The STAPL pView*

Antal Buss, Adam Fidel, Harshvardhan, Timmie Smith,
Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco,
Nancy M. Amato and Lawrence Rauchwerger

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

Abstract. The Standard Template Adaptive Parallel Library (STAPL) is a C++ parallel programming library that provides a collection of distributed data structures (pContainers) and parallel algorithms (pAlgorithms) and a generic methodology for extending them to provide customized functionality. STAPL algorithms are written in terms of `pViews`, which provide a generic access interface to pContainer data by abstracting common data structure concepts. Briefly, `pViews` allow the same pContainer to present multiple interfaces, e.g., enabling the same pMatrix to be 'viewed' (or used) as a row-major or column-major matrix, or even as a vector. In this paper, we describe the STAPL `pView` concept and its properties. `pViews` generalize the iterator concept and enable parallelism by providing random access to, and an ADT for, collections of elements. We illustrate how `pViews` provide support for managing the tradeoff between expressivity and performance and examine the performance overhead incurred when using `pViews`.

1 Introduction

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

* 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. Tanase is now a Research Staff Member at IBM T.J. Watson Research Center. Bianco is now a scientist at the Swiss National Supercomputing Centre.

algorithms requiring write operations cannot work on input iterators and lists do not support random access iterators. The major capability provided by the iterator is a mechanism to traverse the data of a container.

The Standard Template Adaptive Parallel Library (STAPL) [3] provides building blocks for writing parallel programs – parallel algorithms (`pAlgorithm`s), parallel and distributed containers (`pContainers`), and `pViews` to abstract data accesses to `pContainers`. `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. The STAPL `pView` generalizes the iterator concept by providing an abstract data type (ADT) for the data it represents. While an iterator corresponds to a single element, a `pView` corresponds to a collection of elements. Also, while an iterator primarily provides a traversal mechanism, `pViews` provide a variety of operations as defined by the ADT. For example, all STAPL `pViews` support size() operations that provide the number of elements represented by the `pView`. A STAPL `pView` can provide operations that return new `pViews`. For example, a `pMatrix` supports access to rows, columns, and blocks of its elements through row, column and blocked `pViews`, respectively.

`pViews` are designed to enable parallelism. In particular, `pViews` provide random access to partitioned collections of elements of each ADT supported by STAPL. This characteristic is essential for the scalability of STAPL programs. The size of these collections can be dynamically controlled and typically depends on the desired degree of parallelism. For example, the `pList pView` provides concurrent access to segments of the list, where the number of segments could be set to match the number of parallel processes. The `pView` provides random access to a partitioned data space. To mitigate the potential loss of locality incurred by the flexibility of the random access capability, `pViews` provide, to the degree possible, a remapping mechanism of a user specified `pView` to the container's physical distribution (known as the native `pView`).

In this paper, we describe the STAPL `pView` concept and its properties. As outlined above, critical aspects of the `pView` are:

- STAPL `pViews` generalize the iterator concept — a `pView` corresponds to a collection of elements and provides an ADT for the data it represents.
- STAPL `pViews` enable parallelism — this is done by providing random access to the elements, and support for managing the tradeoff between the expressivity of the views and the performance of the parallel execution.

2 STAPL Overview

STAPL [3, 20, 15, 16] is a framework for parallel C++ code development (Figure 1). Its core is a library of parallel algorithms (`pAlgorithms`) and distributed data structures (`pContainers`)[18] that have interfaces similar to the (sequential) C++ standard library (STL) [12]. Analogous to STL algorithms that use *iterators*, STAPL `pAlgorithm`s are written in terms of `pViews` so that the same algorithm can operate on multiple `pContainers`.

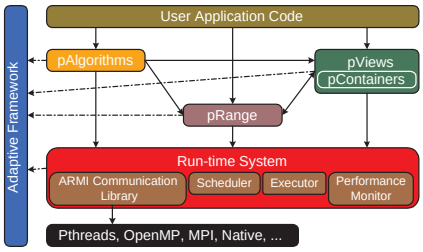


Fig. 1. STAPL Overview

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

STAPL `pContainers` are distributed, thread-safe, concurrent objects, i.e., shared objects that provide parallel methods that can be invoked concurrently. They are composable and extendible via inheritance. Currently, STAPL provides counterparts of all STL containers (e.g., `pArray`, `pVector`, `pList`, `pMap`, etc.), and two `pContainers` that do not have STL equivalents: parallel matrix (`pMatrix`) and parallel graph (`pGraph`). `pContainers` include a set of `bContainers`, that are the basic storage components for the elements, and distribution information that manages the distribution of the elements across the parallel machine.

`pContainers` provide methods corresponding to the STL container methods, and some additional methods specifically designed for parallel use. For example, STAPL provides an `insert_async` method that can return control to the caller before its execution completes, or an `insert_anywhere` that does not specify where an element is going to be inserted and is executed asynchronously. 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 supported by assigning each `pContainer` element a unique global identifier (GID) and by providing each `pContainer` an internal translation mechanism that can locate, transparently, both local and remote elements. The physical distribution of `pContainer` data can be determined automatically by STAPL or it can be user-specified.

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. ARMI uses the remote method invocation (RMI) communication abstraction to hide the lower level implementation (e.g., MPI, OpenMP, etc.). 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).

3 Related Work

The view concept has been used in different areas.

One of first uses of the view concept was in database systems. In particular, views can be defined by a database query and used to represent a virtual table in a relational database or an entity in an object oriented database. Generally, database views are read-only. However, views can be updatable (writable) if the database supports reverse mappings from a view to the database. Some systems implement updatable views using an "instead of" trigger that is executed when an insert, delete or update over the view is executed. A similar approach is presented in [1] where lenses implement bidirectional transformations.

GIL (Generic Image Library) [2, 8] is a C++ image library from Adobe for image manipulation. It provides the concept of an image view, which generalizes STL's range concept [13] to multiple dimensions. GIL's image views are specialized for operating on two-dimensional images which may have different storage distributions in memory, but are always in the same address space.

The VTL (View Template Library) [21] project worked with views as an adaptor layer on top of STL. This project, which has been inactive since 2000, was heavily inspired by the Views library of Jon Seymour [17]. A VTL view is a container adaptor, that provides a container interface to access a portion of the data, to rearrange the data, to transform data, or to combine data. The STAPL `pView` provides similar capabilities for `pContainers`.

The view concept has been used in some PGAS (Partitioned Global Address Space) languages. X10 [6] provides the notion of a region to specify a section of data. Chapel [5] provides the user a global view over a container, specifies subarrays with domains, and uses iterators [10] as abstractions for algorithms. STAPL `pViews` provide a container interface in addition to support for iterators.

The Hierarchically Tiled Array (HTA) data type [7] provides a rich interface to specify array views. It also implements advanced support for handling the boundary communication of common patterns arising in scientific computing. STAPL overlap `pViews` are similar to HTA overlapped tiling, though STAPL supports arbitrary, static and dynamic data types.

4 STAPL `pView` Concept

In this section, we introduce the `pView` concept and explain how it can be exploited in the parallel and distributed environment of STAPL.

A `pView` is a class that defines an abstract data type (ADT) for the *collection of elements* it represents. As an ADT, a `pView` provides *operations* to be performed on the collection, such as read, write, insert, and delete.

`pViews` have *reference semantics*, meaning that a `pView` does not own the actual elements of the collection but simply *references* to them. The collection is typically stored in a `pContainer` to which the `pView` refers; this allows a `pView` to be a relatively light weight object as compared to a container. However, the collection could also be another `pView`, or an arbitrary object that provides a

container interface. With this flexibility, the user can define `pViews` over `pViews`, and also `pViews` that generate values dynamically, read them from a file, etc.

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

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

$$\mathcal{V} \stackrel{def}{=} (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}) \tag{1}$$

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

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

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

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

Composition of views. Since a `pView` and the collection it represents can be used interchangeably, the `pView` definition (Equation 1) naturally enables *composition*, i.e., `pViews` defined over other `pViews`. Figure 2(a) shows the construction of `pViews` over other `pViews`, and the possibility of having multiple `pViews`

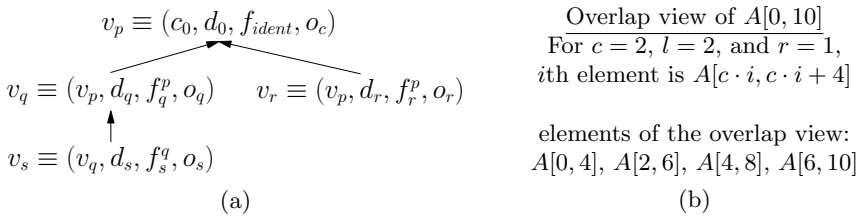


Fig. 2. (a) Example of construction of `pViews` over the same collection. v_p is a `pView` over the collection c_0. v_q and v_r are two, possibly different, `pViews` over v_p, and v_s is a `pView` over v_q. (b) Example overlap `pView` for $A[0, 10]$.

concurrently referencing the same container. Thus, composition makes possible the representation of complex data organizations and naturally supports the recursive partitioning of domains.

4.1 Useful views

There are several types of `pViews` worthy of note because they enable optimizations or are useful in expressing computations.

By providing certain operations and not others, `pViews` can be classified as *read-only* or *write-only*. This is analogous to the STL input and output iterators.

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

Other important `pViews` include the *balanced* `pView` where the data is split into a specified number of approximately equal-sized chunks, and the *native* `pView`, which provides `bViews` that are *aligned* with the `pContainer` distribution. These turn out to be very useful in the context of STAPL.

Transform `pViews` apply a user specified function to the elements returned from the collection. This feature can be used to change the value, type, or both, of the returned element. Important aspects of the transform `pView` are that the elements in the underlying collection are not modified and the result is computed and made available only when an element accessed through the `pView` is actually referenced in the program. In contrast, for example, a `for_each` algorithm applied to a `pContainer`, would traverse and modify all elements of the container within the relevant range.

There are also a number of useful views that have more complex elements. One example is a *zip* `pView`, which takes two (or more) collections and provides a `pView` where each element is a pair (or tuple) including an element from each collection. Zip views are useful for expressing algorithms that operate on multiple collections. Another `pView` heavily used in STAPL is the *overlap* `pView`, in which one element of the `pView` overlaps another element of the view. This `pView` is naturally suited for specifying many algorithms, such as adjacent differences,

	read	write	[ ]	begin end	insert erase	insert any
array_1d_pview	✔	✔	✔	✔		
array_1d_ro_pview	✔		✔	✔		
static_list_pview	✔			✔		
list_view	✔	✔		✔	✔	✔
matrix_pview	✔	✔	✔			
graph_pview	✔	✔			✔	✔
strided_1D_pview	✔	✔	✔	✔		
transform_pview	✔		-	-		
balanced_pview	✔		✔	✔		
overlap_pview	✔		✔	✔		
native_pview	✔		✔	✔		
repeated_pview	✔		✔	✔		

Table 1. Major views implemented in STAPL and corresponding operations. `tranform_view` implements an overridden read operation that returns the value produced by a user specified function, the other operations depends on the `pView` the transform `pView` is applied to. `insert_any` refers to the special operations provided by STAPL `pContainers` that insert elements in unspecified positions.

string matching, etc. The *repeated* `pView` is a special case of an overlapped `pView` in which each element includes the entire collection. As an example, we can define an overlap `pView` for a one-dimensional array $A[0, n-1]$ using three parameters, c (core size), l (left overlap), and r (right overlap), so that the ith element of the overlap `pView` $v^o[i]$ is $A[c \cdot i, c \cdot i + l + c + r - 1]$. See example in Figure 2(b).

5 The `pView` class

The `pView` is an object that builds on the STAPL `pContainer` framework. To create a `pView`, the user specifies the partitioned collection (often a `pContainer`), the partitioned domain $\mathcal{D}$, and the mapping functions $\mathcal{F}$, as (template) arguments of the `pView` class, while the operations $\mathcal{O}$ must be implemented by the class itself. All STAPL `pViews` are derived from the `core_view` templated base class. This class provides constructors, and stores references to $\mathcal{C}$, $\mathcal{D}$, and $\mathcal{F}$.

To ease the implementation of the basic operations, and thus the implementation of the generic `pView` concepts needed by STAPL algorithms, the user can derive the `pView` class from classes implementing those operations, e.g., a `pContainer`. Usually, the `pView` can directly invoke the `pContainer` methods. An exception is the transform `pView`, where the read operation is implemented as `return F(container.operation(f(i), ...))` and `F` is the transformation function, and `f` is the mapping function.

`pViews` *in* STAPL. Table 1 shows a list of some `pViews` available in STAPL. These `pViews` are implemented using the schema discussed above, and new `pViews` can

be implemented and created in the same way. The *native* `pView` is a `pView` whose partitioned domain $\mathcal{D}$ matches the data partition of the underlying `pContainer`, allowing data references to it to be local. The *balanced* `pView` partitions the data set into a user specified number of pieces. The sizes of the pieces differs by at most by one. This `pView` can be used to balance the amount of work in a parallel computation. If STAPL algorithms can use balanced or native `pViews`, then performance is greatly enhanced.

Optimizations. There are trade-offs between the expressivity offered by the `pViews` and performance. For this reason, the `pViews` are designed to allow the implementation of different optimizations to improve the performance of data access. Below, we present a few such examples.

The repeated composition of `pViews`, an important technique to develop new `pViews`, can result in an increasing chain of indirect data references due to the repeated composition of the mapping functions ($\mathcal{F}$s). In certain cases, such as when where $\mathcal{F}$ is statically known, and has a relatively simple closed form expression, STAPL can reduce the chain of indirections to one. For instance, composing identity functions results in another identity function, while composing an arbitrary function $\mathcal{F}$ with an identity function is the same $\mathcal{F}$.

Another important optimization is localization of memory references. STAPL `pViews` can determine which sections of consecutive references are local (within the same address space). This allows the `pView` to use a much simpler, and thus much faster mechanism to reference local data.

6 Results: Expressivity, Genericity, and Performance

In this section, we present experimental results to study the trade-offs between the enhanced expressivity enabled by `pViews` and their performance. For this purpose, we compare the performance of functionally equivalent STAPL programs written using `pViews` and C++ MPI programs.

We conducted our experimental studies on two architectures: an 832 processor IBM cluster with p575 SMP nodes (16 cores per node) available at Texas A&M University (called P5-CLUSTER) and a 38,288 processors Cray XT4 with quad core Opteron processors (4 cores per node) available at NERSC (called CRAY4-CLUSTER). The compiler used for the experiments was gcc (version 4.3.1 on P5-CLUSTER and version 4.4.1 on CRAY4-CLUSTER) with the `-O3` optimization flag. In all experiments, a location contains a single processor, and the terms can be used interchangeably.

Genericity. We can solve many problems using the `stapl::count_if(view, pred)` algorithm which takes an `array_1D_view` and counts how many times the referenced elements satisfy a user provided predicate `pred`.

For instance, we can compute π using the well known Monte Carlo method [14]. Random points are generated inside the unit square and we count how many of these fall inside the unit circle. The ratio between these and the total number of points generated is $\pi/4$. The `pView` used to represent the input does

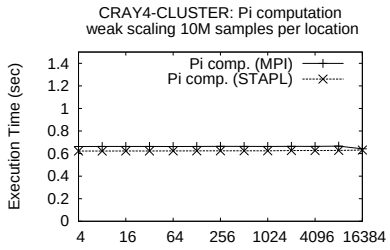
```
struct in_circle {
 template<typename Reference>
 bool operator()(Reference ref) const {
   std::pair<double, double> val = ref;
   return sqrt(pow(val.first,2.0) +
        pow(val.second,2.0)) < 1.0;
 }
}

void stapl_main(int argc, char** argv){
  typedef array_1D_ro_view
     <rand_gen_cont> rand_gen_view;

  rand_gen_cont rgc(N);
  rand_gen_view rgenv(rgc);

  int result =
      stapl::count_if(rgenv,in_circle());
  double pi = 4*(result/(double)N);
}
```

(a)



(c)

```
MPI_Init(&argc, &argv);
int P;

MPI_Comm_size(MPI_COMM_WORLD, &P);

N=N/P;

int pid;
MPI_Comm_rank(MPI_COMM_WORLD,&pid);

int cnt(0);
for (int i=0; i<N; ++i) {
  double xcoord = drand48();
  double ycoord = drand48();

  double dist =
          sqrt(pow(xcoord, 2.0)
          + pow(ycoord, 2.0));

  if (dist < 1.0)
     ++cnt;
}

int res;
MPI_Reduce(&cnt,&res,1,MPI_INT,MPI_SUM,0,
            MPI_COMM_WORLD);
MPI_Finalize();

double pi = 4*(res/(double)(N*P));
```

(b)

Fig. 3. Computing π. (a) STAPL code using a view over a generator container. (b) MPI version. (c) Execution times on CRAY4-CLUSTER.

not need a reference to storage because the points can be generated on demand. Hence, the container provided to the `pView` is a simple class that exports the container interface and whose read method returns a randomly generated point in the unit square. Passing this `pView` to `stapl::count_if`, with a predicate to check if the point lies within the unit circle, will execute the π computation. We also evaluated an equivalent C++ MPI program for computing π. The code snippets are shown in Figures 3(a) and 3(b), respectively. Note that the two programs are comparable in terms of complexity for this embarrassingly parallel algorithm. Figure 3(c) shows that the performance for the two implementations is comparable, with the STAPL program slightly outperforming the MPI version.

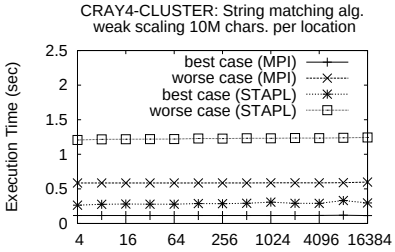
String matching can also be implemented by calling `stapl::count_if(view, pred)` with an appropriate `pView` and predicate. In this case, given a pattern of length M, we create an overlap `pView` over the text, with a core of length 1, left overlap of size 0 and right overlap of size $M-1$. This will give a `pView` over all the sub-strings of size M of the input text. The code sample is shown in Figure 4(a). In Figure 4(b), an MPI version of the program is shown. In this case it becomes possible to appreciate the additional complexity of the MPI

```
struct strmatch {
  const string& S;
  strmatch(const string& s): S(s) {}

  template<typename View>
  bool operator()(View v) const {
    return equal(S.begin(),S.end(),
                 v.begin());
  }
};

void stapl_main(int argc, char** argv) {
  typedef stapl::p_array<char>
      p_string_type;
  typedef stapl::array_1D_view
      <p_string_type> pstringView;
  ...
  result=stapl::count_if(
           stapl::overlap_view(text,
           1,0,pattern.size()-1),
           strmatch(pattern));
  ...
}
```

(a)



(c)

```
int main(int argc, char** argv) {
  ...
  MPI_Comm_size(MPI_COMM_WORLD, &P);

  N=N/P;
  std::vector<char> V(N);
  int M=S.length();

  for (int i=0; i <= N-M+1; ++i)
    if (equal(S.begin(), S.end(),
              V.begin()+i)) ++cnt;
  if (pid>0)
    MPI_Send((&V[0]), M-1, MPI_CHAR,
          pid-1, 1, MPI_COMM_WORLD);

  if (pid<P-1) {
    vector<char> BUFF(2*(M-1));
    copy(V.begin()+N-M+1, V.end(),
              BUFF.begin());

    MPI_Recv( &BUFF[M-1], M-1, MPI_CHAR,
              pid+1, 1, MPI_COMM_WORLD,
              &status );

    for (int i=0; i <= M-1; ++i)
      if (equal(S.begin(), S.end(),
            BUFF.begin()+i ))
        ++cnt;
  }

  int res;
  MPI_Reduce ( &cnt, &res, 1, MPI_INT,
               MPI_SUM, 0, MPI_COMM_WORLD );
  ...
}
```

(b)

Fig. 4. String matching. (a) STAPL code using an overlap partitioned view. (b) MPI version. (c) Execution times on CRAY4-CLUSTER.

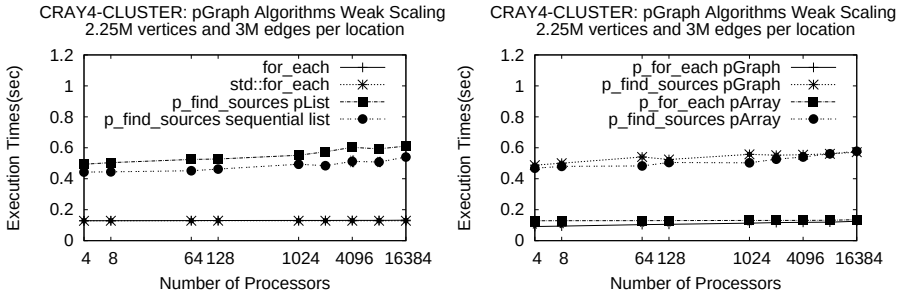
code with respect to the STAPL version, since in MPI the programmer must take explicit care of the boundary regions (this is a special case of the use of ghost nodes, a well known technique in parallel processing [11, 7]). Figure 4(c) shows that performance of the two versions is comparable. In the best case, the first character of the substring is not in the text and the number of occurrences is zero. In the worse case, both text and substring are composed of the same character, maximizing the number of occurrences.

Graph views. STAPL provides the `pGraph`, a parallel and distributed graph data structure. Algorithms operating on `pGraphs` are written generically in terms of graph `pView` concepts. In this section, we describe `pGraph` specific `pViews` and discuss the performance of generic algorithms using them.

For simple operations such as initializing the data of each vertex or edge, we can use a view over the set of vetrices and edges. These views implement the `static_list` concept and support efficient parallel map and map_reduce operations. In Figure 5, we show a STAPL program that performs an initialization of

```
1  stapl::p_graph<vertex_property>                          graph;
2  stapl::graph_view<stapl::p_graph<vertex_property> >  graph_view(graph);
3  stapl::p_list<vertex>                                    list;
4  stapl::list_view<stapl::p_list<vertex> >             result_view(list);
5  stapl::for_each(graph_view.vertices(), init_property());
6  stapl::p_find_sources(graph_view.edges(), result_view);
```

Fig. 5. Find sources and sinks in a graph



(a) (b)

Fig. 6. Weak scaling on CRAY4-CLUSTER of `pGraph` methods with 2.25M vertices and ∼3M edges per location. (a) `pView` overhead: comparison of calling `stapl::for_each` versus STL `for_each`, and of storing sources in a `pList` versus a sequential STL list; (b) Comparison of graph algorithms on graph views defined over `pGraph` and `pArray`.

the `pGraph` vertex properties (Figure 5, Line 5), and then computes and stores the set of source vertices in a parallel list (Figure 5, Line 6). The `list_view` (Figure 5, Line 4) defined over a parallel list [19] supports an interface to efficiently insert and erase elements concurrently. The `stapl::find_sources` algorithm uses the `insert_anywhere` method of the `list_view` to populate the parallel list with source vertices. To evaluate the algorithms we perform a weak scaling experiment using a 2D sparse mesh as input, where each processor holds a stencil of 1500×1500 vertices. The number of edges per location is on average two thirds the maximum number of edges in a 2D mesh while the number of remote edges is ∼1500 (0.3%) per location.

Figure 6 shows the performance of the two algorithms on the CRAY4-CLUSTER. `stapl::for_each` is a simple do-all operation that applies a functor to every element of a view. It scales well when the number of processors is varied from 4 to 16384. `stapl::find_sources` performs a `stapl::for_each` on a `pView` over the edges of the graph, marking their targets as non source vertices. To evaluate the overhead of using views and STAPL containers, we performed the following experiments: first we compared the performance of the `stapl::for_each` using a `vertex_set_view` versus a simple STL `for_each` applied to individual elements stored inside the `pGraph`'s `bContainers`. We observe in Figure 6(a) that `stapl::for_each` has no overhead relative to the STL `for_each`. A second experiment performed was to evaluate the overhead of storing the source vertices in a `pList` through a `list_view` versus storing the vertices directly in sequential

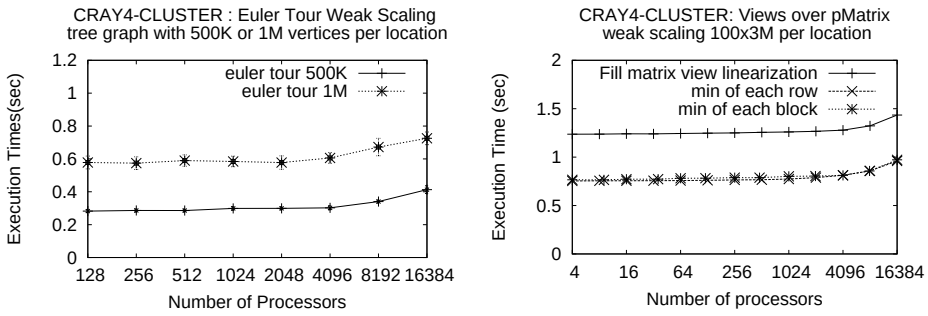


Fig. 7. Weak scaling experiments on CRAY4-CLUSTER. (a) Euler Tour computation on binary tree with 500K or 1M subtrees per processor. (b) `pMatrix` fill with random values using a linearization view and computing the minimum of each row or block.

STL lists, one for each location considered. As we can see from Figure 6(a), the `pList` incurs an overhead of only 4%.

Another important feature of using views is that new interfaces can be defined on top of existing data structures. For example, a graph view can be defined for a `pArray` of lists of edges. Generic parallel graph algorithms in STAPL will operate properly on the data stored in a `pArray`, provided a suitable graph `pView` is implemented. In Figure 6(b) we show the performance of `stapl::for_each` and `stapl::find_sources` when accessing data using a graph view defined on a `pGraph` and a graph view defined on a `pArray`. We observe that both views provide good scaling. When data is stored in the `pArray`, `stapl::for_each` is slightly faster. On the `stapl::find_sources` algorithm there is additional overhead because `stapl::find_sources` uses additional graph methods (e.g., `find_vertex`) that are more efficiently implemented in the native `pGraph`.

The Euler Tour (ET) is an important `pView` of a graph for parallel processing. In particular, the ET, which traverses every edge of the graph exactly once, corresponds to an edge view of the graph. Since the ET represents a depth-first-search traversal, when it is applied to a tree it can be used to compute a number of tree functions such as rooting a tree, postorder numbering, computing the vertex level, and computing the number of descendants [9]. The parallel Euler Tour algorithm [9] implemented in STAPL uses a STAPL `pGraph` to represent the tree and a `pList` to store the final Euler Tour. The algorithm executes parallel traversals on the `pGraph` view, generating Euler Tour segments that are stored in a temporary `pList`. Then, the segments are linked together to form the final `pList` containing the Euler Tour. The performance is evaluated by performing a weak scaling experiment on CRAY4-CLUSTER using as input a tree distributed across all locations. The tree is generated by first building a binary tree 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 for each location (one to the root and two to the children of the root in each location, with directed edges for

```
block_partition_t     blkpart(m,n);
p_matrix_t            pmat(N1,N2,blkpart);
matrix_view_t         vmat(pmat);

// Row major linearization of the p_matrix
typedef array_1D_view<p_matrix_t,
    dom1D<size_t>,
    f1d_row_major_2d<size_t,p_matrix_index_type> >      linear_row_t;
linear_row_t lrow = vmat.linear_row();
// Fill the matrix using the linearization view

// One dimensional view over the p_matrix's rows
typedef partitioned_view<matrix_view_t,
    rows_partition<matrix_domain_t,row_domain_t>,
    map_fun_gen1<fcol_2d<size_t,matrix_dom_t::index_type> >,
    matrix_view_t::row_type>                            rows_view_t;
rows_view_t       rowsv( vmat, rows_partition_t(vmat.domain()) );

// Computing the minimum of each row
stapl::transform(rowsv, resv, stapl::min_value<int>());

// One dimensional view of blocks over the p_matrix
typedef partitioned_view<matrix_view_t,
    block_partition_t,
    map_fun_gen<f_ident<mat_view_t::index_type> > >     blocks_view_t;
blocks_view_t     blocksv(vmat,blkpart);

// Computing the minimum of each block
stapl::transform(blocksv, resv, stapl::min_value<int>());
```

Fig. 8. Snippets of code used to create different types of views over `pMatrix`: row major linearization of the matrix, partition the matrix view in rows and partition the matrix view in blocks

both directions). Figure 7(a) shows the execution time on CRAY4-CLUSTER for different sizes of the tree. The running time increases with the number of vertices per location because the number of edges in the ET to be computed increases correspondingly.

Matrix views. The `pMatrix` is a `pContainer` that implements a dense, two-dimensional array [4]. We can create different types of views over a `pMatrix` to adapt the container to the algorithm requirements. For example, we can initialize the values of a container using `stapl::generate` or `stapl::copy`. Both algorithms require the data layout in a one-dimensional container. Using a mapping function to translate indices from one to two dimensions, we can define a linearization view over the `pMatrix` (e.g., `f1d_row_major_2d` in Figure 8). Similarly, we can create row and blocked `pViews` of the `pMatrix`. Figure 8 shows two of these views: a `pView` over the rows (`rows_view_t`) and a `pView` over blocks

(`blocks_view_t`). They differ in the partitioner and the mapping function generator used.

Figure 7(b) shows the execution time on CRAY4-CLUSTER of three algorithms using `pMatrix`: filling the `pMatrix` using `stapl::copy` from a generator container through the row major linearization `pView`, computing the minimum element of each row and computing the minimum element of each block using `stapl::transform(input_view,output_view,functor)` algorithm, where the `functor` finds the minimum of a sequence of elements.

7 Conclusion

In this paper we have introduced the `pView` a higher level concept that allows programmers to be more expressive. Furthermore, it is a concept that hides some of the details of parallel programming. It has been assumed that programming at higher levels of abstraction inevitably reduces performance, an unwelcome side-effect in general, and in parallel programming in particular. In this paper we have shown that, at least as far the `pView` is concerned, performance does not always have to suffer. In fact, in some cases we have shown that the `pView` offers more structural and semantic information than, for example, the STL iterator, and thus enables better performance. We believe that a programming environment such as STAPL will prove to be both expressive and productive as well as high performance.

References

1. Barbosa, D. M. J., Cretin, J., Foster, N., Greenberg, M., Pierce, B. C.: Matching Lenses: Alignment and View Update. In Proc. ACM SIGPLAN Int. Conf. on Functional Programming, Baltimore, Maryland, Sept. (September 2010).
2. Bourdev. L.: Generic Image Library. Software Developer's Journal, pp 42–52, (2007).
3. Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Tanase, G., Thomas, N., Xu, X., Bianco, M., Amato, N. M., Rauchwerger, L.: STAPL: Standard template adaptive parallel library. In Proc. Annual Haifa Experimental Systems Conference (SYSTOR), pp 1–10, New York, NY, USA, 2010. ACM.
4. Buss, A., Smith, T., Tanase, G., Thomas, N., Bianco, M., Amato, N. M., Rauchwerger, L.: Design for interoperability in STAPL: pMatrices and linear algebra algorithms. In Amaral, J. N. (eds,) LCPC 2008. LNCS, vol. 5335, pp 304–315, Springer, Heidelberg (2008).
5. Callahan, D., Chamberlain, B. L., Zima, H. P.: The Cascade High Productivity Language. In The 9th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments, vol. 26, pp 52–60, Los Alamitos, (2004).
6. Charles. P., Grothoff. C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications, pp 519–538, ACM Press. New York, NY, USA, (2005).

7. Guo, J., Bikshandi, G., Fraguela, B. B., Padua, D.: Writing Productive Stencil Codes with Overlapped Tiling. Concurr. Comput. : Pract. Exper., 21(1):25–39, (2009).
8. Adobe Inc.: Generic Image Library, http://opensource.adobe.com/wiki/display/gil/Generic\+Image\+Library.
9. JàJà, J.: An Introduction Parallel Algorithms. Addison–Wesley, Reading, Massachusetts, (1992).
10. Joyner, M., Chamberlain, B. L., Deitz, S. J.: Iterators in Chapel. (April 2006).
11. Lumsdaine, A., Gregor, D., Hendrickson, B., Berry, J. W.: Challenges in Parallel Graph Processing. Parallel Processing Letters, 17(1):5–20, (2007).
12. Musser, D., Derge, G., Saini, A.: STL Tutorial and Reference Guide, Second Edition. Addison-Wesley, (2001).
13. Ottosen, T.: Range Library Proposal. Technical report, JTC1/SC22/WG21 - The C++ Standards Committee, (2005), http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1871.html.
14. Quinn. M.: Parallel Programming in C with MPI and OpenMP. McGraw-Hill, (2003).
15. Rauchwerger, L., Arzu, F., Ouchi, K.: Standard Templates Adaptive Parallel Library. In Proc. of the 4th Int. Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR), Pittsburgh, PA, (May 1998).
16. Saunders, S.,Rauchwerger, L.: ARMI: An Adaptive, Platform Independent Communication Library. In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP), pages 230–241, San Diego, California, USA, (2003).
17. Seymour, J.: Views - a C++ Standard Template Library Extension, (January 1996). http://www.zeta.org.au/~jon/STL/views/doc/views.html.
18. Tanase, G., Buss, A., Fidel, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Thomas, N., Xu, X., Mourad, N., Vu, J., Bianco, M., Amato, N. M., Rauchwerger, L.: The STAPL Parallel Container Framework. In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP), San Antonio, Texas, USA, (2011).
19. Tanase, G., Xu, X., Buss, A., Harshvardhan, Papadopoulos, I., Pearce, O., Smith, T., Thomas, N., Bianco, M., Amato, N. M., Rauchwerger, L.: The STAPL pList. In Gao, G., Pollock, L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp 16–30, Springer, Heidelberg (2009).
20. Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N. M., Rauchwerger, L.: A framework for adaptive algorithm selection in STAPL. In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP), pp 277–288, Chicago, IL, USA, (2005).
21. Weiser, M., Powell, G.: The View Template Library. In 1st Workshop on C++ Template Programming, Erfurt, Germany, (October 2000).