The STAPL Parallel Graph Library*

Harshvardhan, Adam Fidel, Nancy M. Amato and Lawrence Rauchwerger

Parasol Lab, Dept. of Computer Science and Engineering, Texas A&M University
{ananvay, fidel, amato, rwerger}@cse.tamu.edu

Abstract. This paper describes the STAPL Parallel Graph Library, a high-level framework that abstracts the user from data-distribution and parallelism details and allows them to concentrate on parallel graph algorithm development. It includes a customizable distributed graph container and a collection of commonly used parallel graph algorithms. The library introduces `pGraph pViews` that separate algorithm design from the container implementation. It supports three graph processing algorithmic paradigms, level-synchronous, asynchronous and coarse-grained, and provides common graph algorithms based on them. Experimental results demonstrate improved scalability in performance and data size over existing graph libraries on more than 16,000 cores and on internet-scale graphs containing over 16 billion vertices and 250 billion edges.

1 Introduction

Processing large graphs is essential in many domains, from social network and web-scale graphs to scientific meshes and nuclear reactor-design [2]. As the graphs span billions of vertices and edges, they may not fit in the memory of a single-processor system. Using a distributed data-structure allows massive graphs to be processed quickly and concurrently.

There have been many attempts over the past decade [7, 3, 8] to allow programmers to easily express their graph computations in parallel. Despite this, graph algorithms remain notoriously hard to scalably parallelize, and existing graph libraries are restrictive in allowing users to express algorithms and require them to manage many details regarding data-distribution and communication.

This paper describes the STAPL Parallel Graph Library (SGL), a generic parallel graph library that provides a high-level framework that allows the user to concentrate on parallel graph algorithm development and relieves them from the details of the underlying distributed environment. It consists of the STAPL

* 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 DOE NNSA under the Predictive Science Academic Alliances Program 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.

`pGraph pContainer`, `pGraph pViews` that allow for the separation of the algorithm design from the container implementation, and a collection of parallel graph algorithms. In addition, the library supports level-synchronous, asynchronous and coarse-grained algorithmic paradigms, which are designed to support graph processing applications and algorithms. Further, it automates load balancing and simplifies some locality-related optimizations.

The STAPL Parallel Graph Library makes several contributions:

Programmability. One of the main goals of SGL is to provide a similar interface and level of abstraction as sequential graph libraries by allowing seamless access to local and remote elements through virtualization using Shared-Object Views, while providing good performance.

Abstraction. The `pGraph pView` – a high-level graph abstraction – allows programmers to completely decouple the design of the graph algorithm from the implementation of the graph container. Users are left free to express the graph as is most natural to the problem, while the underlying data-structure and implementation can be chosen to offer maximum performance.

Multiple Algorithmic Paradigms. We provide three paradigms for expressing graph algorithms, including two fine-grained (level-sync and async) and one coarse-grained paradigm. These enable natural expression of algorithms while extracting the best performance from different input graphs.

Scalable Performance. We demonstrate improved scalability in performance and data size over tens of thousands of cores compared with existing graph libraries on standard benchmarks. Moreover, we provide light-weight support for load balancing through asynchronous data migration, and demonstrate improved performance and scalability in a real-world production application by mitigating load-imbalance through automatic redistribution of vertices.

2 STAPL Overview

The `pGraph pContainer` is built using the `pContainer` framework (PCF) provided by the Standard Template Adaptive Parallel Library (STAPL). STAPL [4, 5, 13] is a framework for parallel C++ code development. STAPL's core is a library of C++ components implementing parallel algorithms (`pAlgorithms`) and distributed data structures (`pContainers`) that have interfaces similar to the C++ standard library (STL) [10]. Analogous to STL algorithms that use *iterators*, `pAlgorithm`s are written in terms of `pViews` [4] so that the same algorithm can operate on multiple `pContainer`s. `pViews` facilitate parallel processing by supporting independent random access to ranges (partitions) of a container's elements. The `PARAGRAPH` represents computations as parallel task graphs.

STAPL abstracts the physical parallel elements to the notion of *locations* – units of a parallel machine capable of performing computations that have a contiguous memory address space. Asynchronous communication is allowed through remote method invocations (RMIs) on shared objects. The STAPL runtime system is portable to different platforms and architectures without modifying other STAPL components.

```
//create pgraph with 10 vertices :
p_graph<Directed, Multiedges> pg(10);
 size_t  V = pg.num_vertices();
pg.add_vertex(11);

  parallel_for_each   ( i  = 0..E)
    pg.add_edge(rand() % V, rand() % V);

// get the out-degree of vertex 3
 size_t  deg_3 = pg[3].edges(). size ();
// delete a specific edge between two vertices
bool success  = pg.delete_edge (3,2);
```

Fig. 1. Typical interaction with the `pGraph` `pContainer` through methods.

```
// directed graph, multiple allowed edges b/w
// same source & target
p_graph<Directed, Multiedges>

// graph with custom vertex and edge properties
p_graph<Directed, Multiedges, vertex_prop , edge_prop>

// vertices block-partitioned,
// with custom traits for graph
p_graph<Directed, Multiedges, int , bool,
          blocked_partition ,  my_traits >
```

Fig. 2. Example traits of the `pGraph`.

3 The pGraph Container and Implementation

Graphs can be *directed* or *undirected*, with *weighted* or *unweighted* edges, and may or may not allow multiple edges between the same source and target (*multigraph*) or self-loops. Applications may associate information (properties/weights) with vertices and edges.

API. The `pGraph` `pContainer` exports a uniform interface for accessing and manipulating all types of graphs. Every vertex and edge in the graph is uniquely identified by a vertex (or edge) descriptor that is used for accessing and referencing the element, and for adding or deleting elements.

The `pGraph` API makes it simple to create graphs and perform common graph operations (Fig. 1), such as adding, deleting or accessing vertices and edges, applying functors on graph elements, etc. Issues of concurrency and consistency are handled by the `pGraph`. Importantly, users do not have to reason or know about the locality of the graph elements – they refer to vertices and edges using descriptors and the `pGraph` handles the details of locality and forwarding requests to the required location. This is not the case with many other graph libraries, e.g., in PBGL, the user can only get the out-degree of a vertex from a local process, whereas in our model this information is available from all locations.

Users can customize a `pGraph` by selecting properties and traits (e.g., directedness, graph representation, storage). SGL provides common options and implementations for storages, etc., but users may provide their own, or implement bridges to adapt their data structures to our algorithms. These choices may affect the performance. For example, a `pGraph` using vector storage may be faster than one using map storage if the graph is static (i.e., the number of vertices is known *a priori*). It is straightforward to customize a `pGraph` (Fig. 2). Further customizations are possible through trait-classes.

Implementation. The `pGraph` is built using the STAPL `pContainer` framework (PCF), which provides base classes that handle issues dealing with data distribution and parallelism and allows the design of the `pGraph` `pContainer` to focus on graph-specific concerns. The `pGraph` `pContainer` consists of a set of base containers (`bContainers`) and the infrastructure to make them work together in parallel. For the `pGraph`, a `bContainer` is a base graph data structure that exports the `pGraph`'s interface. The `bContainer` has three layers: the representation of the graph, the graph storage, and the underlying storage. The graph storage is tied to the representation, exporting an interface that allows the representation to work with the underlying storage. It provides the policies for the type of underlying storage used by the graph (e.g., vector, hash map, map) for vertices and edges. It also specifies the type for a vertex and type for an edge, along with how properties are stored on these. The underlying storage may be a sequential container unaware of parallelism that is used by the graph to store vertices and edges, or possibly another `pContainer`.

The PCF provides a shared-object view [14] that allows users to address any element globally. `pGraph` users interact with the container by method invocation, which the framework forwards to the location where the needed graph elements reside. Fig. 3 shows the internal base-class implementation for apply_async, which provides an example of address resolution for graph elements using asynchronous communication. The apply_async method is provided by the PCF for applying a higher-order function object on an element of the container. This may be used to implement methods such as add_edge and set_vertex_property for the `pGraph`. Internally, this forwarding is supported by a distributed directory service – which is contained within the `pGraph` – that provides a two-level lookup of the requested vertex's location. This is described in the next section.

Shared-Object View Provided by Distributed Directory. The `pGraph` is a dynamic container, where vertices may be added and removed, and so vertex IDs need not be contiguous or even ordered. The `pGraph` uses a distributed directory to provide a shared-object view to users and abstract them from dealing with the details of distribution. While a distributed directory can increase access costs, other solutions such as centralized models (e.g., the master-slave model employed by Pregel [8]) which store the entire directory information in a single location, or replicated directory on all locations, may not scale to large systems.

In this two-level distributed directory scheme, every vertex has a *home location* associated with it, which may not be the location of the vertex, but is rather the location that stores information about the vertex's locality. It is calculated using simple closed-form solution (a hash of the vertex's descriptor), so any requesting location knows quickly and precisely where to send the request.

In this mechanism, the `pGraph` first checks if the graph vertex is local, and if so, then services the request immediately. If the vertex is not found locally, the local directory computes the *home location* of that vertex and forwards the request there. The *home location* is responsible for knowing the exact location of the vertex. In some cases, the *home location* may own the vertex itself, at which point, the requested action is performed on the vertex. However, in the case that it does not, the request is forwarded to the location that owns the vertex, where the request is serviced. As shown by Tanase et. al. [14], address resolution using asynchronous forwarding provides improved performance over a directory that determines the element's location using synchronous communication.

```
void _base :: apply_async( vertex_descriptor  s,
                            Functor f)
if  base_container . contains(s)
   base_container . apply(s, f)
else
   home = home_location(s) // hash−based lookup
   if  my_location  == home
      owner = directory . lookup(s)
      async_rmi  (owner, apply_async(s, f))
   else
      async_rmi  (home, apply_async(s, f))
```

Fig. 3. Internal base-class implementation of `apply_async` method illustrating address resolution.

```
// asynchronous migration and  redistribution :
g.migrate(vertex,  location)
g. redistribute (cost_map, action_function =no_op)
```

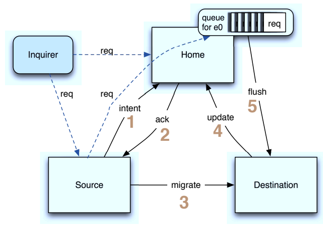


Fig. 4. Asynchronous migration protocol for `pGraph`.

Vertex Migration. The SGL provides the novel ability to migrate vertices asynchronously between locations during the execution of the program. An important property of the migration protocol is that it ensures that SGL algorithms can be oblivious to the data distribution and also to any migration occurring during the execution of an algorithm.

The protocol for migration of graph elements implemented by the `pGraph` is inspired by directory-based cache coherence techniques [6] and is described in Fig. 4. When processing an element-migration request from a source location to a destination location, the source first informs the home location of its intent to migrate (1). The home location, upon receiving this request, marks the element as in the process of migration and creates a queue for all requests addressed to that element. It then sends an acknowledgement (2) to the source location allowing the source to then proceed to migrate the element data to the destination (3). When the destination receives the element's data, it stores it and informs the home location (4) to update its metadata to record that the destination is now the owner of that element. Finally, the home location updates its metadata and forwards all pending requests for that element to the destination location (5). If at any point during migration a location requests access to the element that is currently being migrated, the requests are forwarded to the home location for that element, where they are buffered in the queue. The queue is flushed at the end of migration and requests are forwarded to the new location of the element.

Redistribution. As users of SGL generally may be unaware of localilty, SGL provides a convenient way to rebalance a `pGraph`.

Redistribution of a `pGraph` requires some process for determining the new distribution. This can be user provided or it can be computed based on some cost function. For many graph-based scientific applications, a cost function (cost map) can be determined representing the expected computational costs associated with vertices and edges. In SGL, such cost maps can be user provided, or if no additional information is available, uniform costs can be assumed for all elements. Given a cost map, a new partition that attempts to address the imbal-

```
p_graph<Directed, Multiedges> pg(N);
graph_view view(pg);

strongly_connected_components(view);
connected_components(undirected_view(pg));
page_rank( implicit_view (N, binary_tree_func ()));
```

```
struct binary_tree_func
  size_t size ( size_t parent) { return 2; }

  size_t operator()( size_t parent, size_t idx)
     if (idx == 0)
       return 2*parent+1;
     else return 2*parent+2;
```

Fig. 5. A few examples of creating and using `pGraph` `pViews`, with binary-tree functor.

ance can be computed by an SGL graph partitioning algorithm. Given a desired partition, each location computes the vertices that need to be migrated to itself from other locations and invokes a migrate call on those vertices. Internally, the asynchronous directory forwards the migration request to the correct location where the element is located and initiates the migration of that vertex. Fig. 4 illustrates how redistribution is invoked, and the protocol used in SGL. SGL allows application programmers to optionally provide callback functions that are invoked along with each migration call on the corresponding element to allow any action that needs to be performed during the process of migrating a single element, such as updating auxiliary data structures.

4 pGraph pViews

`pGraph` algorithms are written in terms of `pViews` that export the full interface of the `pGraph` and allow iteration over vertices and edges. While arbitrary partitions can be specified, the default partition of a `pGraph` `pView` matches the physical partition of the graph on the system. This is the `pView` that can offer the best performance and it should be used unless it is not suitable for the algorithm. The `pGraph` supports the standard `pViews` provided by STAPL, as well as some graph-specific `pViews` that are described in this section.

Useful pGraph pViews. The `pGraph` provides many useful views that can be used to logically view and manipulate the structure of a graph. For example, by applying an undirected `pGraph` `pView` to a directed graph, one can use an algorithm that was designed for undirected graphs on a directed graph without explicitly modifying the graph. Fig. 5 shows the creation of an undirected view over a directed `pGraph` which is then used as input for a parallel connected components algorithm. This is a particular need for a motion planning application which constructs a digraph and uses this connectivity information to view the results and as a stopping condition [16]. We also evaluate the performance of a parallel connected components algorithm using this `pView` in Sec. 6.

As another example, some strongly connected components algorithms [9] need access to the predecessors of a vertex in a digraph. For this, a predecessor `pView` can be used to provide the predecessor information without modifying the underlying graph. Or, in some cases, one may wish to work with the complement of a graph which has the edges of the graph complemented. In this case, instead of constructing another graph, one could simply apply a complement view.

Implicit pGraph pViews. A `pView` is a partitioned collection of element descriptors. While these collections are often explicit, with memory associated with each element, STAPL provides for the creation of `pViews` that do not have an underlying collection of elements, but instead evaluate expressions to provide vertices and edges lazily. These views may be used when the graph structure can be described by a series of formulae, with the benefit of having virtually no storage overhead, e.g., users can specify the formula using a functor that, given the descriptor of a vertex, returns the descriptor of its neighboring vertices. This is most useful in scientific applications that work on regular meshes, where the structure may be expressed by formulae. This allows the application to avoid storing the vertices and edges of the graph, freeing up memory for larger problems, or allowing the program to run on memory-constrained systems.

In Fig. 5, the PageRank algorithm is invoked on an implicit binary tree `pView` by specifying the view size and the function object (binary_tree_func) that describes the parent-child relationship for a complete binary tree (Fig. 5). This `pView` can be passed as input to any generic `pGraph` algorithm, the execution of the algorithm lazily creates graph elements on which to operate. Similarly, an n-dimensional hypercube, a mesh, a torus and other classes of regular graphs can be generated by using the appropriate algrebraic expressions.

5 Parallel Graph Algorithms

The SGL provides three paradigms to help users design parallel graph algorithms: the level-synchronous paradigm, the asynchronous paradigm and the coarse-grained paradigm. Using these paradigms, the SGL provides standard fundamental graph algorithms, including breadth-first search, connected components, single-source shortest path, and topological sort, and also more specialized algorithms such as page rank and particle-swarm optimization.

These paradigms are built on top of algorithmic primitives provided by STAPL (e.g., `map_func`, `map_reduce`) that execute higher-order functions (workfunction) on elements of a view. To express a new parallel graph algorithm, users choose a suitable paradigm and provide a workfunction that describes the computation, either in a fine-grained manner for the level-sync and async paradigms, or in a coarse-grained manner for the `coarse-grained` paradigm. Fig. 6 is an example of a workfunction for SGL's parallel breadth-first search (BFS).

The workfunction is generic and oblivious to the paradigm (either level-sync or async). The differences between level-sync and async versions are taken care of by the paradigm itself. For example, the generic BFS workfunction (bfs_wf) will use the visitor (visit_wf) (Fig. 6), and may be used in both, the level-sync or async paradigms. For fine-grained algorithms (workfunctions that operate on individual vertices), the `pViews` provide optimizations that are transparent to the user to better exploit data locality. Coarse-grained workfunctions receive a partition of the graph on which to work.

Level-Synchronous Algorithms. The Level-synchronous paradigm iteratively executes tasks on the active vertices of the graph in a BSP fashion [20], with a

```
void BFS (graph_view graph, vertex source)
  source.color = GREY;
  Paradigm(graph_view, bfs_wf(), bfs_visitor ());

bool bfs_visitor (Vertex v, int level)
  if (v.level > level)
    v.level = level;
    v.color = GREY;
    return true;
  return false;
```

```
bool bfs_wf(Vertex v)
  if (v.color == GREY)
    for (u : v.neighbors())
      spawn(Visit( bfs_visitor (_1, v.level +1)), u)
    v.color = BLACK;
    return true;
  return false;
```

Fig. 6. Pseudocode for generic BFS and workfunctions (bfs_wf and bfs_visitor).

global synchronization between each level. Iterative application of the map/reduce pattern is one way to express the level-sync paradigm.

The algorithm's communication happens asynchronously during and after a level, but is guaranteed to have completed before the next level. In this paradigm, each level is a phase that works on some set of active vertices, which may change through the levels. Level-synchronous algorithms tend to perform best when the number of levels is small since each level requires a costly global synchronization.

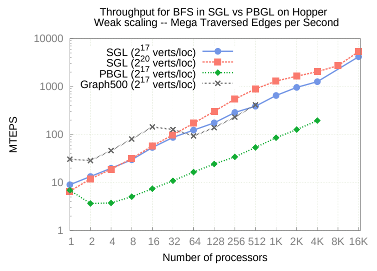
Examples of level-sync algorithms are PageRank [11] and level-sync BFS [18]. To create a level-sync BFS, a user would plug-in the generic BFS workfunction (Fig. 6) into the level-sync paradigm. The workfunction should return true if it was active for a vertex, and false otherwise. This is used to decide the termination condition, which occurs when all vertices are inactive (all vertices return false).

Asynchronous Algorithms. The async paradigm, on the other hand, has no internal synchronizations, and therefore, may perform better on graphs with high diameter. However, asynchronous algorithms may perform redundant work, as there are no guarantees for the execution order. For example, an async BFS may re-visit a vertex multiple times as shorter paths are discovered [12].

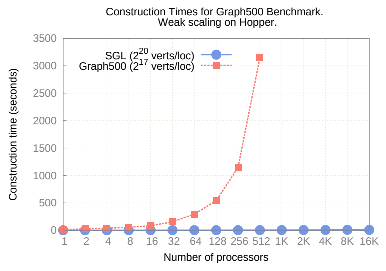
The algorithm typically starts with a few fine-grained *source tasks* over an initial set of vertices. These may spawn additional tasks on their neighboring vertices that are asynchronously forwarded to the location where the neighbor target vertex is currently located (using task forwarding). The algorithm execution ends when there are no more tasks currently executing or in-flight, as detected by a termination-detection algorithm. Termination detection is supported by internal mechanisms that track the number of tasks executing and in waiting and that performs a reduction across locations.

Since most libraries for graph processing provide one of the two paradigms, users either have to use different libraries for different input graphs, or potentially settle for lower performance depending on their input graphs. SGL provides both paradigms, such that the user workfunction is oblivious to the paradigm selected, so it is easy to switch paradigms to obtain the best performance in different cases.

Coarse-Grained Algorithms. The `coarse-grained` paradigm is useful to express graph computations in which a `pGraph` may be partitioned into subgraphs, each of which is processed separately. An example of this type of computation is the coarse-grained connected components algorithm [21]. The first level of the



(a)



(b)

Fig. 7. Comparison of MTEPS (a) and construction times (b) for `pGraph`, PBGL and Graph 500 ref. implementation. Note log-scale y-axis for (a)

algorithm computes the connected-components of the local subgraph, ignoring remote edges. In the second level, the local connected components are merged by applying a level-synchronous connected components to the graph. Then, the CC vertices are relabeled with the CC-identifier of their connected component. This allows the algorithm to reduce communication by coarsening local computation.

As an example, the Motion-Planning applications [17] follow this paradigm, where they build the graph locally in coarse-grained partitions, and then merge the graphs to get the final result.

6 Results

We evaluate SGL using multiple input graphs and over multiple platforms and show that our library performs better, both in terms of scalability and memory used, than other available graph libraries, Parallel Boost Graph Library (PBGL) (v0.7.0), Multi-Threaded Graph Library (MTGL) (v1.1.1), and the Graph500 MPI Reference Implementation (benchmark) (v1.2). (see Sec. 7).

We show scalability of SGL algorithms over a representative subset of input graphs, including the Graph500 Benchmark-generated input (that simulates internet-scale webgraphs and social-networks) and torus graphs (that simulate scientific meshes). Our experimental studies are conducted on two massively parallel systems: a 153,216 core Cray XE6 (HOPPER) and a 832 core Power5 cluster (P5-CLUSTER). For testing MTGL, we run strong-scaling on an 8-core node of a 2,400 core Opteron cluster (OPTERON). We also run a strong-scaling experiment on a real-world production application using SGL on OPTERON.

Graph 500 Benchmark. We implemented the Graph 500 benchmark [1] for SGL, using the level-sync BFS. We show the results and scalability on HOPPER.

In our experiments, while PBGL and benchmark could only accomodate 2^{17} vertices per core at scale, SGL was able to fit a maximum of 2^{20} vertices per core due to less memory needed for storing outstanding communication requests. We show a weak-scaling plot comparing the scalability of the Graph500 benchmark, PBGL, and SGL for 2^{17} and SGL for 2^{20} vertices per location in Fig. 7(a). The

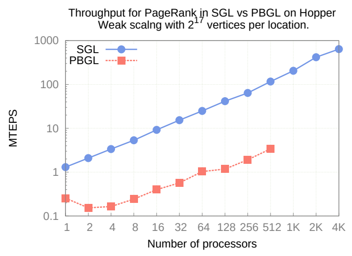


Fig. 8. Level-Sync PageRank on Hopper: SGL and PBGL.

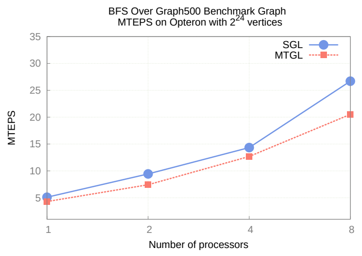


Fig. 9. Strong scaling for the Graph 500 benchmark: MTGL vs `pGraph`.

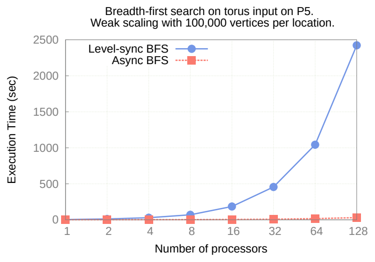
y-axis reports the throughput in Mega Traversed Edges Per Second (MTEPS). Both the benchmark and PBGL suffered from memory bottlenecks. While the Graph500 reference implementation was able to construct the graph, it crashed during the execution of the algorithm on 1,024 cores and PBGL was unable to run the algorithm beyond 4,096 cores. On a single core, PBGL performed similar to SGL, whereas the Graph500 benchmark implementation was 5x faster due to the use of Compressed Sparse-Row (CSR) representation of the graph, while PBGL and SGL used the adjacency-list representation. While CSR is faster for executing the algorithm, it takes a considerable amount of time to build the graph (Fig. 7(b)) as edges need to be globally shuffled to maintain contiguous access through the edgelist. The high overhead of generating the CSR prompted us to use the adjacency-list representation (which is also timed by the Graph500 benchmark specification). We also observed that SGL scaled better than both the benchmark and PBGL. This is more evident for larger inputs, as more local work better hides the communication overhead.

The poor scalability of PBGL and the reference implementation may be explained in part due to insufficient aggregation of messages and the use of ghost nodes for PBGL. The benchmark generates a large number of small messages while executing the algorithm, which overloads the MPI buffers of the machine. The STAPL runtime-system aggregates messages by combining messages being sent to the same location, as well as buffering them and then sends fewer messages, of bigger size through MPI. This helps achieve better performance – as is more evident when going off-node (24 cores) – due to sending messages in bulk, as well as prevents the communication sub-system from running out of memory. This is also why SGL can run on larger graphs than PBGL and benchmark.

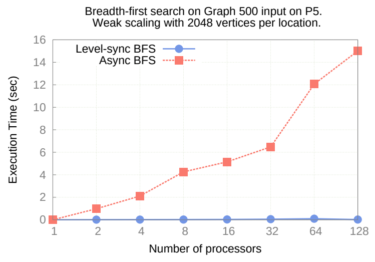
We also compare SGL's level-sync BFS with MTGL's BFS implementation using Qthreads in Fig. 9. We can see that MTGL and SGL exhibit similar behavior on a shared-memory node in terms of strong scaling.

Parallel Graph Algorithms. In this section, we analyze the performance of several parallel graph algorithms for various input types.

Level-Synchronous vs Asynchronous Paradigms. Fig. 10 compares SGL's async and level-sync BFS variants on a torus and a Graph500 Benchmark graph. The



(a)



(b)

Fig. 10. Scalability (time) for both async and level-sync BFS over (a) torus and (b) the Graph 500 Benchmark graph.

async BFS spawns new computation (tasks) asynchronously as it reaches a remote edge, whereas the level-sync algorithm (used in the Graph500 benchmark) is a BSP-style [20] computation with asynchronous communication-phases. In both cases, communication is proportional to the number of remote edges.

The torus graph represents the worst-case scenario for parallel BFS scalability – the algorithm is serialized due to the topology of the torus, and its mapping on the machine (blocked distribution, sliced vertically). In this worst-case scenario (Fig. 10 (a)), the async BFS performs much better than the level-sync BFS, due to the absence of synchronization-points. This trend continues at scale, (upto 4,096 cores shown in Fig. 11). However, for the Graph500 input graph, where there are vertices with massive out-degree, the async BFS performs much worse due to the large number of asynchronous tasks created that may need to be re-created if the vertex is revisited in the traversal (with a smaller distance-from-source, for example, as the ordering of tasks is not guaranteed). The level-sync BFS performs well in this case due to the input graph's low diameter, which implies fewer synchronization points (one fence per level of BFS, i.e., the number of global synchronizations is directly proportional to the diameter of the graph).

These experiments suggest that the async paradigm is better suited for large-diameter graphs, while graphs with smaller diameters and high out-degrees are better suited to the level-sync paradigm.

Coarse Grained Paradigm. To compare the fine-grained and `coarse-grained` paradigms, we ran three versions of the connected components algorithm on a torus graph: a naive fine-grained, level-sync algorithm, a fine-grained connected components algorithm on an undirected view of a directed input graph (Sec. 4), and the coarse-grained connected components algorithm.

Fig. 12 shows weak scaling results for these algorithms. The coarse-grained algorithm provides better performance and scalability since it reduces the communication and the graph size significantly for the subsequent phases by coarsening local connected-components. Up to four-cores, the level-sync paradigms are faster due to communication-overhead being negligible and the overhead of coarsening. However, at scale (>256 cores), the communication overhead starts

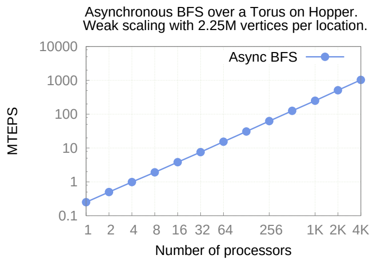


Fig. 11. Asynchronous BFS over a torus `pGraph` on Hopper.

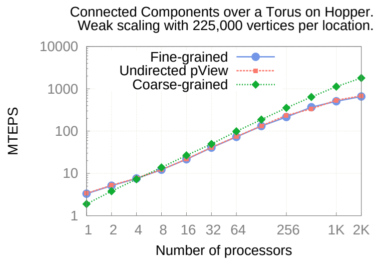
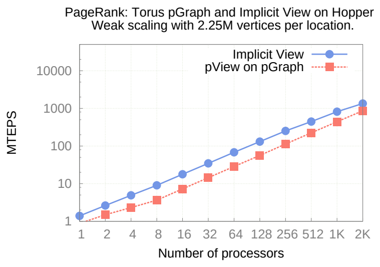
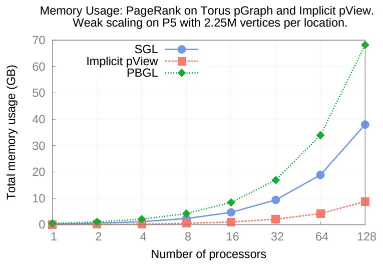


Fig. 12. Weak scaling for connected components over a torus `pGraph`.



(a)



(b)

Fig. 13. Evaluation of PageRank execution time (a) and memory usage (b) using `pGraph` vs. Implicit View on a Torus graph.

becoming more significant. In this scenario, doing extra local work to reduce communication benefits the performance of the algorithm at high core-counts. The performance for the level-sync paradigms degrades beyond 1,024 cores, while the coarse-grained variant scales better. There is also no significant overhead of the undirected-view over a digraph vs. using the undirected graph as input.

PageRank. We ran SGL PageRank on the input graph generated by the Graph 500 benchmark. Fig. 8 shows weak-scaling results for PageRank on the Graph500 input for `pGraph` compared to PBGL's implementation. SGL scales better than PBGL on HOPPER up to 512 cores, after which PBGL crashes while executing the algorithm, while SGL PageRank continues to scale to the tested 4,096 cores.

Implicit Views. We evaluate the performance and impact of Implicit Views in (Fig. 13), which are based on evaluation of expressions and use negligible storage (Sec. 4). We run the PageRank algorithm on a torus graph (weak-scaling), and compare it with a view over a `pGraph` in terms of the throughput (Fig. 13(a)). The Implicit View outperforms the `pGraph`, as the edges are generated with simple formulae and do not have the overhead of accessing and traversing the underlying container storage. In addition, the `pGraph` exhibits a slight increase in execution time going from 8 to 16 cores. This can be attributed to the saturation of the memory bus on a node for this particular architecture. The performance

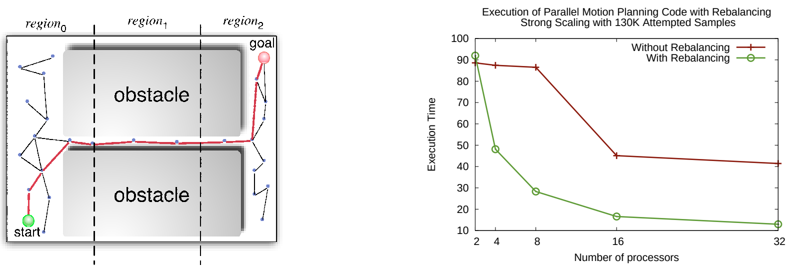


Fig. 14. Strong scaling for a parallel motion planning application on a poorly balanced environment with and without rebalancing techniques.

of the Implicit View is not affected by this phenomenon, as it does not need to go to memory but evaluates formulae to generate temporary vertices and edges instead. We can also observe that the amount of memory consumed (Fig. 13(b)) by Implicit Views is much less, as there is no memory used for storing the vertices and edges of the graph. The only storage needed is for storing properties that are written to by PageRank. Also shown is memory consumed by PBGL.

Redistribution: Application and Performance. Motion planning is the problem of finding a path for a movable object through an environment from a start to a goal configuration. Sampling-based motion planning is a probabilistic method consisting of two phases: generation and connection of samples representing valid (e.g., collision-free) points in configuration space (C-space) of the object, and querying of the roadmap for valid paths.

Jacobs et. al [17] introduced a scalable parallel application for sampling-based motion planning that subdivides the C-space into regions and constructs independent roadmaps for each region. The regions are then connected to form a single roadmap. This algorithm was implemented using SGL where both the regions and roadmap are `pGraph`s. In complex environments, regions could have varying numbers of obstacles, creating regions with fewer nodes, and leading to an imbalance in computation during the connection phase. Instrumenting this real-world production application to invoke `pGraph` redistribution support on the region graph helps the application scale, as well as run faster on unbalanced inputs (Fig. 14) with minimal input from the application, as the application needs only provide the costs it associates with each vertex.

7 Related Work

While much effort has been put into making array-based data structures suitable for parallel programming, graphs have not received as much attention. This section reviews some of the more relevant projects in this area.

Graph500 [1] provides a reference implementation for its benchmark, but is not intended to be a generic library. It provides a baseline for our performance comparisons as it is how users naturally express parallel BFS in MPI.

The Parallel Boost Graph Library (PBGL) [7] is a stand-alone graph library that is closest to the goals of SGL. An important difference from SGL is that since PBGL does not have a shared-object view, it exposes users to explicit knowledge of parallelism and data distribution details through the use of process groups. PBGL's interface requires the user to know explicitly the location of a vertex before any operations may be performed on it. In particular, many methods in PBGL require the vertex/edge they are operating on to be local to the process, and therefore, there is no locality-agnostic way to access remote vertices and edges. This added complexity affects the programmer's ability to create scalable graph algorithms. Another difference is that PBGL only provides the ability to express level-sync algorithms. Further, PBGL is based on MPI, whereas SGL can use different communication libraries through the portable STAPL runtime system.

The Multi-Threaded Graph Library (MTGL) [3] is designed to work on Cray XMP massively multithreaded machines, and utilize their unique architectural features. It can be ported to other platforms using the QThreads library, which requires the programmer to know the QThreads API, as well as details of multi-threaded programming. However, MTGL is limited to shared-memory systems.

Google's Pregel [8] is a library for processing graphs in parallel that emphasizes vertex-centric computation and algorithm design that supports Bulk Synchronous Processing (BSP) style [20] algorithms. However, it is restrictive in allowing users to read remote vertices, as it does not provide a shared-memory view. Further, Pregel employs a master-slave model which allows for fault-tolerance, but may limit scalability. Neither Pregel, nor MTGL, nor PBGL provide asynchronous or coarse-grained paradigms.

Green-Marl is a domain-specific language for graph analysis and provides an implementation for shared-memory systems [19]. It allows users to write algorithms naturally, while the compiler generates parallel code for different targets.

8 Conclusion

This work describes the STAPL Parallel Graph Library, a generic, extensible and scalable parallel graph library built on the STAPL infrastructure. It provides a highly customizable parallel graph container, support for various algorithmic paradigms to express parallel graph algorithms, and useful abstractions in the form of `pGraph` `pViews`. We presented the general design for the `pGraph`, along with various features to improve the performance of graph applications. We compared against relevant graph benchmarks and libraries, and showed that SGL algorithms scale beyond tens of thousands of cores and are comparable to a real-world tuned benchmark code implementation. Further, algorithms were able to scale to more cores and run on larger graphs than comparable graph libraries without sacrificing expressivity.

9 Acknowledgements

The authors would like to thank Sam Jacobs and Nicolas Castet for their help with implementing, instrumenting and running the Motion Planning application.

References

1. The graph 500 list. http://www.graph500.org.
2. M. Adams and E. Larsen. Fast iterative methods for discrete-ordinates particle transport calculations. *Progress in nuclear energy*, 40(1):3–159, 2002.
3. J. W. Berry, et. al. Software and algorithms for graph queries on multithreaded architectures. *Par. and Dist. Proc. Symp., Int.*, 0:495, 2007.
4. A. Buss, et. al. The STAPL pView. *Int. Wksp. on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, Houston, TX, USA, September 2010.
5. A. Buss, et. al. STAPL: Standard template adaptive parallel library. *Proc. Annual Haifa Exp. Sys. Conf.*, p1–10, New York, USA, 2010. ACM.
6. D. Culler, et. al. *Par. Comp. Architecture: A Hardware/Software Approach.* The Morgan Kaufmann Series in Comp. Arch. and Design, 1998.
7. D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. *Par. Object-Oriented Scientific Computing*, July 2005.
8. G. Malewicz, et. al. Pregel: a system for large-scale graph processing. *Proc. Int. Conf. on Management of Data*, p135–146, New York, USA, 2010. ACM.
9. W. McLendon III, et. al. Finding strongly connected components in distributed graphs. *J. Par. Dist. Comp.*, 65(8):901–910, 2005.
10. D. Musser, et. al. *STL Tutorial and Ref. Guide, Second Ed.* Addison-Wesley, 2001.
11. L. Page, et. al. The pagerank citation ranking: Bringing order to the web. 1998.
12. R. Pearce, et. al. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. *Proc. of the ACM/IEEE Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Wash. DC, USA, 2010.
13. S. Saunders and L. Rauchwerger. ARMI: an adaptive, platform independent communication library. *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog.*, p230–241, San Diego, CA, USA, 2003. ACM.
14. G. Tanase, et. al. The STAPL Parallel Container Framework. *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog.*, p235–246, San Antonio, TX, USA, 2011.
15. N. Thomas, et. al. A framework for adaptive algorithm selection in STAPL. *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog.*, p277–288, Chicago, IL, USA, 2005.
16. S. Thomas, et. al. Parallel protein folding with STAPL. *Concurrency and Computation: Practice and Experience*, 17(14):1643–1656, 2005.
17. S. A. Jacobs, et. al. A scalable method for parallelizing sampling-based motion planning algorithms. *Proc. IEEE Int. Conf. Robot. Autom.*, 2012.
18. M. J. Quinn, et. al. Parallel graph algorithms. *ACM Comp. Surv.*, p319–348, 1984.
19. S. Hong, et. al. Green-marl: a dsl for easy and efficient graph analysis. *Proc. Int. Conf. Arch. Sup. Prog. Lang. Operat. Sys.*, p349–362, New York, USA, 2012. ACM.
20. L. Valiant. Bridging model for parallel computation. *Comm. ACM*, p103–111, 1990.
21. F. Dehne, et. al. Efficient Parallel Graph Algorithms for Coarse-Grained Multicomputers and BSP. *Algorithmica*, p183–200, 2002.