The STAPL Parallel Container Framework *

Gabriel Tanase Antal Buss Adam Fidel Harshvardhan Ioannis Papadopoulos Olga Pearce
Timmie Smith Nathan Thomas Xiabing Xu Nedal Mourad Jeremy Vu Mauro Bianco
Nancy M. Amato Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science and Engineering
Texas A&M University, College Station, TX
stapl@cse.tamu.edu

Abstract

The Standard Template Adaptive Parallel Library (STAPL) is a parallel programming infrastructure that extends C++ with support for parallelism. It includes a collection of distributed data structures called `pContainer`s that are thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. In this work, we present the *STAPL Parallel Container Framework (*PCF*)*, that 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 PCF provides a large number of basic parallel data structures (e.g., `pArray`, `pList`, `pVector`, `pMatrix`, `pGraph`, `pMap`, `pSet`). The PCF provides a class hierarchy and a composition mechanism that allows users to extend and customize the current container base for improved application expressivity and performance. We evaluate STAPL `pContainer` performance on a CRAY XT4 massively parallel system and show that `pContainer` methods, generic `pAlgorithm`s, and different applications provide good scalability on more than 16,000 processors.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming

General Terms Languages, Design, Performance

Keywords Parallel, Programming, Languages, Libraries, Data, Structures

* This research supported in part by NSF awards CRI-0551685, CCF-0833199, CCF-0830753, IIS-096053, IIS-0917266, NSF/DNDO award 2008-DN-077-ARI018-02, by the DOE NNSA under the Predictive Science Academic Alliances Program by grant DE-FC52-08NA28616, by THECB NHARP award 000512-0097-2009, by Chevron, IBM, Intel, Oracle/Sun and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Mourad is a masters student at KAUST who did an internship at the Parasol Lab. Tanase is currently a Research Staff Member at IBM T.J. Watson Research Center. Bianco is currently a scientist at the Swiss National Supercomputing Centre.

PPoPP'11, February 12–16, 2011, San Antonio, Texas, USA.

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. The Standard Template Adaptive Parallel Library (STAPL) [3] 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 [19]. 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 building blocks for writing parallel programs – a collection of parallel algorithms (`pAlgorithm`s), parallel and distributed containers (`pContainer`s), and `pViews` to abstract data accesses to `pContainer`s. `pAlgorithm`s are represented in STAPL as task graphs called `pRange`s. The STAPL runtime system includes a communication library (ARMI) and an `executor` that executes `pRange`s. Sequential libraries such as STL [19], BGL [10], and MTL [9], provide the user with a collection of data structures that simplifies the application development process. Similarly, STAPL provides the Parallel Container Framework (PCF) which includes a set of elementary `pContainer`s and tools to facilitate the customization and specialization of existing `pContainer`s and the development of new ones.

`pContainer`s are distributed, thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. A large number of parallel data structures have been proposed in the literature. They are often complex, addressing issues related to data partitioning, distribution, communication, synchronization, load balancing, 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 are developing the *STAPL Parallel Container Framework* (PCF). It consists of a collection of elementary `pContainer`s and methods to specialize or compose them into `pContainer`s of arbitrary complexity. Thus, instead of building distributed containers from scratch in an *ad-hoc* fashion, programmers can use inheritance to derive new specialized containers and composition to generate complex, hierarchical containers that naturally support hierarchical parallelism. 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 issues such as data distribution.

The STAPL PCF makes several novel contributions.

- Object oriented design: Provides a set of classes and rules for using them to build new `pContainer`s and customize existing ones.
- Composition: Supports composition of `pContainer`s that allows the recursive development of complex `pContainer`s 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`s constructed using the PCF as initial building blocks.

Some important properties of `pContainer`s 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. This is accomplished using a generic software address translation mechanism that uses the same concepts for all `pContainer`s in our framework.

Arbitrary degree and level of parallelism. For `pContainer`s 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 `pContainer`s to allow concurrent access to each level of their hierarchy to better exploit locality.

Instance-specific customization. The `pContainer`s 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 `pContainer`s that support advanced customizations so that they can easily be 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.

Previous STAPL publications present individual `pContainer`s, focusing on their specific interfaces and performance (e.g., associative containers [25], `pList` [26], `pArray` [24]) or provide a high level description of the STAPL library as a whole, of which `pContainer`s are only one component [3]. This paper presents for the first time the `pContainer` definition and composition formalism (Section 3), and the `pContainer` framework (PCF) base classes from which all `pContainer`s derive (Section 4).

2. STAPL Overview

STAPL [2, 3, 22, 27] is a framework for parallel C++ code development (Fig. 1). Its core is a library of C++ components implementing parallel algorithms (`pAlgorithm`s) and distributed data structures (`pContainer`s) that have interfaces similar to the (sequential) C++ standard library (STL) [19]. Analogous to STL algorithms that use *iterators*, STAPL `pAlgorithm`s are written in terms of `pView`s [2] so that the same algorithm can operate on multiple `pContainer`s.

`pAlgorithm`s are represented by `pRange`s. Briefly, a `pRange` is a task graph whose vertices are tasks and whose edges represent dependencies, if any, between tasks. A task includes both *work* (represented by *workfunctions*) and *data* (from `pContainer`s, generically accessed through `pView`s). The `executor`, itself a distributed shared object, is responsible for the parallel execution of computations represented by `pRange`s. Nested parallelism can be created by invoking a `pAlgorithm` from within a task.

The runtime system (RTS) and its communication library ARMI (Adaptive Remote Method Invocation) provide the interface to the underlying operating system, native communication library and hardware architecture [22]. ARMI uses the remote method invocation (RMI) communication abstraction to hide the lower level implementations (e.g., MPI, OpenMP, etc.). A remote method invocation in STAPL can be blocking (`sync_rmi`) or non-blocking (`async_rmi`). ARMI provides the fence mechanism

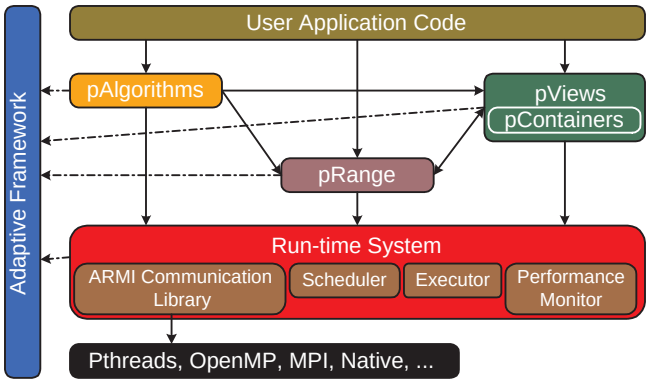


Figure 1. STAPL Overview

(`rmi_fence`) to ensure the completion of all previous RMI calls. The asynchronous calls can be aggregated by the RTS in an internal buffer to minimize communication overhead.

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

3. The STAPL Parallel Container

Data structures are essential building blocks of any generic programming library. Sequential libraries such as STL [19], BGL [10], and MTL [9], provide data structures such as arrays, vectors, lists, maps, matrices, and graphs. A parallel container is an object oriented implementation of a data structure designed to be used efficiently in a parallel environment. Design requirements of the STAPL `pContainer` are listed below.

Scalable performance. `pContainer`s must provide scalable performance on both shared and 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 respect a memory consistency model. Currently, we support a relaxed consistency model. Due to space constraints, these issues are not addressed further in this paper; see [23] for details.

Shared object view. Each `pContainer` instance is globally addressable, i.e., it provides a shared memory address space. This supports ease of programming, allowing programmers to ignore the distributed aspects of the container if they so desire.

Composition. The capability to compose `pContainer`s (i.e., `pContainer`s of `pContainer`s) provides a natural way to express and exploit nested parallelism while preserving locality. This feature is not supported by other general purpose parallel libraries.

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.

3.1 pContainer definition

A STAPL *pContainer* is a distributed data structure that holds a finite *collection of typed 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 $\mathcal{D}$ to the elements in $\mathcal{C}$. 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.

Thus, a `pContainer` is defined as:

$$pC \stackrel{def}{=} (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}, \mathcal{S}) \quad (1)$$

The tuple $(\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O})$ is known as the *native pView* of the `pContainer`. As described in [2], 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 processor). 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 `pView`s.

A `pContainer` stores its elements in a non-replicated fashion in a distributed collection of *base containers* (`bContainer`s), each having a corresponding native `pView`. `pContainer`s can be constructed from any base container, sequential or parallel, so long as it can support the required interface. The `pContainer`s 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 [9] for matrices), containers available in libraries developed for multicore (e.g., TBB [14] concurrent containers), or other `pContainer`s. 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. As described in more detail in Section 4.3, 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 perform the desired operation.

3.2 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 `pContainer`s be a `pContainer`, i.e., that `pContainer`s are closed under composition.

An important feature of composed `pContainer`s 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. Consider the example of a `pArray` of `pArray`s. This can be declared in STAPL using the following syntax:

```
p_array<p_array<int>> pApA(10);
```

Such a composed data structure can distribute both the top level and the nested `pArray`s in various ways across the machine. Accessing the elements of the nested `pArray`s is done naturally using a concatenation of the methods of the outer and inner `pArray`s. For example, `pApA.get_element(i).get_element(j)` returns a reference to the j-th element belonging to the i-th nested `pArray`. Moreover, data stored in a composed data structure can be efficiently exploited by nested `pAlgorithm`s. In the `pApA` example, computing the minimum element of each of the nested `pArray`s can be done using a parallel `forall` on the outer `pArray`, and within each nested one, a reduction to compute the minimum value.

In the remainder of this section we formalize the `pContainer` composition definition. The *height* of a `pContainer` is the depth of the composition, i.e., the number of nested `pContainer`s. 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 `pContainer`s of height H_1 and H_2, respectively. The composed `pContainer` $pC = pC_1 \circ pC_2$ is of height $H_1 + H_2$. In pC, each element of $pC_1[i]$, $i \in \mathcal{D}_1$, 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

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

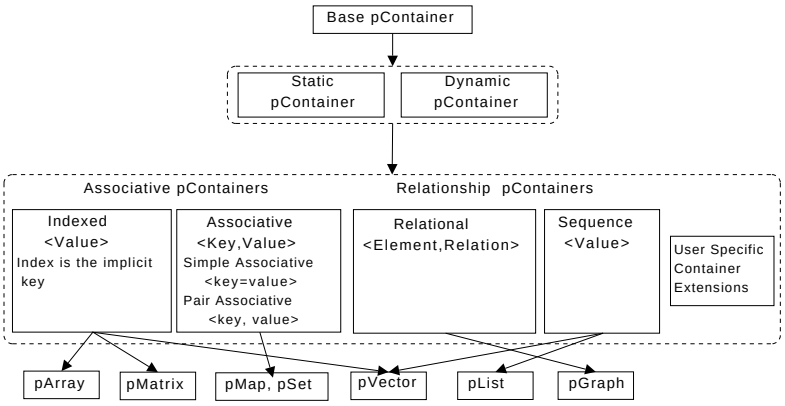
where $\mathcal{F}(x, y) = (\mathcal{F}_1, \mathcal{F}_2)(x, y) = (\mathcal{F}_1(x), \mathcal{F}_2(y))$, $(x, y) \in \mathcal{D}$. The components $\mathcal{C}$ and $\mathcal{S}$ are isomorphic to $\mathcal{D}$ and defined similarly to it. With this formalism, arbitrarily deep hierarchies can be defined by recursively composing `pContainer`s.

Given a composed `pContainer` $PC = (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}, \mathcal{S})$, of height H, the GID of an element at level $h \leq H$ is a tuple $(x_1, \ldots, x_h)$. Similarly, the mapping function to access a `pContainer` at level h is a subsequence (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 $\mathcal{O}_h$. To invoke a method at level h, the appropriate element of the GID tuple has to be passed to each method invoked in the hierarchy, as shown in the example given at the beginning of this section.

`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 `pContainer`s are composed, then the distribution information of the initial `pContainer`s will be preserved in the new `pContainer`. A feature of our composition operation is that it allows for (static) specialization if machine mapping 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.

4. The Parallel Container Framework (PCF)

An objective of the STAPL *Parallel Container Framework (PCF)* is to simplify the process of developing generic parallel containers. It is a collection of classes that can be used to construct new `pContainer`s through inheritance and specialization that are customized for the programmer's needs while preserving the properties of the base container. In particular, the PCF can generate a wrapper for any standard data structure, sequential or parallel, that has the meta information necessary to use the data structure in a distributed, concurrent environment. 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



(a)

```
   //bcontainer definition:
   //sequential graph using vector storage
1: typedef pg_base_container<vector,..> bpg_s;
   //and a static partition
2: typedef pg_static<bpg_s,..> partition_s;

   //sequential graph using std::map storage
3: typedef pg_base_container<map,..> bpg_d;
   //and a dynamic partition
4: typedef pg_fwd<bpg_d,..> partition_d;

   //pgraph with static partition
5: p_graph<DIRECTED,MULTI,partition_s> pg_s(N);

   //pgraph with dynamic storage and
   //method forwarding
6: p_graph<DIRECTED,MULTI,partition_d> pg_d(N);
```

(b)

Figure 2. (a) PCF design. (b) `pContainer` customization.

enabling the use of parallel or sequential containers from other libraries, e.g., MTL [9], BGL [10] or TBB [14].

STAPL provides a library of `pContainer`s constructed using the PCF. These include counterparts of STL containers (e.g., `pVector`, `pList` [26], and associative containers [25] such as `pSet`, `pMap`, `pHashMap`, `pMultiSet`, `pMultiMap`) and additional containers such as `pArray` [24], `pMatrix`, 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.

4.1 pContainer Framework design

The PCF is designed to allow users to easily build `pContainer`s 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 `pContainer`s. Figure 2(a) shows the main concepts and the derivation relations between them; also shown are the STAPL `pContainer`s that are defined using those concepts. All STAPL `pContainer`s are derived from the `pContainer` base class which is in charge of storing the data and distribution information. The remaining classes in the PCF provide minimal interfaces and specify different requirements about `bContainer`s. First, the `static` and `dynamic` `pContainer`s are classes that indicate if elements can be added to or removed from the `pContainer`. The next discrimination is between *associative* and *relational* `pContainer`s. 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 `pContainer`s. In other cases, such as a hashmap, keys must be stored explicitly. The PCF provides an `associative base pContainer` for such cases. The `relational pContainer`s 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 default implementations that can be customized for each `pContainer` instance using template arguments called *traits*. This allows users to specialize various aspects, e.g., the `bContainer` or the data distribution, to improve the performance of their data structures. In Figure 2(b) 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 2(b), line 5, shows the declaration of a directed `pGraph` allowing multiple edges between the same source and target vertices and using a static partition. With a static partition, users need to declare the size of the `pGraph` at construction time and subsequent invocations of the `add_vertex` method will trigger an assertion. Figure 2(b), 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 Section 5.2.

4.2 `pContainer` Interfaces

For each concept in the PCF (Figure 2(a)) there is a corresponding interface consisting of various constructors and methods as described in [23]. `pContainer` methods can be grouped in three categories: synchronous, asynchronous and split phase. Synchronous methods have a return type and guarantee that the method is executed and the result available when they return. Asynchronous methods have no return value and return immediately to the calling thread. Split phase execution is similar to that in Charm++ [17]. The return type of a split phase method is a future that allocates space for the result. The invocation returns immediately to the user and the result can be retrieved by invoking the `get` method on the future which will return immediately if the result is available or block until the result arrives. Performance trade-offs between these three categories of methods are discussed in Section 5.5.

4.3 Shared Object View implementation

Recall that the elements of a `pContainer` are stored in non-replicated fashion in a distributed collection of `bContainer`s. 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. Its performance is studied in Section 5.2.

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 `GID`s are indices for `pArray`s, keys for `pMap`s, and vertex identifiers for `pGraph`s.

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 review the PCF components involved in this address translation. As defined in Section 3.1, the set of `GID`s of a

pContainer is called a *domain* $\mathcal{D}$. For example, the domain of a pArray is a finite set of indices while it is a set of keys for an associative pContainer. A pContainer's domain is partitioned into a set of non-intersecting *sub-domains* by a partition class, itself a distributed object that 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 and in general, there can be multiple bContainers allocated in a *location*. Finally, a class called a partition-mapper maps a sub-domain (and its corresponding bContainer) to the location where it resides, and a location-manager manages the bContainers of a pContainer mapped to a given location.

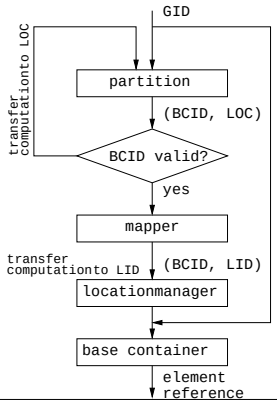


Figure 3. pContainer modules for performing address resolution to find the element reference corresponding to a given GID.

Figure 3 describes the address resolution procedure. Given the unique GID identifying a pContainer element, the partition class is queried about the sub-domain/bContainer associated with the requested GID. If the bContainer information (specified by a bContainer identifier, or BCID) is not available, the partition provides information about the location (LOC) where the bContainer information might be retrieved, and the process is restarted on that location. If the BCID is available and valid, then the partition-mapper returns information about the location where the bContainer resides (LID).

For 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 is possible to optimize the address translation mechanism by employing a static partition that computes the mapping from a GID to a bContainer and a location in constant time by evaluating an expression or doing a table lookup. For example, a pArray with N elements can map the index i to the location $i\%P$, where P is the number of available locations. Or, for a pGraph where the number of vertices is fixed at construction, we can use a static partition that computes a vertex location in constant time using a hash table.

Dynamic pContainers, where elements can be added or deleted, need to employ dynamic partitions. Currently, we provide a dynamic partition implemented as a distributed directory. The directory statically associates a home for an individual GID that always knows the location where the respective GID is stored. If the element happens to migrate to a new location the home needs to be updated with information about the new location. With this directory implementation, accessing an element corresponding to a given GID involves two steps. First, find and query the home where the element lives and second, access the element on its location.

4.4 Method Forwarding

For a pContainer to operate on a non-local element, it must determine the element's location and invoke the method at that location. Hence, if the element's address cannot be determined locally, then the address resolution process may add significantly to the critical path of the method's execution. To alleviate this problem, we combine the address resolution and the method execution into one operation. This mechanism, referred to as *method forwarding*, allows the method to be forwarded along with the address resolution process instead of first fetching the address from a remote location and then instantiating the remote method invocation.

As will be shown in Section 5.2, a partition using forwarding provides improved performance over a directory that determines the GID's location using synchronous communication.

5. pContainer Performance Evaluation

In this section, we evaluate the performance of representative pContainers developed using the PCF: pArray, pList, pMatrix, pGraph, pHashMap and composed pContainers. In Section 5.1, we study the scalability of parallel methods for pArray, pGraph and pList. We examine trade-offs for various address resolution mechanisms in Section 5.2 and for pContainer composition in Section 5.3. Sections 5.4, 5.5, and 5.6 analyze generic parallel algorithms, graph algorithms, and a MapReduce application, respectively.

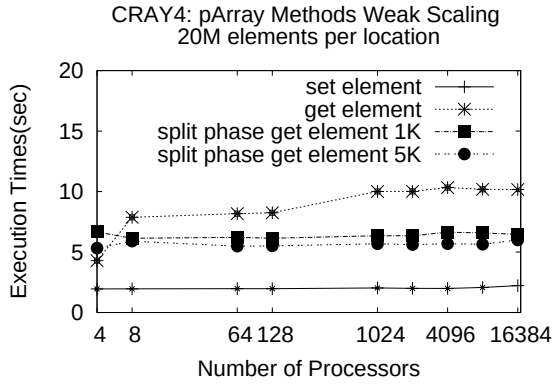
With the exception of MapReduce, we conducted all our experimental studies on a 38,288 core Cray XT4 (CRAY4) available at NERSC. There are 9,572 compute nodes each with a quad core Opteron running at 2.3 GHz and a total of 8 GB of memory (2 GB of memory per core). The MapReduce study was performed on a Cray XT5 (CRAY5) with 664 compute nodes, each containing two 2.4 GHz AMD Opteron quad-core processors (5,312 total cores). In all experiments, a location contains a single processor core, and the terms can be used interchangeably.

All the experiments in this paper, with the exception of MapReduce, show standard weak scaling where the work per processor is kept constant. As we increase the number of cores the amount of work increases proportionally and the baseline for each experiment is the time on four cores which is the number of cores in a compute node on CRAY4. Confidence intervals are included in the plots. The machines used are very stable though and the variations for each experiment are small, so the confidence intervals are often not visible in the plots.

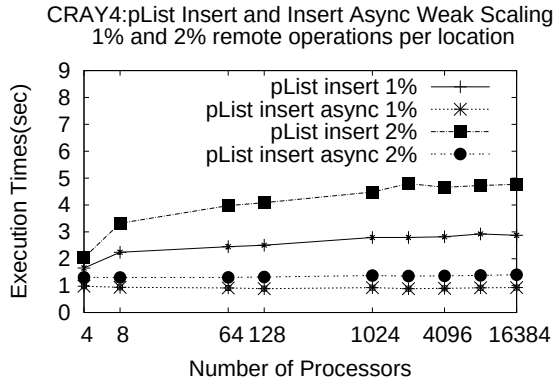
5.1 pContainer Methods

The performance of various STAPL pContainers has been studied in [4, 24–26]. In this section, we examine the performance of novel pContainer methods in the context of the pArray, pList and pGraph data structures.

To evaluate the scalability of the pContainer methods we designed a simple kernel in which all P available cores (locations) concurrently perform N/P invocations, for a given number of elements N. We report the time taken to perform all N operations globally. The measured time includes the cost of a fence to ensure the methods are globally finished. In Figure 4(a) we show the performance for pArray set_element, get_element and split_get_element. We performed a weak scaling study with 20M elements and 20M method invocations per location. In this experiment there are 1% remote accesses. We observe good scalability for the asynchronous invocations with only 5.8% increase in execution time as we scale the number of cores from 4 to 16384. For the synchronous methods, the execution time increases 237% relative to 4 cores and 29% relative to 8 cores. The big increase in execution time from 4 to 8 is due to the inter-node communication which negatively affects performance, especially for synchronous communication. For the split_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 execu-

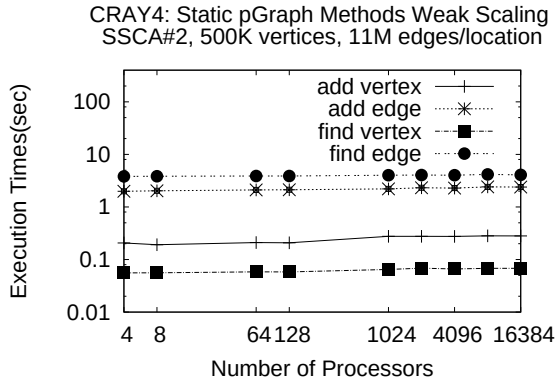


(a) `pArray`

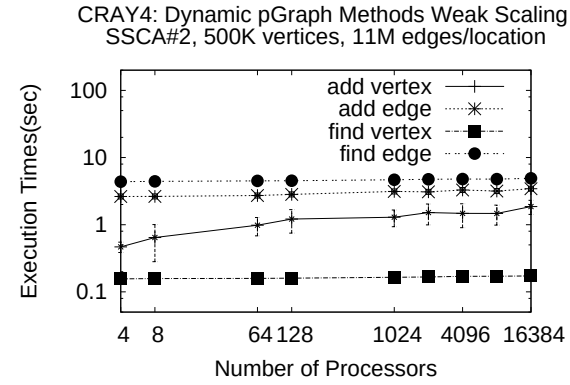


(b) `pList`

Figure 4. CRAY4: (a) `pArray` methods `set_element, get_element` and split phase `get_element`. 20M method invocations per location with 1% remote (b) 5M `pList` method invocations with 1% and 2% remote. The number of remote accesses is a variable that we can control as part of our experimental setup.



(a) Static `pGraph`



(b) Dynamic `pGraph`

Figure 5. Evaluation of static and dynamic `pGraph` methods while using the SSCA2 graph generator; 500k vertices, 11.5M edges, $\sim$40 remote edges per location; $\sim$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.

tion enables the aggregation of the requests by the runtime system as well as allowing communication and computation overlap. For the `split_get_element` the overall execution time increases 4.5% as we scale the number of cores from 4 to 16384, when 5000 invocations are started before waiting the result.

In Figure 4(b), we show a weak scaling experiment on a `pList` using 5 million elements per core and up to 16384 cores (81.9 billion total method invocations performed). The synchronous `insert` adds an element at a given position and returns a reference to the newly inserted element. The `insert_async` inserts the element asynchronously and has no return value. In this experiment, the majority of the invocations are executed locally with 1% and 2%, respectively, being executed remotely. We observe good scalability of the two methods up to 16384 cores. The asynchronous versions of the `pContainer` methods are faster as they can overlap communication with computation and don't return information about the position where the element was added.

The `pGraph` is a relational data structure consisting of a collection of vertices and edges. The `pGraph` is represented as an adjacency list and depending on its properties, different `bContainer`s 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 an expression and has a `bContainer` that uses a `std::vector` to store the vertices, each of which uses a `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 shown in Figure 2(a), the static or dynamic behavior is achieved by passing the corresponding template arguments to the `pGraph` class.

We performed a weak scaling experiment on CRAY4 using a 2D torus where each core holds a stencil of 1500$\times$1500 vertices and corresponding edges, and a random graph as specified in the SSCA2 benchmark [1]. SSCA2 generates a set of clusters where each cluster is densely connected and the inter cluster edges are sparse. We use the following parameters for SSCA2: cluster size = $(V/P)^{1/4}$, where V is the number of vertices of the graph, 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 5 shows the execution time for `add_vertex`, `add_edge`, `find_vertex` and `find_edges` for

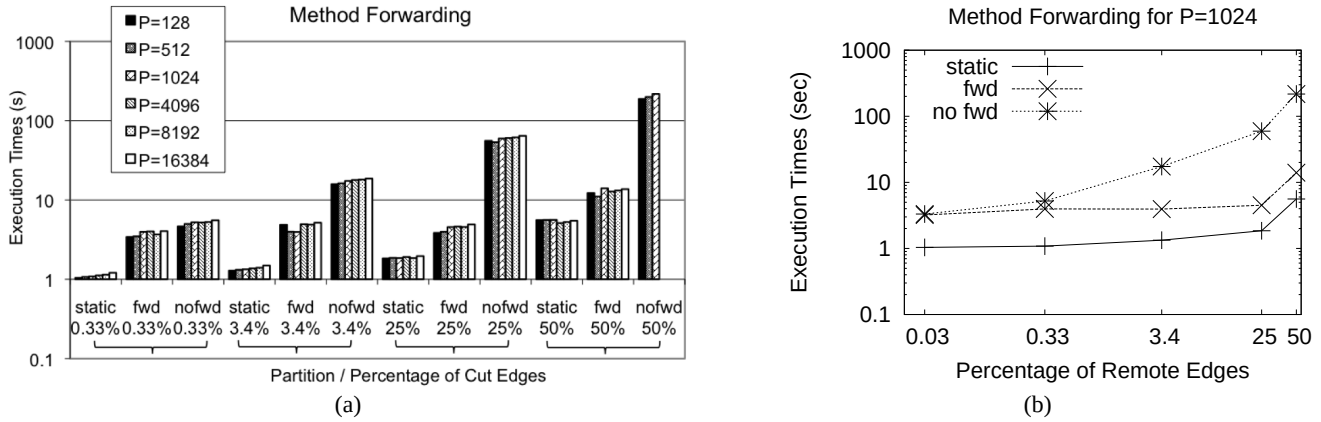


Figure 6. Find sources in a directed `pGraph` using static, dynamic with forwarding and dynamic with no forwarding partitions. Execution times for graphs with various percentages of remote edges for (a) various core counts and for (b) 1024 cores.

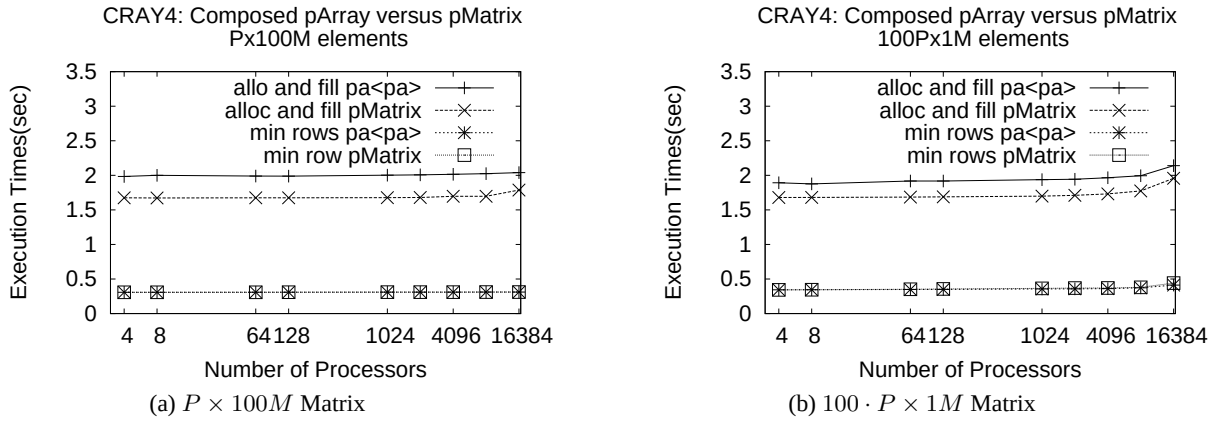


Figure 7. Comparison of `parray<parray<>>` ($pa < pa >$) and `pMatrix` on computing the minimum value for each row of a matrix. Weak scaling experiment with (a) $P \times 100M$ and (b) $100 \cdot P \times 1M$ elements. `parray<parray<>>` takes longer to initialize while the algorithm executions are very similar.

the SSCA2 input. For the dynamic `pGraph` the container is initially empty and vertices are added using `add_vertex`. As seen in the plots, the methods scale well up to 16384 cores. The addition of edges is a fully asynchronous parallel operation. Adding vertices in the dynamic `pGraph` causes additional asynchronous communication to update the directory information about where vertices are 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 cores. The execution time increases 2.96 times for the `add_vertex` in the dynamic `pGraph` as we scale from 4 to 16384 cores. The mesh results are not included due to space limitations but they exhibit similar trends as the SSCA2 results.

5.2 Evaluation of address translation mechanisms

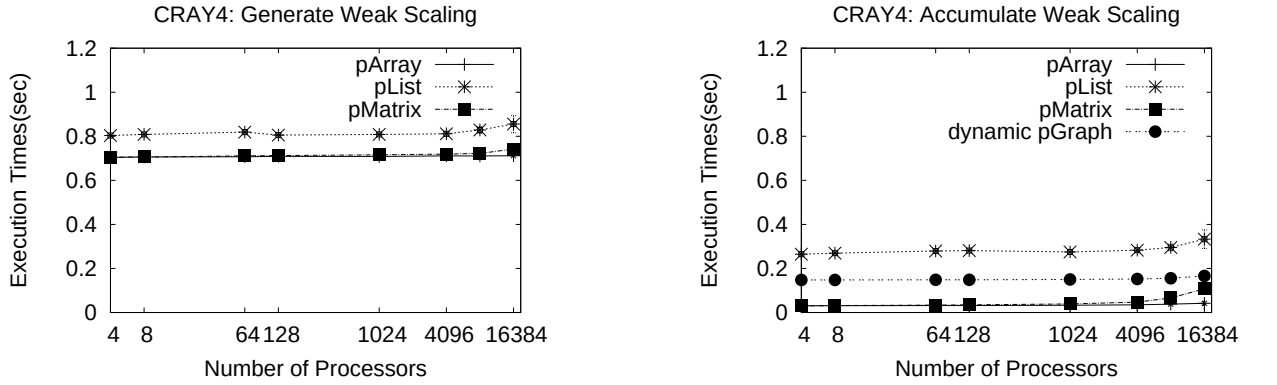
In this section, we evaluate the performance of the three types of address translation mechanisms introduced in Section 4.3: a static partition mapping `GID`s to `bContainer`s, and distributed dynamic partitions with and without method forwarding.

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 `bContainer`s. We considered four graphs, all 2D tori, which vary according to the percentage of remote edges: .33%, 3.4%, 25% and 50%. This was achieved by having each core hold a stencil of 150×15,000, 15×150,000, 2×1,125,000 and 1×2,250,000, respectively.

Figure 6(a) provides a summary of the execution times for the different percentages of remote edges and different numbers of cores, where scalability can be appreciated together with the increasing benefit of forwarding as the percentage of remote edges increases. In Figure 6(b) we include results for the three approaches on all four types of graphs for 1024 cores. 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.

5.3 `pContainer` composition evaluation

So far we have made a case for the necessity of `pContainer` composition to increase programmer productivity. Instead of di-



(a) `stapl::generate` (b) `stapl::accumulate`

Figure 8. Execution times for `stapl::generate` and `stapl::accumulate` algorithms on CRAY4. Same algorithm applied to different data structures. The `pArray`, `pList` and `pMatrix` are with 20M elements per location. The input `pGraph` has a 1500x1500 mesh per location.

rectly building a complex `pContainer`, the programmer can compose one from the basic `pContainer`s available in the PCF and shorten development and debugging time. The issue we study here is the impact of composition on performance.

For this comparative performance evaluation we compute the minimum element in each row of a matrix using both the `pMatrix pContainer` (which is available in the PCF library) and the composed `pArray` of `pArray`s. The algorithm code is the same for the two cases, due to the access abstraction mechanism provided by STAPL. It calls a parallel `forall` across the rows, and within each row, a reduction to compute the minimum value. We also measure the time to create and initialize the storage. The `pMatrix` allocates the entire structure in a single step, while the `pArray` of `pArray`s allocates the outer structure first and then allocates the single `pArray` elements individually. In Figure 7 we include, for CRAY4, the execution times for allocating and initializing the two data structures and the times to run the min-of-each-row algorithm, in a weak scaling experiment. Figure 7(a) shows the case of a $P \times 100M$ element matrix (P is the number of cores), while Figure 7(b) shows the case of a $100 \cdot P \times 1M$ element matrix. The aggregated input sizes are overall the same.

As expected, the `pArray` of `pArray`s initialization time is higher than that for a `pMatrix`. The time for executing the algorithm, however, is very similar for the two data structures and scales well to 16384 cores. While we cannot state with certainty that our PCF allows for efficient composition (negligible additional overhead) for any pair of `pContainer`s, the obtained results are promising.

5.4 Generic pAlgorithms

Generic `pAlgorithm`s can operate transparently on different data structures. We use `pViews` to abstract the data access and an algorithm can operate on any `pContainer` provided the corresponding `pViews` are available. We use the `stapl::generate` algorithm to produce random values and assign them to the elements in the container. It is a fully parallel operation with no communication for accessing the data. `stapl::accumulate` adds the values of all elements using a cross location reduction that incurs communication on the order of $O(\log P)$.

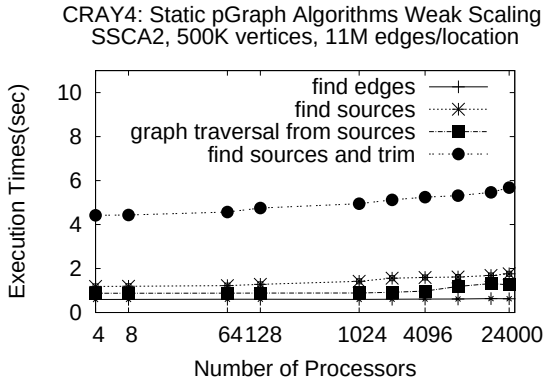
In Figure 8, we show the execution times for the `pAlgorithm`s on `pArray`, `pList`, `pGraph`, and `pMatrix`. We performed a weak scaling experiment using 20M elements per location for `pArray`, `pList` and `pMatrix` and a torus with a 1500×1500 stencil per location for `pGraph`. The `pArray` and `pMatrix` are efficient static containers for accessing data based on indices and linear traversals [4, 24]. For `pMatrix` the algorithms are applied to a linearization of the matrix. The `pList` is a dynamic `pContainer` optimized for fast insert and delete operations at the cost of a slower access time relative to static data structures such as `pArray`. `pGraph` is a relational data structure consisting of a collection of vertices and edges. Generic STL algorithms are used with `pGraph` to initialize the data in a `pGraph` or to retrieve values from vertices or edges. The `stapl::accumulate` adds the values of all the vertex properties.

The algorithms show good scalability as we scale the number of cores from 4 to 16384. There is less than 5% increase in execution time for `pArray stapl::generate` and about 33% for `stapl::accumulate` due to increased communication performed in the reduction ($O(\log P)$). All three algorithms on a `pList` with 20M elements per location provide good scaling. There is less than 6% increase in execution time for `stapl::generate` as we scale from 4 to 16384 cores and 26% for `stapl::accumulate`. The `pList` is generally slower than the other containers especially when the traversal cost dominates the computation performed. The `pMatrix` [4] considered was of size $P \times 20M$ where P is the number of locations, leading to a weak scaling experiment where each locations owns a row of size 20M. Similar to the `pArray`, there is less than 5% increase in execution time for `stapl::generate`, and less than 25% for `stapl::accumulate`.

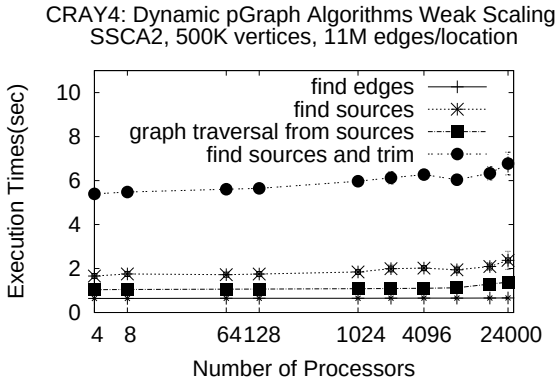
This kind of analysis is useful to help users understand the performance benefits of various data structures. From all three plots we observe that the access time for a `pList` is higher than the access time for static `pContainer`s, and this is due to the different behavior of the STL containers used as `bContainer`s. The difference in performance is less for `stapl::generate` because it involves heavier computation (the random number generator).

5.5 `pGraph` algorithms

In this section, we analyze the performance of several `pGraph` algorithms for various input types and `pGraph` characteristics. `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 depth-first search 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 com-

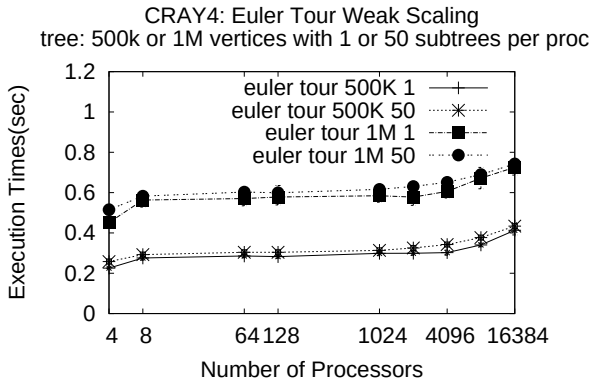


(a)

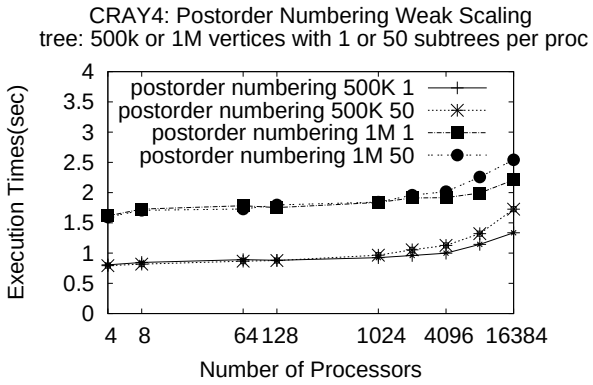


(b)

Figure 9. Execution times for different `pGraph` algorithms on on CRAY4. Static versus dynamic `pGraph` comparison. The input is generated using the SSCA2 scalable generator with 500K vertices per core



(a) Tree Graph



(b) Postorder Numbering

Figure 10. Execution times for Euler Tour and its post-order numbering applications using a tree made by a single binary tree with 500k or 1M subtrees per core.

pute all vertices and edges accessible from a set of starting points. `trim` is another useful computation when computing cycles or strongly connected components. 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 ran the algorithms on various input types including a torus, a sparse mesh and SSCA2 random graphs. In Figure 9, weak scaling results are shown for SSCA2 for both static and dynamic `pGraph`s. The number of cores is 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. `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 spawns 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 `pGraph`s. Figure 6 illustrates that the execution time of the `pGraph` algorithms increases with the number of remote edges.

The Euler Tour (ET) is an important representation of a graph for parallel processing. Since the ET represents a depth-first-search traversal, when it is applied to a tree it can be used to compute a number of tree functions such as rooting a tree, postorder numbering, vertex levels, and number of descendants [15]. The parallel ET algorithm [15] tested here uses a `pGraph` to represent the tree and a `pList` to store the final Euler Tour. In parallel, the algorithm executes traversals on the `pGraph` `pView` and generates Euler Tour segments that are stored in a temporary `pList`. Then, the segments are linked together to form the final `pList` containing the Euler Tour. The tree ET applications are computed using a generic algorithm which first initializes each edge in the tour with a corresponding weight, and then performs the partial sum algorithm. The partial sum result for each edge is copied back to the graph, and the final step computes the desired result.

Performance is evaluated by 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 the number of subtrees for each location (for each subtree root, one to its root and two to its children in each location, with directed edges in both directions). Figure 10(a) and 10(b) show the execution time on CRAY4 for different sizes of the tree and

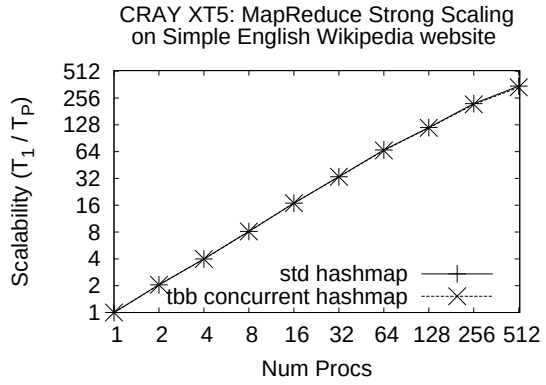


Figure 11. MapReduce used to count the number of occurrences of every word in Simple English Wikipedia website (1.5GB).

varying numbers of subtrees. The running time increases with the number of vertices per location because the number of edges in the computed ET increases correspondingly. When there are more subtrees specified in each location, there is more communication required to link them. Figure 10(b) shows the execution time for computing the postorder numbering. The running time increases with the number of vertices per location because the number of edges increases which are proportional to the computation. When more subtrees are specified in a location, more segments are formed in the `pList` and more communication is needed for the partial sum.

5.6 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`, a dynamic associative `pContainer`[25]. The application splits the input data across the available cores 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 function 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 multiplicity of each word in a 1.5GB text input of the Simple-English Wikipedia website (simple.wikipedia.org). Because the input size was fixed, we include a strong scaling study where we measure the time taken to compute the multiplicity for all input words on CRAY5. In Figure 11 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 cores without noticeable differences for different storages. The slowdown on 256 and 512 cores is due to the small computation performed per core relative to the communication required to insert the data into the `pHashMap`.

6. Related Work

There is a large body of work in the area of parallel data structures with projects aiming at shared memory architectures, distributed memory architectures or both. Parallel programming languages [5–7, 29] typically provide built in arrays and provide minimal guidance to the user on how to develop their own specific parallel data structures. STAPL `pContainer`s are generic data structures and this characteristic is shared by a number of existing projects such as PSTL [16], TBB [14], and POOMA [21]. The Parallel Standard Template Library (PSTL) provides vector, list, queue and associative containers as self contained implementations and without emphasizing a common design. Intel Threading Building Blocks (TBB) provides generic data structures such as vectors, queues and hash maps adapted for shared memory systems. STAPL is distinguished from TBB in that it targets both shared and distributed systems and is explicitly designed for *extendibility*, providing the user with the means of developing new distributed data structures. A large number of projects provide individual parallel data structures such as Parallel Boost Graph Library [10], and Hierarchically Tiled Arrays [11] and Multiphase Specifically Shared Array in Charm++[17]. The STAPL PCF differs from them by providing a uniform design for all data structures provided.

There has been significant research in the area of concurrent data structures for shared memory architectures. Most of the related work [8, 12, 13, 18, 20, 28] is focused either on how to implement concurrent objects using different locking primitives or how to implement concurrent lock-free data structures. In contrast, the STAPL `pContainer`s are designed to be used in both shared and distributed memory environments and address the additional complexity required to manage the data distribution. Ideas from these papers can be integrated in our framework at the level of `bContainer`s for efficient concurrent access on one location.

The STAPL PCF differs from other languages and libraries by focusing on developing a generic infrastructure that will efficiently provide a shared memory abstraction for `pContainer`s. The framework automates, in a very configurable way, aspects relating to data distribution and thread safety. We emphasize interoperability with other languages and libraries [4], 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.

7. Conclusion

In this paper, 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 `pContainer`s 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 in cooperation with other libraries, and (c) support for the (recursive) composition of `pContainer`s into nested, hierarchical `pContainer`s that can support arbitrary degrees of nested parallelism. Furthermore, we have developed a library of basic `pContainer`s constructed using the PCF as initial building blocks and demonstrated the scalability of its components on very large computer systems. We have shown how we have implemented a *shared object view* of the `pContainer`s 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 `pContainer`s 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 that 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 show that `pContainer`s provide good scalability for both static and dynamic `pContainer`s.

References

[1] D. 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*, Springer, 2005.

[2] 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.

[3] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato and L. Rauchwerger "STAPL: Standard template adaptive parallel library," In *Proc. of the 3rd Annual Haifa Experimental Systems Conf. (SYSTOR)*, pp. 1–10, 2010.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] J Guo, G. Bikshandi, B. B. Fraguela and D. Padua. Writing productive stencil codes with overlapped tiling. *Concurr. Comput. : Pract. Exper.*, 21(1):25–39, 2009.

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

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

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

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

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] G. Tanase, "The STAPL Parallel Container Framework". PhD thesis, Texas A&M University, College Station, 2010.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.