STAPL: Standard Template Adaptive Parallel Library[*]

Antal Buss[†]
abuss@cse.tamu.edu

Harshvardhan[†]
ananvay@cse.tamu.edu

Ioannis Papadopoulos[†]
ipapadop@cse.tamu.edu

Olga Pearce[†]
olga@cse.tamu.edu

Timmie Smith[†]
timmie@cse.tamu.edu

Gabriel Tanase[†]
gabrielt@cse.tamu.edu

Nathan Thomas[†]
nthomas@cse.tamu.edu

Xiabing Xu[†]
xiabing@cse.tamu.edu

Mauro Bianco [†]
bmm@cse.tamu.edu

Nancy M. Amato[†]
amato@cse.tamu.edu

Lawrence Rauchwerger[†]
rwerger@cse.tamu.edu

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming; E.1 [**Data Structures**]; D.2.11 [**Software Architectures**]

General Terms

High Productivity Parallel Programming, Library, Parallel Data Structures

ABSTRACT

The Standard Template Adaptive Parallel Library (STAPL) is a high-productivity parallel programming framework that extends C++ and STL with unified support for shared and distributed memory parallelism. STAPL provides distributed data structures (`pContainers`) and parallel algorithms (`pAlgorithms`) and a generic methodology for extending them to provide customized functionality. The STAPL runtime system provides the abstraction for communication and program execution. In this paper, we describe the major components of STAPL and present performance results for both algorithms and data structures showing scalability up to tens of thousands of processors.

1. INTRODUCTION

The increasing availability of multiprocessor and multicore architectures and the need to solve larger and more complex problems makes parallel programming crucial for application development. A major challenge of parallel programming is productivity, which is mainly affected by programmability and portability of performance. Absolute performance is crucial since the main goal of parallel applications is to outperform sequential ones, thus justifying the investment in a parallel architecture. Parallel algorithms are sensitive to system architecture (latency, topology, etc.) and to application data (data type, distribution, density, etc.). Consequently, it is essential for parallel software to provide good performance when moving from one platform to another. Programmability refers to the problem of simplifying parallel program development by hiding the details of concurrency orchestration and by providing linguistic constructs and tools that can naturally express parallel algorithms.

To reconcile these two conflicting requirements, we are developing the Standard Template Adaptive Parallel Library (STAPL) [2, 30, 24, 26, 1]. The STAPL framework for parallel C++ code development is intended to play a similar role in facilitating parallel program development as the ISO C++ standard library [21] plays for sequential C++ programming. STAPL supports portable performance by providing the tools and mechanisms that enable STAPL programs to continually adapt to the system and the data at all levels – from selecting the most appropriate algorithmic implementation to tuning communication granularity and frequency. Although more experienced users may later choose to refine performance manually using mechanisms provided by STAPL, a major design goal of STAPL is to provide good performance even for the novice parallel programmer.

STAPL provides three levels of abstraction appropriate to an *application developer* (level 1), a *library developer* (level 2), and a *run-time system developer* (level 3). At the highest abstraction level (application developer), STAPL offers an STL-like interface to a generic parallel machine. Parallel programs can be composed by non-expert parallel programmers using building blocks from the core STAPL library. Users don't have to (but can) be aware of the distributed nature of the machine. At the intermediate level, STAPL exposes sufficient information to allow a library developer to implement new STAPL-like algorithms and containers, i.e., to expand the STAPL base or build a domain specific library. This

[*]This research supported in part by NSF Grants CRI-0551685, CCF-0833199, CCF-0830753, IIS-096053, IIS-0317266 by the DOE NNSA under the Predictive Science Academic Alliances Program by grant DE-FC52-08NA28616, by Chevron, IBM, Intel, and HP. 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.

[†]Parasol Lab, Dept. of Computer Science and Engineering, Texas A&M University, 3112 TAMU, College Station, Texas, 77843-3112

SYSTOR 2010, May 24-26, Haifa, Israel

is the lowest level at which the "usual" user of STAPL operates. At the lowest level, the RTS developer has access to the implementation of the communication and synchronization library, the interaction between OS, STAPL thread scheduling, memory management and machine specific features such as topology and memory subsystem organization.

There are several applications being developed using STAPL, such as seismic ray-tracing for geophysics applications, particle transport for radiation simulation, and motion planning for robots and complex molecules such as proteins. In this paper, we describe some of the main STAPL concepts. In Section 2 we provide a more detailed overview of the STAPL components followed by related work in Section 3. Sections 4 and 5 provide descriptions of **pContainers** and **pAlgorithms**, respectively. Section 6 describes the STAPL runtime system. Finally, Section 7 shows some preliminary results about scalability and execution times of STAPL **pContainers** and **pAlgorithms**.

2. STAPL OVERVIEW

STAPL is a framework for parallel code development in C++. Its core is a library of C++ components with interfaces similar to the (sequential) ISO C++ standard library [21]. STAPL offers the parallel system programmer a shared object view of the data space. The objects are distributed across the memory hierarchy which can be shared and/or distributed address spaces. Internal STAPL mechanisms assure an automatic translation from one space to another, presenting a unified address space to the less experienced user. For more experienced users, the local/remote distinction of accesses can be exposed and performance enhanced for a specific application or application domain. To exploit large hierarchical systems, such as BlueGene, Cray Jaguar, STAPL allows for (recursive) nested parallelism (as in NESL [4]).

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

pContainers, the distributed counterpart of STL containers, are thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. They are composable and extensible by users via inheritance. Currently, STAPL provides counterparts of all STL containers (e.g., **pArray**, **pVector**, **pList**, **pMap**, etc.), and **pContainers** that do not have STL equivalents: parallel matrix (**pMatrix**) and parallel graph (**pGraph**). **pContainers** provide two kinds of methods to access their data: methods which are semantically equivalent to their sequential counterpart and methods which are specific to parallel computations. For example, STAPL provides an **insert_async** method that can return control to the caller before its execution completes. While a **pContainer**'s data may be distributed, **pContainers** offer the programmer a *shared object view*, i.e., they are shared data structures with a global address space. This is provided by an internal object translation method which can transparently locate both local and remote elements. The physical distribution of a **pContainer** data can be assigned automatically by STAPL or can be user-specified.

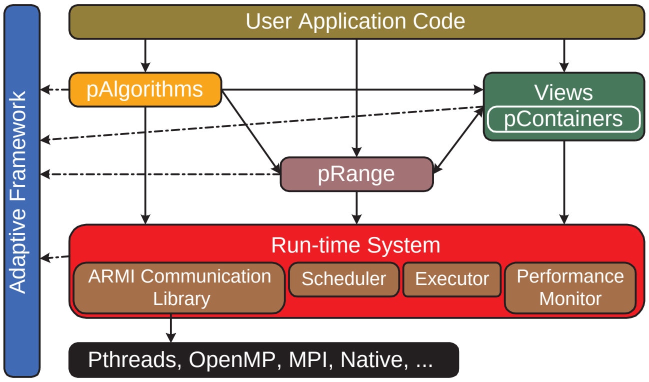


Figure 1: STAPL Overview

A **pAlgorithm** is the parallel equivalent of an STL algorithm. STAPL currently includes a large collection of parallel algorithms, including parallel counterparts of STL algorithms, **pAlgorithms** for important parallel algorithmic techniques (e.g., prefix sums, the Euler tour technique), and some for use with STAPL extensions to STL (i.e., graph algorithms for **pGraph**). Analogous to STL algorithms that use *iterators*, STAPL **pAlgorithms** are written in terms of **views**. Briefly, **views** allow the same **pContainer** to present multiple interfaces to its users, e.g., enabling the same **pMatrix** to be 'viewed' (or used) as a row-major or column-major matrix or even as linearized vector.

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

The platform dependent STAPL components are mainly contained in the STAPL runtime system (RTS), which provides the API to the OS and several important functions. The RTS includes the ARMI (Adaptive Remote Method Invocation (RMI)) communication library that abstracts interprocessor communication for the higher level STAPL components. The RTS also contains the **executor** and **scheduler** modules that are responsible for allocating resources for the computation and for executing it, and the performance monitor. ARMI abstracts communication of data and work across the distributed memory machine by providing a common remote method invocation (RMI) interface to all other STAPL components [26]. There is also support in ARMI for collective operations common in parallel program-

Features/ Project	Paradigm[1]	Architecture	Nested	Adaptive	Generic	Data Distribution	Scheduling	Overlap comm/comp
STAPL	S/MPMD	Shared/Dist	Yes	Yes	Yes	Auto/User	Customizable	Yes
PSTL	SPMD	Shared/Dist	No	No	Yes	Auto	Tulip RTS	No
Charm++	MPMD	Shared/Dist	No	No	Yes	User	prioritized execution	Yes
CILK	S/MPMD	Shared/Dist	Yes	No	No	User	work stealing	No
NESL	S/MPMD	Shared/Dist	Yes	No	Yes	User	work and depth model	No
POOMA	SPMD	Shared/Dist	No	No	Yes	User	pthread scheduling	No
SPLIT-C	SPMD	Shared/Dist	Yes	No	No	User	user	Yes
X10	S/MPMD	Shared/Dist	No	No	Yes	Auto	-	Yes
Chapel	S/MPMD	Shared/Dist	Yes	No	Yes	Auto	-	Yes
Titanium	S/MPMD	Shared/Dist	No	No	No	Auto	-	Yes
Intel TBB	SPMD	Shared	Yes	Yes	Yes	Auto	work stealing	No

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

Table 1: Comparison with related projects

ming, such as broadcasts and reductions. Its implementation is machine dependent and it can generate synchronous or asynchronous messages (e.g., MPI messages) or synchronizations, e.g., OpenMP synchronizations. The `executor` and `scheduler` provide support for an assortment of common scheduling policies and load balancing strategies, and also provide mechanisms through which users can add their own versions of these services. The STAPL run-time system will support nested parallelism if the underlying architecture allows nested parallelism via a hierarchical native runtime system. Otherwise, the runtime system will serialize the nested parallelism.

3. RELATED WORK

There is a relatively large body of work that has similar goals to STAPL. PSTL [17], TBB [15], POOMA [25], and STAPL borrow from the STL philosophy, i.e., they provide concepts such as containers, iterators, and algorithms. The Parallel Standard Template Library (PSTL) had similar goals to STAPL; it uses parallel iterators as a parallel equivalent to STL iterators and provides some parallel algorithms and containers. However, PSTL was focused on STL compatibility, while STAPL extends STL by introducing additional parallel data structures and management of data dependencies to specify generic parallel algorithms. Intel's Threading Building Blocks (TBB) [15] implements some of the STAPL concepts but targets only shared memory systems (multi-cores in particular). STAPL targets both shared and distributed systems with heavy emphasis on developing an extensible infrastructure. There are also other important differences in the way computational patterns are specified. Instead of taking the exclusive *d*ivide-and-conquer of data approach, STAPL also also allows (and perhaps favors) a *c*ompose-and-conquer tasks strategy.

Projects like NESL [3], CILK [12], Split-C [9], Chapel [7], and X10 [8] provide the ability to exploit nested parallelism. In addition to nested parallelism, STAPL is intended to automatically generate recursive parallelization without user intervention. Several languages/libraries abstract shared memory machines, hiding the details of the data distribution from the user. Others provide the user with a *partitioned global address space (PGAS)*, including Split-C [9], X10 [8], Chapel [7], and Titanium [32]. In PGAS languages, memory accesses have different costs for local and remote data. STAPL provides a shared memory abstraction for the naive user while exposing a PGAS architecture to the more advanced user, decoupling the application from library development. STAPL also focuses on interoperability with other languages and libraries [6], to allow the use of well established and highly optimized routines.

Adaptivity refers to an application's ability to improve resource usage based on architectural parameters and dynamic information, such as input characteristics, computational load, etc. STAPL is designed to emphasize adaptivity, providing frameworks for algorithm selection, data distribution selection, and to refine or coarsen the degree of parallelism during execution of a program. TBB also provides some degree of adaptivity by allowing the recursive refinement of data ranges according to specified policies and dynamic conditions. STAPL addresses irregular applications both by providing irregular data structures (such as trees, lists, and graphs), and by emphasizing the development of algorithms by specifying classes of data dependencies. X10 [8] and Charm++ [19] also provide support for irregular applications through their concepts of *activity* and *chare* objects, respectively. In Table 1 we report a summary of the comparison with other projects in terms of programming and execution models, and supported architectures.

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 lock-free data structures [14]. Valois [31] 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 [13, 20, 10, 23] proposes different concurrent list implementations for shared memory architectures, emphasizing the benefits on non-blocking implementations in comparison with lock based solutions.

In the area of runtime systems, there are other projects that share some of the characteristics of the STAPL RTS. The runtime system of TBB [15] provides the ability to express algorithms using tasks and uses work-stealing algorithms and task affinity to provide efficient mapping of tasks to hardware threads. However it is only suited for shared memory systems. TBB is planned to be used as a multithreaded back-end for the STAPL RTS. Nexus [11] has the Remote Service Requests (RSR) that are non-blocking Remote Procedure Calls (RPC). An RSR can spawn a thread at the destination to execute the procedure. On the other hand it does not provide blocking communication primitives and is restricted to only POSIX threads. ARMCI [22] provides one-sided communication primitives that focus on optimizing strided data communication. ARMCI also employs buffering to minimize sent messages and maximize bandwidth. GASnet [5] provides a global shared memory abstraction to the user and allows memory to copied or be copied from every process to any other process. Both ARMCI and GASnet do not offer any kind of task support. Converse [18] provides a means for making calls to remote objects via remote pointers and supports user-level threads and provides load-balancing. Converse requires significant effort to be ported onto new systems, since it does not take advantage of native communication or threading libraries, which is a key design choice of the STAPL RTS to speed up the process of porting the library to different platforms.

4. THE PARALLEL CONTAINER FRAMEWORK

`pContainers` are collections of elements that are distributed across a parallel machine and accessed concurrently. The *STAPL Parallel Container Framework (PCF)*, consists of a set of formally defined concepts and a methodology for facilitating the process of developing generic parallel containers which are thread safe and composable. At the highest level, the framework allows users to assemble a `pContainer` from a collection of already existing containers, called *base containers*, or `bContainers`, and some additional information (called *data-distribution information*) to handle data distribution and locality information. This information is managed by the *data distribution manager*.

The STAPL PCF distinguishes itself from existing work through a number of features. We provide an integrated approach for all `pContainers`, encouraging software reuse and improving user productivity. This allows us to develop a large number of data structures in a uniform fashion, e.g., `pArray`, `pList`, `pVector`, `pMatrix`, `pGraph`, `pMap`, `pSet`, `pHashMap`, and others [28, 27, 6, 29]. STAPL `pContainers` can be arbitrarily composed, by allowing the elements of a `pContainer` to be `pContainers` themselves. This enables users to express more complicated data structures and is one natural way to specify nested parallelism. By design, at the interface level, data elements of `pContainers` are not replicated, so the user does not need to deal with data replication and caching.

The `pContainers` provide *synchronous* methods to access their data elements, such as assignment `x = container[i]`. Their semantics is similar to their sequential counterpart. They also provide *asynchronous* methods, by leveraging the corresponding RMI primitives described in Section 6. For instance, a synchronous method to insert an element in a container looks like `it = container.insert_synch(value)`, while the asynchronous version looks like `container.insert_async(value)`. Because the asynchronous method does not return a value, its caller can proceed to the next instruction without delay.

Like all other STAPL components, `pContainers` allow full customization. The PCF is open and allows users to develop new containers by deriving from the provided base implementations. Moreover, existing `pContainers` support data redistribution at runtime, and data distributions can be customized for specific applications. Thread safety and data consistency policies can also be tailored to algorithm and application specific characteristics.

The concepts defined by the PCF for managing the data and the distribution information are the *Global Identifier, Domain, Data Distribution, Partition, Partition Mapper, Base Containers, Location Manager* and *View*. We provide an overview of these concepts next.

Base Container: The data stored in a `pContainer` is organized as a collection of containers that we refer to as *Base Containers* or `bContainers`. The `bContainers` are obtained by extending existing containers with an appropriate interface to allow easy and uniform interaction with the PCF. For example, the `bContainers` of the current STAPL `pContainers` that are counterparts of STL containers are based on the STL containers themselves. However, this is a flexible decision and other implementations from other libraries can be used.

Global Identifier (`GID`): A STAPL `pContainer` uniquely identifies its elements by associating them with a `GID`. This is an important requirement that allows us to provide a shared object view. Having access to the `GID` of an element, the `pContainer` can identify its location on a distributed memory machine. For example, for a parallel array (`pArray`) the `GID` is the index of the element, for a simple associative `pContainer` the `GID` is the *key* while for a multi-associative `pContainer` the `GID` is a pair (key, m), where m is an integer used to manage multiplicity.

Domain: The `pContainer` *domain* is the universe of `GIDs` that identifies its elements. The domain for a `pArray` is a finite set of indices while for an associative `pContainer` is a set of keys. Domains can be ordered sets, thus allowing their elements to be uniquely traversed.

Data Distribution: The Data Distribution is responsible for determining the location where an element associated with a `GID` is located. A *location* is a component of a parallel machine that has a contiguous memory address space and has associated execution capabilities (e.g., threads). A location may be identified with a process address space. The data distribution manager uses (i) a `partition` to determine a decomposition of a domain into sub-domains and to specify for every `GID` in the domain to which sub-domain it has been allocated, and (ii) a `partition-mapper` to determine to which location each sub-domain has been allocated. This class provides the routing of the methods to the proper location and guarantees the thread safety of the operations performed by the user. The user has the option to provide additional `partition` and `partition-mapper` classes if those provided by STAPL do not meet their needs.

Location Manager: A location may store a sub-set of the `bContainers` of a `pContainer`. The `pContainer` will employ, within each location, a *Location Manager* to maintain the collection of `bContainers`. The location manager may use different optimizations for the storage of `bContainers`

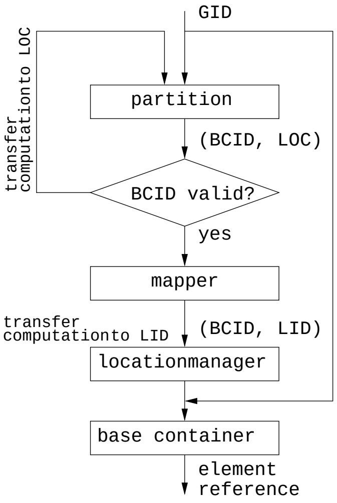


Figure 2: `pContainer` modules interaction to find the element reference corresponding to a given `GID`.

depending on the properties of particular data structures. For example, different memory allocators may be used to allocate the space required by the `bContainer`s.

pContainer Container Framework: To automate the process of developing `pContainers`, the PCF includes an (almost) hierarchically structured collection of `pContainer` base classes. For each component of this collection the framework provides a default implementation for `GID`s, domains, partitions, partition mappers and location managers that developers can use as is or specialize to further improve the performance of their data structures.

In Figure 2 we depict the address resolution that a `pContainer` performs for methods accessing the element corresponding to a given `GID`. The `GID` is first mapped using the partition to a `bContainer` identified by its unique identifier (BCID). If the BCID is unknown in the location where the method is invoked, then a location identifier is returned (LOC) which indicates a location that may know about the requested BCID. The computation is then transferred to the LOC and the method re-evaluated. Otherwise, the `bContainer` identifier is mapped into a particular location using the partition mapper which returns a location identifier (LID) indicating where to execute the methods. At this point, the `pContainer` invokes the method on the location owning the `bContainer`, where a location manager is used to map the BCID to the particular `bContainer`. The `bContainer` is then used to access the element corresponding to the given `GID`.

5. PARALLEL ALGORITHMS IN STAPL

As mentioned previously, algorithms in STAPL are expressed as *task graphs*, where vertices represent tasks and edges represent the dependencies among them. A task is a pair consisting of data, represented by a view, and work, also called a *workfunction*. To help the application developer in specifying algorithms, the library provides *computational patterns*. STAPL provides generic algorithms to the application developer and the corresponding patterns that extend the C++ STL library, and other useful algorithms, such as parallel prefix, and common graph algorithms.

```
1:struct strmatch {
2:  typedef access_list<R> view_access_types;
3:  string pat;
4:  strmatch(string _pat) : pat(_pat) {}

5:  template<typename View>
6:  bool operator()(View x) const {
7:    return stapl::p_equal(pat,x);
8:  }
9:};
```

Figure 3: Implementation of the user workfunction for the string matching algorithm.

Common patterns provided by STAPL are `map`, `map_reduce` and `partial_sum`. To use the patterns the user indicates the input data and the operators to be applied during the computation. The data, represented as (and accessed though) a `view`, can be partitioned as desired by the user. For instance, assume the user wants to count how many times a given string appears in a text. This instance of the string matching problem can be implemented as a `map_reduce`, where the map operation compares each substring of the text with the string to match, while the reduction counts how many matches have been encountered. To do this, the user partitions the input text into the substrings of the size of the pattern to match, and then executes a `map_reduce` pattern with the `strmatch` as the map operator and `plus` as the reduce operator. One way of implementing this algorithm is as follows:

```
int mts = map_reduce(
  overlap_view(text_view, 1, 0, pattern.size()-1),
  strmatch(pattern),
  std::plus<char>);
```

where `text_view` contains the text to be searched, while `pattern` contains the string to match, and `mts` will store the result of the computation. `overlap_view(V, s, l, r)` indicates that the `view V` has to be partitioned into pieces where the elements are accessed with stride `s` and each element comprises `l` elements on the left and `r` elements on the right. In this context, this means that we are selecting all elements with stride 1 and 0 elements on the left and `pattern.size()-1` on the right. Since only full portions are considered, we are partitioning the text into all its `text.size()-pattern.size()+1` substrings of length `pattern.size()`.

The map operation `strmatch` can be implemented using a function-object as in Figure 3. The function operator of `strmatch` invokes the STAPL algorithm `p_equal` that can execute in parallel (nested parallelism).

The computational pattern is always specified exposing the maximum amount of parallelism. It is left to the library to translate the fine grain parallel algorithm into an appropriate task graph that exploits the characteristics of the underlying platform.

It is worth noting that the function operator of the class is templated (line 5 of Figure 3). This detail exposes an important feature of STAPL. In the attempt to optimize the data access performed by a task, the type of view passed to the user functions (function-objects) can be substituted by the library itself. For instance, the access through `pContainer` methods that provide a shared memory view of the data can be substituted with direct memory access through pointers if the proper conditions hold. In the same way, the user function describes if its arguments are read-only, write-only, or read-write. To do this, the user can specify, as in line 2 of Figure 3, that the input of the function operator is only read by the function operator. This is used by the STAPL task placement policy to decide in which location and what representation of the data is the best fit for a given task. If the user does not specify the access type, STAPL assumes, conservatively, that the access is read-write.

6. RUNTIME SYSTEM

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

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

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

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

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

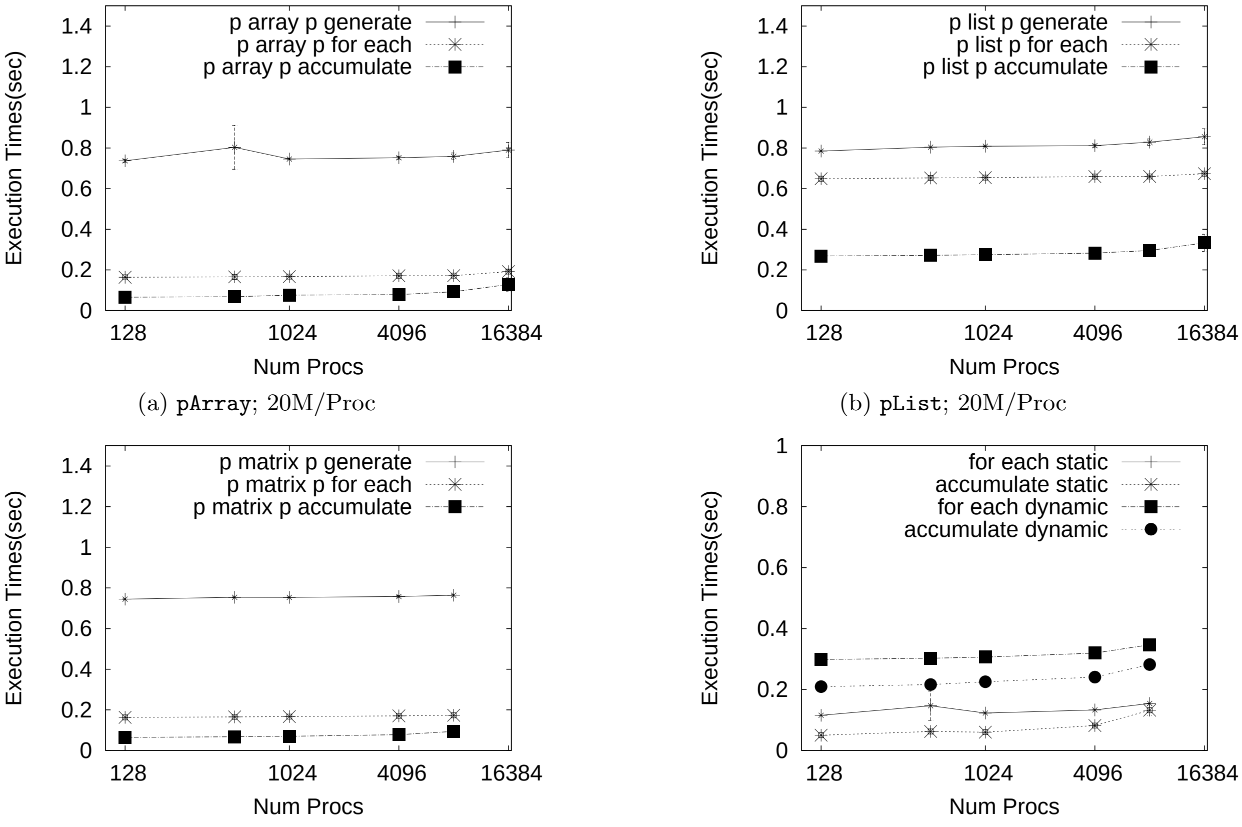
Another RTS component, the *executor*, has the role of executing task graphs corresponding to `pAlgorithm`s. The executor identifies sets of independent tasks to be executed, and schedules them according to the customizable scheduler module. From its perspective, the executor treats incoming RMI requests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

7. PERFORMANCE EVALUATION

In this section, we evaluate the scalability of algorithms and data structures in STAPL. We conducted our experimental study on two architectures: a 38,288 processor Cray XT4 with quad core Opteron processors available at NERSC (referred to as CRAY) and a 128 processor IBM cluster with p575 SMP nodes available at Texas A&M University (referred to as P5-CLUSTER). In all experiments, a location contains a single processor, and the terms can be used interchangeably.

In Figure 4, we show the execution times for the `pAlgorithm`s `p_generate`, `p_for_each` and `p_accumulate` on `pArray`, `pList`, `pGraph`, and `pMatrix`. `p_generate` is a mutating algorithm that overwrites each of the elements passed to it with the result of a nullary function, which in our experiments generates random values. `p_for_each` is a mutating algorithm that applies a function to each of the elements passed to it and overwrites each element with the corresponding result; the order in which the function is applied is not specified. `p_accumulate` computes the sum of all the elements of the container passed to it; it implements a basic parallel reduction operation on the elements passed to it. The figure shows a weak scaling experiment for all three algorithms on (a) `pArray`, (b) `pList`, (c) `pMatrix`, and (d) `pGraph`. The experiment is performed with 20M elements per processor for `pArray`, `pList`, and `pMatrix`, while for `pGraph` we study both static and dynamic graph representations in which each processor stores a stencil of a 2D torus of size 1500 × 1500. This analysis highlights the performance differences of the various data structures.

From the plot in Figure 4(a) we observe that for the `pArray` the performance degradation is 5% when scaling from 128 to 16,385 processors for `p_generate` and `p_for_each`. For `p_accumulate`, which performs a global reduction to provide the result, the performance degradation is about 40%. This is due to the limited amount of computation performed to access and add local elements relative to the communication cost of the reduction. The `pMatrix` results included in Figure 4(c) show the same behavior as the `pArray` for all three algorithms as we scale from 128 to 8192 processors. For `pList`, we show the results in Figure 4(b). We observe the same overall good scalability as we increase the number of processors from 128 to 16,384 but the overall execution times are slower for all three algorithms when compared to `pArray` and `pMatrix`. This is due



(a) `pArray`; 20M/Proc

(b) `pList`; 20M/Proc

(c) `pMatrix`; 20M/Proc

(d) `pGraph`; 1500x1500 stencil/Proc

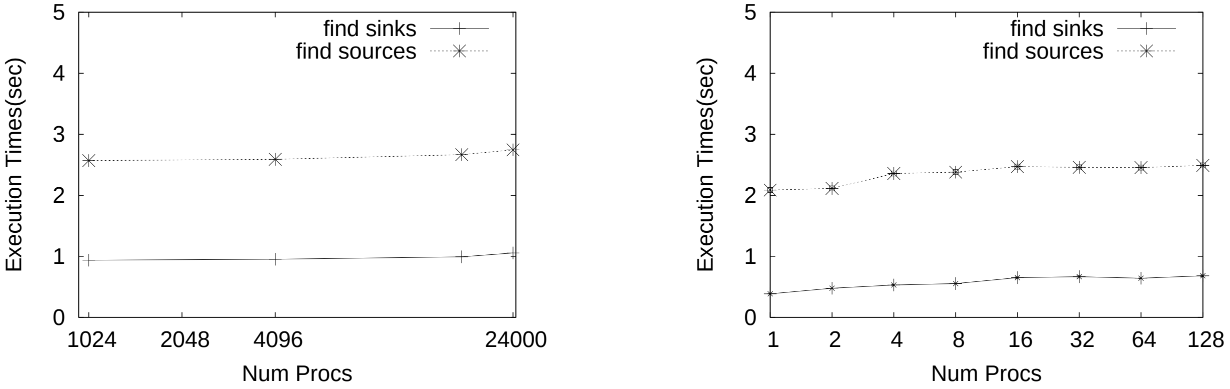
Figure 4: Execution times for `p_for_each`, `p_generate`, `p_accumulate` algorithms on CRAY for different data structures. (a) `pArray` with 20M elements per processor, (b) `pList` with 20M elements per processor, (c) `pMatrix` with 20M elements per processor, and (d) `pGraph` with 2.25M elements (1500x1500 stencil) per processor.

to the higher access time in a `pList` relative to the static `pContainers`, behavior that is inherited from the STL containers used as `bContainers`. In Figure 4(d) we show the scalability for `p_for_each` and `p_accumulate` when the algorithms use views defined over a linearization of the vertices in a `pGraph`. The weak scaling experiment is for a 1500×1500 torus stencil per processor and for two different graph partitions, a static graph partition that optimizes the storage (e.g., `std::valarray`) but doesn't allow adding or deleting vertices after creation, and a dynamic graph partition that employs a slower access time storage (e.g., `std::set`) that allows for addition and deletion of vertices. The static partition outperforms the dynamic partition by almost a factor of three due to its more efficient traversal of elements in the `bContainer`.

Figure 7 shows the execution times of two `pGraph` `pAlgorithms`. One is `find_sinks`, which scans through the vertices and identifies those with no outgoing edges. The other is `find_sources` which works in two phases: the first scans all the vertices and marks the target of their outgoing edges, and the second scans the vertices and outputs the ones that have not been marked in the first phase. We run the two algorithms on a 2D torus of size 1500 × 1500 per processor. Figure 7(a) shows the execution times on CRAY up to 24,000 processors and Figure 7(b) shows the execution times on P5-CLUSTER when scaling from 1 to 128 processors. The two algorithms considered scale well as most of the computation is local with minimal communication for boundary (only 0.03% remote edges).

We next examine the performance of the methods of `pGraph`. This weak scaling experiment is carried out by building a 2D torus where each processor holds a stencil of 1500×1500 vertices. In a first phase the vertices of the graph are added using the `add_vertex` method, then the edges are added using the `add_edge` method. Figure 8(a) shows the execution time of the two phases on CRAY, and Figure 8(b) on P5-CLUSTER. The methods used are *asynchronous*, which return the control to the caller before their completion, thus allowing for overlapping of communication and computation. It also provides opportunities for the runtime system to aggregate the messages for a better use of the bandwidth. As seen in the plots, this computation scales well for both architectures considered.

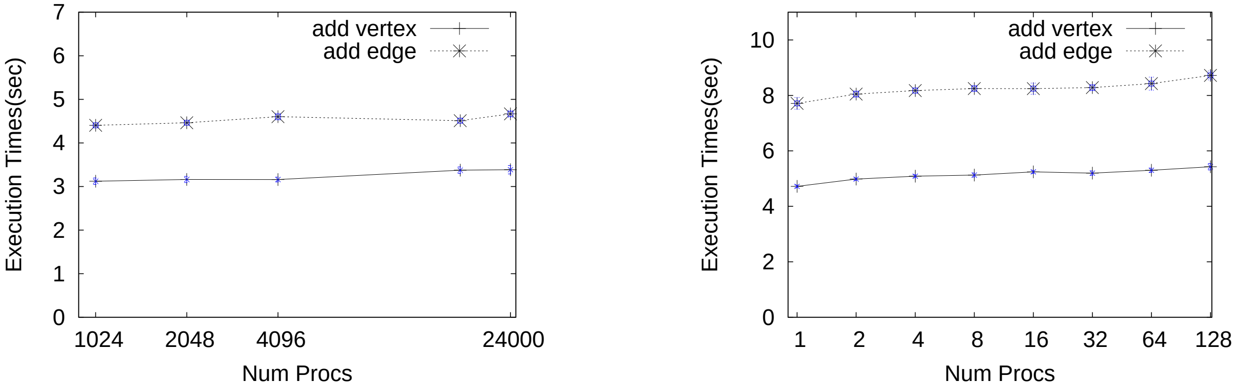
In Figure 6 we examine the performance of the `pList` methods `insert`, `insert_async`, `push_anywhere`, and `push_anywhere_async` on two systems, CRAY and P5-CLUSTER. In addition to the asynchronous methods provided for all `pContainers`, the `pList` has some additional methods to support efficient parallel processing. In particular, the `push_anywhere` and the `push_anywhere_async` methods are provided as alternatives to the STL `push_back` operation. The `push_back`, which adds an element to the end of the list,



(a)

(b)

Figure 5: Execution times for `pGraph` algorithms `find_sinks` and `find_sources` on (a) CRAY and (b) P5-CLUSTER.



(a)

(b)

Figure 6: Execution times for `pGraph` methods `add_vertex` and `add_edge` on (a) CRAY and (b) P5-CLUSTER.

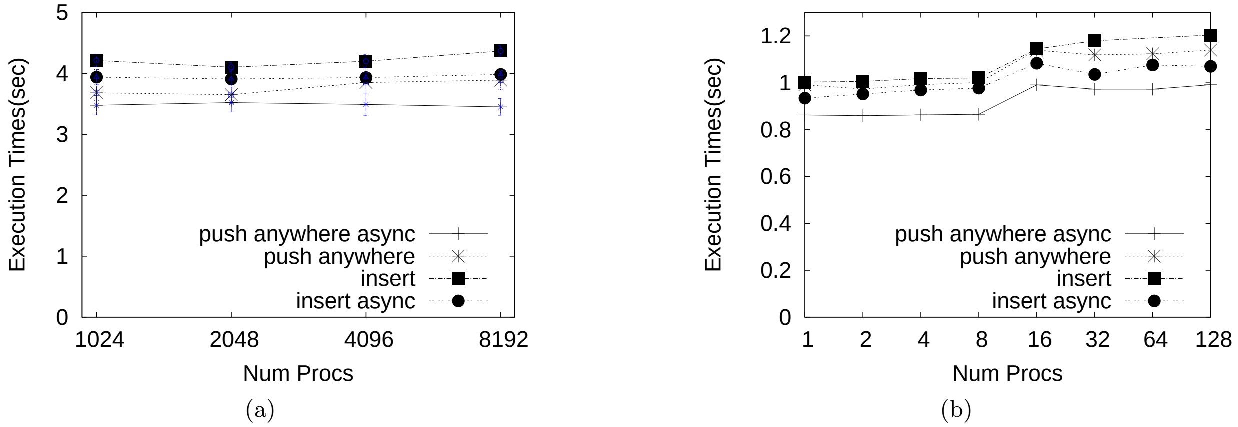
is a serializing operation. In contrast, the `push_anywhere` methods offer no guarantees as to where the element will be added, and hence allow STAPL to select efficient options, such as local insertions. In Figure 6(a) we provide a weak scaling experiment on CRAY showing good scalability for `pList` methods `insert`, `insert_async`, `push_anywhere`, and `push_anywhere_async` up to 8192 processors. Similar results are observed on P5-CLUSTER as shown in Figure 6(b), where we register a performance drop when more than 8 processors per node are used. Our experiment distributes the work on a node in a round robin manner across all sockets (dual cores) and then across all cores of the node. When both cores on each chip is utilized, the effective bandwidth per core is reduced leading to a significant performance degradation. This effect can be observed in all our P5-CLUSTER experiments.

Finally, we show an example of use of `pContainer` composition. For this comparative performance evaluation we will compute the minimum element in each row of a matrix using two data representations: the `pMatrix` `pContainer` and the composed `pArray` of `pArrays`. The generic algorithm code is the same for the two cases, due to the access abstraction mechanism provided by STAPL `views`. It calls a parallel for-all on each row, and within each row, a `map-reduce` to compute the minimum value. We measure also the time to create and initialize the storage. The `pMatrix` allocates the entire structure in a single step, while the `pArray` of `pArrays` allocates the outer structure first and then allocates the single `pArray` elements individually. In Figure 9, we show for CRAY 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 9(a) shows the case of a $P \times 100M$ element matrix (P is the number of processors), while Figure 9(b) shows the case of a $100 \cdot P \times 1M$ element matrix. The aggregated input sizes are overall the same.

As is expected, the `pArray` of `pArrays` initialization time is higher than that for a `pMatrix`. The time for executing the algorithm, however, is the same for the two data structures and scales well to 8192 processors. While we cannot state with certainty that our PCF allows for efficient composition (no additional overhead) for any pair of `pContainers`, the presented experiment indicates it is possible.

8. CONCLUSION

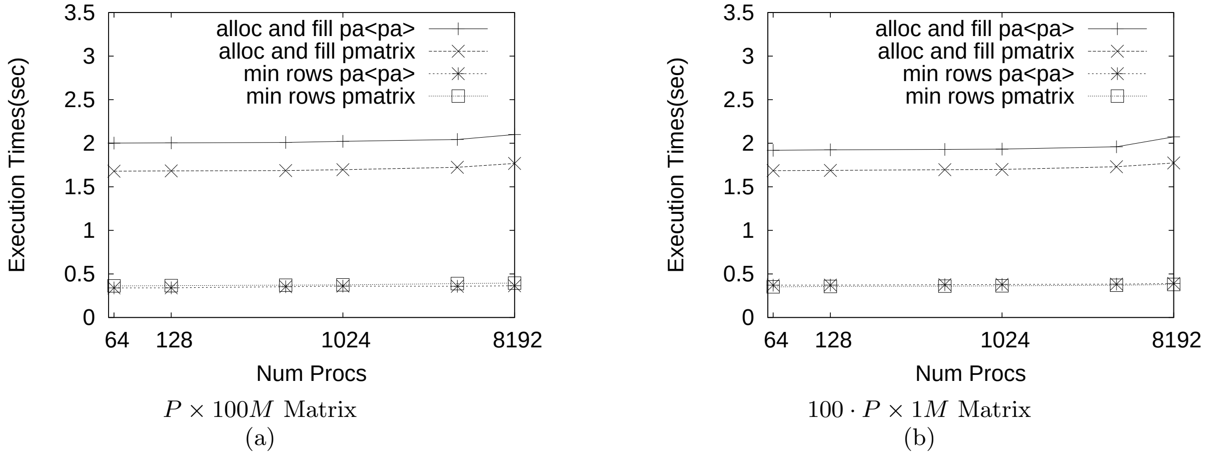
In this paper we described STAPL, the Standard Template Adaptive Parallel Library. The library provides an infrastructure for parallel programming in C++ with similar interfaces as the STL, the ISO C++ standard library. The objective of the library is to allow productive parallel ap-



(a)

(b)

Figure 7: Weak scaling study for `pList` methods on (a) CRAY using 25M method invocations per processor and (b) P5-CLUSTER using 5M method invocations per processor.



$P \times 100M$ Matrix
(a)

$100 \cdot P \times 1M$ Matrix
(b)

Figure 8: 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 similar.

plication development. By productivity we mean ease of programming and portability of performance. We described the major components of the library, with specific attention to the `pContainers` and `pAlgorithms`. We also described the STAPL runtime system, which provides the communication library and the execution mechanism of STAPL parallel algorithms. Our results show that the performance of `pAlgorithms` and `pContainers` is scalable up to tens of thousand of processors and that the performance is portable to different architectures.

9. 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. W-shop on Advanced Compiler Technology for High Performance and Embedded Processors*, Bucharest, Romania, 2001.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. M. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Int. W-shop on Languages and Compilers for Parallel Computing, LNCS*, Cumberland Falls, KY, 2001.

[3] G. Blelloch. NESL: A Nested Data-Parallel Language. Tech. Rep. CMU-CS-93-129, Carnegie Mellon Univ., 1993.

[4] G. E. Blelloch, *et. al.* Implementation of a portable nested data-parallel language. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 102–111, 1993.

[5] D. Bonachea. Gasnet specification, v1.1. Tech. Rep. UCB/CSD-02-1207, U.C. Berkeley, 2002.

[6] 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. W-shop on Languages and Compilers for Parallel Computing*. In LNCS, vol. 5335, pp. 304–315, Edmonton, Alberta, Canada, 2008.

[7] D. Callahan, Chamberlain, B.L., and H. Zima. The Cascade high productivity language. In *The 9-th Int.*

W-shop on High-Level Parallel Programming Models and Supportive Environments, vol. 26, pp. 52–60, Los Alamitos, CA, 2004.

[8] P. Charles, *et. al.* X10: an object-oriented approach to non-uniform cluster computing. In *Proc. of the 20th ACM SIGPLAN OOPSLA*, New York, NY, 2005.

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

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

[11] I. Foster, C. Kesselman, and S. Tuecke. The NEXUS task-parallel runtime system. In *Proc. 1st Intl W-shop on Parallel Processing*, pp. 457–462, 1994.

[12] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1998.

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

[14] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Prog. Lang. Sys.*, 15(5):745–770, 1993.

[15] J. Reinders. Intel Thread Building Blocks. O'Reilly Media, 2007.

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

[17] E. Johnson. *Support for Parallel Generic Programming.* PhD thesis, Indiana University, Indianapolis, IN, 1998.

[18] L. Kal'e, *et. al.* Converse : An interoperable framework for parallel programming. In *Proc. of the 10th Int. Parallel Processing Symp.*, 1996.

[19] L. V. Kale and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. *SIGPLAN Not.*, 28(10):91–108, 1993.

[20] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. of the 14-th ACM Symp. on Parallel Algorithms and Architectures*, pp. 73–82, Winnipeg, Manitoba, Canada, 2002.

[21] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide, 2nd Edition.* Addison-Wesley, 2001.

[22] J. Nieplocha and B. Carpenter. ARMCI: A portable remote memory copy libray for ditributed array libraries and compiler run-time systems. In *Proc. of the 11 IPPS/SPDP'99 W-shops in conjunction with the 13th Int. Parallel Processing Symp.*, 1999.

[23] W. Pugh. Concurrent maintenance of skip lists. Tech. Report UMIACS-TR-90-80, University of Maryland at College Park, College Park, MD, USA, 1990.

[24] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Parallel Library. In *Proc. of the 4th Int. W-shop on Languages, Compilers and Run-Time Systems for Scalable Computers*, LNCS, Pittsburgh, PA, May 1998.

[25] J. V. W. Reynders, *et. al.*. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming in C++*, chapter 14, MIT Press, 1996.

[26] S. Saunders and L. Rauchwerger. ARMI: an adaptive, platform independent communication library. In *Proc. of the 9-th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 230–241, San Diego, California, USA, 2003. ACM.

[27] G. Tanase, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pArray. In *Proc. of the 2007 W-shop on Memory Performance (MEDEA)*, pp. 73–80, Brasov, Romania, 2007.

[28] G. Tanase, C. Raman, M. Bianco, N. M. Amato, and L. Rauchwerger. Associative parallel containers in STAPL. In *Int. W-shop on Languages and Compilers for Parallel Computing*, in LNCS, vol. 5234, pp. 156–171, Urbana-Champaign, IL, 2008.

[29] 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. W-shop on Languages and Compilers for Parallel Computing*, in LNCS, Newark, DE, 2009.

[30] 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 10-th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, 2005.

[31] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proc. ACM Symp. on Princ. of Dist. Proc.*, pp. 214–222, New York, NY, 1995.

[32] K. Yelick, *et. al.* . Titanium: A high-performance Java dialect. In *ACM W-shop on Java for High Performance Network Computing* , New York, 1998.