A Julia-based Problem Solving Environment for HPC

Youfeng Wu, Todd Anderson, Raj Barik, Chunling Hu, Victor Lee, Hai Liu, Hongbo Rong, Tatiana Shpeisman, Geoff Lowney, Paul Petersen

Intel Corporation

1. Introduction

Traditionally, scripting is glue code that connects system components and library modules to create useful applications. It is considered much more productive than C/C++ programming due to its concise syntax and the Read-Eval-Print-Loop (REPL) for quick feedback [2][8][10]. The superior productivity has propelled scripting languages to become very popular nowadays [14], e.g. in technical computing (Matlab, R, Julia), web design (JavaScript), serve-side scripting (PHP), and general programming (Python, Ruby).

The glue code itself, however, usually runs much slower than C/C++ programs [3], as it is often interpreted/JITed and run serially. With scripting languages becoming more general purpose, significant portion of program execution time is spent in "glue code". Consequently, many of the programs written in scripting languages currently run slowly [9]. In fact, most of script programs for high performance computing (HPC) would be converted into C/Fortran programs by expert programmers to be performant. For the "missing middle" [12] HPC users, we need a Problem Solving Environment (PSE) to achieve both scripting level productivity and significantly better performance than existing scripting language systems.

A PSE provides the computational facilities needed to solve a target class of problems at a high level of abstraction on human terms [4][5]. The high level abstraction contains domain specific interfaces and mathematical formulations. The computational facilities include a host language and tools, the target machine descriptions, and the hand-crafted high performance library building blocks. By exploring both domain specific and target machine knowledge, PSE can achieve both productivity and high performance.

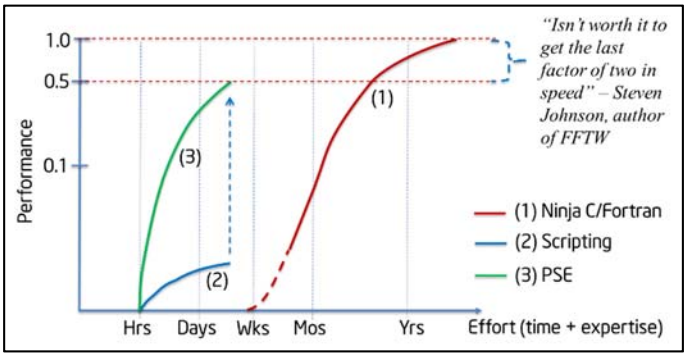

Figure 1. PSE motivation

Figure 1 illustrates the motivation for PSE targeting HPC. Line (1) may represent the traditional approach, where expert programmers spend months or years to come up with a "ninja version" of programs, achieving high performance. Line (2) would be what "average" programmers are achieving using scripting languages. Although a scripting language programmer can obtain a functional program quickly, it usually runs 10x to 100x slower than the ninja version. PSE (line 3) would achieve the same level of productivity as scripting languages, but run much faster than the script program, achieving a significantly portion (e.g. ~50%) of the ninja version performance. People usually would be satisfied with 50% ninja performance if they may get to the solution quickly, as stated by the FFTW author, "FFTW (FFTs) and BLAS libraries (matrix multiplication) take ~100,000 lines [of code] to solve problems that can be implemented in ~15 lines of (slow) code… It usually isn't worth it to get the last factor of two in speed" [15]. Furthermore, a PSE that is both productive and performant would allow users to experiment alternative solutions quickly, which can potentially leads to new algorithms that may perform even better than the ninja version.

2. PSE overview

Our position is that we may augment an existing scripting language to build a PSE for HPC and achieve high performance with scripting level productivity. We choose Julia for its LLVM based infrastructure and good glue code performance [9]. The PSE has high-level abstractions such as vectors, matrices, linear algebra (both dense and sparse), stencils, statistics, and graphs that typical scientists, engineers, and data analysts can use to develop HPC applications productively. The PSE also provides parallelism analysis/optimizations and library description meta-data, so that the abundant parallelism in the high level abstraction can be discovered and mapped to efficient execution on parallel systems. This is in contrast to other approaches that extend scripting languages with explicit parallel constructs and interface, such as PyCuda and PyOpenCL [1], Parallel MATLAB [7], and Hierarchically Tiled Arrays (HTA [6]), etc. Those explicit parallel extensions could be more involved for general users to program them productively.

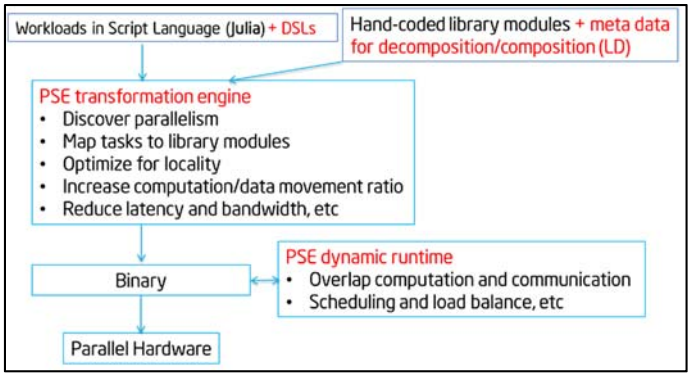

Figure 2. PSE execution flow

Figure 2 shows how our PSE works. It starts with workloads written in serial Julia with domain specific extensions. The PSE has a collection of hand-coded library modules. Each library module includes Library Description (LD) meta-data written in a Library Description Language that tells how the library module would be decomposed and composed with others. The PSE code, together with LD, is then transformed through the PSE transformation engine, which does domain specific optimizations and parallelism discovery, maps or decomposes operations to low level building blocks, and generates code and library invocations. The final code is run on parallel systems, e.g. Intel Xeon and Xeon Phi, orchestrated by PSE's dynamic runtime system.

3. Preliminary evaluation

Our current PSE prototype focuses on HPC workloads with data-parallel and stencil computations. Our experiment runs on an Intel Xeon X5680 system (2-sockets, 6-cores/12 threads each, with 196GB of memory) with two Phi SE10P cards. Since the

multi-threading support in Julia is still under development, we developed and open-source released a "Julia to C" module that converts parallelized PSE code to C and compiles and runs it via C tools [13]. Our performance measurement does not include compilation and JITing overhead.

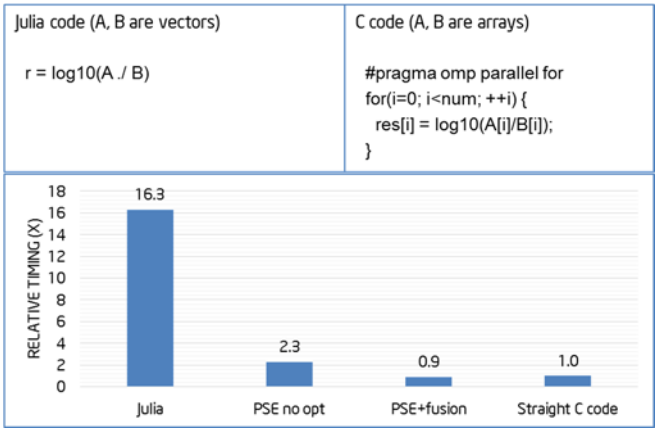

Figure 3. PSE achieves Straight C performance for a data parallel program

Figure 3 shows a data parallel program that does simple vector operations (vector size = 100 million elements) running on the Xeon host. The program written in serial Julia is much simpler than the straightforward C code (with OpenMP pragma for parallelization). Not surprisingly, though, the Julia version is much slower than the Straight C version (16.3x). Our PSE takes advantage of data parallelism and improves the performance by ~7x. Furthermore, PSE optimizations, such as fusion, improves the program speed by another 2.5x, bringing its performance to about the same as the Straight C code. Even if an expert programmer can improve the C code by 2x via ninja tricks, PSE performance would still reach ~50% of the ninja performance.

```
# Iterate to solution
for i in 1:ni
  Apu = Array(Float32, w, h)
  Apv = Array(Float32, w, h)
  runStencil(Apu, Apv, pu, pv, Ix, Iy, :oob_src_zero)
                do Apu, Apv, pu, pv, Ix, Iy
    ix = Ix[0,0]
    iy = Iy[0,0]
    Apu[0,0] = ix * (ix * pu[0,0] + iy * pv[0,0]) + lam *
       (4.0f0 * pu[0,0] - (pu[-1,0] + pu[1,0]+ pu[0,-1] + pu[0,1]))
    Apv[0,0] = iy * (ix * pu[0,0] + iy * pv[0,0]) + lam *
       (4.0f0 * pv[0,0] -  (pv[-1,0] + pv[1,0] + pv[0,-1] + pv[0,1]))
  end
  pTAp   = sum(pu .* Apu) + sum(pv .* Apv)
  α = rsold / pTAp
  xu    += α * pu
  xv    += α * pv
  ru    -= α * Apu
  rv    -= α * Apv
  zu, zv = blockJacobiPrecond(Ix, Iy, ru, rv, lam)
  rsnew  = sum(ru .* zu) + sum(rv .* zv)
  β = rsnew / rsold
  pu     = zu + β * pu
  pv     = zv + β * pv
  rsold  = rsnew
end
```

Figure 4 Inner loop of the Optical Flow in PSE/Julia

For Stencil computation, we select the Optical Flow workload [11], which features a ten-point stencil. Figure 4 shows the inner loop of the Optical Flow in Julia/PSE, where **runStencil** is a domain specific interface for Stencil computation. For comparison, we also developed an idiomatic and an optimized Julia versions, as well as a ninja C version, contributed by a domain expert with ninja programming expertise.

Figure 5 shows the relative timing of four versions of code on an image of size 5184x2912, 100 scales, 44 levels, running on the Xeon host. The idiomatic Julia version is about 115x slower than the ninja C version. We hand-optimize the Julia version (e.g. manually inline Stencil kernels) and improve its performance by 7.4x, although it is still 15.6x slower than ninja C version. Our PSE starts with the version similar to idiomatic Julia, automatically inlines Stencil kernels, aggressively fuses loops, and parallelizes and vectorizes the stencil and other operations. It achieves ~77x improvement over idiomatic Julia and reaches ~67% of ninja C performance.

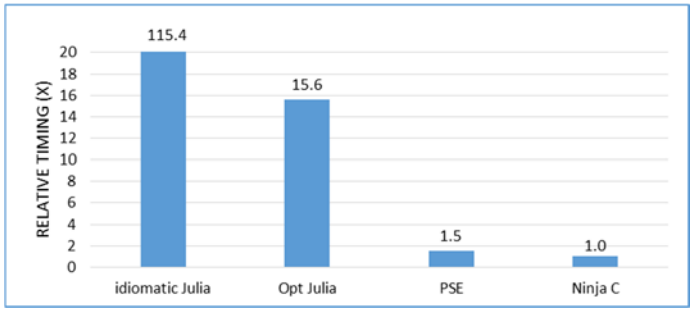

Figure 5. PSE achieves 67% Ninja C performance for Optical Flow

We use HPL (High Performance Linpack) benchmark [16] to demonstrate the benefit of library description and dynamic scheduling in PSE. HPL is the most popular benchmark to rank supercomputers in TOP500, and it spends majority of time in numeric libraries. Figure 6 shows the relative timing of three versions of code for a matrix of size 30K*30K using block size of 1536. The Julia version basically shows the performance of straightforward calls to MKL libraries, such as dgetrf, dtrsm, dlaswp and dgemm, executing on Xeon host-only mode. The Ninja C version is 3x faster than the Julia version, via careful partitioning and scheduling of tasks between Xeon host and Xeon Phi cards. PSE embeds within each MKL library module the ninja level knowledge, such as input/output shapes, min/max threads, schedule hints, tile sizes, and the corresponding performance data for specific systems, which enables PSE to efficiently divide tasks between Xeon host and the 2 Phi cards and schedule them intelligently. PSE also overlaps data transfer between the Xeon host and Phi cards with computation on the Phi cards. As the result, the PSE version reaches 73% of Ninja C performance.

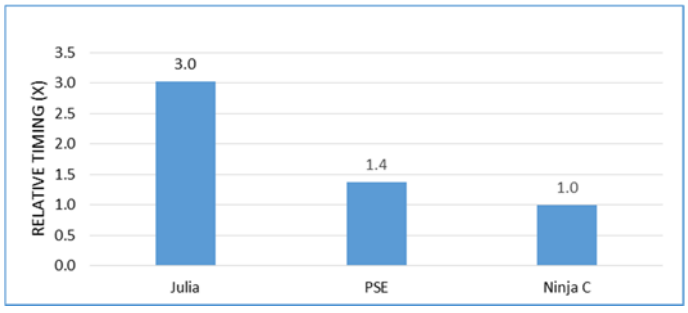

Figure 6. PSE achieves 73% Ninja C performance for HPL

4. Summary and future work

The Julia-based PSE system presented here features three novel components: 1) high level domain specific extensions for HPC applications; 2) library description (LD) for decomposition and composition of low-level high performance library modules; 3) advanced transformation engine and dynamic runtime for

optimization and scheduling. Our preliminary evaluation with data parallel and stencil workloads shows promising results, with the PSE achieving a significant portion of ninja performance without sacrificing the productivity of scripting languages.

As the ongoing efforts, we will add more domain specific interfaces in PSE to handle sparse and graph workloads and extend our experiment from a single node system to multi-nodes. We are also collaborating with Julia Computing, LLC, to add a threading abstraction to Julia language for expert programmers and develop native code generation for Xeon Phi. Besides further improving the PSE performance, we would also like to experimentally qualify the productivity benefit of PSE over traditional programming systems.

Acknowledgements

We would like to thank Neal Glew, Arch Robison, Kiran Pamnany, Pradeep Dubey, Michael Lischke, Intel compiler team, and Julia team for their contributions and collaborations.

References

[1] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, Ahmed Fasih, “PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation”, Parallel Computing, Volume 38, Issue 3, March 2012, Pages 157–174

[2] Andrew Binstock with Peter Hill, “The Comparative Productivity of Programming Languages” Dr Dobb's, http://www.drdobbs.com/jvm/the-comparative-productivity-of-programm/240005881, August 20, 2011

[3] Andrew Funk, Victor Basili, Lorin Hochstein and Jeremy Kepner, “Application of a Development Time Productivity Metric to Parallel Software Development,” in Proceedings of the second international workshop on Software engineering for high performance computing system applications, May 15, 2005

[4] Daniel C. Stanzione Jr., Walter B. Ligon III, “Problem Solving Environment Infrastructure for High Performance Computer Systems”, 15 IPDPS 2000 Workshops, May 1-5, 2000

[5] E. Gallopoulos, E. Houstis, J.R. Rice, ”Problem-solving environments for computational science," IEEE Computational Science & Engineering, 1, 1994, 11--23

[6] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, Mar´ıa J. Garzar´an, David Padua and Christoph von Praun, “Programming for Parallelism and Locality with Hierarchically Tiled Arrays”, PPoPP’06 March 29–31, 2006

[7] J. Kepner, “Parallel Matlab for Multicore and Multinode Computers”, SIAM Press, 2009.

[8] J.K. Ousterhout, “Scripting: Higher Level Programming for the 21st Century,” Computer, Mar. 1998, pp. 23-30.

[9] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman, “Julia: A Fast Dynamic Language for Technical Computing,” CoRR, 2012

[10] Lutz Prechelt. “An Empirical Comparison of Seven Programming Languages”. Computer 33, 10 (October 2000).

[11] Michael Anderson, Forrest Iandola and Kurt Keutzer, “Quantifying the Energy Efficiency of Object Recognition and Optical Flow”, UC Berkeley Tech Report (EECS-2014-22),

[12] Sharan Kalwani, “On Democratizing HPC: Addressing the Missing Middle Lobbying for a new paradigm”, Dec. 2011.

[13] Julia to C: https://github.com/IntelLabs/julia/tree/j2c

[14] Stephen Cass, “Top 10 Programming Languages,” IEEE Spectrum, July 2014.

[15] http://www.slideshare.net/acidflask/julia-compilercommunity

[16] http://www.netlib.org/benchmark/hpl/