pap747 Reviews Details

Submission: Ace: An Actively-Typed Language and Compilation Environment for High-Performance Computing
Contributors: Aldrich, Omar

Summary of reviews of pap747s2

Reviewer

Reviewer 1

Reviewer 2

Reviewer 3

Reviewer 4

Commitee Comments jump
Author Rebuttal jump

Review of pap747s2 by Reviewer 1

top

Summary and High Level Discussion

The authors present Ace, a python-based code-generation environment for programming HPC systems. Ace consists of a Python-based, compilation interface that revolves around the concept of generic programming via programmatic manipulation of first-class types and environments. Ace is extensible through a common type-inference and code generation interface. "Backend" libraries are provided that group types, interfaces, and environments for a code generation context (e.g., OpenCL), and these backends are targeted explicitly by application developer code. The authors' stated contributions include (1) a novel, comprehensive set of design criteria for HPC languages, (2) a design methodology designed from these criteria, and (3) a novel mechanism (i.e., Ace itself) developed via the aforesaid methodology that satisfies the design criteria. This evaluation is mainly laid out in terms of these explicit contributions.

First, however, it should be noted that this paper is well written with helpful examples. Its structure is somewhat confusing, but reasonably well done for an initial draft submission. It is ambitious but seems to be lacking in its analysis and attribution of prior work--though not necessarily in a fatal way---as indicated in the text below.

At a high level, I feel that their "novel" design criteria is (1) poorly defined, (2) arbitrary, and (3) of indeterminate novelty. In particular, they acknowledge that "ease-of-use" is already a first-class criteria in language design but introduce "continuity, extensibility, and interoperability" as additional, and thus presumably disjoint, first-class design criteria as their novel contribution. It is difficult to see how these three criteria (as loosely defined in the introduction) can considered be anything other than extensions to ease-of-use, though ease-of-use has no formal definition other than a single, brief reference to "conciseness." Even

given some suitable definition for ease-of-use that suggests that the inclusion of continuity, extensibility, and interoperability as first-class design criteria is valuable, no argument is given as to why these three properties are either necessary or sufficient for successful HPC language design. Furthermore, no citations appear to be provided for these three criteria, where a simple literature search quickly pulls out numerous potential

references featuring these terms prominently. Finally, other than numerous references to their obvious disdain for template metaprogramming in C++, there is no treatment of how previous attempts at language design for HPC have addressed their novel set of design criteria. Their related work section does not contain any of the seven words presented as "design criteria," nor does the text contain a characterization of the success or failure of other proposed HPC languages---UPC, Chapel, X10, Fortress, etc.---in these terms.

The claim that Ace is the result of system design centered on this more comprehensive set of design criteria is difficult to judge. What is clear is that the discussion of the design of Ace attributes various Ace properties to various design criteria. Given how I feel about the design criteria themselves this is neither a plus nor a minus, though I am tempted to conclude that the design criteria has been distilled from the experience of developing Ace, as opposed to the alternative claim. This impression is possibly an artifact of the discussion's structure, which is centered around the properties of the Ace environment itself, rather than the novel criteria or supposed design methodology.

Finally is Ace itself. I appreciate Ace's goals, and agree with much of the authors' analysis of the situation in HPC. Python has become a de facto standard for a certain kind of HPC user, and its performance---particularly as systems scale up and become more heterogeneous---is a problem facing the HPC community. Techniques and systems that allow programmers familiar with Python to leverage their knowledge to become more productive are welcome. Furthermore I am impressed with the amount of effort that appears to have been put into providing a stable, usable system. As such, "I am not opposed to Acceptance" of this work, with suitable revision, on these grounds, however I feel that the paper, as written, will be of little practical value.

I am disappointed that the large amount of ineffective language-design text subtracts from both Section 3.3, where a detailed presentation of even one of the conjectured use cases would be a compelling argument in favor of Ace, and Section 5, where the sole case study leaves much to be desired in terms of details of the original Brian framework, the C++ alternative, and the Ace solution.

I appreciate Ace's choice of metraprogramming and template specialization as a mechanism for providing the benefits of static typing to Python programmers. To my knowledge, this is a novel approach, though it is unclear that the resulting "language" can continue to be considered Python. As they state, Python is the scripting language for Ace, rather than Ace being an implementation of Python. At least a brief description of previous attempts at typing and compiling Python would be appropriate (e.g., RPython, PyPy) and is completely lacking, as well as a discussion of other uses of Python as a scripting language for lower-level code generation (e.g., CorePy).

I am slightly concerned that Ace requires static backend bindings. It seems obvious that many generic functions do not depend on the peculiarities of the backends themselves (e.g., "add5" is trivial where "map" depends on ace.OpenCL's gid). It seems that abstracting across backends would be a welcome technique in many cases, and a frustrating source of manual code cloning if unavailable.

Comments for Rebuttal

You acknowledge that "ease-of-use" is already a first-class criteria in language design but introduce "continuity, extensibility, and interoperability" as additional, and thus presumably disjoint, first-class design criteria as a novel contribution. A reasonable interpretation of these three properties could be that they are simply components of ease-of-use. Can you elaborate on the distinction you are attempting to make?

You suggest that the (now) seven design criteria presented here are (in some difficult-to-define sense) necessary and sufficient for HPC programming language design---or at least more necessary than the previous four. Were there other criteria that you evaluated for first-class treatment, and why were they rejected if you

did?

Detailed Comments for Authors

The authors present Ace, a python-based code-generation environment for programming HPC systems. Their stated contributions include (1) a novel, comprehensive set of design criteria for HPC languages, (2) a design methodology designed from these criteria, and (3) a novel mechanism (i.e., Ace itself) developed via the aforesaid methodology that satisfies the design criteria. I read and reviewed the paper in terms of these claimed contributions.

Generally, this paper is fairly well written with helpful examples. I find the structure somewhat confusing (e.g., a great deal is made of the novel design criteria, and the claim is made that Ace follows from such criteria, but the treatment of Ace is organized by Ace properties, rather than by the design criteria themselves). This paper is clearly ambitious, but seems to be lacking in its analysis and attribution of prior work. For example, there are no citations for the three additional design criteria expressed here, but a simple literature search results in any number of citations the prominently feature each term. In addition, there is no mention of recent attempts at HPC languages including X10 or Fortress (which shares many of the design goals presented here), and Chapel and UPC are simply mentioned in the context of PGAS.

Overall, I felt that the novel design criteria is (1) poorly defined, (2) arbitrary, and (3) of indeterminate novelty. In particular, even given some suitable definition for ease-of-use that suggests that the inclusion of continuity, extensibility, and interoperability as first-class design criteria is valuable (see rebuttal), no argument are examples are given as to why these three properties are either necessary or sufficient for successful HPC language design (also see rebuttal).

The claim that Ace is the result of system design centered on this more comprehensive set of design criteria is difficult to judge. What is clear is that the discussion of the design of Ace attributes various Ace properties to various design criteria. I am tempted to conclude that the design criteria has been distilled from the experience of developing Ace, as opposed to the alternative claim. This impression is possibly an artifact of the discussion's structure, which is centered around the properties of the Ace environment itself, rather than the novel criteria or supposed design methodology.

Finally is Ace itself. I appreciate Ace's goals, and agree with much of the authors' analysis of the situation in HPC. Python has become a de facto standard for a certain kind of HPC user, and its performance---particularly as systems scale up and become more heterogeneous---is a problem facing the HPC community. Techniques and systems that allow programmers familiar with Python to leverage their knowledge to become more productive are welcome. Furthermore I am impressed with the amount of effort that appears to have been put into providing a stable, usable system.

I am, however, disappointed that the large amount of ineffective language-design text subtracts from both Section 3.3, where a detailed presentation of even one of the conjectured use cases would be a compelling argument in favor of Ace, and Section 5, where the sole case study leaves much to be desired in terms of details of the original Brian framework, the C++ alternative, and the Ace solution. As an interested reader I have all sorts of questions that remain unanswered (but are not suitable to ask as rebuttal questions).

Is the Brian framework designed for CPU-only execution? Were explicit GPU-based binding solutions (i.e., using the oft-mentioned numpy/pyopencl modules directly) evaluated? How much of the Brian framework could be leveraged in designing the Ace solution? Was Ace viable as an incremental migration path, or was it necessary to write the entire cl.egans library before achieving any utility? How long did the port to Ace take, relative to the C++/CUDA port? How much of this was due to unfamiliarity with C++? Is the cl.egans module typically used through acec, or is it meant to be used and instantiated through implicit Python interaction? Given ace.OpenCL, how much low-level Ace code was required to implement cl.egans (in terms of type inference and transformation code)?

I'm particularly interested in the last two questions, as it seems unlikely that the HPC Python programmers that Ace targets will want to learn Ace to the extent that they can implement their own backends, but with a sufficient availability of complete backend modules, and implicit use from Python, I can see the potential for widespread adoption. More details about the memory management benefits alluded to would be welcome as well.

I appreciate Ace's choice of metraprogramming and template specialization as a mechanism for providing the benefits of static typing to Python programmers. At least a brief description of previous attempts at typing and compiling Python would be appropriate (e.g., RPython, PyPy), as well as a discussion of other uses of Python as a scripting language for lower-level code generation (e.g., CorePy).

I am slightly concerned that Ace requires static backend bindings. It seems obvious that many generic functions do not depend on the peculiarities of the backends themselves (e.g., add5 is trivial where map

functions do not depend on the peculiarities of the backends themselves (e.g., adds is trivial where map depends on ace.OpenCL's gid). It seems that abstracting across backends would be a welcome technique in many cases, and a frustrating source of manual code cloning if unavailable.

Review of pap747s2 by Reviewer 2

top

Summary and High Level Discussion

Paper presents Ace, a new programming language intended for end users (scientists) in the HPC community. Ace is built on Python, incorporates an explicit phase distinction (between compile-time and run-time), and uses static typing. Programs are presented, built on top of the internalization of OpenCL as a library. The authors argue (rather sweepingly!) that new languages have found it difficult to gain a foothold in HPC because they do not support "continuity", "extensibility" and "interoperability".

Comments for Rebuttal

The age-old question: If Ace is the answer, what is the question? Please explain in concrete terms what specific, actual, tangible problems you believe practicing scientists are facing that will be addressed by Ace. e.g. will ACE help address heterogeneity in intra-node programming in a principled way? Will ACE help unify an intra-node programming model with an inter-node programming model? Help support interoperability between MPI / PGAS? Provide a framework for supporting the development of domain specific languages (e.g. for stencil-based computations, adaptive mesh refinement etc).

Detailed Comments for Authors

The paper has a lot of interesting technical content and definitely represents an interesting viewpoint. I question, though, whether it is appropriate for SC in its present form. It appears more suited for a programming languages conference -- e.g. SPLASH -- given its focus on language design issues (phase design, AT&T etc) that are rather abstract and applicable to a lot of domains other than HPC.

Review of pap747s2 by Reviewer 3

top

Summary and High Level Discussion

This paper presents a compile-time, meta-programming system based on Python. The work focuses on providing continuity, extensibility, and interoperability with the approach -- all points that the authors feel is an important reason for the lack of adoption for novel approaches to programming in HPC. The majority of the paper leverages OpenCL as a building block/target for code generation.

The ACE system builds upon a type-based approach to meta-programming

Comments for Rebuttal

One of the paper's fundamental arguments is for continuity and interoperability yet I think the paper only makes scratches the surface of this aspect and doesn't really provide an adequate exploration of the details to provide a credible story (the motivations and approach are reasonable but the details are missing).

These are clearly two very important aspects but I feel the paper misses the full scope by restricting the solution space to Python. In particular, I would say interoperability and some form of continuity between Python, ACE and other languages (C, C++, FORTRAN, etc.) is a critical aspect of what is needed to novel approaches to succeed. While it is possible to package other code bases into Python

Detailed Comments for Authors

This work is very similar to a recent work that will appear in PLDI '13:

Terra: A Multi-Stage Language for High-Performance Computing Z. DeVito, J. Hegarty, A. Aiken, P.

Hanrahan, and J. Vitek

Although based on Lua, it is worthwhile to compare/contrast with your work with this effort. This is a recent effort so it is likely you missed it given the timing of the SC submission deadline and the availability of the Terra paper.

A significant weakness for this paper is a presentation of very high-level use cases with very little more than a simple text explanation stating ACE is applicable to the specific topic area. Dropping the number of topics here and providing a few solid working examples would have made the paper much stronger; as it stands it simply seems to be much more of a discussion about what could be future work. In particular, the point on interoperability (a critical point of the paper) state it would be possible to do something but doesn't show this in practice, or fully describe the entire scenario. Another example is from the details regarding functional parallelism where the text is not succinct (i.e. are you thinking strict or non-strict here?).

Furthermore, the approach taken in the case study seems to go against your overall goals by rewriting an entire application over -- granted the metaprogramming features are valuable, but the overall approach goes against some of your initial arguments in the paper.

Finally, it is not clear that this is best suited as a SC paper -- the work has some very good potential but I think it needs to be strengthened to meet the goals of SC. More complex, larger scale applications, and a detailed handling of interoperability and other aspects related to adoption of novel approaches would really have to be significantly improved to reach the bar. .

Review of pap747s2 by Reviewer 4

top

Summary and High Level Discussion

This paper described a new programming language, "Ace", for HPC. Ace is a Python-like language with a translation target of OpenCL or other low-level langauges. Ace lets users to annotate the types of a Python program via library extensions and then it translates the high-level Python program into a statically-typed low-level program in OpenCL. Type propagation and inference of Ace are discussed. However, application examples and performance study for supercomputing are very limited .

I completely agree that the idea of using high-level languages for HPC is crucial and static typing is necessary to get performance and check errors. I can imagine that Ace would become the game changer in the

community some day. However, what makes me stumble frequently in reading this paper is that the flow of the paper appears to be discontinuous. For example, I have to jump to later sections to find some keywords before I can go back to previous sections to understand the examples. In addition, I don't think it is necessary to "sell" the language by referring to the "design criteria" throughout the paper. A brief discussion of the design criteria in the introduction would be enough for me to buy it and I think the space can be better used to describe the the language itself and the details of the implementation.

Comments for Rebuttal

None

Detailed Comments for Authors

1) This paper listed many examples in Ace without first defining the syntax and the semantics, which causes ambiguities when comprehending the examples. It mentioned about extending Python but it was unclear what the extensions were exactly.

2) The question in the paper "can one develop a statically-typed language with the same low-level memory and execution model of C but syntactic overhead comparable to a high-level scripting language?" is ill-posed because the overheads of a high-level scripting language (e.g., Python) is obviously higher than a lower-level statically-typed language (e.g., C) -- why would anyone want to have the overheads as high as Python?

3) There is only one application and no performance study for scalability, which make it very difficult to

3) There is only one application and no performance study for scalability, which make it very difficult to evaluate the productivity or the performance of the language in the context of supercomputing.

I would suggest this paper to be revised with the following objectives:

1) Describe the key features, syntax and semantics of the language

2) Clearly explain how a program written in the language gets translated to the low-level targets

3) Show several real applications and the performance results at scale

4) Compare the language with other existing HPC languages/programming systems, e.g., X10 and Chapel, or Julia.

Committee Comments to Authors top

None

Authors' Rebuttal of Reviews top

submit rebuttal
