Efficient Parallel Graph Algorithms in Python

Jonathan Harper† Shoaib Kamil Armando Fox

†Mississippi State University and UC Berkeley

jwh376@mstate.edu, {skamil,fox}@cs.berkeley.edu

Abstract

Domain experts in a variety of fields utilize large-scale graph analysis; however, creating high-performance parallel graph applications currently involves expertise in both graph theory and parallel programming which might not be available to the domain specialist. This project explores methods for bringing efficient parallel performance to graph applications written in Python using selective embedded just-in-time specialization (SEJITS). The Knowledge Discovery Toolbox (KDT) is a tool for analyzing graph data on distributed systems. KDT provides a high-level interface for analysis in Python and graph algorithm building blocks in C++. Users of the KDT have access to operations on matrix and vector elements which are implemented in the efficiency layer. Combination of these operations forms the computational core of many graph applications. A method was developed to extend the KDT with new operators written in Python using a Python SEJITS implementation, Asp.

1. Introduction

Graphs are used to model data from a variety of application areas; biological systems, communication networks, social systems, and atomic structures can all be represented using a graph structure. Experts in a variety of domains are beginning to find analysis of large quantities of graph data necessary. However, efficient parallel graph algorithm implementation requires expertise in graph theory, application programming in an *efficiency level language* (ELL) such as C or C++, and parallel programming. For domain experts, this expertise may not be accessible.

Domain experts commonly learn *productivity-level languages* like Python which stress programmer productivity and succinctness. Case studies have shown that these languages can reduce development time by a factor of 3 to 5 while also decreasing the lines of code needed [6, 11] in comparison to efficiency languages. This makes these languages appropriate for the type of rapid prototyping and debugging domain experts require for exploratory development. However, PLLs generally come with a performance cost compared to efficiency languages. The common utilization of parallel processing on multicore systems, distributed systems and graphics processors makes this performance gap increasingly evident because generally PLLs do not expose hardware specifics. Graph analyses often involve large data sets which require efficient parallel or distributed applications to achieve practical run times.

An ideal solution for graph analysis for domain experts would involve the brevity and rapid prototyping of a PLL combined with the speed and hardware utilization of an ELL.

1.1 SEJITS

Selective embedded just-in-time specialization (SEJITS) [5] uses metaprogramming and introspection features, which are becoming common in modern scripting languages such as Python and Ruby, to combine the productivity of PLLs with the performance and parallel hardware utilization of ELLs. SEJITS involves generating efficiency language code within the interpreter of the scripting language (*just-in-time specialization*). The JIT mechanism is also *selective*, only being utilized when significant performance increases can be realized, and *embedded* into the productivity language, so that the specialized code can take advantage of language features and libraries available in the PLL.

SEJITS involves creation of an embedded domain-specific language (DSEL). A domain-specific language (DSL) is a language which is restricted to a certain problem set, allowing domain-specific optimizations to be applied. However, creating a completely new DSL for your problem domain requires a significant amount of effort parsing, optimizing, and generating code. Additionally, the user of this DSL must learn new syntax specific to the language. A DSEL avoids these problems by being implemented as a subset of the language it is embedded in. For example Asp, a SEJITS framework for Python, implements these DSELs as valid Python syntax. However, certain language constructs may be specialized and optimized for the specific domain. Each specializer (and thus problem domain) will have its own DSEL. A benefit of this is the integration with the existing Python codebase. Code which has no domain-specific optimizations can be run in pure Python.

1.2 The Knowledge Discovery Toolbox

The Knowledge Discovery Toolbox (KDT) seeks to provide domain experts an easy to use package for graph analytics which scales to distributed systems. KDT has a Python interface with a set of high-level, reusable graph operations for domain experts as well as a set of building blocks written in C++ for graph algorithm developers. These graph algorithm building blocks rely on the Combinatorial BLAS [3], an efficiency-level library used for graph analysis and data mining using linear algebra primitives, as their computation engine. Buluç and Madduri [4] show an example of parallel, distributed breadth-first search using linear algebra primitives and semiring operations.

KDT provides a flexible PLL environment for developers as well as an efficient, scalable engine for computation. Other projects approach this problem, but none seem to provide the same range of features. Pregel [10] is a distributed graph API which focuses on the desire to "think like a vertex" and perform graph-centric computation, a primitive that KDT also provides. Other high performance graph libraries include the Parallel Boost Graph Library [9] and the Combinatorial BLAS [3]. However, all of these libraries are designed for use by ELL programmers, and may not be suitable for the domain experts KDT is targeting. The Parallel Boost Graph Library has Python bindings [8], but their development has been discontinued.

Users of KDT have access to operations on matrix or vector elements which are implemented in the efficiency layer. Combination of these operations forms the computational core of many graph applications. KDT provides a feature allowing users to extend functionality by implementing new operators as Python methods. However, testing has shown operators written in Python to be slower than their C++ equivalents by as much as a factor of 80. This slow down is the result of creation and destruction of Python objects for the purpose of performing a relatively small amount of computation. This matrix/vector operation performance loss is the target of our specializer.

2. KDT Specialization

A specializer was developed using Asp to solve the slowdown due to the use of Python objects for user-provided KDT operators. The Python interface to KDT contains wrappers around the efficiency language graph algorithm building blocks generated by the Simplified Wrapper Interface Generator (SWIG) [2]. SWIG provides automated generation of these Python wrappers. However, the Asp infrastructure previously supported only the Boost Python library [1] for run time wrapping of dynamically generated code. As KDT is a fairly large project and this project is experimental, re-wrapping KDT with the Boost Python Library before being sure of significant performance gains was an excessive investment. However, the modular design of Asp allowed a new back-end build and wrapper generation system to be implemented.

SWIG must generate wrappers before the Python application is run, and requires the application to already be compiled ahead of time, unlike the Boost Python Library. This means the specializer for KDT operations requires being run before from the KDT application execution. The new back-end takes generated code, compiles and links it with the existing KDT efficiency level application, generates a SWIG interface file for the generated code and calls SWIG to generate wrappers for the project using a GNU make [7] build system.

3. Results

A KDT operator was written using the specializer created to perform the element-wise binary operation used in the Graph 500 benchmark 1 (http://www.graph500.org/). This algorithm seeks to benchmark performance for distributed graph analysis systems. A graph is generated in the first kernel of the benchmark, and randomly rooted breadth-first search trees are generated for its second kernel.

Users of KDT currently can write operators for their problem using Python. However, due to overhead from creation and destruction of Python objects at runtime, operators in Python run up to 80x slower than built-in C++ equivalents. Figure 1 shows that after adapting SEJITS to work with KDT and writing a specializer for these operators, most of the performance loss from using Python is recovered; however, users still only need to write Python code for their operator. Some performance was lost using the specializer because optimizations for the sparse element-wise multiply performed for Graph500 has not yet been implemented. Future work will include a general method for optimizing operator performance.

4. Conclusion

A specializer was developed using the Asp Python SEJITS framework to improve performance of matrix and vector operators in KDT. To facilitate its use, a back-end for Asp was developed which works with KDT's build system and its wrapper generator, SWIG. This demonstrated Asp's modularity and ability to work well with other projects. There was also a significant speed improvement for the specialized Python operators versus their non-SEJITS Python equivalents. In fact, as Figure 1 shows, the performance of the SEJITS KDT operators was comparable to that of hand-written C++ operators.

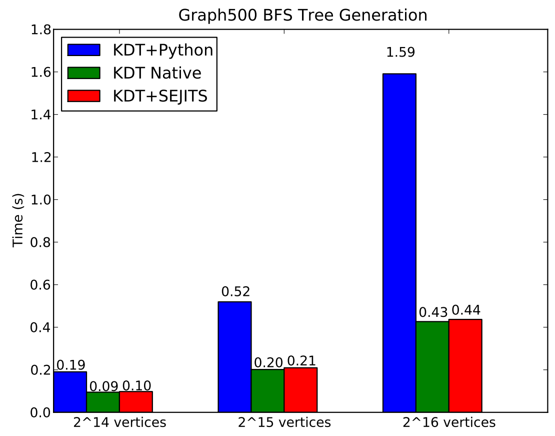

Figure 1. Graph 500 tree generation performance for scale factors 14, 15, and 16 on an Intel Q720 Clarksfield processor (1.60Ghz). SEJITS allows coding new KDT functions in Python while matching the performance of its native C++ functions.

Acknowledgments

This work was performed during the SUPERB-ITS 2011 summer REU funded by the NSF. Thanks to the coordinators of SUPERB-ITS as well as the faculty and graduate student mentors, Armando Fox and Shoaib Kamil.

This work was performed at the UC Berkeley Parallel Computing Laboratory (Par Lab), supported by DARPA (contract #FA8750-10-1-0191) and by the Universal Parallel Computing Research Centers (UPCRC) awards from Microsoft Corp. (Award #024263) and Intel Corp. (Award #024894), with matching funds from the UC Discovery Grant (#DIG07-10227) and additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung.

References

[1] ABRAHAMS, D., AND GROSSE-KUNSTLEVE, R. W. Building Hybrid Systems with Boost . Python. Tech. rep., Boost Consulting, 2003.

[2] BEAZLEY, D. M. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C ++. In *Fourth USENIX Tcl/Tk Workshop* (1996).

[3] BULUÇ, A., AND GILBERT, J. R. The Combinatorial BLAS : Design , Implementation , and Applications. *Science* (2010), 1–35.

[4] BULUÇ, A., AND MADDURI, K. Parallel breadth-first search on distributed memory systems. *CoRR abs/1104.4518* (2011).

[5] CATANZARO, B., KAMIL, S., LEE, Y., ASANOVIĆ, DEMMEL, J., KEUTZER, K., SHALF, J., YELICK, K., AND FOX, A. SEJITS: Getting productivity and performance with selective embedded jit specialization. Tech. rep., Technical Report UCB/EECS-2010-23, EECS Department, University of California, Berkeley, 2010.

[6] CHAVES, J. C., NEHRBASS, J., GUILFOOS, B., GARDINER, J., AHALT, S., KRISHNAMURTHY, A., UNPINGCO, J., CHALKER, A.,

WARNOCK, A., AND SAMSI, S. Octave and Python: High-Level Scripting Languages Productivity and Performance Evaluation. *2006 HPCMP Users Group Conference (HPCMP-UGC'06)* (June 2006), 429–434.

[7] FREE SOFTWARE FOUNDATION. GNU make. `http://www.gnu.org/software/make/`.

[8] GREGOR, D. Boost graph library python bindings. `http://osl.iu.edu/~dgregor/bgl-python`.

[9] GREGOR, D., AND LUMSDAINE, A. The Parallel BGL : A Generic Library for Distributed Graph Computations. In *Parallel Object-Oriented Scientific Computing* (2005).

[10] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel : A System for Large-Scale Graph Processing. Tech. rep., 2010.

[11] PRECHELT, L. An Empirical Comparison of Seven Programmign Languages. *Computing*, October (Mar. 2000), 23–29.