The STAPL pList*

Gabriel Tanase, Xiabing Xu, Antal Buss, Harshvardhan, Ioannis Papadopoulos,
Olga Pearce, Timmie Smith, Nathan Thomas, Mauro Bianco,
Nancy M. Amato and Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science and Engineering, Texas A&M University
{gabrielt,xiabing,abuss,ananvay,ipapadop,olga,timmie,nthomas,bmm,amato,rwerger}
@cse.tamu.edu

Abstract. We present the design and implementation of the STAPL `pList`, a parallel container that has the properties of a sequential list, but allows for scalable concurrent access when used in a parallel program. The Standard Template Adaptive Parallel Library (STAPL) is a parallel programming library that extends C++ with support for parallelism. STAPL provides a collection of distributed data structures (`pContainers`) and parallel algorithms (`pAlgorithms`) and a generic methodology for extending them to provide customized functionality. STAPL `pContainers` are thread-safe, concurrent objects, providing appropriate interfaces (e.g., `views`) that can be used by generic `pAlgorithms`. The `pList` provides STL equivalent methods, such as insert, erase, and splice, additional methods such as split, and efficient asynchronous (non-blocking) variants of some methods for improved parallel performance. We evaluate the performance of the STAPL `pList` on an IBM Power 5 cluster and on a CRAY XT4 massively parallel processing system. Although lists are generally not considered good data structures for parallel processing, we show that `pList` methods and `pAlgorithms` (`p_generate` and `p_partial_sum`) operating on `pList`s provide good scalability on more than 10^3 processors and that `pList` compares favorably with other dynamic data structures such as the `pVector`.

1 Introduction and Motivation

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

* This research supported in part by NSF Grants EIA-0103742, ACR-0081510, ACR-0113971, CCR-0113974, ACI-0326350, CRI-0551685, CCF-0833199, CCF-0830753, by the DOE NNSA under the Predictive Science Academic Alliances Program by grant DE-FC52-08NA28616, by Chevron, IBM, Intel, HP, and by King Abdullah University of Science and Technology (KAUST) Award KUS-C1-016-04. 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.

Library (STAPL) [1, 20, 23]. STAPL is a parallel C++ library with functionality similar to STL, the C++ Standard Template Library. STL is a collection of basic algorithms, containers and iterators that can be used as high-level building blocks for sequential applications. STAPL provides a collection of parallel algorithms (`pAlgorithms`), parallel and distributed containers (`pContainers`), and `views` to abstract the data access in `pContainers`. These are the building blocks for writing parallel programs using STAPL. An important goal of STAPL is to provide a high productivity development environment for applications that can execute efficiently on a wide spectrum of parallel and distributed systems.

`pContainers` are collections of elements that are distributed across a parallel machine and support concurrent access. STAPL provides a unified approach for developing `pContainers`. It uses object-oriented technology to build distributed thread-safe containers that can easily be extended and customized. This approach allows us to provide a large variety of static and dynamic `pContainers` such as `pArray` [20], `pMatrix`[4], associative containers [21], `pVector` and `pGraph`.

Contribution. In this paper, we present the design and implementation the STAPL `pList`, a parallel container that has the properties of a sequential list, but allows for scalable concurrent access when used in a parallel program. In particular, the `pList` is a distributed doubly linked list data structure that is the STAPL parallel equivalent of the STL list container. The `pList` provides thread safe, concurrent methods for efficient insertion and removal of elements from a collection, as well as splicing and splitting. The `pList` interface includes methods that are counterparts of the STL list such as insert, erase, and splice, additional methods such as split, and asynchronous (non-blocking) variants of some methods for improved performance in a parallel and concurrent environment.

Lists have not been considered beneficial in parallel algorithms because they do not allow random access to its elements. Instead, they access elements through a serializing traversal of the list. In this paper, we will show how our pList offers the advantages of a classical list while providing almost random access to its components, thus enabling scalable parallelism. The `pList` allows certain classes of algorithms to use their most appropriate container, i.e., lists, instead of having to replace it with a more parallel, but less efficient one.

We evaluate the performance of the `pList` on an IBM Power5 cluster and an Opteron-based CRAY XT supercomputer. We analyze the running time and scalability of different `pList` methods as well as the performance of different algorithms using `pList` as data storage. We also compare the `pList` to the `pArray` and `pVector` to understand the relative trade-offs of the various data structures. Our results show that the `pList` outperforms the `pVector` when there are a significant number of insertions or deletions.

2 Related Work

There has been significant research in the area of distributed and concurrent data structure development. Most of the related work is focused either on how to implement concurrent objects using different locking primitives or how to implement concurrent data structures without locking, namely lock free data

structures [11]. Valois [24] was one of the first to present a non-blocking singly-linked list data structure by using Compare&Swap (CAS) synchronization primitives rather than locks. The basic idea is to use auxiliary nodes between each ordinary node to solve the concurrency issues. Subsequent work [10, 16, 8, 17] proposes different concurrent list implementations for shared memory architectures, emphasizing the benefits on non-blocking implementations in comparison with lock based solutions. In contrast, `pList` and the rest of the STAPL `pContainers` are designed to be used in both shared and distributed memory environments. In STAPL we focus on developing a generic infrastructure that will efficiently provide a shared memory abstraction for `pContainers` by automating, in a very configurable way, aspects relating to data distribution and thread safety. We use a compositional approach where existing data structures (sequential or concurrent) can be used as building blocks for implementing parallel containers.

There are several parallel languages and libraries that have similar goals as STAPL [2, 3, 6, 9, 15, 18]. While a large amount of effort has been put into making array-based data structures suitable for parallel programming, more dynamic data structures that allow insertion and deletion of elements have not received as much attention. The PSTL (Parallel Standard Template Library) project [14] explored the same underlying philosophy as STAPL of extending the C++ STL for parallel programming. They planned to provide a distributed_list, but the project is no longer active. Intel Threading Building Blocks (TBB) [12] provide thread-safe containers such as vectors, queues and hashmaps for shared memory architectures, but they do not provide a parallel list implementation. Our work is further distinguished from TBB in that we target both shared and distributed memory systems. New parallel languages like Chapel [5] (developed by CRAY), X10 [7] (developed by IBM), and many others are all aiming to ease parallel programming and to improve productivity for parallel application development. However, most of these languages only provide high level constructs such as multi-dimensional arrays and facilities to specify the distribution of the arrays. A major difference between STAPL and all these new programing languages is that STAPL is a parallel programming library that is written in standard C++ thus making it compatible with existing applications.

3 STAPL Overview

STAPL consists of a set of components that includes `pContainers`, `pAlgorithms`, `views`, `pRanges`, and a runtime system. `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 parallel performance offered 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. Elements in `pContainers` are not replicated. Thread-safety is guaranteed by providing mechanisms that guarantee

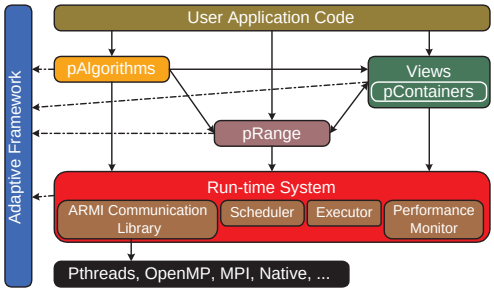


Fig. 1: STAPL components

all operations leave the `pContainer` in a consistent state. An important aspect of STAPL components is their composability, e.g., we can construct `pContainers` of `pContainers`. This supports nested parallelism.

`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 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 workfunction 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 [19, 22]) 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 fence mechanism (`rmi_fence`) to ensure the completion of all previous RMI calls. The asynchronous calls can be aggregated by the RTS in an internal buffer to minimize communication overhead. The buffer size and the aggregation factor impact the performance, and in many cases should be adjusted for the different computational phases of an application.

The RTS provides *locations* as an abstraction of processing elements in a system. A *location* is a component of a parallel machine that has a contiguous memory address space and has associated execution capabilities (e.g., threads). A location can be identified with a process address space.

4 STAPL pList

The linked list is a fundamental data structure that plays an important role in many areas of computer science and engineering such as operating systems, algorithm design, and programming languages. A large number of languages and libraries provide different variants of lists with C++ STL being a representative example. The STL list is a generic dynamic data structure that organizes the elements as a sequence and allows fast insertions and deletions of elements at any point in the sequence. The STAPL `pList` is a parallel equivalent of the STL list with an interface for efficient insertion and deletion of elements in parallel. Analogous to STL lists, elements in a `pList` are accessed through iterators. All STL equivalent methods require a return value, which in general translates into a blocking method. For this reason, we provide a set of asynchronous methods, e.g., `insert_async` and `erase_async`. These non-blocking methods allow for better communication/computation overlap and enable the STAPL RTS to aggregate messages to reduce the communication overhead [19].

Since there is no data replication, operations like `push_back` and `push_front`, if invoked concurrently, may produce serialization in the locations managing the head and the tail of the list. For this reason we added two new methods to the `pList` interface, `push_anywhere` and `push_anywhere_async`, that allow the `pList` to insert the element in an unspecified location in order to minimize communication and improve concurrency. Part of the interface of `pList` is shown in Table 1.

4.1 `pList` Design and Implementation

STAPL provides a uniform approach for developing parallel containers by specifying a set of base concepts and a common methodology for the development of thread-safe, concurrent data structures that are extendable and composable. From the user perspective, a parallel container is a data structure that handles a collection of elements distributed on a parallel machine. A task executing within a location sees the `pContainer` as a shared data structure where elements can be referenced independent of their physical location. Internally, the `pContainer` distributes its data across a number of locations in pieces of storage called *components*. To provide a shared object view, the `pContainer` associates a *Global Unique Identifier (GID)* with every element and uses a *Data Distribution Manager* module to keep track of where the elements corresponding to individual `GIDs` are located. The Distribution Manager in turn uses a `partition` to decide the component in which each element in the `pContainer` is stored, and a `partition-mapper` to determine the location where a component is allocated.

In the following, we provide a more detailed description of the functionality of these modules in the context of the `pList`, but they are general and apply to other `pContainers`.

Component: The `pContainer` component is the basic storage unit for data. For the STAPL `pList`, we use the STL `list` as the component. Similarly, for other `pContainers`, the components are extensions of STL containers or other

pList Interface	**Description**
Collective Operations (must be called by all locations)	
p_list(size_t N, const T& value = T())	Creates a `pList` with N elements, each of which is a copy of `value`.
p_list(size_t N, partition_type& ps)	Creates a `pList` with N elements based on the given partition strategy.
void splice(iter pos, pList& pl);	Splice the elements of `pList pl` into the current list before the position `pos`.
Non-collective Operations	
size_t size() const	Returns the size of the `pList`.
bool empty() const	True if the `pList`'s size is 0.
T& [front|back]()	Access the first/last element of the sequence.
void push_[front|back](const T& val)	Insert a new element at the beginning/end of the sequence.
void pop_[front|back]()	Remove the first element from the beginning/end of the sequence.
iterator insert(iterator pos, const T& val)	Insert val before position pos and return the iterator to the new inserted element.
void insert_async(iterator pos, const T& val)	Insert val before pos with no return value.
iterator erase(iterator pos)	Erases the element at position pos and returns the iterator pointing to the new location of the element that followed the element erased.
void erase_async(iterator pos)	Erases the element at position pos with no return value.
iterator push_anywhere(const value_type& val)	Push val on to the last local component and return the iterator pointing to the new inserted element.
void push_anywhere_async(const T& _val)	Push val on to the last local component with no return value.

Table 1: Representative methods of the `pList` container

available sequential data structures. Most `pContainer` methods will ultimately be executed on the component level using the corresponding method of the component. For example, `pList` insert will end up invoking the STL list insert method. The `pList` component can also be provided by the user so long as insertions and deletions never invalidate iterators, and that components provide the *domain* interface (see below). Additional requirements are relative to the expected performance of the methods (e.g., insertions and deletions should be constant time operations).

Within each location of a parallel machine, a `pList` may store several components and the `pList` employs a `location-manager` module to allocate and handle them. The `pList` has the global view of all of the components and knows the order between them in order to provide a unique traversal of all its data. For this reason each component is identified by a globally unique *component identifier* (`CID`). For static or less dynamic – in terms of number of components

– pContainers such as pArray or associative containers, the component identifier can be a simple integer. The pList, however, needs a component identifier that allows for fast dynamic operations. During the splice operation, components from a pList instance need to be integrated efficiently into another pList instance while maintaining the uniqueness of their CIDs. For these reasons the CID for the pList components is currently defined as follows:

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

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

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

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

Domain: In the STAPL pContainer framework the domain is the universe of GIDs that identifies the elements. A domain also specifies an order that defines how elements are traversed by the iterators of pList. This order is specified by two methods: get_first_gid() which returns the first GID of the domain and get_next_gid(GID) which returns the next GID in the domain of the GID provided as argument. The domain interface for the pList is provided by the pList components.

Data Distribution: The data distribution manager for a STAPL pContainer uses a partition and a partition-mapper to describe how the data will be grouped and mapped on the machine. Given a GID the partition provides the CID containing the corresponding value. The partition-mapper extracts from the CID the location information, and, on that location, a location-manager extracts the actual component to obtain the address of the requested element.

View: In the STAPL framework, views are the means of accessing data elements stored in the pContainer within generic algorithms. STAPL pAlgorithms are written in terms of views, similar to how STL generic algorithms are written in terms of iterators. The pList currently supports sequence views that provide an iterator type and begin() and end() methods. A view can be partitioned into sub-views. By default the partition of a pList view matches the subdivision of the list in components, thus allowing random access to portions of the pList. This allows parallel algorithms to achieve good scalability (see Section 5).

pList Container: The pList class as well as any other pContainer in STAPL has a distribution and a location-manager as its two main data members. The pList methods are implemented using the interfaces of these two classes. A typical implementation of a pList method that operates at the element level is included in Figure 2 to illustrate how the pContainer modules interact. The runtime cost of the method has three constituents: the time to decide the location

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

Fig. 2: `pList` method implementation

and the component where the element will be added (Figure 2, lines 3-5), the communication time to get/send the required information (Figure 2, line 9), and the time to perform the operation on a component (Figure 2, line 7).

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

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

`splice` is a `pList` method whose signature is

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

where `iter` stands for an iterator type. `pos` is an iterator of the calling `pList`, `pl` is another `pList`, and the optional iterators `it1` and `it2` are iterators pointing to elements of `pl`. `splice` removes from `pl` the portion enclosed by `it1` and `it2` and inserts it at `pos`. By default `it1` denotes the begin of `pl` and `it2` the end.

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

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

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

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

```
1 evaluate_performance(N,P)
2    tm = stapl::start_timer();   //start timer
3    //insert N/P elements concurrently
4    for(i=0; i<N/P; ++i)
5       plist.push_anywhere(random_value);
6    rmi_fence(); //ensure all inserts are finished
7    elapsed = stapl::stop_timer(tm); //stop the timer
8    - Reduce elapsed times, getting the max time from all processors.
9    - Report the max time
```

Fig. 3: Kernel used to evaluate the performance of `pList` methods.

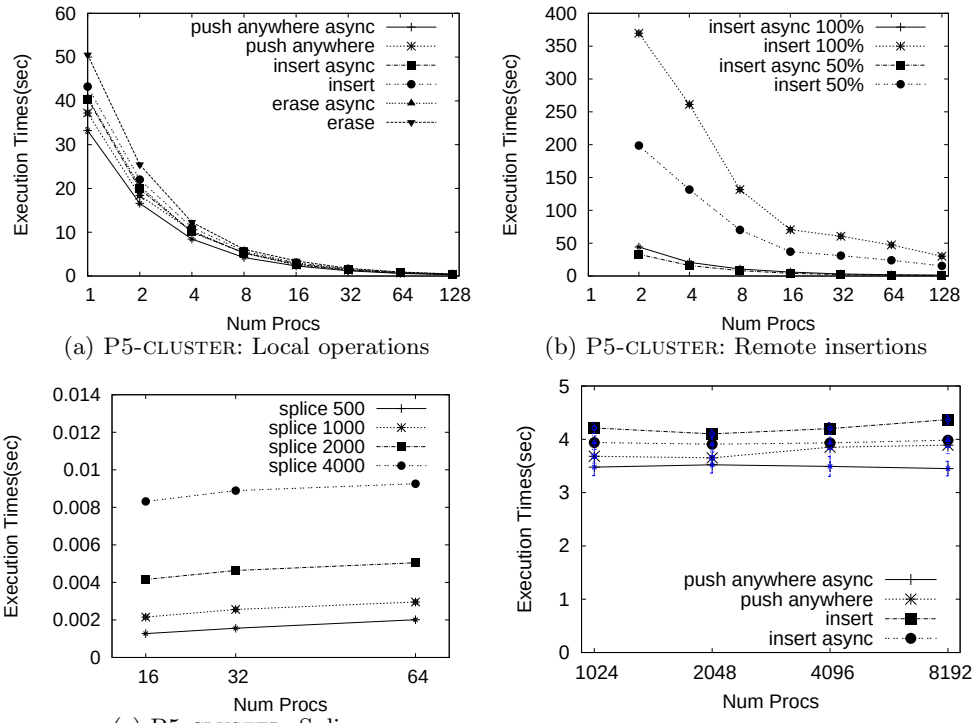
5 Performance Evaluation

In this section, we evaluate the scalability of the parallel methods described in Section 4. We compare `pList` and `pVector` performance, evaluate some generic `pAlgorithms` (`p_generate` and `p_partial_sum`) on `pList`, `pArray` and `pVector`, and evaluate an Euler tour implementation using `pList`. We conducted our experimental studies on two architectures: a 832 processor IBM cluster with p575 SMP nodes available at Texas A&M University (called P5-CLUSTER) and a 38,288 processors Cray XT4 with quad core Opteron processors available at NERSC (called CRAY). In all experiments a location contains a single processor, and the terms can be used interchangeably.

5.1 `pList` Method Evaluation

In this section we discuss the performance of the `pList` 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 `push_anywhere`, but the same kernel is used to evaluate all methods. For a given number of elements N, all P available processors (locations) concurrently insert N/P elements. We report the time taken to insert all N elements globally. The measured time includes the cost of a fence call which, as stated in Section 3, is more than a simple barrier.

Figure 4 shows the execution time of different `pList` methods. In a first study, all methods are executed locally and we observe in Figure 4(a) that both synchronous and asynchronous method exhibit scalable performance as they do not incur any communication. In Figure 4(b) we show the execution time for a mix of local and remote method invocations to highlight the advantages of the asynchronous method over the synchronous ones. When all methods invoked will be executed remotely the `insert` method that returns an iterator to the newly inserted element is on average seven times slower than the `insert_async` method, which does not return a value and can exploit message aggregation. In Figure 4(d) we show a weak scaling experiment we performed on CRAY using 25 million elements per processor and up to 8192 processors (104.8 billion method invocations performed globally). The `push_anywhere` methods are faster than `insert` as they don't perform any additional searches for a position where to add the element. The asynchronous versions are faster as they don't return an iterator to the newly added element.



(a) P5-CLUSTER: Local operations

(b) P5-CLUSTER: Remote insertions

(c) P5-CLUSTER: Splice

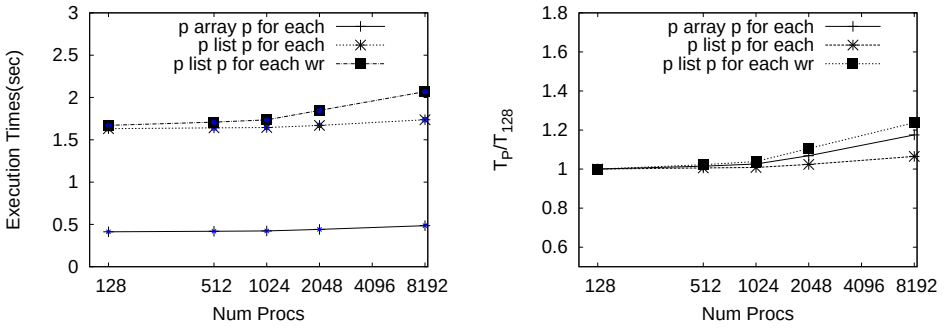
(d) CRAY: Weak scaling

Fig. 4: `pList` methods comparison. (a) Execution times when all operations are executed locally (N=64 millions). (b) Execution times for `insert` and `insert_async` when all operation are remote (100%) and when 50% of operations are remote. (c) `splice` for 500 to 4000 components per location. (d) Weak scaling for `pList` methods on large number of processors using 25 million elements per processor.

To evaluate the `splice` method, we splice a `pList` with a fixed number of components per location into another `pList`. Figure 4(c) shows the execution time on P5-CLUSTER for the splice operation for different number of components per location and for various number of locations. As expected, the time increases linearly with the number of spliced components, but increases only slowly with the number of processors (almost constant), ensuring good scalability.

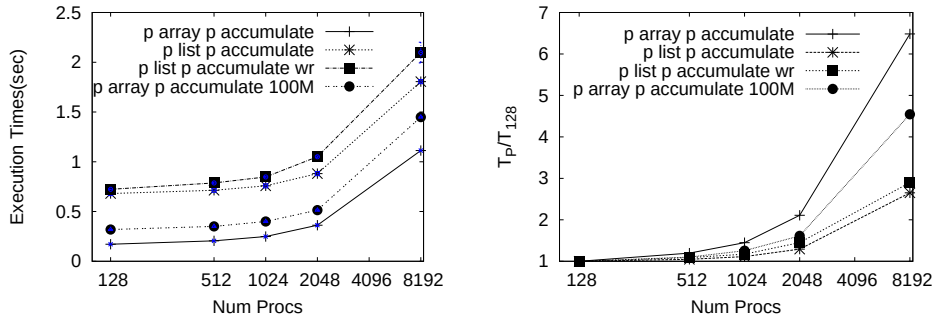
5.2 pAlgorithms Comparison

In this section we present the performance of two generic non-mutating `pAlgorithms`, `p_for_each` and `p_accumulate`, which store their data in two different STAPL `pContainers`: `pList` and `pArray`. For all the algorithms considered in this section, for both P5-CLUSTER and CRAY, we conducted weak scaling experiments. Strong scaling would be difficult to evaluate due to the short execution times of the algorithms even when run on very large input sizes.



(a) `p_for_each` execution times

(b) `p_for_each` normalized execution times



(c) `p_accumulate` execution times

(d) `p_accumulate` normalized execution times

Fig. 5: CRAY: `pAlgorithms` weak scaling on large number of processors. The number of elements per processor is 50M except for `p_array_p_accumulate_100M` where we used 100M.

The `p_for_each` algorithm increments the elements of the container with a given constant performing read, add and store operations on each individual element without the need of any remote accesses. The `p_accumulate` accumulates all the elements in the container using a generic map reduce operation available in STAPL. To execute the reduce with a non-commutative operator, the rank of each component of the `pList` has to be computed. When a `pList` is instantiated it initializes the rank of all of its components. However, dynamic operations like splice and split modify the number of components of a `pList` and invalidate the ranking of the `pList`'s components. In this case the ranks of the components must be re-computed, thus increasing the overall execution of the algorithm.

Figure 5(a) presents the execution times for `p_for_each` on CRAY from 128 to 8192 processors. The three curves corresponds, respectively, to `pArray`, `pList` when the ranks are available, and finally when list ranking is invoked (wr). Figure 5(b) shows the corresponding normalized execution time (T_P/T_{128}).

We observe that the execution time for `pList` is higher than for `pArray`, which is the result of the longer access time to the elements of the STL list with respect to the STL valarray. In Figure 5(b) we see that on 8192 processors there is a 18% performance degradation for `pArray`, and a 20% for `pList` when

ranking is (re)computed, and a 5% degradation when the ranking of the `pList` can be reused. Figures 5(c) and (d) show the performance of `p_accumulate`, an algorithm that uses the map-reduce computation with a communication term accounting for a time proportional to $\log p$, p being the number of processors. For `pArray`, due to the low execution time taken by the map phase, the overhead of communication is close to 650% when using 50 million elements per processor and 450% when we increase to 100 million elements per processor. For `pList` due to a larger computation time in the map phase, this value reduces to almost 300%.

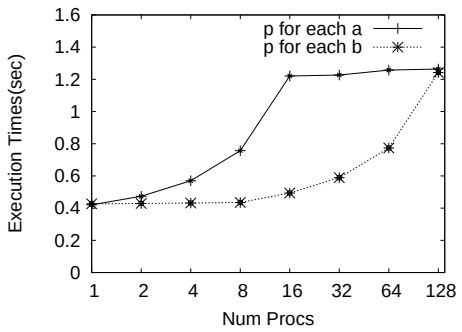


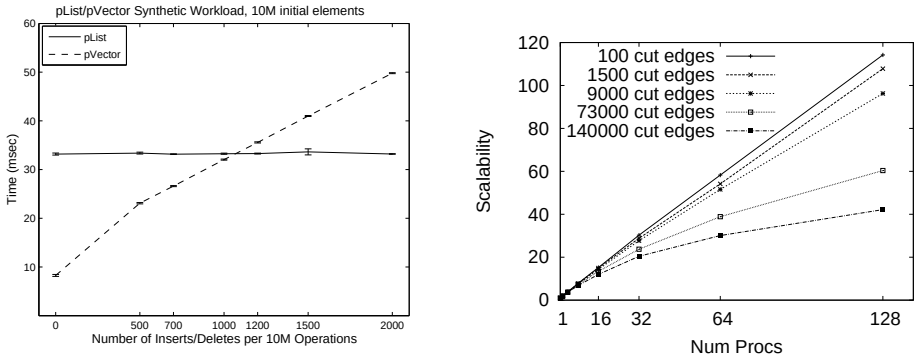
Fig. 6: P5-CLUSTER: Weak scaling for `p_for_each` allocating processes on the same nodes when possible (curve a) or in different nodes (curve b). Experiments are for 20 million elements/processor.

Figure 6 shows two weak scaling experiments on P5-CLUSTER for two different processor allocation strategies. Each node of P5-CLUSTER has 16 processors. In the figure, `p_for_each`-a represents the case where all processors are allocated on a single node (possible for 1-16 processors). `p_for_each`-b represents the case where we use cyclic allocation across 128 processors, e.g., 16 processors would be allocated one per node, and in general, there will be $P/8$ processors allocated on each node for $P < 128$. The reason why the two curves do not match is related to memory bandwidth saturation within a node. In `p_for_each`-b, the nodes are fully utilized only when running on 128 processors, while for `p_for_each`-a we use all processors in a node when running on 16 processors or more. These experiments emphasize the importance of a good task placement policy on the physical processors.

5.3 Comparison of Dynamic Data Structures in STAPL

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

In Figure 7 (a), we show results for both containers on the P5-CLUSTER for 16 processors and 10 million elements. We perform 10 million operations per



(a) `pList` and `pVector` comparison (b) Euler tour scaling

Fig. 7: (a) Comparison `pList` and `pVector` dynamic data structures using a mix of 10M operations (read/write/insert/delete). (b) Strong scaling for Euler tour on various input trees: 32M vertices and different number of cut edges.

container. Each operation is either a read or write of the next element in the container, an insertion at the current location, or deletion of the current element. These operations are distributed evenly among the processors, which perform them in parallel. For these experiments, the combined number of insertions and deletions is varied from 0 to 2000, with the remaining operations being an equal number of reads and writes. More insertions or deletions than this cause the runtime of the `pVector` to increase dramatically.

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

5.4 Application using `pList`: Euler Tour

To evaluate the utility of the `pList` in a real application we implemented the parallel Euler tour algorithm described in [13]. The algorithm takes as input an arbitrary tree, as a STAPL `pGraph`, and generates a list with the edges of the Euler tour using `insert_async` operations. The algorithm first generates a `pList` with chunks of the Euler tour and a `pMap` with the starting points of the chunks. A second phase rearranges the links of the `pList` to produce the final Euler tour.

Performance evaluation is done by using different randomly generated trees that allow us to control the average number of cross-location edges. Figure 7(b) shows the speedup of the Euler tour algorithm for different number of remote edges per processor. The scalability is relative to P=1 when there are no remote

edges. For any other processor count the number of remote edges grows linearly with the number of processors. As expected, we observe that the algorithm scales well as long as the number of remote edges is relatively small and the scalability degrades as the overall number of remote edges increases. The good overall scalability is due to the efficient `pContainers` and methods used, in particular, the `pList` and the `insert_async` method.

6 Conclusion

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

In summary, and most importantly, we have designed and implemented a parallel container that has the properties of the sequential list, but allows for scalable concurrent access when used in a parallel program.

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 Int. Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Bucharest, Romania, 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, 1993.
4. A. 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 (LCPC)*, LNCS 5335, pages 304–315, Edmonton, Alberta, Canada, July 2008.
5. D. Callahan, Chamberlain, B.L., and H. Zima. The Cascade high productivity language. In *The 9th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments*, volume 26, pages 52–60, Los Alamitos, CA, USA, 2004.
6. A. Chan and F. K. H. A. Dehne. CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on PC clusters. In *EuroPVM/MPI*, pages 117–125, 2003.
7. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proc. of the 20th ACM SIGPLAN conf. on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.
8. M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. ACM Symp. on Principles of Distributed Processing (PODC)*, pages 50–59, New York, NY, USA, 2004. ACM.

9. D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not.*, 40(10):423–437, 2005.
10. T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proc. Int. Conf. Distributed Compututing*, pages 300–314, London, UK, 2001. Springer-Verlag.
11. M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Prog. Lang. Sys.*, 15(5):745–770, 1993.
12. Intel. *Reference for Intel Threading Building Blocks, version 1.0*, April 2006.
13. J. JàJà. *An Introduction Parallel Algorithms*. Addison–Wesley, Reading, Massachusetts, 1992.
14. E. Johnson and D. Gannon. HPC++: experiments with the parallel standard template library. In *Proc. of the 11th Int. Conference on Supercomputing (ICS)*, pages 124–131, Vienna, Austria, 1997.
15. L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993.
16. M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. of the 14th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA)*, pages 73–82, Winnipeg, Manitoba, Canada, 2002. ACM Press.
17. W. Pugh. Concurrent maintenance of skip lists. Technical Report UMIACS-TR-90-80, University of Maryland at College Park, College Park, MD, USA, 1990.
18. 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.
19. S. Saunders and L. Rauchwerger. Armi: an adaptive, platform independent communication library. In *Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 230–241, San Diego, California, USA, 2003. ACM.
20. G. Tanase, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pArray. In *Proceedings of the 2007 Workshop on Memory Performance (MEDEA)*, pages 73–80, Brasov, Romania, 2007.
21. 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 (LCPC)*, LNCS 5234, pages 156–171, Urbana-Champaign, 2008.
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, 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. of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288, Chicago, IL, USA, 2005. ACM.
24. J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. ACM Symp. on Principles of Distributed Processing (PODC)*, pages 214–222, New York, NY, USA, 1995. ACM.