Associative Parallel Containers In STAPL[⋆]

Gabriel Tanase, Chidambareswaran Raman, Mauro Bianco, Nancy M. Amato and Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science, Texas A&M University
{gabrielt,chids,bmm,amato,rwerger}@cs.tamu.edu

Abstract. The Standard Template Adaptive Parallel Library (STAPL) is a parallel programming framework that extends C++ and STL with support for parallelism. STAPL provides a collection of parallel data structures (`pContainers`) and algorithms (`pAlgorithms`) and a generic methodology for extending them to provide customized functionality. STAPL `pContainers` are thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. They also provide appropriate interfaces that can be used by generic `pAlgorithms`. In this work, we present the design and implementation of the STAPL associative `pContainers`: `pMap`, `pSet`, `pMultiMap`, `pMultiSet`, `pHashMap`, and `pHashSet`. These containers provide optimal insert, search, and delete operations for a distributed collection of elements based on keys. Their methods include counterparts of the methods provided by the STL associative containers, and also some asynchronous (non-blocking) variants that can provide improved performance in parallel. We evaluate the performance of the STAPL associative `pContainers` on an IBM Power5 cluster, an IBM Power3 cluster, and on a linux-based Opteron cluster, and show that the new `pContainer` asynchronous methods, generic `pAlgorithms` (e.g., `pfind`) and a sort application based on associative `pContainers`, all provide good scalability on more than 10^3 processors.

1 Introduction

Parallel programming is becoming mainstream due to the increased availability of multiprocessor and multicore architectures and the need to solve larger and more complex problems. To help programmers address the difficulties of parallel programming, we are developing the Standard Template Adaptive Parallel Library (STAPL) [1, 21, 23]. STAPL is a parallel C++ library with functionality similar to STL, the ANSI adopted C++ Standard Template Library [18]. 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 containers (`pContainers`), and `views` to abstract the data access in `pContainers`. These are the building blocks for writing parallel programs. An important goal of STAPL

[⋆] This work supported in part by NSF (EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, EIA-9810937, ACI-0326350, CRI-0551685), the DOE and HP.

is to provide a high productivity development environment for applications that can execute efficiently on a wide spectrum of parallel and distributed systems.

Contribution. In this work, we present the STAPL *associative* `pContainers`, a set of data structures intended to be used as parallel counterparts of the STL associative containers. The STAPL associative `pContainers` provide interfaces for the efficient storage and retrieval of their distributed data based on keys. The STAPL associative `pContainers` are thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. They also provide appropriate interfaces (`views`) that can be used to access their distributed elements efficiently in parallel by generic `pAlgorithms`. The methods of the STAPL associative containers include counterparts of the methods provided by the STL associative containers, `insert`, `erase`, and `find`, and also some asynchronous (non-blocking) variants, `insert_async` and `erase_async`, that can provide improved performance in parallel.

We present the design and implementation of the STAPL associative `pContainers`: `pMap`, `pSet`, `pMultiMap`, `pMultiSet`, `pHashMap`, and `pHashSet`. We provide a unified framework for constructing thread-safe, distributed and shared STAPL associative `pContainers` from their corresponding STL counterparts. Our performance evaluation on an IBM Power5 cluster, a large IBM Power3 cluster and on a linux-based Opteron cluster show that the new `pContainer` asynchronous methods, `insert_async` and `erase_async`, generic `pAlgorithms` (e.g., `pfind`), and a sort application based on an associative `pContainer` provide good scalability and low overhead relative to their sequential counterparts.

Outline. The rest of this document is structured as follows: we provide an overview of related work in Section 2, give a high level description of the STAPL library in Section 3, introduce the STAPL associative `pContainers` in Section 4, and present experimental results in Section 5.

2 Related Work

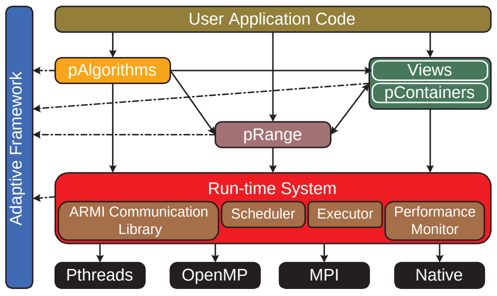
There has been significant research in the field of parallel and concurrent data structures. Much work has focused on providing efficient locking mechanisms and methodologies for transforming existing sequential data structures into concurrent data structures [6–8, 10, 17]. Investigations of concurrent hash tables [7, 8, 17] and search trees (the most common internal representation for maps and sets) [15, 16] explore efficient storage schemes, different lock implementations, and different locking strategies (e.g., critical sections, non-blocking, wait-free [10]), especially in the context of shared memory architectures. In contrast, STAPL associative `pContainers` are designed for use in both shared and distributed memory environments, and we focus on developing an infrastructure that will efficiently provide a shared memory abstraction for `pContainers` (called a shared object view in STAPL) by automating aspects relating to the data distribution and management. We use a compositional approach where data structures (sequential or concurrent) can be used as building blocks for implementing `pContainers`.

There are several parallel languages and libraries that have similar goals as STAPL[2, 3, 5, 9, 14, 19]. While a large amount of effort has been put into making array-based data structures suitable for parallel programming, associative data structures have not received as much attention. The PSTL (Parallel Standard Template Library) project [12, 13] explored the same underlying philosophy as STAPL of extending the C++ STL for parallel programming. PSTL provided distributed associative containers with support for specifying data distributions and local and global iterators for data access. STAPL differs from PSTL by providing an integrated framework for all associative `pContainers`, which also allows users to customize the default behavior, such as specifying different data distributions. PSTL is not an active project. Intel Threading Building Blocks (TBB) [11] provide thread-safe containers such as vectors, queues and hashmaps for shared memory architectures. The TBB *concurrent_hash_map* maps keys to values and the interface provided resembles that of a typical STL associative container, but with some modifications to support concurrent access. In STAPL all associative containers provide both STL compatible interfaces and additional interfaces optimized for parallelism. While TBB was inspired by STAPL, our work is distinguished from TBB in that we target both shared and distributed memory systems. Chapel is a new programming language developed by Cray that is focused on reducing the complexity of parallel programming [4]. The language proposes a formal approach for containers and data distributions, and provides default data distributions and specifies a methodology for integrating new ones. Also, although Chapel mentions associative domains, it does not appear to support multiple associative containers at this point. Finally, STAPL differs from Chapel and other parallel languages in that it is a library.

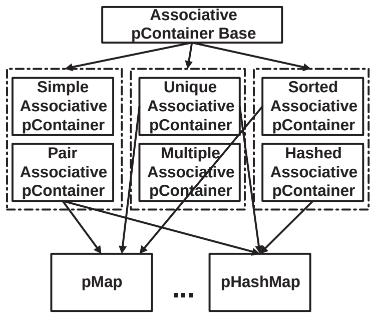
3 STAPL Overview

STAPL consists of a set of components that include `pContainers`, `pAlgorithms`, `views`, `pRanges`, and a runtime system (see Figure 1). `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. While all `pContainers` provide *sequentially equivalent interfaces* that are compatible with the corresponding STL methods, individual `pContainers` may introduce additional methods to exploit the performance offered by parallelism and by the runtime system. `pContainers` have a *data distribution* manager that provides the programmer with a *shared object view* that presents a uniform access interface regardless of the physical location of the data. Thread-safety is guaranteed by providing mechanisms that guarantee all operations leave the `pContainer` in a consistent state. Important aspects of all STAPL components are *extendability* and *composability*, e.g., the `pContainers` implemented within the framework allow users to extend and specialize them for performance, and to use `pContainers` of `pContainers`. Specialization is one avenue to improve performance in STAPL's layered architecture.

`pContainer` data can be accessed using `views` which can be seen as generalizations of STL iterators that represent sets of data elements and are not related to the data's physical location. `views` provide `iterators` to access individual `pContainer` elements . Generic parallel algorithms (`pAlgorithms`) are written



(a)



Associative pContainer Base
Simple Associative pContainer
Unique Associative pContainer
Sorted Associative pContainer
Pair Associative pContainer
Multiple Associative pContainer
Hashed Associative pContainer
pMap
...
pHashMap

(b)

Fig. 1: (a) STAPL components, and (b) associative `pContainer` hierarchy.

in terms of `views`, similar to how STL algorithms are written in terms of iterators. The `pRange` is the STAPL concept used to represent a parallel computation. Intuitively, a `pRange` is a task graph, where each task consists of a work function and a `view` representing the data on which the work function will be applied. The `pRange` provides support for specifying data dependencies between tasks that will be enforced during execution.

The runtime system (RTS) and its communication library ARMI (Adaptive Remote Method Invocation [20]) provide the interface to the underlying operating system, native communication library and hardware architecture. ARMI uses the remote method invocation (RMI) communication abstraction to hide the lower level implementations (e.g., MPI, OpenMP, etc.). A remote method invocation in STAPL can be blocking (`sync_rmi`) or non-blocking (`async_rmi`). When a `sync_rmi` is invoked, the calling thread will block until the method executes remotely and returns its results. An `async_rmi` doesn't specify a return type and the calling thread only initiates the method. The completion of the method happens some time in the future and is handled internally by the RTS. ARMI provides the `rmi_fence` mechanism to ensure the completion of all previous RMI calls. The asynchronous calls can be aggregated by the RTS in an internal buffer to minimize communication overhead. The buffer size and the aggregation factor impact the performance, and in many cases should be adjusted for the different computational phases of an application. For more details on runtime performance tuning please consult [20, 22].

4 Associative pContainers

An associative container provides optimized methods for storing and retrieving data using keys. In STAPL, similar to STL [18], 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 1(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, unique). The STAPL associative `pContainers` provide the following generic specification (data types and methods):

- Data Types:
  - `key_type`: the type of the Key
  - `value_type`: the type of the Value (not available for simple)
  - `key_compare`: the type for key comparisons (not available for hashed)
  - `view_type`: the view type
- STL compatible methods:
  - `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.
- New STAPL methods:
  - `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.

4.1 Associative `pContainer` Design and Implementation

The STAPL `pContainer` framework aims to provide a set of base concepts and a common methodology for the development of thread-safe, concurrent data structures that are extendable and composable. The major concepts in the `pContainer` framework that provide the support for the properties listed in the previous section are the *global identifier, domain, data distribution, partition, partition_mapping, pContainer component, view* and *pContainerBase*. We define the functionality of these modules in the context of associative `pContainers`, but they are general and apply to other `pContainers`.

Global Identifier (GID): In the STAPL `pContainer` framework, each element is uniquely identified by its GID. This is an important requirement that allows us to provide a shared object view. 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 `pContainer` *domain* is the universe of possible GIDs that will identify its elements. 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 order). At any instant, there is only a finite set of elements in the container. The GIDs associated with these elements are referred to as 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'`}. Domain instances are ordered sets to allow their elements to be enumerated or scanned. The enumeration order is specified by implementing two methods: `GID get_first_gid()` which returns the first GID/index of the set and `GID get_next_gid(GID)` which returns the GID that immediately follows the one provided as input to the method.

Data Distribution: The Data Distribution is responsible for determining the location where an element associated with a GID is located. A *location* is a component of a parallel machine that has a contiguous memory address space and has associated execution capabilities (e.g., threads). A location can be identified with a process address space. 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: The `partition` is a policy class used to specify how a domain is decomposed into sub-domains. The main functionality provided by a `partition` is a mapping from a GID to the *sub-domain* that contains it. Associative `pContainers` are dynamic containers supporting concurrent additions and deletions of elements, thus the corresponding partitioning strategies 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 strategy 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

```
template<class Domain>
class partition_strategy{
 partition_strategy(...);
 //compute the sub-domain
 //to which the GID is associated
 ComponentID 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);
```

(a) Partition Strategy

```
value associative_pc_base::find(key){
Location loc;
dist_manager.lookup(key)
 C_ID = part_strategy.map(key)
 loc = part_mapper.map(C_ID)
if loc is local
 return component(C_ID).find(key)
else
 return sync_rmi (loc,find(key));
```

(b) Implementation of `find()` method

Fig. 2: Interfaces for associative `pContainer` concepts

sub-domains as illustrated in Figure 2(a). 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)%num_subdomains`).

Partition Mapper: A partition is mapped onto a set of locations using a `partition-mapper`, which maps a sub-domain identifier (from 0 to $m-1$) to a location (from 0 to $L-1$). There are two partition mappers currently available in STAPL: `cyclic_mapper`, where sub-domains are distributed cyclically among locations, and `blocked_mapper`, where m/L consecutive sub-domains are mapped to a single location.

pContainer Components: The data corresponding to a sub-domain is stored in *components* within the location where that sub-domain is mapped. The GIDs associated with the stored elements of a component constitute a *sub-domain instance*. There is no data replication. We have implemented the associative `pContainer` components by extending the corresponding sequential container (typically STL containers) with functionality needed to implement domain instances.

Associative pContainer Views: `views` are defined as the accessors for the data elements stored in the `pContainer`. `pAlgorithms` in STAPL are written in terms of `views`, similar to how STL algorithms are written in terms of iterators. A `view` is defined by an ordered domain of GIDs which is a subset of the domain instance of the pContainer. For all the GIDs of the domain a view provides corresponding iterators that can be used to access the data elements. A `view` has associated a `partition` and a `partition-mapper` to allow parallel processing of the data. The default `view` provided by a `pContainer` matches the partition and the mapping of the `pContainer` data because this view provides the most efficient data access since all the elements in a sub-view are in the same physical location. The `views` 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 `views` 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, the distribution manager, and a default view. 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 1(b).

A typical implementation of a `pContainer` method is included in Figure 2(b) to illustrate how the `pContainer` modules interact. The runtime cost of the methods in the associative `pContainer` interface has three constituents: the time to decide the location and the component where the element is stored, the communication time to get/send the required information, and the time to perform the operation on a component. The time to find the location and the component depends on the partition used. For sorted associative `pContainers` (`pSet`, `pMap`, `pMultiSet`, and `pMultiMap`), an optimal search is logarithmic in the number of sub-domains of the partition, while for hashed associative `pContainers` (`pHashSet` and `pHashMap`) it is amortized constant time. The search for location and component IDs is illustrated in Figure 2(b), lines 3-5. The communication time affects only the operations that are executed remotely (Figure 2(d), line 9). For asynchronous operations, this is the time to initiate the RMI call, while for methods that return values it is the time to send the request and to receive the results. The time for performing the operation on the component is logarithmic or amortized constant time for sorted and hashed `pContainers`, respectively (Figure 2(b), line 7). 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 hashed partition the overhead is constant in each location. Different partitions, with more complex invariants, may incur different computational and memory overheads.

5 Performance Evaluation

In this section, we evaluate the scalability of the parallel methods described in Section 4, we evaluate three generic `pAlgorithms`, `pfind`, `paccumulate`, and `pcount`, and we consider a simple sorting algorithm as an example of an application based on a STAPL associative `pContainer`.

5.1 Architectures Used

We evaluated the associative `pContainer` performance on three architectures. The first system, referred to as P5-CLUSTER, is an IBM HPC cluster consisting of 122 p5-575 nodes, each node with 8 Power5 chips (1.9GHz, 64-bit PowerPC architecture) and 32GB of memory per node. The second system, referred to as P3-CLUSTER, is a 6,656 processor IBM RS/6000 SP system that consists of 416 SMP nodes, each with 16 Power3+ CPUs and where processors on each node have a shared memory pool of between 16 and 64 GB. The third system, referred to as OPTERON-CLUSTER, is a 712-CPU Opteron (2.2 GHz) cluster running the Linux operating system. Processors are organized two on a node with 6GB of memory per node. The nodes are interconnected with a high-speed InfiniBand network. We have used GNU GCC v4 on P5-CLUSTER and OPTERON-CLUSTER

```
1 evaluate_performance(N,P){
2     - generate N/P elements in a local vector local_data
3     rmi_fence();                //Barrier
4     tm = stapl::start_timer();   //start timer
5     for(it=local_data.begin(); it != local_data.end(); ++it) {
6            pmap_test.insert_async(*it); //insert N/P elements concurently
7     }
8     rmi_fence(); //ensure all insert are finished
9     elapsed = stapl::stop_timer(tm); //stop the timer
10     - Reduce elapsed times, getting the max time from all processors.
11     - Report the max time
12     //repeat lines 2-11 for the rest of the methods
```

Fig. 3: Kernel used to evaluate the performance of individual methods provided by associative containers.

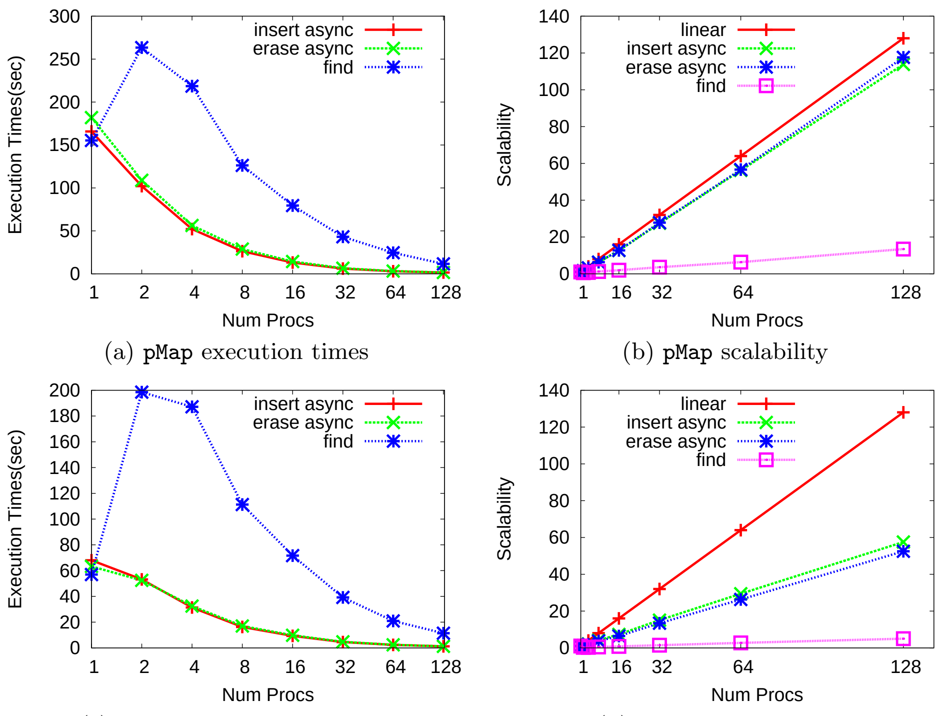
and GCC 3.4 on P3-CLUSTER, and the O3 optimization level. All systems are operated by NERSC at Lawrence Berkeley National Laboratory.

5.2 Evaluation of the Associative `pContainer` Methods

Methodology: We recall from Section 4 that a STAPL associative parallel container provides a set of methods to insert, find and erase elements. We discuss next the performance of the methods and the factors influencing the running time. To evaluate the scalability of individual methods we designed the kernel shown in Figure 3. The figure shows `insert_async`, but the same kernel is used to evaluate all methods. For a given number of elements N, all P available processors concurrently insert N/P elements. The elements are generated randomly so the resulting data distribution is approximately balanced across the machine. We report the time taken to insert all N elements globally. The measured time includes the cost of an `rmi_fence` call which is more than a simple barrier. An `rmi_fence` guarantees that all remote method calls in flight are finished when the method returns. Unless specified, all experiments have been conducted using integer keys. All associative `pContainers` were evaluated but due to the similarity of the behavior observed and space limitations, we include in this section results only for `pMap` and `pHashMap`.

Strong Scaling: In this section we analyze the scalability of the methods using the kernel described in Figure 3. We define scalability as the ratio between the time taken to complete the kernel when using one processor and the time taken when using P processors.

For the strong scaling experiment, the number of elements, N and the number of processors, are chosen differently depending on the architecture. On P5-CLUSTER we used $N = 50$ million elements and the number of processors is varied from 1 to 128. Figure 4 shows the execution times and scalability observed for the `pMap` and `pHashMap` methods; since the performance of the synchronous methods (`find`, `insert`, and `erase`) was indistinguishable from each other, only `find` is shown to simplify the figure. Although the times decrease when increasing the number of processors, the synchronous methods, `insert`, `erase`, and `find`, show poor scalability. Due to their blocking nature, these methods cannot



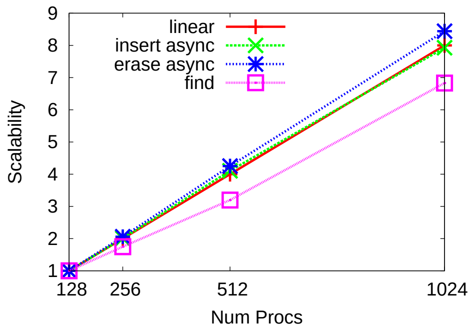
(a) `pMap` execution times (b) `pMap` scalability

(c) `pHashMap` execution times (d) `pHashMap` scalability

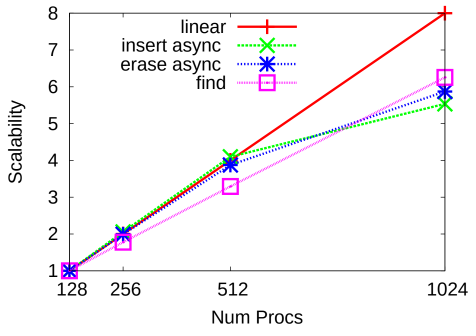
Fig. 4: P5-CLUSTER: Execution times and scalability for (a-b) `pMap` and (c-d) `pHashMap` methods with 50 million elements. Results are shown for `insert_async`, `erase_async`, and `find`; the performance of `insert` and `erase` is indistinguishable from `find`.

employ aggregation of messages or overlap communication and computation. In contrast, the asynchronous methods, `insert_async` and `erase_async`, exhibit good scalability for both `pMap` and `pHashMap`, benefiting from the aggregation and communication/computation overlap support provided by ARMI. Accessing an element in a `pMap` component requires a number of memory accesses that is logarithmic in the size of the component, while in a `pHashMap` the number of memory accesses is essentially independent of the size of the component. Hence, since the size of the components decreases as the number of processors increases, the strong scalability of the `pMap` methods should be higher than for the `pHashMap` methods. The time for synchronous methods increases from 1 to 2 processors because of the communication overhead, and then shows a steady decline. This is more evident for `pHashMap` than for `pMap` since the lower access time for the former makes the communication overhead relatively more significant.

We also evaluated the performance of the `pContainer` methods on P3-CLUSTER which allows us to study a large number of processors and larger input sizes. In Figure 5 we show results where the `pMap` and `pHashMap` methods are executed using 400 million elements and the number of processors varies from 128 to 1024; since the performance of the synchronous methods (`find`, `insert`, and



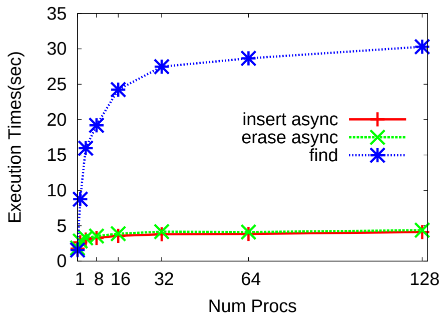
(a) pMap



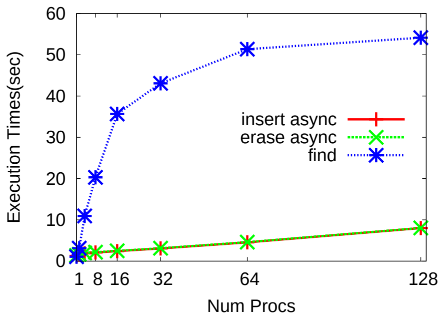
(b) pHashMap

Fig. 5: P3-CLUSTER: Scalability for (a) `pMap` and (b) `pHashMap` methods with 400 million elements. Results are shown for `insert_async`, `erase_async`, and `find`; the performance of `insert` and `erase` is indistinguishable from `find`.

`erase`) was indistinguishable from each other, only `find` is shown to simplify the figure. The scalability of the `pMap` asynchronous methods is super-linear, while the `pHashMap` scalability is sub-linear. The super-linear scalability for the `pMap` is due to the faster access time for a `pMap` component as its size decreases, i.e., as the number of processors increases. All synchronous methods show better scalability than on P5-CLUSTER (Figure 4) because the reference point is 128 processors and not 1.



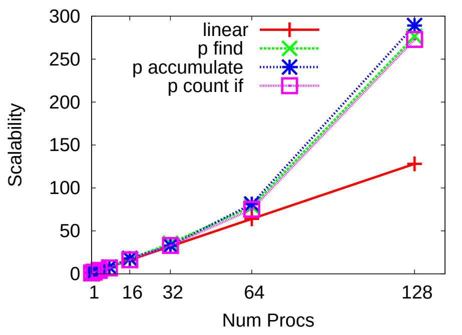
(a) P5-CLUSTER



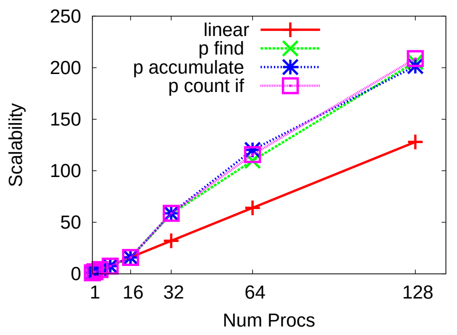
(b) OPTERON-CLUSTER

Fig. 6: Execution Times for weak scaling analysis with 1 million elements per processor. Results are shown for `insert_async`, `erase_async`, and `find`; the performance of `insert` and `erase` is indistinguishable from `find`.

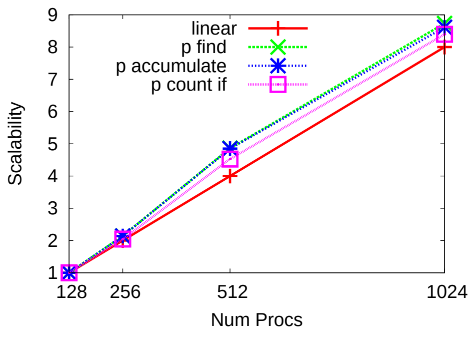
Weak Scaling: In this experiment, we modify the input arguments to the test kernel so that each processor concurrently inserts N elements, leading to a total of $N \times P$ operations. We expect the running time to show a slight increase as we increase the number of processors due to the increase in communication overhead. Results for P5-CLUSTER and OPTERON-CLUSTER are shown in Figure 6; since the performance of the synchronous methods (`find`, `insert`, and `erase`)



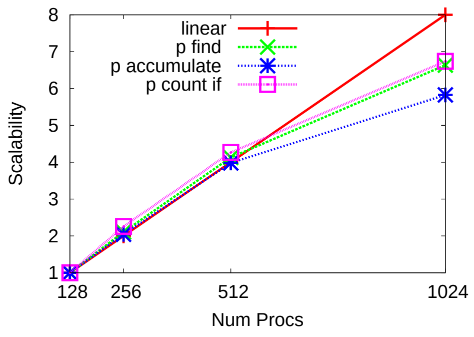
(a) `pMultiSet` (P5-CLUSTER, 50M)



(b) `pHashSet` (P5-CLUSTER, 50M)



(c) `pMultiSet` (P3-CLUSTER, 400M)



(d) `pHashSet` (P3-CLUSTER, 400M)

Fig. 7: Scalability of generic `p_find()`, `p_count()`, and `p_accumulate()` pAlgorithms on `pMultiSet` and `pHashSet` for two architectures and data sizes: (a,b) P5-CLUSTER for 50 million elements, (c,d) P3-CLUSTER for 400 million elements.

was indistinguishable from each other, only `find` is shown to simplify the figure. We notice an increase in the runtime when going from one (no remote communication) to two processors and the times for the asynchronous methods increase slightly, both as expected. On the OPTERON-CLUSTER the communication is more expensive and the overall running time increases at a faster rate. The synchronous operations do not scale well for small numbers of processors, but improve when the number of processors increases beyond 32. When using more processors, even though the calling thread may be blocked waiting for return values, requests from other threads are served thus improving the rate at which methods are executed by the system.

5.3 Support for Generic Parallel Algorithms

Generic parallel algorithms in STAPL are written in terms of `views`. Associative `pContainers` provide views that can be used to access the data and we study here the performance of generic non-mutating `pAlgorithms` such as `pfind` `pcount`, and `paccumulate` when applied to data in an associative `pContainer`. The `paccumulate` algorithm accumulates in parallel the data in each component followed by a reduction to compute the final result. The `pcount` algorithm is sim-

```
p_sort_multiset(INPUT_VIEW view) {
  pair<min, max> = p_min_max_element(view);
  associative_ps = compute_partition_strategy (min, max, P);
  stapl::p_multiset<INPUT_VIEW::data_type> pmultiset(ps);
  - insert in parallel all elements of view into pmultiset;
  - compute prefix sums and align the input view with the
  distribution of the pmultiset
  - copy in parallel from pmultiset back into the input view
  - deallocate the p_multiset
}
```

Fig. 8: Parallel sort using parallel associative containers

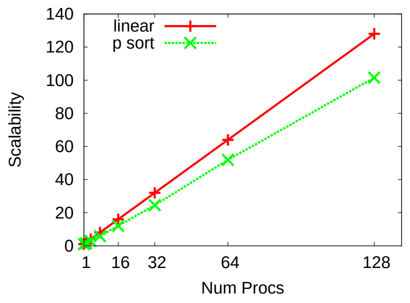
ilar but counts the number of data elements that satisfies a given predicate. The `pfind` algorithm finds the iterator corresponding to an input key. Each processor performs a linear search through all elements in its local `pContainer` components and a reduction is performed at the end to return the iterator corresponding to the first occurrence of the element. We include in Figure 7(a)(b) the scalability of the `pAlgorithms` on P5-CLUSTER using `pMultiSet` and `pHashSet` containers. In Figure 7(c)(d) we show corresponding results on the P3-CLUSTER when using a larger number of processors. The times reported for `pfind` are for the worst case scenario when the element searched for is not in the `pContainer` so the entire data space will be scanned. We observe that the algorithms exhibit super-linear speedup. The super-linear speedup is due to the the sequential (STL) containers used in the `pContainer` components. We performed the following experiment on P5-CLUSTER to verify that our super-linear speedup is justified. We measured sequential `std::acumulate` on `std::multiset` and `hash_set` containers with N=50 million elements and N=50M/128=390625 elements. The running times dropped 217 times for `std::multiset` and 134 times for `hash_set`, while the input size was only 128 times smaller.

5.4 Sorting using Associative `pContainers`

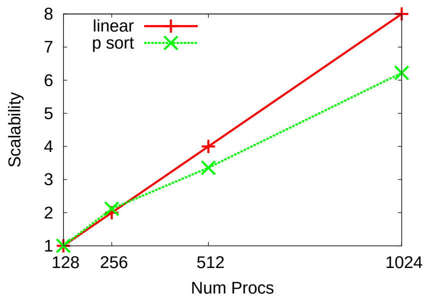
In this section, we consider a sorting algorithm based on `pMultiSet`. The algorithm inserts the elements of a view into a `pMultiSet` which stores the elements in sorted order, followed by a copy of the elements back to the original view; see pseudo-code in Figure 8. We evaluated the scalability of this `pAlgorithm` on P5-CLUSTER (N=50 million) and P3-CLUSTER (N=400 million) for various number of processors (strong scaling). The input `view` is defined over a `pArray`, another `pContainer` in STAPL[21]. In Figure 9 we see that the algorithm scales fairly well. The sub-linear scalability observed for large number of processors is due to the increased overall communication generated by the main steps of the `pAlgorithm` (e.g., insert, prefix sums and copy back).

5.5 Overhead of Associative `pContainers`

One important aspect when introducing a parallel data structure is the run-time overhead added over a corresponding sequential data structure. The run-time overhead depends on the particular container, input sizes, data types, etc. In Table 1 we compare the `pMap` methods and `pAlgorithms` on `pMultiSet` when using one processor with the the corresponding sequential container methods



(a) P5-CLUSTER



(b) P3-CLUSTER

Fig. 9: Scalability for parallel sort using parallel associative containers.

	pMap Methods				pAlgorithms			
	insert_ async	erase_ async	find	insert	pfind	paccumulate	pcount	sort assoc
Overhead(%)	7.00	6.39	10.52	8.30	1.25	3.29	3.89	12.25

Table 1: Overhead for `pMap` methods and `pAlgorithms` using `pMultiSet`.

and algorithms. For the parallel STL algorithms the overhead is relative to the sequential STL algorithms executed on the corresponding STL containers. For the parallel sort the comparison is with an equivalent sequential algorithm that is using an `std::multiset` to sort the elements. We made these measurements on the OPTERON-CLUSTER and the overheads vary between 1.25% and 12.25%. We are working on improving these overheads.

6 Conclusion

In this paper, we presented the STAPL associative `pContainers`, a collection of data structures optimized for fast storage and retrieval of data based on keys. We described the design and implementation of these `pContainers` whose methods include counterparts of the methods provided by the STL associative containers, and also some asynchronous (non-blocking) variants that can provide improved performance in parallel. Our experimental results on a variety of architectures show that STAPL associative `pContainers` provide good scalability and low overhead relative to STL containers.

References

1. 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 International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Bucharest, Romania, Jul 2001.
2. G. Blelloch. *Vector Models for Data-Parallel Computing.* MIT Press, 1990.
3. G. Blelloch. NESL: A Nested Data-Parallel Language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

4. D. Callahan, Chamberlain, B.L., and H. Zima. The cascade high productivity language. In *The Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, volume 26, pages 52–60, April 2004.
5. A. Chan and F. Dehne. CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on PC clusters, 2003.
6. D. Dechev, P. Pirkelbauer, and B. Stroustrup. Lock-free dynamically resizable arrays. In *OPODIS*, pages 142–156, 2006.
7. H. Gao, J. Groote, and W. Hesselink. Almost wait-free resizable hashtables. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, Vol., Iss., 26-30*, April 2004.
8. M. Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using dcas, 2002.
9. D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not.*, 40(10):423–437, 2005.
10. M. Herlihy. A methodology for implementing highly concurrent data structures. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 197–206, New York, NY, USA, 1990. ACM Press.
11. Intel. *Intel. Reference for Intel Threading Building Blocks, version 1.0, April 2006.*
12. E. Johnson. *Support for Parallel Generic Programming.* PhD thesis, Indiana University, 1998.
13. E. Johnson and D. Gannon. HPC++: Experiments with the parallel standard library. In *International Conference on Supercomputing*, 1997.
14. L. V. Kale and S. Krishnan. Charm++: a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, 1993.
15. H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.
16. P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.
17. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, New York, NY, USA, 2002. ACM Press.
18. D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide, Second Edition.* Addison-Wesley, 2001.
19. J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. D. Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming in C++*, chapter 14, pages 547–588. MIT Press, 1996.
20. S. Saunders and L. Rauchwerger. ARMI: An adaptive, platform independent communication library. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, San Diego, CA, June 2003.
21. G. Tanase, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pArray. In *Proceedings of the 8th MEDEA Workshop*, pages 81–88, Brasov, Romania, 2007.
22. N. Thomas, S. Saunders, T. Smith, G. Tanase, and L. Rauchwerger. ARMI: A high level communication library for STAPL. *Parallel Processing Letters*, 16(2):261–280, Jun 2006.
23. N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 277–288, 2005.