixes of container operations (i.e., read(), write(), insert() and delete()). We show that the proportion of operations that modify the container size has substantial effects on runtime, demonstrating the utility of each and that care must be taken in selecting the appropriate parallel data structure.

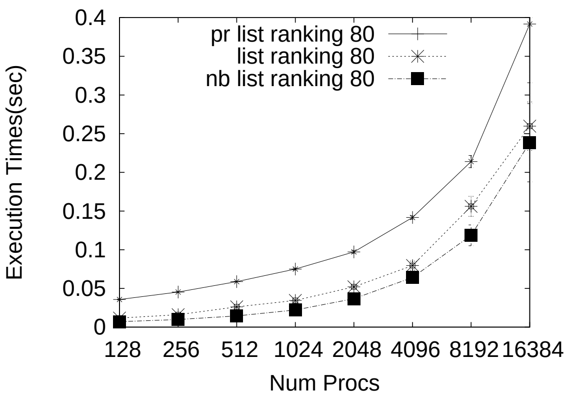
In Figure 28, we show results for both containers on the P5-CLUSTER for 16 processors and 10 million elements. We perform 10 million operations per container. Each operation is either a read or write of the next element in the container, an insertion at the current location, or deletion of the current element. These operations are distributed evenly among the processors, which perform them in parallel. For these experiments, the combined number of insertions and deletions is varied from 0 to 2000, with the remaining operations being an equal number of reads and writes. More insertions or deletions than this cause the runtime of the `pVector` to increase dramatically.

As expected, the runtime of the `pList` remains relatively unchanged regardless of the number insertions or deletions, as both operations execute in constant time. The performance of the `pVector` bests the `pList` when there are no insertions or deletions. However, at 1200 insertions/deletions, the heavy cost of the operations (all subsequent elements must be shifted accordingly) causes the performance of the two containers to crossover with the `pList` taking the lead. This experiment clearly justifies the use of the `pList` in spite of not being truly random access containers like the `pVector` .

F. List Ranking



(a) 1 `bContainer`per location



(b) 80 `bContainer`s per location

Fig. 29. List Ranking performance comparison (LR-glob-sync refers to Pointer Jumping algorithm. LR-pt2pt-sync refers to point-to-point synchronization Pointer Jumping algorithm. LR-STAPL refers to STAPL implementation of point-to-point synchronization Pointer Jumping algorithm. The number means how many `bContainer`s per location).

Figure 29 shows the performance of different versions of the parallel list ranking algorithm performed on different numbers of processors by using different numbers of base containers per location. The larger the number of base containers per location the more computation needs to be performed requiring longer computation time (section A in chapter V). The LR-glob-sync is slower because of the global synchronization performed at each step. The LR-STAPL is getting slower than the LR-pt2pt-sync with the number of base containers increases because LR-STAPL needs to create the work function and the view dynamically first and then perform the algorithm compared to LR-pt2pt-sync which uses lower level MPI primitives directly. The LR-STAPL are implemented using high level representations under the STAPL programming model which is more general and natural to express the algorithm.

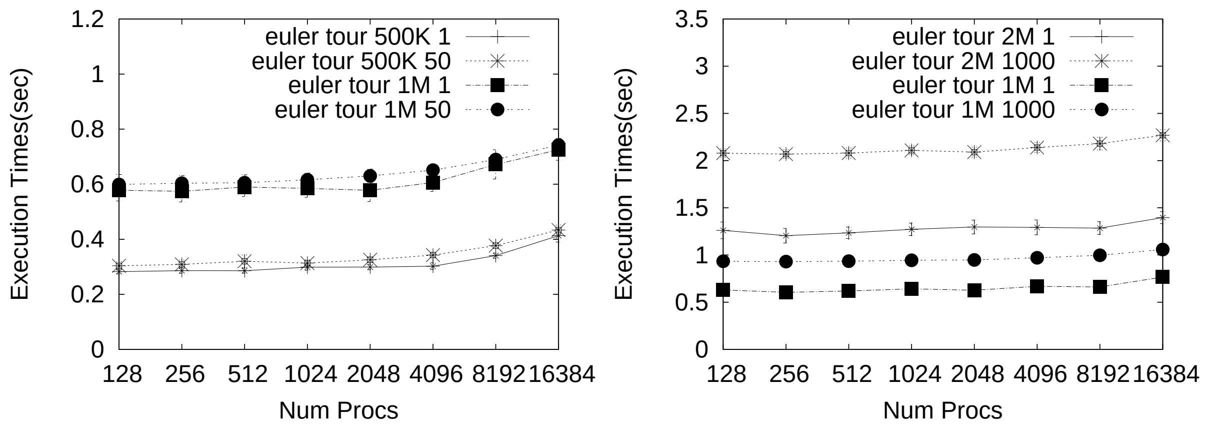
G. Euler Tour and Its Application

The Euler Tour (ET) is an important representation of a graph for parallel processing. Since the ET represents a depth-first-search traversal, when it is applied to a tree it can be used to compute a number of tree functions such as rooting a tree, postorder numbering, vertex levels, and number of descendants [30].

1. Euler Tour

The parallel ET constructor algorithm [30] tested here uses a `pGraph` to represent the tree and a `pList` to store the final Euler Tour. In parallel, the algorithm executes traversals on the `pGraph` view and generates ET segments that are stored in a temporary `pList`. Then, the segments are linked together to form the final `pList` containing the ET.

Performance is evaluated by a weak scaling experiment on CRAY using as input



(a) binary tree of binary trees tree

(b) list tree

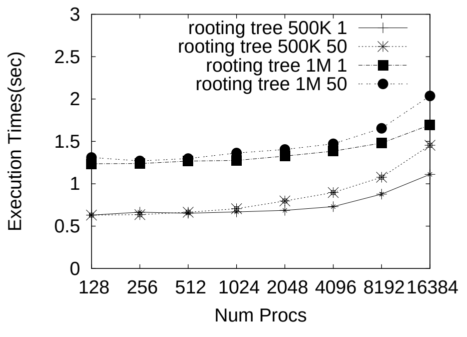
Fig. 30. Euler Tour with two different types of trees.

a tree distributed across all locations. We consider two types of trees. One is a binary tree of binary tree that is generated by first building a specified number of binary trees in each location and then linking the roots of these trees in a binary tree fashion. The number of remote edges is at most six times the number of subtrees for each location (for each subtree root, one to its root and two to its children in each location, with directed edges in both directions). The second tree was simply a list. Figure 30(a) shows the execution time on CRAY for different sizes of the tree and varying numbers of subtrees. The running time increases with the number of vertices per location because the number of edges in the computed ET increases correspondingly. When there are more subtrees specified in each location, there is more communication required to link them. The same analysis applies to lists graph too (Figure 30(b)) which simply links all the vertices to form a list.

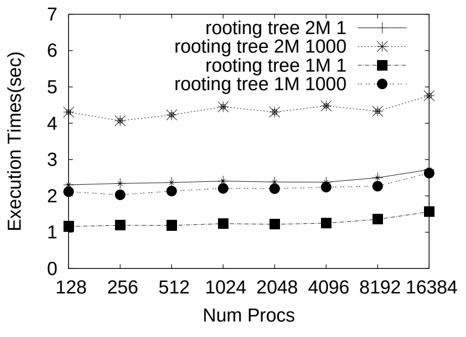
2. Euler Tour Application

The tree ET applications are computed using a generic algorithm which first initializes each edge in the tour with a corresponding weight, and then performs the prefix sum algorithm. The prefix sum result for each edge is copied back to the graph, and the final step computes the desired result. For more details please refer to Chapter V.

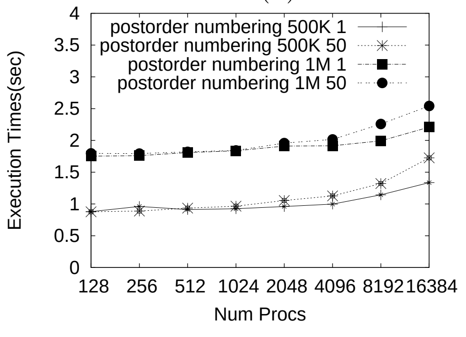
Figure 31(a,b,c,d) shows the execution time for rooting the tree, computing the posorder numbering, vertex level and number of descendants using tree graph which is generated the same way as in ET section. The running time increases with the number of vertices per location because the number of edges increases and the computation cost is proportional to the number of edges. When more subtrees are specified per location, more segments are formed in the `pList` and more communication is needed for the prefix sum. Rooting the tree is faster than the other three applications because the initialization phase simply assigns each edge a weight of 1 compared to others which have to check for the parent information and then assign the corresponding weight. The same analysis applies to the list graph (Figure 31(e,f,g,h)).



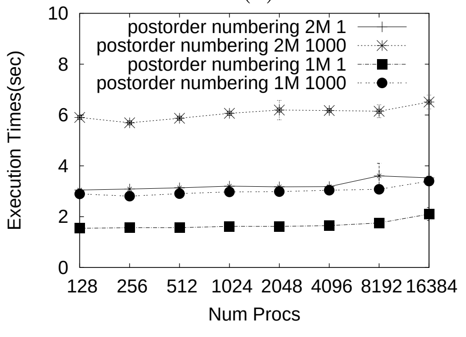
(a)



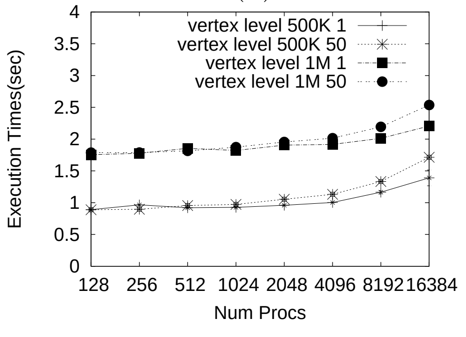
(e)



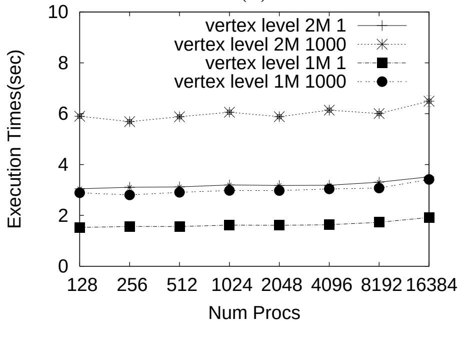
(b)



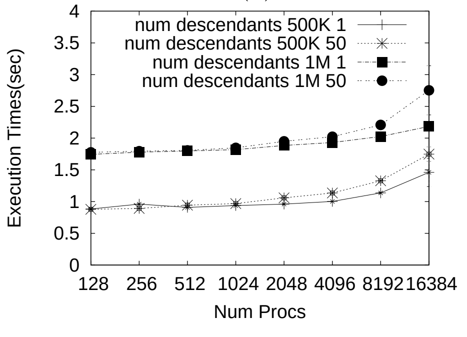
(f)



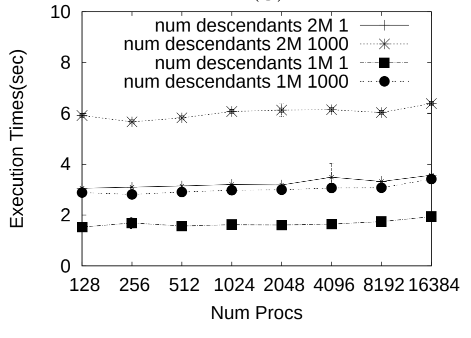
(c)



(g)



(d)



(h)

Fig. 31. Euler Tour application.

CHAPTER VII

CONCLUSION

In this thesis, we presented the STAPL `pList`, a distributed data structure optimized for fast dynamic operations such as `push_anywhere`, `push_back`, and `erase`. We described the design and implementation of the `pList`, whose methods include counterparts of the STL list container methods, and new methods that provide improved parallel performance. Our experimental results on a variety of architectures show that `pList` provides good scalability and compares favorably with other STAPL dynamic `pContainers`.

We also described the design and implementation of various parallel algorithms using the `pList` such as list ranking, Euler Tour and some of its applications (rooting a tree, postorder numbering, computing vertex level and computing number of descendants).

We have demonstrated that `pList` is a generic and dynamic parallel container which is appropriate to work with for certain parallel algorithms.

In the future, we would like to use and evaluate the `pList` in real world applications. According to the needs of those applications, more methods can be designed and implemented. Also if there is any other generic and more efficient list than the STL list available in the future, then we can use it instead of our current base container for improved performance.

REFERENCES

[1] A. Buss, A. Fidel, Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The STAPL pView," Dept. Comp. Sci., Texas A&M Univ., Tech. Rep., TR10-001, July 2010.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger, "STAPL: An adaptive, generic parallel programming library for C++," in *In. Workshop on Languages and Compilers for Parallel Computing (LCPC), published in Lecture Notes in Computer Science (LNCS)*, Cumberland Falls, KY, Aug 2001, vol. 2624, pp. 193–208.

[3] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: Standard Template Adaptive Parallel Library," in *Proc. Annual Haifa Experimental Systems Conference (SYSTOR)*, New York, NY, 2010, pp. 1–10.

[4] A. Buss, T. Smith, G. Tanase, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger, "Design for interoperability in STAPL: pMatrices and linear algebra algorithms," in *In. Workshop on Languages and Compilers for Parallel Computing (LCPC), published in Lecture Notes in Computer Science (LNCS)*, Edmonton, Alberta, Canada, July 2008, vol. 5335, pp. 304–315.

[5] L. Rauchwerger, F. Arzu, and K. Ouchi, "Standard Templates Adaptive Parallel Library," in *Proc. of the 4th In. Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburgh, PA, May 1998, pp. 402–409.

[6] L. Rauchwerger, F. Arzu, and K. Ouchi, "Standard templates adaptive parallel library (STAPL)," *Lecture Notes in Computer Science*, vol. 1511, pp. 402–413, 1998.

[7] S. Saunders and L. Rauchwerger, “Armi: an adaptive, platform independent communication library,” in *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, San Diego, CA, 2003, pp. 230–241.

[8] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger, “Associative parallel containers in STAPL,” in *In. Workshop on Languages and Compilers for Parallel Computing (LCPC), published in Lecture Notes in Computer Science (LNCS)*, Urbana-Champaign, IL, 2008, vol. 5234, pp. 156–171.

[9] G. Tanase, X. Xu, A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger, “The STAPL pList,” in *In. Workshop on Languages and Compilers for Parallel Computing (LCPC), published in Lecture Notes in Computer Science (LNCS)*, Newark, DE, 2009, vol. 5898, pp. 16–30.

[10] G. Tanase, M. Bianco, N. M. Amato, and L. Rauchwerger, “The STAPL pArray,” in *Proc. of the 2007 Workshop on Memory Performance (MEDEA)*, Brasov, Romania, 2007, pp. 73–80.

[11] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, “A framework for adaptive algorithm selection in STAPL,” in *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, Chicago, IL, 2005, pp. 277–288.

[12] J. D. Valois, “Lock-free linked lists using compare-and-swap,” in *Proc. ACM Symp. on Princ. of Dist. Proc. (PODC)*, New York, NY, 1995, pp. 214–222.

[13] M. M. Michael and M. L. Scott, “Correction of a memory management method for lock-free data structures,” Dept. Comp. Sci., Univ. of Rochester, Rochester, NY, Tech. Rep., TR599, 1995.

[14] T. L. Harris, “A pragmatic implementation of non-blocking linked-lists,” in *Proc. Int. Conf. Dist. Comput.*, London, UK, 2001, pp. 300–314.

[15] M. M. Michael, “High performance dynamic lock-free hash tables and list-based sets,” in *Proc. of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Winnipeg, Manitoba, Canada, 2002, pp. 73–82.

[16] M. Fomitchev and E. Ruppert, “Lock-free linked lists and skip lists,” in *Proc. ACM Symp. on Princ. of Dist. Proc. (PODC)*, New York, NY, 2004, pp. 50–59.

[17] W. Pugh, “Concurrent maintenance of skip lists,” Dept. Comp. Sci., Univ. of Maryland, College Park, MD, Tech. Rep., CS-TR-2222, 1990.

[18] M. T. Goodrich, R. Tamassia, and D. M. Mount, *Data Structures and Algorithms in C++*, 2nd ed. New York: Wiley, 2009.

[19] J. McCarthy, “Recursive functions of symbolic expressions and their computation by machine, part i,” *Commun. ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[20] A. Newell and J. C. Shaw, “Programming the logic theory machine,” in *IRE-AIEE-ACM ’57 (Western): Papers presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, New York, NY, 1957, pp. 230–240.

[21] M. Herlihy, “A methodology for implementing highly concurrent data objects,” *ACM Trans. Prog. Lang. Sys.*, vol. 15, no. 5, pp. 745–770, 1993.

[22] A. Paul and J. Rohrig, “Implementing a parallel list on the sb-pram,” in *Proc. Int. Conf. on High-Performance Comput. (HIPC)*, Washington, DC, 1998, p. 52.

[23] R. J. Anderson and G. L. Miller, “Deterministic parallel list ranking,” in *Proc. the 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures*, London, UK, 1988, pp. 81–90.

[24] D. R. Helman and J. JáJá, "Designing practical efficient algorithms for symmetric multiprocessors," in *ALENEX '99: Selected papers from the In. Workshop on Algorithm Engineering and Experimentation*, London, UK, 1999, pp. 37–56.

[25] M. Reid-Miller, "List ranking and list scan on the cray c90," *J. Comput. Syst. Sci.*, vol. 53, no. 3, pp. 344–356, 1996.

[26] U. Vishkin, "Randomized speed-ups in parallel computation," in *STOC '84: Proc. of the Sixteenth Annual ACM Symposium on Theory of Computing*, New York, NY, 1984, pp. 230–239.

[27] J. C. Wyllie, "The complexity of parallel computations," Dept. Comp. Sci., Cornell Univ., Ithaca, NY, Tech. Rep., 1979.

[28] R. Cole and U. Vishkin, "Approximate parallel scheduling, part I: the basic technique with applications to optimal parallel list ranking in logarithmic time," *SIAM J. Comput.*, vol. 17, no. 1, pp. 128–142, 1988.

[29] R. Cole and U. Vishkin, "Faster optimal parallel prefix sums and list ranking," *Inf. Comput.*, vol. 81, no. 3, pp. 334–352, 1989.

[30] J. JàJà, *An Introduction Parallel Algorithms*. Reading, MA: Addison–Wesley, 1992.

[31] R.E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM J. on Computing*, pp. 862–874, 1985.

[32] G. Cong and D. A. Bader, "The euler tour technique and parallel rooted spanning tree," in *In Proc. In. Con. on Parallel Processing (ICPP)*, 2004, pp. 448–457.

[33] D. Musser, G. Derge, and A. Saini, *STL Tutorial and Reference Guide*, 2nd ed. Boston, MA: Addison-Wesley, 2001.

[34] N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger, “ARMI: A high level communication library for STAPL,” *Parallel Processing Letters*, vol. 16(2), pp. 261–280, 2006.

[35] G. E. Blelloch, *Vector Models for Data-Parallel Computing*, Cambridge, MA: MIT Press, 1990.

[36] G. Blelloch, “NESL: A Nested Data-Parallel Language,” Dept. Comp. Sci., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep., CMU-CS-95-170, 1993.

[37] A. Chan and F. Dehne, “Cgmgraph/cgmlib: Implementing and testing cgm graph algorithms on pc clusters,” in *In. J. of High Performance Computing Applications*, 2003, p. 2005.

[38] D. Gregor and A. Lumsdaine, “Lifting sequential graph algorithms for distributed-memory parallel computation,” in *Proc. of the 20th annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, New York, NY, 2005, pp. 423–437.

[39] L. V. Kale and S. Krishnan, “CHARM++: A portable concurrent object oriented system based on C++,” *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, 1993.

[40] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn, “POOMA: A Framework for Scientific Simulations of Parallel Architectures,” in *Parallel Programming in C++*, Gregory V. Wilson and Paul Lu, Eds., Cambridge, MA: MIT Press, 1996, pp. 547–588.

[41] E. Johnson and D. Gannon, “HPC++: experiments with the parallel standard template library,” in *Proc. of the 11th In. Conf. on Supercomputing (ICS)*, Vienna, Austria, 1997, pp. 124–131.

[42] Intel, *Reference for Intel Threading Building Blocks, version 1.0*, April 2006.

[43] D. Callahan, Chamberlain, B.L., and H.P. Zima, "The Cascade high productivity language," in *The Ninth In. Workshop on High-Level Parallel Programming Models and Supportive Environments*, Los Alamitos, CA, 2004, vol. 26, pp. 52–60.

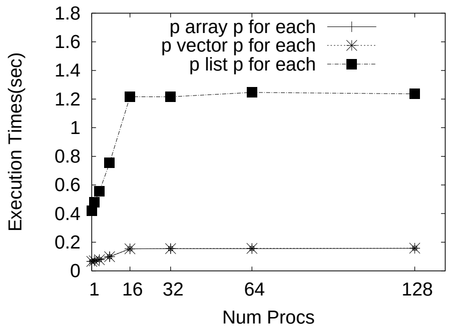
[44] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proc. of the 20th annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, New York, NY, 2005, pp. 519–538.

[45] G. Tanase, "The STAPL parallel container framework," Ph.D. dissertation, Dept. of Computer Science and Engineering, Texas A&M Univ., College Station, TX, 2010, to appear.

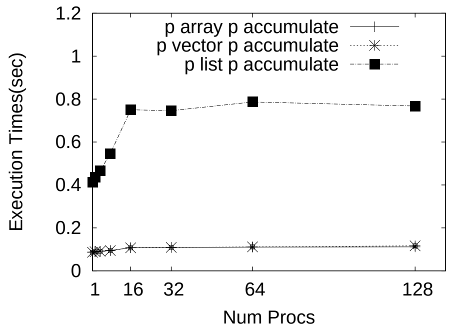
[46] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. Cambridge, MA: MIT Press, 2002.

APPENDIX A

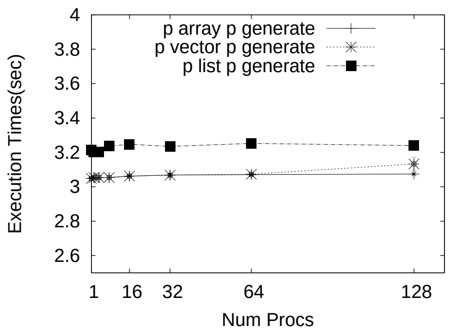
PERFORMANCE RESULTS

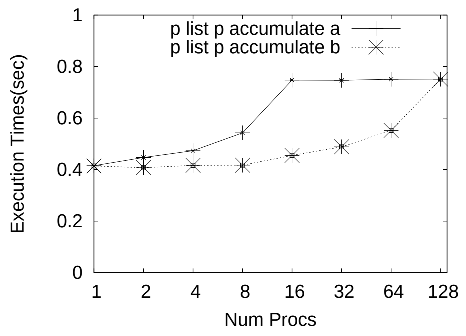


(a) P5-CLUSTER: For each algorithm



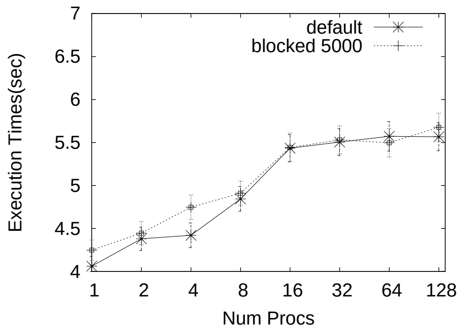
(b) P5-CLUSTER: Accumulate algorithm



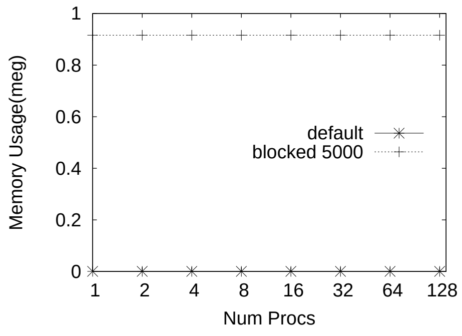


(c) P5-CLUSTER: Generate algorithm(d) P5-CLUSTER: Accumulate with 2 configurations

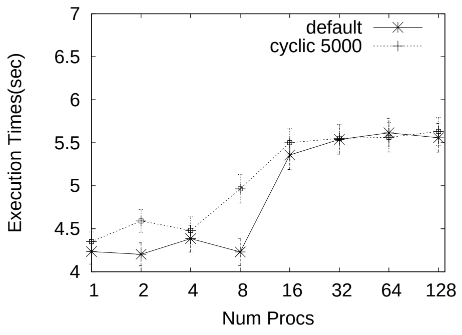
Fig. A-1. `pList` algorithms comparison. Weak scaling each processing unit has 20 millions elements. (a) Parallel for each (b) Parallel accumulate (c) Parallel generate (d) Configuration a use as many processing units as possible in a node; configuration b use as many nodes possible.



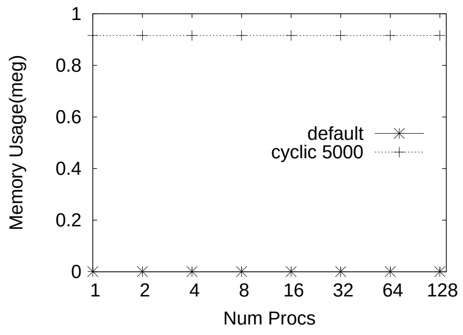
(a) P5-CLUSTER: Constructor(Blocked)



(b) P5-CLUSTER: Metadata

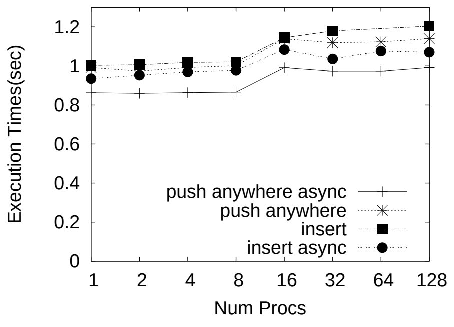


(c) P5-CLUSTER: Constructor(Cyclic)

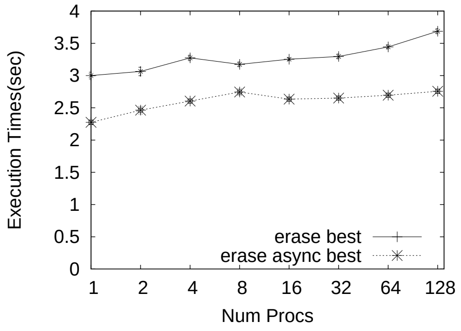


(d) P5-CLUSTER: Metadata

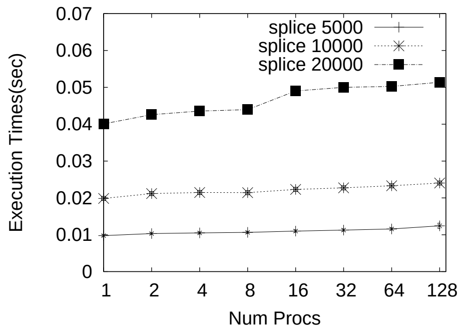
Fig. A-2. `pList` constructor. (a) Execution times of constructors using different number of base containers per location in a blocked distributed fashion (weak scaling each processing unit has 20 millions). (b) Metadata size for the same constructor used in (a). (c) Execution times of constructors using different number of base containers per location in a cyclic distributed fashion (weak scaling each processing unit has 20 millions). (d) Metadata size for the same constructor used in (c).



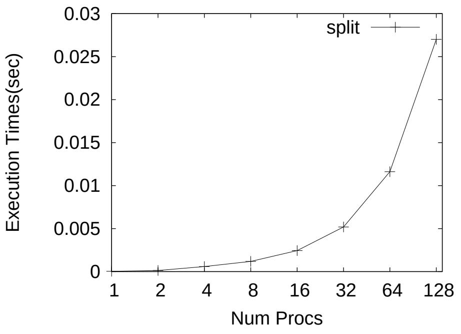
(a) P5-cluster: Methods



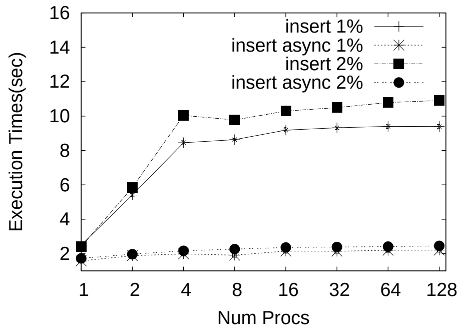
(b) P5-cluster: Erase



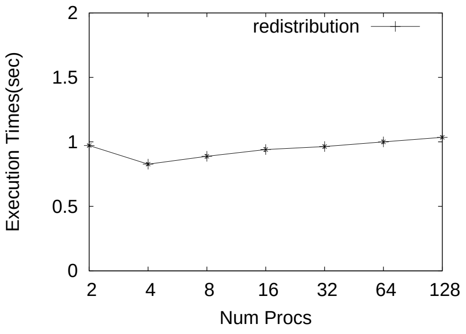
(c) P5-cluster: Splice



(d) P5-cluster: Split

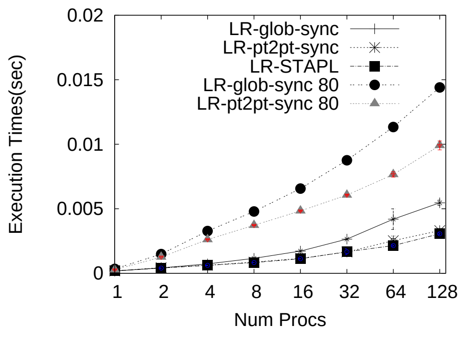


(e) P5-cluster: Insert

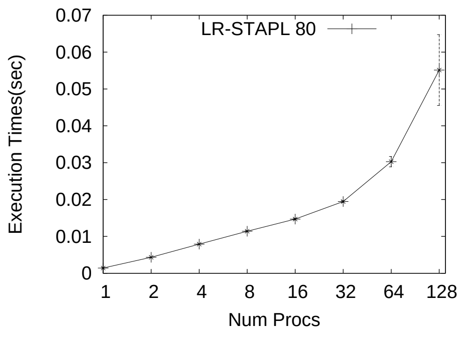


(f) P5-cluster: Redistribution

Fig. A-3. `pList` methods comparison. (a) Weak scaling of representative methods P5-cluster using 5M method invocations per location.(b) Erase method using 5M method invocations per location. (c) Split method with worst case scenario. (d) Splice with various number of base containers per location. (e) Weak scaling of insert usiing 5M method invocations per location with 1% or 2% remote. (f) Redistribution method using 1M elements 1000 base containers per location.

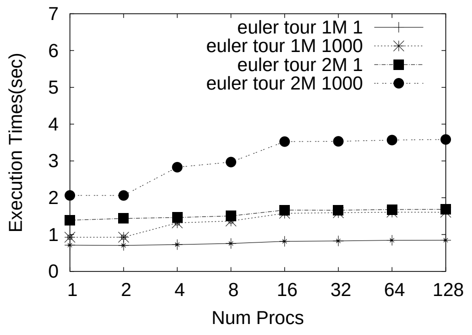


(a)

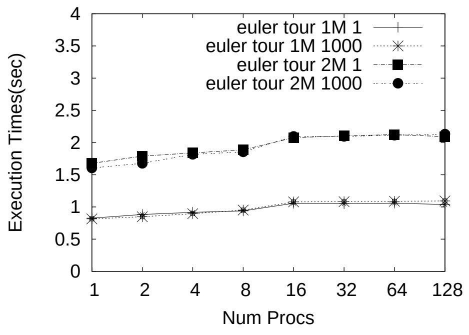


(b)

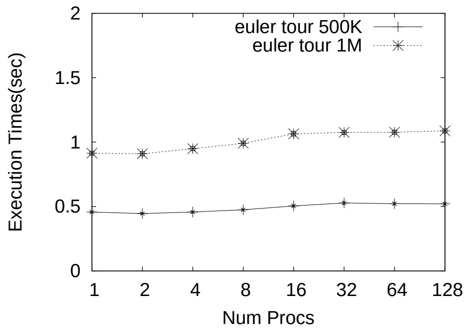
Fig. A-4. List Ranking performance comparison.



(a) List graph

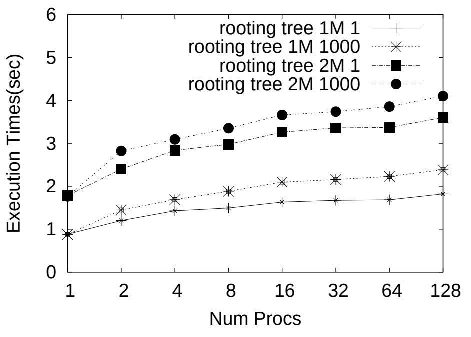


(b) Tree graph

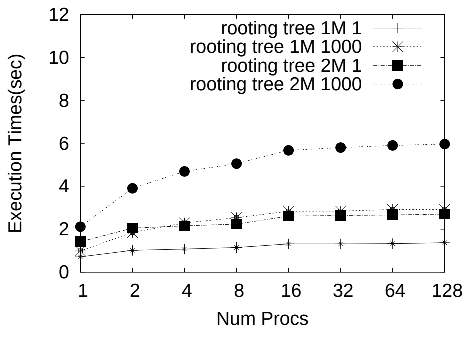


(c) Random tree graph

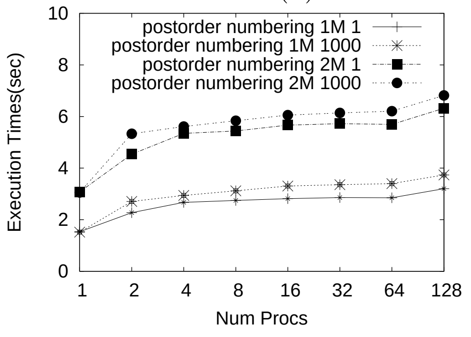
Fig. A-5. Euler Tour algorithm using three different types of trees.



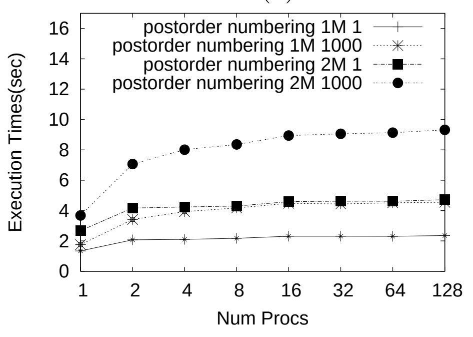
(a)



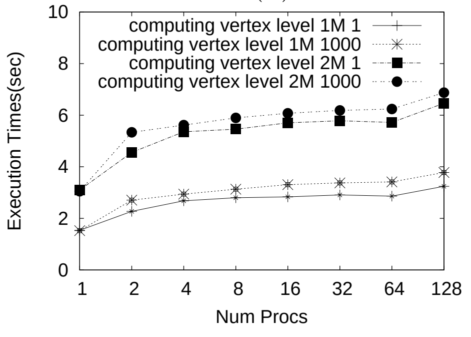
(e)



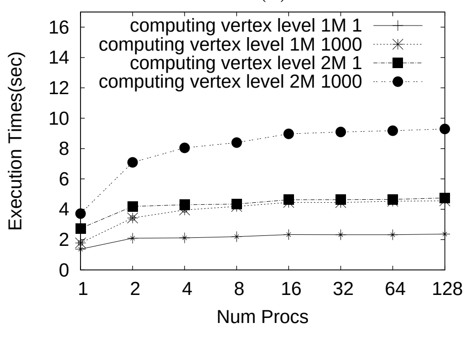
(b)



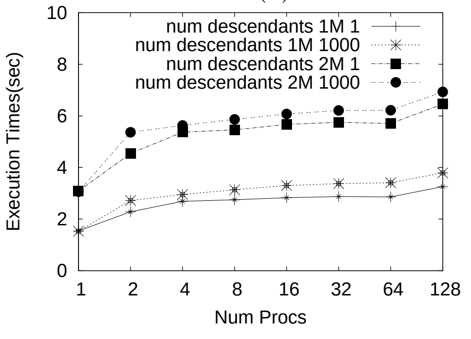
(f)



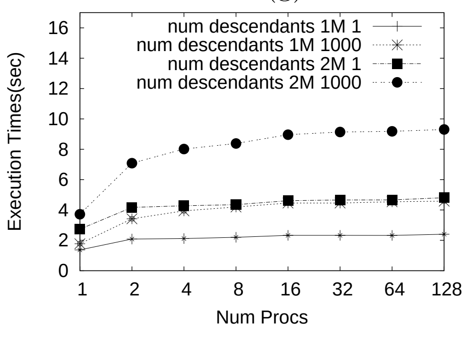
(c)



(g)

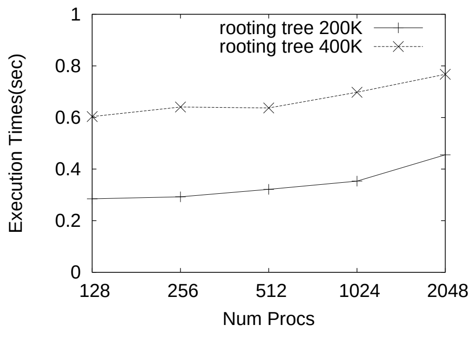


(d)

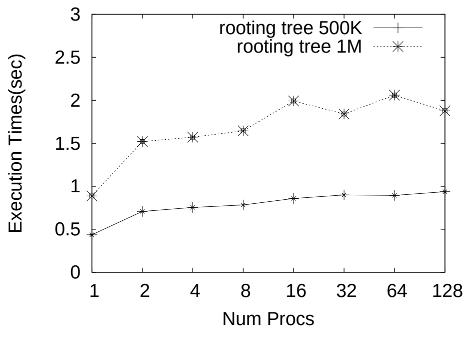


(h)

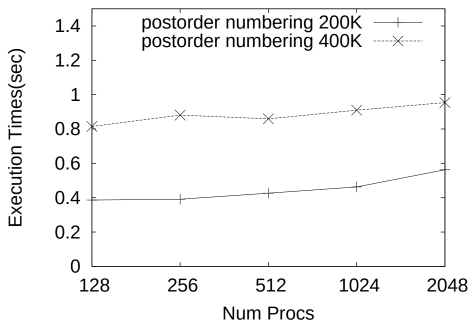
Fig. A-6. Euler Tour application using tree graph (a,b,c,d) and list graph (e,f,g,h).



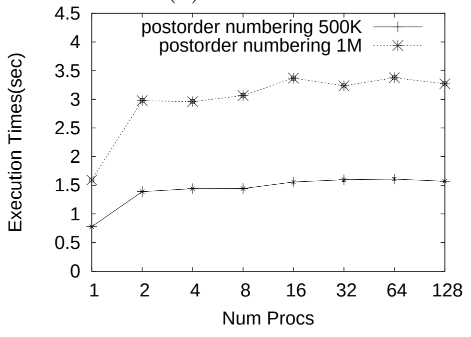
(a) CRAY



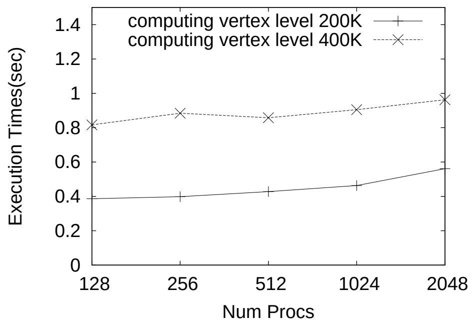
(e) P5-CLUSTER



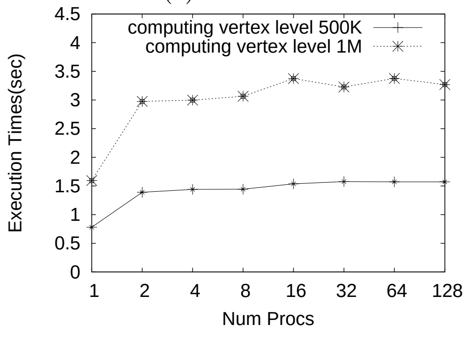
(b) CRAY



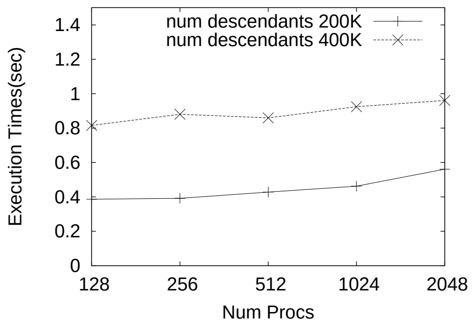
(f) P5-CLUSTER



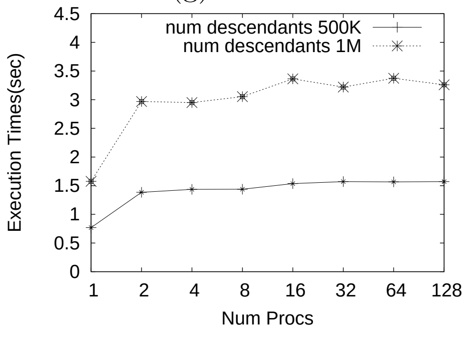
(c) CRAY



(g) P5-CLUSTER

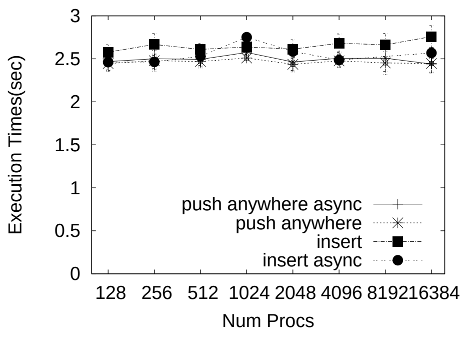


(d) CRAY

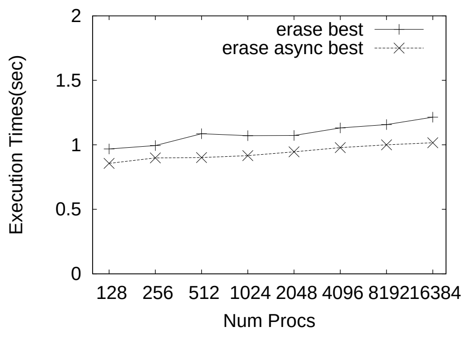


(h) P5-CLUSTER

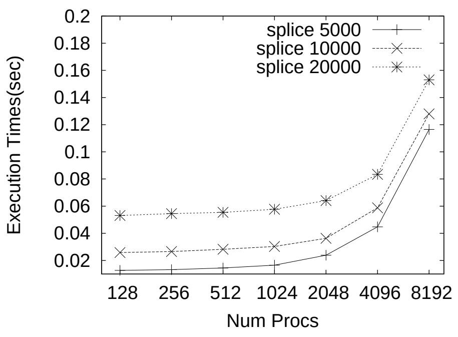
Fig. A-7. Euler Tour application using random tree graph.



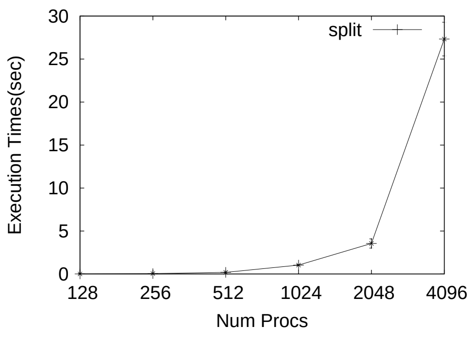
(a) CRAY: Methods



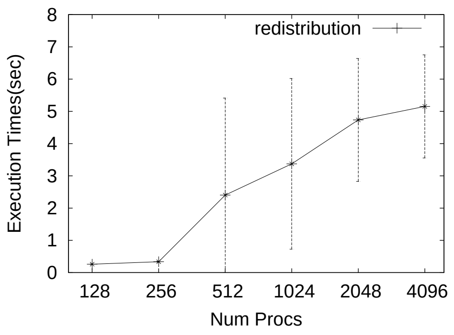
(b) CRAY: Erase



(c) CRAY: Splice



(d) CRAY: Split



(e) CRAY: Redistribution

Fig. A-8. (a) Weak scaling of representative methods using 20M method invocations per location. (b) Weak scaling of erase methods with 5 million invocations per location. (c) Splice method with different number of base containers per location. (d) Split method with worst case scenario. (e) Redistribution method using 1M elements 1000 base containers per location.

APPENDIX B

STL LIST INTERFACE

Table B-I. STL List Interface

Members	**Complexity**	**Where defined**
value_type	O(1)	Container
pointer	O(1)	Container
reference	O(1)	Container
const_reference	O(1)	Container
size_type	O(1)	Container
difference_type	O(1)	Container
iterator	O(1)	Container
const_iterator	O(1)	Container
reverse_iterator	O(1)	Reversible Container
const_reverse_iterator	O(1)	Reversible Container
iterator begin()	O(1)	Container
iterator end()	O(1)	Container
const_iterator begin() const	O(1)	Container
const_iterator end() const	O(1)	Container
reverse_iterator rbegin()	O(1)	Reversible Container
reverse_iterator rend()	O(1)	Reversible Container
const_reverse_iterator rbegin() const	O(1)	Reversible Container
const_reverse_iterator rend() const	O(1)	Reversible Container
size_type size() const	O(1) or O(N)	Container
size_type max_size() const	O(1) or O(N)	Container
bool empty() const	O(1)	Container
list()	O(1)	Container
list(size_type n)	O(N)	Sequence
list(size_type n, const T& t)	O(N)	Sequence
list(const list&)	O(N)	Container
list(InputIterator f, InputIterator l)	O(N)	Sequence
˜list()	O(N)	Container
list& operator=(const list&)	O(N)	Container
reference front()	O(1)	Front Insertion Sequence
const_reference front() const	O(1)	Front Insertion Sequence
reference back()	O(1)	Sequence
const_reference back() const	O(1)	Back Insertion Sequence
void push_front(const T&)	O(1)	Front Insertion Sequence
void push_back(const T&)	O(1)	Back Insertion Sequence
void pop_front()	O(1)	Front Insertion Sequence
void pop_back()	O(1)	Back Insertion Sequence

Table B-I continued. STL List Interface

Members	**Complexity**	**Where defined**
void swap(list&)	O(1)	Container
iterator insert(iterator pos, const T& x)	O(1)	Sequence
void insert(iterator pos, InputIterator f, InputIterator l)	O(N)	Sequence
void insert(iterator pos, size_type n, const T& x)	O(N)	Sequence
iterator erase(iterator pos)	O(1)	Sequence
iterator erase(iterator first, iterator last)	O(N)	Sequence
void clear()	O(N)	Sequence
void resize(n, t = T())	O(N)	Sequence
bool operator==(const list&, const list&)	O(N)	Forward Container
bool operator<(const list&, const list&)	O(N)	Forward Container
void splice(iterator position, list& x)	O(1)	list
void splice(iterator position, list& x, iterator i)	O(1)	list
void splice(iterator position, list& x, iterator f, iterator l)	O(1)	list
void remove(const T& val)	O(N)	list
void remove_if(Predicate p)	O(N)	list
void unique()	O(N)	list
void unique(BinaryPredicate p)	O(N)	list
void merge(list& x)	O(N)	list
void merge(list& x, BinaryPredicate Comp)	O(N)	list
void reverse()	O(N)	list
void sort()	O(NlogN)	list
void sort(BinaryPredicate comp)	O(NlogN)	list

VITA

Xiabing Xu received his B.S. in computer science and engineering at Jilin University, Changchun City, in 2007. He has worked on the research area involving motion planning and parallel library development and published some papers.

- A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: Standard Template Adaptive Parallel Library", in *Proc. of SYSTOR 2010: The 3rd Annual Haifa Experimental Systems Conference*, Haifa, Israel, May 2010.
- G. Tanase, X. Xu, A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger, "The STAPL pList", in *Proc. of the 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Newark, DE, Oct 2009.

More information about Xiabing Xu's research and publications may be found at http://parasol.tamu.edu/people/xiabing. He may be reached at: Parasol Lab, 301 Harvey R. Bright Bldg, 3112 TAMU, College Station, TX 77843-3112.

The typist for this dissertation was Xiabing Xu.