THE STAPL PARALLEL CONTAINER FRAMEWORK

A Dissertation

by

ILIE GABRIEL TANASE

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

DOCTOR OF PHILOSOPHY

December 2010

Major Subject: Computer Science

THE STAPL PARALLEL CONTAINER FRAMEWORK

A Dissertation

by

ILIE GABRIEL TANASE

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

DOCTOR OF PHILOSOPHY

Approved by:

Co-Chairs of Committee,	Lawrence Rauchwerger
	Nancy M. Amato
Committee Members,	Jennifer Welch
	Marvin Adams
Head of Department,	Valerie E. Taylor

December 2010

Major Subject: Computer Science

ABSTRACT

The STAPL Parallel Container Framework. (December 2010)

Ilie Gabriel Tanase, B.S., Polytechnic University Bucharest, Romania;

M.S., Polytechnic University of Bucharest, Romania

Co–Chairs of Advisory Committee: Dr. Lawrence Rauchwerger
Dr. Nancy M. Amato

The Standard Template Adaptive Parallel Library (STAPL) is a parallel programming infrastructure that extends C++ with support for parallelism. STAPL provides a run-time system, a collection of distributed data structures (`pContainers`) and parallel algorithms (`pAlgorithms`), and a generic methodology for extending them to provide customized functionality.

Parallel containers are data structures addressing issues related to data partitioning, distribution, communication, synchronization, load balancing, and thread safety. This dissertation presents the *STAPL Parallel Container Framework (*PCF*)*, which is designed to facilitate the development of generic parallel containers. We introduce a set of concepts and a methodology for assembling a `pContainer` from existing sequential or parallel containers without requiring the programmer to deal with concurrency or data distribution issues. The STAPL PCF provides a large number of basic data parallel structures (e.g., `pArray`, `pList`, `pVector`, `pMatrix`, `pGraph`, `pMap`, `pSet`). The STAPL PCF is distinguished from existing work by offering a class hierarchy and a composition mechanism which allows users to extend and customize the current container base for improved application expressivity and performance.

We evaluate the performance of the STAPL `pContainers` on various parallel machines including a massively parallel CRAY XT4 system and an IBM P5-575 cluster. We show that the `pContainer` methods, generic `pAlgorithms`, and different applica-

tions, all provide good scalability on more than 10^4 processors.

To my wife, Aniela who is always by my side

To my parents

ACKNOWLEDGMENTS

First of all I would like to thank my advisors, Dr. Lawrence Rauchwerger and Dr. Nancy Amato, for their continuous support and encouragement throughout my PhD studies. It has been a long and difficult road with numerous challenges. However together we managed to overcome them no matter how difficult they seemed to be initially. Throughout the years I have learned from them the importance of the high level discussions on why is our research relevant, what distinguishes us from previous and other similar projects and what is the general direction we want to stir our project. As a fresh graduate student I remember being always eager to get things done as soon as possible. From them I have learned that the implementation is not necessarily the most important part. A careful design and documentation that can be discussed and argued for, is more important than the implementation itself. The skills they thought me helped me throughout my PhD studies and hopefully I passed some of this knowledge to the undergraduate and graduate students that I mentored.

I would like to thank my committee members, Dr. Marvin Adams and Dr. Jennifer Welch, who have been of great help with both their comments and creative ideas. With Dr. Adams I interacted while developing a large scale parallel transport application that involved a large number of people from different area of expertise like Computer Science, Nuclear Engineering and Math. I learned from him the importance of providing the right layers of abstractions so that Nuclear Engineering students not familiar with parallel programming can still write efficient programs. From Dr. Jennifer Welch I learned the importance of formally specifying a distributed system behavior so that there is no ambiguity for users when interacting with such a system.

Throughout the graduate school I had the opportunity to interact with a large number of colleagues that all have contributed in one way or another to my education

as a scientist. First I would like to thank all my STAPL colleagues: Alin Jula, Ping An, Paul Thomas, Silvius Rus, Chidambareswaran Raman, Tao Huang, Paul Thomas, Steven Saunders, William McLendon, Lidia Smith, Timmie Smith, Nathan Thomas, Xiabing Xu, Antal Buss, Adam Fidel, Ioannis Papadopoulos, Shuai Ye, Harshvardhan, Mani Zandifar, Jeremy Vu, Olga Pearce, Tarun Jain, Shishir Sharma. For Mauro Bianco, postdoc in our group, special thanks as he carefully read my design documents and help me clarify the aspects that were not well explained. Special thanks to all my colleagues that were next to me during the long nights we spent in the office while working on papers before deadlines. I will never forget those times.

As part of a large research group I also had the opportunity to interact with students in various areas of research. By getting involved with their work on motion planning, protein folding and particle transport, and listening to their needs in terms of data structure support I greatly improved the usability of my work as presented in this thesis. I would like to thank Shawna Thomas, Lydia Tapia, Sam Ade Jacobs, Roger Pearce, Jae Chang, Teresa Bailey and Alex Maslowski. Also I would like to thank you Anna Tikhonova, Jessie Berlin and Anthony Nowak, undergraduate students that I had the opportunity to mentor throughout the years.

For my late master thesis advisor, Irina Athanasiu, I have only words of gratitude. She first introduced me to the world of research and gave me the opportunity to teach as a teaching assistant at the Polytechnic University of Bucharest, Romania. As a fresh graduate student I still remember the emotions I had while teaching colleagues that were only one year younger than me. Irina first mentioned to me that I can enroll in a PhD program abroad and she recommended me Texas A&M University and Dr. Lawrence Rauchwerger. Thank you, Irina!

Finally a big thank you to all my family for their help and support. Aniela, my wife, has been always by my side supporting me in the difficult times and sharing

with me the numerous happy moments we had throughout graduate school. Thank you to my children, David and Daria, for their always joyful mood. They were always the highlight of my day when coming home tired from school after long debugging sessions. Thank you to my father and my late mother for their constant guidance during my first years of school. I am grateful to them for helping me realize at a very young age the importance and the joy of learning. I am trying to pass on to my kids the values they taught me throughout the years.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION . . . 1
- A. Outline . . . 3

II RELATED WORK . . . 5

III STAPL OVERVIEW . . . 10
- A. STAPL pView Concept . . . 13
- B. Runtime System . . . 18

IV PARALLEL CONTAINER . . . 21
- A. pContainer Requirements . . . 21
- B. pContainer Definition . . . 23
  - 1. Set Theory Definitions . . . 24
  - 2. pContainer Domain . . . 26
  - 3. Ordered Domain . . . 26
  - 4. Partition . . . 29
  - 5. Ordered Partition of Total Ordered Domains . . . 30
- C. pContainer Composability . . . 32

V PARALLEL CONTAINER FRAMEWORK . . . 37
- A. pContainer Framework Design . . . 38
- B. pContainer Interfaces . . . 39
- C. Shared Object View Implementation . . . 42
  - 1. Base Container Interface . . . 46
  - 2. Location Manager Interface . . . 48
  - 3. Domain Interface . . . 49
  - 4. Partition Interface . . . 51
  - 5. Partition Mapper Interface . . . 54
  - 6. Data Distribution Manager . . . 55
- D. Specification for pContainer Framework Concepts . . . 59
  - 1. pContainer Base . . . 59
  - 2. Static pContainer . . . 62
  - 3. Dynamic pContainer . . . 63
  - 4. Indexed pContainer . . . 63

CHAPTER Page

5. Associative pContainer 67
6. Relational pContainer 68
7. Sequence pContainer 70
E. Integrating all Concepts using pArray Example 71
F. pContainers Implemented in the Framework 74
G. pContainer Support for Redistribution 77
1. Data Marshaling 79
H. pContainer Customization using Traits 80

VI THREAD SAFETY 83

A. pContainer Thread Safety Design 84
B. Data Distribution Manager 85
C. Thread Safety Manager 86
D. Partition Locking Specification 88
E. pArray and pMatrix 89
F. pList 90
G. Associative pContainers 90
H. pGraph 91

VII MEMORY CONSISTENCY MODEL 92

A. pContainer Interfaces 93
B. Completion Guarantees 95
C. Memory Consistency Conditions 98
1. pContainer Default Memory Consistency Model 98
D. Memory Consistency Example 102
E. Other Memory Consistency Models 103
1. The Default pContainer MCM is not Sequentially Consistent 103
2. The Default pContainer MCM is not Processor Consistent 105
3. Modifying the Default pContainer MCM 105
F. pContainer Method: Developer Side 107
G. Consistency of Other pContainer Methods 110
H. Enforcing Synchronization Points Automatically 111

VIII PCONTAINER PERFORMANCE EVALUATION 113

A. Experimental Setup 113
B. Evaluation of pContainer Methods 114

CHAPTER Page

C. Evaluation of Generic Algorithms 115
D. Specific Applications 115

IX THE STAPL PARRAY 116

A. Example 117
B. The pArray Specification 118
C. pArray Partitions 119
D. pArray Customization 120
E. Performance Evaluation 121
1. Methods 122
2. Algorithms 128
F. Memory Consumption Study 130

X THE STAPL PLIST 133

A. pList Example 134
B. pList Specification 134
C. pList Design and Implementation 136
D. Performance Evaluation 140
E. pList Method Evaluation 141
F. pAlgorithm Comparison 142
G. Comparison of Dynamic Data Structures in STAPL 144
H. Application using pList: Euler Tour 145

XI THE STAPL PGRAPH 149

A. pGraph Example 150
B. pGraph Concepts and Interfaces 151
C. pGraph Class Hierarchy 153
D. pGraph Design and Implementation 157
E. pGraph pViews 159
F. Performance Evaluation 161
1. pGraph Methods Evaluation 161
2. Evaluation of Address Translation Mechanisms 164
3. pGraph Algorithms 167
4. Page Rank 172

XII ASSOCIATIVE PCONTAINERS 173

A. Associative pContainer Specification 174
B. Associative pContainer Design and Implementation 176

CHAPTER Page

C. Performance Evaluation 179
1. MapReduce 179
2. Generic Algorithms 180

XIII PCONTAINER COMPOSITION EVALUATION 182

XIV CONCLUSION AND FUTURE WORK 186

REFERENCES 189

VITA 199

LIST OF TABLES

TABLE Page

I Comparison with related projects. 8

II STAPL pViews and corresponding operations. 16

III Base container interface. 47

IV Location manager interface. 48

V Ordered domain interface. 50

VI Finite ordered domain interface. 51

VII Partition base interface. 52

VIII Ordered partition interface. 53

IX Partition mapper interface. 54

X Data distribution manager interface. 58

XI Base pContainer interface. 60

XII Static pContainer interface. 62

XIII Dynamic pContainer interface. 63

XIV Indexed pContainer interface. 64

XV Indexed partition interface. 65

XVI Associative pContainer interface. 67

XVII Relational pContainer interface. 68

XVIII Sequence pContainer interface. 70

XIX pArray interface. 118

XX pArray partitions. 119

TABLE Page

XXI pArray traits. 120

XXII pArray memory consumption. 130

XXIII Theoretical memory usage for pArray. 131

XXIV pList interface. 135

XXV Vertex reference interface. 152

XXVI Edge reference interface. 153

XXVII pGraph interface. 155

XXVIII Associative pContainers interface. 175

LIST OF FIGURES

FIGURE Page

1 STAPL overview. 11

2 Overlap pView example. 17

3 Composed `pArray` of `pArrays`. 33

4 Composed `pContainers`. 34

5 PCF design. 38

6 Shared object view. 43

7 `pContainer` address resolution. 45

8 The `invoke` method of the data distribution manager. 56

9 The `pContainer` indexed method implementation. 65

10 Example of pContainer deployment on two locations. 71

11 Pseudocode of `pArray` `set()` method. 73

12 `pContainers` inheritance. 74

13 Redistribution for two given partitions. 78

14 Marshaling interfaces. 80

15 `pContainer` template arguments. 81

16 `pGraph` customization. 82

17 Generic `invoke` method implementation with locking statements. . . 87

18 User, pContainer, run time system interaction. 93

19 STAPL program execution. 94

20 Completion guarantees. 97

FIGURE Page

21 Asynchronous methods ordering. 101

22 Relaxed completion order: (a) Operations on different pContainer elements receive their acknowledgments out of order (b) Dekker's mutual exclusion. 104

23 Processor consistency counter example. 106

24 Kernel used to evaluate the performance of pContainer methods. . . 114

25 Derivation chain for pArray. 116

26 pArray example. 117

27 pArray constructor execution time for various input sizes on (a) CRAY4 and (b) P5-cluster. 121

28 CRAY4: pArray local method invocations for various container sizes. 123

29 CRAY4: pArray methods for various input sizes. 124

30 CRAY4: pArray methods set_element, get_element and split phase get_element. 125

31 pArray methods for various percentage of remote invocations. 126

32 CRAY4: pArray local and remote method invocations for various container sizes. 127

33 Execution times for generic algorithms on CRAY4 for a pArray with 20M elements per processor. 129

34 CRAY4: pArray memory usage study. 132

35 Derivation chain for pList. 133

36 pList example. 135

37 Different partitions and mappings for pList. 138

38 pList method implementation. 140

FIGURE Page

39 Execution times for pList methods. 141

40 Execution times for p_for_each, p_generate, p_accumulate algorithms on CRAY4 for pArray and pList. 142

41 P5-CLUSTER: Weak scaling for p_for_each allocating processes on the same nodes when possible (curve a) or in different nodes (curve b). 143

42 Comparison pList and pVector dynamic data structures using a mix of 10M operations (read/write/insert/delete). 144

43 Weak scaling of Euler Tour algorithm. 147

44 Execution times for Euler Tour and its applications using a tree made by a single binary tree with 500k or 1M subtrees per processor. 148

45 pGraph example. 150

46 Graph hierarchy of concepts. 154

47 pGraph pViews example. 159

48 pGraph pViews: (a) pGraph partitioned pView, (b) region_pview, (c) inner_pview and (d) boundary_pview. 160

49 CRAY4: Evaluation of static and dynamic pGraph methods while using the SSCA2 graph generator. 162

50 P5-CLUSTER: Evaluation of static and dynamic pGraph methods while using the SSCA2 graph generator. 163

51 Find sources in a directed pGraph using static, dynamic with forwarding and dynamic with no forwarding partitions. 165

52 Comparison of various pGraph partitions. 166

53 CRAY4: Execution times for different pGraph algorithms. 168

54 CRAY4: pGraph algorithms. 169

55 P5-CLUSTER: Execution times for different pGraph algorithms. . . . 170

FIGURE Page

56 Page rank for two different input meshes: 1500x1500 and 15x150000. 172

57 Associative pContainer: (a) derivation from the framework base classes (b) associative pContainers internal hierarchy. 174

58 Value based partition for sorted associative pContainers. 177

59 MapReduce used to count the number of occurrences of every word in Simple English Wikipedia website (1.5GB). 179

60 CRAY4: Scalability for generic algorithms when using associative pContainers. 181

61 Example of pContainer composition and nested pAlgorithm invocation. 183

62 Comparison of pArray<pArray<>> ($pa < pa >$), plist<pArray<>> ($plist < pa >$) and pMatrix on computing the minimum value for each row of a matrix. 184

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. The Standard Template Adaptive Parallel Library (STAPL) [16, 4, 67, 5, 68, 52, 63, 67, 65, 64, 15, 13, 66] is being developed to help programmers address the difficulties of parallel programming. STAPL is a parallel C++ library with functionality similar to STL, the ISO adopted C++ Standard Template Library [49]. STL is a collection of basic algorithms, containers and iterators that can be used as high-level building blocks for sequential applications. Similar to STL, STAPL provides a collection of parallel algorithms (`pAlgorithms`), parallel and distributed containers (`pContainers`) [63, 65, 64, 15, 66], and `pViews` to abstract the data access in `pContainers`. STAPL provides the building blocks for writing parallel programs and the mechanisms (glue) to put them together in large programs. An essential building block for such a generic library is its data structures. Sequential libraries such as STL [49], BGL [30], and MTL [28], provide the user with a collection of data structures and algorithms that simplifies the application development process. Similarly, STAPL provides the Parallel Container Framework (PCF) to facilitate the development of `pContainers` which are parallel and concurrent data structures.

`pContainers` are containers that are distributed across a parallel machine and accessed concurrently. A large number of parallel data structures have been proposed in the literature. They are often complex data structures, addressing issues related to data partitioning, distribution, communication, synchronization, load balancing,

---

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

and thread safety. The complexity of building such structures for every parallel program is one of the main impediments to parallel program development. To alleviate this problem we have developed the *STAPL Parallel Container Framework* (PCF). It consists of a collection of elementary `pContainers` and methods to specialize, or compose them into `pContainers` of arbitrary complexity. Thus, instead of building their distributed containers from scratch in an *ad-hoc* fashion, programmers can use inheritance to derive new specialized containers and composition to generate complex data structures. Moreover, the PCF provides the mechanisms to enable any container, sequential or parallel, to be used in a distributed fashion without requiring the programmer to deal with concurrency mechanisms such as data distribution or thread safety. Furthermore, when composed, these containers carry over, at every level, their "interesting" features for parallelism.

The STAPL PCF presented in this thesis makes several novel contributions.

- Modular design: Provides a set of classes and rules for using them to build new `pContainers` and customize existing ones.
- Composition: Supports composition of `pContainers` that allows the recursive development of complex `pContainers` that support nested parallelism.
- Interoperability: Provides mechanisms to generate a wrapper for any data structure, sequential or parallel, enabling it to be used in a distributed, concurrent environment.
- Library: It provides a library of basic `pContainer` constructed using the PCF as initial building blocks.

Some important properties of `pContainers` supported by the PCF are noted below.

Shared object view. Each `pContainer` instance is globally addressable. This supports ease of use, relieving the programmer from managing and dealing with the distribution explicitly, unless desired.

Arbitrary degree and level of parallelism. For `pContainers` to provide scalable performance on shared and/or distributed memory systems they must support an arbitrary, tunable degree of parallelism, e.g., number of threads. Moreover, given the importance of hierarchical (nested) parallelism for current and foreseeable architectures, it is important for composed `pContainers` to allow concurrent access to each level of their hierarchy.

Instance-specific customization. The `pContainers` in the PCF can be dynamic and irregular and can adapt (or be adapted by the user) to their environment. The PCF facilitates the design of `pContainers` that support advanced customizations so that they can be easily adapted to different parallel applications or even different computation phases of the same application. For example, a `pContainer` can dynamically change its data distribution or adjust its thread safety policy to optimize the access pattern of the algorithms accessing the elements. Alternatively, the user can request certain policies and implementations which can override the provided defaults or adaptive selections.

Portions of this dissertation have been published in the following papers [63, 65, 15, 64, 13, 68].

A. Outline

The dissertation is outlined as follows. In Chapter II we present related work. In Chapter III we provide an overview of STAPL discussing the main modules of the library and emphasizing the ones that the `pContainers` will interact directly with:

parallel view (pView) and run time system (RTS). Chapter IV introduces the notion of a parallel container as a parallel and distributed data structure describing the required functionality that it needs to provide to improve parallel programming productivity. Chapter V provides a detailed description of the Parallel Container Framework and of the individual modules that make up a parallel container. Chapter V, Section E shows an example of STAPL pContainer implemented using the framework describing how all modules introduced in Chapter V interact to provide the overall functionality of a parallel array (pArray) data structure.

Chapter V , Section D introduces the base pContainers provided by the framework. Chapter VI discusses the thread safety support provided by default by all STAPL pContainers and introduces the interfaces that advanced users need to customize in order to implement custom thread safety policies. Chapter VII describes the memory consistency model provided by the pContainers and specifies how additional models can be implemented.

Starting with Chapter VIII we describe individual pContainers implemented using the PCF and an evaluation of their performance. We describe their derivation relation from the classes of the framework, their interfaces and show experimental results or both pContainer methods and pAlgorithms. Chapter VIII introduces the general methodology used to evaluate the pContainer methods and application performance. Chapter IX discusses the pArray, Chapter X the pList, Chapter XI the pGraph and Chapter XII associative pContainers. Experimental results evaluating the pContainer composition are included in Chapter XIII. We conclude with some final remarks in Chapter XIV.

CHAPTER II

RELATED WORK

There is a large body of work in the area of parallel and distributed data structures with projects aiming at shared memory architectures, distributed memory architectures or both. Parallel programming languages [19, 18, 17, 70] typically provide built in arrays and provide minimal guidance to the user on how to develop their own specific parallel data structures. STAPL `pContainers` are generic data structures and this characteristic is shared by a number of existing projects such as PSTL [39], TBB [37], and POOMA [53].

There are several parallel libraries that have similar goals to STAPL. Some of the libraries provide application specific data structures that are fine tuned for certain applications. While they achieve high efficiency, lack of generality makes them hard to use in different applications. A large amount of effort has been put into regular data structures, like arrays and matrices, to make them suitable for parallel programming. Irregular data structures, like graph, tree, etc., are not widely studied for this purpose.

PSTL (Parallel Standard Template Library) [39, 40] explores the same underlying philosophy as STAPL, which is to extend the C++ STL for parallel programming. PSTL emphasizes regular data structures, such as vector, multidimensional array, etc., which are more suited for scientific computation. The underlying runtime system provides support for explicit shared memory operations, such as put and get. The data distribution mechanism allows for regular distribution (block distribution) and the containers are treated as place holders for data accessed by stand alone parallel algorithms. PSTL project is unfortunately not maintained anymore. The Amelia Vector Template Library (AVTL) [59], provides a parallel vector data structure, which can be distributed in an uniform fashion. STAPL is different that both PSTL and AVTL,

providing a larger variety of data structures integrated uniformly in a framework.

Hierarchically Tiled Arrays (HTA) [8, 9] is introduced as a useful programming paradigm where the user writes programs in a single threaded fashion and the data structure takes care of parallelization in a transparent manner. HTA is a parallel data container whose data is partitioned in tiles and can be distributed across different computation servers. Operations on the data are dispatched to the servers owning the data. The most representative characteristic of an HTA is the support for hierarchical partitioning and indexing. Through a flexible indexing scheme for its tiles and elements HTAs allows communication to be expressed as array assignments. Hierarchical data structures can provide more flexibility to the user to express its algorithms, can be used to improve data locality (e.g., tiling[45]), and to express nested parallelism[10]. Some of the HTA concepts have been adopted in STAPL, like providing hierarchical views of the data available in `pContainers`.

POOMA[53] is a C++ library designed to provide a flexible environment for data parallel programming of scientific applications. POOMA provides a collection of parallel data types together with a set of algorithms geared specifically toward scientific applications. STAPL shares similar goals with POOMA. Code developed using POOMA is intended to be portable, efficient, allows rapid application development by reusing existing components. The data structures provided by POOMA are referred to as "Global Data Types". They are similar to STAPL `pContainers` but they are oriented toward scientific computing. POOMA provides n-dimensional arrays, vector and matrix classes. STAPL's `pContainer` infrastructure is more generic providing a larger variety of data structures like graphs, list, hash maps.

Multiphase Specifically Shared Array (MSA) [21] is proposed as a data structure that allows the users to benefit from having a distributed data structure with shared memory interface. MSA is an important data structure developed using Charm++

language[41]. To improve the performance of accessing the elements of the MSA the authors optimize for three access patterns that are common in many algorithms: read only, write many and accumulate. The authors emphasize that while a read write mode would be more general it is often hard to guarantee memory consistency without performance loss. The access modes can be interchanged at certain points in the program (synchronization points).

There has been significant research in the field of parallel and concurrent data structures. Much work has focused on providing efficient locking mechanisms and methodologies for transforming existing sequential data structures into concurrent data structures [20, 24, 26, 33, 34]. Valois [69] was one of the first to present a non-blocking singly-linked list data structure by using Compare&Swap (CAS) synchronization primitives rather than locks. The basic idea is to use auxiliary nodes between each ordinary node to solve the concurrency issues. Subsequent work [31, 48, 27, 51] proposes different concurrent list implementations for shared memory architectures, emphasizing the benefits on non-blocking implementations in comparison with lock based solutions. Investigations of concurrent hash tables [24, 26, 48] and search trees (the most common internal representation for maps and sets) [43, 47] explore efficient storage schemes, different lock implementations, and different locking strategies (e.g., critical sections, non-blocking, wait-free [33]), especially in the context of shared memory architectures. In contrast, the STAPL `pContainers` are designed to be used in both shared and distributed memory environments and addresses the additional complexity required to manage the data distribution. Ideas pioneered in these papers can be integrated in our framework for efficient concurrent access on a shared memory location.

UPC[25], Titanium[70], Chapel[17] and X10[18] are several other languages that are part of the large family of Partitioned Global Address Space (PGAS) languages.

They all provide minimal support for parallel data structures in the form of regular arrays. Dynamic data structures like graphs, maps, hash maps are often user responsability with no explicit support from the language. All these languages recognizes the importance of allowing users to customize the data distribution and they provide appropriate support for parallel arrays. In [22] it is mentioned that Chapel intends to provide support for distributions for dynamic data structures, a feature that is supported in our framework.

Table I.: Comparison with related projects.

Features/ Project	Paradigm [1]	Architecture	Adaptive	Generic	Data Distribution
STAPL	S/MPMD	Shared/Dist	Yes	Yes	Auto/User
PSTL	SPMD	Shared/Dist	No	Yes	Auto
Charm++	MPMD	Shared/Dist	No	Yes	User
CILK	S/MPMD	Shared/Dist	No	No	User
NESL	S/MPMD	Shared/Dist	No	Yes	User
POOMA	SPMD	Shared/Dist	No	Yes	User
SPLIT-C	SPMD	Shared/Dist	Np	No	User
X10	S/MPMD	Shared/Dist	No	Yes	Auto
Chapel	S/MPMD	Shared/Dist	No	Yes	Auto
Titanium	S/MPMD	Shared/Dist	No	No	Auto
Intel TBB	SPMD	Shared	No	Yes	Auto

[1] SPMD - Single Program Multiple Data, MPMD - Multiple Program Multiple Data

The STAPL PCF differs from the other languages and libraries by focusing on developing a generic infrastructure that will efficiently provide a shared memory abstraction for `pContainers`. The framework automates, in a very configurable way, aspects relating to data distribution and thread safety. We emphasize on interoperability with other languages and libraries [15], and we use a compositional approach

where existing data structures (sequential or concurrent, e.g., TBB containers) can be used as building blocks for implementing parallel containers. We include in Table I a comparison between STAPL and a number of other projects according to a number of criteria. While all the libraries above share a common goal, to make parallel programming easier, STAPL `pContainer` distinguish itself by placing emphasis on *general* (regular and irregular) data structures (vector, list, graph, hash, etc), flexible mechanisms to specify *data distribution*, a *shared object view* programming paradigm with *implicit communication*, *adaptivity* support for both algorithms and containers.

CHAPTER III

STAPL OVERVIEW

STAPL [16, 4, 67, 5, 68, 52] is a framework for parallel code development in C++. Its core is a library of C++ components with interfaces similar to the (sequential) ISO C++ standard library [49]. STAPL offers to the parallel system programmer a shared object view of the data space. The objects are distributed across the memory hierarchy which can be shared and/or distributed address spaces. Internal STAPL mechanisms assure an automatic translation from one space to another, presenting a unified address space to the less experienced user. For more experienced users, the local/remote distinction of accesses can be exposed and performance enhanced for a specific application or application domain. To exploit large hierarchical systems, such as BlueGene [50], Cray XT5 [56], STAPL allows for (recursive) nested parallelism.

The STAPL infrastructure consists of platform independent and platform dependent components that are revealed to the programmer at an appropriate level of detail through a hierarchy of abstract interfaces (see Figure 1). The platform independent components include the core parallel library, and an abstract interface to the communication library and run-time system. The core STAPL library consists of `pAlgorithms` (parallel algorithms) and `pContainers` (distributed data structures) [66]. Important aspects of all STAPL components are *extendability* and *composability*. For example, users can extend and specialize STAPL `pContainers` (using inheritance) and/or compose them. For example, STAPL users can employ `pContainers` of `pContainers` in `pAlgorithms` which may themselves call `pAlgorithms`.

`pContainers`, the distributed counterpart of STL containers, are thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. They are composable and extensible by users via inheritance. Cur-

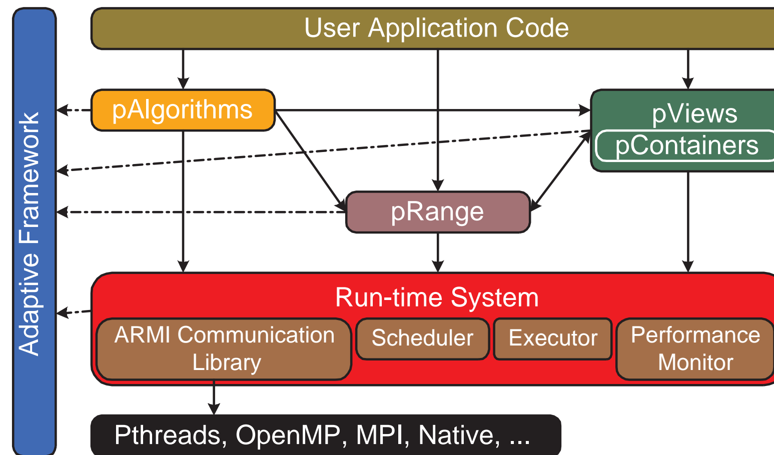


Fig. 1. STAPL overview.

rently, STAPL provides counterparts of all STL containers (e.g., `pArray`[63], `pVector`, `pList`[65], `pMap`[64], etc.), and `pContainers` that do not have STL equivalents: parallel matrix (`pMatrix` [15]) and parallel graph (`pGraph`). `pContainers` provide two kinds of methods to access their data: methods which are semantically equivalent to their sequential counterpart and methods which are specific to parallel computations. For example, STAPL provides an `insert_async` method that can return control to the caller before its execution completes. 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 provided by an internal object translation mechanism which can transparently locate both local and remote elements. The physical distribution of a `pContainer` data can be assigned automatically by STAPL or can be user-specified.

A `pAlgorithm` is the parallel equivalent of an STL algorithm. STAPL currently includes a large collection of parallel algorithms, including parallel counterparts of STL

algorithms, `pAlgorithms` for important parallel algorithmic techniques (e.g., prefix sums [38], the Euler tour technique [38]), and some for use with STAPL extensions to STL (i.e., graph algorithms for the `pGraph`). Analogous to STL algorithms that use *iterators*, STAPL `pAlgorithms` are written in terms of `pViews` [13, 14]. Briefly, `pViews` allow the same `pContainer` to present multiple interfaces to its users, e.g., enabling the same `pMatrix` to be 'viewed' (or used) as a row-major or column-major matrix or even as linearized vector.

`pAlgorithms` are represented by `pRanges`. Briefly, a `pRange` is a graph whose vertices are tasks and the edges the dependencies, if any, between them. A task includes both *work* (represented by what we call *workfunctions*) and *data* (from `pContainers`, generically accessed through `pViews`). The `executor`, itself a distributed shared object, is responsible for the parallel execution of computations represented by `pRanges`; as tasks complete, the `executor` updates dependencies, identifies tasks that are ready for execution, and works with the `scheduler` to determine which tasks to execute. Nested parallelism can be created by invoking a `pAlgorithm` from within a task.

The platform dependent STAPL components are mainly contained in the STAPL runtime system (RTS) [54, 55, 57, 58], which provides the API to the OS and several important functions. The RTS includes the communication abstractions that are used by the higher level STAPL components.

In the following two sections we provide more details regarding the STAPL `pView` concept and the runtime system because they are required to properly describe `pContainer` functionality.

A. STAPL pView Concept

Decoupling of data structures and algorithms is a common practice in generic programming. STL, the C++ Standard Template Library, obtains this abstraction by using *iterators*, which provide a generic interface for algorithms to access data which is stored in containers. This mechanism enables the same algorithm to operate on multiple containers. In STL, different containers support various types of iterators that provide appropriate functionality for the data structure, and algorithms can specify which types of iterators they can use. The major capability provided by the iterator is a mechanism to traverse the data of a container.

The STAPL `pView` [13, 14] generalizes the iterator concept by providing an abstract data type (ADT) for the data it represents. While an iterator corresponds to a single element, a `pView` corresponds to a collection of elements. Also, while an iterator primarily provides a traversal mechanism, `pViews` provide a variety of operations as defined by the ADT. For example, all STAPL `pViews` support size() operations that provide the number of elements represented by the `pView`. A STAPL `pView` can provide operations to return new `pViews`. For example, a `pMatrix` supports access to rows, columns, and blocks of its elements through row, column and blocked `pViews`, respectively.

A primary objective of the `pViews` is that they are designed to enable parallelism. In particular, each ADT supported by STAPL provides random access to collections of its elements. The size of these collections can be dynamically controlled and typically depends on the desired degree of parallelism. For example, the `pList` `pView` provides concurrent access to segments of the list, where the number of segments could be set to match the number of parallel processes. The `pView` provides random access to a partitioned data space. This capability is essential for the scalability of STAPL

programs. To mitigate the potential loss of locality incurred by the flexibility of the random access capability, `pViews` provide, to the degree possible, a remapping mechanism of a user specified `pView` to the collection's physical distribution (known as the native `pView`).

In this section, we first introduce the `pView` concept and then explain how it can be generalized for the parallel and distributed environment of STAPL. A `pView` is a class that defines an abstract data type (ADT) for the *collection of elements* it represents. As an ADT, a `pView` provides *operations* to be performed on the collection, such as read, write, insert, and delete.

`pViews` have *reference semantics*, meaning that a `pView` does not own the actual elements of the collection but simply *references* to them. The collection is typically stored in a `pContainer` to which the `pView` refers; this allows a `pView` to be a relatively light weight object as compared to a container. However, the collection could also be another `pView`, or an arbitrary object that provides a container interface. With this flexibility, the user can define `pViews` over `pViews`, and also `pViews` that generate values dynamically, read them from a file, etc.

All the operations of a `pView` must be routed to the underlying collection. To support this, a mapping is needed from elements of the `pView` to elements of the underlying collection. This is done by assigning a unique identifier to each `pView` element (assigned by the `pView` itself); the elements of the collection must also have unique identifiers. Then, the `pView` specifies a *mapping function* from the `pView`'s *domain* (the union of the identifiers of the `pView`'s elements) to the collection's domain (the union of the identifiers of the collection's elements).

More formally, a `pView` $\mathcal{V}$ is a tuple

$$\mathcal{V} \stackrel{def}{=} (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}) \tag{3.1}$$

where $\mathcal{C}$ represents the underlying collection, $\mathcal{D}$ defines the domain of $\mathcal{V}$, $\mathcal{F}$ represents the mapping function from $\mathcal{V}$'s domain to the collection's domain, and $\mathcal{C}$ is the set of operations provided by $\mathcal{V}$.

To support parallel use, the $\mathcal{C}$ and $\mathcal{D}$ components of the `pView` can be partitioned so that they can be used in parallel. Also, most generally, the mapping function $\mathcal{F}$ and the operations $\mathcal{O}$ can be different for each component of the partition. That is, $\mathcal{C} = \{c_0, c_1, \ldots, c_{n-1}\}$, $\mathcal{D} = \{d_0, d_1, \ldots, d_{n-1}\}$, $\mathcal{F} = \{f_0, f_1, \ldots, f_{n-1}\}$, and $\mathcal{O} = \{o_0, o_1, \ldots, o_{n-1}\}$. This is a very general definition and not all components are necessarily unique. For example, the mapping functions f_i and the operations o_i may often be the same for all $0 \leq i < n$. The tuples (c_i, d_i, f_i, o_i) are called the *base views* (`bViews`) of the `pView` $\mathcal{V}$. The `pView` supports parallelism by enabling random access to its `bViews`, which can then be used in parallel by `pAlgorithms`.

Note that we can generate a variety of `pViews` by selecting appropriate components of the tuple. For instance, it becomes straightforward to define a `pView` over a subset of elements of a collection, e.g., a `pView` of a block of a `pMatrix` or a `pView` containing only the even elements of an array. As another example, `pViews` can be implemented that transform one operation into another. This is analogous to back-inserter iterators in STL, where a write operation is transformed into a push back invocation in a container.

Example. A common concept in generic programming is a one-dimensional array of size n supporting random access. The `pView` corresponding to this will have an integer domain $\mathcal{D} = [0, n)$ and operations $\mathcal{O}$ including the random access read and write operators. This `pView` can be applied to any container by providing a mapping function $\mathcal{F}$ from the domain $\mathcal{D} = [0, n)$ to the desired identifiers of the container. If the container provides the operations, then they can be inherited using the mechanisms

Table II. STAPL `pViews` and corresponding operations. `tranform_pview` implements an overridden read operation that returns the value produced by a user specified function, the other operations depends on the `pView` the transform `pView` is applied to. `insert_any` refers to the special operations provided by STAPL `pContainers` that insert elements in unspecified positions.

	read	write	[ ]	begin/end	insert/erase	insert_any	find
`array_1d_pview`	X	X	X	X			
`array_1d_ro_pview`	X		X	X			
`static_list_pview`	X			X			
`list_pview`	X	X		X	X	X	
`matrix_pview`	X	X	X				
`graph_pview`	X	X			X	X	X
`strided_1D_pview`	X	X	X	X			
`transform_pview`	O		-	-			
`balanced_pview`	X		X	X			
`overlap_pview`	X		X	X			
`native_pview`	X		X	X			

provided in the base `pView` in STAPL. If new behavior is needed, then the developer can implement it explicitly.

Table II shows an initial list of `pViews` available in STAPL. Some special cases of `pViews` are particularly useful in the context of parallel programming. For instance the *single-element partition*, where the domain of the collection is split into single elements and all mapping functions are identity functions. This is the default partition adopted by STAPL when calling a `pAlgorithm` to express maximum parallelism.

Other `pViews` that can be defined include the *balanced* `pView` where the data is split into a given number of chunks, and the *native* `pView`, where the partitioner

<u>Overlap pView of $A[0, 10]$</u>

For $c = 2$, $l = 2$, and $r = 1$,

ith element is $A[c \cdot i, c \cdot i + 4]$

elements of the overlap pView:

$A[0, 4]$, $A[2, 6]$, $A[4, 8]$, $A[6, 10]$

Fig. 2. Overlap pView example. The input is the `pContainer` $A[0, 10]$.

takes information directly from the underlying container and provides `bViews` that are *aligned* with the `pContainer` distribution. This turns out to be very useful in the context of STAPL. Another `pView` heavily used in STAPL is the *overlap* `pView`, in which one element of the `pView` overlaps another element. This `pView` is naturally suited for specifying many algorithms, such as adjacent differences, string matching, etc. As an example, we can define an overlap `pView` for a one-dimensional array $A[0, n-1]$ using three parameters, c (core size), l (left overlap), and r (right overlap), so that the ith element of the overlap `pView` $v^o[i]$ is $A[c \cdot i, c \cdot i + l + c + r - 1]$. See example in Figure 2.

The *native* `pView` is a `pView` whose partitioned domain $\mathcal{D}$ matches the data partition of the underlying collection, allowing references to its data to be local. The *balanced* `pView` partitions the data set into a user specified number of pieces. This `pView` can be used to balance the amount of work in a parallel computation. If STAPL algorithms can use balanced or native `pViews`, then performance is greatly enhanced.

All STAPL `pContainers` provide native `pViews` that have the same interface as

the pContainer. For example, pArray provides array_1d_pview, pList provides p_list_pview, pVector provides p_vector_pview, pGraph provides p_graph_pview, simple associative pContainers provide p_set_pview, pair associative pContainers provide p_map_pview and pMatrix provides array_2d_pview. Additional pViews with certain ADT can be defined on top of existing data structures. For example a pGraph pView can be defined on top of a pArray of list of edges as shown in [14].

B. Runtime System

The STAPL runtime system (RTS) [54, 55, 57, 58] is the only platform specific component of the library that needs to be ported to each target architecture. It provides a communication and synchronization library (ARMI), an *executor*, and a *scheduler* of the tasks of the pRanges. The RTS is not intended to be used directly by the STAPL user or library developer.

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 address space and has associated execution capabilities (e.g., threads). Different locations can communicate exclusively through ARMI, the Adaptive Remote Method Invocation library, which represents the communication layer of the RTS. Special types of objects, called p_objects, implement the basic concept of a shared object. The representative of a p_object in each location has to *register* with the RTS to enable Remote Method Invocations (RMIs) between the representative objects. This is the reason why all the parallel objects in STAPL inherit from the base p_object class. RMIs enable the exchange of data between locations and the transfer of the computation from one location to another.

RMIs are divided into two classes: *asynchronous* RMIs and *synchronous* RMIs.

The former execute a method on a registered object in a remote location without waiting for its termination, while the latter block waiting for the termination of the invoked method. A mechanism is provided to asynchronously execute methods that return values to the caller. As parallel machine sizes reach processor counts into the millions, it becomes essential for algorithms to be implemented using only asynchronous RMIs. In STAPL, these operations implement computation migration, which allows scalability for very large numbers of processors. We also provide `sync_rmi`s for completeness, but their use is discouraged. The RTS guarantees that requests from a location to another location are executed in order of invocation at the source location.

The RTS provides RMI versions of common aggregate operations. These primitives come in two flavors: *one-sided*, in which a single requesting location invokes the execution of a method in all others, eventually receiving a result back, and *collective*, in which all locations participate in the execution of the operation. All the RMI operations, point-to-point, single-sided, and collective, are defined within communication groups, thus enabling nested parallelism. Collective operations have the same semantics as the traditional MPI collective operations. The provided operations include broadcast, reduce, and fence. The fence operation, called `rmi_fence`, when completed, guarantees that no pending RMIs are still executing in the group where it is called. This is essential for guaranteeing correctness of phases of computations that have to be completed before the next one can start.

The RTS provides some optimizations to use bandwidth and reduce overhead. The major techniques used are *aggregation*, that packs multiple requests to a given location into a single message, and *combining*, that supports the repetitive execution of the same method in a given location without incurring a large overhead for object construction and function calls. Memory management and the number of messages

aggregated are managed by the RTS adaptively according to the application needs.

Another RTS component, the *executor*, has the role of executing task graphs corresponding to `pAlgorithms`. The executor identifies sets of independent tasks to be executed, and schedules them according to the customizable scheduler module. From its perspective, the executor treats incoming RMI requests and algorithmic tasks as *RTS tasks*. Tasks can be assigned to execution threads and they are considered independent.

CHAPTER IV

PARALLEL CONTAINER

Data structures are essential building blocks of any generic programming library. Sequential libraries like STL [49], Leda[44], BGL [60], and MTL [61, 28], provide to the user a collection of data structures and algorithms. For simple regular data structures such as arrays, vectors, and lists, the implementation may be relatively straightforward. More complicated data structures such as matrices (MTL) or graphs (BGL) require the developer to consider a modular design with different functional units that allows customization of different aspects of the data structure to improve the performance of the algorithms. For example, the users of a matrix may want dense or sparse storage and the layout in memory to be row or column oriented.

In a multiprocessor environment, the complexity of a data structure increases due to a number of challenges which are not present in sequential computing. For example, there are issues related to data management such as partitioning, distribution, communication, synchronization, load balancing, and thread safety that have to be considered. To minimize the user's effort in dealing with all these factors, we have developed the *STAPL Parallel Container Framework (PCF)* which consists of a set of formally defined concepts and a methodology for developing generic parallel containers starting from sequential, STL-like containers. Users, by implementing the appropriate interfaces, can assemble with minimal effort a data structure that will provide methods to build and access a distributed collection of elements.

A. pContainer Requirements

Design requirements of the STAPL `pContainers` developed within the PCF include: scalable performance, a clearly specified memory consistency model, a shared object

view, thread safety, composition, and adaptivity.

- **Scalable performance.** `pContainers` must provide scalable performance on shared and/or distributed memory systems. The performance of the `pContainer` methods must achieve the best known parallel complexity. This is obtained by efficient algorithms coupled with non-replicated, distributed data structures that allow a degree of concurrent access proportional to the degree of desired parallelism, e.g., the number of threads.

- **Thread safety and memory consistency model.** When needed, the `pContainer` must be able to provide thread safe behavior and it must respect a well specified memory consistency model as discussed in Chapter VI and Chapter VII, respectively. When a level of a `pContainer` is distributed across shared memory where multiple threads can access and modify it, the PCF must ensure thread safety.

- **Shared object view.** Each `pContainer` instance is globally addressable, i.e., it provides a shared memory address space (Chapter V, Section C). Individual `pContainer` elements can be accessed from any computation thread independent of their physical location. This supports ease of programming, allowing programmers to ignore the distributed aspects of the container if they so desire.

- **Composition.** The capability to compose `pContainers` (i.e., build `pContainers` of `pContainers`) provides a natural way to express and exploit nested parallelism while preserving locality. New `pContainers` can be created by composing existing `pContainers`, e.g., a `pVector` of `pLists`, which would be one way to implement an adjacency list representation of a graph. This feature is not supported by other general purpose parallel libraries. `pContainer` composition is

discussed in Section C.

- **Adaptivity.** A design requirement of the STAPL `pContainer` is that it can easily be adapted to the data, the computation and the system. For example, different storage options can be used for dense or sparse matrices or graphs or the data distribution may be modified during program execution if access patterns change.

B. pContainer Definition

A STAPL *pContainer* is a distributed data structure that holds a finite *collection of elements* $\mathcal{C}$, each with a unique global identifier (GID), their associated storage $\mathcal{S}$, and an interface $\mathcal{O}$ (methods or operations) that can be applied to the collection. The interface $\mathcal{O}$ specifies an Abstract Data Type (ADT), and typically includes methods to read, write, insert or delete elements and methods that are specific to the individual container (e.g., `splice` for a `pList` or `out_degree` for a `pGraph` vertex).

The `pContainer` also includes meta information supporting data distribution: a *domain* $\mathcal{D}$, that is the union of the GIDs of the container's elements, and a mapping $\mathcal{F}$ from the container's domain to the storage. To support parallel use in a distributed setting, the collection $\mathcal{C}$ and the domain $\mathcal{D}$ are partitioned in a manner that is aligned with the storage of the container's elements. The sets $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{S}$ are isomorphic. For each element of the collection $\mathcal{C}$ there is a unique `GID` in the domain $\mathcal{D}$ and a unique memory storage in $\mathcal{S}$.

Definition 1. A `pContainer` is defined as:

$$pC \stackrel{def}{=} (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}, \mathcal{S}) \tag{4.1}$$

The tuple $(\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O})$ is known as the *native* `pView` of the `pContainer`. As in-

troduced in Chapter III, Section A and described in more detail in [14], STAPL `pViews` generalize the iterator concept and enable parallelism by providing random access to collections of their elements. In `pViews`, the partition of $\mathcal{D}$ can be dynamically controlled and depends on the needs of the algorithm (e.g., a column-based partition of a `pMatrix` for an algorithm that processes the matrix by columns) and the desired degree of parallelism (e.g., one partition for each core). The native `pView` associated with a `pContainer` is a special view in which the partitioned domain $\mathcal{D}$ is aligned with the distribution of the container's data. Performance is enhanced for algorithms that can use native `pViews`.

We formally introduce now the concepts that were briefly introduced in this section: *Global Identifier (*`GID`*), Domain*, and *Partition*. Since data structures are collections of elements we start first by introducing some elementary theory about sets and then properly introduce the PCF concepts used to implement a distributed collection of elements.

1. Set Theory Definitions

We often refer in this document to the notion of a collection of elements or identifiers and discuss different properties associated with them. In this section we introduce the notions of *set* and *ordered set* to facilitate subsequent discussions about containers.

Definition 2. A *set* is a collection S, of distinct objects $\{e_0, e_1, ..., e_i...\}$, which are called the *elements* of S. If e is an element (or member) of S, we write $e \in S$.

For example, we can define sets of integer numbers $S_I = \{2, 7, 5\}$, strings $S_s = \{'Red', 'Blue', 'Black'\}$ or memory addresses $S_A = \{0xa0, 0xa4, 0xa8\}$.

Definition 3. For a given type T, the *universe* of type T ($Universe(T)$) denotes the maximal set of distinct elements that are of type T.

Definition 4. The *cardinality* $|S|$ of a set S is the number of elements of S. If the cardinality of a set is finite, then we call it a *finite set*. If the cardinality is infinite, then we call it an *infinite set*. We use $\{\}$ or ϕ to denote an *empty set*, and its cardinality is zero.

We introduce the notion of relations between elements of a set to express ordering and traversals of the elements of a set:

Definition 5. A set of elements S, is called a *partial ordered set* if there is a binary relation R defined on it that is reflexive ($\forall a \in R,\ aRa$), antisymmetric (if aRb and bRa then $a = b$) and transitive (if aRb and bRc then aRc). If the relation R is antisymmetric, transitive and total ($\forall a, b \in S, aRb\ or\ bRa$), than it is a *total ordered set*.

For example, for $S_i = \{2, 7, 5\}$, we can define a total ordered set by associating correspondingly the $\leq$ relation on integer numbers. For $S_s = \{"Red", "Blue", "Black"\}$ we can define a total order relation using lexicographical compare for strings.

For a *finite total ordered set* we can define the following useful notions:

Definition 6. The *first, last, next, prev element* and *unique enumeration (linearization)* imposed by a total order relation, R, on a finite set S, are defined as follows:

1. The first element of the set, $first : S \to S,\ first(S) = x,\ such\ that\ \forall z \in S,\ xRz$
2. The last element of the set, $last : S \to S,\ last(S) = y,\ such\ that\ \forall z \in S,\ zRy$
3. The next element of an element, $next : S \to S,\ next(x) = y,\ such\ that\ xRy\ and\ there\ is\ no\ other\ z \in S,\ xRz\ and\ xRy$
4. The previous element of an element, $prev : S \to S,\ prev(x) = y,\ such\ that\ xRy\ and\ there\ is\ no\ other\ z \in S,\ yRz\ and\ zRx$

5. A unique enumeration (*linearization*) imposed by R. Starting from the first element e_0 there is a unique enumeration $e_0\ R\ e_1\ R...R\ e_{n-1}$ that contains all elements in S.

Linearizations are used to specify various traversals of the elements in a set. As described in the following sections linearizations will be used by a `pContainer` to specify various traversals of its elements.

2. pContainer Domain

For each `pContainer` element there is a unique identifier or `GID` associated with it.

Definition 7. A domain is a set of `GIDs`.

Throughout PCF we use various types of domains with different properties that we properly introduce here. For a static container, the number of elements is fixed at the construction time. For dynamic containers, the number of elements, in general, is bounded only by the amount of available storage, and so in principle can be infinite. We distinguish then, for dynamic `pContainers`, the `pContainer`'s *domain* which is the universe of all `GIDs` that will identify its elements and the `pContainer`'s *domain instance* which is the set of `GIDs` that identifies the current elements of the pContainer. Hence, the dynamic `pContainer` domain is infinite while the domain instance is always a finite domain.

3. Ordered Domain

The domain as defined in Section 2 does not specify any requirement on the order of the `GIDs` it contains. To specify an order we introduce the notion of *ordered domain.*

Definition 8. A domain with a binary relation R on its elements that is reflexive, antisymmetric and transitive is called a *partial ordered domain.* If the relation R is

antisymmetric, transitive and total than we have a *total ordered domain*. We will use the notation $OD \stackrel{def}{=} (D, R)$ to represent an ordered domain.

In this document, whenever we refer to ordered domains they are total ordered domains, unless otherwise specified. An often used relation is the total order $\leq$ on integers. For example for a domain $D = \{2, 1, 3\}$ we can have $(D, \leq) = \{1 \leq 2 \leq 3\}$ as an ordered domain. An ordered `pContainer` domain is often used in the PCF to specify how the `pContainer`'s data is organized in memory and to specify a linear traversal order for the elements.

Example of Domains used by `pContainers`

The domain is an important concept that users and developers can interact with while using the PCF. Different `pContainers` require specific domains and specific implementations to guarantee certain properties, such as the time to perform an operation. In the following, we describe the domains that are provided and used by our framework:

Finite Domains:

1. An enumeration of individual elements.
   Examples: $D = \{1, 3, 2\}$ or $D = \{a, c, b\}$
   Enumerations are considered unordered. A straightforward order can be implied for these domains, by considering the order in which they are specified. Other orders such as $\leq$ or $\geq$ are possible;
2. Range: a sub-domain of a bigger domain. We will represent a range by a first element, a last element and a *next* operator that allows us to enumerate all its `GIDs`.

Examples: $1DRange = \{ [0..2), \leq\} = \{0,1\}$ as a sub-domain of the integers domain.

$2DRange = [(0,0),(2,2))$. A 2DRange can be ordered row-wise or column-wise to obtain a total ordered domain. $2dRange_row = \{(0,0),(0,1),(1,0),(1,1)\}$ and $2dRange_column = \{(0,0),(1,0),(0,1),(1,1)\}$

Infinite Domains:

1. Open ordered domains for associative containers,

   $\{([key1, key2), lexicographicalcompare(\leq_s)\}$

   Example: Strings domain defined on a set of characters, between "a" and "c", $\{["a","c"), lexicographical\ order)\}$, contains an infinity of elements (e.g., $\{"a","aa"\ ,"aaa","ab","aba",...\}$).

Domains defined as compositions of existing domains:

1. Cartesian products over ordered domains,

   $OrderedD = (\ [(D_1,\leq_1),(D_2,\leq_2)],\ R = lexicographical\ order$ based on $\leq_1, \leq_2$).

   Example: $OrderedD = (\ [(0..10,\leq),(0..10,\leq)],\ sR = lexicographical$ $order\ over\ integer\ pairs$. The gids are pairs (i,j) and $(i,j) \leq (p,q)$ if $i < p$ or if $i == p$ and $j \leq q$

2. Set operations on exiting ordered domains, $OD_3 = OD_1\ op\ OD_2$ where $op = \{\cap,\ \cup, -\}$ and the restrictions that OD_1 and OD_2 are defined over the same type of `GIDs`. If OD_1 and OD_2 are ordered according to the same relation R, then OD_3 is also ordered according to R.

3. Filtered domain:

   $D = (D_1, filter_function) = \{y | y \in D1\ and\ filter_function(y) = true\}$

Ex: $D = (([0..10], \leq), f = every_second_element)$ defines every second element in the enumeration of the original domain according to $\leq$.

For the ordered domains we mentioned that the last *gid* is not part of the domain. This last element is a convention (e.g., predefined value type) that has the property that all other elements of the domain are in relation with it. For integral types this value is predefined by our framework but for other *gid* types the users will have to define the last element as part of the domain interface.

4. Partition

The `pContainer` manages a distributed storage where individual locations store a subset of its elements. The `pContainer` uses a partition to group its elements in individual units of storage. The partition specifies a decomposition of a domain into sub-domains and how to map an individual `GID` to the sub-domain that contains it. Later, in Chapter V, Section C.4, we will describe additional functionality that partitions provide.

Definition 9. A partition $P = \{D_0, D_1, .., D_{n-1}\}$ of a domain D is a collection of sub-domains of D, such that:

1. $D = D_0 \cup D_1 \cup ... \cup D_{n-1}$ (the union of the elements of all sub-domains is the set of elements of the original domain)
2. $Di \cap Dj = \emptyset, \forall\ i, j,\ 0 <= i, j < n, i \neq j$ (sub-domains are disjoint)
3. The *sub-domain set*, $\{D_0, D_1, .., D_{n-1}\}$ is ordered by the relation $\leq$ over domain indices ($D_0 \leq D_1 \leq, .., \leq D_{n-1}$).

Partition properties (interface):

1. Partition a domain D into a set of sub-domains according to Definition 9.

2. Specify the cardinality of the sub-domain set (e.g., how many sub-domains the partition defines).
3. Given an element, identify the sub-domain to which it is associated.

5. Ordered Partition of Total Ordered Domains

For a STAPL `pContainer` that is the parallel equivalent of a STL container we need to support a total order among its elements in order to provide a linearization of its data. For this reason we describe in this section the notion of an *ordered partition* that is used by the `pContainer` to provide a linearization of its data.

Definition 10. A partition of a total ordered domain $OD(D, R)$, $P = \{(D_0, R), ..., (D_{n-1}, R)\}$ is an *Ordered Partition (OP)* if $\forall D_i \in P,\ (D_i, R)$ are total ordered domains and there is a relation R_D across the domains of P, such that, $\forall x, y \in D$ and $x\ R\ y$ then either:

1. x and y belong to the same sub-domain D_i and xRy OR
2. x and y belong to different sub-domains D_i and D_j respectively, and $D_i R_D D_j$

We are using the notation (P, R, R_D) to denote an ordered partition. An ordered partition of a total ordered domain preserves the order among elements from the domain to the sub-domains.

Definition 11. Given a totally ordered domain $OD \equiv (D,\ R)$ we define the *split* as a blocked partition $P = \{D_0, ..., D_{n-1}\}$ of D such that D_0 contains the first $|D_0|$ elements from D according to R, D_1 the next $|D_1|$ elements from D, etc.

The split is a blocking of the unique enumeration of a total ordered domain and it is an important way of defining a partition that preserves the relation between elements in the original domain. For example, for the totally ordered domain

$([0,10), \leq)$ we can have $P = \{D_0 = ([0,5), \leq), D_1 = ([5,10), \leq)\}$ and $R_D \equiv D_0 \leq D_1$ as a possible split.

A split of a total ordered domain and the relation $R_D \equiv D_0 \leq ... \leq D_{n-1}$ induces an ordered partition. The split can be seen as a top-down procedure of specifying an ordered partition for an ordered domain. Next we show how to specify an ordered domain corresponding to a set of ordered domains.

Lemma 1. Given a set of ordered domains $P = \{(D_0, R), ..., (D_{n-1}, R)\}$, and a total relation across the domains, R_D, then P and R_D defines an ordered partition of the total ordered domain OD(D,R), where $D = \bigcup_{i=0..n-1} D_i$ and R is defined as:

1. if x and y belong to the same domain D_i then $xRy \equiv xRy$
2. if x and y belong to different domains D_i and D_j then $xRy \equiv D_i R_D D_j$

The proof for Lemma 1 is immediate from the definition of the ordered domains. To exemplify how Lemma 1 is useful let us consider two totally ordered domains $D_0 = \{"Red", "Blue"\}$, $R_0 = "Red" \leq "Blue"$ and $D_1 = \{"Black", "White"\}$, $R_1 = "Black" \leq "White"$ and $R_D = D_1 \leq D_0$. Then $\{(D_0, R_0), (D_1, R_1)\}, R_D$ is an ordered partition of the totally ordered domain

$D = \{"Red", "Blue", "Black", "White"\}$, $R = "Red" \leq "Blue" \leq "Black" \leq "White"$

C. pContainer Composability

There are many common data structures that are naturally described as compositions of existing structures. For example, a `pVector` of `pList`s provides a natural adjacency list representation of a graph. To enable the construction and use of such data structures, we require that the composition of `pContainers` be a `pContainer`, i.e., that `pContainers` are closed under composition.

An important feature of composed `pContainers` is that they support hierarchical parallelism in a natural way – each level of the nested parallel constructs can work on a corresponding level of the `pContainer` hierarchy. If well matched by the machine hierarchy, this can preserve existing locality and improve scalability.

In this section, we examine the properties of the composed `pContainer`, and analyze the relations between the `pContainers` that are composed and the final composed `pContainer`. As a simple example, consider a `pArray` of `pArrays`. The following code is used to declare the composed `pArray` and correspondingly resize each of the nested `pArrays` to obtain the hierarchy depicted in Figure 3 :

```
p_array<p_array<int,...>,...> pApA(3);
pApA[0].resize(2); pApA[1].resize(3); pApA[2].resize(4);
```

The composed data structure obtained using composition can be thought of as a single data structure whose domain, interface, storage, etc., are compositions of the corresponding modules at the two levels of the hierarchy. For example, the domains of the nested `pArrays` depicted in Figure 3 are $\mathcal{D}_{10} = \{0, 1\}$, $\mathcal{D}_{11} = \{0, 1, 2\}$ and $\mathcal{D}_{12} = \{0, 1, 2, 3\}$ and the domain of the outer `pArray` is $\mathcal{D}_0 = \{0, 1, 2\}$. The composed data structure however can be viewed as a data structure by itself where the domain consists of the following `GIDs`: $\mathcal{D}_{composed} = \{(0, 0), (0, 1), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2), (2, 3)\}$. We formally describe the domain of the composed `pContainer` as the

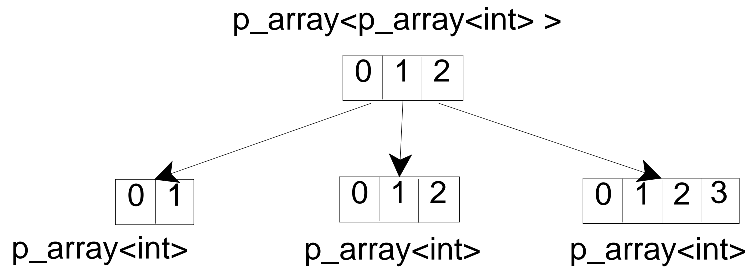


Fig. 3. Composed `pArray` of `pArrays`.

union of the cross products of each element in $\mathcal{D}$ with the corresponding domains of the nested `pContainers`.

$$\begin{aligned}\mathcal{D}_{composed} &= \bigcup_{i \in \mathcal{D}} (\{\mathcal{D}_0[i]\} \times \mathcal{D}_{1i}) \\ &= (\{0\} \times \mathcal{D}_{10}) \cup (\{1\} \times \mathcal{D}_{11}) \cup (\{2\} \times \mathcal{D}_{12}) \\ &= \{(0,0),(0,1),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2),(2,3)\} \quad (4.2)\end{aligned}$$

The corresponding interface of the composed container can be thought of as the composition of the interfaces of the `pArrays` that are composed. For example accessing the element corresponding to `GID` (1,0) in the composed container can be done with the following invocation: `pApA.get_element(1).get_element(0)`. The method of the composed `pArray` is naturally an application in series of methods at both levels of the hierarchy.

We define the *height* of a composed `pContainer` as the number of `pContainers` present in the composed type. For a `pContainer` whose element is a non `pContainer` type, the height is defined as one. For the composed `pArray` example in Figure 3, the height of the composition is two. The composition can be done an arbitrary

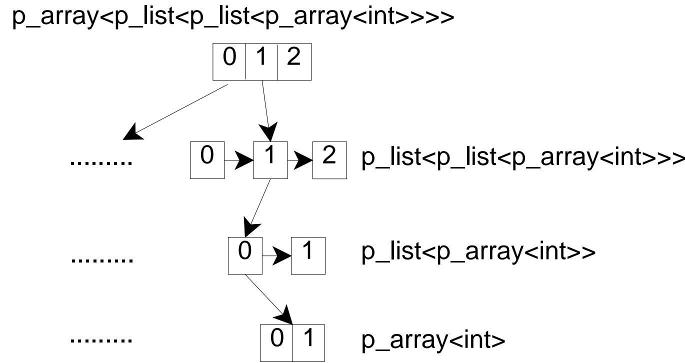


Fig. 4. Composed `pContainers`.

number of times and include various types of `pContainers`. For example, composing a `pArray`$<$`pList`$<$`T`$>>$ with a `pList`$<$`pArray`$<$`T`$>>$ results in the composed `pContainer` `pArray` $<$`pList` $<$`pList`$<$`pArray`$<$`T`$>>>>$ with height four. In Figure 4, we include the hierarchical organization of a possible instance of such a composed `pContainer`.

In the remainder of this section we formalize the process of composing two arbitrary `pContainers` that themselves can be the result of a previous composition.

Definition 12. Let $pC_1 = (\mathcal{C}_1, \mathcal{D}_1, \mathcal{F}_1, \mathcal{O}_1, \mathcal{S}_1)$ and $pC_2 = (\mathcal{C}_2, \mathcal{D}_2, \mathcal{F}_2, \mathcal{O}_2, \mathcal{S}_2)$ be two composed `pContainers` of height H_1 and H_2, respectively. The composed `pContainer` $pC = pC_1 \circ pC_2$ is of height $H = H_1 + H_2$. In pC, each element of pC_1, $pC_1[i]$, $i \in \mathcal{D}_1$, is an instance of pC_2, called $pC_{2i} = (\mathcal{C}_{2i}, \mathcal{D}_{2i}, \mathcal{F}_{2i}, \mathcal{O}_{2i}, \mathcal{S}_{2i})$.

Each component of pC is derived appropriately from the corresponding components of pC_1 and pC_2. For example, in the special case when all the mapping functions

$\mathcal{F}_{2i}$ and operations $\mathcal{O}_{2i}$ are the same, we have

$$\begin{aligned}\mathcal{D} &= \bigcup_{i \in \mathcal{D}_1} (\{\mathcal{D}_1[i]\} \times \mathcal{D}_{2i}) \\ \mathcal{F} &= (\mathcal{F}_1, \mathcal{F}_2) \\ \mathcal{O} &= (\mathcal{O}_1, \mathcal{O}_2)\end{aligned}$$

where, for $(x, y) \in \mathcal{D}$, $\mathcal{F}(x, y) = (\mathcal{F}_1, \mathcal{F}_2)(x, y) = (\mathcal{F}_1(x), \mathcal{F}_2(y))$. The components $\mathcal{C}$ and $\mathcal{S}$ are isomorphic to $\mathcal{D}$ and they are defined similarly (for each element of the collection, there is a unique `GID` and a unique storage). With this formalism, arbitrarily deep hierarchies can be defined by recursively composing `pContainers`.

Given a composed `pContainer` $PC = (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}, \mathcal{S})$, with a hierarchy of height H, we need a mapping function to access a `pContainer` at level $1 \leq h \leq H$. This function is a sub-sequence (prefix) of the tuple of functions $\mathcal{F}$, $\mathcal{F}^h(x_1, \ldots, x_h) = (\mathcal{F}_1(x_1), \mathcal{F}_2(x_2), \ldots, \mathcal{F}_h(x_h))$. The operations available at level h are $\mathcal{O}_h$. The `pContainer` composition is made without loss of information, preserving the meta information of its components in the same hierarchical manner. For example, if two distributed `pContainers` are composed, then the distribution information of the initial `pContainers` will be naturally preserved in the new `pContainer`. In Figure 4 we show an example of a hierarchy with four levels, where levels are counted from top down. Accessing an element at level 3 of the hierarchy is possible with the following interface:

```
pList<pArray<int>>& plpa = pc.get_element(1).get_element(1);
```

The `pList<pArray<int>>` reference obtained with the composed method invocation above is possible using the mapping functions of the `pContainers` at level 1 and 2 of the hierarchy.

A possible research direction for `pContainer` composition is to allow for (static)

specialization of the mapping functions if machine information is provided. For example, if the lower (bottom) level of the composed `pContainer` is distributed across a single shared memory node, then its mapping $\mathcal{F}$ can be specialized for this environment, e.g., some methods may turn into empty function calls.

CHAPTER V

PARALLEL CONTAINER FRAMEWORK

One of the main objectives of the STAPL *Parallel Container Framework (PCF)* is to simplify the process of developing generic parallel containers as defined in Chapter IV. The PCF is a collection of classes that can be used to construct new `pContainers` through inheritance and specializations 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. This allows 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. Moreover, the PCF facilitates interoperability by enabling the use of parallel or sequential containers from other libraries, e.g., MTL [28], BGL [30] or TBB [36].

STAPL provides a library of `pContainers` constructed using the PCF. These include counterparts of STL containers (e.g., `pVector`, `pList` [65], and associative containers [64] such as `pSet`, `pMap`, `pHashMap`, `pMultiSet`, `pMultiMap`) and additional containers such as `pArray` [63], `pMatrix` [15], and `pGraph`.

Novice programmers can immediately use the available data structures with their default settings. More sophisticated parallel programmers can customize or extend the default behavior to further improve the performance of their applications. If desired, this customization can be modified by the programmer for every `pContainer` instance.

The STAPL Parallel Container Framework has been designed in a modular fashion. This allows developers to implement new `pContainers` or customize existing

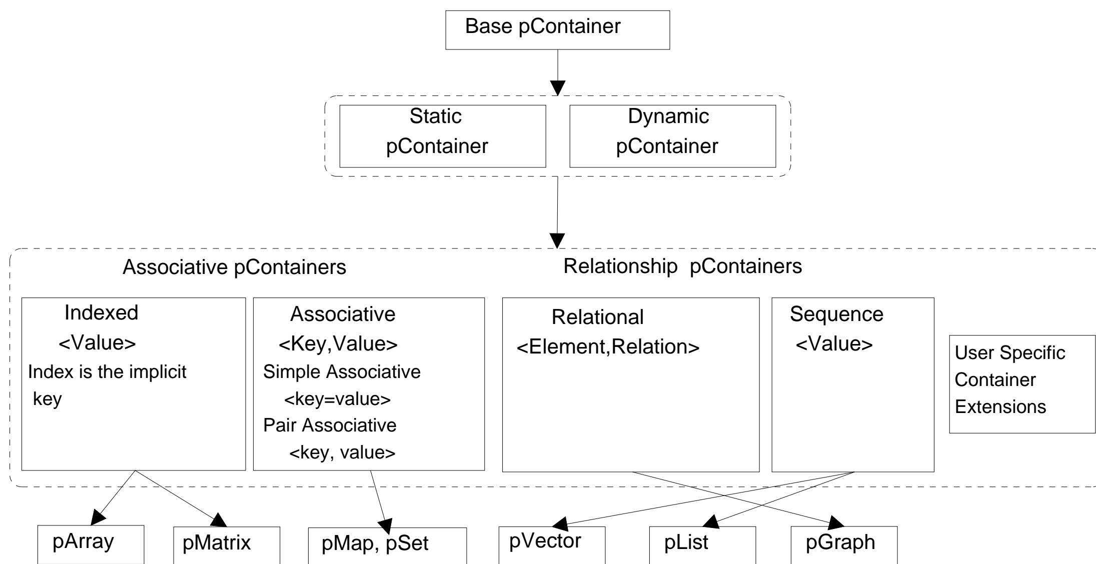


Fig. 5. PCF design.

ones by implementing the appropriate set of interfaces.

A. pContainer Framework 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 5 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` derive from `p_container_base` class. This class is in charge of storing the data using a `location-manager` and data distribution information using a `data-distribution-manager`. It provides a simple interface to initialize the `pContainer` based on the traits class provided as a template argument, and a domain and partition instance. The complete interface is described in Section D.1.

The remaining classes in the PCF provide additional interfaces and requirements. First, `static` and `dynamic pContainers` are classes to indicate if elements can be added to or removed from the `pContainer`. The property that the number of elements is fixed allows for more efficient implementations of domains, partitions and `pViews` to be used. The interfaces are discussed in Sections D.2 and D.3.

The next discrimination is between *associative* (Section D.5) and *relational* (Section D.6) `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` (Section D.4). 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), and lists where the relations between elements are implicit.

All classes of the PCF have well defined interfaces as described in Section D and 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 data distribution, to improve the performance of their data structures. The `pContainer` customization using traits is further discussed in Section H.

B. pContainer Interfaces

STAPL `pContainers` are commonly extensions of existing sequential data structures and they often support the same interface as the sequential counterpart. However it

is often the case that a `pContainer` provides an extended interface to better optimize for parallelism. Relative to a sequential data structure the methods provided by a `pContainer` have more complex semantics that need to be understood correctly to reason about the correctness of an application that is using them. We distinguish the following categories of methods:

Collective methods: These methods have to be invoked on all locations where there is a `pContainer` representative in an Single Program Multiple Data (SPMD) fashion. Examples of such methods are constructors, destructors and methods that redistribute the `pContainer`'s data. A location cannot invoke another method until the pending method completes.

Element-wise methods: The methods in this category operate on an individual `pContainer` element and include methods such as `get_element`, `set_element`, `split_phase_get_element`, insert, erase, etc. The element-wise methods are split in the following three categories based on the guarantees we provide about their completions:

- Synchronous methods: have a return type and guarantee that the method is executed and the result available when they return. A thread cannot invoke another method until the pending method completes.
- Asynchronous methods: have no return value and return immediately to the calling thread. The execution will complete subsequently.
- Split phase methods: execution is similar to Charm++ [41], X10 [18]. The return type of a split phase method is a *future* that allocates space for the result. The invocation returns immediately to the user. When the `get` method is invoked on the future, the calling thread will return immediately if the result is available or block until the result arrives. This type of method may benefit an

application if additional work can be performed while waiting for the result and is provided as an alternative to synchronous methods. If one of the phases of a split method has been invoked, then the thread cannot invoke another method until that phase completes.

Global property methods: These methods include operations returning global properties about the data structure such as `size()` or `empty()`. These are not collective, synchronous operations but they do require partial information from all locations where there is a `pContainer` representative. A thread cannot invoke another method until the pending method completes.

New methods facilitating parallel use: For certain `pContainers` we extend their interfaces with methods that improve their efficiency in a parallel environment. For example, the `insert_anywhere` method of a `pList` adds an element to the `pContainer` to an unspecified position. The `pContainer`, in such situations may optimize the insertion to improve load balance and speed.

The STAPL runtime provides a fence construct that when invoked guarantees that all pending `pContainer` methods are completed. Each category of methods described above provides different guarantees about the completion of the invocations and ordering among them when invoked concurrently from multiple threads. The complete specification of the guarantees provided to the user makes the `pContainer` memory consistency model (MCM) discussed in detail in Chapter VII.

The information about the particular semantics of an element-wise method is embedded in the return type. For example, the parallel array data structure available in STAPL (described in more detail in Chapter IX) provides a simple interface to access elements based on their indices. The interface contains:

```
void set_element(gid, value);
```

```
value get_element(gid);
pc_future<value> split_phase_get_element(gid);
```

Based on the signature a user will be aware that `set_element` is implemented asynchronously (b/c no return type), `get_element` is synchronous (returns a value) and `split_phase_get_element` is implemented using a split phase execution. More details on the semantics of the three methods and guarantees about their completion are included in Chapter VII where we discuss the memory consistency model for `pContainers`. The performance trade offs between these three categories of methods are discussed for various `pContainers` in Chapters IX, X, XI and XII.

C. Shared Object View Implementation

As depicted in Figure 6, a `pContainer` stores its elements in a non-replicated fashion in a distributed collection of *base containers* (`bContainers`, Section C.1). `pContainers` can be constructed from any existing container, sequential or parallel, so long as it can support the required interface as specified in Section C.1. The `pContainers` currently provided in STAPL use the corresponding STL containers (e.g., the STAPL `pVector` uses the STL vector), containers from other sequential libraries (e.g., MTL for matrices), containers available in libraries developed for multicore (e.g., TBB concurrent containers), or other `pContainers`. This flexibility allows for code reuse and supports interoperability with other libraries.

The `pContainer` provides a shared object view that enables programmers to ignore the distributed aspects of the container if they so desire. When a hardware mechanism is not available, the shared object view is provided by a software address resolution mechanism that first identifies the `bContainer` containing the required element and then invokes the `bContainer` methods in an appropriate manner to

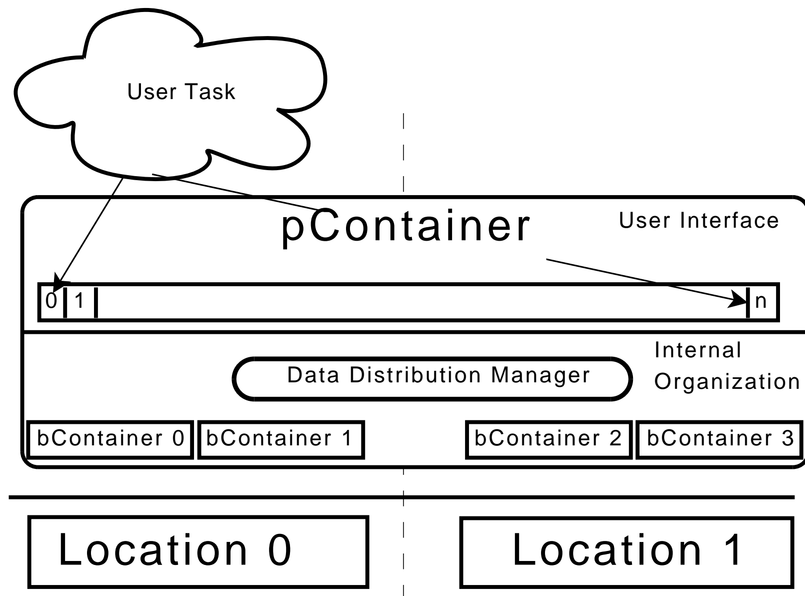


Fig. 6. Shared object view. STAPL `pContainer` provides a shared object view to the user. Internally it distributes the data across available locations and uses a distribution manager to find where individual elements are stored.

perform the desired operation.

The elements of a `pContainer` are stored in non-replicated fashion in a distributed collection of `bContainers`. An important function of the PCF is to provide a shared object view that relieves the programmer from managing and dealing with the distribution explicitly, unless he desires to do so. In this section, we describe how this is done.

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 indices for `pArrays`, keys for `pMaps`, and vertex identifiers for `pGraphs`.

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 introduce the PCF components involved in the address translation. The set of `GIDs` of the elements of a `pContainer` is the `pContainer` domain ($\mathcal{D}$). A domain is partitioned into a set of non-intersecting *sub-domains* by a partition class, itself a distributed object that provides the map $\mathcal{F}$ 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 `bContainer`. In general, there can be multiple `bContainers` 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 concept called `partition-mapper` is used to map a sub-domain (and its corresponding `bContainer`) to the location where it resides, and a `location-manager` to manage the `bContainers` of a `pContainer` mapped to a given location.

We now describe how a `pContainer` method is executed using the above concepts. In Figure 7 we show a flowchart of the address resolution procedure. Given the unique `GID` identifying a `pContainer` element, the `pContainer`'s partition is queried about the sub-domain associated with the requested `GID`. If the `bContainer` (specified by a `bContainer` 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` receives information about the location where the `bContainer` resides (`LID`); if the operation is not local, the method is re-evaluated on that location, otherwise the `location-manager` provides the proper `bContainer` address and the operation is performed.

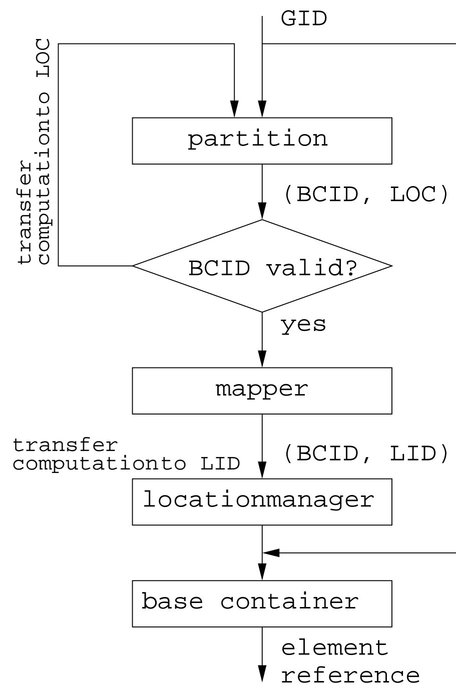


Fig. 7. `pContainer` address resolution. `pContainer` modules for performing address resolution to find the element reference corresponding to a given `GID`.

In dynamic `pContainers`, the domain may change during execution through the insertion or deletion of elements. To properly update the domain and partition information, `pContainer` operations are routed to the partition. In general, each method of the `pContainer` interface has two corresponding methods in the partition class: the *where* method that returns information about the sub-domain that may include the specified `GID`, and the *execution* method that actually performs the operation and updates the partition information if needed. To allow for work migration, the *where* method can provide an incomplete answer if the sub-domain information is not available on the location where the *where* method is invoked. In this case, the answer is the identifier of a location that *may* know the sub-domain information. This mechanism

is referred to as *method forwarding* and allows the request to be migrated instead of fetching remote information for the requester. More details on partition functionality and method forwarding are included in Sections C.4 and C.6. Experimental results showing the benefits of method forwarding are included in Chapter XI, Section F.2.

In static `pContainers`, i.e., containers that do not support the addition and deletion of elements, the domain does not change during execution. In this case, it may be possible to optimize the address translation mechanism. In particular, if the mapping from `GID` to sub-domain (and hence to `bContainer`) has a closed form solution, then address translation is immediate and forwarding is not needed.

Next we introduce the specification and the interfaces for all modules briefly described in this section: *Base Container, Location Manager, Domain, Partition, Partition Mapper* and *Data Distribution Manager*.

1. Base Container Interface

The `pContainer` allocates a base container (`bContainer`) for every sub-domain defined by the partition to store the data corresponding to the sub-domain. The `bContainer` concept specifies a minimal interface that allows for any existing container sequential or parallel to be used as storage for the parallel container. We can not directly use the sequential container because different non standard implementations provide different interfaces for similar functionality. With the base container concept we unify sequential containers such that they can be integrated with the PCF serving as a bridge between existing data structures and PCF. The minimal interface required for a `bContainer` is included in Table III.

Table III.: Base container interface.

Define Types	**Description**
value_type	The element type stored in the `bContainer`.
reference	The element reference type. Do not assume that T is the reference type.
gid_type	The gid type associated with the elements of the `bContainer`.
bcid_type	The `bContainer` identifier type.
domain_type	The domain type associated with the `bContainer`.
Method	**Description**
constructor (domain_type*, const bcid_type&)	Construct a `bContainer` with the given `bContainer` id and a given domain
destructor(void)	Deallocate the memory space occupied by the `bContainer`.
size_t size() const	Returns the number of elements in the `bContainer`.
bool empty() const	Returns true if the `bContainers` has zero elements and false otherwise.
void clear()	Deallocate the space taken by the `bContainer` elements. After clear(), size() returns zero and empty() returns `true`
bcid_type get_bcid()	Return the `bContainer` identifier
void define_type(typer &)	Define type for packing `bContainer`'s data
std::pair < size_t,size_t > memory_size(void) const	Return the memory size used by `bContainer`. The first member of the pair is the memory used by data and the second argument represent memory used by metadata

The main constructor used to instantiate a `bContainer` takes as input a reference to the associated sub domain from the partition and a `BCID`. Additional methods are typical for containers and include the `size()` and `empty()`, methods to report the memory size and a method for serializing the data of a `bContainer`.

2. Location Manager Interface

In Definition 4.1 it is shown that a `pContainer` $pC(\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}, \mathcal{S})$, stores elements identified by the `GIDs` encompassed in the domain D, into a storage S that is distributed across available locations $L = \{L_0, L_1, ...L_{p-1}\}$. The `pContainer` storage consists of a collection of `bContainers` stored in a distributed fashion across the available locations.

A location may store a sub-set of the `bContainers` of a `pContainer`. The `pContainer` employs within each location a *Location Manager* to maintain the collection of `bContainers`. The complete interface is presented in Table IV. It includes methods to add and delete `bContainers` and methods to access individual `bContainers` based on their global unique `BCID`. The `bContainers` are allocated either by the `pContainer` or provided from outside when external storage is used. The location manager may use different optimizations for storing `bContainers`. For example different memory managers may be used to allocate the space required by the `bContainers`.

Table IV.: Location manager interface.

Define Type	**Description**
`bcontainer_type`	`bContainer` used to store `pContainer`'s data.
`bcid_type`	`bContainer` identifier type.
`iterator`	iterator type used when iterating over `bContainers`.
Method	**Description**
`default constructor`	Initialize an empty location manager
`void add_bcontainer(const bcid_type&, bContainer_type*)`	Add a new `bContainer` to the `pContainer`.
`size_t size() const`	Returns the number of `bContainers` on the current location

Table IV continued

Method	Description
`iterator begin()`	Iterator pointing to the first local `bContainer`.
`iterator end()`	Iterator pointing one past the last local `bContainer`.
`bcontainer_type* get_bcontainer(const bcid_type&)`	Returns a pointer to a given `bContainer` specified as argument
`void clear(void)`	Deletes the `bContainers` and their corresponding memory
`pair<size_t,size_t> memory_size(void) const`	Compute the memory size for location manager meta data and data

3. Domain Interface

Derived from its definition in Chapter IV, Section B.2, a `pContainer` domain specifies the `GID` type of the domain, test if a specific `GID` belongs to the domain, compute its cardinality if finite and provide methods to enumerate its `GIDs`. The framework most commonly uses ordered domains to describe the set of `GIDs` of a `pContainer` and the order among the elements corresponding to the `GIDs`. An ordered domain implementation supports the additional functionality described next and is required to implement the interface from Table V.

1. Specify the *first* and the *last* element of the domain according to a total order R. The first element belongs to the domain but the last element does not belong to the domain.

   $\forall x \in OD, (firstRx)\ and\ (xRlast)\ and\ not(lastRx)$.

   This is a requirement that provides us compatibility with C++ STL, where the end of a range is a convention such that all elements in the range are less than it.

2. Compare two `GIDs` according to R (e.g., $gid_1 R gid_2$ is either true or false).

Table V.: Ordered domain interface.

Define Type	**Description**
gid_type	The global unique identifier type
Method	**Description**
gid_type get_first_gid() const	The first gid of the domain
gid_type get_last_gid() const	The last gid of the domain; a convention with the property that every other gid of the domain will be less than it
bool contains_gid (gid_type _gid) const	Returns true or false depending if the domain is part of the domain
bool compare_less_gids (gid_type, gid_type) const	Compare for 'less then' two `GIDs`
gid_type get_invalid_gid() const	Required to represent NULL/invalid iterators in the `pView`

A *Finite Ordered Domain (FOD)* extends the ordered domain with the following functionality and the interface included in Table VI.

1. Return the number of elements in the domain (e.g., cardinality).
2. The next *gid* of a *gid* defined as $next : FOD \rightarrow FOD,\ next(x) = y,\ such\ that\ xRy\ and\ there\ is\ no\ other\ z \in FOD,\ xRz\ and\ zRy$
3. The previous *gid* of a *gid* defined as $prev : FOD \rightarrow FOD,\ prev(x) = y,\ such\ that\ xRy\ and$
   $there\ is\ no\ other\ z \in FOD,\ yRz\ and\ zRx$
4. The nth *gid* following a *gid* defined as $advance_n : FOD \rightarrow FOD,\ advance_n(x) = y,\ such\ that$

$y = next(...next(next(x))...),\ ntimes$; apply $next$ n times starting with x.

5. An unique enumeration imposed by R. Starting from the first element e_0 there is a unique enumeration $e_0\ R\ e_1\ R...R\ e_{n-1}$ that contains all elements in FOD.
6. An offset of a *gid* within the unique enumeration specified by R which is defined as
$offset : FOD \rightarrow N(natural\,numbers),\ offset(x) = n\,iff\,advance_n(first) = x$.

Table VI.: Finite ordered domain interface.

Define Type	**Description**
gid_type	The global unique identifier type
Method	**Description**
size_t size (void) const	The size of the domain
gid_type get_next_gid (gid_type) const	The next gid of the argument according to the domain ordering relation
gid_type get_prev_gid (gid_type) const	The previous gid of the argument
gid_type advance (gid_type, size_t n) const	The nth gid after the current one
size_t offset (gid_type) const	The offset of the gid in the linearization of the domain

4. Partition Interface

In Chapter IV, Sections B.4 and B.5 we introduced the main functionality of the partition. It specifies a decomposition of a domain into a collection of sub-domains. The partition is one of the main functional modules of a `pContainer` and we envision that this will be the most common mechanism used to customize the behavior of

the `pContainer`. In addition to specifying a collection of sub-domains the partition will provide the information regarding the `BCID` associated with a given `GID`, and will specify the behavior of individual `pContainer` methods.

For a dynamic `pContainer` as described later in Section D.3 the partition will specify the `bContainer` where a new element will be added and the specific actions that need to happen when the element is added to an individual `bContainer`. For this reason, we decided that every `pContainer` method will have two corresponding methods at the partition level describing the `bContainer` where the method will be executed and how it will be executed. In Section D we include details on the additional interface requirements for the partitions of various `pContainer` specializations. When atomicity is provided by the framework the partition will additionally specify the locking modes for each individual method in the `pContainer` interface. This will be discussed in more detail in Chapter VI.

Derived from the partition description and properties introduced in this section we designed the interface included in Table VII.

Table VII.: Partition base interface.

Define Type	**Description**
domain_type	domain type
bcid_type	sub domain identifier type
bcid_type get_info(const GID&) const	Returns the `bContainer` identifier associated with input `GID`.
const bcids_info_type& get_cids_info(void) const	Returns a structure with the information about the order among `BCIDs`.
size_t size() const	Returns the total number of sub domains
void get_sub_domains_sizes(std::vector<size_t>&) const	Return the sizes of the sub domains
void set_domain(domain_type*)	Sets the domain of the partition. The partition initializes its sub domains to reflect a partition of the input domain.

Table VII continued

Method	Description
domain_type* get_domain() const	Get the domain of the partition.
domain_type* get_sub_domain(const bcid_type&)	Get a certain sub domain.
const std::vector<domain_type*>* get_sub_domains() const	Get the sub domains of the partition.
void set_partition_mapper (partition_mapper_type* _pm)	Set a reference to the associated partition mapper.
partition_mapper_type* get_partition_mapper(void)	Get a reference to the associated partition mapper.
size_t memory_size() const	Compute the memory used. Counted as part of the `pContainer` metadata.

Table VIII.: Ordered partition interface.

Define Type	Description
bcid_type	`bContainer` identifier type
Method	**Description**
bcid_type get_first()	The identifier of the first `bContainer`.
bcid_type get_last()	Last identifier. Convention such that `get_next(last valid bContainer identifier)` returns `get_last()`
bcid_type get_next(bcid_type)	Computes the identifier of the `bContainer` following the one given as argument, according to the relation order the ordered partition implements
bcid_type get_prev(bcid_type)	Computes the identifier of the `bContainer` before the one given as argument

The ordered partition is another concept of the PCF and it requires users to

provide, as a nested data type, a class encapsulating the order among sub domains. This concept is referred to as `bcids_info` and has the interface included in Table VIII.

5. Partition Mapper Interface

In the PCF a unique `bContainer` identifier (`BCID`) is associated with every sub-domain of a partition. The `partition-mapper` of a `pContainer` provides the mapping from the set of sub domain identifiers (`BCIDs`) to a set of locations. The `pContainer` will use the partition mapper to decide the locations where individual `bContainers` will be allocated and an interface to find where a specific `BCID` has been mapped. The complete interface is included in Table IX.

Table IX.: Partition mapper interface.

Define Type	**Description**
bcid_type	`bContainer` identifier type
location_type	Location identifier type; a type convertible to armi::location_type
partition_mapper()	Default constructor; the mapper is not initialized; init can be used afterward
bool is_local(const bcid_type&) const	Returns true or false depending if the argument `BCID` is local or not
const std::vector<bcid_type>& get_local_cids() const	Returns the list of local allocated `BCIDs`.
location_type map(const bcid_type& sub_domain_id) const	Returns the location where the sub domain identifier passed as argument may live
void init(const cids_info_type&)	Initialize the partition mapper with information about the list of `BCIDs` passed as argument
size_t get_num_bcontainers()	Returns the total number of `bContainers` managed by the current mapper
size_t memory_size(void) const	Returns the memory size occupied by this object. It will be counted by the `pContainer` as part of the metadata memory usage

The PCF provides a set of partition mappers that are briefly introduced next. Assuming the sub-domain identifiers are from 0 to $m-1$ and the location identifiers are from 0 to $L-1$. The `cyclic_mapper`, for which sub-domains are distributed cyclically among locations; `blocked_mapper`, where m/L consecutive sub-domains are mapped in a single location and `general_mapper` that can arbitrary map any sub-domain to any location.

6. Data Distribution Manager

The `data_distribution_manager` base class is responsible for managing the `pContainer` partition and partition mapper. All `pContainer` methods that deal with elements are forwarded to the data distribution manager. This class uses the partition and the partition mapper to determine the locations and the `bContainers` where the method will be executed finally.

As shown in Chapter VI, to simplify the `pContainer` developers effort while interacting with the partition, the partition mapper, and thread safety management, the `data-distribution-manager` provides a skeleton for any element-wise method that users can customize by providing appropriate functors. The generic method execution support is encapsulated within a set of methods called `invoke` which are shown in Figure 8. The actions performed inside `invoke` require cooperation from all previously introduced `pContainer` modules. The method receives as input a unique method identifier and two functors : `FunctorWhere` and `FunctorAction`. Both functors are redirections to the appropriate methods in the partition. The first action performed by invoke is to query the partition for the `BCID` of the `bContainer` where the method needs to be invoked (Figure 8, line 5). The partition returns a `bContainer` info structure that either contains the exact `bContainer` identifier or a new location where the method needs to be forwarded to be further processed (Figure 8, line 9). If

```
1  template<typename FunctorAction, typename FunctorWhere>
2  invoke(size_t m_id, FunctorAction f, FunctorWhere where){
3   location_type loc;
4   bcid_type cid;
5   //next query the partition where is the element located
6   bcid_type cinfo = where(this->m_ps);
7   if(!cinfo.cid_valid()){
8    // the partition returned partial information with a new
9    // location that may know more information about the
10   // mapping from gid to bcid
11   loc = cinfo.lid();
12  }
13  else {
14   //the partition was able to map from gid to bcid
15   loc = this->get_partition_mapper()->map(cinfo.cid());
16  }
17  if (this->get_location_id() == loc) {//if local
18   cid = cinfo.cid();
19   //partition performs the method on the bContainer cid
20   f(this->get_partition(),cid);
21  }
22  else {
23    async_rmi(loc, this->getHandle(),
24              &this_type::invoke,
25              m_id,f,w);
26  }
27 }
28
29 p_array::p_container_indexed::set_element (_gid, _val) {
30   this->m_dist->invoke(MP_SET_ELEMENT,
31     boost::bind(&partition_type::set_element, _gid, _val),
32     boost::bind(&partition_type::get_info,_gid));
33 }
```

Fig. 8. The `invoke` method of the data distribution manager.

the exact `BCID` is returned by the partition, then the `pContainer` uses the partition mapper to identify the location where the method needs to be executed or forwarded (Figure 8, line 13). If the identified location is the current one, then the partition will be asked to perform the `FunctorAction` on the corresponding `bContainer` (Figure 8, line 17). If the location is a remote one, than the method will be forwarded and re-executed on that location (Figure 8, line 20).

With this mechanism the `pContainer` can implement the shared object view. The methods are forwarded and executed on corresponding locations according to the policies specified in the partition class. In Figure 8, starting with line 26, we show the implementation of the `pArray` `set_element` method as a simple redirection to the distribution manager `invoke` method with the FunctorWhere querying the partition `get_info` and the FunctorAction invoking `set_element` method of the partition for indexed `pContainer`.

The invoke methods perform additional actions related to the atomicity of execution that will be discussed in Chapter VI. The `data-distribution-manager` interface is included in Table X.

Table X.: Data distribution manager interface.

Define Types	**Description**
domain_type	The partition domain type.
gid_type	The partition `GID`.
bcid_type	The partition `BCID`.
partition_mapper_type	Partition mapper type
location_type	The location type
partition_type	The partition type
location_manager_type	The location manager type
ths_manager_type	The thread safety manager type
Method	**Description**
data_distribution_base()	Default constructor; the distribution information is uninitialized
data_distribution_base(domain_type& `domain`, partition_type p)	Initialize the data distribution information based on the input domain and partition; the partition mapper will be allocated based on the template argument
data_distribution_base()	Destructor in charge of calling the destructor of the partition and partition mapper
void clear()	Deallocate the partition and the partition mapper
bcid_type get_info(const gid_type&_gid) const	Returns the sub domains that contains the input `GID`.
bool is_local(const gid_type& _gid) const	Returns true or false if the `GID` argument is mapped on the current location
location_type lookup(const gid_type&_gid) const	Returns the location that owns or that may have more information about the `GID`
partition_type* get_partition()	Returns a pointer to the partition.
partition_mapper_type* get_partition_mapper()	Returns a pointer to the partition mapper.
template<typename FunctorAction, typename FunctorWhere> void invoke(size_t m_id, const FunctorAction& f, const FunctorWhere& w)	Support for asynchronous method execution
typename FunctorAction::result_type invoke_ret(size_t m_id, const FunctorAction& f, const FunctorWhere& w)	Support for synchronous method execution
typename FunctorAction::result_type invoke_opaque_ret(size_t m_id, const FunctorAction& f, const FunctorWhere& w)	Support for split phase method execution

Table X continued

Method	**Description**
size_t memory_size(void) const	Memory used by this class and its data members (partition, partition mapper); reported as metadata

The `pContainer` uses the data distribution (partition, partition mapping), the location manager, and the `bContainer`, to determine the complete location information (location, memory reference within location) where the data element corresponding to a certain `GID` is allocated. The data distribution manager, the location manager, and the `bContainer` are the modules in our framework that allow the `pContainer` to provide a shared memory view to the user.

D. Specification for pContainer Framework Concepts

In this section we introduce the complete interface of the base `pContainer` classes introduced in Section A and depicted in Figure 5. Each of these concepts will have associated interfaces and requirements for `bContainers`, Domains, Partitions, Partition Mappings and Location Managers which are discussed in the following sections.

1. pContainer Base

All STAPL `pContainers` derive from `p_container_base` class. This class is in charge of storing the data using a `location-manager` and data distribution information using a `data-distribution-manager`. It provides a simple interface to initialize the `pContainer` based on the traits class provided as a template argument, and a domain and partition instance. The complete interface is described in Table XI and includes constructors, copy constructors that copy both data and metadata, and methods

to retrieve references to the `location-manager` and `data-distribution-manager`. The type of the domain, partition, partition mapper, `location-manager` are passed to this base class using the template traits class. Users will be able to customize the behavior of this class by passing proper traits. This process will be exemplified in Section H.

Table XI.: Base `pContainer` interface.

Template Arguments	**Description**
`Traits`	`pContainer` traits;
Define type	**Description**
`partition`	`pContainer` partition; Specified in the traits.
`domain`	`pContainer` domain; Specified in the partition type.
`location_manager_type`	`pContainer` location manager; Specified in the traits.
`distribution_type`	`pContainer` distribution manager; Specified in the traits.
Method	**Description**
`constructor`	Collective Operation. Default constructor, registers the `pContainer` with RTS.
`constructor( p_container_base& other)`	Collective Operation. Registers the `pContainer` with RTS and copy the content of the other.
`void init(domain*, partition*)`	Initialize based on a domain and partition. The partition is used to define the sub domains and implicitly the `bContainers` of the `pContainer`.
`location_manager_type* get_location_manager()`	Returns a pointer to the `pContainer` location manager
`distribution_type* get_distribution()`	Returns a pointer to the data distribution manager

All STAPL `pContainers` are `pObjects`, which means that they need to be allocated in an SPMD style on all locations where the `pContainer` will distribute its

data. The pContainer will have a representative on all these locations and the union of all pContainer representatives makes the overall pContainer. Having access to a pContainer representative is equivalent to having access to the whole pContainer. According to these requirements, the constructors are collective operations and they are responsible for registering the pContainer with the RTS. The registration will happen in the base class of the p_container_base, p_object. The registration returns a handle object that is stored and used during the pContainer's live time to perform remote method invocations across different locations.

The init method is in charge of the initial setup of the pContainer classes. It takes as argument a domain and a partition. It will ask first the partition to provide a decomposition of the given domain, followed by a query of the partition mapper to decide which of the sub-domains will be mapped to which individual locations. This process happens simultaneously on all locations and subsequently all locations will allocate the bContainers for the local allocated sub domains. The bContainers are then added to the location manager that will further administer them.

All STAPL pContainers are designed to report their memory usage and this is provided by the memory_size method. This is a collective operation which will return a pair containing the metadata size in the first field and the data size in the second field. The p_container_base achieves this by recursively invoking the method size on the data-distribution-manager and location-manager. The size is reported in bytes.

In the following sections we describe the extensions and specializations of the base concepts introduced for the different pContainer specializations proposed in our taxonomy.

2. Static pContainer

The size of a static pContainers is fixed when the `pContainer` is declared and will not change afterward. All interfaces described for the base classes will be available for static `pContainers`. The property that the number of elements is fixed allows for more efficient implementations of domains, partitions and `pViews` to be used. For example partitions and `pViews` based on closed form solutions can be used with this type of container.

Table XII.: Static `pContainer` interface.

Template Arguments	**Description**
`Traits`	`pContainer` traits;
Method	**Description**
size_t local_size() const	Returns the local size of the pContainer.
size_t size() const	Returns the size of the pContainer.
bool local_empty() const	Check if all the local bcontainers are empty.
bool empty() const	Check if the pContainer is empty; return true if the pContainer is empty and false otherwise.
template<class Functor> typename Functor::result_type apply_get(gid_type i, Functor f)	Apply a functor f to the data corresponding to the `GID`; The functor has a return type.
template<class Functor> void apply_set(gid_type i, Functor f)	Apply a functor f to the data corresponding to the `GID`; The functor does not have a return type.
iterator begin() const	Iterator to the first element of the `pContainer`.
iterator end() const	Iterator pointing one past last valid element.
bool is_local(gid_type gid) const	Returns true or false if the argument `GID` is local or not.
location_type lookup(gid_type gid) const	Returns the location where the given `GID` may be found.
bool is_local(gid_type gid, cid_type& bcid) const	Returns true or false if the argument `GID` is local or not.

The additional interface that a static pContainer provides is included in Table XII. It includes query methods for different properties such as size() and empty() methods to obtain references to the first or last element of the pContainer, and references to an element with a given GID.

3. Dynamic pContainer

A dynamic pContainer allows elements to be added and deleted from a pContainer. We include in Table XIII the additional interface a dynamic pContainer supports.

Table XIII.: Dynamic pContainer interface.

Template Arguments	**Description**
Traits	pContainer traits;
Method	**Description**
void clear()	Equivalent to remove all elements; The distribution and location manager remain valid
cid_type add_bcontainer(bcontainer_type* c)	Add a bContainer allocated dynamic outside; the pContainer will just use it without any copying involved
cid_type delete_bcontainer(bcontainer_type* c)	Delete the requested bContainer.

4. Indexed pContainer

Containers in this category provide an interface to access the elements based on their index. The domains associated with the containers in this category are derived from Cartesian domains or compositions of Cartesian domains.

The indexed pContainer inherits either from a static or dynamic pContainer

and this is specified in the traits class. The additional interface supported by an indexed pContainer is included in Table XIV.

Table XIV.: Indexed pContainer interface.

Methods	**Description**
void set_element(const gid_typ& gid, const value_type& val)	Set the value of an element associated with a certain gid.
value_type get_element(const gid_type& gid) const	Get the value of an element associated with a certain gid
pc_future<value_type> split_phase_get_element(const gid_type& gid) const	split phase get element (two phase get element); It returns a future that can be queried if the value is available or not
reference operator[](gid_type gid)	Returns a reference to the element corresponding to the GID;
value_type operator[](gid_type _gid) const	Returns the element

Corresponding to the element-wise interface introduced (e.g., set_element, get_element, split_phase_get_element) the partitions of indexed pContainers require the additional interface included in Table XV. The new interface is used by the pContainer indexed as the FunctorAction. For the FunctorWhere the generic get_info method of the partition base class is used as exemplified in Figure 9.

```
1  p_container_indexed::set_element (_gid, _val) {
2    this->m_dist->invoke(MP_SET_ELEMENT,
3      boost::bind(&partition_type::set_element, _gid, _val),
4      boost::bind(&partition_type::get_info, _gid));
5  }
6
7  value_type p_container_indexed::get_element (_gid) {
8    this->m_dist->invoke(MP_GET_ELEMENT,
9      boost::bind(&partition_type::get_element, _gid),
10     boost::bind(&partition_type::get_info, _gid));
11 }
```

Fig. 9. The `pContainer` indexed method implementation.

Table XV.: Indexed partition interface.

Methods	**Description**
void set_element(const gid_typ& gid, const value_type& val, const bcid_type& bcid)	Set the value of an element associated with a certain gid in the specified `bContainer`.
value_type get_element(const gid_type& gid, const bcid_type& bcid) const	Get the value of an element associated with a certain gid

The domains used with indexed partitions need to implement the finite ordered domain interface as specified in Chapter IV, Section B.3. The framework provides default implementations for one and two dimensional indexed `pContainers` that are used by `pArray`, `pVector`, `pMatrix`.

Indexed Partitions:

- `partition_balanced`: used by `pArray`. For a given domain of size N it will create P sub-domains, each of size N/P. If $N < P$ then there will be N sub domains of size 1;
- `partition_blocked` : used by `pArray`. For a given domain of size N and a block size BS there will be N/BS sub domains created, each of size BS
- `partition_blocked_explicit`: used by `pArray`. The constructors will accept an explicit decomposition of a domain in sub-domains of arbitrary sizes.
- `pv_unbalanced_partition`: used by `pVector`. It is initially constructed similar to `balanced_partition` of the `pArray`. Subsequent insert or delete operations may lead to unbalanced blocked partitions.
- `p_matrix_partition`: used by `pMatrix`. Allows user to specify block or block cyclic decompositions, with row or column wise decompositions.

User-Partition Interaction

Advanced users will interact with the partition to specify the granularity of data for different computations and to control how data will map on the machine. Creating a new partition object will be performed by invoking the appropriate constructors. For example, for static array-like data structures the users can choose from among the following partition strategies:

Examples: We assume a domain D = [1..10].

1. `partition_balanced(domain, 2 /*num_subdomains*/);`

   P = { 0..5, 6..10 }

2. `partition_blocked(domain, 3/*block size*/);`

   P = { 0..2, 3..5, 6..8, 9..10 }

3. `partition_block_cyclic(domain, 2, BLOCK_CYCLIC(3));`

   P = { {0,1,2 ,, 6,7,8} {3,4,5 ,, 9,10} }//two domains, cyclic, group size 3

4. `partition_block_cyclic(domain, 2, BLOCK_CYCLIC(1));`

   P = { {0,2,4,6,8,10} {1,3,5,7,9} } //two domains, cyclic, block 1

5. `partition_blocked_explicit(domain, BLOCK(v{3,4,4}));`

   P = { 0..2, 3..6, 7..10 }

The above examples can be correspondingly extended to multi-dimensional domains such as 2D or 3D arrays.

5. Associative pContainer

Containers in this category are dynamic `pContainers` that have associated two data types: *Key* and *Value*. The interface supported by an associative container extends the interface provided by the base and dynamic to accommodate the fact that we have key/value pairs and it is included in Table XVI.

Table XVI.: Associative `pContainer` interface.

Template Arguments	**Description**
`Traits`	`pContainer` traits;
`Key`	Key type
`Value`	Value type
Methods	**Description**
reference operator[](const gid_type& gid)	Returns a reference to the element corresponding to the `GID`
void erase_async(const key_type& key)	Erases key asynchronously

void clear()	Clears the pContainer; equivalent to erase(begin,end)
iterator find(const key_type& key)	Returns an iterator pointing to an element identified by its key. If the key does not exist the `pContainer`'s end is returned.
pair<value_type,bool> find_val(const key_type& key) const	Find value_type corresponding to key_type. The boolean of the pairs signals the fact that the argument key exists in the `pContainer` (true) or not (false)
pc_future<value_type> split_phase_find(const key_type& key) const	Split phase find value corresponding to the key; It returns a future that can be queried if the value is available or not

6. Relational pContainer

Containers in this category store elements (e.g., vertices in a graph) and relationships between elements (e.g., edges). The interface extends the base classes with methods to specify relationships between elements and it is included in Table XVII.

Table XVII.: Relational `pContainer` interface.

Template Arguments	**Description**
`Traits`	`pContainer` traits;
Define Type	**Description**
vertex_property	Vertex property type
vertex_descriptor	Vertex descriptor type
edge_descriptor	Edge descriptor type
edge_property	Edge property type
vertex_iterator	Vertex iterator type
const_vertex_iterator	Const vertex Iterator
adj_edge_iterator	Adjacency edge iterator
const_adj_edge_iterator	const adjacency edge iterator
Method	**Description**

Table XVII continued

Method	Description
gid_type add_vertex(void)	Add a new vertex into the graph; returns the vertex descriptor.
gid_type add_vertex(const vertex_property& vp)	Add a new vertex into the graph with the specified property.
void add_vertex(const gid_type& gid, const vertex_property& vp)	Add a vertex with value val and vertex descriptor gid.
void delete_vertex(const gid_type& _vd)	The following method is not a transaction; Deleting edges and vertices are atomic but the whole method is not atomic since the execution of this method may be done across different locations.
edge_descriptor add_edge(const edge_descriptor& ed, const edge_property& ep, bool multi=true)	Add an edge.
void add_edge_async(const edge_descriptor& ed, bool multi=true)	Add an edge asynchronous.
void add_edge_async(const edge_descriptor& ed, const edge property& ep, bool multi=true)	Add an edge with given property asynchronously.
bool delete_edge(const edge_descriptor& ed)	Delete the edge identified by its descriptor.
size_t get_num_vertices(void) const	Returns the total number of vertices.
size_t get_local_num_edges(void) const	Returns the number of edges on the local storage.
size_t get_num_edges(void) const	Returns the total number of edges.
void clear(void)	Erase all vertices and edges in the graph; reset distribution/partition.
vertex_iterator find_vertex(gid_type gid)	Returns a vertex iterator corresponding to the `GID` argument.
bool find_edge(const edge_descriptor& ed, vertex_iterator& vi, adj_edge_iterator& ei)	Returns a vertex iterator corresponding to the source vertex of the edge ed and an edge iterator of the edge.

Partitions that can be used with generic relational `pContainers` are arbitrary

maps where individual `GIDs` can be map randomly to sub-domains. Block and block cyclic partitions can be used for regular `pGraphs` with fixed number of vertices.

7. Sequence pContainer

For conformance with the STL taxonomy, we provide the *Sequence* as a relation container with an implicit relationship (next). The `pList` implements the sequence interface while the `pVector` implements both sequence and indexed. The sequence interface is included in Table XVIII.

Table XVIII.: Sequence `pContainer` interface.

Template Arguments	**Description**
`Traits`	`pContainer` traits;
Method	**Description**
void push_back(const value_type& val)	Add a new element at the end of the container.
void pop_back()	Remove an element at the end of the container.
void push_front(const value_type& val)	Add a new element at the beginning of the container
void pop_front()	Remove an element at the end of the container.
void insert_element_async(const gid_type& gid, const value_type& val)	Insert asynchronously a new element before gid specified.
gid_type insert_element(const gid_type& gid, const value_type& val)	Insert synchronously a new element before gid specified. Returns the `GID` of the newly inserted element.
void erase_element(const gid_type& gid)	Erase the element corresponding to the specified gid.
void push_anywhere_async(const value_type& val)	Add an element to the `pContainer`at an arbitrary position.
reference get_anywhere()	Returns a reference to a random element in the `pContainer`.
void remove_element()	Remove a random selected element

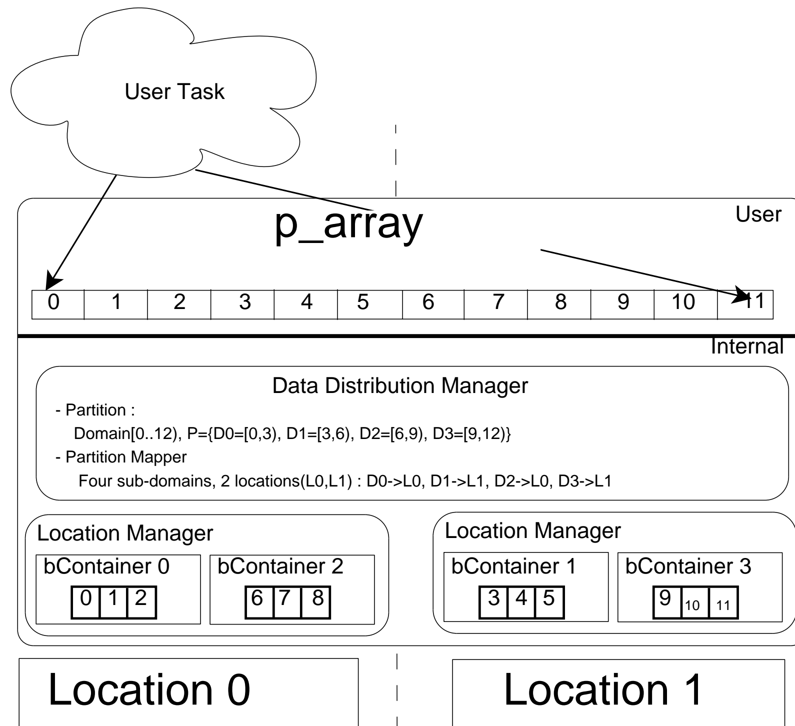


Fig. 10. Example of pContainer deployment on two locations.

E. Integrating all Concepts using pArray Example

In this section we exemplify how all concepts described in Section B are integrated together to implement a simple `pArray` data structure. The STL `valarray` container is a fixed size data structure optimized for storing and accessing data based on one dimensional indices. The STAPL `pArray` is the parallel equivalent of the STL `valarray`, providing an efficient interface to access data elements using indices. More information about the complete interface is included in Chapter IX.

Global Identifiers (`GIDs`) and Domains: For the `pArray`, the `GIDs` are the indices of the elements. The `pArray` domain is the universe of `GIDs`that identify its elements and is represented as an integer range corresponding to the indices of the

elements (e.g., 1DRange(0,12)). The `pArray`'s domain is a total ordered domain that specifies how elements are traversed by iterators of the default view. The `pArray` constructors accept an unsigned integer as an argument (N) that internally is converted into a finite order domain (1DRange(0,n)) and the resulting index space and order of the elements will be as specified by the domain. It is trivial to extend the interface to have the `pArray` with an arbitrary domain, e.g., `p_array<>(Domain(5,12))`. This will declare a `pArray` whose first and last elements have indices 5 and 11, respectively. In Figure 10, the `pArray` has the range $[0, 12)$ as its domain.

Partition The `pArray` is a static container, e.g., we can use `blocked partitions` using a *block_size* as an argument. Assuming N is the size of the domain to be partitioned, this partition creates $N/block_size$ sub-domains of size *block_size*, except the last one which may be smaller. Other partitions for `pArray` are balanced partitions, that will divide the elements of the domain into the specified number of sub-domains, each of whose size is $N/\#sub_domains$. Explicit partitions are built by explicitly enumerating the sub-domains. The STAPL `pArray` can be built with any of these partitions. An important feature of STAPL is that the well-defined partition interface enables advanced users to implement their own partitions.

In Figure 10 we show the `pArray` with a blocked partition with blocks of size 3. The corresponding sub-domains that this partition strategy (split) generates for the input domain $OD = ([0, 12), \leq)$ are:
$P = \{OD_0 = [0, 3), OD_1 = [3, 6), OD_2 = [6, 9), OD_3 = [9, 12)\}$

Partition Mapper: A partition is mapped onto a set of locations using a partition-mapper, which maps a sub-domain identifiers to a location. Any of the mappers introduced in Section B.5 can be used with the `pArray` data structure. Additional mappers with more information about the machine and interconnect can be implemented by users provided the interface included in Table IX is implemented. In

Figure 10 we show the blocked partition $P = \{OD_0, OD_1, OD_2, OD_3\}$ being mapped onto available locations in a cyclic fashion. Thus sub-domain OD_0 is mapped to location 0, OD_1 is mapped to location 1, OD_2 is mapped to location 0 and OD_3 is mapped to location 1.

Storage `bContainer`: The `pArray` associates with every sub-domain of the partition a `bContainer` for data storage. The `bContainers` are implemented as STL `valarrays`. In Figure 10 we show the `pContainer` with two location managers instances, one in each location were the `pContainer`'s data will reside. Each location manager handles the `bContainers` for the sub-domains that were determined by the partition and partition mapper.

```
1   value p_array::set_element(GID, value){
2     bcid = distribution_manager.partition.map(GID)
3     location = distribution_manager.partition_mapper.map(bcid)
4     if location is local
5       return location_manager.bcontainer(bcid).set(GID, value)
6     else // set remotely the element
7       return async_rmi(loc, &set(), GID, value);
```

Fig. 11. Pseudocode of `pArray` `set()` method.

The simplified code for the `pArray` method `set_element` is shown in Figure 11 to illustrate how the `pArray` modules interact. The complete method performs additional actions as described in this chapter, Section B.6 and Chapter VI, Section B. The runtime cost of the methods in the `pArray` interface has three main constituents: the time to decide the location and the `bContainer` in which the element

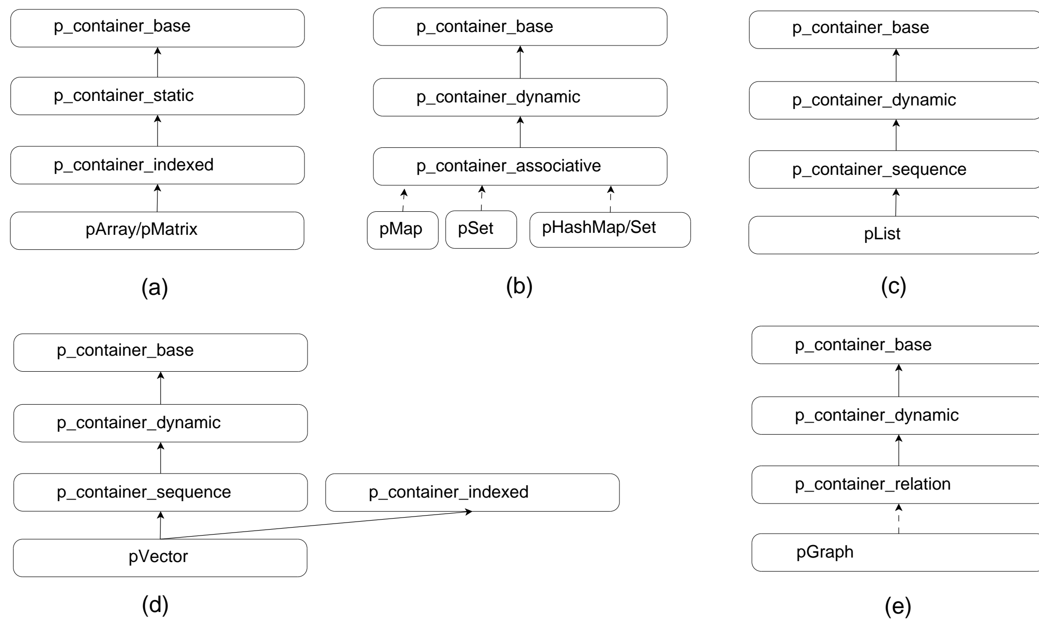


Fig. 12. `pContainers` inheritance. The most derived classes inherit all the methods of the base classes. A dotted line denotes that other classes are inheritted as explained in Chapters XI and XII.

is stored (Figure 11, lines 2-3), the communication time to get/send the required information (Figure 11, line 7), and the time it takes to perform the operation within a `bContainer`, which is currently an STL `valarray` (Figure 11, line 5).

F. pContainers Implemented in the Framework

STAPL provides a collection of commonly used `pContainers` that are constructed using the PCF. This includes counterparts of STL containers (e.g., `pArray` [63], `pVector`, `pList` [65], and associative containers such as `pSet`, `pMap`, `pHashMap`, `pMultiSet`, `pMultiMap` [64] and additional containers such as `pMatrix` [15], and `pGraph`. In Figure 12 we depict the relationship between these `pContainers` and the classes of the

framework.

The pArray and pMatrix (Figure 12(a)) are indexed containers using one or two dimensional indices (GIDs) respectively. The interface for these simple data structures is mainly provided by the base classes with the final classes providing only constructors and necessary type definitions. The complete interface for the pArray is discussed in Chapter IX. The pMatrix is described in more detail in [15], and provides a similar interface with the pArray. The pList (Figure 12(c)) derives from the sequence pContainer from which it inherits a reach interface to add and erase elements at the begining, the end and an intermediate point in the sequence. The complete interface for pList, discussed in more detail in Chapter X implements all the methods in the sequence interface in constant time. The pVector is a sequence pContainer that also implements the indexed pContainer interface as shown in Figure 12(d). Due to constraints on complexity for the indexed interface the pVector incurs a bigger overhead when implementing insert operations. While both pList and pVector are two sequence pContainers providing similar interfaces, there is a well known performance/usability tradeoff between the two. The pVector provides constant access time to the elements based on their indices, linear time for inserts, and amortized constant time for push back type methods. The pList does not provide random access to the data based on indices but implements dynamic operations such as insert and push back in constant time. Depending on the particular needs of an application, these two data structures can be used for different computational phases with possible conversions from one to the other. More details about the tradeoffs between pList and pVector are included in [65].

Associative pContainers provide efficient storage for elements based on keys. They include sorted associative containers, which guarantee logarithmic access time to the elements, and hashed associative containers that guarantee amortized constant

time. They provide a simple interface that includes insert find and erase as described in more detail in Chapter XII. The `pGraph` is a relational `pContainer` consisting of a collection of vertices and relations between vertices called edges. The framework provides a relational `pContainer` base that implements a minimal interface to add and delete vertices and edges. Additional functionality is supported by the `pGraph` specific classes as described in Chapter XI.

G. pContainer Support for Redistribution

One of the design goals of the PCF is to allow `pContainers` with various partitions and mapping on the machine. This can be achieved by specifying the desired partitions and partition mapper as template arguments at compile time. The data redistribution is the process of reorganizing the data of a `pContainer` based on a new data distribution (new partition and/or partition mapping) as described in Chapter IV, Section C. We integrate into the PCF support for allowing an individual `pContainer` to change its partition and partition mapping dynamically during the execution. This is achieved by using polymorphic implementations for both the partition and partition mapper and providing the necessary support to move data across different locations. For example, for `pArray` we mentioned we currently support balanced, blocked and explicitly blocked partitions. These three types of partitions can be interchanged within the same `pContainer` instance dynamically. In this section we describe the support implemented in the PCF to allow this functionality.

The **Partition Proxy** is a polymorphic wrapper for real partitions and provides the necessary support to change the underlying partition at runtime. There are some trade-offs when using a partition proxy. While giving users more flexibility it involves virtual methods with the associated overhead and missed opportunities for compile time optimizations. The default partitions of all `pContainers` are proxy partitions and the framework provides proxies for all components of the taxonomy: partition_proxy_indexed, partition_proxy_dynamic, partition_proxy_sequence, partition_proxy_associative, and partition_proxy_relation. When a partition proxy is used, the interface of the `pContainer` is automatically extended with the the following interface:

```
1    void redistribute(new_partition [, partition_mapper]);
2    void redistribute (redistribution_map);
```

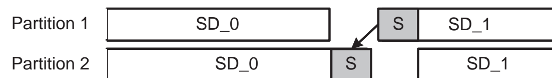


Fig. 13. Redistribution for two given partitions. Section S (one or more elements) of sub-domain 1 in first partition will migrate to sub-domain 0 in the second partition.

Trying to invoke the above methods on a `pContainer` with a non proxy partition will generate a compiler error.

While performing the redistribution, there is the new partition and/or partition mapper and the original ones. A naive redistribution approach can simply create a new `pContainer` organized according to the new partition, copy data from the old storage and delete the old storage. However this is a very inefficient approach. To assist users in performing efficient redistribution, we introduce the *redistribution map* which contains only the elements that will migrate from a sub-domain to a new sub-domain. A common case occurs when the repartition moves elements across neighboring sub-domains. The redistribution map will benefit the redistribution in this situation. In Figure 13, we depict a case were the redistribution will have to migrate only the data corresponding to sub-domain S in order to match the second partition. The framework will provide a set of predefined constructs for common redistribution maps while the users will provide their own for more specific patterns.

Some simple redistribution patterns that can be easily provided by the framework are:

- rebalance() : Redistribute the N elements of the `pContainer` across P locations such that each location will own N/P elements
- rotate (how many positions, direction) : Redistributes the elements of the

pContainer by cyclically rotating them a given number of locations in a given direction.

- custom redistribution for certain pContainers: transpose for two dimensional containers, graph redistributions, etc.

1. Data Marshaling

To support the default redistribution, the framework requires that both pContainer data elements and bContainers be marshaled. Support for data marshaling is provided by the STAPL RTS[54] and requires users to implement a define_type() method as part of the class that needs to be marshaled. The bContainer interface includes the define_type as part of its required interface. This can easily be achieved as STAPL provides built in support for all STL containers and these are the building blocks for most bContainers we employ in our pContainers.

In Figure 14 we include examples showing the define_type() implementation for some simple classes and for the pArray bContainer. The method receives as input an object of type stapl::typer. Subsequently, the typer is made aware of all the class data members and this process continues in a recursive fashion. In Figure 14(a), class classB has as a data member an object of classA so the define type of classB will recursively invoke the define type of it. In Figure 14(b), we show the define_type for the pArray bContainer which invokes recursively the define type of the STL valarray data member.

With bContainer marshaling support available, the pContainer redistribution implementation is greatly simplified. The redistribution map specifies the sub-domains and their new locations. Data corresponding to sub-domains is appropriately packed in bContainers and shipped to the destination

```
1  class classA {
2  int a;
3  double b[10] ;
4  void define_type(typer&t){
5    t.member(a,10);
6    t.member(b);
7  }
8  }
9  class classB {
10 objectA a;
11 void define_type(typer&t){
12   t.member(a);
13 }
14 }
```

(a) Simple classes

```
1 template<class T>
2 class p_array_bcontainer{
3  std::valarray<T> m_data;
4  //specific interface ...
5
6  void define_type(typer&t){
7    t.member(a);
8  }
9 }
```

(b) `pArray` `bContainer`

Fig. 14. Marshaling interfaces.

H. pContainer Customization using Traits

When building a `pContainer`, a developer has the ability to customize the `pContainer` main functional modules as described in Chapter IV, Section B. For example, a developer may want to use a certain storage or certain partition and mapping on the machine, enforce a certain memory consistency model, or enable/disable thread safety, etc. This leads us to design the PCF in a very configurable way where users can select individual functional modules. This is accomplished using *traits* classes that are passed as template arguments to `pContainer` base classes. The traits can be customized by developers and end users for classes of `pContainers` or even on a per `pContainer` instance.

In addition to the data structure specific arguments, all `pContainer` classes take as template arguments the partition and the `pContainer` traits. For example, in

```
1  //the STAPL Parallel Array Definition
2  template<typename T,
3            typename Partition=partition_balanced<>,
4            typename Traits=p_array_traits>
5  class p_array : public p_container_indexed<Trais>{...}
6
7  //the STAPL Parallel Graph Definition
8  template <graph_attributes D, graph_attributes M,
9    typename VertexP = no_property,
10   typename  EdgeP   = no_property,
11   typename Partition=partition_relation,
12   typename Traits   = p_graph_traits >
13 class p_graph : public p_container_relation<Traits> {...}
```

Fig. 15. `pContainer` template arguments.

Figure 15 we list the definition of the `pArray` and `pGraph`; more detail on these specific data structures is provided in Chapter IX and XI, respectively:

The partition specifies the `GID`, `domain`, `bContainer`, and `thread_safety_manager`. The `pContainer` will use the definition for these types from the partition. The `pContainer` traits can be used to specify lower level details such as `partition_mapper`, `data-distribution-manager` and `location-manager`. We expect these to be less often customized by users than the storage and partition.

When interacting with the framework users can provide alternative implementations for domain, partition, `bContainer`, `partition_mapper` and `location-manager` using the template arguments. The custom provided modules need to implement the required interfaces as specified in Sections B and D.

In Figure 16 we show an example of STAPL pseudocode illustrating how users can customize an existing `pGraph` implementation. Users can select the storage by providing the type of an existing `bContainer` and similarly for the partition. Figure 16, line 5, shows the declaration of a directed `pGraph` allowing multiple edges between

```
1  //bcontainer definition:
2  //  sequential graph using vector storage
3  typedef pg_base_container<vector,> bpg_s;
4  //  and a static partition
5  typedef pg_static<bpg_s,...> partition_s;
6
7  //parallel graph using std::map storage
8  typedef pg_base_container<map,...> bpg_d;
9  //and a dynamic partition
10 typedef pg_fwd<bpg_d,...> partition_d;
11
12 //pgraph with static partition
13 p_graph<DIRECTED,MULTI, partition_s> pg_s(N);
14 //pgraph with dynamic storage and
15 //method forwarding
16 p_graph<DIRECTED,MULTI, partition_d> pg_d(N);
```

Fig. 16. `pGraph` customization.

the same source and a target vertex and using a static partition. With a static partition, users need to declare the size of the `pGraph` at the construction time and subsequent invocations of the `add_vertex` method will trigger an assertion. Figure 16, line 6, shows the declaration of a `pGraph` using a dynamic partition that allows for addition and deletion of both vertices and edges. More details and performance results regarding the benefits of having different partitions and types of storage are discussed in Chapter XI, Section F.2 in the context of a dynamic `pGraph` data structure.

CHAPTER VI

THREAD SAFETY

In this chapter, we describe the infrastructure provided by the `pContainer` framework to implement thread safe `pContainers`. The goal of STAPL is to allow users to easily develop thread safe containers while giving them the possibility to override the default locking policies implemented by the framework. We define that a `pContainer` is thread safe [33, 34, 35] if concurrent method invocations from various threads are perceived as being atomic thus always leaving a `pContainer` in a consistent state.

We start by first motivating the need for thread safety. Data stored in `pContainers` is accessed by `pAlgorithms` and at this level STAPL employs the task dependency graph (TDG) to encode data dependences across various tasks of a computation. However there are a large number of parallel algorithms employing commutative tasks. In this case, two tasks A and B are safe to be executed in $A \rightarrow B$ order (A followed by B) or $B \rightarrow A$ but not $A||B$ (A in parallel with B). For such scenarios the dependencies between two tasks can be eliminated under certain conditions as long as atomicity of the `pView` and `pContainer` method invocations are guaranteed.

As a simple example consider the sample sort parallel algorithm where each task inserts elements from an input `pArray` into a `pArray` of `pVectors` (buckets). This computation can be expressed as a no dependency TDG where each task determines the bucket an element belongs to and inserts the element into that bucket. This algorithm will properly insert all the elements into buckets as long as atomicity at the bucket level is guaranteed.

A. pContainer Thread Safety Design

As introduced in Chapter V, Section A, a `pContainer` is implemented as a collection of existing data structures to store data (e.g., `bContainers`) augmented with information for parallelism management or meta data (e.g., partition, partition mapper, thread safety manager).

When designing the `pContainer` thread safety mechanisms we considered the following:

- A `pContainer` method in general accesses and/or modifies both metadata and data so these two entities need to be considered in an integrated approach.
- A `pContainer` method invocation typically involves more than one location due to the forwarding mechanism introduced in Chapter V, Section C. This means that a `pContainer` method may access only metadata on certain locations while on other locations it may access both data and metadata.
- For most `pContainers`, the `bContainer` implementation is a black box. This is encapsulation and it is key to object oriented programming. When performing operations using the public interface we do not know what memory addresses are touched inside the `bContainer`, especially by dynamic operations such as insert/erase. For example, while an insert in a vector may touch all memory addresses for all elements encompassed in it, a user is not generally aware of it. For this reason, the framework takes a high level general approach where thread safety can be provided with or without support at the `bContainer` level.
- The framework has to be able to integrate existing thread safe and non-thread safe data structures to store both data and metadata. This is a crucial requirement in order to incorporate a large body of work that has been done in the

area of thread safe data structures for shared memory machines and multicores [20, 24, 26, 33, 34, 37].

- The framework needs to provide a generic, customizable solution, where different custom locking policies can be used with a particular `pContainer`. A desired thread safety manager can be selected by the user using custom traits.

Based on the above requirements we implemented the `pContainer` thread safety as a set of specifications across the following PCF modules:

- **Distribution Manager**: implements `pContainer` methods providing call back points to classes responsible for thread safety. This allows for customizable locking policies. See Section B.
- **Thread Safety Manager**: provides a generic interface such that users can control the type of mutual exclusion mechanism desired. See Section C.
- **Partition**: specification for the locking modes and the locking granularity performed by the thread safety manager.
- `bContainer`: locking at the base container level.

B. Data Distribution Manager

This module ensures the safe access to data and metadata by employing a thread safety manager. The data distribution implements a generic method skeleton such that individual `pContainer` element-wise methods are implemented as invocations of this generic method with customized functors to determine the location where the method will be executed and how it will be effectively executed. This was described in more detail in Chapter V, Sections C.4 and C.6. In Figure 17, we include the generic

method `invoke` presented in Chapter IV, Section B.6 but this time augmented with the locking support.

As depicted in Figure 17, the `pContainer` framework informs the thread safety manager about the actions it is about to perform. All the decisions on the granularity and type of locking to be performed are completely managed by the thread safety manager. This is either a default implementation or is provided by the user. It uses knowledge about the particular metadata and data implementation used to perform adequate locking. Please note that the framework doesn't implement a particular locking algorithm (there is no best one) but it enables users/developers to perform custom locking according to their specific data structure.

C. Thread Safety Manager

A thread safety manager class will be associated with every `pContainer`. The thread safety manager maintains the necessary information to ensure atomic updates to data and metadata of the `pContainer`. The functionality of this module will be employed by the PCF when thread safe access is required. The interface of the thread safety manager is described next:

```
1  class ths_manager{
2  public:
3  //define type containing the locking info
4  typedef ...(impl specific)...   ths_info;
5  constructor(MethodIdentifier, Partition[, gid]);
6  method_access_pre(ths_info);  //when entering the method
7  method_access_post(ths_info);  //when exiting the method
8   metadata_access_pre(ths_info); //before accessing metadata
9   metadata_access_post(ths_info); //after accessing metadata
10 data_access_pre(ths_info, bcid); //before accessing data
11 data_access_post(ths_info, bcid); //after accessing data
12 };
```

```
1    void invoke(method_id, where_functor, action_functor) {
2      location_type loc; cid_type cid;
3      //initialize the thread safety information
4      //for the current method
5      ths_info tinfo(method_id, this->m_partition[, gid]);
6
7      ths_manager()->method_access_pre(tinfo);
8      ths_manager()->metadata_access_pre(tinfo);
9       //query meta data
10      cid_type cinfo = where_functor(this->m_ps);
11     ths_manager()->metadata_access_post(tinfo);
12     if(!cinfo.cid_valid()){
13       //the exact location where the operation
14       //will be executed is known
15       loc = cinfo.lid();
16     }
17     else {
18       //Location not known;  Forward to a location
19       //that may have additional information
20       loc = this->m_pid_lid->map(cinfo.cid());
21     }
22     if (this->get_location_id() == loc) {
23       cid = cinfo.cid();
24       ths_manager()->data_access_pre(tinfo, cid);
25        // > critical section for data
26        action_functor(this->m_ps, cid);
27       ths_manager()->data_access_post(tinfo, cid);
28       ths_manager()->method_access_post(tinfo);
29     }
30     else {
31       ths_manager()->method_access_post(tinfo);
32       async_rmi(loc, this->getHandle(),
33                      &this_type::invoke,
34                      m_id, where_functor, action_functor);
35
36     }
37   }
```

Fig. 17. Generic `invoke` method implementation with locking statements.

In the specification above, the `MethodIdentifier` is an integer value uniquely associated with a `pContainer` method. The partition will define appropriate locking attributes for each of the `pContainer` methods as described in the next section.

D. Partition Locking Specification

A STAPL `pContainer` is designed to support different partitions with different locking modes for the methods in its interface. This is accomplished by routing all `pContainer` methods to the distribution manager, which in turn forwards to the partition class. Users can customize this behavior by specifying partition classes. This way they can specialize existing partitions or implement new ones with custom locking.

The partition class allows `pContainer` users to specify attributes for each individual method regarding the granularity of data accesses and the type of access. For example different methods may access an element in a `bContainer` (e.g., get_element()), the entire `bContainer` (e.g., insert into a vector) or all `bContainers` (e.g., size()). Corresponding to these three situations, the framework specifies the following identifiers: `ELEMENT, BCONTAINER, LOCAL`. Additionally each individual method accesses the data and metadata in a READ or WRITE mode, information that can be used in conjunction with read/write locks to allow multiple readers shared access to the data.

A special tag that can be used when specifying the granularity is `NONE`, which is used to inform the thread safety manager than no action is required. This can be the case for example for a read only static data structure such as `pArray` or `pMatrix`. Advanced uses of different thread safety attributes are envisioned where users can modify them dynamically for different computation phases. Below we include the

specification of the method attributes used by default by the `pVector`.

```
1  //initialization inside pvector partition constructor
2  typedef tuple<lock_granularity, rdwr_mode, rdwr_mode> tuple;
3  //Overload the default locking policies
4  m_locking_policy[LP_SET]       =(ELEMENT,WRITE ,MD_READ);
5  m_locking_policy[LP_GET]       =(ELEMENT,READ  ,MD_READ);
6  m_locking_policy[LP_ADD]       =(LOCAL  ,WRITE ,MD_WRITE);
7  m_locking_policy[LP_DELETE]    =(LOCAL  ,WRITE ,MD_WRITE);
8  m_locking_policy[LP_PUSH_BACK]=(LOCAL  ,WRITE ,MD_WRITE);
9  m_locking_policy[LP_POP_BACK]  =(LOCAL  ,WRITE ,MD_WRITE);
10 m_locking_policy[LP_INSERT]    =(LOCAL  ,WRITE ,MD_WRITE);
11 ...
12 // method to retrieve the locking policies associated with
13 // a given method
14 tuple<lock_granularity, data_rdwr_mode, metadata_rdwr_mode>
15 get_locking_policy(method_identifier){...}
```

A thread safety manager uses the above specification to perform appropriate actions (e.g., locking) when correspondingly invoked by the `pContainer` methods. In the remainder of this chapter we discuss various thread safety managers that can be used with STAPL `pContainers`.

E. pArray and pMatrix

Default implementation: The meta data is static and read only for all element-wise methods. Only data/`bContainers` are locked when accessing elements for read or write.

Customizations: Using traits, a custom thread safety manager can be specified that performs no locks. This may be useful for read only `pContainers` or for the case when concurrent accesses are taken care of by the task dependency graph of the application.

An interesting optimization that can be used to refine the lock granularity is to

provide a thread safety manager that uses K locks where K is arbitrary and each individual data access for a `GID` is hashed to one of the locks and that lock will be used when accessing the data for the given `GID`.

F. pList

Default implementation: The meta data is read only for all element-wise methods. Only data/`bContainers` are locked when accessing elements for read or write. Meta data is modified by collective operations and guarded by fence.

Customizations: Using traits, a custom thread safety manager can be specified that performs no locks. This may be useful for read-only `pContainers` or for the case when concurrent accesses are taken care of by the task dependency graph of the application.

Thread safe `bContainers` can be used, in which case no locking is performed by the framework.

G. Associative pContainers

Default implementation: The meta data is either static or closed form. Only data/`bContainers` are locked when methods are executed.

Customizations: Using traits, a custom thread safety manager can be specified that performs no locks. This is useful for the case when concurrent accesses are taken care of at the prange level.

Thread safe `bContainers` can be used, in which case no locking is performed by the framework.

H. pGraph

Default implementation: The meta data for a `pGraph` is maintained in a parallel map. It is dynamic and modified by every add/delete vertex. In general the vertices are added on one location and meta data inserted on another location. So adding a vertex is naturally broken into two individual operations. Since the meta data holder is thread safe, the `pGraph` performs no locking for it. Only when accessing the data in a `bContainer` locking is invoked.

Customizations: A static partition where the number of vertices is fixed at construction time. This kind of graph will allow for edges to be added/deleted. The atomicity can be enforced using a set of locks (or a set of locks per `bContainer`) and a hashing scheme to decide the lock to use based on the `GID`.

A thread safe graph `bContainer` can be employed to eliminate the locking performed by the framework.

CHAPTER VII

MEMORY CONSISTENCY MODEL

In this section we describe the memory consistency model (MCM) provided by STAPL `pContainers`. This consists of a set of guarantees about the termination, ordering and values returned by the `pContainer` method invocations that users rely upon when reasoning about the correctness of their applications [6, 46, 1, 3, 2]. The `pContainer` provides a shared memory view to the user while storing its data distributed on different memory address spaces. In Figure 18 we depict a `pContainer` that stores a `pContainer representative` on all locations where it will distribute its data. Each representative stores a subset of the data and there is no data replication across locations. The `pContainer` methods hide the distributed nature of a `pContainer` and allow users to access the data elements in a shared memory fashion. Users interact with a `pContainer` through a series of method invocations and corresponding responses. The memory consistency model is the set of guarantees provided to the users about the possible values returned by methods or when responses or acknowledgments that an operation is finished are received.

For the STAPL PCF, we considered a modular design where different memory consistency models can be employed by different `pContainers`. Similar to other `pContainer` properties this can be customized using the traits template arguments. In Section A we show general considerations for all `pContainer` methods and in Sections B and C we introduce the default memory consistency model provided by STAPL `pContainers`. In Section E we show how the default memory consistency model can be constrained or relaxed to implement other models.

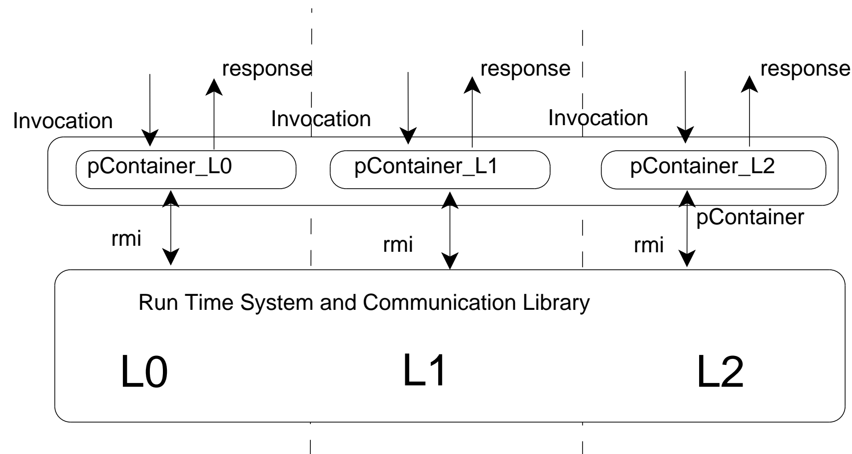


Fig. 18. User, pContainer, run time system interaction.

A. pContainer Interfaces

As described in Chapter V, Section B the `pContainer` interface contains the following categories of methods: collective (COLL), synchronous (`SM`), asynchronous (`AM`) and split phase (`SPM`). Each of these types of methods provides different guarantees about the completion of the invocations and ordering among them when invoked concurrently from multiple threads. The complete specification of the guarantees provided to the user is the `pContainer` memory consistency model (MCM).

To further motivate the need for a MCM for the `pContainers` we include in Figure 19 an example of a simple STAPL program to exemplify how users can rely on the MCM guarantees to reason about the correctness of their application. The simple program starts by instantiating a `pArray` container. This is a collective operation that will build a `pArray` representative on all locations. The acknowledgment that the constructor is finished is received when the constructor returns. Next we build a `pView` on the `pArray`'s data. Similarly, there is a `pView` representative on all

locations and the acknowledgment the `pView` is built is received when the `pView` constructor returns. Next, we spawn a number of parallel tasks using the `map_func` STAPL construct. The `map_func` takes as input a `pView` representing the data and a work function that is applied to data in the `pView`. For the `map_func` included in Figure 19 we assume the STAPL infrastructure creates a task for each location where the `pView` has a representative. The tasks will be scheduled and run in parallel by the STAPL RTS. The code that each task executes does not have to be the same across all tasks.

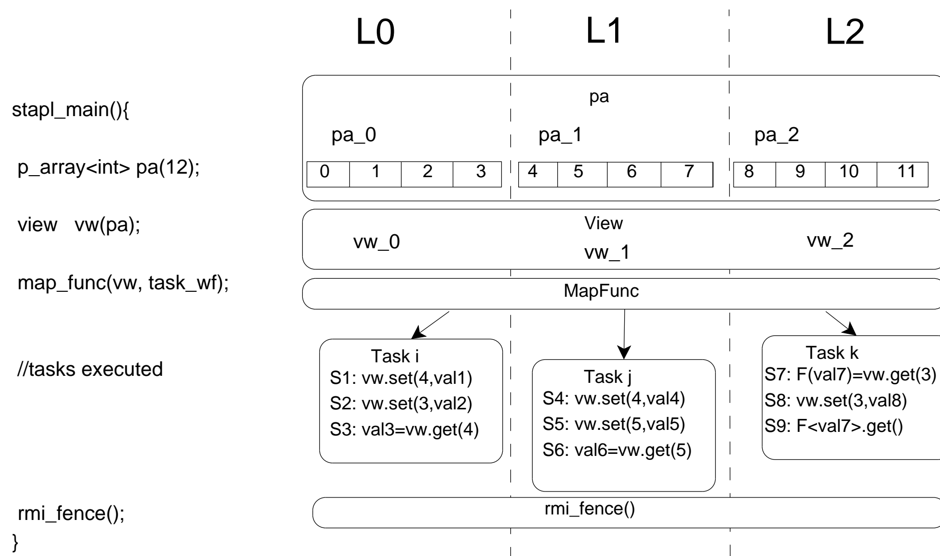


Fig. 19. STAPL program execution.

For the example in Figure 19 we assume all initial elements in the `pArray` are zero. When $Task_i$ executes $S1$ we notice that this can be executed concurrently with $S4$ from $Task_j$ and the question is what value the read of `pa[4]` in $S3$ may

return. The MCM describes the completion and ordering guarantees of operations that are concurrently executed by multiple threads of computation as we describe in the following sections. We come back to the example in Figure 19 in Section D.

B. Completion Guarantees

In Chapter III, Section B we introduced the RTS and described the `sync_rmi`, `async_rmi` and `rmi_fence` as the main communication and synchronization primitives. The `rmi_fence` construct is provided by the RTS to ensure that previously spawned messages are received on their target locations. There are currently two fence calls available in STAPL: `rmi_fence` collective call, where all threads in the execution group need to perform the invocation and one sided `os_fence` that can be invoked independently of the other threads of computation. From the individual thread's point of view, both fences ensure all acknowledgments for the pending operations are received. The global fence ensures the acknowledgments are globally received by all pending operations. The STAPL runtime provides additional primitives that subsume the fence semantic such as one-sided/collective reduce/broadcast.

We introduce in this section a terminology to specify the start and the termination of the `pContainer` methods and the specification of when `pContainer` methods are completed.

- Collective methods: start `CollM` and termination `ACK_CollM` (e.g., constructor). `ACK_CollM` is received by the invoking thread when the method returns.
- Synchronous methods: start `SM` and the termination is denoted as `ACK_SM` (e.g., `get_element`). For a `SM`(x), the `ACK_SM`(val) is received by the invoking thread when the method returns.

- Asynchronous methods: start `AM` and termination `ACK_AM` (e.g., `set_element`). For a `AM`(x) there is no explicit acknowledgment sent from the `pContainer` to the user. To reason about correctness we logically assume that the `ACK_AM` is received by the invoking thread at any point in the program order before any one of the following events:
  - Encountering a `fence` call
  - A subsequent `SM`(x) or `SPM`(x) receives its acknowledgment. Synchronous and split phase methods on an element x of a `pContainer` forces acknowledgments for all pending asynchronous methods operating on the same element x.
  - A subsequent `AM`(x) receives its acknowledgment. Asynchronous method invocations from the same thread and on the same element x of a `pContainer` receive their acknowledgments in order.
- Split phase methods: start `SPM` and termination `ACK_SPM` (e.g., split phase get element). The invocation of a split phase method returns immediately to the user a future. We denote the creation of the future as `SFuture`. The return value corresponding to the future is obtained by invoking the method `get` and we represent this as `Future.get`. The acknowledgment for the `Future.get` is denoted as `ACK_F` and it is received when the method returns. The acknowledgment of a split phase method and implicitly the value returned, is logically received by the invoking thread at any point in the program order after the future creation and before any one of the following events:
  - Encountering a `fence` call.
  - A subsequent `SM`(x), `AM`(x) or `SPM`(x) receives its acknowledgment.

– When `Future.get` receives its acknowledgment `ACK_F`.

The fence receives its acknowledgment when it returns together with the acknowledgments of all pending asynchronous and split phase methods. Our framework guarantees that for all method invocations there is an acknowledgment and this is an important property of a distributed system referred to as *liveness*[6].

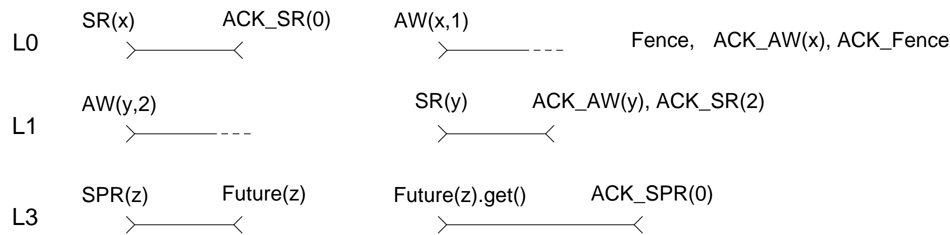


Fig. 20. Completion guarantees. The time increases from left to right.

In the following sections we use $\texttt{SR}_{L_i}(x)$ to denote the beginning of a synchronous method that will read an element x of a `pContainer`. We use $\texttt{ACK_SR}_{L_i}(val)$ to denote the acknowledgment and the returned value. The index L_i is a thread identifier and is used to distinguish among invocations in different threads. Similarly, for asynchronous write operations we use `AW` and `ACK_AW` and for split phase reads we use `SPR` and `ACK_SPR`. In Figure 20 we include a picture to exemplify the termination guarantees introduced in the previous section. We exemplify using asynchronous writes (AW), synchronous reads (SR), and split phase reads (SPR), but the same holds for any element-wise `pContainer` method. The example accesses elements corresponding to three different `pContainer` elements identified by their `GIDs`, x, y, z. *For all examples*

considered in this chapter the elements have all value 0 initially. Each row in the figure contains method invocations in program order from left to right. Each invocation takes a certain amount of time which is represented as the length of the segment between start and acknowledgment. Location L0 performs a synchronous read of element x. The invocation returns `ACK_SR`(0) when the method returns. The next AW operation receives the acknowledgment before the subsequent fence returns. Location L1 performs an AW on y followed by a SR on y. The SR operation since it is on y implies the `ACK_AW`(y) and the `ACK_SR`(2). Location L2 performs a split phase read of z. The invocation returns immediately to the user a future. When the get method of the future is invoked the `ACK_SPR`(2) is received.

C. Memory Consistency Conditions

In sequential programming, invocations are completed in the order in which they were issued in the program. In a parallel system it is often the case that this requirement is relaxed in order to provide improved performance.

1. pContainer Default Memory Consistency Model

Based on the termination guarantees introduced in the previous section we now describe the default memory consistency conditions of the `pContainer` methods. We first introduce a set of notations and rules and later in this section we show the interaction of this rules with a set of examples.

We adapted from [6] a set of notations to formally specify the `pContainer` memory consistency model. Let E be an execution of a program which has concurrent method invocations by multiple threads in multiple locations and on multiple `pContainer` elements. E is a sequence of method invocations as they occurred as

a result of interleavings of the actions of all the threads in the system. E can be thought as a trace of a particular execution of a program. We use the notation $E|i$ to denote the subsequence of E consisting of all invocations performed by and responses received by a thread i. Similarly we use $E|x$ to indicate the subsequence of E consisting of all invocations and responses that are performed on an element x of the `pContainer`.

To simplify reasoning about the possible method interleavings and values returned by an execution E, we introduce the notion of a permutation of the method invocations as a linear sequence of all method invocations in the system. The MCM specifies the restrictions on the possible permutations corresponding to a particular execution E. Fences serve as global synchronization points that force the completion of all previous `pContainer` methods. In the following we discuss the guarantees for the method invocations between fences.

The `pContainer` MCM: For an execution E, a `pContainer` guarantees that there is a permutation P of all method invocations in E such that:

1. The methods in P occur sequentially (no overlapping).

2. For each element x, the restriction of P to just those methods on x, denoted $P|x$, satisfies the specification of the data type of x. (E.g., if x is a register that supports `Read` and `Write`, then each `Read` returns the value of the latest preceding `Write` invoked on x.)

3. For each thread i, the restriction of E to just the *Coll* and *Synch* methods invoked by i, denoted $E|(Coll \cup Synch)|i$, must equal $P|(Coll \cup Synch)|i$. That is, the permutation P has all the collective and synchronous methods by i in the same order as they were invoked. However, no guarantee is given as to how Synch methods at different locations are ordered in P.

4. For each element x and each thread i, the restriction of P to the methods on x invoked by i, denoted $P|x|i$, consists of all the *Synch*, *Asynch*, and *Split Phase* methods on x invoked by i in E, in the order of their invocation.

5. Consider any element x and let O_i and O_j be two operations on x in E such that O_i is invoked by some thread i, O_j is invoked by some other thread j, and O_i completes (i.e., receives its ACK) before O_j is invoked. Then O_i is ordered in P before O_j.

In the remaining of this section we exemplify some of the ordering relations for `pContainer` methods as derived from the consistency conditions previously introduced.

For asynchronous methods the PCF guarantees that subsequent invocations from the same thread affecting the same element x will receive their implicit acknowledgments in the order in which they were invoked (condition 4). For example, let us assume a thread in location $L0$ performs the sequence of asynchronous write invocations depicted in Figure 21(a). The acknowledgment for two consecutive writes on x ($\mathtt{AW}^0(x), \mathtt{AW}^1(x)$) are guaranteed to be in the order in which they were issued. The superscript is used to distinguish subsequent invocations of the same type on the same element/location. The acknowledgment for the write to y has no relationship with the acknowledgment for x. Figure 21(a) shows three possible valid interleavings with the acknowledgment for y received in an arbitrary order relative to the acknowledgments for writes to x. Corresponding to Figure 21(a) we include in Figure 21(b) three possible valid interleavings as perceived by the user.

When invoking methods concurrently on the same `pContainer` element from threads in different locations there is no guarantee about the order in which they will terminate. Let us assume the following method invocations from two different

$$L0 : \texttt{AW}^0(x,1), \texttt{AW}(y,1), \texttt{AW}^1(x,2)\ fence\ \texttt{ACK_AW}^0(x), \texttt{ACK_AW}^1(x), \texttt{ACK_AW}(y)$$

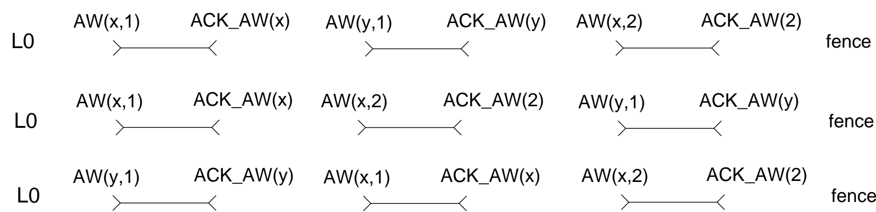
or

$$L0 : \texttt{AW}^0(x,1), \texttt{AW}(y,1), \texttt{AW}^1(x,2)\ fence\ \texttt{ACK_AW}^0(x), \texttt{ACK_AW}(y), \texttt{ACK_AW}^1(x)$$

or

$$L0 : \texttt{AW}^0(x,1), \texttt{AW}(y,1), \texttt{AW}^1(x,2)\ fence\ \texttt{ACK_AW}(y), \texttt{ACK_AW}^0(x), \texttt{ACK_AW}^1(x)$$

(a) Asynchronous method relative execution order



(b) Interleavings as perceived by user

Fig. 21. Asynchronous methods ordering. (a) Relative order for acknowledgments. The superscript is used to distinguish subsequent invocations of the same type on the same element/location (e.g., two writes or reads of the same variable). The *or* is used to denote that any interleaving is a valid one (b) Possible interleavings.

locations:

$$Li : \texttt{AW}(x,1), \texttt{SR}^0(x), \texttt{ACK_SR}(1 or 2)\texttt{ACK_AW}(x) fence\ \texttt{SR}^1(x), \texttt{ACK_SR}(a)$$

$$Lj : \texttt{AW}(x,2), \texttt{SR}^0(x), \texttt{ACK_SR}(1 or 2)\texttt{ACK_AW}(x) fence\ \texttt{SR}^1(x), \texttt{ACK_SR}(a)$$

The first read invocations (SR^0) on both locations do not have a deterministic result. It can be either 1 or 2 and the result can be different on the two locations. After the fence, it is guaranteed that both reads (SR^1) will return the same value a, though is not known if it is 1 or 2. Assuming the element x was initially zero it is guaranteed that none of the reads in the example will return 0 because of the ordering guarantees provided for accesses to the same element in a thread. The reads are always executed after the previous writes in the program order.

D. Memory Consistency Example

In Figure 19 we show an example of a simple STAPL program to exemplify how users can rely on the completion and ordering guarantees introduced in the previous sections to reason about the correctness of their application. For $Task_i$ invocation $S1$ can be executed concurrently with $S4$ from $Task_j$. Thus the read in $S3$ is not deterministic. The answer can be either $val1$ or $val4$. The read from $S6$ is deterministic and it returns $val5$. Invocations $S7$ and $S8$ respect the program order so the read in $S9$ will return zero. After the `map_func` construct finishes it is guaranteed that a read of an element of the `pArray` will return the same value on all threads provided there is no other concurrent write in the system.

The example in Figure 19 is for illustration purposes only to describe various possible interleavings. In practice individual tasks access an exclusive set of elements from a `pContainer` which simplifies the reasoning about the correctness of the application.

E. Other Memory Consistency Models

The default STAPL `pContainer` MCM is relaxed and similar to *weak consistency (WC)*[3, 23] discussed in the context of shared memory architectures. Under WC model there are regular memory accesses (method invocations in our case) and synchronization operations. Reordering of the operations is allowed in between synchronization points and no operation is allowed to be reordered relative to synchronization operations. As described in Sections B and C, `pContainers` provide additional guarantees about the ordering of the methods affecting the same data element that make the model stronger than weak consistency.

In this section we show that the default `pContainer` MCM is more relaxed than *sequential consistency (SC)* or *processor consistency (PC)* and describe how individual `pContainers` can restrict their interfaces to provide the requirements of these stricter models. Additionally, we show how we can further relax the ordering of the methods on the same data element to further relax the default MCM.

1. The Default pContainer MCM is not Sequentially Consistent

A natural memory consistency model for a parallel system is sequential consistency as introduced by Lamport[46].

Definition 13. A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Using our formalism, an execution E is *sequentially consistent* if there exists a permutation P of the operations in E such that

$$Li : \texttt{AW}(x,1), \texttt{AW}(y,1), \texttt{ACK_AW}(y), \texttt{SR}(y), \texttt{ACK_SR}(1) fence\ \texttt{ACK_AW}(x)$$

(a)

$$L1 : \texttt{AW}(f1,1), \texttt{SR}(f2), \texttt{ACK_SR}(0), fence, \texttt{ACK_AW}(f1)$$

$$L2 : \texttt{AW}(f2,1), \texttt{SR}(f1), \texttt{ACK_SR}(0), fence, \texttt{ACK_AW}(f2)$$

(b)

Fig. 22. Relaxed completion order: (a) Operations on different `pContainer` elements receive their acknowledgments out of order (b) Dekker's mutual exclusion.

1. For every element x, the restriction $P|x$ satisfies the specification of the data type of x.

2. If the acknowledgment for operation o_1 at thread i occurs in E before the invocation for operation o_2 at thread i, then o_1 appears before o_2 in P (e.g., $E|i = P|i$, for all threads i).

In STAPL when both synchronous and asynchronous methods are invoked from the same thread on a given location Li on different elements, then there is no guarantee about the order in which these operations will be executed. In the example in Figure 22(a) we show that even though the read and write on the variable y is after the asynchronous write to x in the program order, the operations on y may finish before the write operation on x finishes.

This relaxed order for methods operating on different elements breaks the sequential consistency semantic. In Figure 22 (b) we show the Dekker mutual exclusion

algorithm described in various memory consistency model papers [1, 2, 3]. The flags $f1$ and $f2$ are two elements in a `pArray` both of which are initially zero. Dekker's exclusion algorithm guarantees that in a SC system the reads for flags should not return both zero. Using the `pArray` methods however it is possible that both reads for the flags return zero due to the relaxation described in the previous paragraph.

Claim 1 : When the `pContainer` interface includes `SM` and `AM` methods, concurrent invocations of them on different locations do not satisfy sequential consistency.

2. The Default pContainer MCM is not Processor Consistent

Processor consistency[1, 2] guarantees that writes from a thread are seen by all other threads in the order in which they were issued. This is not guaranteed by STAPL for example in the case where the writes are on two different elements. In this situation, threads on other locations may perceive the writes on the two distinct variables in different orders. In Figure 23 we depict how threads on different locations may see the effects of two writes from location $L0$. Location $L3$ may see the element y being written before element x is written. This would violate the processor consistency assumptions.

Claim 2 : When the `pContainer` interface includes `SM` and `AM` methods, concurrent invocations of them on different locations break processor consistency.

3. Modifying the Default pContainer MCM

Claim 3 : When the `pContainer` interface includes synchronous methods only (`SM`), concurrent invocations of them on different locations satisfy sequential consistency.

With this constrained interface no `pContainer` methods are allowed to be reordered. Each method receives its acknowledgment before the next instruction in the program order is executed. This restriction, coupled with the fact that each

$$L0 : \texttt{AW}(x,1), \texttt{AW}(y,2)......................$$

$$L1 : \texttt{SR}(x)\texttt{ACK_SR}(1), \texttt{SR}(y)\texttt{ACK_SR}(2)$$

$$L2 : \texttt{SR}(x)\texttt{ACK_SR}(0), \texttt{SR}(y)\texttt{ACK_SR}(2)$$

$$L3 : \texttt{SR}(y)\texttt{ACK_SR}(2), \texttt{SR}(x)\texttt{ACK_SR}(0)$$

Fig. 23. Processor consistency counter example.

`pContainer` method is executed in an atomic manner, provides the necessary conditions for a SC model.

Claim 4 : The `pContainer` can relax its ordering constraints for methods operating on the same memory element by allowing them to proceed in an arbitrary order. With this relaxation the following interleaving is possible:

$$Li : \texttt{AW}(x,1), \texttt{SR}^0(x), \texttt{ACK_SR}(0) fence\ \texttt{SR}^1(x), \texttt{ACK_SR}(1)$$

Above, the synchronous read of element x follows an asynchronous write but it does not return the value 1 that was previously written in the program order. After the fence, the result of the write becomes visible and the second read ($\texttt{SR}^1$) returns the value 1.

The framework currently provides a default memory consistency model introduced in this chapter. Other more relaxed memory consistency models are possible but their impact on performance needs to be carefully analyzed and judged against

the complexity it may bring to the user code when reasoning about correctness. Supporting stricter memory consistency models for element-wise methods is straight forward and it requires all methods to be synchronous. Applications where this will be a benefit need to be identified and analyzed.

F. pContainer Method: Developer Side

The `pContainer` developer expresses a method as a composition of invocations on data that may reside on different locations. When implementing a `pContainer` method the `pContainer` first decides if the element is local or not. If it is local, then the operation is performed atomically on the corresponding `bContainer`. Otherwise the operation is requested to be executed on a remote location. The remote location can in turn forward the execution to alternative locations in a recursive manner. In the following, we describe the semantics of the main `pContainer` methods in terms of requests and acknowledgments. We provide examples for read, write and split phase read operations, but the specification is the same for any element-wise `pContainer` method (synch, async or split phase).

Synchronous Reads (SR): A synchronous operation `SR` can be described as:

$$
\begin{aligned}
\texttt{SR}_{L_i}(x), \texttt{ACK_SR}_{L_i}(val) \quad \equiv \quad & [\texttt{SR}_{L_i}(mD(x))] //read\ metadata\ locally \\
& (\\
& //if\ x\ is\ in\ local\ bContainer \\
& \texttt{ACK_SR}_{L_i}(bCont_k), [\texttt{SR}_{L_i}(bCont_k, x)], \texttt{ACK_SR}_{L_i}(val) \\
& //else\ x\ lives\ on\ location\ L_j(forwarding) \\
& \texttt{ACK_SR}_{L_i}(L_j), \texttt{SR}_{L_j}(x), \texttt{ACK_SR}_{L_j}(val)\texttt{ACK_SR}_{L_i}(val) \\
&)
\end{aligned}
$$

where a [...] denotes an atomic execution, L_i location i, $mD(x)$ the set of memory addresses accessed when requesting information about element x, $bCont_k$ `bContainer` owning the element x, $(bCont_k, x)$ the set of memory addresses touched by the current operation on the `bContainer`. Everything following // is a comment.

We observe that an SR operation receives an acknowledgment before the method is finished.

Asynchronous writes (AW): An AW method is modeled similarly to a SR method except that the acknowledgment is not immediately received on the location that initiates the method. This is described next:

$$
\begin{aligned}
\mathtt{AW}_{L_i}(x) \;\equiv\; & [\mathtt{SR}_{L_i}(mD(x))] \\
& (\\
& //if\ x\ is\ in\ local\ bContainer \\
& \mathtt{ACK_SR}_{L_i}(bCont_k), [\mathtt{SR}_{L_i}(bCont_k, x)] \\
& //else\ x\ lives\ on\ location\ L_j(forwarding) \\
& \mathtt{ACK_SR}_{L_i}(L_j), \mathtt{AW}_{L_j}(x) \\
&)
\end{aligned}
$$

The acknowledgment that an $AW(x)$ operation is finished is received when encountering:

- A `fence` call.
- When a subsequent SR(x) or future(x).get() receives its acknowledgment.

Split phase reads (SPR): A SPR method is modeled similar to SR except that the acknowledgment is not immediately received on the location that initiates the method. This is described next:

$$
\begin{aligned}
\mathtt{SPR}_{L_i}(x), Future_{L_i}(x, val) \;&\equiv\; [\mathtt{SR}_{L_i}(mD(x))] \\
&\quad (\\
&\quad \mathtt{ACK_SR}_{L_i}(bCont_k), [\mathtt{SR}_{L_i}(bCont_k, x)] \\
&\quad or \\
&\quad \mathtt{ACK_SR}_{L_i}(L_j), \mathtt{SPR}_{L_j}(x) \\
&\quad) \\
Future_{L_i}(x, val).get() \;&\equiv\; \mathtt{ACK_SPR}_{L_i}(x, val)
\end{aligned}
$$

The $Future_{L_i}(x, val)$ is a handle that can be queried later for the acknowledgment of the SPR operation. The acknowledgment that an $SPR(x)$ operation is finished is received when encountering:

- A `fence` call.
- When a subsequent SR(x) receives its acknowledgment.
- When `get()` method of the future returned by $SPR(x)$ is invoked.

G. Consistency of Other pContainer Methods

In this section we discuss the semantics of `pContainer` methods that do not process individual elements. These are collective operations (e.g., constructors, destructors, etc.) and methods that return global properties of the data structure (e.g., `size()`, `empty`, etc.).

Methods such as `size()` or `empty()` for dynamic `pContainers` require information about the entire `pContainer` globally. When designing these methods, we

considered the following alternatives: (1) a one sided reduction across all locations to perform the accumulation of all local sizes. (2) maintaining a data member `size` on all locations. Both these options are prohibitively expensive in terms of communication traffic generated. The solution we decided to support as part of our default implementation is to have the size stored in a replicated fashion across all locations but to update it in a lazy fashion. Dynamic operations that modify the `pContainer` through inserts and deletes make the content of the size variable obsolete. The `pContainer` re-synchronizes the size data member upon the invocation of the `post_execute()` method. This is a collective call that guarantees, when finished, that `pContainer` size is properly reflected across all locations. The insertion of the synchronization points and calls to `post_execute()` is simplified by STAPL as described in the next section.

H. Enforcing Synchronization Points Automatically

We mentioned previously that acknowledgments for asynchronous methods invoked on a `pContainer` are not received until a synchronization point is reached. To simplify the user's effort in controlling synchronization points in STAPL, the `pViews` and the executor introduce them automatically while executing the tasks corresponding to a given computation. As introduced in Chapter III a computation is represented in STAPL as a collection of tasks and eventual dependencies between them. The `pViews` are used to represent the data stored in a `pContainer` and all method invocations on the container are done through `pViews`.

At the end of a task execution, the STAPL executor invokes a `fence` thus ensuring that all acknowledgments for all asynchronous methods are received before executing the next available task in the TDG.

Additionally when all the tasks of a particular computation are finished, the `post_execute()` method of the `pView` is invoked by the executor. As part of this phase, the `pViews` can commit pending operations and re-synchronize the `pContainer` to reflect the changes done through methods in the tasks. Currently, in the `post_execute()` phase, `pViews` of dynamic `pContainers` update their size data member.

CHAPTER VIII

PCONTAINER PERFORMANCE EVALUATION

In this chapter, we describe the methodology used to evaluate the performance of representative `pContainers` developed using the PCF: `pArray`, `pList`, `pMatrix`, `pGraph` `pHashMap` and composed `pContainers`. For all these data structures, we will look at the performance of individual methods using artificial kernels, generic algorithms and real applications.

A. Experimental Setup

We conducted our experimental studies on various parallel machines comprising various processor architectures and network interconnects. This includes a 38,288 core Cray XT4 (CRAY4), a 5312 core Cray XT5 (crayh), both available at NERSC, and a 832 core Power5 Cluster (P5-CLUSTER) available at Texas A&M University. The CRAY4 has 9,572 compute nodes each with a quad core Opteron running at 2.3 GHz and a total of 8 GB of memory (2 GB of memory per core). The compute nodes are connected to a dedicated SeaStar2 router through Hypertransport with a 3D torus topology which ensures low-latency, high bandwidth communication. The CRAY5 has 664 compute nodes, each containing two 2.4 GHz AMD Opteron quad-core processors (5,312 total cores). The P5-CLUSTER is a 832 processor IBM cluster with p575 SMP nodes and 16 cores per node. In all experiments, a location contains a single processor core, and the terms can be used interchangeably.

```
1   evaluate_performance(N,P)
2      tm = stapl::start_timer();
   //start timer
3      //insert N/P elements concurrently
4      for(i=0; i<N/P; ++i)
5         pcontainer.method(arguments);
6         //pcontainer can be any of the PCF containers
7         //and method any of the element wised method
8         //of the pcontainer;
9      rmi_fence(); //ensure all inserts are finished
10     elapsed = stapl::stop_timer(tm); //stop the timer
11     - Reduce elapsed times, getting the max time
12        from all processors.
13     - Report the max time
```

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

B. Evaluation of pContainer Methods

To evaluate the scalability of individual methods we designed the kernel shown in Figure 24. The figure shows a generic method being invoked, and the same kernel is used to evaluate all methods. For a given number of elements N, all P available processors (locations) concurrently perform N/P method invocations. We report the time taken to perform all N methods globally. The measured time includes the cost of a fence call which, as stated in Chapter III, is more than a simple barrier. Each experiment is executed 32 times on P5-CLUSTER and CRAY4, and 10 times on CRAY4 due to large scale experiments performed there and limited execution time available on the machine. The times reported in the graphs in the following chapters are average times with confidence intervals. Due to the stability of the machine the confidence intervals are small for certain experiments and are not always visible in the graphs.

C. Evaluation of Generic Algorithms

We evaluated the performance of generic non-mutating `pAlgorithms`, `p_generate`, `p_for_each` and `p_accumulate`, when applied to data stored in various different STAPL `pContainers`. For all the algorithms considered in this section, for all platforms, 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. Additionally, when scalling to very large number of processors the problem would not fit in memory for lower processor counts.

The `p_generate` algorithm takes as argument a functor that generates a random number and a `pView`. It assigns a random number to every element in the `pView`. The `p_for_each` algorithm increments the elements of the container with a given constant performing read, add and store operations on each individual element without the need of any remote accesses. The `p_accumulate` accumulates all the elements in the container using a generic map reduce operation available in STAPL. These three algorithms are representative of a large class of algorithms that are either map or map reduce patterns.

D. Specific Applications

For various `pContainers` we look at the performance in the context of more complex applications. These are examples of how programmers may use `pContainers` in applications.

CHAPTER IX

THE STAPL PARRAY

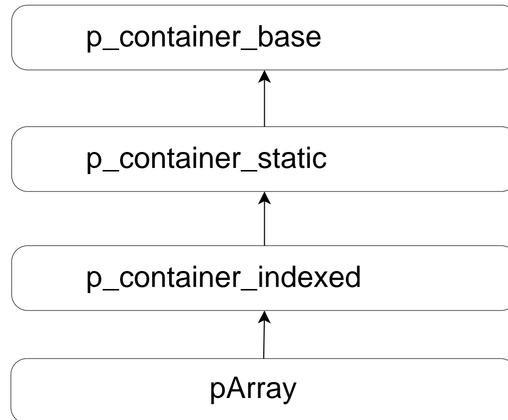


Fig. 25. Derivation chain for `pArray`.

The STL `valarray` container is a fixed size data structure optimized for storing and accessing data based on one dimensional indices and iterators. The STAPL `pArray` is the parallel equivalent of the STL `valarray`, providing an efficient interface to access data elements using indices and `pViews`. An important property of the `pArray` is that it is a static data structure, i.e., the number of elements is known at instantiation and doesn't change during execution. As described in Chapter V, Section A, this enables a number of optimizations such as closed form solutions for partitions and partition mappings. In Figure 25, we show the derivation chain for the `pArray` container. Correspondingly, the `pArray` inherits the interfaces and the default functionality provided by all its base classes. In Chapter V, Section E, we introduced the

default pArray specification for the main functional modules. In this section we show a simple example of pArray usage, provide the user interface and discuss experimental results.

A. Example

```
 1 #include <p_array.h>
 2
 3 void stapl_main(){
 4   p_array<int> pa(100); // parray with 100 elements -
 5                          // default partition
 6   partition_blocked<int> pbl(10);
 7   p_array<int> pa_bal(100, pbl); // parray with 100 elements -
 8                                   // balanced partition
 9   array_1D_pview<p_array<int> > pa_view(pa);
10   p_generate(pa_view, rand());
11 }
```

Fig. 26. pArray example.

In Figure 26 we show a simple example of pArray usage. The program declares a pArray of 100 integers (Figure 26, Line 4) and another pArray of 100 integers with a blocked partition with blocks of size 10 (Figure 26, Line 7). A pView is declared next over a pArray (Figure 26, Line 9) and a generic algorithm is invoked over the data of the pArray(Figure 26, Line 10).

B. The pArray Specification

The `pArray` template declaration is :

```
template<class T, class Partition=Default , class Traits=Default>
class p_array ;
```

We include in Table XIX the complete interface of the `pArray`.

Table XIX.: `pArray` interface.

Template Arguments	**Description**
`T`	The array's value type: the type of object that is stored in the array
`Partition`	Partition used to define the blocks which the `pArray` is divided
`Traits`	`pArray` traits for specifying the low level base container used and distribution features
Define Type	**Description**
`value_type`	The type of stored objects.
`index_type`	The type of the indices of `pArray`
Method	**Description**
`p_array()`	Default constructor. $O(log(P))$
`p_array(size_t m)`	Create a p_array of m elements. $O(m/P + log(P))$
`p_array(size_t m, value_type _val)`	Create a p_array of m elements and initialize the elements to `_val`. $O(m/P + log(P))$
`p_array(const p_array& other)`	Copy constructor. $O(size(other)/P + log(P))$
`template <class PS>` `p_array(size_t m, const PS&)`	Construct a p_array of a given size and using the specified partition. $O(m/P + log(P))$
`void set_element ( index_type i, const value_type& v)`	Set the element of index `i` to the value `v`. $O(1)$
`value_type get_element ( index_type i) const`	Return the value corresponding to index `i`. $O(1)$

Table XIX continued

Method	**Description**
pc_future <value_type> get_element_split (const index_type& i) const	Returns immediately (non blocking) a future for the value corresponding to index i. The future will return the actual value when queried using the corresponding get() method. $O(1)$
template<class Functor> typename Functor::result_type apply_get (index_type i, Functor f)	Apply a functor f to the data corresponding to index i. The returned value is the value returned by functor f. $O(1)$
template<class Functor> void apply_set (index_type i, Functor f);	Apply a functor f to the data corresponding to index i. $O(1)$
p_container_indexed_ref operator[] (index_type i)	Access operator that returns a reference to element i. $O(1)$
template <class NPS> void set_partition(const NPS& ps)	Reset the partition of the pArray to be ps. $O(size/P + log(P))$
bool is_local(index_type i)	Return true if the element with index i is stored locally, false if it is remote. $O(1)$

C. pArray Partitions

The pArray partitions model the partition_indexed concept. Users can provide their own partitions or select among the ones included in the table XX.

Table XX.: pArray partitions.

Partition Type	Description
partition_balanced<T>	Partition the pArray into evenly sized ranges; T is the p_array value type
partition_blocked<T> (size_t bs)	Split the pArray of size n into ceiling(n/bs)-1 blocks of size bs, the last block will have size n-(ceiling(n/bs)-1)*bs

Table XX continued

Partition Type	Description
partition_blocked _explicit<T> (const std::vector<size_t> sizes)	Partition a pArray into blocks whose sizes are defined in the vector sizes (the sum of the sizes have to be equal to the size of the pArray

D. pArray Customization

Low level details about a pArray data structure are customized using the traits mechanism. In Table XXI we describe the modules that are customizable.

Table XXI.: pArray traits.

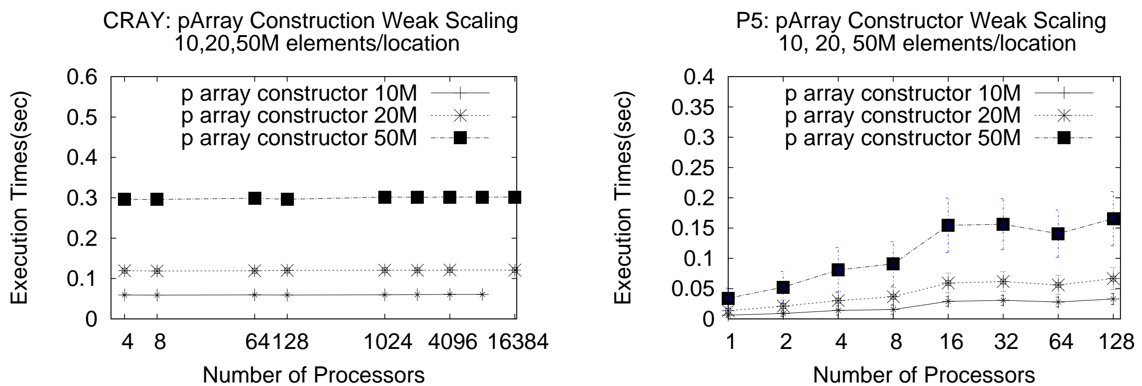
bContainer	pArray storage
partition_type	pContainer partition
mapper_type	Class to specify the mapping of the bContainers into locations
distribution_type	Data distribution manager
iterators_type	Type of iterators exported by pContainer
loc_dist_metadata	Class used to specify pContainer coarsening information

The partition type can be any of the ones described in the Table XX. The partition mapper type can be any of the partition_mapper_generic, partition_mapper blocked, partition_mapper_cyclic or partition_mapper_identity. iterators_type is a struct defining the iterator, const iterator, reference and const reference (global iterator and reference). The default implementation for pArray iterators

is the `pc_iterators` class that provides a generic random access iterator. The `loc_dist_metadata` is required by `pView` to extract the `pContainer` locality metadata. The default provided implementation for this module is a class that specifies to the `pView` to create a sub view for each `bContainer`.

E. Performance Evaluation

We evaluate the performance of the `pArray` parallel methods and STL algorithms operating on data stored inside `pArrays`. To evaluate the performance of the methods, we use weak scaling experiments according to the methodology described in Chapter VIII.



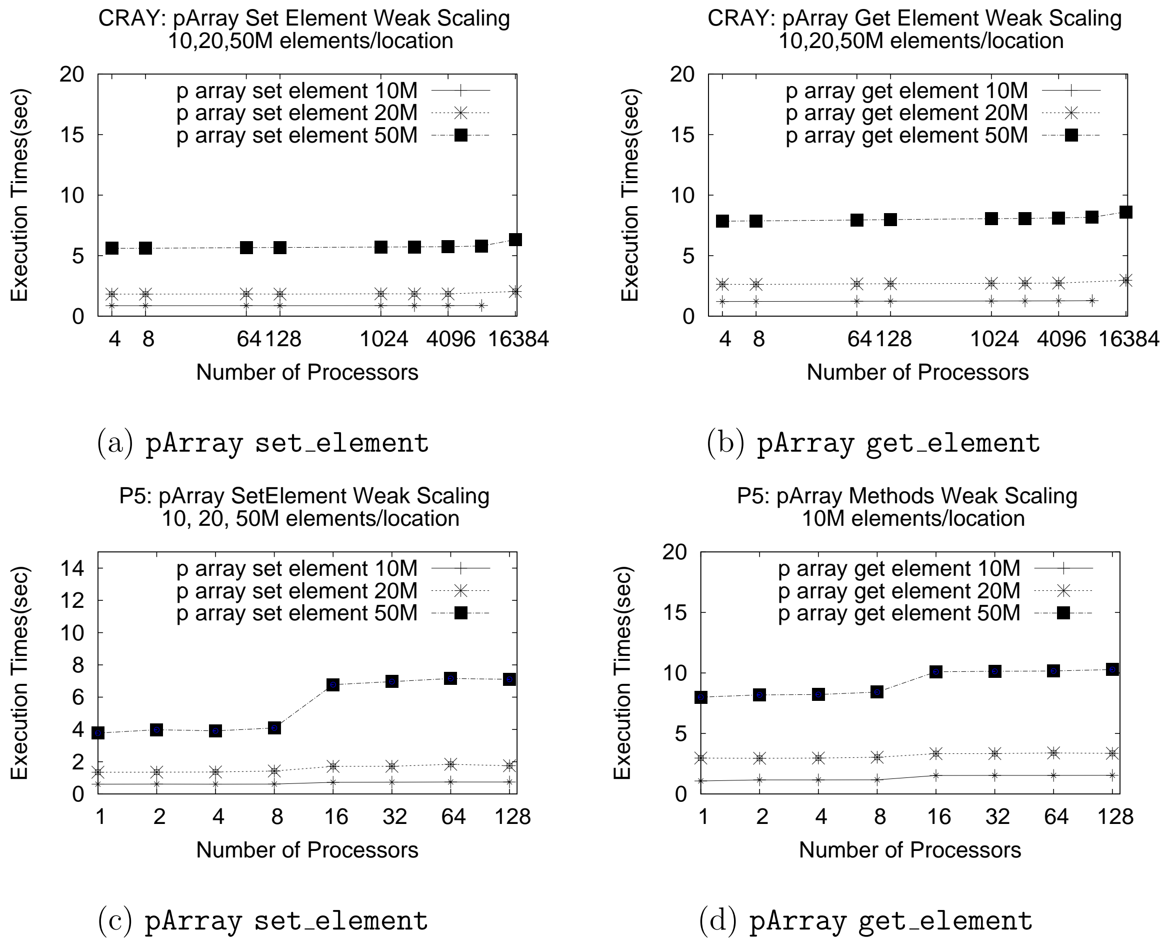
(a) `pArray` constructor on CRAY4 (b) `pArray` constructor on P5-CLUSTER

Fig. 27. `pArray` constructor execution time for various input sizes on (a) CRAY4 and (b) P5-CLUSTER.

1. Methods

In this section we analyze the performance of `pArray` collective operations (e.g., constructors) and representative element-wise methods `set_element`, `get_element` and `split_phase_get_element`. In a first experiment included in Figure 27 we show the performance of the `pArray` constructor on two different parallel architectures for 10, 20 and 50 million elements per location. The `pArray` constructor is a fully parallel operation with none of its modules requiring global synchronizations. On CRAY4 (Figure 27(a)) we observe very good scaling from 4 processors (the size of a compute node) up to 16384. When using 50 M elements per location the total size of the allocated `pArray` on 16384 processors is 819.2 billion requiring 1.19 terabytes of storage. On P5-CLUSTER (Figure 27(b)) we observe the performance slowly degrading as we increase the number of processors from 1 to 16, which uses up a shared memory node of the cluster. From 16 processors up the scaling improves. This is a well known behavior on this architecture and it is due to memory bandwidth limitations within a shared memory node as we use an increasing number of processors.

In Figure 28, we show the scalability of `pArray` `set_element` and `get_element` for various input sizes and numbers of invocations. The number of method invocations is the same as the input size N and each location performs concurrently N/P invocations, where P is the number of locations. The times in this figure are for the case when all invocations are executed locally. In this situation, both methods scale well as they don't involve any additional communication. In Figure 29, we show in the same plot the `pArray` constructor, `set_element` and `get_element`, respectively, to see the relative performance difference of these methods. `set_element` is an asynchronous method with no return type, while the `get_element` returns the value read. For this reason, the `set_element` is faster than `get_element` even though all invocations are



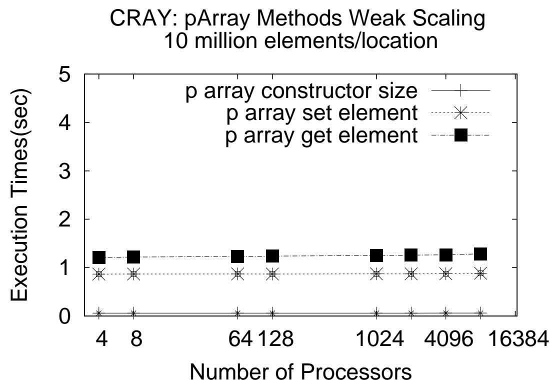
(a) `pArray` `set_element` (b) `pArray` `get_element`

(c) `pArray` `set_element` (d) `pArray` `get_element`

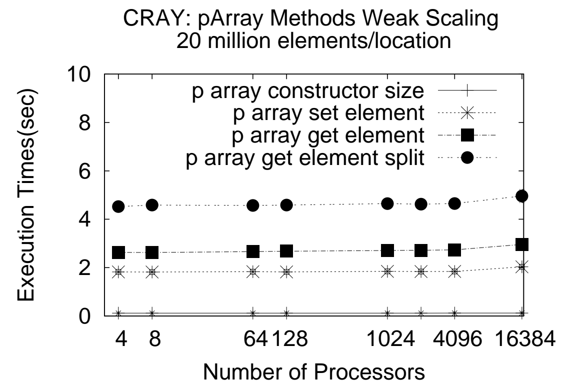
Fig. 28. CRAY4: `pArray` local method invocations for various container sizes. The number of invocations is the same as the `pArray` size (a) `set_element` (b) `get_element`.

local.

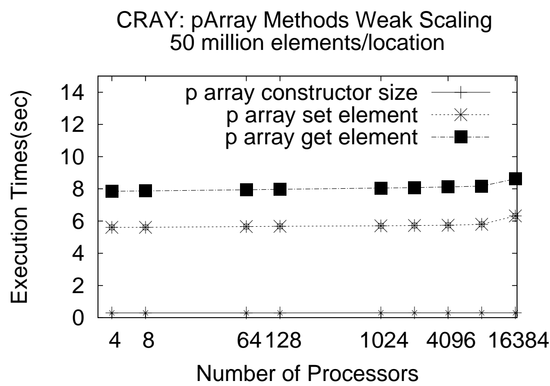
Split phase method study: In this experiment we study the performance of the `pArray` methods when there are 1% remote accesses. The results are included in Figure 30. We observe good scalability with only a 5.8% execution time increase for the asynchronous invocations as we scale the number of processors from



(a) 10M elements per location



(b) 20M Elements per location



(b) 50M Elements per location

Fig. 29. CRAY4: `pArray` methods for various input sizes. All invocations are local.

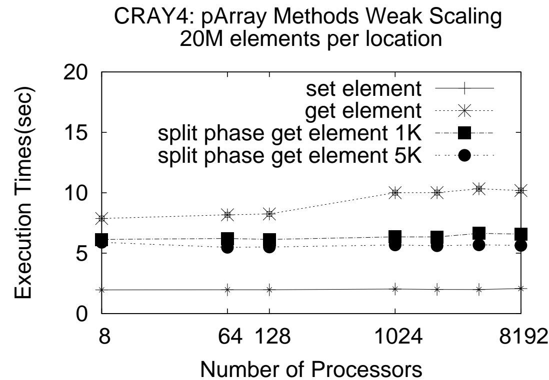
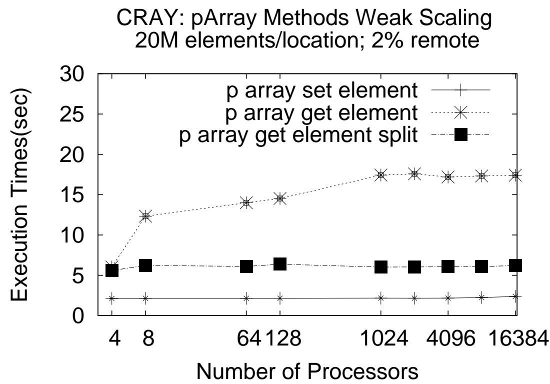


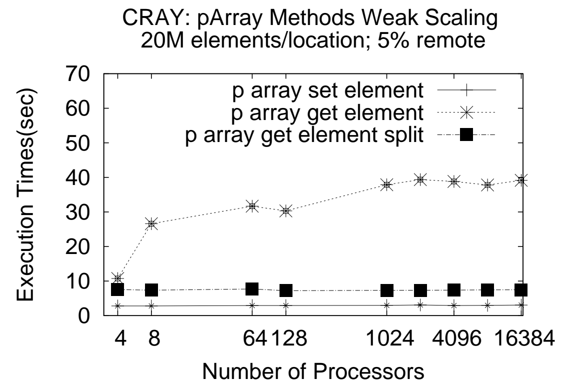
Fig. 30. CRAY4: `pArray` methods `set_element, get_element` and split phase `get_element`. 20M method invocations per location with 1% remote.

8 to 8192. For the synchronous methods the execution time increases 29%. For the `split_phase_get_element` we performed two experiments where we invoke groups of 1000 or 5000 split phase operations before waiting for them to complete. The split phase methods have an inherent overhead for allocating the futures on the heap but they do enable improved performance and scalability relative to the synchronous methods. Split phase execution enables the aggregation of the requests in the runtime as well as allowing communication computation overlap. For the `split_phase_get_element` the overall execution time increases 7.1% and 4.5% when 1000 and 5000 invocations, respectively, are started before waiting the result.

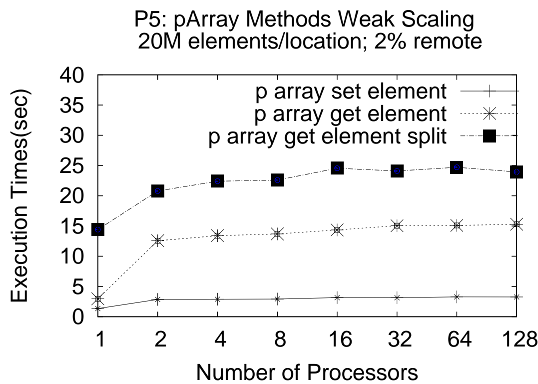
Remote methods study: In this experiment we study the performance of the `pArray` element-wise methods for various percentages of remote invocations. The results for the two architectures considered are included in Figure 31. The percent of remote method invocations is either 2% or 5%. As we increase the number of methods that are executed remotely, the execution time for all methods increases correspondingly. The `set_element` scales well on both architectures as we increase the



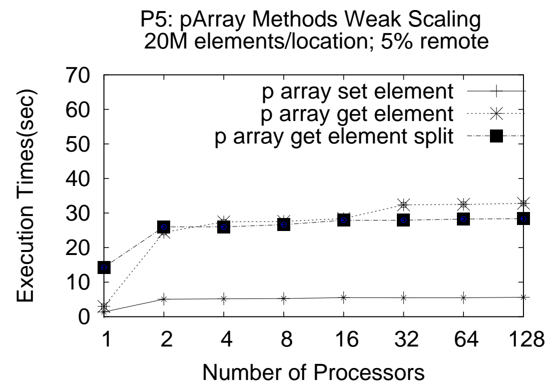
(a) 2% remote



(b) 5% remote



(c) 2% remote



(d) 5% remote

Fig. 31. `pArray` methods for various percentage of remote invocations. Execution times for (a)(b) CRAY4 and (c)(d) P5-CLUSTER.

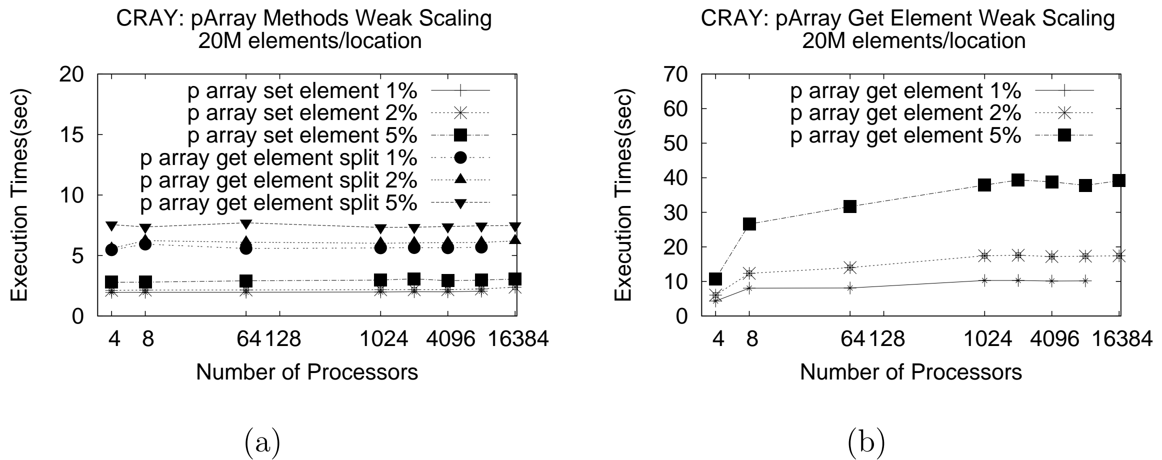


Fig. 32. CRAY4: `pArray` local and remote method invocations for various container sizes. The number of invocations is the same as the `pArray` size. The number of remote invocations is varied from 1% to 5% (a) `set_element` and `split_phase_get_element` (b) `get_element`.

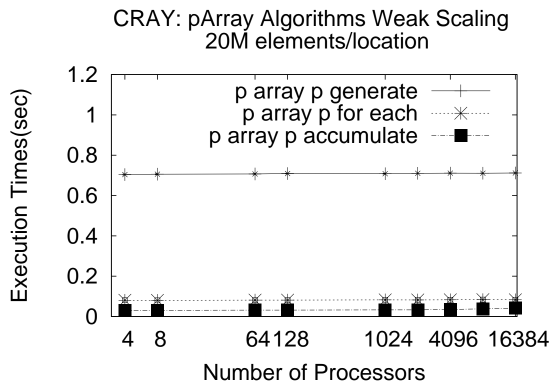
number of processors. For the `split_phase_get_element` we aggregate 5000 methods before waiting for their completion. On CRAY4 this operation scales well and overall is faster that the synchronous `get_element`. On the P5-CLUSTER architecture, the overhead of allocating the futures for the return values is bigger than on CRAY4 and on this platform the benefits of split phase execution are visible only when the number of remote methods is 5%. Overall, we observe that the asynchronous and split phase methods scale better than the synchronous methods but all methods considered scale well up to 16384 processors.

In Figure 32, we show for CRAY4 the impact of increasing the remote method invocations for individual methods. We observe that the runtime increases for all methods. The `set_element` and `split_phase_get_element` scale well while increasing the number of processors while the `get_element` performs slightly worse.

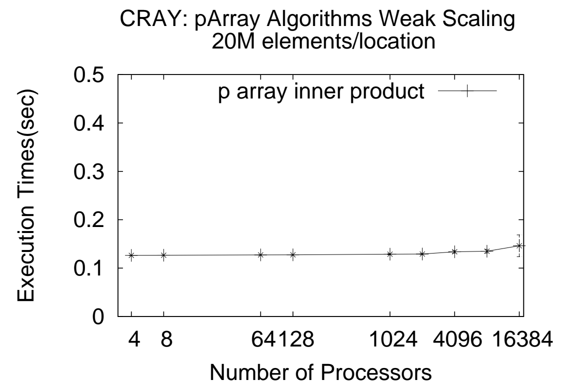
2. Algorithms

In Figure 33(a), we show the execution times for the `pAlgorithms` `p_generate`, `p_for_each` and `p_accumulate` on `pArray`. The figure shows a weak scaling experiment with 20M elements per processor. From the plot we observe that for the `pArray` the performance degradation is 5% when scaling from 128 to 16,385 processors for `p_generate` and `p_for_each`. For `p_accumulate`, which performs a global reduction to provide the result, the performance degradation is about 33%. This is due to the limited amount of computation performed to access and add local elements relative to the communication cost of the reduction. Figure 33(b) shows a generic inner product algorithm operating on the data in two `pArrays`. For two given `pArrays` A[N] and B[N] the algorithm computes $res = \sum_{i=0..N-1}(A[i] * B[i])$. the complexity of the parallel algorithm is $O(N/P + log(P))$ where N is the number of elements and P is the number of available locations.

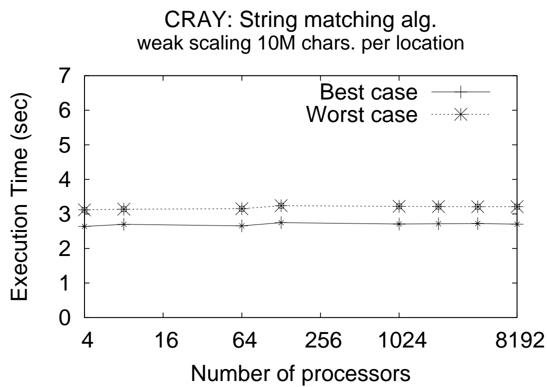
A more complex algorithm that we analyze in this section is the string matching. It takes as input two `pViews` defined over two `pContainers` and counts the number of occurrences of the data of the second `pView` (e.g., pattern) in the first one (e.g., sequence). This algorithm incurs communication that can be at most the size of the pattern. In Figure 33(c) we include results for two weak scaling experiments. In a first experiment, we search for a pattern that doesn't exist in the input sequence. The input sequence is filled with "A" and the pattern is "ZZZ". In this case there is no communication. In a second scenario we search for the pattern "AAA" that occurs $N-2$ times in the input sequence. As we see from the figure, this computation scales well as we scale the number of processors from 4 to 16384.



(a) `p_generate`, `p_for_each`, `p_accumulate`



(b) Inner product



(c) String matching

Fig. 33. Execution times for generic algorithms on CRAY4 for a `pArray` with 20M elements per processor.

F. Memory Consumption Study

In this section we analyze the memory consumption of the `pArray`. We analyze this using `pContainer` internal methods that report memory usage and using the *Integrated Performance Monitoring (IPM)* [62] tool available for CRAY4. All memory sizes reported in this section are in MB.

Table XXII.: `pArray` memory consumption: The first two columns shows the IPM reported memory usage for a simple STAPL program that performs only one fence. The next two columns shows the memory usage for a STAPL program that declares an `std::valarray` of 20M integers on each location; The last two columns shows the memory usage for a STAPL program that declares a `pArray`of size P*20M integers. All sizes are in MB

	stapl_main		stap_main and vallarray		stapl_main and `pArray`	
Processors	Total	PerLoc	Total	PerLoc	Total	PerLoc
4	155.72	38.93	460.89	115.22	461.14	115.29
8	838.96	104.89	1449.3	181.19	1449.8	181.25
64	6721.94	105.08	11604.68	181.37	11608.78	181.5
128	13477.89	105.3	23243.37	181.59	23251.35	181.66
1024	110880.77	108.29	189004.8	184.59	189310.98	185.58
2048	229049.34	111.91	385298.43	188.21	386081.79	190.33
4096	487207.94	118.97	799720.45	195.26	802568.19	199.66
8192	1088993.28	133.05	1714012.16	209.35	1724835.84	218.73

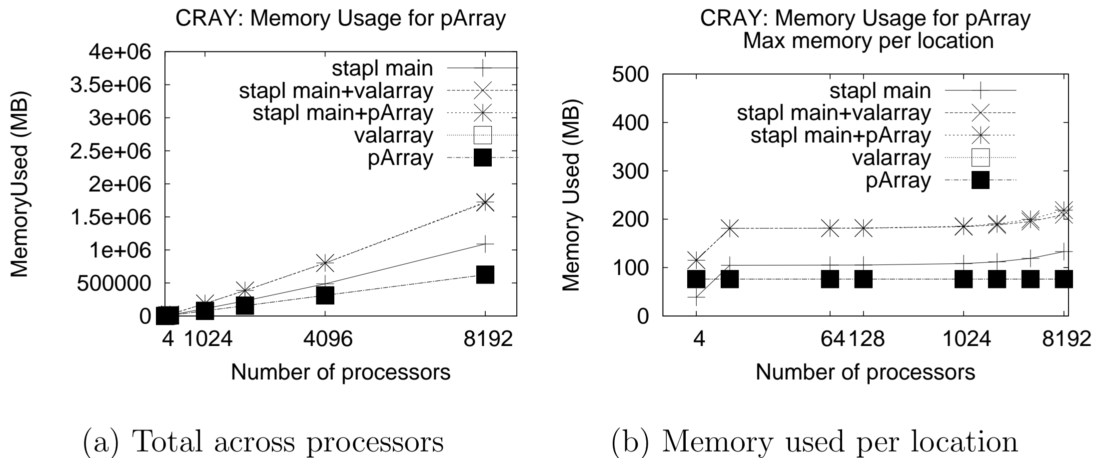
In Table XXII, column 3 and 4, we include the memory usage as reported by IPM for a simple STAPL program that only performs a fence and exits. IPM reports the maximum memory used (watermark) across all processes of the application. We observe that even though there is no data allocated by the program, the memory used is still considerable. This is mainly due to the communication and synchronization buffers allocated by the underlying run time system and MPI. The maximum mem-

ory used per processor increases with the number of processors from 38.93MB on 4 processors to 133.05MB on 8192. A considerable jump happens from 4 to 8 processors when inter-node communication is required.

In Table XXII, columns 4 and 5, show the memory usage as reported by IPM when a sequential array of integers of size 20 million is allocated inside STAPL main. Columns 6 and 7 show the memory used when STAPL main declares a `pArray` with 20M elements per location. We notice very similar memory usage as we increase the number of processors for these two situations. The `pArray` does allocate slightly more memory than a simple `valarray` to store the data distribution information, the location manager and the infrastructure required for synchronization (e.g., thread safety manager). In Table XXIII we show the theoretical minimum memory used by a distributed array with 20M elements per location. This is computed as $20M * P * sizeof(int)$.

Table XXIII.: Theorethical memory usage for `pArray`: minimum required memory (no meta data) and `pArray` reported memory consumption (data and meta data). All sizes are in MB

	Theoretical		`pArray` reported	
Processors	Total	PerLoc	Total	PerLoc
4	305.176	76.294	305.178	76.295
8	610.352	76.294	610.357	76.295
64	4882.813	76.294	4882.850	76.295
128	9765.625	76.294	9765.700	76.295
1024	78125.000	76.294	78125.600	76.295
2048	156250.000	76.294	156251.000	76.295
4096	312500.000	76.294	312503.000	76.295
8192	625000.000	76.294	625005.000	76.295



(a) Total across processors

(b) Memory used per location

Fig. 34. CRAY4: `pArray` memory usage study. The simple program contains either no data (stapl main), a valarray of 20M elements per location (stapl main+valarray), a `pArray` of size P*20M (stapl main+pArray). Theoretical memory usage for a valarray of size P*20M (valarray) and `pArray` reported memory usage (pArray). Lines 2,3 overlap and similar lines 4,5. (a) Total memory used; (b) Memory used per location.

In Figure 34(a), we plot the data from Tables XXII and XXIII. We observe the memory per location increasing for all situations measured with IPM. This is an expected trend that is accordingly documented in the MPI user manual of the machine. In Figure 34(b), we show the memory used per location. We notice a slight difference between the STAPL program declaring a `valarray` and the STAPL program declaring a `pArray`. The difference increases with the number of processors from 0.06% on 4 processors to 4% on 8192 processors.

CHAPTER X

THE STAPL PLIST*

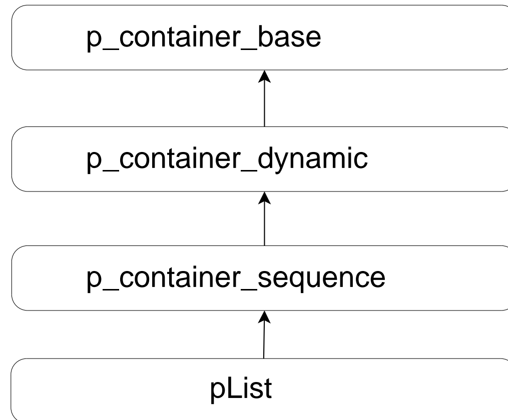


Fig. 35. Derivation chain for `pList`.

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

*Part of the data reported in this chapter is reprinted with the kind permission of Springer Science+Business Media from "The STAPL pList" by G. Tanase, X. Xu, A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, M. Bianco, N. M. Amato, and L. Rauchwerger, 2009. *Lecture Notes in Computer Science*, vol. 5898, pp. 16–30, Copyright 2009 by Springer.

STAPL `pList` is a parallel equivalent of the STL list with an interface for efficient insertion and deletion of elements in parallel. the derivation chain for `pList` is included in Figure 35.

Most STL equivalent methods require a return value, which in general translates into a blocking method. For this reason, we provide a set of asynchronous methods, e.g., `insert_async` and `erase_async`. These non-blocking methods allow for better communication/computation overlap and enable the STAPL RTS to aggregate messages to reduce the communication overhead. 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`, that allow the `pList` to insert the element in an unspecified location in order to minimize communication and improve concurrency.

A. pList Example

We include in Figure 36 a simple example that inserts all elements from an input `pArray` into a `pList` using their corresponding `pViews`. The program declares a `pArray` of size 1000 and an empty `pList` using their default partitions and traits (Figure 36, Lines 11 and 12). Subsequently, in line 14 the STAPL `map_func` construct is called to create tasks that take individual elements from the `pArray` and insert them into the `pList`. The work function of the tasks created by `map_func` is included starting with line 4 and it invokes the `push_anywhere` method of `pList`.

B. pList Specification

The `pList` declaration is:

```
1  #include <p_array.h>
2  include <p_list.h>
3
4  class insert_functor{
5    void operator()(int& elem, list_pview& lview){
6        lview.push_anywhere(elem);
7    }
8  };
9
10 void stapl_main(){
11   p_array<int> pa(1000); // parray with 1000 elements
12   p_list<int> pl; //empty plist
13   array_1D_pview<p_array<int> > pa_view(pa);
14   map_func( pa_view, list_pview(pl), insert_functor() );
15 }
```

Fig. 36. `pList` example.

```
template<class T, class Partition=Default, class Traits=Default>
class p_list;
```

The `pList` has the derivation chain included in Figure 35. We include in Table XXIV the `pList` interface.

Table XXIV.: `pList` interface.

Method	**Description**
p_list(size_t N, const T& value = T())	Creates a `pList` with N elements, each of which is a copy of `value`. $O(N/P + log(P))$
p_list(size_t N, partition_type& ps)	Creates a `pList` with N elements based on the given partition strategy. $O(N/P + log(P))$
void splice(iter pos, pList& pl);	Splice the elements of `pList` `pl` into the current list before the position `pos`.
size_t size() const	Returns the size of the `pList`. $O(log(P))$
bool empty() const	True if the `pList`'s size is 0. $O(log(P))$

Table XXIV continued

Method	**Description**
T& [front|back]()	Access the first/last element of the sequence. $O(1)$
void push_[front|back](const T& val)	Insert a new element at the beginning/end of the sequence. $O(1)$
void pop_[front|back]()	Remove the element from the beginning/end of the sequence. $O(1)$
iterator insert(iterator pos, const T& val)	Insert val before position pos and return the iterator to the new inserted element. $O(1)$
void insert_async(iterator pos, const T& val)	Insert val before pos with no return value. $O(1)$
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. $O(1)$
void erase_async(iterator pos)	Erases the element at position pos with no return value. $O(1)$
iterator push_anywhere(const value_type& val)	Push val on to the last local `bContainer` and return the iterator pointing to the new inserted element. $O(1)$
void push_anywhere_async(const T& _val)	Push val on to the last local `bContainer` with no return value. $O(1)$

C. pList Design and Implementation

In this section, we describe the `pList` modules used for storage and data distribution information.

bContainer: For the STAPL `pList`, we use the STL `list` as the container of the `bContainer`. Most `pList` methods will ultimately be executed on the `bContainer` using the `bContainer`'s corresponding methods. For example, `pList` insert will ultimately invoke the STL list insert method. The `pList bContainer` can also be provided by the user so long as insertions and deletions never invalidate iterators, and the `bContainer` provides the required *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).

The `pList` has a global view of all of the `bContainers` and knows the order between them in order to provide a unique traversal of all its data. For this reason each `bContainer` is identified by a globally unique `BCID`. For static or less dynamic – in terms of number of `bContainers` – `pContainers` such as `pArray` or associative containers, the `BCID` can be a simple integer. The `pList`, however, needs a `BCID` that allows for fast dynamic operations. During the splice operation, `bContainers` 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` `bContainers` is currently defined as follows:

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

Global Identifiers (`GID`): Performance and uniqueness considerations similar to those of the `bContainer` 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;
```

Since the `BCID` is unique, the `GID` is unique as well. With the above definition for `GID`, the `pList` can uniquely identify each of its elements and access them independent of their physical location.

Domain: The domain interface for the `pList` is provided by the `pList` `bContainers`. The `pList` domain is a union of all domains corresponding to individual `bContainers`. The union domain doesn't replicate any data from the `pList` but it stores pointers to its `bContainers`.

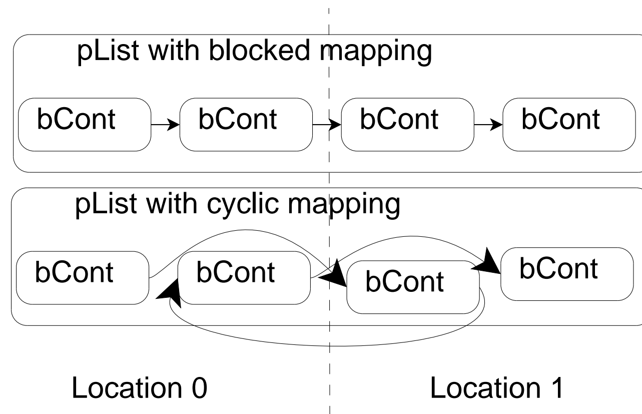


Fig. 37. Different partitions and mappings for pList.

Data Distribution: The data distribution manager for a pList uses a partition and a partition-mapper to describe how the data will be grouped and mapped on locations. The pList specializes the partition mapper to take advantage of the fact that the location identifier is embedded in the bContainer identifier.

The pList uses a dynamic partition that can maintain an arbitrary number of bContainers and elements per location. The partition constructor can take an optional argument, which is the number of desired bContainers and it will allocate them balanced across locations. The allocation can be done in a blocked fashion or in a cyclic fashion as depicted in Figure 37. Subsequent insert and delete operations may lead to imbalanced distributions of elements in the bContainer. The pList provides a method for this situation to redistribute the data so that elements are rebalanced across locations.

pView: 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 partition of a pList pView matches the subdivision of the list in

`bContainers`, thus allowing random access to portions of the `pList`. This allows parallel algorithms to achieve good scalability as shown in Section D.

pList Container: A typical implementation of a `pList` method that operates at the element level is included in Figure 38 and uses the `invoke` skeleton introduced in Chapter IV, Section 6. The run-time cost of the method has three constituents: the time to decide the location and the `bContainer` where the element will be added (Figure 8, line 5-15), the communication time to get/send the required information (Figure 8, line 10), and the time to perform the operation on a `bContainer` (Figure 8, line 17).

The complexity of constructing a `pList` of N elements is $O(M + \log P)$, where M is the maximum number of elements in a location. The $\log P$ term is due to a fence at the end of the constructor to guarantee the `pList` is in a consistent state. The complexities of the element-wise methods are $O(1)$. Multiple concurrent invocations of such methods may be partially serialized due to concurrent thread-safe accesses to common data. The `size` and `empty` methods imply reductions and the complexity is $O(\log P)$, while `clear` is a broadcast plus the deletion of all elements in each location, so the complexity is $O(M + \log P)$. This analysis relies on the `pList bContainer` to guarantee that allocation and destruction are linear time operations and `size`, `insert`, `erase` and `push_back/front` are constant time operations.

The `pList` also provides methods to rearrange data in bulk. These methods are `splice` and `split` to merge lists together and split lists, respectively.

The signature of the `pList splice` method is:

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

where `iter` stands for an iterator type, `pos` is an iterator of the calling `pList`, `pl` is another `pList`, and the optional iterators `it1` and `it2` are iterators pointing to

```
1  p_list::p_container_sequence::insert (_gid, _val) {
2    this->m_dist->invoke(MP_INSERT_ELEMENT,
3      boost::bind(&partition_type::insert_element, _gid, _val),
4      boost::bind(&partition_type::where_insert_element, _gid));
5  }
```

Fig. 38. `pList` method implementation.

elements of `pl`. `splice` removes from `pl` the portion enclosed by `it1` and `it2` and inserts it at `pos`. By default `it1` denotes the begin of `pl` and `it2` the end.

The complexity of `splice` depends on the number of `bContainers` included within `it1` and `it2`. If `it1` or `it2` points to elements between `bContainers`, then new `bContainers` are generated in constant time using sequential list splice. Since the global begin and global end of the `pList` are replicated across locations, the operation requires a broadcast if either of them is modified.

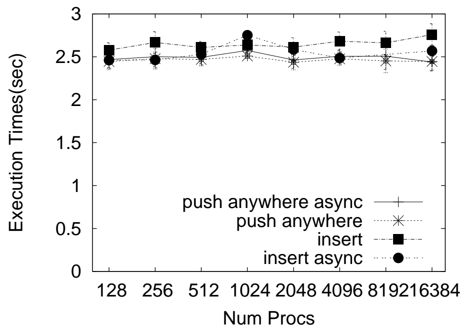
`split` is also a member method of `pList` that splits one `pList` into two. It is a parallel method that is implemented based on `splice` with the following signature:

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

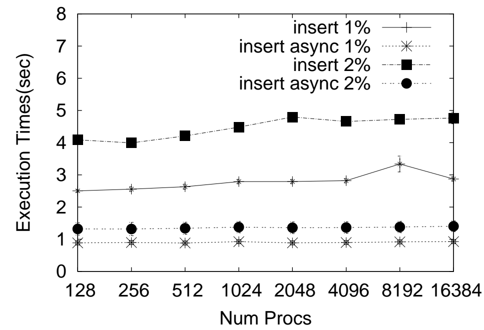
When `pList.split(pos, other_plist)` is invoked, the part of `pList` starting at `pos` and ending at `pList.end()` is appended at the end of the `other_plist`. The complexity of `split` is analogous to the complexity of `splice`.

D. Performance Evaluation

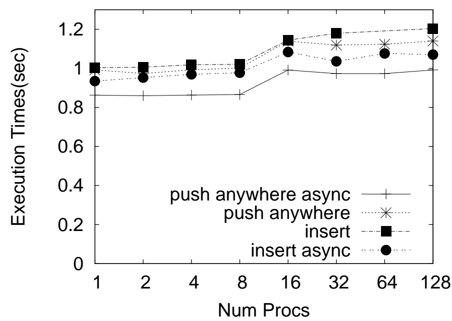
In this section, we evaluate the scalability of the `pList` methods. We compare `pList` and `pVector` performance, evaluate generic `pAlgorithms` (`p_generate` and `p_partial_sum`) on `pList`, `pArray` and `pVector`, and evaluate an Euler tour imple-



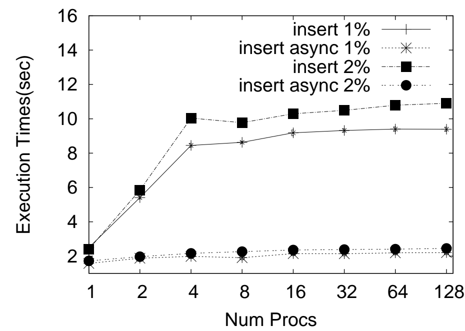
(a) CRAY4: Local Methods



(b) CRAY4: Insert Remote



(c) P5-CLUSTER: Local Methods



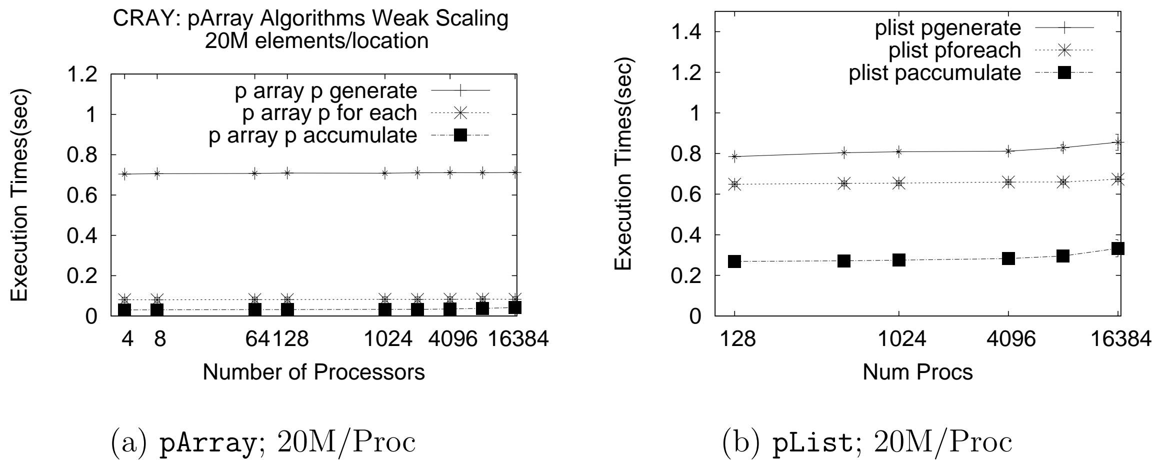
(d) P5-CLUSTER: Insert Remote

Fig. 39. Execution times for `pList` methods.

mentation using `pList`.

E. pList Method Evaluation

Figure 39 shows the execution time of different `pList` methods. In a first study, all methods are executed locally and we observe in Figure 39(a) that both synchronous and asynchronous methods exhibit scalable performance as they do not incur any communication. In Figure 39(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 methods. In this situation, the synchronous methods incur a



(a) `pArray`; 20M/Proc

(b) `pList`; 20M/Proc

Fig. 40. Execution times for `p_for_each`, `p_generate`, `p_accumulate` algorithms on CRAY4 for `pArray` and `pList`.

performance overhead being 2.5 times slower. In Figure 39(c) and 39 (d) we show the same experiment on P5-CLUSTER. We observe the same trends as on CRAY4 except that now the difference between synchronous and asynchronous is much larger with the synchronous operations being 5 times slower when using 128 processors.

F. pAlgorithm Comparison

Figure 40(a) presents the execution times for the `p_generate`, `p_for_each` and `p_accumulate` algorithms on the data of a `pArray` and a `pList` on CRAY4 using from 128 to 16384 processors. Figure 40(b) shows the results for the three algorithms on the `pList`. We observe that the execution time for `pList` is higher than for `pArray`, which is the result of the longer access time for the elements of the STL `list` with respect to the STL `valarray`. When using 16384 processors there is a 18% increase in execution time for `pArray` and a 5% increase for `pList`.

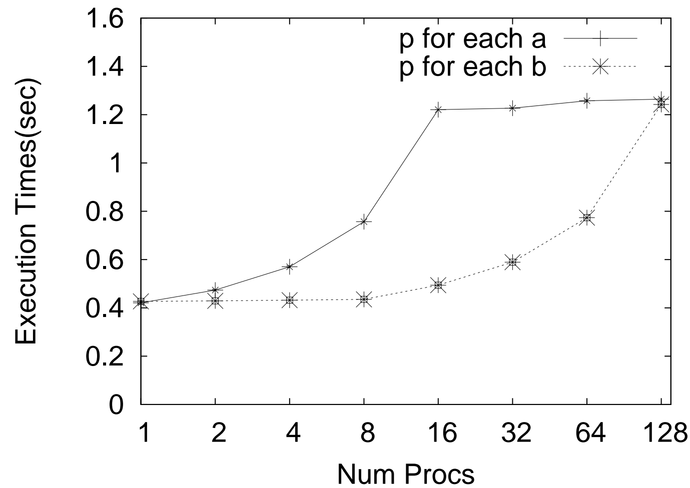
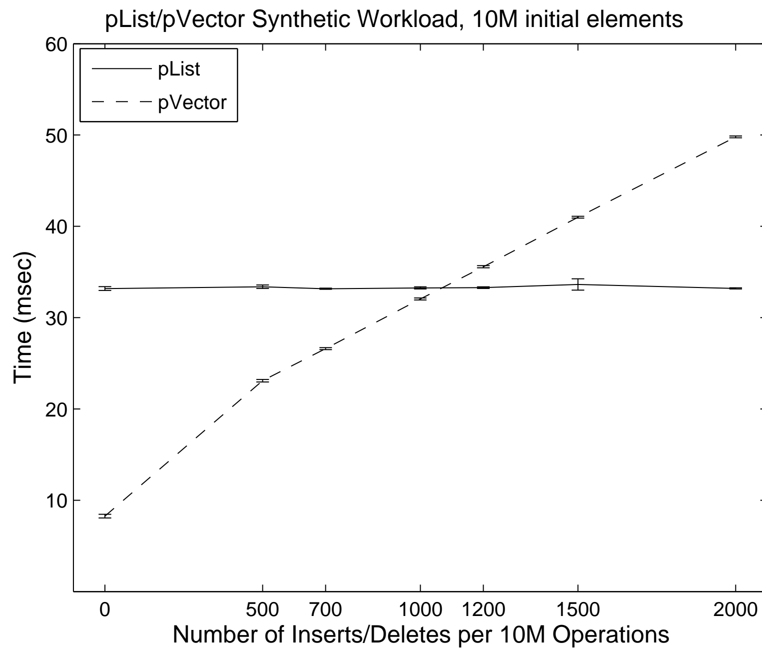


Fig. 41. P5-CLUSTER: Weak scaling for `p_for_each` allocating processes on the same nodes when possible (curve a) or in different nodes (curve b). Experiments are for 20 million elements/processor.

Figure 41 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_for_each`-a represents the case where all processors are allocated on a single node (possible for 1-16 processors). `p_for_each`-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_for_each`-b, the nodes are fully utilized only when running on 128 processors, while for `p_for_each`-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.



`pList` and `pVector` comparison

Fig. 42. Comparison `pList` and `pVector` dynamic data structures using a mix of 10M operations (read/write/insert/delete).

G. 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 42, 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` is better than `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 a truly random access container like the `pVector` .

H. Application using pList: Euler Tour

An Euler Tour (ET) is an important representation of a graph for parallel processing. In particular, the ET, which traverses every edge of the graph exactly once, corresponds to an edge traversal of the graph. 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 (given a vertex to be root, compute the parent of each vertex in the tree), postorder numbering (compute the postorder number for each vertex in the tree), computing the vertex level (the level of each vertex in the tree), and computing the number of descendants (the number of descendants for each vertex in the tree) [38]. In this application, we convert a tree $T = (V, E)$ into a

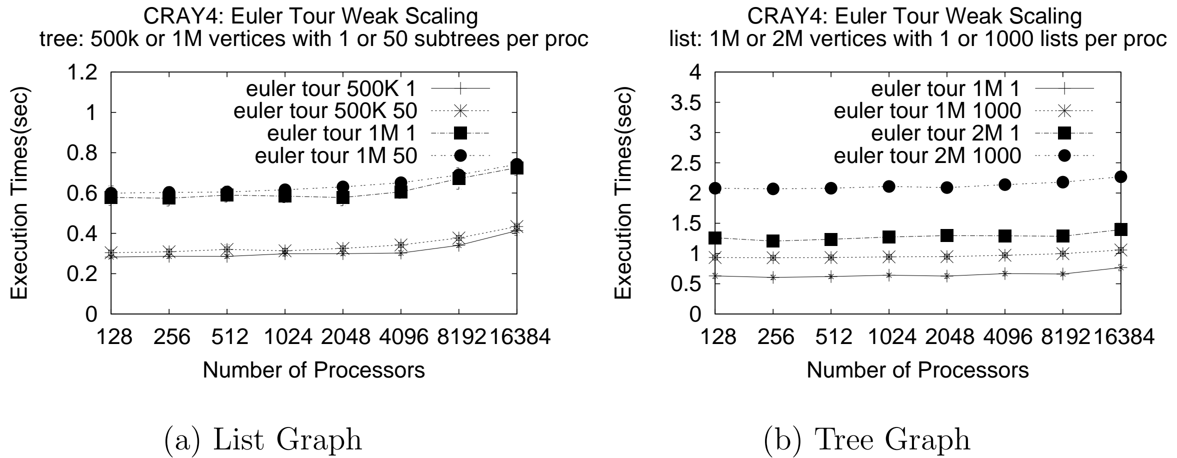
directed tree $T' = (V, E')$ where each edge $(u, v) \in E$ is replaced by two edges (u, v) and (v, u). T' is an Eulerian graph.

The parallel Euler Tour algorithm [38] implemented in STAPL computes an ET of a tree stored in a STAPL `pGraph` and stores it in a `pList`. The algorithm executes in parallel traversals on the `pGraph` `pView` generating 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.

The tree ET applications are computed by first using the ET algorithm to compute the ET, and then applying a generic ET algorithm. The generic algorithm first initializes each edge in the tour with a corresponding weight, and then compute prefix sums using the partial sum algorithm. The partial sum result for each edge is copied back to the graph, and the final step is to compute the desired result, e.g., parent, postorder number, vertex level or descendants, using an appropriate function on the prefix sum values.

The performance of the algorithm for computing ET is evaluated by performing a weak scaling experiment on CRAY4 using as input a tree distributed across all locations. The tree 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 of the number of subtrees for each location (for each root of the subtree, one to its root and two to its children in each location, with directed edges for both directions). Figure 43(a) and (b) shows the execution time on CRAY4 for different sizes of the tree and different number 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 time taken to link them.

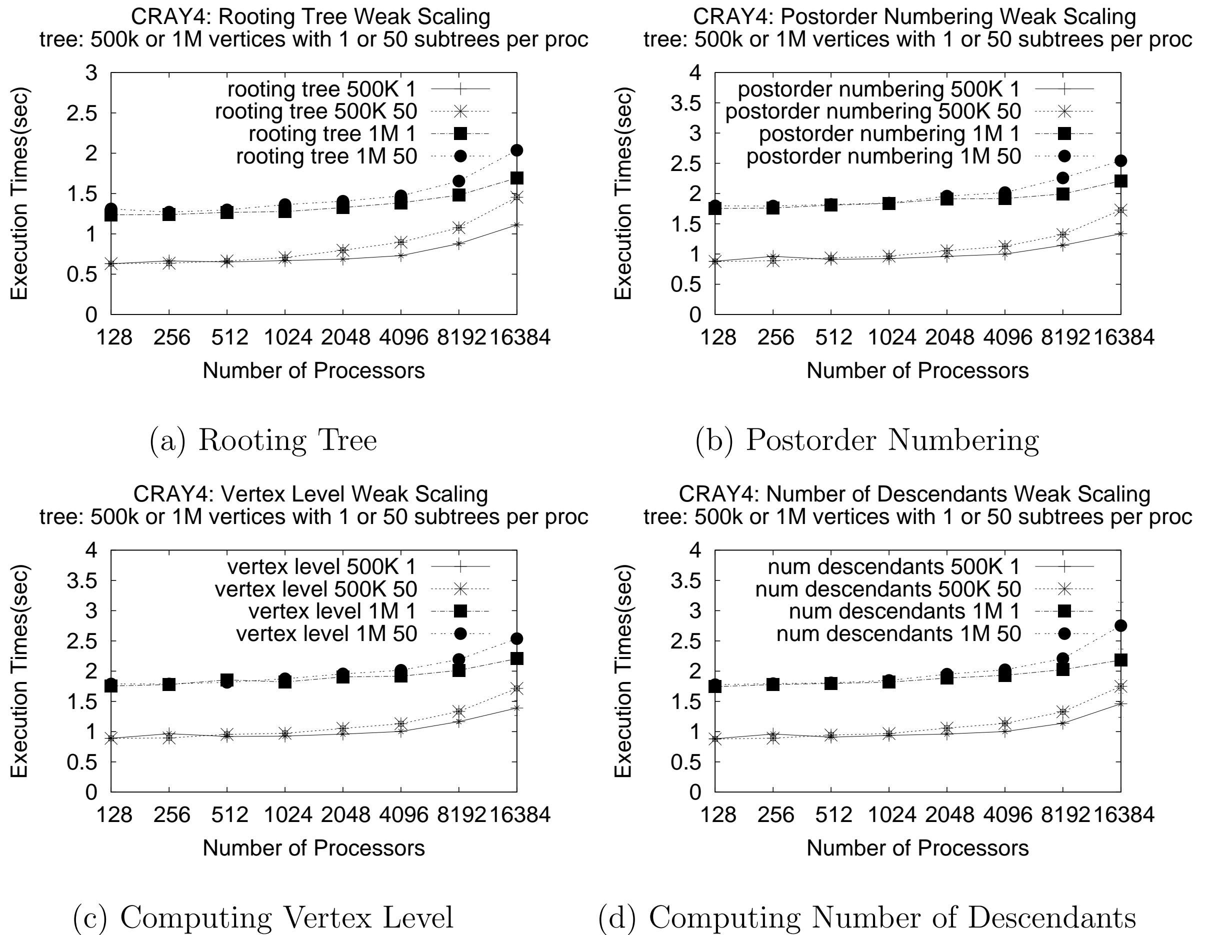
The performance of the ET technique generic algorithms is shown in Figure 44(a),



(a) List Graph (b) Tree Graph

Fig. 43. Weak scaling of Euler Tour algorithm. Tree made by a single binary tree with 500k or 1M subtrees per processor.

(b), (c) and (d) for rooting a tree, computing the postorder numbering, the vertex level, and the number of descendants, respectively. The running time increases with the number of vertices per location because the number of edges increases, which are proportional to the computation. The more subtrees specified in each location, the more segments formed in the `pList` and the more communication time taken for the partial sum algorithm.



(a) Rooting Tree

(b) Postorder Numbering

(c) Computing Vertex Level

(d) Computing Number of Descendants

Fig. 44. Execution times for Euler Tour and its applications using a tree made by a single binary tree with 500k or 1M subtrees per processor.

CHAPTER XI

THE STAPL PGRAPH

The STAPL `pGraph` is a generic data structures that consists of a collection of vertices and relations between vertices called edges. We use the following notation for a graph: $G = (V, E)$. The `pGraph` associates a vertex property with each vertex and an edge property with each edge. These are template arguments that are passed by the user when instantiating a `pGraph`. Additionally, using template arguments, users can indicate if the graph is directed (*directedness*) and if the graph allows multiple edges between same source and destination (*multiplicity*). Similar to other STAPL `pContainers`, users can specify an optional partition and traits. Thus, the `pGraph` declaration is the following:

```
stapl::p_graph<Directnedss,
       Multiplicity,
       vertex_property=no_property, //optional
       edge_property=no_property,   //optional
       partition=default,           //optional
       traits=default_traits        //optional
       >
```

If the user application doesn't require vertex or edge properties, then the special value `stapl::no_property` can be used. The `traits` allow users to customize low level details such as the storage for vertices and edges. Before we introduce the `pGraph` user interface, we define a set of concepts in the next section that are required to properly describe the methods.

A. pGraph Example

```
1  #include <p_array.h>
2  #include <p_graph>
3  struct add_vertex_func(){
4    p_graph_pview pg_view;
5    void operator()(vertex_property& pref){
6      pg_view.add_vertex(pref);
7    }
8  };
9  struct add_edge_func(){
10   p_graph_pview pg_view;
11   void operator()(pair<vertex_descriptor,
12                        vertex_descriptor>& pref){
13     pg_view.add_edge(pref.first, pref.second);
14   }
15 }
16 void stapl_main(){
17  // parray with 1000 vertex properties
18  p_array<vertex_property> pa_verts(1000);
19  // parray with 10000 edges
20  p_array<pair<vertex_descriptor,
21              vertex_descriptor> > pa_edges(10000);
22  p_graph<DIRECTED, MULTI, vertex_property, edge_property> pg;
23  p_for_each(array_1D_pview(pa_verts),
24             add_vertex_func(p_graph_pview(pg));
25  p_for_each(array_1D_pview(pa_edges),
26             add_vertex_func(p_graph_pview(pg));
27 }
```

Fig. 45. `pGraph` example.

In Figure 45 we include a simple example using the `pGraph` data structure. At line 17 and 19 we declare a `pArray` of vertex properties and a `pArray` of edges, respectively.

Subsequently we invoke two `p_for_each` invocations with functors to create vertices and edges for all elements in the input `pArrays`. The functors call `add_vertex` and `add_edge` methods of the `pGraph pView` which recursively invoke the `add_vertex` method of the `pGraph`.

B. pGraph Concepts and Interfaces

When implementing or using STAPL graphs the user needs to be aware of a number of additional concepts relative to the `pArray` and `pList`. The `pGraph` concepts and their associated properties are introduced in this section.

Vertex Descriptor: The vertex descriptor uniquely identifies a vertex. The vertex descriptor has to be default constructable, assignable and equality comparable. Adding edges or finding vertices will be based on vertex descriptors.

Edge Descriptor: The edge descriptor uniquely identifies an edge. It provides methods to return the source and target vertex and a unique edge identifier. The reason the edge identifier is needed is to distinguish between vertices with the same source and destination.

Vertex Iterator: The vertex iterator is an overloaded iterator concept. It implements the STL **bidirectional access iterator** concept (operator ++, - -) and can be dereferenced to provide a **Vertex Reference**.

Vertex Reference: We provide two vertex reference concepts. They correspond to the case where the graph does not have properties associated with the vertex (vertex_reference<no_property>) and the case where the graph has properties associated with a vertex (vertex_reference<vertex property>). The vertex reference is obtained by dereferencing a vertex iterator. It provides the interface in Table XXV.

Table XXV.: Vertex reference interface. Property related interface is only for vertex_reference<vertex property>.

Defined Type	**Description**
vertex_descriptor	Vertex descriptor type
property_type	Property type for graphs with no property is a special type `no_property`
adj_edge_iterator	Adjacency edge iterator type. This type of iterator allows to iterate over the adjacent edges of a vertex referred by the current iterator
Method	**Description**
adj_edge_iterator begin()	Returns an adjacency edge iterator pointing to a first outgoing edge. $O(1)$
adj_edge_iterator end()	Returns an adjacency edge iterator pointing to a last outgoing edge. $O(1)$
view edges()	Returns a `pView` over the adjacent edges. $O(1)$
size_t size() const	Returns the number of outgoing edges. $O(1)$
Property& property()	Returns the property associated with the vertex. $O(1)$

Edge Iterator: Similar to the vertex iterators, the graph provides edge iterators. First, the **adj_edge_iterator** iterates over the adjacency list of a vertex and secondly, **edge_iterator** iterates over all edges of the graph. It can be dereferenced to provide an **Edge Reference**.

Edge Reference: The edge reference is obtained by dereferencing an edge iterator or an adjacent edge iterator. It provides the interface in Table XXVI.

Table XXVI.: Edge reference interface.

Defined Type	**Description**
property_type	Property is of type `no_property` meaning the edges are without properties
vertex_descriptor	Vertex descriptor type
vertex_descriptor source() const	Returns vertex descriptor for the source. $O(1)$
vertex_descriptor target() const	Returns vertex descriptor for the target. $O(1)$
size_t id() const	Returns the unique id of the edge. $O(1)$
Property& property()	Returns the property associated with the edge. $O(1)$

C. pGraph Class Hierarchy

The STAPL `pGraph` provides the following classes of graphs: `incidence_p_graph`, `static_p_graph`, `dynamic_p_graph`, `directed_p_graph`, `undirected_p_graph`, `multi-edges_p_graph` and `non_multi_edges_p_graph`. Each of these specify an interface and corresponding properties as summarized in Figure 46(a). More details about the functionality of individual methods are included in Table XXVII.

Corresponding to each of the graph concepts in Figure 46(a) there is a `pGraph` `pView` interface. Generic parallel graph algorithms need to specify the requirements for the `pView` that they can operate on. For example certain algorithms may require an `incidence_p_graph_view` that models the `incidence_p_graph` concept.

STAPL provides a default implementation of the `pGraph` concepts from Figure 46(a) as depicted in Figure 46(b). There is a `p_container_relation` class derived from `p_container_dynamic` class of the framework that implements the interfaces corresponding to incidence, static and dynamic graph concepts. Directed `pGraph` specifies additional methods such as `get_out_degree()`. Undirected guarantees that

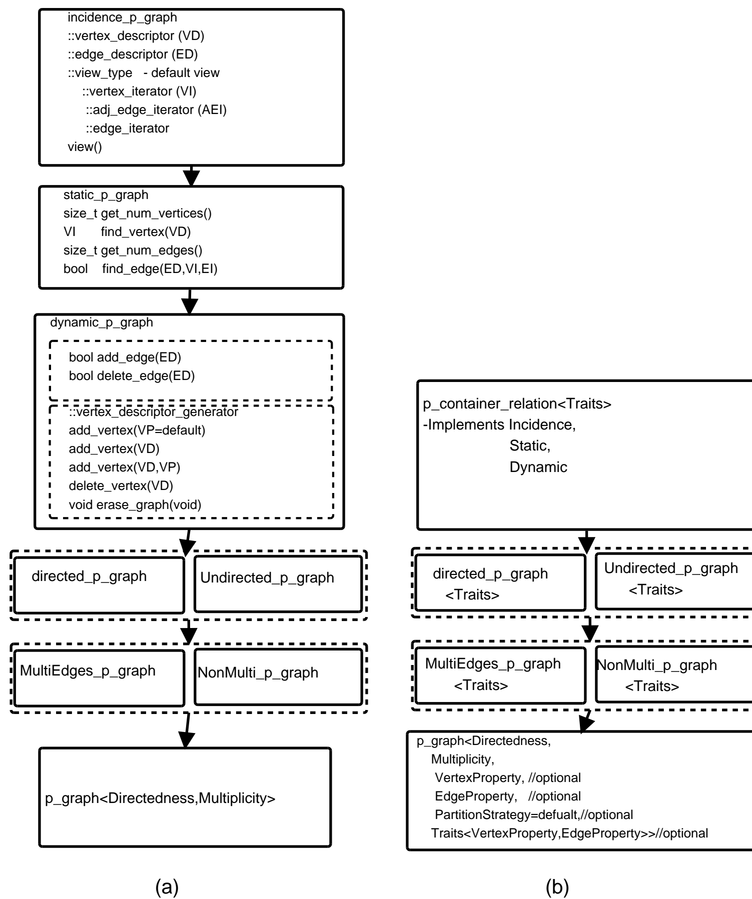


Fig. 46. Graph hierarchy of concepts (a) and hierarchy design for STAPL implementation of graph (b). AEI is an iterator over adjacent edges only; EI iterates over all edges of a graph and it is an extension of the adjacency edge iterator(AEI).

for every edge (u, v) there is an edge (v, u). Additionally, if the graph has properties the property is shared by the two edges. The non multiple edges class specializes the `add_edge` method to guarantee that there are no edge duplicates in the `pGraph`. The `pGraph` class at the bottom of the hierarchy in Figure 46(b) uses the template arguments to select the proper derivation chain and other internals such as partition and storage.

Table XXVII.: `pGraph` interface.

Define Type	**Description**
vertex_property	property stored on vertices; it can be of no_property type
edge_property	property stored on edges; it can be of no_property type
vertex_descriptor	vertex descriptor type
edge_descriptor	edge descriptor type
vertex_reference	vertex reference type
edge_reference	edge reference type
vertex_descriptor add_vertex()	Add a vertex with default allocated property. $O(1)$
vertex_descriptor add_vertex(const vertex_property& pref)	Add a vertex with the specified property. $O(1)$
vertex_descriptor add_vertex(const vertex_descriptor vd)	Add a vertex with the specified descriptor. $O(1)$
vertex_descriptor add_vertex(const vertex_descriptor vd, const vertex_property& pref)	Add a vertex with the specified descriptor and property. $O(1)$
bool delete_vertex(const vertex_descriptor vd)	Delete the vertex with the specified descriptor. $O(NVerts)$
edge_descriptor add_edge(const edge_descriptor ed)	Add an edge with the given descriptor specifying the source and target vertices. $O(1)$
edge_descriptor add_edge(const edge_descriptor ed, const edge_property& pref)	Add an edge with the specified descriptor and property. $O(1)$
void add_edge_async(const edge_descriptor ed)	Asynchronously add an edge with the given descriptor specifying the source and target vertices; Faster than the synchronous variant. $O(1)$
void add_edge_async(const edge_descriptor ed, const edge_property& pref)	Asynchronously add an edge with the specified descriptor and property; Faster than the synchronous variant. $O(1)$
bool delete_edge(const edge_descriptor ed)	Delete the edge with the specified descriptor. $O(NEdges/NVerts)$ - proportional with the number of edges per vertex.
vertex_iterator find_vertex(const vertex_descriptor vd)	Find the vertex with the given vertex_descriptor. Returns `graph.end()` if vertex not found. $O(1)$

Table XXVII continued

Method	**Description**
edge_iterator find_edge(const edge_descriptor ed, vertex_iterator* vi, edge_iterator* ei)	Find the edge with the given edge_descriptor. Returns `graph.end()` if vertex not found. Also returns the vertex iterator of the source vertex, and the corresponding adjacency edge iterator. $O(Nedges/NVerts)$
size_t get_num_vertices()	Returns the number of vertices in the graph. $O(log(P))$
size_t get_num_edges()	Returns the number of edges in the graph. $O(NVerts/P + log(P))$
void clear()	Clears the graph, erasing all edges and vertices. $O(N/P + log(P))$
bool empty()	Returns if the graph is empty (has no vertices and edges). $O(log(P))$

In comparison with other graph libraries such as PBGL[29], for better encapsulation, we decided to have each concept correspond to a C++ class rather than having a mix of classes and stand alone functions. For example, to add a vertex or an edge in STAPL the user would say `graph.add_vertex()` and `graph.add_edge(source, target)`, rather than `add_vertex(graph)` or `add_edge (source, target, graph)` as is done in PBGL. In addition to these syntactic differences, we now briefly summarize some other important semantic differences between the STAPL `pGraph` and PBGL that may favor our approach for certain applications. First, the vertex and edge descriptor are uniquely associated with vertices and edges of the graph and will not change during the lifetime of the container. For PBGL, the descriptors are recomputed when a vertex is removed for certain storages leading to possible inconsistencies. Also, the vertex descriptor changes from BGL to PBGL by incorporating knowledge about the location where the vertex lives. In STAPL, we use the same vertex descriptor for

the sequential and the parallel graph and store location information in the data distribution manager. By not associating location information with the vertex descriptor, we can migrate vertices in our framework without invalidating descriptors to which other locations may have references.

Another important difference between `pGraph` and PBGL is the fact that `pGraph` provides a shared memory view to the user allowing threads on one location to access the entire graph in a shared memory fashion.

D. pGraph Design and Implementation

In this section we describe the `pGraph` default implementation of the main concepts required by the framework

bContainer: We use the sequential STAPL generic graph implementation for `bContainers` but other existing graph libraries such as BGL can be easily integrated. Most `pContainer` methods will ultimately be executed at the `bContainer` level. For example, `pGraph add_vertex` will ultimately invoke the sequential `add_vertex` method in one of the `pGraph bContainers`.

Global Identifiers (`GID`): The vertex descriptor is used as the `GID` and the default `pGraph` implementation uses an unsigned integer for this. An important property that differentiates the `pGraph` from other libraries is the fact that the vertex descriptor associated with a vertex remains valid as long as the vertex exists in the `pGraph`. Other libraries such as PBGL, depending on the storage chosen, rename all vertices in the graph to guarantee that they are in a contiguous range from zero to the number of vertices.

Partition: For `pGraph`, the partition of the set of vertices and edges into subsets is of crucial importance. The performance of the `pGraph` algorithms is often directly

related with the number of edges crossing the different sets. The default partitions provided by the framework partition the set of available vertices. The edges are implicitly partitioned based on their source vertices. We introduce next three partitions that can be used with the `pGraph`. First, a static balanced partition with a closed form solution mapping `GIDs` to `bContainers`. This is similar to the `pArray` balanced partition introduced in Chapter IV, Section E. While this partition performs fast mappings of vertex descriptors to sub-domains, it can be used only with graphs with a fixed number of vertices that only allow edges to be inserted or deleted.

The framework also provides two dynamic distributed partitions that allow arbitrary mappings of vertices to sub-domains. These two partitions can form the basis of smarter partitions that can integrate external tools such as Metis [42] or Chaco [32] to specify the decomposition (partition) of vertices. They are implemented using a distributed directory where a certain location is responsible for always knowing where a particular element lives and the location that actually stores the element needs to inform the directory with information about the element. When invoking a method on a specific element the partition may not have locally all the information about where the element is located. When this happens the partition has two options: (1) perform necessary communication and retrieve the full information about where the `bContainer` that owns the element, followed by the invocations of the method on the `bContainer` or (2) provide to the framework partial information about a new location that may know more information about the given element. In the second situation the `pContainer` simply forwards the method invocation to the provided location and the lookup procedure is recursively applied there. For asynchronous methods, forwarding the method based on partial information rather than trying to completely perform the address resolution, possibly using synchronous communication, has performance benefits that we analyze in Section F.

```
1  #include <p_graph>
2  class init_functor{
3    void operator()(vertex_reference v){
4        v.property() = 0;
5    }
6  };
7
8  void stapl_main(){
9    p_graph<DIRECTED, MULTIEDGES, double, double> pg;
10   // populate the graph with vertices and edges
11   vertex_set_pview vs(pg);
12   for_each( vs, init_functor() );
13 }
```

Fig. 47. `pGraph` `pViews` example.

Partition Mapper: Similar to other STAPL `pContainers` the `pGraph` can use any of the following partition mapper: `partition_mapper_generic`, `partition mapper_blocked`, `partition_mapper_cyclic` or `partition_mapper_identity`.

E. pGraph pViews

In Section C we introduced a taxonomy of various graph concepts. Corresponding to each of the concepts there is a well defined interface and graph algorithms are written in terms of these interfaces (e.g., `pViews`). These `pViews` can be defined on top of the STAPL `pGraph` but they can also be defined on top of other containers or on the composition of containers. For example, an `incidence_p_graph_pview` can be defined on top of a `pArray` of vertices where a vertex is a list of edges. In this section, we describe the `pViews` that can be defined on top of the `pGraph` data structure.

For simple algorithms that access all vertices or all edges, we define a `vertex_set pview` and `edge_set_pview`. As illustrated in Figure 47 these `pViews` can be passed as

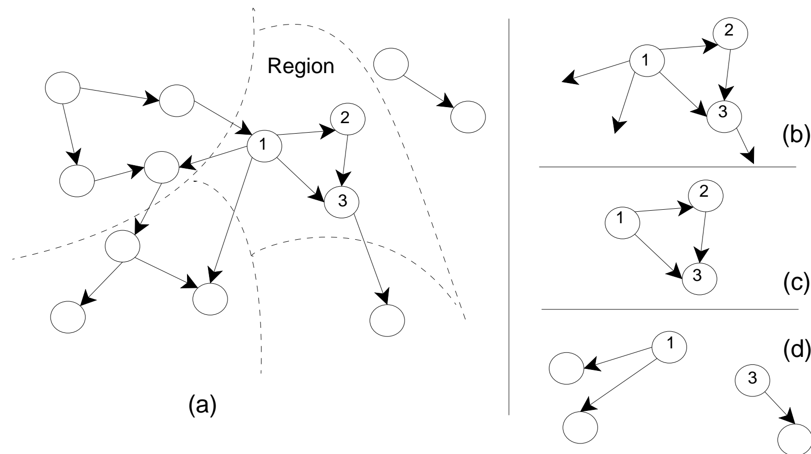


Fig. 48. `pGraph` `pViews`: (a) `pGraph` partitioned `pView`, (b) region_pview, (c) inner_pview and (d) boundary_pview.

arguments to STL like `pAlgorithms`. The `p_for_each` computation is used to initialize all vertex properties to zero.

pGraph Partitioned pView: In Figure 48(a) we depict a possible partition of a `pGraph` according to the dotted lines. For each of the regions of the decompositions we distinguish two categories of edges: **normal edges** where source and destination vertex are within the same sub domain and **boundary edges** where the target of the edge is not within the current region.

The framework provides a partitioned `pGraph` `pView` that can be used by various `pGraph` algorithms. Each of the sub views of the partitioned graph `pView` provides methods to obtain the following `pViews`:

- **Region pView:** (Figure 48(b)) Contains normal and boundary edges; it is **not** a complete graph. This is an intuitive way of partitioning a `pGraph` and leads to efficient parallel algorithms.

- **Inner pView:** (Figure 48(c)) A `pView` with the boundary edges stripped out.

This is a complete graph and can be passed to sequential graph algorithms.

- **Boundary pView:** (Figure 48(d)) An edge `pView` containing only the boundary edges.

F. Performance Evaluation

In this section, we evaluate the scalability of the `pGraph` parallel methods introduced in this chapter, discussing different performance trade-offs. We evaluate several graph `pAlgorithms` such as Euler tour and different traversals.

1. pGraph Methods Evaluation

The `pGraph` is represented as an adjacency list and depending on its properties, different `bContainers` can be used to optimize the data access. Here, we evaluate a static and a dynamic `pGraph`. The static `pGraph` allocates all its vertices in the constructor and subsequently only edges can be added or deleted. It uses a static partition that is implemented as a closed-form solution and has a `bContainer` that uses a `std::vector` to store the vertices and `std::list` to store edges. The dynamic `pGraph` uses a distributed directory to implement its partition and its `bContainer` uses `std::hash_map` for vertices and `std::list` for edges. We chose the `std::hash_map` in the dynamic case because it allows for fast insert and find operations. As described in Chapter V, in Figure 16 the `pGraph` container is one class and the static versus dynamic behavior is achieved by passing the corresponding template arguments to it.

We include in this section results for a weak scaling experiment on CRAY4 and P5-CLUSTER using a 2D torus where each processor holds a stencil of 1500×1500 vertices and corresponding edges, a stencil of 15×150000 and a random graph as specified in the SSCA2 benchmark [7]. SSCA2 generates a set of clusters where each

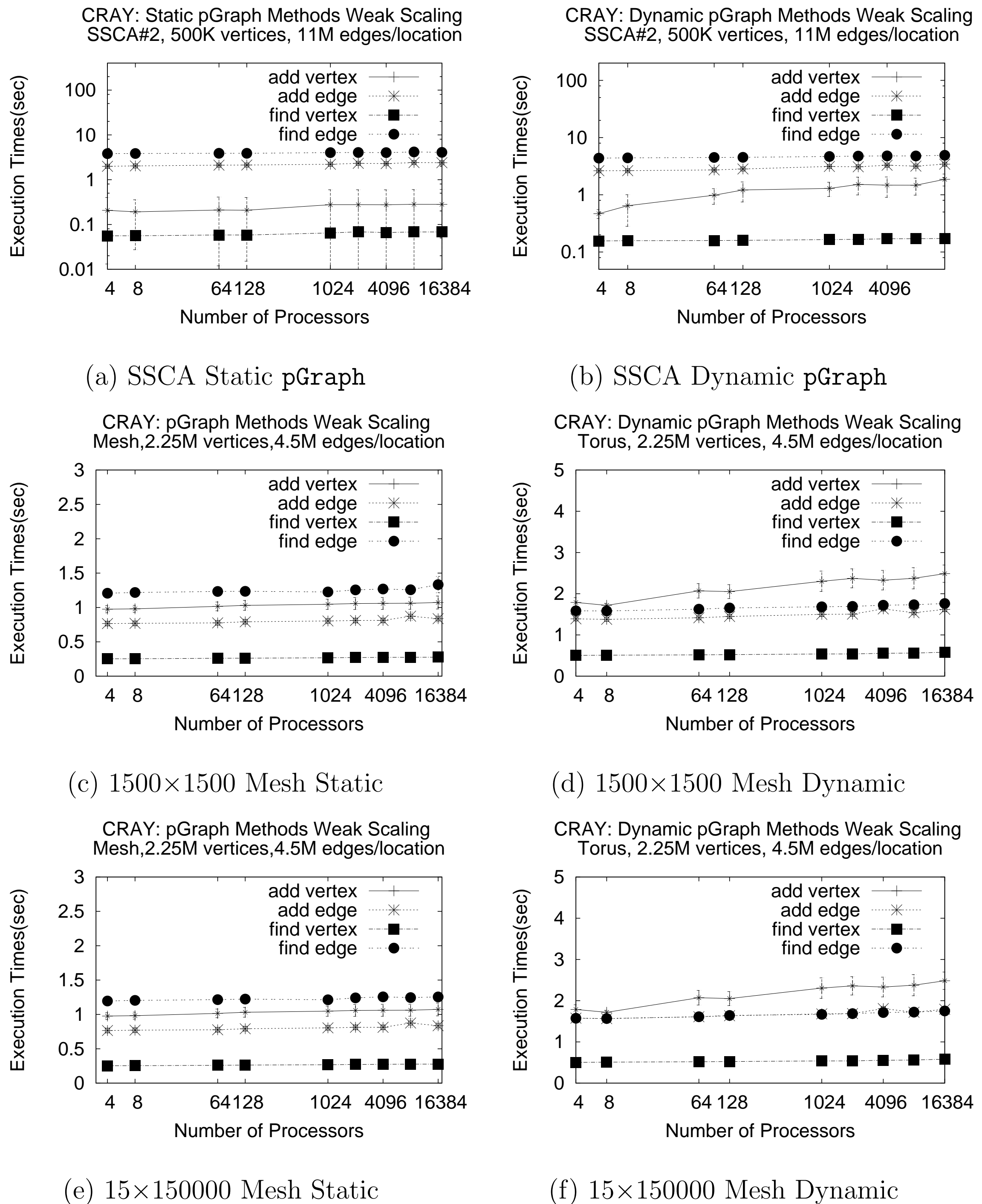
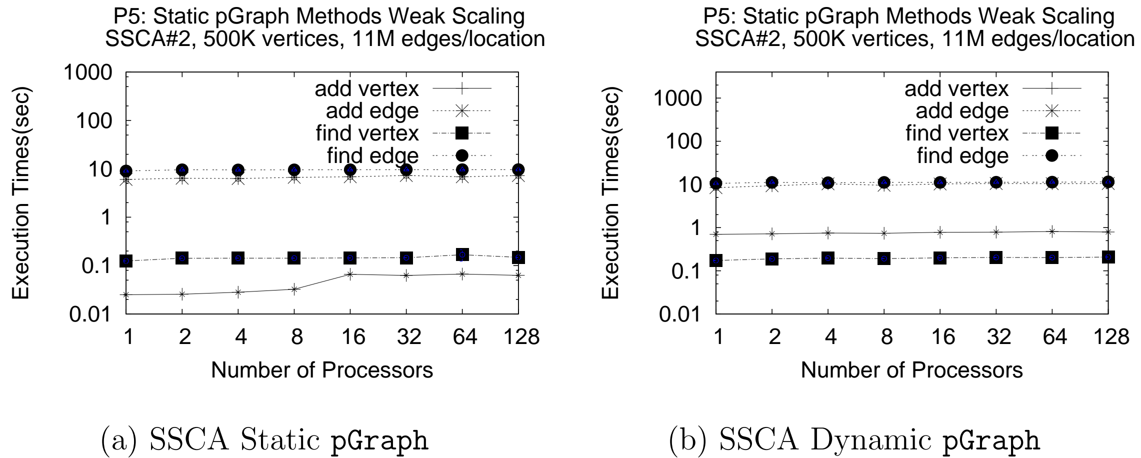


Fig. 49. CRAY4: Evaluation of static and dynamic `pGraph` methods while using the SSCA2 graph generator. The input graph has 500k vertices, 11.5M edges, ∼40 remote edges per location, ∼23 edges per vertex. (a) For the static `pGraph` all vertices are built in the constructor; (b) The dynamic `pGraph` inserts vertices using `add_vertex` method.



(a) SSCA Static `pGraph` (b) SSCA Dynamic `pGraph`

Fig. 50. P5-CLUSTER: Evaluation of static and dynamic `pGraph` methods while using the SSCA2 graph generator. The input graph has 500k vertices, 11.5M edges, ∼40 remote edges per location, ∼23 edges per vertex. (a) For the static `pGraph` all vertices are built in the constructor; (b) The dynamic `pGraph` inserts vertices using `add_vertex` method.

cluster is densely connected and the inter cluster edges are sparse. We use the following parameters for SSCA2: cluster size = $(V/P)^{1/4}$, maximum number of parallel edges is 3, maximum edge weight is V, probability of intra clique edges is 0.5 and probability of an edge to be unidirectional 0.3. Figure 49 shows the execution time for `add_vertex`, `add_edge`, `find_vertex` and `find_edges` on CRAY4 and Figure 50 for P5-CLUSTER. We include results for a static `pGraph` where vertices are allocated in the constructor and a dynamic `pGraph` where the container is initially empty and vertices are added using `add_vertex`. As seen in the plots, the methods scale well up to 24000 processors on CRAY4 and up to 128 on the P5-CLUSTER. The addition of edges is a fully asynchronous parallel operation. Adding vertices in the dynamic `pGraph` causes asynchronous communication to update the directory infor-

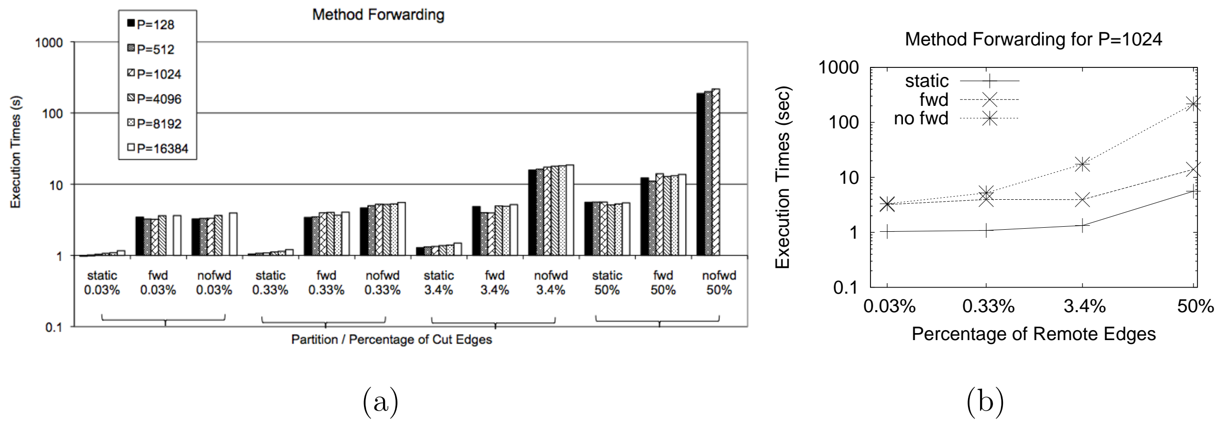
mation about where vertices are actually stored. The asynchronous communication overlaps well with the local computation of adding the vertices in the `bContainer`, thus providing good scalability up to a very large number of processors. There is only 36% increase in execution time for `add_vertex` in the dynamic `pGraph` as we scale from 4 to 16384 processors.

2. Evaluation of Address Translation Mechanisms

In this section we evaluate the performance of the three types of address translation mechanisms introduced in Section D: a static partition with a closed form solution mapping `GIDs` to `bContainers`, and distributed dynamic partitions with and without method forwarding. When method forwarding is not allowed, the `partition` fetches the `GID` mapping information using synchronous operations. When method forwarding is allowed, the method is asynchronously sent first to the location owning the directory which in turn forwards the method to the actual location where the element resides.

We evaluate the performance of the three partitions using a simple `pGraph` algorithm that finds source vertices (i.e., vertices with no incoming edges) in a directed graph. The algorithm traverses the adjacency list of each vertex and increments a counter on the target vertex of each edge. The communication incurred by this algorithm depends on the number of remote edges, i.e., edges connecting vertices in two different `bContainers`. We considered four graphs, all 2D tori, which vary according to the percentage of remote edges: .03%, .33%, 3.4%, and 50%. This was achieved by having each processor hold a stencil of 1,500×1,500, 150×15,000, 15×150,000, 1×2,250,000, respectively.

Figure 51(a) provides a summary of the execution times for the different percentages of remote edges and different numbers of processors, where scalability can



(a) (b)

Fig. 51. Find sources in a directed `pGraph` using static, dynamic with forwarding and dynamic with no forwarding partitions. Execution times for graphs with various percentages of remote edges for (a) various processor counts and for (b) 1024 processors.

be appreciated together with the increasing benefit of forwarding as the percentage of remote edges increases. In Figure 51(b) we include results for the three approaches on all four types of graphs for 1024 processors. As can be seen, for the methods with no forwarding and synchronous communication, the execution time increases as the percentage of remote edges increases. The static method and the method with forwarding track one another and do not suffer as badly as the percentage of remote edges increases. This indicates that the forwarding approach can scale similarly to the optimized static partition.

In Figure 52 we show weak scaling experiments for all four graph types. While the scalability is good for all methods, the static partition is always superior and, again, it is seen that for the dynamic partitions, the benefit of method forwarding increases as the percentage of remote edges increases, though it is quite significant even if the percentage of remote edges is quite low (e.g., 3.4%). When we use a 1x2250000

stencil (50% remote edges), `partition_static` is around 5 seconds, `partition_fwd` is 12 seconds, while `partition_nofwd` is 190 seconds.

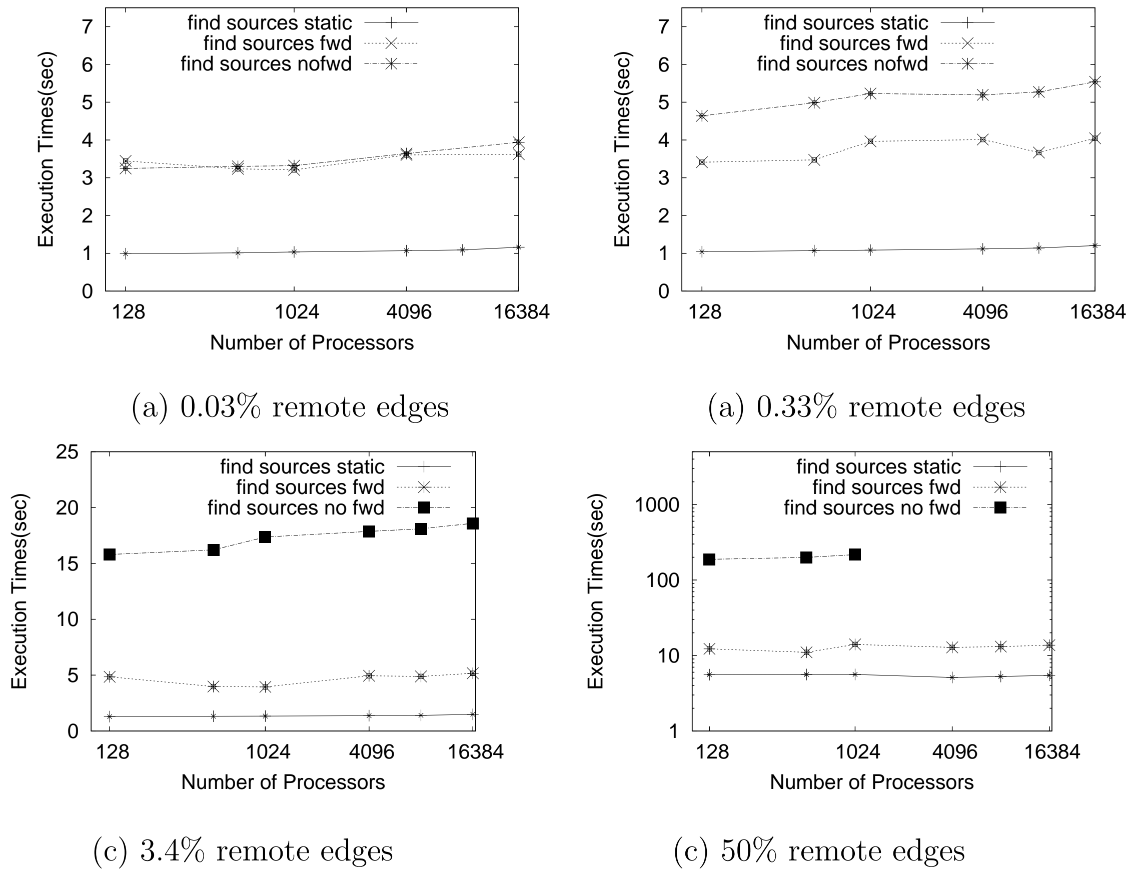
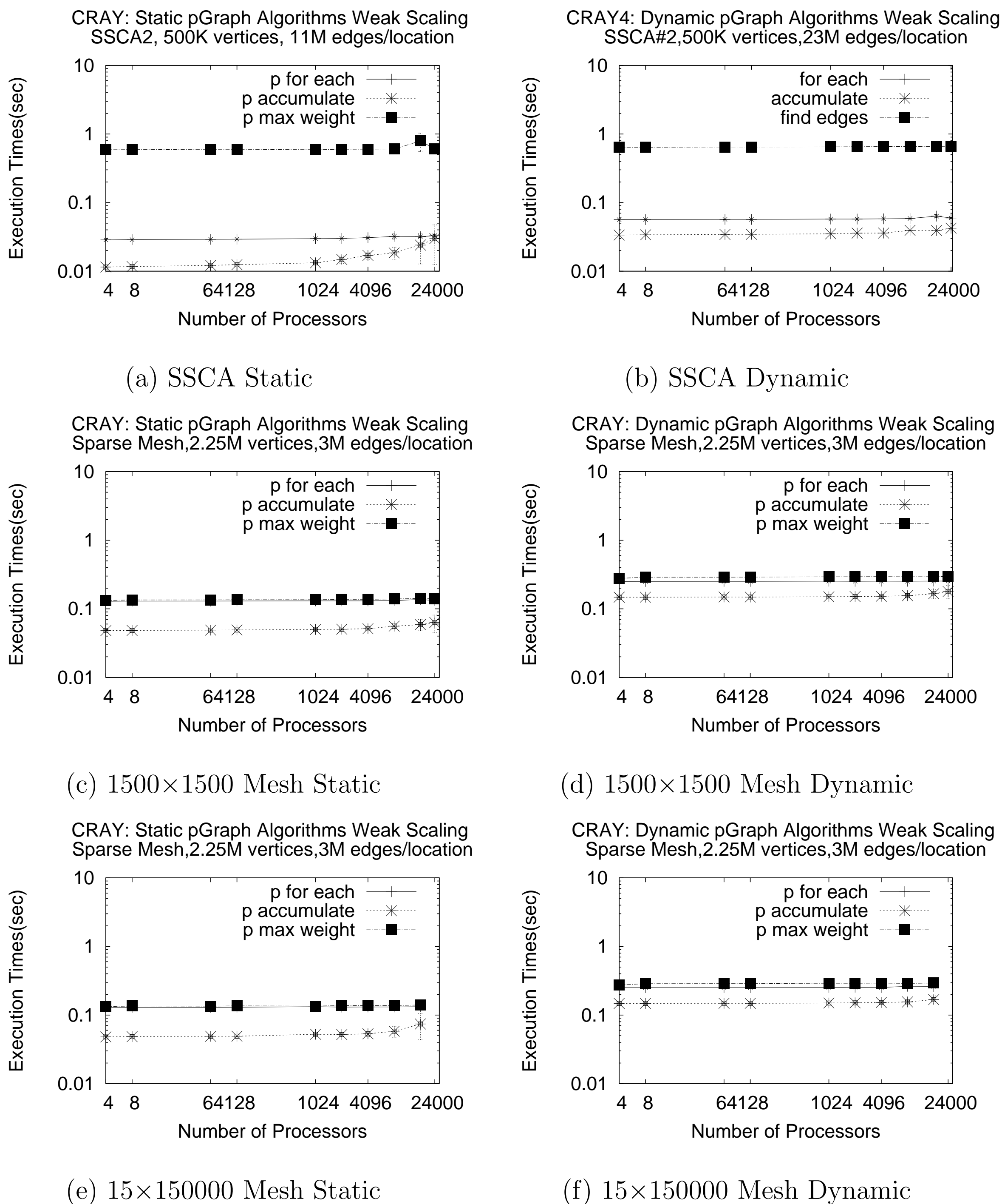


Fig. 52. Comparison of various `pGraph` partitions. Execution times (weak scaling) for graphs with (a) .03%, (b) .33%, (c) 3.4%, and (d) 50% remote edges.

3. pGraph Algorithms

In this section we analyze the performance of several generic STL algorithms and `pGraph` specific algorithms for various input types and `pGraph` characteristics. In Figure 53 we show results for three generic STL algorithms. `p_for_each` applies to every vertex in the graph a functor that sets the vertex property which is a double to zero. The theoretical complexity for this operation is $O(|V|/P)$, where $|V|$ is the number of vertices in the graph and P is the number of processors. `p_accumulate` accumulates all vertex properties. The theoretical complexity for this operation is $O(|V|/P+\log(P))$. The $\log(P)$ factor is due to the reduction performed. The `p_max_weight` find the edge with the maximum edge weight. Its complexity is $O(|E|/P)$, where $|E|$ is the total number of edges. We observe from the figure good scaling for all three algorithms for both static and dynamic `pGraphs`. The `p_accumulate` execution time slowly increases as we increase the number of processors and this is due to the reduction step. For all `pGraphs` considered the number of edges is bigger than the number of vertices and this is properly reflected in the graphs.

In Figures 54 and 55 we include results for graph specific algorithms on CRAY4 and P5-CLUSTER, respectively. The `find_edges` collects all edges with maximum edge weight into an output `pList` (SSCA2 benchmark); `find_sources` collects all vertices with no incoming edges into an output `pList`. `find_sources` takes as input a collection of vertices and performs graph traversals in parallel. The traversal proceeds in a DFS style. When a remote edge is encountered, a new task is spawned to continue the traversal on the location owning the target. The current traversal will continue in parallel with the spawned one. This is useful for example when we want to compute all vertices and edges accessible from a set of starting points. `trim` is another useful computation when computing cycles or strongly connected compo-



(a) SSCA Static

(b) SSCA Dynamic

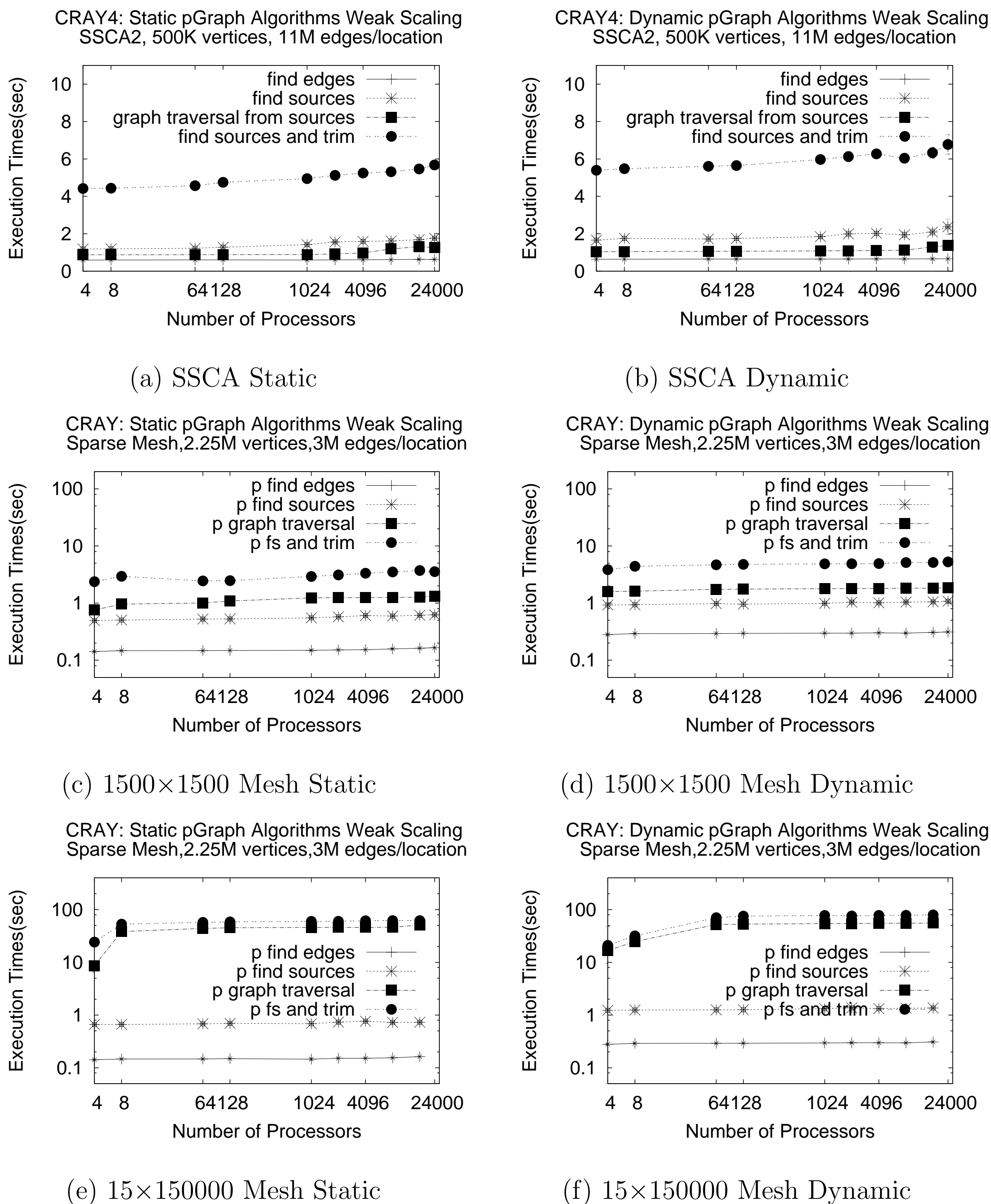
(c) 1500×1500 Mesh Static

(d) 1500×1500 Mesh Dynamic

(e) 15×150000 Mesh Static

(f) 15×150000 Mesh Dynamic

Fig. 53. CRAY4: Execution times for different `pGraph` algorithms. Static versus dynamic `pGraph` comparison. The input is a sparse mesh or generated using the SSCA2 scalable generator with 500K vertices per processor.



(a) SSCA Static

(b) SSCA Dynamic

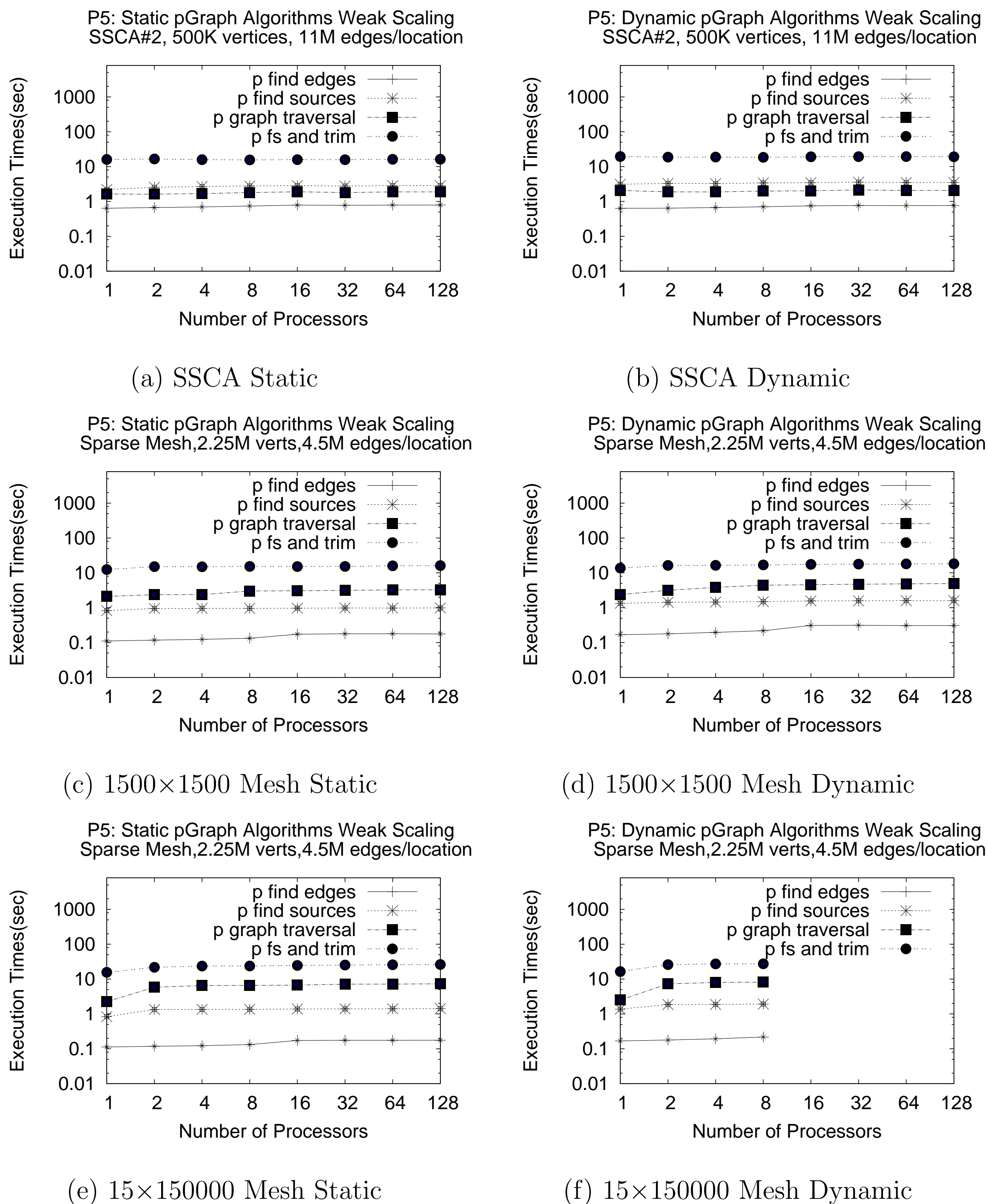
(c) 1500×1500 Mesh Static

(d) 1500×1500 Mesh Dynamic

(e) 15×150000 Mesh Static

(f) 15×150000 Mesh Dynamic

Fig. 54. CRAY4: `pGraph` algorithms. Static versus dynamic `pGraph` comparison. The input is a sparse mesh or generated using the SSCA2 scalable generator with 500K vertices per processor.



(a) SSCA Static

(b) SSCA Dynamic

(c) 1500×1500 Mesh Static

(d) 1500×1500 Mesh Dynamic

(e) 15×150000 Mesh Static

(f) 15×150000 Mesh Dynamic

Fig. 55. P5-CLUSTER: Execution times for different `pGraph` algorithms. Static versus dynamic `pGraph` comparison. The input is a sparse mesh or generated using the SSCA2 scalable generator with 500K vertices per processor.

nents. It computes the set of sources for a directed graph and removes all their edges, recursively continuing with the newly created sources. The process will stop when there are no more sources.

We run the algorithms on various input types including a sparse mesh and SSCA2 random graphs. In Figure 54(a), (b), and Figure 55(a), (b), weak scaling results are shown for SSCA2 for both static and dynamic `pGraphs`. The number of processors varied from 4 to 24000. For all algorithms considered, the static graph performed better due to the faster address resolution and `std::vector` storage for vertices versus `std::hash_map`. `find_edges`, a fully parallel algorithm, exhibits good scalability with less than 5% increase in execution time for both types of graphs on CRAY4. `find_sources` incurs communication proportional to the number of remote edges. The algorithms use two containers, traversing an input `pGraph` and generating an output `pList`. The traversal from sources and trim algorithm spawn new computation asynchronously as it reaches a remote edge. Additionally, the `trim` algorithm removes `pGraph` edges, which negatively impacts performance. The increase in execution time for the trim algorithm is 28% for static and 25% for dynamic `pGraphs` on CRAY4.

Figure 54(c), (d), (e) and (f) illustrate that the execution time of `pGraph` algorithms increases with the number of remote edges. When the 1500×1500 stencil is used the number of remote edges is small relative to the local ones and there is good communication computation overlap enabling the algorithms to scale up to a large number of processors. When the stencil used is 15×150000, the remote to local edges ratio is 3.4%. The increased number of remote edges is reflected in the execution times of the algorithms because they incur communication proportional to the number of edges. Despite an increased execution time we observe that the algorithms scale well up to 24000 processors, proving that the STAPL `pGraph` data structure can be successfully used to solve very large graph problems.

4. Page Rank

In this section we examine the performance of the page rank [12] algorithm. The algorithm performs a number of iterations and in each iteration, for all vertices, we update the ranks of all neighbor vertices based on the rank of the current vertex. The algorithm incurs communication proportional to the number of remote edges. In Figure 56 we show experimental results for two different meshes, one with 0.03% remote edges and one with 3.4% remote edges per location. The algorithm scales well as we scale the number of processors from 4 to 8192, the communication cost being visible only on the larger number of processors.

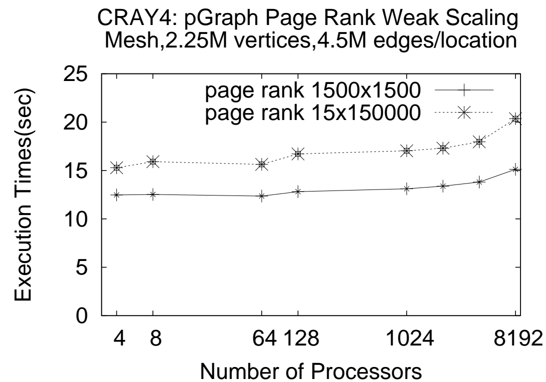


Fig. 56. Page rank for two different input meshes: 1500x1500 and 15x150000.

CHAPTER XII

ASSOCIATIVE PCONTAINERS*

An associative container provides optimized methods for storing and retrieving data using keys. In STAPL, similar to STL [49], we consider the following six basic associative container concepts: *simple, pair, sorted, hashed, unique* and *multiple*. Simple specifies that the container will store only keys while pair means that the container will store pairs of keys and values. Sorted guarantees that the internal organization allows logarithmic time implementations for insert, delete and find operations, while hashed containers guarantee asymptotic constant time for these operations. In addition, traversing the data of a sorted associative container from begin to end guarantees that the elements are traversed in sorted order. Unique guarantees that all data elements have unique keys, while multi allows for duplicate keys. Each of these concepts specifies properties and interfaces, e.g., simple associative `pContainer` methods have keys in the interface (e.g., sets), while pair associative `pContainers` have methods with both keys and values (e.g., maps), hashed and sorted associative `pContainers` specify complexity requirements, and single or multi specify the semantics of the operations.

Based on this taxonomy, STAPL provides six associative `pContainers` that are compositions of the basic concepts (see Figure 57(b)): `pSet` (simple, sorted, unique), `pMap` (pair, sorted, unique), `pMultiSet` (simple, sorted, multiple), `pMultiMap` (pair, sorted, multiple), `pHashMap` (pair, hashed, unique), and `pHashSet` (simple, hashed,

*Part of the data reported in this chapter is reprinted with the kind permission of Springer Science+Business Media from "Associative parallel containers in STAPL," by G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger, 2008. *Lecture Notes in Computer Science*, vol. 5234, pp. 156–171, Copyright 2008 by Springer.

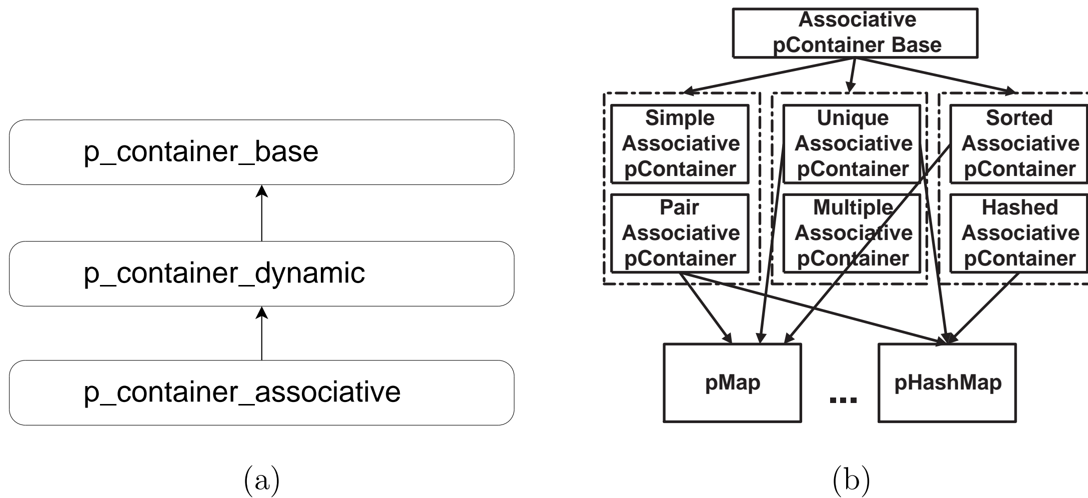


Fig. 57. Associative `pContainer`: (a) derivation from the framework base classes (b) associative `pContainers` internal hierarchy.

unique).

A. Associative pContainer Specification

The template declarations for the six STAPL associative `pContainers` are:

```
template <class Key, Class Value, class Compare,
          class Partition=Default, class Traits=Default>
class p_map;
template <class Key, Class Value, class Compare,
           class Partition=Default, class Traits=Default>
class p_multi_map;
template <class Key, class Compare,
          class Partition=Default, class Traits=Default>
class p_set;
template <class Key, class Compare,
          class Partition=Default, class Traits=Default>
class p_multi_set;
template <class Key, class Value, class Hash,
          class Partition=Default, class Traits=Default>
```

```
class p_hash_map;
template <class Key, class Hash
         class Partition=Default, class Traits=Default>
class p_hash_set;
```

The STAPL associative pContainers provide the generic specification (data types and methods) included in table XXVIII. The complexity of all element-wise methods is $O(\log(N))$ for sorted where N is the pContainer size, and amortized constant time for hashed.

Table XXVIII.: Associative pContainers interface.

Template Arguments	**Description**
Traits	Traits to specify the low level base container used and distribution features.
Define Type	**Description**
key_type	the type of the Key
value_type	the type of the Value (not available for simple associative)
key_compare	the type for key comparisons (not available for hashed)
Method	**Description**
iterator insert(key[,value])	insert the (key,value) pair (no value for simple associative). Return iterator pointing to inserted item.
size_t erase(key)	Erases all elements with key equal to k. Return number of erased elements.
iterator find(key)	Return an iterator pointing to an element with key equal to k or end() if no such element is found.
void insert_async(key[,value]), void erase_async(key)	Non-blocking insert/erase (no value for simple associative)
key find_val(key)	blocking operations returning values (instead of iterators).

All STL equivalent methods require a return type, which in general translates into a synchronous (blocking) method. For this reason, we provide a set of asynchronous methods as part of the associative `pContainer`, e.g., `insert_async` and `erase_async`. These non-blocking methods allow for better communication/computation overlap and enable the STAPL RTS to aggregate messages to reduce the communication overhead.

We also introduce new associative `pContainer` methods that return values instead of iterators. These methods are provided because in STAPL a remote call will be issued when an iterator to a remote element is dereferenced. Hence, if a programmer knows the value will be needed, they should use the method that returns a value rather than the method that returns an iterator.

B. Associative pContainer Design and Implementation

In this section, we describe the `pList` modules used for storage and data distribution information.

bContainer: We have implemented the associative `pContainer` `bContainers` by extending the corresponding sequential container (typically STL containers) with functionality needed to implement domain instances.

Global Identifier (GID): For a simple associative `pContainer` the GID associated with each element is a key, whereas it is a (key, m) pair for a multi associative `pContainer`, where m is an integer used to manage multiplicity

Domain and Domain Instance: The domain of the associative `pContainer` is given by the range of possible keys the `pContainer` can hold. For example, for a `pMap` over strings the domain can be the set of all possible strings or the set of all possible strings between two boundaries according to some order relation (e.g., lexicographical

```
template<class Domain>
class partition{
 partition(vector<Domain>&);
 //compute the sub-domain
 //to which the GID is associated
 BCID map(GID);
}
typedef
 associative_domain<string,
      lexi_compare> Domain;
vector<Domain> doms;
doms.push_back(Domain('a'..'d');
doms.push_back(Domain('d'..'z');
partition_strategy(doms);
```

Fig. 58. Value based partition for sorted associative `pContainers`.

order). At any instant, there is only a finite set of elements in the container. The GIDs associated with these elements is the *domain instance* of the `pContainer`. For example `AssociativeDomain<string>('a','k')` is a domain comprising all strings that are greater than `'a'` and strictly smaller than `'k'` according to the lexicographical order. A domain instance corresponding to the previously defined associative domain might be {`'a', 'aa', 'abc', 'joe'`}.

Data Distribution: The data distribution manager uses (i) a `partition` to decide for every key in the domain to which sub-domain it has been allocated, and (ii) a `partition-mapper` to decide to which location each sub-domain has been allocated.

Partition: Associative `pContainers` are dynamic containers supporting concurrent additions and deletions of elements, thus the corresponding partitions have to provide functionality to add or delete GIDs to/from the corresponding domain instance or, e.g., to perform repartitions to ensure load balance. The default partition implemented by STAPL sorted associative `pContainers` is a static blocked partition over the key space. Users can provide additional partitions for associative

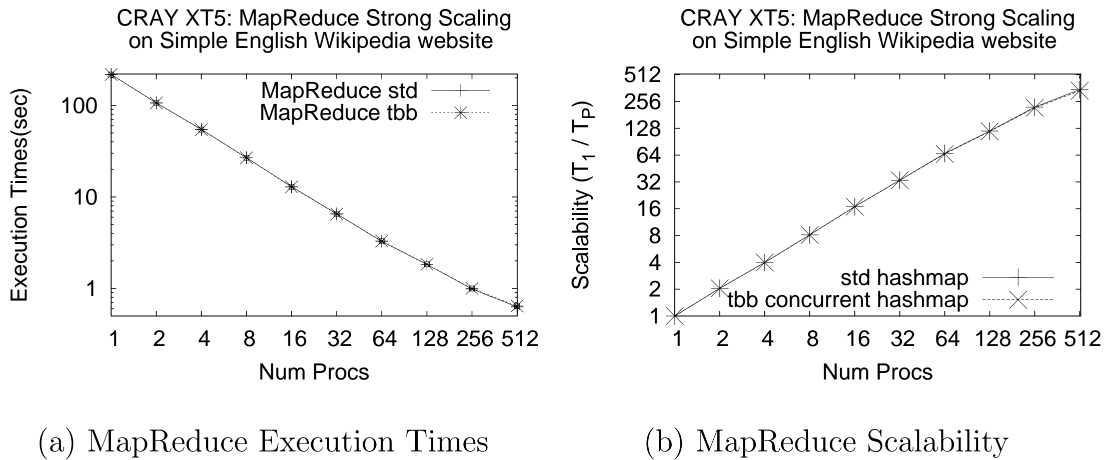
`pContainers` by explicitly enumerating the corresponding sub-domains as illustrated in Figure 58. For a hashed associative `pContainer`, the partition can be specified by providing a hash function that will map a key to a sub-domain ID (e.g., `hash(key) mod num_subdomains`).

Partition Mapper: Similar to other STAPL `pContainers` associative containers can use any of the following partition mappers: `partition_mapper_generic`, `partition`
`mapper_blocked`, `partition_mapper_cyclic` or `partition_mapper_identity`.

Associative pContainer pViews: `pViews` are defined as the accessors for the data elements stored in the `pContainer`. The `pViews` over `pMap`, `pMultiMap`, and `pHashMap` support mutable iterators over data. This allows the value field to be modified. The others (`pSet`, `pMultiSet`, and `pHashSet`) provide read only `pViews` with `const` iterators.

Associative pContainer Base Class: To automate and standardize the process of developing associative `pContainers`, we designed a common base that is responsible for maintaining the data and the distribution manager. The associative `pContainer` base is generic and uses template parameters and traits classes to tailor the data structure to the user's needs. Each basic associative concept (simple, pair, unique, multi, sorted, hashed) is implemented as a class derived from the associative `pContainer` base to provide the specified functionality and enforce the required properties. Each associative `pContainer` (e.g., `pMap`), inherits from three corresponding classes as depicted in Figure 57(b).

The time for performing the operation on the `bContainers` is logarithmic or amortized constant time for sorted and hashed `pContainers`, respectively. The memory overhead depends on the partition used. A blocked partition for a sorted `pContainer` requires space proportional to the number of sub-domains, while for a



(a) MapReduce Execution Times (b) MapReduce Scalability

Fig. 59. MapReduce used to count the number of occurrences of every word in Simple English Wikipedia website (1.5GB).

hashed partition the overhead is constant in each location. Different partitions, with more complex invariants, may incur different computational and memory overheads.

C. Performance Evaluation

In this section, we evaluate the scalability of the parallel methods using a map reduce application and we evaluate two generic `pAlgorithms`, `p_for_each` and `p_accumulate`.

1. MapReduce

Here we examine the performance of a simple application implemented on top of a MapReduce framework developed in STAPL. The MapReduce uses the `pHashMap`[64], a dynamic associative `pContainer`. The application splits the input data across the available processors and first applies the map and reduce functions locally. After the local MapReduce phase is finished, the processor asynchronously inserts its locally reduced data into a `pHashMap`. The asynchronous insert calls the user's reduce func-

tion if the key being inserted already exists in the pHashMap. The communication and data distribution is taken care of entirely by the pContainer. We ran a computation that computes the mutliplicity of each word in a 1.5GB text input of the Simple-English Wikipedia website (simple.wikipedia.org). Because the input size was fixed and given, we include a strong scaling study where we measure the time taken to compute the multiplicity for all input words on CRAY5. In Figure 59 we show experiments corresponding to two different pHashMap storages, one using the STL std::hash_map and another using the TBB concurrent hash map. We observe that the application scales well up to 512 processors and there is no noticeable difference when using different storages. The slowdown on 256 and 512 processors is due to the small computation performed per processor relative to the communication required to insert the data into the pHashMap.

2. Generic Algorithms

In this section, we examine the performance of various generic parallel algorithms operating on a linearization of the associative pContainer's data. Figure 60 shows the performance for p_for_each when operating on a single element pView defined over the pMap. As we scale the number of processors from 4 to 16384 we observe good scalability for all three algorithms considered: p_for_each, p_accumulate, and finding the maximum value in the container.

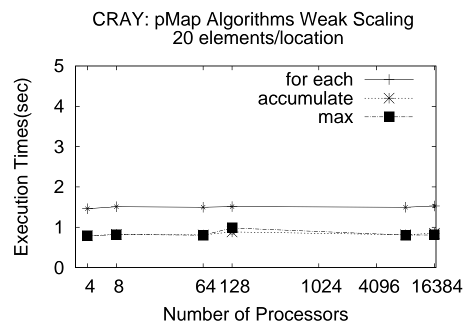


Fig. 60. CRAY4: Scalability for generic algorithms when using associative `pContainers`.

CHAPTER XIII

PCONTAINER COMPOSITION EVALUATION

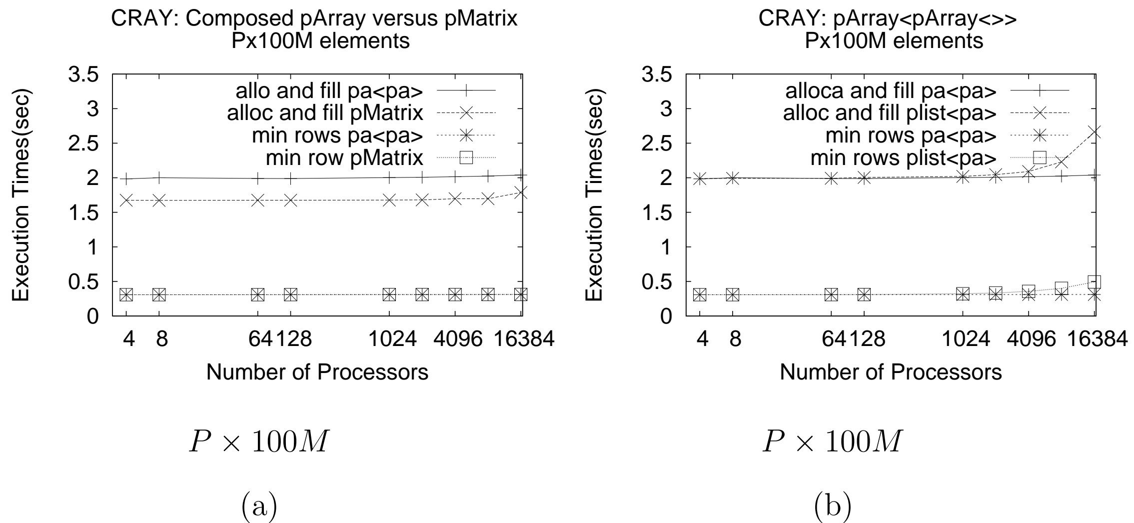
In Chapter V, Section C we introduced the `pContainer` composition as one of the main mechanisms for extending our library with custom data structures. We argue that composition helps increase the programmer productivity. Instead of directly building a complex `pContainer`, the programmer can compose one from the basic `pContainers` available in the PCF and shorten the development and debug time. For example, multi-dimensional arrays can be expressed as the composition of `pArrays` or `pMatrices` or a large scale distributed `pGraph` can be expressed as a `pList` of vertices where each vertex stores another `pList` of edges. In this section, we include experimental results to study the performance overhead of composed data structures relative to custom made ones.

For this comparison, we use a simple application that computes the minimum element in each row of a matrix using a `pMatrix` `pContainer` (which is available in the PCF library), a composed `pArray` of `pArrays`, and a composed `pList` of `pArrays`. The algorithm code is the same for all `pContainers` used, due to the access abstraction mechanism provided by STAPL `pViews`. It calls a parallel `for_each` on each row, and within each row, a `map-reduce` to compute the minimum value. The code for the algorithm is shown in Figure 61. We measure also the time to create and initialize the storage. The `pMatrix` allocates the entire structure in a single step (Figure 61, line 16), while the `pArray` of `pArrays` allocates the outer structure first (Figure 61, line 14) and subsequently allocates the nested `pArrays` in parallel using a parallel `for_each` (Figure 61, line 20). In Figure 61, line 23 a parallel `for_each` is invoked on a `pView` defined over the elements of the outer `pArray`. The functor of the parallel `for each` contains a nested `pAlgorithm` invocation (find minimum) that

```
1  struct resize_init{
2    void operator()(pa_ref& pa_view){
3      pa_view.resize(M)
4      p_generate(pview(pa_view),rand());
5   }}
6
7   struct min_row{
8   void operator()(row_ref& _view, res_ref& res){
9      res = p_min_element(pview(pa_view));// nested pAlgorithm
10  }}
11
12 main () {
13  //composed parray of parrays
14  p_array<p_array<int> > cpa (N);
15  //pMatrix
16  p_matrix<int> pm(N, M);
17  //result parray with minimum of each row
18  p_array<int> result (N);
19  //resize each of the neted parrays
20  p_for_each(pview(cpa),resize_init_nested(M))
21
22  //call minimum of each row on the composed pArray
23  p_for_each(pview(cpa),pview(result),min_row());
24
25  //call minimum of each row on the pMatrix
26  p_for_each(row_pview(pm),pview(result),min_row());
27
28 }
```

Fig. 61. Example of `pContainer` composition and nested `pAlgorithm` invocation.

is applied to each of the nested `pArrays`. In Figure 61, Line 26 the exact code used for computing the minimum of each nested `pArray` is used to compute the minimum of each row of the `pMatrix` by using appropriate `pViews`. This time the parallel `for_each` is invoked over a rows `pView` defined over `pMatrix` data and the functor to be applied to each row contains a nested parallel minimum algorithm. This simple



$P \times 100M$ (a) $P \times 100M$ (b)

Fig. 62. Comparison of pArray<pArray<>> $(pa < pa >)$, pList<pArray<>> $(plist < pa >)$ and pMatrix on computing the minimum value for each row of a matrix. Weak scaling experiment with $P \times 100M$ elements. For the composed pContainer the outer pContainer is of size P and the inner pContainer is of size 100M (a) pArray<pArray<>> versus pMatrix: pArray<pArray<>> takes longer to initialize while the algorithm executions are very similar (b) pArray<pArray<>> versus plist<pArray<>>: The plist<pArray<>> has additional overhead when creating the pViews over its data

example shows both the benefits of composition and the abstraction power provided by the pViews that allows users to easily assemble a parallel application.

In Figure 62(a) we include, for CRAY4, the execution times for allocating and initializing the composed pArray and the pMatrix and the times to run the min-of-each-row algorithm, in a weak scaling experiment. The input used is a $P \times 100M$ matrix where P is the number of processors. In all experimens considered in this section the nested pArrays store their data on a single location and the nested pAlgorithms are executed by a single processor. These are preliminary results of the prototype

of this mechanism. As expected, the pArray of pArrays allocation and initialization time is higher than that for a pMatrix. The time for the composed pArray includes the time of executing a parallel algorithm. The time for min-of-each-row algorithm, is very similar for the two data structures and scales well up to 16384 processors.

In Figure 62(b) we compare two composed pContainers. The composed pArray described in the previous paragraph and a composed pList of pArrays. In this case we observe similar times for the initialization of the data and computing the minimum for the low processor counts. While increasing the number of processors both the time for initialization and the time to run the algorithm increase for the composed pList much faster than the composed pArray. This is mainly due to the overhead of creating the pView abstractions on top of the pList container. Some of the overhead in the pView creation is subject to further research.

While we cannot state with certainty that our PCF allows for efficient composition (no additional overhead) for any combination of pContainers, the presented experiments indicate it is possible.

CHAPTER XIV

CONCLUSION AND FUTURE WORK

In this dissertation, we presented the STAPL Parallel Container Framework (PCF), an infrastructure to facilitate the development of parallel and concurrent data structures. The salient features of this framework are: (a) a set of classes and rules to build new `pContainers` and customize existing ones, (b) mechanisms to generate wrappers around any sequential or parallel data structure, enabling its use in a distributed, concurrent environment and their use in cooperation with other libraries, (c) support for the (recursive) composition of `pContainers` into nested, hierarchical `pContainers` which can support arbitrary degrees of nested parallelism and (d) a library of basic `pContainers` constructed using the PCF as initial building blocks. We have shown how we have implemented a *shared object view* of the `pContainers` on distributed systems in order to relieve the programmer from managing and dealing with the distribution explicitly, unless so desired. The PCF allows users to customize its `pContainers` and adapt to dynamic and irregular environments, e.g., a `pContainer` can dynamically change its data distribution or adjust its thread safety policy to optimize the access pattern of the algorithms accessing the elements. Alternatively, the user can request certain policies and implementations which can override the provided defaults or adaptive selections. The PCF is an open ended project where users can add features as well as to the library and thus continuously improve the PCF's performance and utility.

Our experimental results on a very large parallel machine available at NERSC and a Power 5 cluster at Texas A&M University supercomputing center show that `pContainers` provide good scalability for both static and dynamic `pContainers`.

The `pContainer` framework we developed enables a large number of new research

directions that can be further pursued. First, our research will help users be more productive while developing new `pContainers` tuned for specific applications. For example motion planning [68] applications use a roadmap as their main data structure which is a natural extension of the graph. Using the PCF researchers in this area are provided with automatic support for parallelism by deriving the roadmap from the `pGraph` `pContainer`. Other applications such as particle transport use regular or arbitrary discretizations of the space called grids which can be naturally expressed as extensions of the `pGraph` data structure. We envision that for applications like this and numerous others, deriving data structures from base classes already provided by a library will be an important boost for user productivity.

Implementing various thread safety policies is another dimension that can be further exploited in our framework. The framework provides the proper interfaces for the thread safety manager as described in Chapter VI. Additionally, a set of predefined implementations are available but they have been minimally evaluated due to current limitations of the STAPL runtime system. With multithreading support available where multiple threads can be active within one locations all this functionality can be exercised and novel solutions proposed.

We described in Chapter VII the default relaxed memory consistency model provided by the `pContainers` developed in our framework. We have chosen the current model as it provides a good trade off between programmability and performance. However as mentioned in Chapter VII, Section E, other memory models more restrictive or more relaxed are possible. An interesting research direction is to study alternative models and evaluate their impact on productivity and performance.

`pContainer` composition as described in Chapter IV, Section C and Chapter XIII is a novel feature proposed in our framework as a modality to express new data structures. The composition opens a large number of research directions that require

further exploration. These include deciding what is the optimal data distribution at different levels of the hierarchy, how data can be accessed by nested `pAlgorithms` and how composed `pViews` can be defined on data of a composed `pContainer`.

Another research dimension that our framework enables is the possibility to adapt a data structure to specific applications and architectures. We mentioned throughout this thesis that one of the major design goals of the library is to allow users to specialize functional modules of a `pContainer` by implementing well defined interfaces. Adaptivity will allow a `pContainer` to select among various modules with similar functionality. For example, different partitions and distributions may be available for a particular `pContainer` and currently the user decides which one to use. We envision that a `pContainer` can be augmented with the necessary support to perform the selection automatically.

In addition to providing users with a very large number of data structures, thus improving productivity, we believe that STAPL and the PCF are flexible infrastructures allowing researchers to experiments with data structures as a whole or with the individual functional modules that make up a parallel data structure.

REFERENCES

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[2] S. V. Adve and H. J. Boehm, "Memory models: A case for rethinking parallel languages and hardware," *Commun. ACM*, vol. 53, no. 8, pp. 90–101, 2010.

[3] S. V. Adve and M. D. Hill, "Weak ordering—a new definition," In *Proc. of the 17th annual Int. Symposium on Computer Architecture*, New York, NY, USA, 1990, pp. 2–14.

[4] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "STAPL: A standard template adaptive parallel C++ library," In *Proc. of the Int. Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Bucharest, Romania, Jul 2001, pp. 37–46.

[5] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger, "STAPL: An adaptive, generic parallel C++ library," In *Int. Workshop on Languages and Compilers for Parallel Computing, in Lecture Notes in Computer Science*, vol. 2624, pp. 195–210, 2003.

[6] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, London: McGraw-Hill, 1998.

[7] D. A. Bader and K. Madduri, "Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors," In *The 12th Int. Conf. on High Performance Computing (HiPC 2005)*, pp. 465–476. Springer, 2005.

[8] G. Bikshandi, J. Guo, C. Praun, G. Tanase, B. B. Fraguela, M. J. Garzaran, D. Padua, and L. Rauchwerger, "Design and use of htalib: A library for hierarchically tiled arrays," In *Int. Workshop on Languages and Compilers for Parallel Computing, in Lecture Notes in Computer Science*, vol. 4382, pp. 17–32, 2007.

[9] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B.B. Fraguela, M.J. Garzaran, D. Padua, C. Praun, "Programming for parallelism and locality with hierarchically tiled arrays," In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog.*, New York, NY, 2006, pp. 48-57.

[10] G. Blelloch, "NESL: A nested data-parallel language," Dept. Comp. Sci., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep., CMU-CS-95-170, 1993.

[11] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha, "Implementation of a portable nested data-parallel language," In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog.*, 1993, pp. 102–111.

[12] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, no. 1-7, pp. 107–117, 1998.

[13] A. Buss, A. Fidel, Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The STAPL pView," In *Int. Workshop on Languages and Compilers for Parallel Computing*, Houston, TX, 2010.

[14] 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., College Station, TX, Tech. Rep., TR10-001, July 2010.

[15] 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 *Int. Workshop on Languages and Compilers for Parallel Computing, in Lecture Notes in Computer Science*, vol. 5335, pp. 304–315, July 2008.

[16] 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 the 3rd Annual Haifa Experimental Systems Conf.*, pp. 1–10, 2010.

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

[18] 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*, New York, NY, 2005, pp. 519–538.

[19] D. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick, "Parallel programming in Split-C," In *Int. Conf. on Supercomputing*, November 1993.

[20] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *Proc. of OPODIS*, Bordeaux, France, 2006, pp. 142–156.

[21] J. Desouza and L. V. Kale, "MSA: Multiphase specifically shared arrays," In

Int. Workshop on Languages and Compilers for Parallel Computing, in Lecture Notes in Computer Science, vol. 3602, pp. 268–282, 2005.

[22] R. E. Diaconescu and H. P. Zima, "An approach to data distributions in chapel," *Int. J. of High Performance Computing Applications*, vol. 21, no. 3, pp. 313–335, 2007.

[23] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," In *ISCA '98: 25 years of the Int. Symposium on Computer Architecture*, New York, NY, 1998, pp. 320–328.

[24] H. Gao, J. F. Groote, and W. H. Hesselink. Almost wait-free resizable hashtables. *18th International Parallel and Distributed Processing Symposium*, Santa Fe, New Mexico, 2004.

[25] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared-Memory Programming.* Hoboken, NJ: Wiley-Interscience, 2003.

[26] M. Greenwald. Two-handed emulation: how to build non-blocking implementations of complex data-structures using DCAS. In *Proc. of the Twenty-first Annual Symposium on Principles of Distributed Computing (PODC)*, Monterey, Ca, 2002, pp. 260–269.

[27] M. Fomitchev and E. Ruppert, "Lock-free linked lists and skip lists," In *Proc. Symp. on Princ. of Distributed Programming*, New York, NY, 2004, pp. 50–59.

[28] P. Gottschling, D. S. Wise, and M. D. Adams, "Representation-transparent matrix algorithms with scalable performance," In *Proc. Int. Conf. on Supercomputing*, Seattle, Washington, 2007, pp. 116–125.

[29] 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.

[30] D. Gregor and A. Lumsdaine, "The parallel BGL: A generic library for distributed graph computations," In *Proc. of Workshop on Parallel Object-Oriented Scientific Computing*, July 2005.

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

[32] B. Hendrickson and R. Leland, *The Chaco User's Guide Version 2.* Sandia National Laboratories, Albuquerque NM, 1995.

[33] M. Herlihy, "A methodology for implementing highly concurrent data structures," In *Proc. of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, Washington, 1990, pp. 197–206.

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

[35] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming.* San Francisco, CA: Morgan Kaufmann Publishers Inc., 2008.

[36] Intel. *Reference Manual for Intel Threading Building Blocks, version 1.13.* Intel Corp., Santa Clara, CA, 2009.

[37] Intel. *Reference Manual for Intel Threading Building Blocks, version 1.0.* Intel Corp., Santa Clara, CA, 2006.

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

[39] E. Johnson, "Support for Parallel Generic Programming". PhD thesis, Indiana University, Indianapolis, 1998.

[40] E. Johnson and D. Gannon, "HPC++: Experiments with the parallel standard template library," In *Proc. Int. Conf. on Supercomputing*, Vienna, Austria, 1997, pp. 124–131.

[41] 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.

[42] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," J. of Parallel and Distributed Computing, vol. 48, no. 1, pp. 96-129, 1998.

[43] H. T. Kung and P. L. Lehman. "Concurrent manipulation of binary search trees," *ACM Trans. Database Syst.*, vol. 5, no. 3, pp. 354–382, 1980.

[44] K. Mehlhorn and S. Naher, *LEDA: A Platform for Combinatorial and Geometric Computing.* New York: Cambridge University Press, 1999.

[45] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," In *Proc. of the fourth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, New York, NY, 1991, pp. 63–74.

[46] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. C-28, no. 9,

pp. 690 –691, Sep. 1979.

[47] P. L. Lehman and S. B. Yao. "Efficient locking for concurrent operations on b-trees," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, 1981.

[48] 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*, Winnipeg, Manitoba, Canada, 2002, pp. 73–82.

[49] D. Musser, G. Derge, and A. Saini, *STL Tutorial and Reference Guide,* Second Edition. Reading, MA: Addison–Wesley, 2001.

[50] J. Moreira, V. Salapura, G. Almasi, C. Archer, R. Bellofatto, P. Bergner, R. Bickford, M. Blumrich, J. Brunheroto, A. Bright, M. Brutman, J. Castanos, D. Chen, P. Coteus, P. Crumley, S. Ellis, T. Engelsiepen, A. Gara, M. Giampapa, T. Gooding, S. Hall, R. Haring, R. Haskin, P. Heidelberger, D. Hoenicke, T. Inglett, G. Kopcsay, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. Mundy, M. Ohmacht, J. Parker, R. Rand, D. Reed, R. Sahoo, A. Sanomiya, R. Shok, B. Smith, G. Stewart, T. Takken, P. Vranas, B. Wallenfelt, M. Blocksome and J. Ratterman, "The Blue Gene/L supercomputer: A hardware and software story," *International Journal of Parallel Programming*, vol. 35, no. 3., pp. 181-206.

[51] W. Pugh, "Concurrent maintenance of skip lists," Univ. of Maryland at College Park, Tech. Rep., UMIACS-TR-90-80, 1990.

[52] L. Rauchwerger, F. Arzu, and K. Ouchi "Standard templates adaptive parallel library (STAPL)," In *Wkshp. on Lang. Comp. and Run-time Sys. for Scal. Comp., in Lecture Notes in Computer Science*, vol. 1511, pp. 402–410, 1998.

[53] J. 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 paralllel architectures," In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++* Cambridge, MA: MIT Press, 1996, pp. 547–588.

[54] S. Saunders and L. Rauchwerger, "Armi: An adaptive, platform independent communication library," In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog.*, San Diego, California, 2003, pp. 230–241.

[55] S. Saunders and L. Rauchwerger, "A parallel communication infrastructure for STAPL," In *Wkshp. on Perf. Opt. for High-Level Languages and Libraries* , New York, NY, Jun 2002.

[56] S. R. Alam, J. A. Kuehn, R. F. Barrett, J. M. Larkin, M. R. Fahey, R. Sankaran, P. H. Worley, "Cray XT4: An early evaluation for petascale scientific simulation," In *Proc. of Supercomputing*, Reno, NV, 2007, pp. 1–12

[57] N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger "Armi: A high level communication library for STAPL," *Parallel Processing Letters*, vol. 16, no. 2, pp. 261–280, 2006.

[58] S. Saunders, "Object Oriented Abstractions for Communication in Parallel Programs". M.S. thesis, Texas A&M University, College Station, 2003.

[59] T. J. Sheffler, "The amelia vector template library," In G. V. Wilson and P. Lu, editors, *Parallel Programming in C++*, Scientific and Engineering Computation Series, pages 43–90, Cambridge, MA: MIT Press, 1996.

[60] J. Siek, L. Lee and A .Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Reading, MA: Addison–Wesley, 2001.

[61] J. Siek and A. Lumsdaine, "The matrix template library: Generic components for high-performance scientific computing," *Computing in Science and Eng.*, vol. 1, no. 6, pp. 70–78, 1999.

[62] D. Skinner, "Performance monitoring of parallel scientific applications," National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, Berkeley, CA, Tech. Rep., LBNL-5503, 2005.

[63] 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.

[64] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger, "Associative parallel containers in STAPL," In *Int. Workshop on Languages and Compilers for Parallel Computing, in Lecture Notes in Computer Science*, vol. 5234, pp. 156–171, 2008.

[65] 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 *Int. Workshop on Languages and Compilers for Parallel Computing, in Lecture Notes in Computer Science*, vol. 5898, pp. 16–30, 2009.

[66] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger, "The STAPL pContainer Framework," In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog.*, San Antonio, TX, 2011, to appear.

[67] 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.*, Chicago, IL, 2005, pp. 277–288.

[68] S. Thomas, G. Tanase, L. K. Dale, J. M. Moreira, L. Rauchwerger, and N. M. Amato, "Parallel protein folding with STAPL," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 14, pp. 1643–1656, 2005.

[69] 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.

[70] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance Java dialect," In ACM, editor, *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY, 1998.

VITA

Gabriel Tanase received his Bachelor of Science from the Polytechnic University of Bucharest, Romania in May 1999. He graduated in the top 5% of his class and his thesis was titled "Adaptive Parallelism using TupleSpace". In May, 2000 he received his Master of Science from the Polytechnic University of Bucharest, Romania. His Master's thesis was titled "Parallel Algorithms for STAPL".

Gabriel Tanase did his Ph.D. studies in the Department of Computer Science at Texas A&M University working with Dr. Lawrence Rauchwerger and Dr. Nancy Amato in the Software & Systems Group of the Parasol Lab. His research interests are in the area of high performance computing, including parallel programming languages and libraries, parallel algorithms and generic programming. He received his Ph.D. in computer science from Texas A&M University in December 2010.

More information about Gabriel Tanase' research and publications may be found at http://parasol.tamu.edu/people/gabrielt. 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 Gabriel Tanase.