SAND2021-4774C

Developing a Graph Analytics Code : An Analysis of the Chapel Programming Environment

Richard F. Barrett
Information Operations Center
Sandia National Laboratories
Albuquerque, NM, USA
rfbarre@sandia.gov

Omar Aaziz
Cyber Security and Mission Computing Center
Sandia National Laboratories
Albuquerque, NM, USA
oaaziz@sandia.gov

Jeanine Cook
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
jeacook@sandia.gov

Richard B. Lehoucq
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
rblehou@sandia.gov

Stephen L. Olivier
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
slolivi@sandia.gov

Courtenay T. Vaughan
Center for Computing Research
Sandia National Laboratories
Albuquerque, NM, USA
ctvaugh@sandia.gov

MOTIVATION

We previously reported on our implementation of a Chapel version of a linear algebra based graph hitting time code [1] from an existing MATLAB code. Creating the Chapel implementation was essentially a direct translation, on a Mac laptop, though of course with the manual definition and incorporation of the sparse matrix and vector domains, followed by memory allocation statements. From this experience, the developer of this algorithm, who is not a high performance computing (HPC) expert, now believes that he can perform his algorithm research directly in Chapel, enabling immediate exploration using large data sets on his target HPC machines. This expressiveness, combined with performance portability and robustness, provides a strong case for adoption of the Chapel programming language by the HPC community.

Yet adoption of any new programming language is rightfully challenging [2]. In addition to the productivity characteristics listed above, we believe that the success of a programming language also depends on the programming environment presented to the code developer. In this talk we discuss Chapel's programming environment, within the context of our experiences developing and analyzing code the reference code as well as other applications.

EXPERIENCES

Chapel's global view model presents opportunities as well as challenges in comparison to the local view hierarchical model, such as the ubiquitous C++ with OpenMP and MPI. The global view shields code developers from the details of distributed memory parallel programming environment. Yet meaningful understanding the runtime characteristics of that code requires access to the mapping of this global view to the local components.

At the application level, debugging output is presented as a global view by outputting the state as if the program execution was a single thread, making for significantly easier debugging at the application level. However, we are still limited to line debuggers such as `gdb` and `lldb`.(The primitive yet often useful "print" statement (via `writeln/writef`) also presents a single threaded global view.) A graphical interface, such as provided by the Totalview and Allinea DDT debuggers, would be valuable. It should also provide easy access to individual locales and thread views in order to allow for expert assistance in extracting the performance potential of the target architectures.

As with most parallel processing program execution, the main factor impacting the performance of our applications is inter-locale data movement. Understanding how, where, and when this movement is occurring is critical to realizing the performance potential. Chapel's domains provide reasonable performance for our purposes. However, a stronger understanding of inter-locale data movement may inform custom domains and other techniques for improving performance.

Locale data and information may be extracted via code instrumentation using Chapel task parallel programming functionality. Additionally, Chapel's `CommDiagnostic` module requires the programmer to instrument their code in order to report inter-locale data movement. Both of these methods are useful, but cleaner presentation, without user intervention, is needed. The ChplVis tool presents a two dimensional matrix showing basic data dependences amongst locales, but again, deeper profiling is needed. The now default compiler flag `--cache-remote` aggregates data within a loop on the sender side, significantly improving our code's performance. This aggregate data movement can be reported using CommDiagnostic functionality. We can turn off the flag in order to obtain a finer grained view of data movement, but it is difficult to interpret. A tool could leverage this functionality, and present the information to the user in a more understandable and actionable manner.

The Cray programming environment includes the CrayPAT tool which we've found valuable with local view model

programs. It presents a wealth of performance data, including cache hit percentages, vectorization utilization, arithmetic counts, inter-process communication counts, and node interconnect traffic reports. However, lack of demand led Cray to freeze support of Chapel at version 1.18. Moreover, it is only applicable to Cray computers.

The ParaTools TAU performance toolkit[1] provides powerful up to date capabilities to the HPC community, including individual thread views of line- and loop-level performance counters, with links to the source code, inter-process communication characteristics, and CPU as well as GPU processors. TAU currently provides thread views to Chapel developers, but it is currently limited to a single locale. ParaTools is working on a multi-locale/node implementation. We will present single locale profiling information, and expect to report multi-locale performance in time for the presentation.

HPCToolkit[2] is another valuable profiling tool available to the HPC community. We will illustrate its capabilities. But it currently doesn't include knowledge of Chapel semantics and therefore doesn't provide source code attribution.

We are also interested in GPU execution for our applications, which we've done with other implementations. Once this capability is available in Chapel, we will need to understand the layout of data and computation on it, as well as movement between the GPU and its host (if any).

SUMMARY

Our experiences using the Chapel programming language in implementing some select graph analytics applications provides us with some insights into the programming environment.

presents the code developer with an expressive, portable, performant, and robust syntax and semantics. It also provides important hooks for capturing and reporting debugging and performance data. However, we believe organization and presentation of this data, in user friendly graphical interfaces is crucial to wide spread adoption of the language.

REFERENCES

[1] R. Barrett, J. Cook, S. Olivier, O. Aaziz, C. Jenkins, and C. Vaughan, "Exploring chapel productivity using some graph algorithms," in *Proceedings of the ACM SIGPLAN 7th on Chapel Implementers and Users Workshop*, ser. CHIUW 2020, 2020.

[2] K. Kennedy, C. Koelbel, and H. Zima, "The Rise and Fall of High Performance Fortran: an Historical Object Lesson," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007.

[1] http://www.paratools.com/tau

[2] http://hpctoolkit.org/