A Study of Parallel Graph Processing Paradigms

Ali Khalid, Adam Fidel, Timmie Smith and Nancy M. Amato

Parasol Lab, Dept. of Computer Science and Engineering, Texas A&M University

The Standard Template Adaptive Parallel Library (STAPL) is a parallel superset of the C++ Standard Template Library (STL); STAPL allows execution of programs on uniprocessor and multiprocessor architecture with both shared and distributed memory. Furthermore, STAPL allows ease of development as it provides parallel counterpart of STL containers, algorithms and iterators. STAPL Graph Library (SGL) is a library built on top of the STAPL framework. SGL provides parallel graph data structures, algorithms, and various tools which are helpful in graph processing at scale. In this paper I will discuss various paradigms of execution for parallel graph processing: KLA, Level-synchronous, Asynchronous, Hierarchical, and Hubs and their effect on various inputs like the Twitter network graph, the Kronecker graph and the USA road network. We will then analyze performance of different graph paradigms using various algorithms and list the correlations it has with the topological properties of the input data and machine size.

1 Introduction

Processing big graphs in parallel is crucial in many domains and has wide real-world applications, from studying the topology of Facebook's social media network, ranking web pages on Google based on importance, or computing shortest paths between physical locations in road maps. Many graphs of interests span billions of vertices and edges and may not fit into memory of a single-core machine thus require distribution of graph over multiple cores using parallel computing. Hence, distribution over multiple cores allows efficient and concurrent processing of massive graphs.

STAPL Graph Library (SGL)[1] is a library built on top of the STAPL[2] framework, providing parallel graph data structures, algorithms and various tools for parallel graph processing at scale. In addition it provides paradigms for execution parallel graph algorithms. The paradigms are various strategies which provide differing performance characteristic based on the characteristics of the graph, the algorithm and the machine it is being executed on.

SGL makes several contributions: Firstly, SGL provides an easy to use interface similar to any sequential graph library. Secondly, SGL allows for transparent distribution of data and provides the user with a Shared-Object View, meaning it allows easy access of data, independent of its physical location. Thirdly, the high level graph abstraction with pGraph allows users to focus on design of the graph algorithm rather than the implementation of the underlying graph containers. Moreover, SGL also provides multiple execution paradigms that transparently reduce communication during execution such as: KLA, Hubs and Hierarchical.

2 SGL Overview

SGL follows a vertex-centric programming model. The vertex-centric programming model is a "think-like-a-vertex" model where the computation is expressed from the perspective of an individual vertex in the graph and if active it passes similar information to all its neighbors. The algorithm is expressed with a pair of two functions:

1. Vertex operator that is applied to a vertex
2. Neighbor operator that is applied to all of its neighbors

In a simple BFS example:

```
Bool vertex_op(v)
 if (v.active)
   v.set_inactive();
   VisitAllNeighbors(neighbor_op(
                          v.dist()+1), v);
   return true;         //vertex was active
 else return false;   //vertex was inactive
END
```

```
Bool neighbor_op(u,  dist)
 if (u.dist > new_dist)
   u.dist = new_dist;    //update dist
   u.set_active();        //visited
   return true;           //updated u
 else return false;
END
```

In this trivial BFS example, first the vertex operator is applied to a vertex, where it checks to see if a vertex is active; if the vertex is inactive the operator returns false. Otherwise, for

every active vertex, the operator sets it to inactive and then applies the neighbor operator to every neighbor. The neighbor operator updates the current level of the neighbors to the parent's descriptor and returns true. If the neighbor was not updated, it returns false.

2.1 pGraph

The SGL pGraph allows for uniform interface for accessing, adding, deleting or mutating elements. Importantly, the user does not need to know the physical locations of the graph elements; it is all transparently handled by the pGraph. A descriptor is used to uniquely identify a vertex or an edge in a graph: the same descriptor can be used to perform any user-defined or pre-defined operation on the specified edge or a vertex, without knowing the physical location. The pGraph API also makes it simple to customize the properties of the graph such as directedness and multi-edges. SGL provide many common properties, but a user may define their own custom properties. Hence the API makes it simple and easy to create a graph and to apply pre-defined or user-defined operations like adding, mutating or deleting vertices or edges.

2.2 Shared-Object View

SGL uses distributed storage, yet provides the user with a shared-object view. To a user, accessing elements would be similar to accessing elements for any sequential single-core implementation; hence the user stays oblivious to the inner details of the distribution of the elements. SGL uses a distributed directory scheme in which every vertex has a home location associated with it. It is not the physical location,, but it stores the locality information about the vertex. Whenever accessing elements, if the element is found locally the request is processed. Otherwise, the local directory computes the home directory for the particular vertex and forwards the request to the home directory. In some cases, the element is found in the home directory and the request is serviced. If not found, the request is forwarded to the location where the vertex is stored and the request is serviced.

3 STAPL Overview

Parallel computing is becoming widely popular due to ease of availability of multicore machines and the need to solve larger problems. Parallel libraries like STAPL allow abstraction from communication and data distribution for parallel processing of algorithms, which allow

users to focus on solving the actual problem; all else is taken care of by the framework itself. STL provides iterators, algorithms and containers, STAPL being the parallel equivalent of STL provides pViews[3] which are equivalent to iterators in the sense that they provide access to data in the pContainers[4]. Views emphasize processing data ranges over accessing a single element. pAlgorithms, being the parallel equivalent of STL Algorithm, provides various algorithms in parallel. pALgorithms are written in terms of Views operations. Since parallel programming highly depends on the architecture of the machine, STAPL is designed to continuously adapt to the system and the data at all levels; from balancing data distribution to selecting the most appropriate algorithm implementation. And STAPL pContainers are the parallel equivalent of STL Containers. pContainers are distributed, thread safe and concurrent objects. The data is distributed amongst the machine but the user is given a shared object view. STAPL currently provides containers such as: pVectors, pList, pArray, pMap, pSet, etc.

STAPL pAlgorithms are specified as PARAGRAPHS. The PARAGRAPH is a Task Dependent Graph (TDG), briefly a TDG is a collection of vertices representing tasks and edges represent dependencies between tasks if any. A task represents both work represented by work-functions and data from pContainers.

The STAPL Run-time System[5] provides developers with the following facilities:

1. Adaptive Remote Method Invocation (ARMI)[6] provides interface to underlying Operating System for communication and hardware architecture.

2. Executor is responsible for parallel execution of computation represented by pRanges.

3. Scheduler provides automatic or user defined.

4. Performance Monitor provides user feedback and manages adaptiveness.

4 Execution Paradigms

Communication between processors in a parallel machine is one of the limiting factors for performance. SGL implements various execution paradigms that transparently reduce communication during execution which will be described in this section:

4.1 KLA Paradigm

KLA [7] is an execution paradigm that bridges asynchronous execution with level- synchronous execution. It does so by allowing the level of synchrony to be varied between none (level synchronous) and n (asynchronous) n being greater than or equal to the number of vertices in the graph. Level synchronous is expensive as it requires global synchronous after each step. Therefore it works well in cases which require fewer number of global synchronizations. Asynchronous results in some redundant work but less global synchronizations. The KLA paradigm bridges the gap between the two approaches by combining the two methods and finding an optimal level of synchrony. KLA works in phases, where it allows the algorithm to proceed asynchronously up till K levels similar to BSP model. When k=1 the algorithm proceeds asynchronously up till one level, i.e., level-synchronous fashion. When k=number of vertices, the execution is performed in asynchronous fashion. When $1 < k < d$, d being the number vertices in the graph, the algorithm proceeds in $\frac{k}{d}$ steps, referred to as KLA super steps. Algorithm proceeds asynchronously for each super-step.

4.2 Hierarchical Paradigm

The hierarchical[8][9] approach reduces communication between processors by creating a hierarchical graph representing the machine hierarchy. SGL pGraphs distribute data amongst multiple processors and on large graphs like web graphs, or social media graphs the high number of edges between vertices result in a lot of communication overhead. The Hierarchical approach first creates partitions by grouping graph vertices according to locality of each vertex according to the machine hierarchy. Next, it adds a super-vertex to each partition, which represents the underlying edges between two partitions, thus, creating the hierarchical graph. Finally, all outgoing communication to the vertex neighbor is aggregated into a single message, in turn greatly reducing the volume of communication.

4.3 Hub Paradigm

The Hub[8] execution paradigm follows a similar approach to the one used by the hierarchical paradigm. However the paradigm depends on certain factors, first it reduces communication to hub vertices. A hub vertex is a vertex that has an extreme amount of edges, such as in a social network graph (i.e. Twitter or Facebook). Secondly it requires an additional parameter which acts as a control to the minimum number of in-degree edges required to qualify as a hub. Then

all messages to a given hub are aggregated by a hub representative on the local core and applied to the hub at the end of each hierarchical super-step, hence distributing the work of processing hubs.

5 Methods

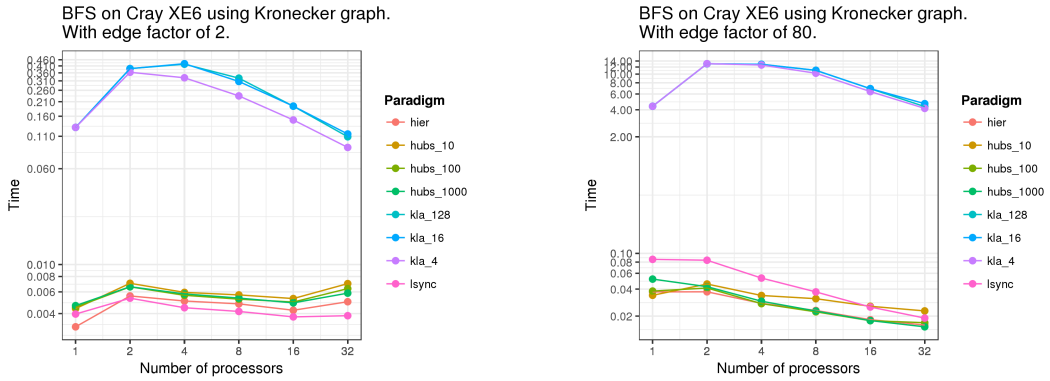
Increased performance from running a program on a two cores compared to a single core does not imply scalability. Our aim is to run the program on hundreds of cores in order to measure execution time and trends that various execution paradigm show. We will run experiments using Cray XE6 up to 512 cores; experiments will have various factors that will influence the execution time and relative trends. Moreover, the input data set will vary as well in order to analyze various features and their correlation to different execution paradigms. The input graphs we used for our experiments were Twitter network graph[10], Kronecker graph, and USA road network. We will vary the size of the graph, number of edges in a graph and graph topology. Moreover, we will test highly connected graphs, graphs with hub vertices and graphs containing many cycles and a large diameter. The experiments will test three hypotheses:

- Hypothesis 1. Highly connected graphs with higher edge factor will work better with hierarchical paradigm.
- Hypothesis 2. Social media graphs that contain high number of hub vertices will show better results with hubs paradigm.
- Hypothesis 3. Graphs with a larger diameter and large number of cycles tend to perform better with KLA.

6 Results

Hypothesis 1. Highly connected graphs with higher edge factor will work better with the hierarchical paradigm.

These experiments compare the result of breadth first search algorithm on the Kronecker graph. A Kronecker graph is a graph model that simulates a power-law graph such as a social network (e.g., Facebook). It contains 2^{Scale} vertices and $V * edgefactor$ edges. Figure 1(a) shows a Kronecker graph with a scale of 15 and an edge factor of 2. Figure 1(b) shows Kronecker graph



(a) (b)

Figure 1: BFS on Cray XE6

with the same scale however the edge factor was increased to 80. In the first experiment with low edge factor, there was relatively less communication between processors because there were fewer edges. In the second experiment by increasing the edge factor we considerably increased the number of edges and increasing the number of edges increased the communication overhead. The trends are very clear, in the first experiment, level-synchronous performs better as the in and out degree of the vertices were low. However, once the edge factor was increased, the trend changed and both the hierarchical approach and Hubs paradigm performed better, as they reduced communication due to adding a hierarchical graph and aggregating communication.

Hypothesis 2. Social media graphs that contain high number of hub vertices will show better results with hubs paradigm.

Social media graphs such as Twitter, or web graphs like Google tend to contain many hub vertices. Hub vertices are vertices with an extreme amount of edges. We ran Pseudo Diameter on Cray XE6 using the Twitter network graph from 2011 on 512 cores in order to measure relative performance between different paradigms as seen in figure 2. Since social media networks tend to be highly connected, the hierarchical approach works better than level-synchronous, but in this case, these networks contain high number of hub vertices. Hence by aggregating all messages to a given hub by a hub representative on the local core and applying to the hub at the end of each super step, we were able to further reduce communication and distribute the work of processing hubs. Therefore, hubs paradigm shows better performance in this case than any other paradigm.

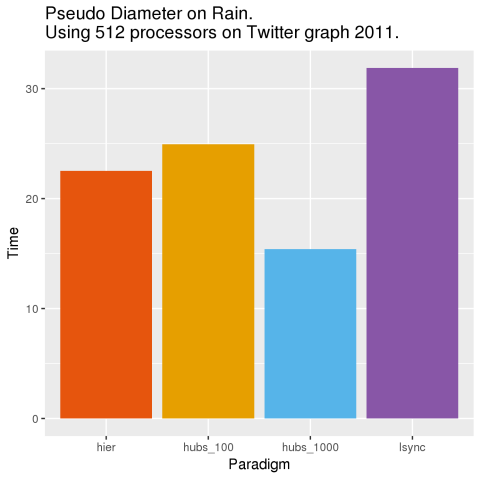


Figure 2: Pseudo diameter on Cray XE6

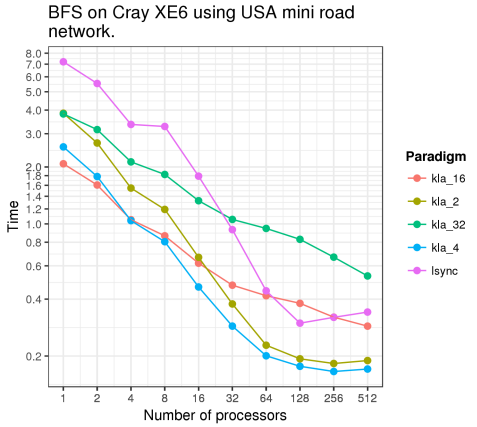


Figure 3: BFS on USA road network

Hypothesis 3. Graphs with a bigger diameter and large number of cycles tend to perform better with KLA.

Figure 3 shows the USA road network. A road network tends to be loosely connected, containing large number of cycles and have large diameters. In these cases using a traditional asynchronous approach could be very costly, as the amount of redundant work can be considerable.The level-synchronous approach does not tend to work well either, as it would require a large number of expensive global synchronizations. KLA paradigm bridges that gap, and runs asynchronously for K steps before synchronizations, reducing redundant work and reducing expensive global synchronization. In this figure, we see that with $k = 4$ we were able to see the

fastest configuration.

7 Summary

SGL provides scalable performance using multiple cores. SGL provides an easy to use interface and makes the inner details of data distribution separate from graph design. In this paper, I examined and analyzed various paradigms of execution implemented in SGL. I was clearly able to show, with supporting evidence, which paradigms work well depending on the topology of the input graphs. Graphs that have a higher edge factor tend to show better performance with Hierarchical paradigms. Graphs containing hub vertices such as social media networks show better performance with Hierarchical Hubs paradigm. We also showed that road networks tend to perform better with KLA, due to a large diameter and high number of cycles.

In the future, it would be interesting to see how different paradigms performs compared to other topological features of the graph.

References

[1] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. *The STAPL Parallel Graph Library* In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Tokyo, Japan, Sep 2012.

[2] Antal Buss, Harshvardhan, Ioannis Papadopoulos, Olga Tkachyshyn, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato, and Lawrence Rauchwerger. *STAPL: Standard Template Adaptive Parallel Library* In Haifa Experimental Systems Conference, Haifa, Israel, May 2010.

[3] Antal Buss, Adam Fidel, Harshvardhan, Timmie Smith, Gabriel Tanase, Nathan Thomas, Xiabing Xu, Mauro Bianco, Nancy M. Amato and Lawrence Rauchwerger *The STAPL pView* In Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Oct 2010. Also, Technical Report, TR10-001, Parasol Laboratory, Department of Computer Science, Texas A&M University, Jul 2010

[4] Gabriel Tanase, Antal Buss, Adam Fidel, Harshvardhan, Ioannis Papadopoulos, Olga Pearce, Timmie Smith, Nathan Thomas, Xiabing Xu, Nedhal Mourad, Jeremy Vu, Mauro

Bianco, Nancy M. Amato and Lawrence Rauchwerger *The STAPL Parallel Container Framework* In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP), Feb 2011

[5] Ioannis Papadopoulos, Nathan Thomas, Adam Fidel, Nancy M. Amato and Lawrence Rauchwerger *STAPL-RTS: An Application Driven Runtime System* In Proc. ACM Int. Conf. Supercomputing (ICS), pp. 425-434 , Newport Beach, CA, USA, Jun 2015

[6] Steven Saunders and Lawrence Rauchwerger *ARMI: An Adaptive, Platform Independent Communication Library* In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP), pp. 12, San Diego, CA, Jun 2003

[7] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. *KLA: A New Algorithmic Paradigm for Parallel Graph Computations* In Proc. IEEE Int.Conf. on Parallel Architectures and Compilation Techniques (PACT), pp. 27-38, Edmonton, AB, Canada, Aug 2014.

[8] Harshvardhan, Nancy M. Amato, and Lawrence Rauchwerger. *A Hierarchical Approach to Reducing Communication in Parallel Graph Algorithms* In Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP), pp. 285-286 (Poster), San Francisco, CA, USA, Jan 2015.

[9] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. *An Algorithmic Approach to Communication Reduction in Parallel Graph Algorithms* In Proc. IEEE Int.Conf. on Parallel Architectures and Compilation Techniques (PACT), San Francisco, CA, Oct 2015.

[10] Jure Leskovec and Rok Sosič *SNAP: A General-Purpose Network Analysis and Graph-Mining Library* ACM Trans. Intell. Syst. Technol. 8, 1, Article 1 (July 2016), 20 pages