The STAPL pView

Antal Buss, Adam Fidel, Harshvardhan,Timmie Smith,
Gabriel Tanase, Nathan Thomas, Xiabing Xu,Mauro Bianco,
Nancy M. Amato and Lawrence Rauchwerger
{abuss,fidel,ananvay,timmie,gabrielt,nthomas,xiabing,bmm,amato,rwerger}
@cse.tamu.edu
Technical Report TR10-001

Parasol Lab
Dept. of Computer Science and Engineering
Texas A&M University
College Staion, TX 77843-3112

July 13, 2010

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 views, which provide a generic access interface to pContainer data by abstracting common data structure concepts. Briefly, views 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 `View` concept and its properties. STAPL `Views` generalize the iterator concept — a `View` corresponds to a collection of elements and provides an ADT for the data it represents. STAPL `Views` enable parallelism 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. `Views` trade additional parallelism enabling information for reduced genericity. We illustrate the expressivity enabled by `Views` for several examples and examine the performance overhead incurred when using `Views`.

1 Introduction

Decoupling of data structures and algorithms is a common practice in generic programming. STL, the C++ Standard Template Library, obtains this by using the abstraction provided by *iterators*, which provide a generic interface for algorithms to access data which 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, 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 STAPL `View` 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 `View` corresponds to a collection of elements. Also, while an iterator primarily provides a traversal mechanism, `Views` provide a variety of operations as defined by the ADT. For example, all STAPL `Views` support size() operations that provide the number of elements represented by the `View`. Or, the `pMatrix` supports access to rows, columns, and blocks of its elements through row, column and blocked `Views`, respectively.

A primary objective of the STAPL `Views` is that they are designed to enable parallelism. In particular, each ADT supported by STAPL provides random access to collections of its elements. The size of these collections can be dynamically controlled and typically depends on the desired degree of parallelism. For example, the STAPL `pList View` provides concurrent access to segments of the list, where the number of segments could be set to match the number of parallel processes. The random access property supported by the STAPL `Views` enables the partitioning and distribution of the `Views` and the associated data. This capability is essential for the scalability of STAPL programs. To mitigate the potential loss of locality incurred by the flexibility of the random access capability, STAPL `Views` provide, to the degree possible, a remapping mechanism of a user specified `View` to the container's physical distribution (aka the native `View`).

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

- STAPL `Views` generalize the iterator concept — a `View` corresponds to a collection of elements and provides an ADT for the data it represents.

- STAPL `Views` 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.

It is important to remark that `Views` trade additional parallelism enabling information for reduced genericity.

In the remainder of this paper, we briefly present the STAPL components and then present the `View` concept and its implementation in STAPL. We then discuss the competing objectives of enhanced expressivity and parallel performance and present mechanisms provided by the `Views` to manage this trade-off.

2 STAPL Overview

STAPL [2, 5, 18, 14, 15, 1] is a framework for parallel C++ code development; see Fig. 1. Its core is a library of C++ components implementing parallel algorithms (`pAlgorithms`) and distributed data structures (`pContainers`) that have interfaces similar to the (sequential) C++ standard library (STL) [12]. Analogous to STL algorithms that use *iterators*, STAPL `pAlgorithms` are written in terms of `Views` so that the same algorithm can operate on multiple `pContainers`.

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 by users 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` are made of 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.

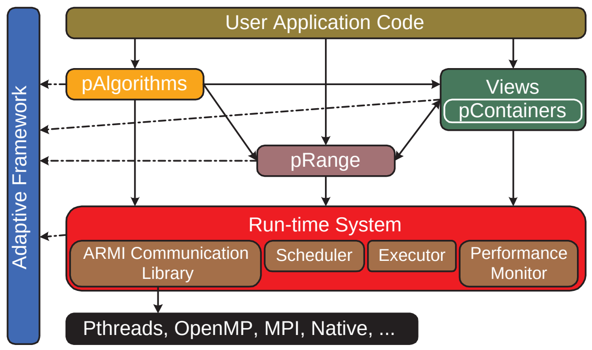


Figure 1: STAPL Overview

`pContainers` provide methods corresponding the STL containers, 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 which 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 proposed and used in a number of different areas including databases, graphics, STL, etc.

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 [3] where lenses implement bidirectional transformations.

GIL (Generic Image Library) [4, 10] 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) [19] 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 [16]. 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 `View` provides similar capabilities for `pContainers`.

View concepts have also been used in some PGAS (Partitioned Global Address Space) languages. Chapel [7] provides the user a global view over a container and uses domains to specify subarrays. X10 [8] provides the notion of a region to specify a section of the data.

The Hierarchically Tiled Array (HTA) data type [9] provides a rich interface to specify array views. It also implements advanced support for handling boundary communication for common patterns arising in scientific computing. STAPL overlap `Views` are similar to HTA overlapped tiling, though STAPL supports arbitrary, static and dynamic data types.

4 STAPL `pView` Concept

In this section, we first introduce the `View` concept and then explain how it can be generalized for the parallel and distributed environment of STAPL.

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

`Views` have *reference semantics*, meaning that a `View` 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 `View` refers; this allows a `View` to be a relatively light weight object as compared to a container. However, the collection could also be another `View`, or an arbitrary object that provides a container interface. With this flexibility, the user can define `Views` over `Views`, and also `Views` that generate values dynamically, read them from a file, etc.

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

More formally, a `View` v is a tuple

$$v \overset{def}{=} (c, d, f, o), \tag{1}$$

where c represents the underlying collection, d defines the domain of v, f represents the mapping function from v's domain to the collection's domain, and o is the set of operations provided by v.

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

Example. A common concept in generic programming is one dimensional array of size n supporting random access. The `View` corresponding to this will have an integer domain $d = [0, n)$

Overlap view of $A[0, 10]$
For $c = 2$, $l = 2$, and $r = 1$,
ith element is $A[c \cdot i, c \cdot i + 4]$

elements of the overlap view:
$A[0, 4]$, $A[2, 6]$, $A[4, 8]$, $A[6, 10]$

(a)

$$\begin{tikzcd}
 & c \equiv (S, D, \mathcal{O}_c) & \\
 & v_p \equiv (c, Dom(c), f_{ident}, \mathcal{O}_c) \arrow[u] & \\
v_q \equiv (v_p, D_q, f_q^p, \mathcal{O}_q) \arrow[ur] & & v_r \equiv (v_p, D_r, f_r^p, \mathcal{O}_r) \arrow[ul] \\
v_s \equiv (v_q, D_s, f_s^q, \mathcal{O}_s) \arrow[u] & &
\end{tikzcd}$$

(b)

Figure 2: (a) Overlap view example for $A[0, 10]$. (b) Composition of views.

and operations o including the random access read and write operators. This View can be applied to any container by providing a mapping function f from the domain $d = [0, n)$ 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 View in STAPL. If new behavior is needed, then the developer can implement it explicitly.

Useful views. There are several types of Views that worthy of note because they enable optimizations or are useful in expressing computations.

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

Transform Views 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 View 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 View 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* View, which takes two (or more) collections and provides a View 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 View heavily used in STAPL is the *overlap* View, in which one element of the View overlaps another element of the view. This View is naturally suited for specifying many algorithms, such as adjacent differences, string matching, etc. The *repeated* View is a special case of an overlapped View in which each element includes the entire collection. As an example, we can define an overlap View 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 View $v^o[i]$ is $A[c \cdot i, c \cdot i + l + c + r - 1]$. See example in Figure 2(a).

Composition of views. A very important aspect of the View definition (Equation 1) is that it naturally enables *composition.* Since the collection referenced by a View can be another View, it is straightforward to define Views over Views. Figure 2(b) shows the construction of Views over other Views, all referencing to the same storage, and the possibility of having multiple Views concurrently referencing the same container. Thus, composition makes possible the construction of complex data organizations and naturally supports the recursive partitioning of domains.

4.1 `pView`: a parallel view

The `pView` generalizes the `View` for parallel use by modifying its components as follows: the collection c becomes a partitioned collection $\mathcal{C}$; the domain d becomes a partitioned domain $\mathcal{D}$; the mapping function f becomes a *mapping function generator* $\mathcal{F}$ that creates a set of mapping functions, one for each element of the partitioned domain; the operations o become an *operation generator* $\mathcal{O}$ that creates a set of operations, one for each element of the partitioned collection. Thus, the `pView` $\mathcal{V}$ is defined as:

$$\mathcal{V} \overset{def}{=} (\mathcal{C}, \mathcal{D}, \mathcal{F}, \mathcal{O}) \tag{2}$$

where $\mathcal{C} = \{c_0, c_1, \ldots, c_{n-1}\}$, $\mathcal{D} = \{d_0, d_1, \ldots, d_{n-1}\}$, $\mathcal{F} = \{f_0, f_1, \ldots, f_{n-1}\}$, 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 *base views* (`bViews`) of the `pView` $\mathcal{V}$. A `bView` may be a `pView` or a `View` (as defined in Eq. 1).

The `pView` supports parallelism by enabling random access to its `bViews`, which can then be used in parallel by `pAlgorithms`.

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 the mapping function generator produces identity functions. This is the default partition adopted by STAPL when calling a `pAlgorithm` to express maximum parallelism.

Other `pViews` that can be defined include the *balanced* `pView` where the data is split into a given number of equal sized chunks, and the *native* `pView`, where the partitioner takes information directly from the underlying container and provides `bViews` that are *aligned* with the `pContainer` distribution. This turns out to be very useful in the context of STAPL.

5 The `pView` class

The `pView` is a distributed object that builds on the STAPL `pContainer` framework. To create a `pView`, the user specifies the partitioned collection (often a `pContainer`), the partitioned domain $\mathcal{D}$, the mapping function generator $\mathcal{F}$, 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 `View`, 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 an initial list of `pViews` available in STAPL. These `Views` are implemented using the schema discussed above, and new `Views` 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

	read	write	[ ]	begin/end	insert/erase	insert_any	find
`array_1d_pview`	X	X	X	X			
`array_1d_ro_pview`	X		X	X			
`static_list_pview`	X			X			
`list_view`	X	X		X	X	X	
`matrix_pview`	X	X	X				
`graph_pview`	X	X			X	X	X
`strided_1D_pview`	X	X	X	X			
`transform_pview`	O		-	-			
`balanced_pview`	X		X	X			
`overlap_pview`	X		X	X			
`native_pview`	X		X	X			
`repeated_pview`	X		X	X			

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 `View` the transform `View` is applied to. `insert_any` refers to the special operations provided by STAPL `pContainers` that insert elements in unspecified positions.

parallel computation. If STAPL algorithms can use balanced or native `pViews`, then performance is greatly enhanced.

Optimizations. We are aware of the trade-offs between the expressivity offered by the `pViews` and the challenges in obtaining 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 the repeated composition of the $\mathcal{F}$s. In cases where $\mathcal{F}$ is statically known, and relatively simple, STAPL can reduce the chain of indirections to one. For instance, composing identity functions results in another identity function, while composing an arbitrary function F with an identity function is the same F. The case of arbitrary maps must be treated dynamically. While not treated automatically yet, a general solution is to pre-compute the outcome of the application of the composed function, reducing the number of indirections needed subsequently.

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 the data than its general method of global data referencing (STAPL provides shared object view).

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: a 832 processor IBM cluster with p575 SMP nodes available at Texas A&M University (called P5-CLUSTER) and a 38,288 processors Cray XT4 with quad core Opteron processors available at NERSC (called CRAY4-CLUSTER). In all experiments a location contains a single processor, and the terms can be used interchangeably. Due to space limitations, results for the P5-CLUSTER are contained in Appendix A.

6.1 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`.

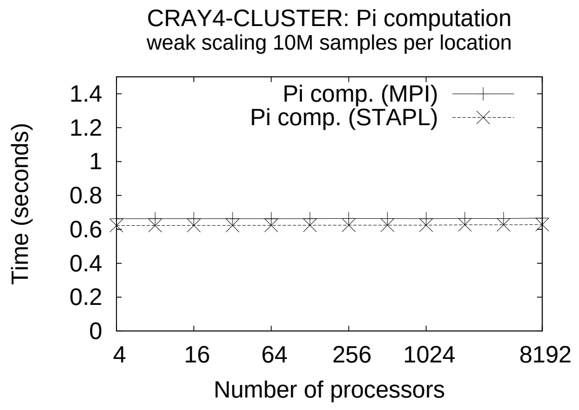
For instance, we can compute π using the well known Monte Carlo method: we generate a number of random points inside the unit square and count how many of these fall inside the unit circle. The ratio between these and the total number of points generated is $\pi/4$. The `View` used to represent the input does not need a reference to storage because the points can be generated on demand. Hence, the container provided to the `View` is a simple class that exports the container interface and whose read method returns a randomly generated point in the unit square. Passing this `View` 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 to compute π. The code snippets are shown in Figures 4 and 5. As it may be noted, the two programs are comparable in terms of complexity for this embarrassingly parallel algorithm. Figure 3(a) 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 `View` and predicate. In this case, given a pattern of length M, we create an overlapped `View` 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 `View` over all the sub-strings of size M of the input text. The code sample is shown in Figure 6. In Figure 7, an MPI version of the program is shown. In this case it becomes possible to appreciate the additional complexity of the MPI 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). Figure 3(b) shows that performance of the two versions is comparable.

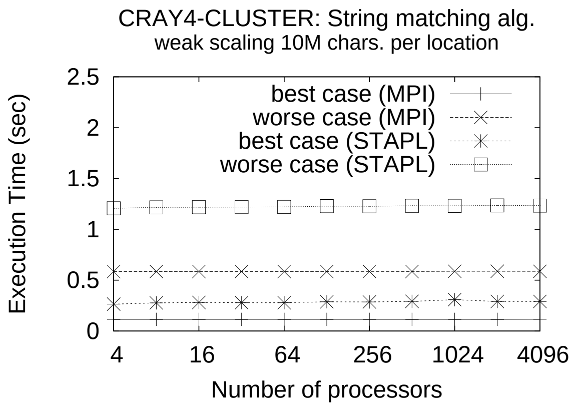
In Figure 3(c), we also show the performance of the basic `stapl::count_if` algorithm when we count how many times a value appears in the input `View`. The plot exhibits the same scalability as the other two, with the execution time smaller since the predicate computation is constituted by a single if statement.

6.2 Matrix views

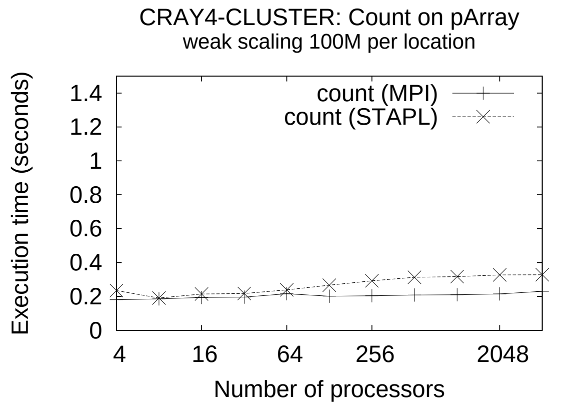
The `pMatrix` is a `pContainer` that implements a two-dimensional dense array [6]. On top of `pMatrix` we can create different types of views to adapt the container to the algorithm requirements. For example, we can initialize the values of a container using `stapl::generate` or `stapl::copy`. Both algorithms require the data layout in a one-dimensional container. Using



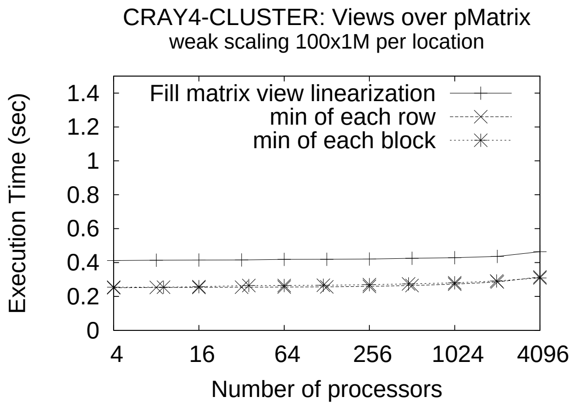
(a)



(b)



(c)



(d)

Figure 3: Execution times on CRAY4-CLUSTER of (a) π computation, (b) string matching, and (c) `stapl::count_if` algorithms on a `pArray`. All algorithms use the `stapl::count_if` generic algorithm on different `Views`. (d) Weak scaling on CRAY4-CLUSTER of matrix filling with random values using a linearization view over `pMatrix` and computing the minimum of each row.

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`). The differences between them are the partitioner and the mapping function generator used.

Figure 3(d) shows the execution time on CRAY4-CLUSTER of three algorithms using `pMatrix`: filling the `pMatrix` using `stapl::copy` from a generator container through the linearization `View`, computing the minimum element of each row and computing the minimum element of each block using `stapl::transform(input_view,output_view,functor)`, where the `functor` finds the minimum of a sequence of elements.

6.3 Graph views

STAPL provides a parallel and distributed graph data structure (`pGraph`). Algorithms operating

```
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);
}
```

Figure 4: STAPL code to compute π, using a view over a generator container

```
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));
```

Figure 5: MPI version of compute π

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 define a `vertex_set_view`, `edge_set_view` and `adjacent_edges_view`. They implement the `static list` concept and support efficient parallel map and map_reduce operations. In Figure 9, we show a STAPL program that performs an initialization of the `pGraph` vertex properties (Figure 9, Line 4), and then computes and stores the set of source vertices in a parallel list (Figure 9, Line 5). The list view defined over a parallel list [17] 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 as input a 2D sparse mesh, 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 10 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 and it scales well when the number of processors is varied from 4 to 2048. `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 10(a) that `stapl::for_each` has no overhead relative to the simple `for_each`. A second experiment we 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 lists, one for each location considered. As we can see from Figure 10(a), the `pList`

```
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));
  ...
}
```

Figure 6: STAPL code to compute string matching, using an overlap partitioned view

```
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 );
  ...
}
```

Figure 7: MPI version of the string matching algorithm

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 `pGraph` view can be defined for a `pArray` of lists of edges. Generic `pGraph` algorithms will operate properly on the data stored in a `pArray`, provided a suitable graph `View` is implemented. In Figure 10(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. `stapl::find_sources` incurs additional overhead because `stapl::find_sources` uses additional graph methods (e.g., `find_vertex`) that are more efficiently implemented in the native `pGraph`. If graph views cannot be defined on `pArray` storage, than the alternative is to copy the data into a `pGraph` and invoke the algorithms on the `pGraph`. We analyze the overhead of this approach in Figure 10(c) where copying the `pGraph` and invoking `stapl::for_each` is ten times slower while `stapl::find_sources` is 2.6 times slower. Unless subsequent algorithms invocations benefit from having the `pGraph` with its native storage (such as using find methods), working directly with the view defined over the `pArray` is a better option.

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 [11]. The parallel Euler Tour algorithm [11] implemented in STAPL uses a STAPL `pGraph` to represent the tree and a `pList`

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

stapl::transform(rowsv, resv, stapl::min_value<int>());
...
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);

stapl::transform(blocksv, resv, stapl::min_value<int>());
...
```

Figure 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

```
p_graph<DIRECTED,MULTIEDGES, vertex_property> pg;
p_list<vertex> pl;
list_view(pl) list_view;
for_each(vertex_set_view(pgraph), init_property());
p_find_sources(vertex_set_view(pgraph), list_view);
```

Figure 9: Find sources and sinks in a graph

to store the final Euler Tour. The algorithm executes in 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 both directions). Figure 10(d) 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.

7 Conclusion

In this paper we have introduced the `pView` a higher level concept that allows programmers to be more expressive and more productive. 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 like STAPL will prove to be both expressive and productive as well as high performance.

References

[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A standard template adaptive parallel C++ library. In *Proc. of the International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors (IWACT)*, Bucharest, Romania, Jul 2001.

[2] 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 *International Workshop on Languages and Compilers for Parallel Computing (LCPC), published in Lecture Notes in Computer Science (LNCS)*, Cumberland Falls, Kentucky, Aug 2001.

[3] Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: Alignment and view update. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, September 2010.

[4] Lubomir Bourdev. Generic image library. *Software Developer's Journal*, page 4252, 2007.

[5] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. STAPL: Standard Template Adaptive Parallel Library. In *Proc. Annual Haifa Experimental Systems Conference (SYSTOR)*, pages 1–10, New York, NY, USA, 2010. ACM.

[6] Antal A. Buss, Timmie Smith, Gabriel Tanase, Nathan Thomas, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. Design for interoperability in STAPL: pMatrices and linear algebra algorithms. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC), published in Lecture Notes in Computer Science (LNCS)*, volume 5335, pages 304–315, Edmonton, Alberta, Canada, July 2008.

[7] D. Callahan, Chamberlain, B.L., and H.P. Zima. The Cascade high productivity language. In *The Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, volume 26, pages 52–60, Los Alamitos, CA, USA, 2004.

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[9] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, and David Padua. Writing productive stencil codes with overlapped tiling. *Concurr. Comput. : Pract. Exper.*, 21(1):25–39, 2009.

[10] Adobe Systems Inc. Generic image library. `http://opensource.adobe.com/wiki/display/gil/Generic\+Image\+Library`.

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

[12] David Musser, Gillmer Derge, and Atul Saini. *STL Tutorial and Reference Guide, Second Edition.* Addison-Wesley, 2001.

[13] Thorsten Ottosen. 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] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Parallel Library. In *Proc. of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburgh, PA, May 1998.

[15] Steven Saunders and Lawrence Rauchwerger. 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. ACM.

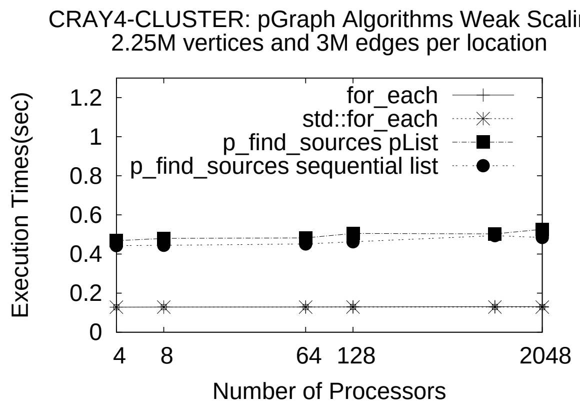
[16] Jon Seymour. Views - a C++ standard template library extension, January 1996. http://www.zeta.org.au/ jon/STL/views/doc/views.html.

[17] Gabriel Tanase, Xiabing Xu, Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. The STAPL pList. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC), published in Lecture Notes in Computer Science (LNCS)*, volume 5898, pages 16–30, Newark, 2009.

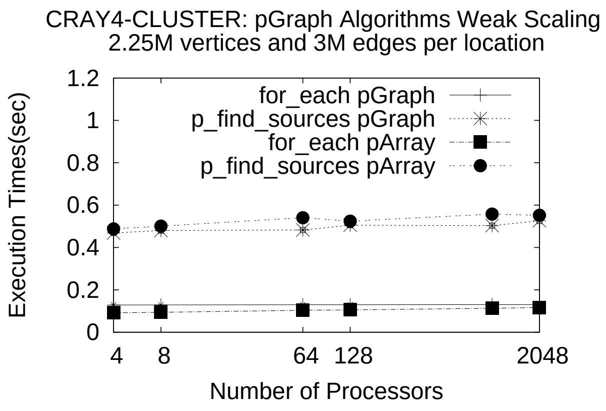
[18] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPoPP)*, pages 277–288, Chicago, IL, USA, 2005. ACM.

[19] Martin Weiser and Gary Powell. The view template library. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000.

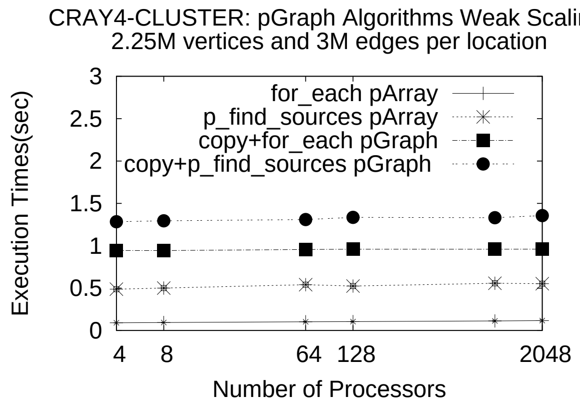
A Results on P5-CLUSTER



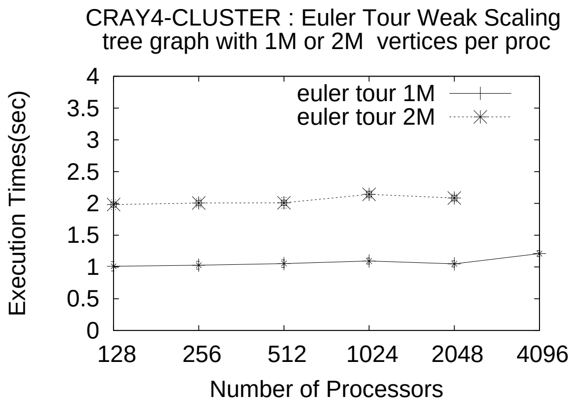
(a)



(b)

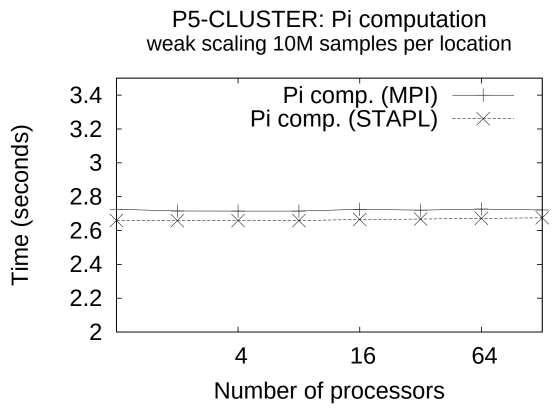


(c)

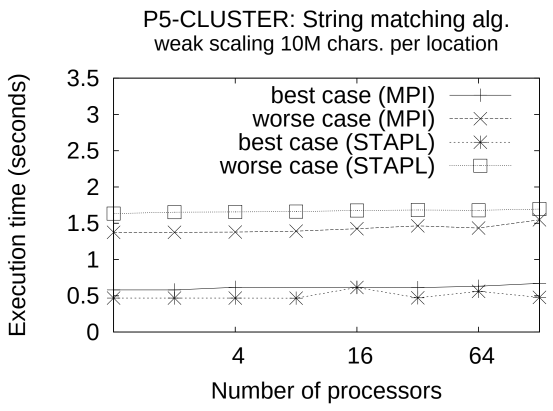


(d)

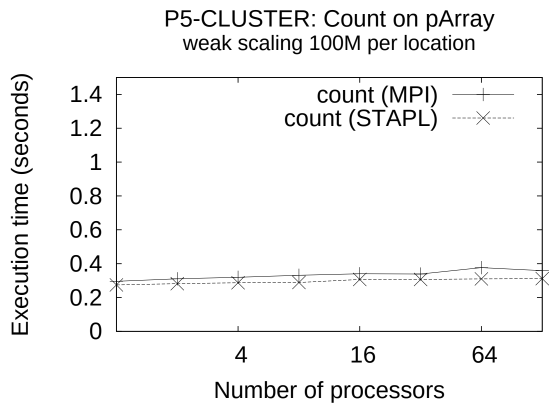
Figure 10: Weak Scaling for `pGraph` methods on CRAY4-CLUSTER 2.25M vertices and ∼3M edges per location. (a) Low overhead of using views: comparison of calling `stapl::for_each` versus a low level `for_each`, and comparison of storing sources in a `pList` versus a sequential list; (b) Comparison of graph algorithms on graph views defined over `pGraph` and `pArray`; (c) Benefits of using a graph view defined over a `pArray` of adjacencies versus copying the data first into a `pGraph` and then invoking algorithms; (d) Weak scaling of Euler Tour algorithm. Tree made by a single binary tree with 1M or 2M subtrees per processor.



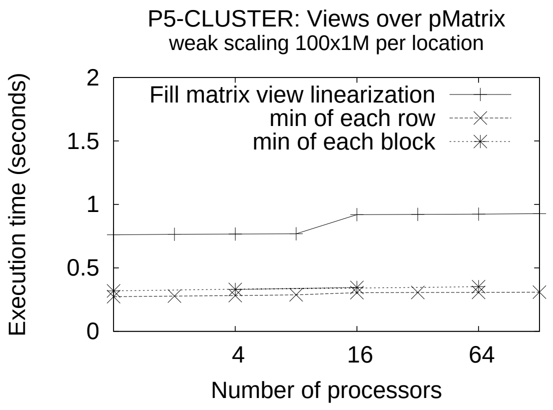
(a)



(b)

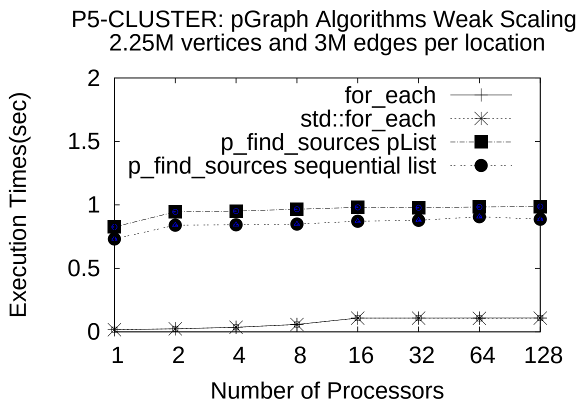


(c)

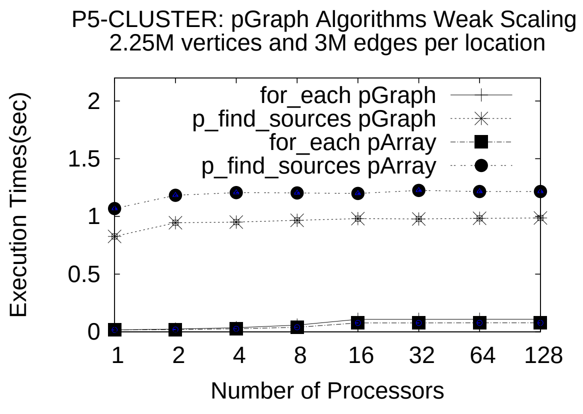


(d)

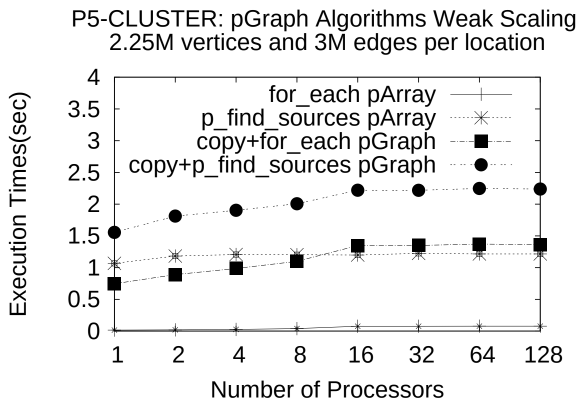
Figure 11: Figure (a), (b), and (c) shows execution times of π computation, string matching, and `stapl::count_if` algorithms on a `pArray`. All algorithms call `stapl::count_if` generic algorithm on different `Views`. 100M elements per location. Figure (d) shows Weak scaling of filling a matrix with random values using a linearization view over `pMatrix` and computing the minimum of each row on CRAY4-CLUSTER. Number of elements per location: 100×1M



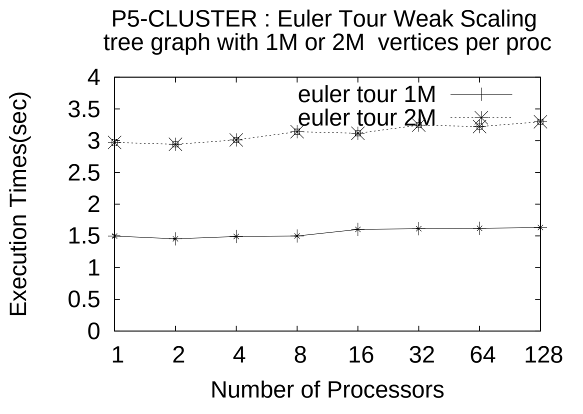
(a)



(b)



(c)



(d)

Figure 12: Weak Scaling for `pGraph` methods on P5-CLUSTER 2.25M vertices and ∼3M edges per location. (a) Low overhead of using views: comparison of calling `stapl::for_each` versus a low level `for_each`, and comparison of storing sources in a `pList` versus a sequential list; (b) Comparison of graph algorithms on graph views defined over `pGraph` and `pArray`; (c) Benefits of using a graph view defined over a `pArray` of adjacencies versus copying the data first into a `pGraph` and then invoking algorithms; (d) Weak scaling of Euler Tour algorithm. Tree made by a single binary tree with 1M or 2M subtrees per processor.