PyGAS: A Partitioned Global Address Space Extension for Python

Michael Driscoll, Amir Kamil, Shoaib Kamil
Computer Science Division
University of California, Berkeley
{*driscoll,kamil,skamil*}*@cs.berkeley.edu*

Yili Zheng, Katherine Yelick
Lawrence Berkeley National Laboratory
{*yzheng,kayelick*}*@lbl.gov*

***Abstract*—High-level, productivity-oriented languages such as Python are becoming increasingly popular in HPC applications as "glue" and prototyping code. The PGAS model offers its own productivity advantages [6], and combining PGAS and Python is a promising approach for rapid development of parallel applications. We discuss the potential benefits and challenges of a PGAS extension to Python, and we present initial performance results from a prototype implementation called PyGAS.**

I. Introduction

Like other high-level languages, Python's concise syntax, interactivity, and dynamic features support development of complex workflows and rapid prototyping of new algorithms. Furthermore, its vast libraries comprise a powerful scientific toolkit, and its low-level API allows for optimization where appropriate. The PGAS model, with its global address space for building distributed data structures and scalable parallelism model, is a natural choice for parallel programming in Python. This short paper explores the major design considerations that may arise when extending Python with a global address space. We present a prototype implementation and discuss its implications for productivity and performance.

We are interested in extending Python for several reasons. First, we aim for interoperability with existing code libraries, thereby enabling their use in a PGAS environment. Second, we hope to lower the programming model's barrier to entry by making as few changes to syntax and semantics as possible. We expect that programmers familiar with Python will be able to introduce PGAS constructs into their programs with relative ease.

II. PyGAS: A PGAS Python Prototype

We have developed a prototype implementation of a PGAS extension to Python, which we call PyGAS. PyGAS implements all PGAS logic in a library and therefore requires no modification to the Python interpreter. The library is built on the GASNet Communication Library [1], so PyGAS programs can target shared memory or distributed memory machines. PyGAS is designed for the single program, multiple data (SPMD) execution model, which has been established as the predominant model for massively parallel programs. We currently run one interpreter per SPMD rank, and one or more threads per interpreter.

The primary PyGAS primitive for representing shared entities is the Proxy object. Proxies are initialized with an actual object, and subsequent operations on the Proxy are intercepted by Python's attribute resolution system, dispatched over the network, and executed on the actual object. This behavior is consistent with Python's duck-typing system: the Proxy 'quacks' like the actual object, so it can be treated as such. For those familiar with UPC, the Proxy can be viewed as a *pointer-to-shared* containing the owner's rank and the address of the actual object within the owner's memory space. To exploit locality, the program can query a Proxy's owner and branch accordingly. Finally, Proxies themselves are objects, so Proxies-to-Proxies can be constructed.

A method call on Proxy is executed on the thread with affinity to the actual object, regardless of which thread called the method. This behavior is a departure from UPC- and Titanium-like "caller computes" semantics, but is necessary in the light of dynamic code generation because the remote method may differ from local versions with the same name.

III. Implementation Challenges

Python was not design for parallelism, so naturally some challenges arise when extending it with PGAS functionality.

A. The Global Interpreter Lock

Python's reference implementation uses a "big lock," called the Global Interpreter Lock (GIL), that must be held during any language-level operations. The GIL inhibits true multithreading, but there are ways to leverage on-chip parallelism. Performance-oriented libraries can be internally multithreaded if their operations are independent of the interpreter. For example, the numerical package Numpy [4] can use a multithreaded BLAS implementation. Alternatively, PGAS extension libraries could be built on Python implementations that don't possess a GIL, such as the JVM-based Jython [2]; however, such implementations offer limited interoperability with mainline packages.

The GIL also complicates the use of Active-Message-based communication libraries like GASNet. When message handlers are invoked upon message receipt, the interrupted thread often holds the big lock, preventing the

handler from safely performing language-level operations. As a workaround, handlers in the PyGAS runtime register secondary functions with the interpreter that are called asynchronously with the lock, albeit with a moderate performance penalty.

B. Automatic Memory Management

Python's automatic memory management system provides significant productivity advantages, but implementing garbage collection schemes efficiently in a distributed environment remains a challenge. Python uses reference counting, so PyGAS must keep accurate tallies of references from both local objects and remote Proxies. We plan to implement a scheme wherein a Proxy would decrement the remote count when it's garbage-collected, at the expense of extra communication.

C. Object serialization

Serialization is a necessary step for communicating arbitrary objects. Currently, PyGAS uses Python's general purpose serialization module `pickle`, which generates a platform-independent object representation. In practice, PyGAS doesn't require such portability because parallel machines often have uniform processor architecture. We expect to achieve moderate speedups by writing a custom serializer that disregards portability concerns.

IV. Performance and Productivity Assessment

Initial results indicate that PyGAS provides substantial productivity advantages while maintaining performance comparable to `mpi4py` [3], the most popular MPI bindings for Python. Figure 1 shows the bandwidth attained during a remote read operation between two Python 2.7.3 interpreters running on separate nodes of NERSC's Carver machine, an IBM iDataPlex machine with 2.67 GHz Intel Nehalem cores and 32 Gb/s Infiniband interconnect links. The performance is weak compared to UPC: PyGAS achieves 10% of the bandwidth seen by a UPC implementation of the same microbenchmark. However, PyGAS achieves more than twice the performance of the MPI bindings for large messages when using the same serialization routine.

We also implemented Cannon's algorithm for distributed matrix multiplication in three ways: in PyGAS with Numpy, in Python with `mpi4py` and Numpy, and in C with MPI and BLAS. Both Python variants required 50% fewer lines of code than the C+MPI variant and the code complexity was considerably reduced. All three variants exhibited roughly the same performance on four nodes of Carver, with the Python variants being 10% slower due to communication overhead.

V. Conclusion

Python is a viable candidate for a PGAS extension, even in light of moderate challenges to overcome in its design and implementation. PyGAS provides substantial productivity advantages at the price of moderate performance loss. Such performance may be tolerable in a rapid prototyping environment, and the savings in development cost may justify the use of PyGAS.

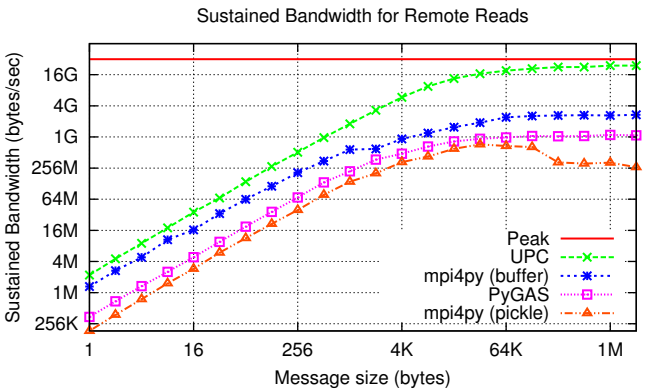

Figure 1. The average bandwidth attained during a remote read operation. Both the PyGAS variant and the "pickle" `mpi4py` variant use the same serialization method. The "buffer" `mpi4py` variant avoids serialization of Python objects with a contiguous memory layout, but this optimization is not applicable to arbitrary objects.

Acknowledgment

Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung.

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.

References

[1] "GASNet Communication System," UC Berkeley, accessed Sept. 20, 2012, http://gasnet.cs.berkeley.edu.

[2] "The Jython Project," accessed Sept. 20, 2012, http://www.jython.org.

[3] "mpi4py - MPI for Python," accessed Sept. 20, 2012, http://packages.python.org/mpi4py/usrman/index.html.

[4] "Scientific Computing Tools For Python – Numpy," accessed Sept. 20, 2012, http://numpy.scipy.org.

[5] "UPC Home Page," accessed Sept. 20, 2012, https://upc-lang.org.

[6] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, Productivity and Performance Using Partitioned Global Address Space Languages, in *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, ser. PASCO 07*. New York, NY, USA: ACM, 2007, pp. 2432.