COMPARATIVE ANALYSIS OF DIFFERENT ASYNCHRONOUS MULTITASKING SYSTEMS USING SINGLE SOURCE SHORTEST PATH PROBLEM

by

Prateek Srivastava

A thesis

submitted to the faculty of the University Graduate School

in partial fulfillment of

the requirements for the degree

Master of Science in Computer Science

School of Informatics, Computing and Engineering

Code for this text : https://github.com/prateek22sri/runtime_system_analysis

Indiana University

May 2018

Acknowledgements

This report would have not been completed without the guidance of Professor Thomas Sterling, Dr. Matthew William Anderson and Dr. Maciej Brodowicz.

Special thanks to Alexander Frolov(https://www.dislab.org/~alexfrolov), who provided the implementations of Charm++ and the PBGL's Dijkstra's implementation. He is also a collaborator for this project.

Jesun Feroz helped in understanding HPX5's implementation and Marcin Zalewski helped with the core understanding of the Parallel Boost Graph Library's Delta Stepping implementation.

Contents

Abstract

Graph problems are one of the most important problems used for benchmarking purposes. Among other graph problems, single source shortest path problem(SSSP) is relatively more difficult compared to other graph problems like breadth first search. There are many frameworks or runtime systems that are dedicated to solve such problems. These systems use different architectures, different methods for parallelization, different means for synchronicity and so on. All these parameters responsible for the performance of these systems individually make these comparisons heterogenous. Here the author aims to achieve a comparison for the performance of these runtime systems which will provide a baseline and help understand the innate differences among them. The comparison is done by assuming these runtime systems are individual black boxes with a fixed input and output. The metrics calculated in the process quantize the differences or benchmark each runtime systems in reference to SSSP.

Motivation

Single source shortest path problem is one such problem which does not have a heuristic or a hack which can make the problem a bit easy. Being a difficult problem makes it a necessity to parallelize the problem and consequently interesting for the scientific community. Graph500 also because of similar reasons included this problem as a benchmark as part of Kernel 2 in its code. However, since we have a lot of APIs, libraries, runtime systems etc. to parallelize a code, choosing the means to parallelize a code becomes extremely confusing for the programmer. This paper intends to help mitigate that confusion. This paper will talk about the performance results for a few systems i.e. HPX5, charm++, Parallel boost graph library and the reference implementation of Graph500. The performance results won't prove that a system is better than any other system. It would simply act as a baseline in order to understand why a technique of parallelization acts better for a specific problem which might eventually lead to a better parallelization technique. This paper will be further using the word "system" for these APIs/libraries/runtime systems for the purpose of ease of explanation. In addition to this metric of performance, the decision of choosing a system for parallelization also depends on the ease of programming, level of control, scalability, robustness etc. which is out of scope for this paper.

Introduction

Graph Theory has always been an important part of mathematics and in the field of computing. The evolution of computers starting from single bit manipulations, to vector multiplication, to complex linear algebraic problems, graph problems have been the most difficult to be solved. The systems that perform well in LINPACK benchmarks might not necessarily perform better in graph500 benchmarks. For instance, Sunway TaihuLight ranks first in the LINPACK benchmark's Top500 list but is beaten by the K Computer in Graph500 which ranks tenth in LINPACK. Current trends show that attempts to build newer machines include careful considerations in their graph solving capabilities. In fact, continuum computing architecture is an example of one such architecture which is being designed to give potentially the best results to solve static as well as dynamic graph based problems like breadth first search(BFS), single source shortest path(SSSP).

Additionally, graph problems in general have a wide variety of applications such as scheduling, network routing, global positioning systems like google maps. The current artificial intelligence algorithms also use a graph based sample space for any robot based computing. The probabilistic models are being constructed in the form of directed acyclic graphs in a lot of applications for efficient computations. The current extensive set of graph applications along with the potential set has resulted in the rise of research interest among the scientific community.

The evolution of distributed computing starting from synchronized parallel algorithms to frameworks such as MapReduce leading to asynchronous runtime systems has consistently tried to reduce time to solution and maximize processor utilization while reducing the busy wait. This provides MPI a leverage as it gives the programmer complete control of whether to make

the computation synchronous or asynchronous. However, if not handled correctly, MPI sometimes give bad results. This leads to dedicated asynchronous multi-tasking systems like HPX5, charm++, legion etc. where the runtime system does the work, making the programmer's life a little simple. The programmer in these systems does not have to specify the the parallelism based on number of processes, critical sections, communication mechanism, workload balancing etc. The runtime systems handles all of this by itself making the programmer's life a little simple.

This paper intends to provide a head to head comparison on a problem that is considered to be one of the difficult problems in graph theory i.e. SSSP. Different systems have slight differences in the implementation of the program optimized specifically for the system. However, this paper could be good starting point to make a design decision of which framework/runtime system/library to be used to exploit parallelism in a cluster keeping in mind the tradeoff between scalability, ease of programming and time to solution i.e. strong scaling.

Background

Graph500

Graph500[1] is a benchmark used to guide the design of hardware architectures which are intended for intensive data analytic workload. Graph problems is the core of these analytic workload. The current release of Graph 500 version 3.0.0 introduced SSSP in its benchmark specification. It uses 3 Kernels to benchmark hardware architectures. Kernel 1 uses breadth first search as a computation problem. The second kernel which was introduced only in this release uses single source shortest path and the third kernel is the validation kernel. It uses native MPI for parallelization. It maintains a list of top performing computers which complements the list of Top500 which uses LINPACK benchmark.

HPX5

HPX5[2] is a runtime system developed by Indiana University based on the ParalleX Execution model. HPX5 essentially has a few features that helps it parallelize computation efficiently e.g., active global address space(AGAS), local control objects(LCOs) such as futures, execution based on interdependent lightweight tasks(see figure 1). The concept of locality i.e. work where the data is makes it easier to reduce latency resulting in decreased time to solution.

[1] "Graph 500." https://graph500.org/. Accessed 25 Apr. 2018.
[2] "Advances in HPX Runtime Implementation and Application - Russian ...." http://2015.russianscdays.org/files/talks/Sterling-RSCDays-2015.pdf. Accessed 25 Apr. 2018.

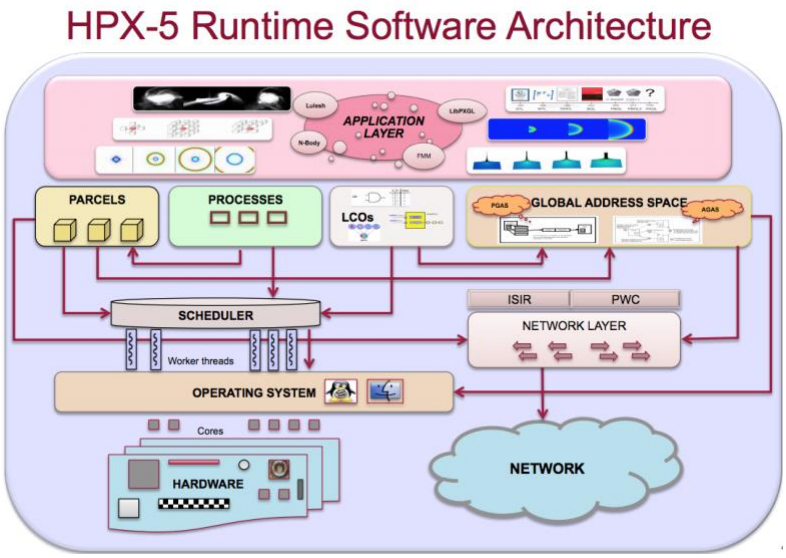

Figure 1. HPX5 Architecture[3]

Charm++

Charm++[4] is a parallel programming paradigm which follows object oriented approach developed by University of Illinois Urbana-Champaign. It relies on asynchronous message passing interface called “Adaptive Message Passing Interface” which is an implementation of MPI on top of charm++ runtime system. The key feature of AMPI is that it implements the MPI ranks as user threads which makes them faster to context switch and also make them easy to migrate between cores and nodes for load balancing or handling failures.

A Charm++ program is essentially a collection of chare objects. These objects communicate with each other using the above mentioned AMPI(see figure 2). Once these communications are received, a process may readily execute, buffer the contents of the message or do nothing at all. This makes the communication between these chare objects non-blocking.

[3] Koniges, Alice, Jayashree Ajay Candadai, Hartmut Kaiser, Kevin Huck, Jeremy Kemp, Thomas Heller, Matthew Anderson et al. *HPX Applications and Performance Adaptation*. No. SAND2015-8999C. Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2015.

[4] "Parallel Programming with Migratable Objects: Charm++ ... - IEEE Xplore." http://ieeexplore.ieee.org/abstract/document/7013040/. Accessed 25 Apr. 2018.

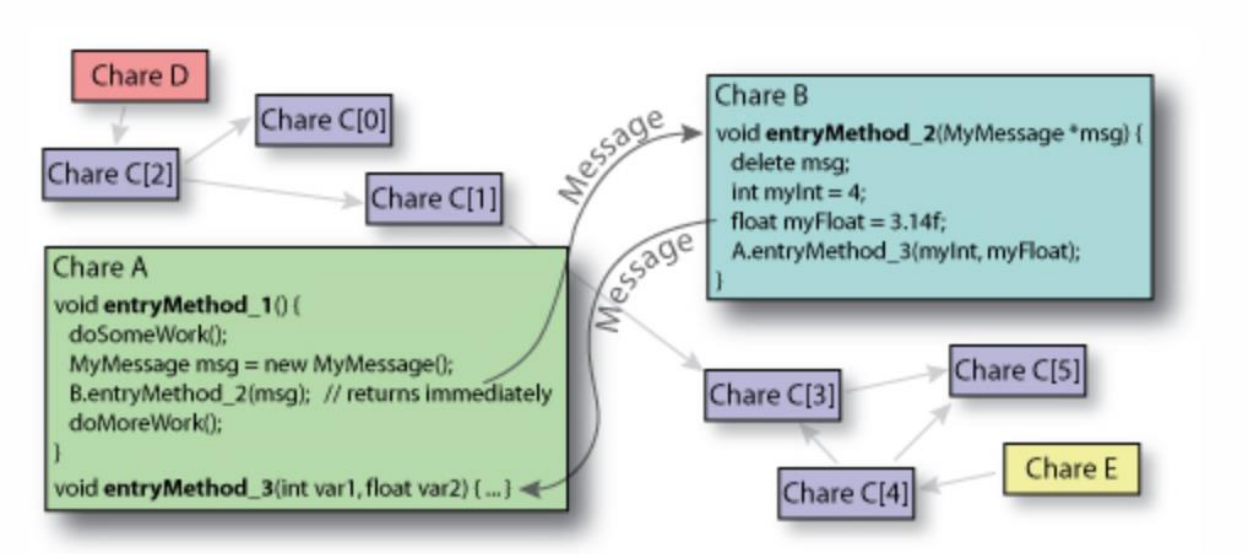

Figure 2. User's view of charm++ application[5]

Parallel Boost Graph Library

The parallel boost graph library[6](PBGL) is a C++ library for parallel and distributed computation. This library is dedicated to solve graph problems using high performance data structures optimized for distributed systems. These distributed graph data structures have a property map which store the properties of that particular graph like weights, colors etc. These components are tied together with the help of process groups and algorithms(see figure 3). The communication mechanism is hidden for the user exposing the interactions of process groups with multiple processes and data structures. The following figure illustrates the architecture for communication among the components.

[5] "Charm++: Tutorial." http://charmplusplus.org/tutorial/CharmConcepts.html. Accessed 30 Apr. 2018.
[6] "An Overview of the Parallel Boost Graph Library ... - Boost C++ Libraries." 31 May. 2009, https://www.boost.org/doc/libs/1_67_0/libs/graph_parallel/doc/html/overview.html. Accessed 25 Apr. 2018.

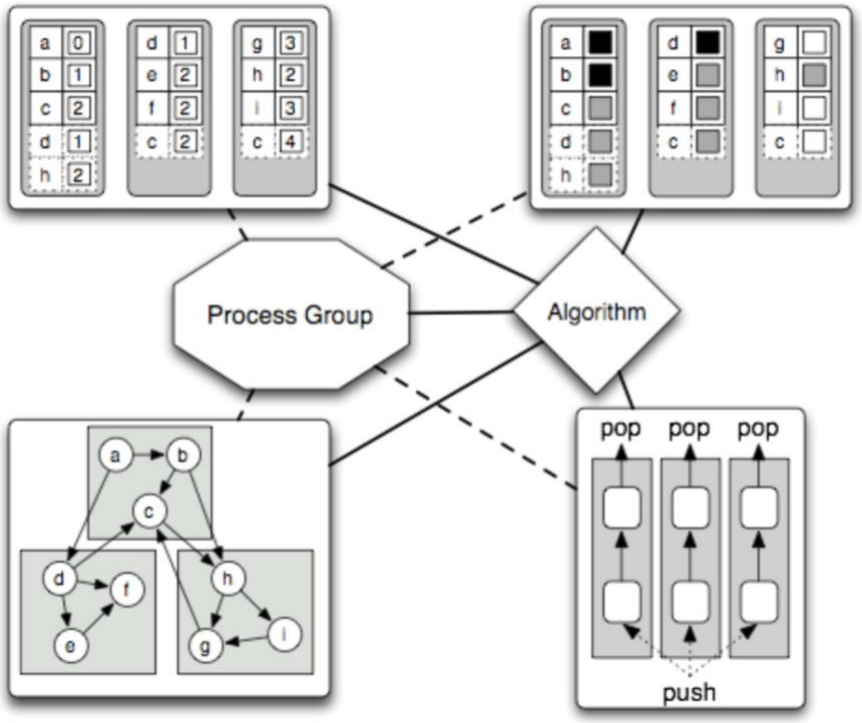

Figure 3. PBGL Architecture[7]

[7] "An Overview of the Parallel Boost Graph Library - 1.67.0." 31 May. 2009, https://www.boost.org/doc/libs/1_67_0/libs/graph_parallel/doc/html/overview.html. Accessed 30 Apr. 2018.

Results

The different graph processing systems compared in this paper are Charm++, parallel boost graph library, Graph500 and HPX5. According to the experiment conducted in this paper, Charm++ was the fastest in terms of time to solution followed by Graph500, then PBGL's Delta-stepping, dijkstra's algorithm and HPX5(see figure 4).

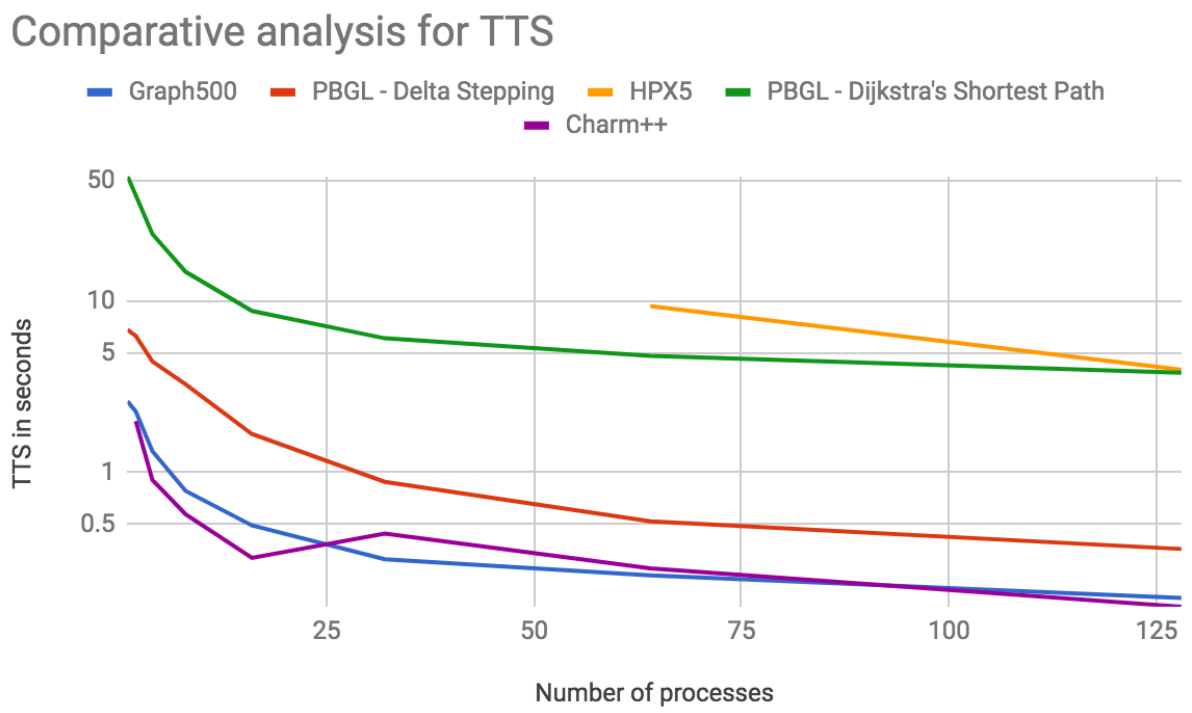

Figure 4. Performance evaluation of different runtime systems

Platform Used: Cray XE6 machine(Big Red II) - CPU nodes

***Note**: The number of processes in the above figure was mapped to individual cores of the machine. Since, there are 32 cores per node, every reading consisting less than or equal to 32 processes is just from a single node being in a shared memory setup while the readings having processes greater than 32 are in a distributed setup divided into multiple nodes.*

The above results show that the fastest system to compute SSSP was charm++. The implementation of charm++ and graph500 were very close in terms of time to solution. PBGL, on the other hand, had two implementations of SSSP. Just by changing the type of algorithm i.e. from Dijkstra's algorithm to delta-stepping algorithm, a huge speed up was gained. HPX5 on the other hand was intensive in memory. The results could not be fetched as the hardware ran out of memory for fewer number of processes.

Future Work

This analysis even though it provides a baseline, but can be still be argued about being fundamentally invalid. Even though, the statistical comparative arguments were same in all the systems i.e. the scale of the graph was 16 which means that the number of vertices in the graph used was always 2^{16} and the edge factor was 500 which means each vertex was always connected to 500 other vertices, the graph generated by the systems were not the same. Graph500 and PBGL implementations used the same graph500 graph generator version 3.0.0. This version had its own SSSP implementation which is why they had weights in their graphs. HPX5 used an older version of Graph500, which did not have weights so the code that implemented it, added random weights making the problem different, there by making the solution different. Charm++ uses RMAT graph generator which makes the graph different and so the solution. PBGL's delta stepping implementation uses the octave version of Graph500 to generate the graph, which works on one random seed as opposed to the other Graph500 graphs which are the c++ implementation which work on 2 seeds finally being merged to one to generate their respective graphs. Thus, even though these results have been statistically been extracted with similar parameters, they are still not completely accurate.

The next steps for the future work of this project would be input the exact same graphs with the exact same random roots averaged out throughout the systems to get the same output solution. The performance metrics thus generated in the process would be an accurate comparative analysis thus, completing the project assuming the system being used is a black box ignoring the heterogeneity in terms of the capabilities of the system i.e. having a leverage by controlling the hardware.

Works Cited

2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS 2015): Melbourne, Australia, 14-17 December 2015. IEEE, 2015.

Sterling, Thomas, Daniel Kogler, Matthew Anderson, & Maciej Brodowicz. "SLOWER: A performance model for Exascale computing." *Supercomputing Frontiers and Innovations* [Online], 1.2 (2014): 42-57. Web. 26 Apr. 2018

Edmonds, Nick, et al. “Single-Source Shortest Paths with The Parallel Boost Graph Library.” *DIMACS Series in Discrete Mathematics And Theoretical Computer Science The Shortest Path Problem*, 2009, pp. 219–248., doi:10.1090/dimacs/074/09.

Kale, Laxmikant V., and Sanjeev Krishnan. "CHARM++: a portable concurrent object oriented system based on C++." *ACM Sigplan Notices*. Vol. 28. No. 10. ACM, 1993.