Programming Systems on the Road to Exascale Computing

Cyrus Omar

Carnegie Mellon University

Jeffrey Vetter

Oak Ridge National Laboratory

ABSTRACT

From the perspective of computational scientists, programming systems are the most visible component of HPC systems and they serve a critical role in enabling HPC architectures that are high-performance, energy-efficient, scalable, robust, and productive. However, systems designed to achieve these goals must handle increasing architectural complexity associated with hierarchical parallelism, the inertia associated with investments in legacy software and platforms and a variety of other design requirements. Many of these challenges are already emerging in today's multicore and heterogeneous computing systems and recent reports conclude that the road to Exascale Computing will require addressing these challenges even more directly. In this survey, we expand upon the design criteria that constrain programming systems for high-performance computing and highlight some emerging approaches, including recent research on general-purpose computing on graphics processors.

1. Introduction

Computer-aided simulations and data analysis techniques have transformed science and engineering. Surveys show that scientists and engineers now spend more than 40% of their time writing software (1; 2). Most of this software targets conventional desktop hardware, but about 20% of scientists also target either local clusters or supercomputers for more numerically-intensive computations (2), a number that continues to grow as scientists face ever-larger datasets and increasingly complex systems, such as those studied in the biological, environmental and social sciences.

Historically, scientists needing better performance simply purchased new hardware. This behavior was justified by appealing to Moore's Law, which has accurately predicted a doubling in the transistor count of integrated circuits every 18 months since the invention of the transistor in 1958 (3). Although Moore's Law remains valid, it is projected that by the end of this decade, the doubling period will increase substantially, due to fundamental physical limitations. More problematic, however, is that an assumed corollary of Moore's Law – that *clock speeds* will also double every 18 months – is no longer valid. Although transistor counts have continued to increase, clock speeds have remained flat for several years. It is now widely acknowledged that further performance gains will require that scientists make effective use of parallel computation.

Modern high-performance computing systems provide substantial hardware support for parallel computation. A single node in a cluster or a single workstation may contain several general-purpose multi-core processors as well as one or more co-processors, such as a GPU, containing hundreds of specialized compute units. Modern clusters in turn contain many such nodes – the largest supercomputers today feature hundreds of thousands of nodes. These machines boast peak performance on the order of 10^{15} floating point operations per second (1 petaFLOPS). By the end of the decade, researchers hope to achieve a further thousand-fold increase in peak performance, a goal termed the *exascale initiative* (4). At this scale, power consumption, hardware reliability, data movement and storage infrastructure become critical

issues. Comprehensively addressing these issues has emerged as one of the grand challenges in modern computing, and is a major focus of ongoing research spanning a variety of disciplines, including computer science, computer architecture, applied mathematics and a number of application domains in science and engineering.

In an ideal world, programmers would simply continue to write programs as they have in the past, relying on a compiler to optimize them for execution on a variety of hardware configurations at high speed. Unfortunately, such heroic compilers are likely impractical. Many problems central to automatic parallelization have been shown to be NP-Hard, or require additional information that can't be extracted from a program directly, so it is unlikely that fast, exact, fully-automatic techniques will emerge. Instead, optimization procedures must increasingly rely on heuristics, profiling data and domain-specific knowledge. Such so-called 'fuzzy' approaches tend to require significant human involvement, meaning that programmers who hope to make full use of modern high-performance computing hardware must deal with complex issues beyond those they have faced in programming sequential hardware in the past. Recognition of this issue has led to calls to develop and improve the programming systems that developers rely on for assistance with these difficult issues. We use the term *programming systems* to encompass a broad range of tools , including operating system APIs, programming languages, abstractions, libraries, compilers, editing environments, verification tools, documentation tools and so on.

The ultimate goal of this line of research is to maximize the end-to-end *productivity* of practitioners, measured by the time taken by all aspects of the process leading to a desired scientific result. In taking this view – with its focus on human aspects of software development and design in addition to algorithmic and architectural considerations – it is important to first identify and characterize the developer communities who will use these tools. Researchers typically classify software developers as either *end-user developers* or *professional developers*. End-user developers are those who have little formal training related to the tools that they use. These tools, as a consequence, tend to be relatively simple (spreadsheets, for

example). Professional developers, on the other hand, have explicit training or extensive experience with software engineering practices and software development tools and techniques. Unfortunately, scientists and engineers do not cleanly fall into either of these groups. Although formal training in software development is rare (5), scientists and engineers are more technically literate and demanding than typical end-users. For this reason, researchers now define a third group, *professional end-user developers* (6), to describe "people working in highly technical, knowledge rich professions, such as financial mathematicians, scientists and engineers, who develop their own software in order to advance their own professional goals."

In this survey, we examine the needs of professional end-user developers who currently work with, or will in the future need to work with, high-performance computing systems, including future exascale machines. As we do so, we describe both existing and emerging approaches, noting where additional research and development may be needed. We hope that the reader will emerge with a clear view of the state of the art and develop a clear, high-level understanding of the significant challenges remaining as we move toward future high-performance, high-productivity computing systems and applications.

2. System Software

Systems software is the portion of the programming system typically associated with the operating system or low-level runtime environment. It is responsible for interacting with hardware resources, usually in a relatively transparent way to the user. This category includes operating system APIs, file systems and I/O libraries, and systems management software (4).

a. Operating Systems

Today, some flavor of Linux is the dominant choice in HPC machines. Variants of UNIX, such as AIX, and lightweight kernel operating systems such as Compute Node Kernel (CNK) are also used by some vendors (notably, IBM and Cray) (7). Operating system APIs are the most primitive abstractions available to programmers, so small choices (e.g. in thread scheduling) can have a large impact on performance. The choices made to accomodate conventional desktop and server workloads may not be ideal in an HPC context, so there has been considerable interest in developing HPC-specific operating systems. Exascale machines introduce additional complications, because resource management will require consideration of power budgets, and hardware reliability issues will become increasingly prominent. In some cases, collective decisions may need to be made based on information integrated across a cluster, further complicating matters.

Researchers differ about whether the simpler, more specialized lightweight kernel (LWK) operating systems dedicated to high-performance computing applications are better suited to exascale applications, outweighing the flexibility and broad developer base of systems based on a general-purpose operating system like Linux. In either case, however, there is a need for improved APIs for memory and thread management, performance monitoring and debugging, energy management and access to specialized hardware. To evaluate these APIs at scale before the associated hardware is available, whole-system simulation techniques must also be developed. These are all active research areas.

b. File Systems and I/O Libraries

Large, distributed file systems require specialized support at the systems level. A number of different high-performance file systems are in wide use today, including Lustre, GPFS and NFS. Together with I/O libraries such as netCDF, HDF5 and MPI-IO, these systems form the basis for data storage and retrieval on high-performance machines today (7).

Unfortunately, it is not clear that these systems can continue to scale without serious changes. A major component of the overall energy budget of an exascale machine will be consumed by data movement. Today's systems are based on traditional file systems and continue to offer strong guarantees about conflicts, snchronization and coherence. I/O is also generally thought of as an activity that occurs periodically (e.g. during checkpointing) or at the end of a simulation, rather than as an activity that occurs on an ongoing basis throughout a computation. These constraining requirements and assumptions make scaling a challenge, even as new storage architectures and devices emerge, such as solid state disks. Fortunately, many applications do not require such strong guarantees in some cases. As more applications interleave data analysis with computation, relieving a major burden on I/O systems, they can be targeted to more specialized tasks such as checkpointing, improving their performance. As such, scalable, purpose-driven, hardware-aware I/O systems that are more tightly integrated into the programming language and environment are a major focus of ongoing research.

c. Systems Management

High-performance computing systems require additional management tools to ensure that needed resources are available and divided amongst users fairly and securely. For example, *batch systems* allow users to request jobs on shared machines, and schedule these jobs in an equitable way on available resources. Popular choices for batch systems include Torque, MOAB, LoadLeveler and SLURM. A number of other tools are also used by systems administrators to maintain the cluster's software systems, monitor hardware and ensure long-term data integrity. Although there are fewer scaling challenges with this aspect of the software stack than with some of the others, these tools interact with several aspects of both the software stack and the hardware, and must be updated as these aspects of the high-performance computing systems change on the road to exascale.

3. Programming Environments

The programming environment consists of the software that developers interact with directly to develop and debug programs. This category centers around programming languages and includes parallel programming abstractions, domain-specific abstractions, runtime systems, compilers, debuggers, performance analysis tools, data analysis and visualization tools, and editors (4).

Scientists and engineers generally prefer high-level scripting languages like MATLAB, Python, R and Perl (8), particularly those bundled with powerful domain-specific libraries. These languages are typically interpreted rather than compiled and generally lack static type systems, relying instead on run-time (or "dynamic") type lookup and error handling. Although this may increase the flexibility of the language, these features also negatively impact performance and can lead to unexpected run-time errors that are generally impossible to rule out statically. This is particularly severe given that rigorous software testing practices are also quite rare in these domains (8). When running at scale, a run-time error can be very costly, so better verification and testing of scientific code will be necessary.

Along code paths where performance appears to be a bottleneck, developers today turn to statically-typed, though still unsafe, low-level languages like C and Fortran (9). These languages give users explicit control over data layout and heap allocation and require explicit type annotations on all variables. Although this makes writing programs more tedious which can increase the number of logic errors a programmer makes, it can also potentially improve performance, particularly if the programmer understands the target architecture well (e.g. cache behavior). While a sequential reimplementation of a critical code path using a low-level language will generally produce a modest speedup, more significant speedups on modern hardware require parallelizing over many processor cores and, on massively-parallel machines, many interconnected nodes. Despite well-known difficulties, low-level, unsafe approaches that use a form of shared-memory multithreading (e.g. pthreads, OpenMP) paired with explicit message passing between nodes (typically using MPI) remain widely used for these

tasks as well (9; 10).

Although researchers often propose higher-level language features and new parallel programming abstractions that aim to strike a better balance between raw performance, productivity, code portability and verifiability (see below), end-users remain skeptical of new approaches. This viewpoint was perhaps most succinctly expressed by a participant interviewed in a recent study by Basili et al. (10), who stated "I hate MPI, I hate C++. [But] if I had to choose again, I would probably choose the same." Although this sentiment is easy to dismiss as paradoxical, we believe that it demands direct examination by those in the research community working to advance the practice of scientific and high-performance computing.

As in many areas of design, it can be difficult to objectively evaluate the merit of language and tool designs. To better support such evaluations, researchers in design disciplines typically develop a set of high-level design criteria that serve as a rubric for evaluating and guiding their design efforts. In programming language design, particularly for scientific and high-performance computing, there have been few concerted efforts to develop a coherent set of design criteria that capture the needs of the targeted developer communities. Similarly, there have been few treatments of adoption criteria for new languages and abstractions in practice, an important issue given the slow rates of adoption today. As such, we organize this section around a set of design and adoption criteria for new languages and abstractions based on prior empirical studies of this class of *professional end-user developer*, as well as our observations of characteristics common to successful projects in the past.

a. Design Criteria

We define design criteria as aspects of an abstraction or language's specification, rather than its implementation, that help developers express their intent naturally and correctly.

1) Concise, Familiar and Readable Syntax

Despite a considerable amount of evidence pointing toward the value of a well-designed syntax, particularly for end-users in specialized domains (11), the issue is sometimes marginalized. It is likely the case, however, that some libraries and languages experience low adoption due in part due to the use of a verbose, unfamiliar or unreadable syntactic style.

A simple and concise syntax is common to most of the high-level languages that are widely used in scientific computing. Cordy identifies the principle of conciseness with elimination of redundancy and the availability of reasonable defaults (12). The high-level languages listed above have all either made optional, or removed entirely, much of the syntactic overhead characteristic of low-level languages, such as explicit variable declarations and extensive headers or preambles that contain information that can be inferred from the body of the program in most cases. They also typically feature simple array indexing syntax and concise literal forms for common data structures like arrays, matrices, sets and maps. A concise and minimal syntax eliminates unnecessary keystrokes and keeps more code visible on screen at a time. Studies have shown that there may be a correlation between lines of code entered and overall error rate, independent of other factors (13).

However, the benefits of concise syntax must be balanced with concerns about readability and familiarity. A number of principles have been proposed to operationalize the notion of code readability. The most widely-used collection of principles are Green's cognitive dimensions (14). Of particular relevance is the notion of self-consistency, which serves to ensure that similar forms have similar meaning and that there are few subtle or context-dependent distinctions that users must be mindful of. Another important principle has been called *closeness of mapping*, expressing the value of a close correspondence between mental models and the formal model as expressed concretely using the language.

Most researchers become familiar with formal notation by studying mathematics. In nearly all commonly used languages in scientific computing, common mathematical notations or close approximations thereof are generally used. In contrast, many academic languages

have settled on alternative notational styles. For example, the LISP family of languages uses a highly uniform list-based notation, while most functional languages typically borrow notation for function invocation from the lambda calculus, using the form `f x` rather than `f(x)` for function invocation. Although both of these styles have certain benefits, they can impose mental burdens on users who continue to mentally translate them into more familiar notation (15). Although these and related difficulties can decrease with experience, they remain an important barrier for new users to these languages.

Syntactic cues like whitespace and typography that do not have a formal meaning but rather exist to assist developers are called secondary notation (16). Most languages are whitespace-insensitive, while others (notably, Python) enforce consistent uses of whitespace, both in pursuit of consistency and as a technique to eliminate the need for block delimiters. Additionally, a few languages, such as Mathematica, have support for more typographically-rich mathematical notation via a structural editing interface. This technique appears to be helpful, although we do not know of formal studies that provide evidence for this claim.

2) Support for Multiple Paradigms and Abstractions

Although the difficulties of low-level and parallel programming are widely acknowledged, there is little consensus on which high-level abstractions are most appropriate for easing this burden. Indeed, it appears likely that no one abstraction will emerge triumphant over the others. Although library-based abstractions are useful, they often suffer from limitations of the language, particularly if they require compile-time support. Primitive language support for a parallel abstraction has been shown to be more useful than a library-based implementation in at least one case (17). As such, it is important that a language support several modes of operation, ideally without excessively favoring one over another.

Languages that have been designed specifically to explore a single abstraction as a core language feature often see limited adoption because they remain difficult to use in circumstances for which the favored abstraction is inappropriate. Examples of these abstractions

and languages that feature them include:

- shared-memory concurrency (e.g. Java, C#)
- share-nothing message-passing (e.g. Erlang)
- flat data and task parallelism (e.g. OpenMP and OpenCL)
- nested data and task parallelism (e.g. NESL, Copperhead)
- transactional memory (e.g. the language described by Harris (18))
- automatic parallelization of functional primitives (e.g. Data Parallel Haskell, MapReduce and others)
- (partitioned) global address spaces (e.g. UPC, Co-array Fortran, Global Arrays, Fortress, X10, Chapel and others)
- adaptive thread migration (e.g. Charm++)

3) Extensibility

Although support for multiple paradigms and primitive abstractions can be built into a language design, this leaves control in the hands of the language designer. Novel or domain-specific constructs that may be useful to a small number of users or in rare situations can be difficult to develop and distribute for this reason, leading to the proliferation of new languages as described above. Language extensibility mechanisms support these use cases by giving users the ability to develop new abstractions and constructs that behave as if they were primitive constructs. If a mechanism is powerful enough, nearly all language constructs may be implemented using it, greatly simplifying the core semantics of a language.

Dynamic languages commonly rely on mechanisms like operator overloading and metaobject protocols (19) to provide extensibility via indirection. For example, the Python language

allows objects to overload nearly every operator and as well as operations like attribute lookup (`obj.attr`) and assignment. This mechanism is used by a number of libraries to create a more natural interface to a low-level API (e.g. `pycuda`). More open-ended dynamic mechanisms, such as programmatic macros, have also been widely studied but as of yet have seen little adoption in languages used by professional end-users.

Language extensibility for statically-typed languages, on the other hand, remains an active research area. Compile-time metaprogramming systems, which are related to programmatic macros, have been developed for a number of languages (e.g. Template Haskell (20)) but these too have seen relatively limited development or adoption thus far.

Some researchers have advocated the use of compiler extension mechanisms, rather than mechanisms built into a language itself (cf. (21)). Modern compilers now offer some support for front-end language extensions, although often these can be quite difficult to use. Although potentially powerful, this approach can also lead to issues when extensions are not easily composable or when multiple compilers exist for a language. Back-end extensions (that is, extensions that preserve the semantics of the language, improving only performance) are better supported in modern compilers.

Domain-specific language frameworks are a related approach that can serve many of the same goals as language-based extension mechanisms (22). These tools ease the development of "little languages" and allow for the development and distribution of language features as modules. Domain experts use these tools to develop highly specialized languages that capture domain requirements precisely and allow for natural and concise specifications of scientific models and other structures. A major issue with this approach arises at language boundaries – interoperability between different domain-specific languages is difficult due to feature mismatches. Another issue is that domain-specific languages often outgrow their initial implementations and begin to need increasingly powerful general-purpose features.

All extensibility mechanisms must be used carefully. Indeed, inexperienced developers can abuse mechanisms like operator overloading to create inscrutable interfaces that cause

more problems than they solve. Some languages (notably Java) have taken up the philosophy that even simple language extension mechanisms should not be in the hands of end-users for this reason. Although this continues to be debated, we argue that extensibility is crucial for parallel and scientific programming in particular due to the significant levels of ongoing research into new parallel abstractions and the diverse set of domain-specific use cases.

4) Support for Higher-Order Constructs

A number of parallel programming data structures and algorithms are of higher-order, meaning that they take other functions or types as arguments or parameters. We argue that support for higher-order programming is critical to language usability in our target domains.

Well-known examples of functions that operate using other functions include fundamental parallel primitives like map, reduce and scan. Fortunately, most modern languages now support using functions as values. Languages like C and Fortran implement this using function pointers, although at considerable syntactic expense. However, unlike functional languages and most dynamic languages, they do not allow anonymous functions (that is, functions that are defined as inline expressions rather than statements), nor functions that close over variables in the surrounding scope. The most recent revision of C++ now has support for closures and closures are also being considered for a future revision of Java. OpenCL does not come with any support for higher-order functions, even via function pointers.

Functions and types that are parameterized by types are often referred to as *polymorphic* or *generic*. C++ supports this using its template system. Java, ML and Haskell support a simpler form of *parametric polymorphism*, and Haskell also supports a more flexible *type class* system. C++, Java and Fortran also support *function overloading*, allowing multiple versions of a function that differ only according to their argument types. C and OpenCL do not support any form of polymorphism or user-defined function overloading.

Dynamic languages do not require that variables be assigned a type, so generic functions are written using run-time checks that ensure that a particular value supports the specific

interface that a function expects. This is sometimes known as "duck typing" and is generally considered as a useful feature, due to its considerable flexibility. A promising approach that resembles duck typing, while operating statically, is known as *structural typing* (23).

5) Modularity and Packaging

Modularity is a broad term that refers to mechanisms useful for combining and reusing independently developed libraries of code. Languages with good support for modularity promote information hiding, thus localizing the effect of code changes, and allow modules to communicate over well-defined interfaces. There remains widespread disagreement and considerable ongoing research on language support for modularity. Object-oriented methodologies, although common in industrial projects, are used less frequently in scientific codes due to a perceived loss of control and performance (10). Many dynamic languages support modularity only implicitly using duck typing. Functional languages, on the other hand, often have module systems that enforce correct usage of an interface at compile-time, albeit at some notational cost (24).

The packaging and linking mechanisms available in a language can also significantly impact its usability. Languages like C and C++ use a fragile preprocessor-based packaging mechanism and require separate header files, which often leads to subtle errors and compilation inefficiencies. More recent high-level languages have developed a varied set of mechanisms that are significantly simpler for packaging and deployment.

6) Verifiability

Verifying that a program does not contain errors and that it will operate according to specification (if a specification exists, which can be rare in this domain (5)) is a critical concern across all areas of software development. Errors in scientific programs may arise due to problems in basic program logic, as in other domains, but also due to the accumu-

lation of numerical approximation errors or by violation of domain-specific constraints (e.g. inconsistent scientific units.)

A number of design decisions can influence the difficulty of formal program verification. Broadly stated, unconstrained support for dynamic indirection and direct access to memory have made program verification more difficult in many commonly-used languages. As a result, many classes of errors are only caught at run-time, often due to edge cases that are difficult to test for.

Advanced type systems, matched with appropriately constrained data structures and control constructs, can dramatically increase the likelihood that an error is found at compile-time and even eliminate entire classes of errors (24). A large portion of academic research on programming language design has focused on designing such type systems. Unfortunately, as with many new parallel abstractions, these generally require that a language be modified with new primitives and most work has either been with prototype languages or functional languages like Ocaml, Haskell or Coq, which, due to some of the factors mentioned here, have seen limited (though growing) adoption in scientific computing so far.

In the absence of a formal proof of correctness, users must test programs with specific inputs, relying both on language-provided run-time checks and user-provided run-time assertions to catch errors. Most languages have unit testing frameworks designed to minimize the burden of developing unit tests, though these are used infrequently in science (5; 2). Several languages (e.g. Eiffel) also have support for specifying pre- and post-conditions for functions, so that the assertions that must hold about arguments are visible in the function signature, but this feature has not been well-supported by widely-used languages to date.

b. Adoption Criteria

While most of the design criteria described in the previous section have been the topic of significant (and ongoing) research, there are a number of other, more practical issues that can significantly affect adoption of a new language or tool. It should be noted that

certain design decisions may make it easier to satisfy these adoption to criteria as well, an important consideration given that incentive structures in academia often do not reward efforts to improve a language along some of the following dimensions (1).

1) Performance

Performance is, of course, a critical concern in scientific and (particularly (10)) high-performance computing. Low-level languages typically elect to give the user direct access to hardware primitives. High-level languages must rely on a sophisticated compiler to translate high-level abstractions into performant code. Although the latter approach would be ideal, program optimization remains an active research area with many open problems. Indeed, many relevant algorithms have been shown to be NP-Complete, so compilers must rely on heuristics. Humans remain better at inventing and applying heuristics, given enough motivation and experience, than computers in many cases. Indeed, even when the compiler can sometimes produce better optimizations, users generally insist on being able to form a mental model of what their code is doing at the machine-level so that they can *reason* about the effects of code changes on overall performance (25). As such, users generally demand that low-level abstractions remain available alongside high-level abstractions. It may also be useful to support a model where programmers can formalize, package and directly apply optimization heuristics programmatically, rather than relying on a "black-box" compiler.

2) Portability

A number of surveys have revealed that portability is one of the most important issues considered by scientists and engineers (2)(10). Several processor architectures and operating systems are in widespread use, with more appearing on a fairly regular basis. Low-level languages like C have generally achieved a reasonable level of portability, although aspects of the language that are underspecified can be a source of errors. The OpenCL language

was designed with portability as a major design criteria, and compilers for OpenCL are now available for a diverse collection of processor and accelerator architectures. High-level languages are generally highly portable due to the use of a virtual machine architecture.

It should be noted, however, that simple portability is not entirely sufficient – users want to be able to write programs that can be ported to new architectures and operating systems and achieve high performance without significant retuning. This remains an area of active research. Recent efforts to build abstractions that generate efficient code for devices with multi-level memories, as implemented in the Sequoia language (26), have progressed toward this goal.

3) Useful Error Messages

When the compiler or run-time system of a language generates an error, users must determine the source of the problem using the information provided in an error message. Studies have shown that good error messages can dramatically reduce the time it takes to debug programs (27). Indeed, a widely-reported frustration with C++ is that it produces overly verbose and cryptic error messages, particularly when using templates.

4) Tool and Infrastructure Support

Modern software development now relies on a number of tools to assist with common tasks, such as debuggers, profilers, syntax-aware editors, documentation generators, style checkers and interactive interpreters. Similarly, most established languages benefit from a centralized package repository (e.g. PyPI) paired with a package manager that can automate the installation of new packages. These tools and infrastructure are not trivial to develop from scratch, and few incentives exist for researchers to do so, but few users are willing to do without these tools for non-trivial projects (25).

5) Backwards Compatibility

End-user communities have produced a number of highly tuned and tested packages that are widely used in their fields. Porting these libraries to a new language requires significant effort and few communities will be willing to do so, particularly before a language is very well established. There is generally a greater willingness to develop wrappers that invoke functions from these packages, however. A powerful foreign function interface, ideally able to handle native code as well as code in existing widely-used high-level languages, enables these efforts to proceed smoothly and can provide a language with a large package library relatively early in its development.

Many of the reasonably successful approaches to date are built incrementally on existing languages as libraries or as simple extensions to existing languages (e.g. OpenMP, UPC, Co-array Fortran, Cilk++, MPI, CUDA, OpenCL). Notably, this allows existing programs to continue to function correctly, while enabling the gradual integration of the new constructs in parts of the code that would derive the greatest benefit. In contrast, approaches that have required total rewrites have had more difficulty recruiting early adopters (e.g. X10 (28).)

6) Learning Material

Existing languages benefit from a large collection of books, presentations and tutorials that allow new users to get started quickly. The most mature languages benefit further from the availability of courses and professional training seminars. New languages often suffer due to the lack of such polished learning material, or in some cases, due to the lack of *any* significant learning material targeted at end users (rather than other language designers.)

7) Open Availability

New languages and abstractions are often viewed with suspicion if they do not come with source code that can be modified under a free software license. The most unencumbered

licenses (other than public domain releases) are BSD-style licenses. So called *copyleft* licenses like the GPL are also popular, requiring primarily that modifications to the compiler be distributed under the same license.

8) Social Proof

Finally, users look to the language's user community for evidence that the language is a serious effort that will not be abandoned and that there are other developers building libraries and sharing information over established communication channels. The availability of non-trivial demonstrations have been cited in surveys as critical to adoption as well (10). These criteria, as with many of the others described above, can be difficult to satisfy, particularly for "clean-slate" language designs, which may help to explain why language adoption is often slow. However, even projects with a narrower scope, such as new parallel or domain-specific abstractions, must tackle these issues. Funding agencies can aid in adoption by guaranteeing development will be supported for a number of years for languages that appear to have the greatest promise.

REFERENCES

[1] J. Howison and J.D. Herbsleb. Scientific software production: incentives and collaboration. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, pages 513–522. ACM, 2011.

[2] J.E. Hannay, C. MacLeod, J. Singer, H.P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8. IEEE Computer Society, 2009.

[3] G.E. Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[4] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Yutaka Ishikawa, Zhong Jin, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias Mueller, Wolfgang Nagel, Hiroshi Nakashima, Michael E. Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad van der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The international exascale software project roadmap.

[5] Judith Segal. Models of scientific software development, May 2008.

[6] J. Segal. Some problems of professional end user developers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 111–118. IEEE Computer Society, 2007.

[7] Bernd Mohr. Survey of system software stacks in the iesp community.

[8] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana. A survey of scientific software development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 12. ACM, 2010.

[9] J.C. Carver, R.P. Kendall, S.E. Squires, and D.E. Post. Software development environments for scientific and engineering software: A series of case studies. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 550 –559, may 2007.

[10] V.R. Basili, J.C. Carver, D. Cruzes, L.M. Hochstein, J.K. Hollingsworth, F. Shull, and M.V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *Software, IEEE*, 25(4):29–36, 2008.

[11] J.F. Pane and B.A. Myers. *Usability issues in the design of novice programming systems*. Citeseer, 1996.

[12] J.R. Cordy. Hints on the design of user interface language features: lessons from the design of turing. In *Languages for developing user interfaces*, pages 329–340. AK Peters, Ltd., 1992.

[13] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *Software Engineering, IEEE Transactions on*, 27(7):630–650, 2001.

[14] T.R.G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[15] J.R. Anderson and R. Jeffries. Novice lisp errors: Undetected losses of information from working memory. *Human-Computer Interaction*, 1(2):107–131, 1985.

[16] TRG Green. Programming languages as information structures, 1990.

[17] V. Cavé, Z. Budimlić, and V. Sarkar. Comparing the usability of library vs. language approaches to task parallelism. In *Evaluation and Usability of Programming Languages and Tools*, page 9. ACM, 2010.

[18] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Notices*, volume 38, pages 388–402. ACM, 2003.

[19] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[20] T. Sheard and S.P. Jones. Template meta-programming for haskell. *ACM SIGPLAN Notices*, 37(12):60–75, 2002.

[21] A.T. Clements. *A comparison of designs for extensible and extension-oriented compilers*. PhD thesis, Citeseer, 2008.

[22] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[23] D. Malayeri and J. Aldrich. Is structural subtyping useful? an empirical study. *Programming Languages and Systems*, pages 95–111, 2009.

[24] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[25] S. Squires, WG Tichy, and L. Votta. What do programmers of parallel machines need? a survey. In *Second Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2005.

[26] K. Fatahalian, D.R. Horn, T.J. Knight, L. Leem, M. Houston, J.Y. Park, M. Erez, M. Ren, A. Aiken, W.J. Dally, et al. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 83–es. ACM, 2006.

[27] G. Marceau, K. Fisler, and S. Krishnamurthi. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 499–504. ACM, 2011.

[28] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Notices*, volume 40, pages 519–538. ACM, 2005.