A Survey of the Practice of Computational Science

Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang
Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard
Taewook Oh, Matthew Zoufaly, David Walker, David I. August

Princeton University
{pprabhu,tjablin,rarun,yunzhang,jialuh}@princeton.edu
{hanjunk,npjohnso,fengliu,soumyade,sbeard}@princeton.edu
{twoh,mzoufaly,dpw,august}@princeton.edu

ABSTRACT

Computing plays an indispensable role in scientific research. Presently, researchers in science have different problems, needs, and beliefs about computation than professional programmers. In order to accelerate the progress of science, computer scientists must understand these problems, needs, and beliefs. To this end, this paper presents a survey of scientists from diverse disciplines, practicing computational science at a doctoral-granting university with very high research activity. The survey covers many things, among them, prevalent programming practices within this scientific community, the importance of computational power in different fields, use of tools to enhance performance and software productivity, computational resources leveraged, and prevalence of parallel computation. The results reveal several patterns that suggest interesting avenues to bridge the gap between scientific researchers and programming tools developers.

1. INTRODUCTION

Computational science [53], a multidisciplinary field encompassing various aspects of science, engineering, and computational mathematics is increasingly being seen as the "third approach" [23], after theory and experiment, to answering fundamental scientific questions. Researchers practicing computational science typically face two concerns competing for their time. Primarily, they must concentrate on their scientific problem by forming hypotheses, developing and evaluating models, performing experiments and collecting data. At the same time, they also have to spend considerable time converting their models into programs and testing, debugging, and optimizing those programs.

In the past two decades, there has been an exponential increase in the amount of data generated and computation performed within many scientific disciplines [53, 55], signifying an increasing need for high performance computing. Writing correct and high performance programs is difficult even for computer scientists [33]. Given this background, this paper seeks to answer the question: How are scientists coping with the growing computing demands?

Recently, an online survey conducted a *broad* study of the programming practices of a wide range of researchers, revealing many potential problems encountered in correctly writing scientific programs [30, 43]. Continuing in the same spirit, this paper presents an *in-depth* study of the practice of computational science at Princeton University, a RU/VH [1] institution. This study is conducted through a survey of researchers from diverse scientific disciplines. This survey covers important aspects of computational science including programming practices commonly employed by researchers, the importance of computational power, and performance enhancing strategies in use. The results are presented in the context of the university's prevailing computational environment, providing insights into diverse computational practices followed within the institution.

The analysis of survey results reveals several patterns that suggest various areas of improvement. In contrast to the popular view that scientists use only numerical algorithms written in MATLAB and FORTRAN, the survey discovered that C, C++, and Python were popular among many scientists and there is a growing need for non-numerical algorithms. Despite the availability of clusters and large-scale shared memory systems within the University and a general desire for higher performance through parallel computation, a substantial portion of scientific computation still takes place on scientists' personal computers. Although many scientists use shared-memory multicore desktops and not clusters for scientific computation, knowledge of shared-memory parallelization techniques in the scientific community is virtually non-existent. Furthermore, the survey determined that scientists frequently do not leverage performance analysis tools to track down the causes of poor performance and consequently "optimize" cold-code while ignoring a computation's real bottlenecks. The contributions of this paper are:

- An in-depth survey of the practice of computational science at a RU/VH institution. The survey is conducted through personal interviews with 114 researchers randomly selected from diverse fields of natural sciences, engineering, interdisciplinary sciences, and social sciences.

[1] RU/VH stands for "very high research activity doctoral-granting university", as classified by the Carnegie Foundation [11].

SC'11, November 12–18, 2011, Seattle, Washington, USA.
ACM 978-1-4503-0771-0/11/11.

Field	Discipline	Count
Natural Sciences	Astrophysics	3
	Atmospheric and Oceanic Sciences	2
	Chemistry	5
	Ecology and Evolutionary Biology	5
	Physics	5
	Geosciences	6
	Molecular Biology	4
	Plasma Physics	2
Engineering	Chemical	7
	Civil and Environmental	5
	Mechanical and Aerospace	11
	Electrical	12
	Operations Research and Financial	5
Interdisciplinary Sciences	Music	4
	Applied and Computational Math	2
	Computational Biology	4
	Neuroscience/Psychology	13
Social Sciences	Economics	10
	Sociology	5
	Politics	4
Total		114

Table 1: Subject population distribution

- An analysis of survey results that suggests several areas of improvement, both in terms of practices employed by scientific researchers and future research directions for programming tools developers

2. SURVEY METHODOLOGY

The survey covers a set of 114 randomly selected researchers from diverse fields of science and engineering at Princeton University. The pool of survey candidates includes all graduate students, post doctoral associates, and research staff in various scientific disciplines at Princeton University. An email soliciting participation in the survey was initially sent to randomly selected candidates from the university database. The email mentioned "use of computation in research" as a criterion for participation. After a candidate replied indicating interest in the survey, an interview was conducted by at least two of the authors, exploring, in depth, the various aspects of scientific computing related to the candidate's research.

Table 1 shows the distribution of subjects across different scientific fields. In this paper, the word "scientist" is used in a broad sense, to cover researchers from natural sciences, engineering, interdisciplinary sciences, and social sciences. A total of 20 disciplines were represented. Of the 114 subjects, 32 were from the natural sciences, 40 from engineering, 19 from interdisciplinary sciences and 19 from the social sciences. Most of the interviewees were graduate students in different stages of their research. Six interviewees were postdoctoral researchers and research staff. Barring two instances, researchers from the same discipline were from different research groups.

The survey was conducted through personal interviews, in order to allow for a deeper understanding of the different computing scenarios and situations unique to each subject. Each interview conducted was in the form of a discussion that lasted for about 45 minutes. All the interviews were conducted over a period of 8 months. The survey covered three major themes central to scientific computing: (a) programming practices (b) computational time and resource use, and (c) performance enhancing methods.

3. RESULTS

This section presents the results of the survey. To begin with, the scientific computing environment at Princeton University is characterized. With this as the background, the results of the survey are presented suitably categorized into the three themes mentioned above. Each theme is introduced by posing a broad set of questions, and then answering these questions through a general set of patterns observed during the survey along with data to substantiate each observation. To highlight these key patterns, and other central ideas or conclusions that appear later in the paper, we set them apart from the main text as an italicized comment.

3.1 Computing Environment

Researchers at Princeton University are heavily supported in terms of computational resources and expertise. The Princeton Institute for Computational Science and Engineering (PICSciE) [13] aims to foster the computational sciences by providing computational resources as well as the experience necessary to capitalize on those resources. At the time of writing, these resources include the larger cluster hardware available through the Terascale Infrastructure for Groundbreaking Research in Engineering and Science (TIGRESS) [10]. TIGRESS is a high performance computing center that is an outcome of collaboration between PICSciE, various research centers [9, 8, 12, 14], and a number of academic departments and faculty members.

TIGRESS offers four Beowulf clusters (with 768, 768, 1024, and 3584 processors), and a 192 processor NUMA machine with shared memory and 1 petabyte of network attached storage. These clusters serve the computational needs of 192 researchers. Administrators at TIGRESS estimate that their systems are at 80% utilization. Additionally, PICSciE offers courses, seminars and colloquia to aid the computational sciences. Since 2003, PICSciE has offered mini-courses on data visualization, scientific programming in Python, FORTRAN, MATLAB, Maple, Perl and other languages, technologies for parallel computing (including MPI and OpenMP), as well as courses on optimization and debugging parallel programs. Recently, PICSciE began offering a course on scientific computing. PICSciE also offers programming support for troubleshooting malfunctioning programs, parallelizing existing serial codes, and tuning software for maximum performance.

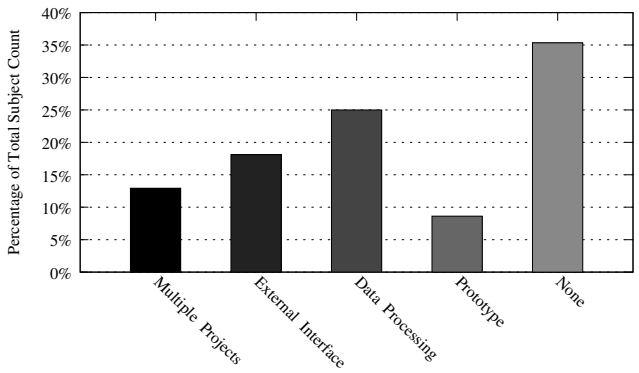
3.2 Programming Practices

Representative questions concerning this theme included: What kind of programming paradigms, languages and tools do scientists use to perform scientific computation? Do scientists employ effective testing methods to certify the results of their programs? What fraction of research time do scientists spend in programming or developing software?

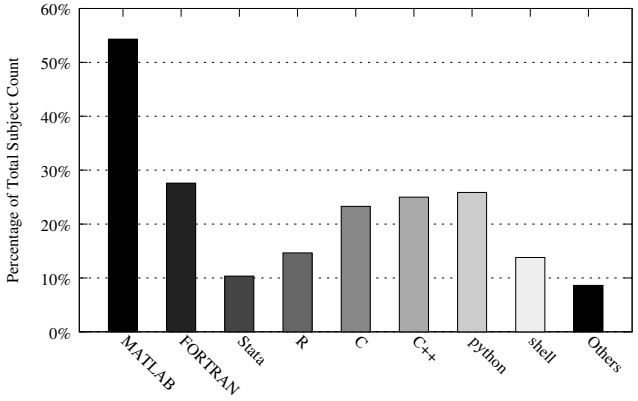
Scientists commonly interface a diverse combination of numerical, general purpose, and scripting languages.

One common perception is that scientific computation is dominated by heavy array-based computation in languages that are specially tuned for numeric computation like FORTRAN and MATLAB [44, 51]. Contrary to this perception, our survey indicates that scientists commonly interface a diverse combination of numerical, general purpose and scripting languages in an ad-hoc manner. Around 65% of scientists use at least one combination of numerical/scripting language and a general purpose language.

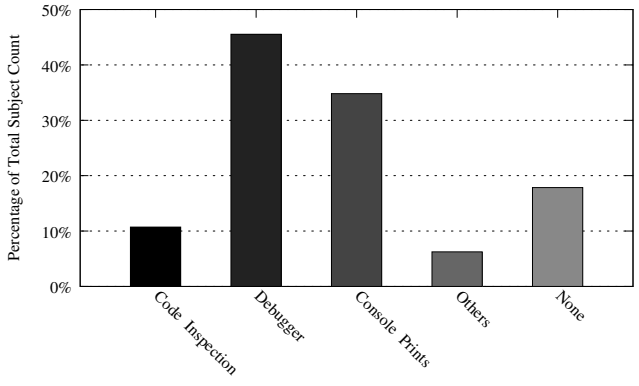
The distribution of multiple-language use-case scenarios is shown in Figure 1a. Close to one-fourth of scientists used



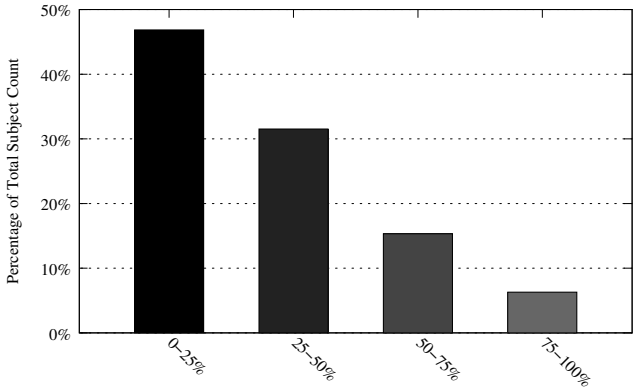
(a) Multiple Language Use-Case Scenarios



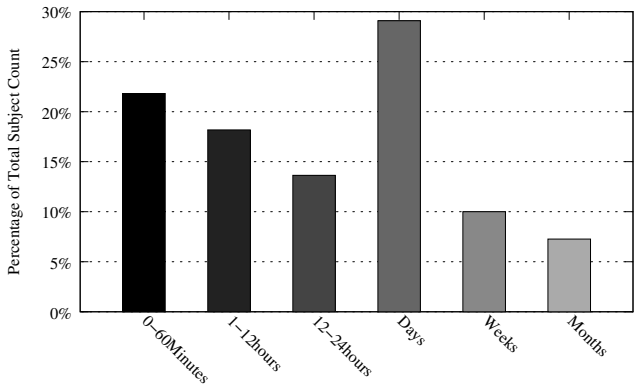
(b) Programming Language Use Distribution†



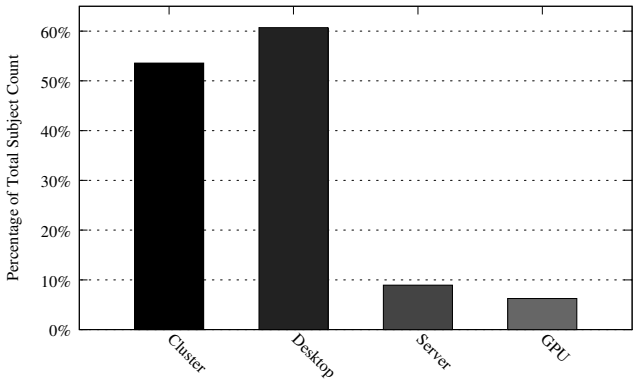
(c) Debugging techniques employed



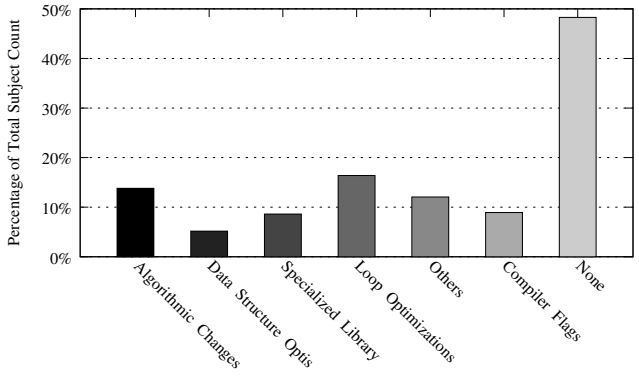
(d) Programming time, as percentage of research time



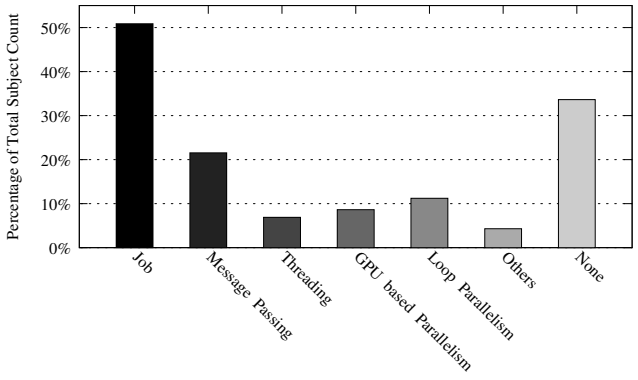
(e) Program Execution Time Distribution



(f) Computational Resource Use†



(g) Distribution of Performance enhancing strategies†



(h) Use of Parallelism†

Figure 1: Survey Data. The categories in the graphs marked † are not mutually exclusive and do not sum to 100%

numerical computing/scripting languages only for pre- and post- processing of simulation data, while writing all of the heavy duty computation code in general purpose languages like C and C++ for performance.

18% of scientists leveraged external interface functionality provided by languages like MATLAB to call into libraries pre-written in C/C++. Interfacing of this kind often involved writing wrappers to emulate the native programming model. For instance, researchers in Ecology and Evolutionary Biology, wrote a "call by reference" emulation framework in MATLAB when interfacing pointer-based data acquisition code written in C.

Nearly 9% of researchers used scripting/numerical languages for fast prototyping. They wrote their programs first in MATLAB/Python-like languages and tested their algorithms on small data sets. Once tested, these programs were re-written in C and C++ for bigger data sets. Around 13% of researchers used both numerical and general purpose languages for different projects, typically influenced by the perceived productivity versus performance trade-offs for the given problem at hand.

The distribution of different kinds of programming languages employed by scientists in the survey is shown in Figures 1b and 2. As Figure 2 shows, there is considerable overlap between the use of general purpose, scripting and numerical languages. The most dominant numerical computing language in use was MATLAB – more than half the researchers programmed with it. This is followed by FORTRAN, which was used by around 27% of researchers. Researchers using FORTRAN were influenced by the availability of legacy code, typically written by their advisors during the early nineties. More than 40% of surveyees relied on a general purpose language to deliver on computational performance. Two specific languages, C and C++ dominated in equal measure. An interesting point was the discipline-wise stratification of researchers using C/C++ and FORTRAN code. Researchers working in emerging interdisciplinary fields like Psychology, Neuroscience, and Computational Biology wrote programs in C/C++. By contrast, most of the FORTRAN use was restricted to established scientific fields in natural sciences like Astrophysics, Chemistry, and Geosciences. The dominant scripting language in use was Python. Around one-fourth of interviewees used Python, with shell scripts as the second favorite. Apart from normal string and data parsing and processing, researchers leveraged several scientific packages written in Python like SciPy [34], NumPy [47], and Biopython [22].

Scientists spend substantial amount of research time programming.

On average, scientists estimate that 35% of their research time is spent in programming/developing software. The distribution is shown in Figure 1d. While initially some time is spent on writing code afresh, a considerable portion of time is spent in many tedious activities. For example, researchers in Politics and Sociology who used R/Stata had to do considerable programming to retrofit census data into formats that individual packages in R/Stata understood. Some researchers in Chemical Engineering had to reverse engineer undocumented legacy code that performed flame simulation, long after the original authors had graduated, in order to adapt the code to newer fuels. Many researchers also re-wrote code for performing similar tasks rather than templatize and re-use code. Quite often, this was done via a copy, paste and modify procedure which is highly error prone. None of these activities were well tested. Despite this, a vast majority of these researchers felt that they "spend more time programming than they should," and that research time was better spent in focusing on scientific theories or on experiments ("more concerned about physics," said one researcher).

Scientists do not rigorously test their programs.

Given that the computational method has taken over as the method of first choice in many scientific fields [55], one would expect scientists to rigorously test their programs using state-of-the-art software testing techniques. However, our survey results point to the contrary. Although researchers spend 57% of their programming time on finding and fixing errors in their programs, the debugging and testing methods employed were primitive. Only one researcher considered the use of assertions at all in her code. Only three researchers wrote unit tests. Not many researchers were aware of version control systems, given their utility in detecting sources of regressions.

More than half of the researchers did not use any debugger (Figure 1c) to detect and correct errors in their code. 18% of researchers never tested their programs once they were written, either because they thought it was "too simple" or relied on code written by others, which was assumed to be "well tested." 11% of researchers relied on "trial and error" to detect bugs, which typically involved checking subsections of code for correctness by commenting out the rest of code. Given the inherent combinatorial space of code changes that are possible, this process itself was error-prone and slow. A small minority of researchers relied on the expertise of their more knowledgeable peers to fix errors ("call up people who have experience," said one researcher).

A few scientific programs conform to best software engineering practices and have high standards of reproducibility.

A recent article that appeared in Nature News [43] cast a rather bleak picture of scientific programming practices, citing complete absence of known software engineering methods, testing, and validation procedures in most scientific computing projects. While our survey results do agree in large part with those results, we also found many survey interviewees leveraged a small set of open source scientific programs that stood out both in terms of best software engineering practices and high standards of reproducibility. Typically, each discipline had a few open source programs developed by scientific teams world-wide that were popular and were utilized by many others in their field. In our subject population, around 48% of interviewees used or modified these open source software at some stage of their research.

Table 2 lists some of the representative open source programs in each field that are in this category. Most of these programs are results of projects characterized by sophisticated software engineering practices and evolved with goals of reproducibility, validation, and extensibility. These programs were written by researchers in science rather than professional programmers. These projects are distinguished from the various scientific programs developed in-house by the following critical features:

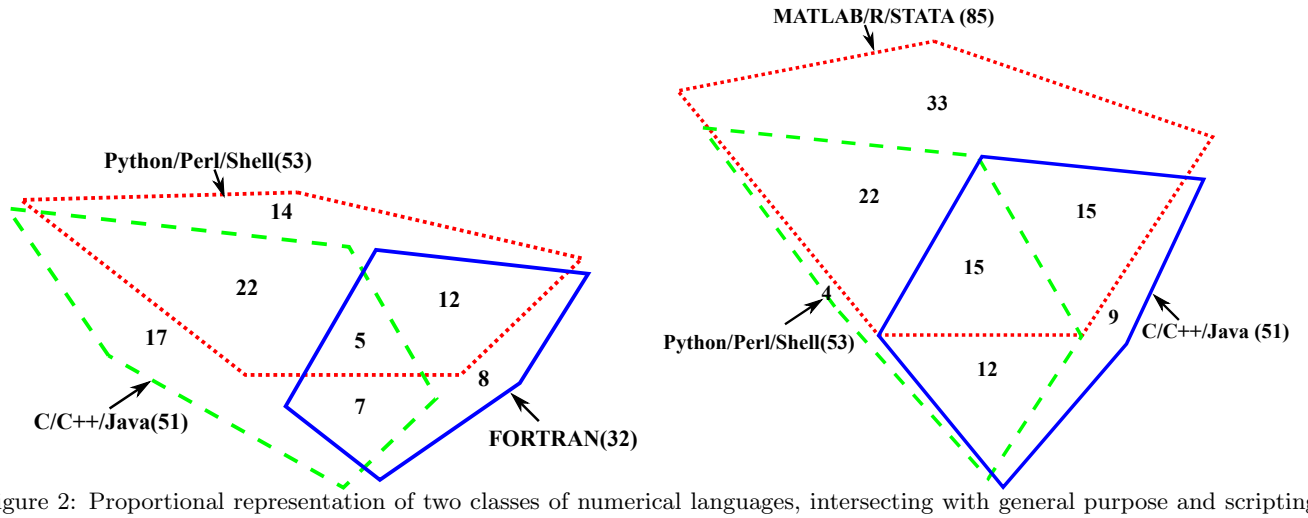


Figure 2: Proportional representation of two classes of numerical languages, intersecting with general purpose and scripting languages

- **Focus on Extensibility.** Most of these open source programs had extensibility and interoperability as a first order design concern. This often meant that the codes were modular and portable. For instance, AFNI [21] has a plug-in based architecture. AFNI users can write their own plug-ins for analysis and visualization of MRI data. GEANT4 [15] developers specifically adopted object oriented paradigms to allow for customized implementation of several generic interfaces. JAGS [50], although written in C++, has well defined R interfaces. The developers of LAMMPS [49] chose portability over optimality by opting not to use vendor specific APIs for message passing.

- **Long History of Software Development.** The level of sophistication achieved by these programs is the result of years of dedicated programming effort. Often, these programs were in development since the 1990s and underwent significant changes that often made the newer versions more general and portable than the earlier designs. In many cases, the open source program was an outgrowth of collaboration between scientists across different institutions. For instance, SPECFEM3d [38] was initially prototyped for the Thinking Machine using High Performance FORTRAN, and later redesigned using MPI to make it portable to run on different clusters. JAGS was written to achieve a platform independent implementation of an existing Bayesian analyzer. GEANT4, Quantum Espresso [27] and SPECFEM3d were designed and developed over many years by international teams with common interests, and with an experience of having developed similar programs in the past.

- **Performance and "Separation of Concerns."** The current set of open source programs adopt diverse performance optimization techniques. At the same time, the sections of the program that included machine specific optimizations were carefully separated from the machine independent portions of code using modularized interfaces. Many of these programs provide an interface to the user to choose accuracy-performance tradeoffs, often as a compile time or a run time option. Developers of these programs did not make any compromises on performance for providing flexible interfaces. For instance, fftw [26] provides a module called the "planner" that is responsible for choosing the best optimization plan for a given architecture without requiring any additional code changes from the user. SPECFEM3d's flexibility has given rise to versions that are being used across 150 different academic institutions and at the same time has also won the Gordon Bell prize [39, 3] for scalable performance.

- **Extensive testing and validation.** The programs listed in Table 2 have a large user base. These programs had extensive testing and validation methods in place. Very often, these open source projects had robust testing frameworks and used version control systems. These programs were accompanied by considerable documentation in the form of tutorials and user guides that made it easier to detect discrepancies in the program behavior. An active user community around the software existed that regularly reported bugs which were quickly fixed. Some of the projects even had a bug tracking system.

3.3 Computational Time and Resource Use

Representative questions concerning this theme included: How long do scientists wait for a computer to complete project runs? What kind of computational resources do scientists typically use to meet their computational needs? How would their research change with faster computation?

Programs run by scientists take on the order of days to complete.

Scientists spend a significant amount of time waiting for programs to run to completion. The distribution of waiting times is shown in Figure 1e. Nearly half of the researchers who participated in the survey spend more than a few days waiting for program completion. Of this, around 15% of researchers wait on the order of months. While the researchers who ran programs for days and weeks were randomly distributed across different scientific disciplines, the researchers who waited for months were from three distinct departments: Chemistry, Geosciences, and Chemical Engineering. The corresponding programs in Chemistry and Chemical Engineering involved various forms of molecular dynamics simulations, although in each case different scientific theories were leveraged. The program from Geosciences performed ocean modeling.

Around a third of the researchers wait on the order of a

Scientific Discipline	Application	Software Engineeering					Programming Model/ Language	Typical Execution Scenario	
		Ver	Mail	Doc	Bug	EI		Run time	Architecture(s)
Astrophysics	Athena [56]	✓	-	Extensive	✓	-	MPI + FORTRAN/C	20 hours	Cluster with 192 nodes, 2 cores per node
Neuroscience / Psychology	AFNI [21]	-	✓	Moderate	✓	✓	OpenMP + C	15 hours	Cluster with 52 nodes, 8 cores per node
Chemical Engineering	LAMMPS [49]	✓	✓	Moderate	✓	✓	MPI + C	3 days	Cluster with 4 nodes, 4 cores each
Chemistry	Quantum Espresso [27]	✓	✓	Moderate	-	-	MPI + FORTRAN/C	1 month	Cluster with 384 nodes, 2/4 cores per node
Civil and Environmental Engineering	VIC [41]	-	✓	Moderate	-	-	Sequential + C	1.5 weeks	Cluster with 384 nodes, 2.4 cores per node
Computational Biology	Sleipnir [31]	✓	-	Extensive	-	✓	Pthreads + C++	1 day	Cluster with 58 nodes, 8 cores per node
Economics	Dynare [35]	-	-	Extensive	✓	✓	Sequential + MATLAB/C++	12 hours	Desktop with 2 cores 2 cores
Electrical Engineering	fftw [26]	-	-	Moderate	-	✓	MPI/Pthreads + C	30 mins	Desktop and GPU
Geosciences	SPECFEM3D [38]	✓	✓	Extensive	✓	-	MPI/Pthreads + FORTRAN/C	30mins to 9hrs	Cluster with 384 nodes, 8 cores per node
Molecular Biology	HMMER [24]	-	-	Moderate	-	✓	Pthreads + C/C++	1 week	Cluster with 58 nodes, 8 cores per node
Physics	GEANT4 [15]	-	-	Extensive	✓	✓	Sequential + C	1 week	Cluster with 70 nodes, 2 cores per node
Politics	JAGS [50]	✓	-	Moderate	✓	✓	Sequential + C++	4 days	Desktop with 2 cores

Table 2: Open source compute-intensive scientific applications collected during the survey, along with typical execution scenarios for many interviewees. Software Engineering category notes the use of Version control systems, Mailing Lists, Documentation, Bug tracking, and Extensible Interfaces.

few hours for program completion while one-fifth waited on the order of minutes. Almost all the people who waited only for a few minutes used numerical computing environments like MATLAB to run their programs. A vast majority of these researchers were primarily experimentalists/theorists and employed computation to either validate theories or do data analysis. In the latter case, the analysis programs dealt with small data sets and were written by the researchers themselves. On the other hand, researchers who waited for days used an even mix of programs written in numerical computing languages and programs written in general purpose programming languages like C, C++ and FORTRAN and performed analysis over larger data sets along with other computational tasks like simulation and optimization.

Despite enormous wait times, many scientists run their programs only on desktops.

Traditionally, large scale scientific computing problems have been solved by relying on powerful supercomputers, massively parallel computers, and compute clusters [37]. In practice, do scientists take advantage of these powerful computing resources when they are available and easily accessible?

While researchers use a wide variety of computational resources, a substantial portion (40%) of them only use desktops for their computation. This is despite the fact that the computational tasks performed take more than a few hours for completion, for more than half of these researchers. Although more than three-fourths of these desktops have multiple cores (most commonly two), almost all of them run only single threaded code. Furthermore, surveying the algorithms and computational techniques employed in these programs reveals that several of these are amenable to parallelization.

Around half of the interviewees use clusters, with about one-fifth leveraging both desktops and clusters in their research. Use of multiple clusters with varying architecture types was common in this group as well. Typically the same code was run unaltered on multiple machines. Only a small proportion of researchers used GPUs and shared-memory compute servers in their research. Servers offer a unified memory address space as opposed to a divided address space in the case of clusters. The servers were further characterized by the lack of a job submission system, whereas clusters usually had a job submission system for batch processing. The distribution of computational resources employed is shown in Figures 1f and 3.

A vast majority of subjects continually trade-off speed for accuracy of computation. However, even while doing so, scientists very rarely tune their applications to perform optimally on each specific cluster that it is run on. Frequently seen patterns include increasing the error thresholds to allow for faster convergence of optimization solvers, performing fewer simulation runs which made the results more susceptible to outliers, coarse-grained scientific models that are greater approximations of the physical reality, and analyzing fewer data points to do hypothesis testing. Most of these choices were guided by the execution time witnessed on the first machine on which these programs were run. Some researchers while testing their initial implementations on their desktops wanted the runs to complete overnight. Often, these programs were run unaltered after an initial implementation on multiple machines with widely varying architectures.

3.4 Performance Enhancing Methods

Representative questions concerning this theme included: Do researchers care enough about performance to optimize

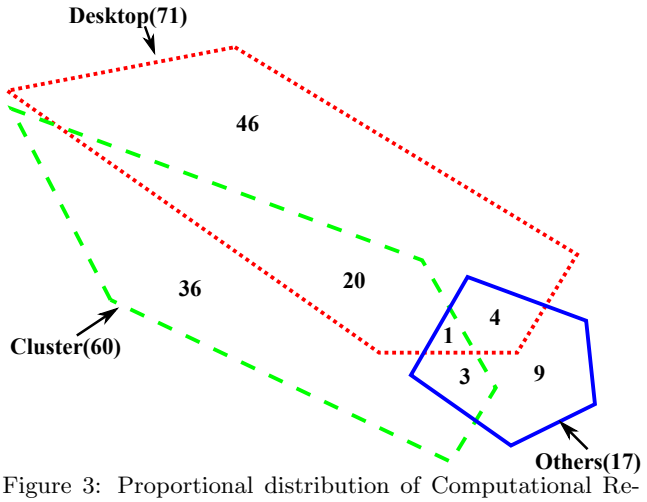


Figure 3: Proportional distribution of Computational Resource Usage. *Others* include Servers and GPUs

their code? What tools and techniques are used to optimize for performance? Do researchers target the most time consuming portions of their code? Are researchers aware of parallelism and do they make use of different parallelization paradigms to deliver performance in their code?

Scientists do not optimize for the common case.

Compiler writers and computer architects are well acquainted with the adage "90% of the execution time is spent in 10% of the code." Therefore, the general belief is, by optimizing hot code, greater performance gains can be expected per unit of effort put in. The survey results indicate that scientists have hypotheses on which portions of their code are hot, but more often than not do not test these hypotheses. Consequently, scientists may not be targeting the important sections of their program. A little more than half of the interviewees manually optimized code for performance. However, only 18% of researchers who optimize code leveraged profiling tools to inform their optimization plans. More than one-third of researchers were not aware of any profiling tools and did not time different portions of their code, while nearly one-fourth of researchers had heard about profiling but did not take advantage of them.

The most common reasons given for not using profilers were "not a first order concern," "not heard of them," "will not help," "I know where time is spent." As a result, quite a few scientists misjudged the portions of code that were the most time consuming and ended up optimizing code that did not contribute much to overall runtime. For example, one researcher performing biophysics simulations mistook MPI communication in his code for a performance bottleneck, even though it contributed to only 10% of the execution time. Profiling the application revealed that the real bottleneck was in a sorting algorithm, which accounted for more than 40% of execution time. Replacing the sorting algorithm with a faster one gave greater improvements in performance than optimizing for communication. Of the people who leveraged profiling information, close to three-fourths used a standard profiling tool like `gprof` and `MATLAB profiler`, while the remaining wrote custom timers (for example, function call counts rather time spent in certain functions) in specific portions of code to profile for execution time.

Researchers employed a wide variety of performance enhancing strategies in their code, as shown in Figure 1g. These strategies can be grouped into two categories. High level optimizations (done by 28% of researchers) included algorithmic changes, choice of better data structures, and use of specialized libraries. Typical examples for these include better heuristics for optimization problems, choice of a concurrent data structure over sequential one, use of specialized digital signal processing engines and built-in libraries in MATLAB. Low level optimizations (done by 26% researchers) included manual loop optimizations and use of compilation flags. The most popular of loop optimizations was manual loop vectorization. Of the people doing high level optimizations, only around half actually profiled their code. Of the people doing loop optimizations, only a third ever profiled their code. A little less than 10% of researchers use compiler optimization flags (e.g, -O3). Many researchers were either unaware of command line compiler optimization flags or thought of compiler optimizations as too low-level to actually make a significant difference in performance.

Scientists are unaware of parallelization paradigms.

Very few researchers were aware of or took complete advantage of different parallelization paradigms like message passing and shared memory parallelism directly in their code. Less than one-third of researchers had heard about different forms of parallelism (data, task/functional, and pipeline) that could potentially be applicable to improve the performance of their code. About a third of researchers did not use any form of parallelism in their research at all. Half the researchers relied heavily on batch processing. Researchers who leveraged parallelization paradigms in their programs can be categorized as follows (Figure 1h):

- **Message Passing based Parallelization.**

  Around 22% of researchers exploited message passing based parallelism in their research. Except for one researcher who used UNIX sockets directly, the rest of the researchers used MPI [29]. The vast majority of these people did not actually write MPI code themselves, but rather ran open source programs based on MPI. Many of them modified only the sequential portions of these programs, and lacked knowledge about MPI concepts. Many researchers wanted to learn more about MPI via peer training and academic courses. Some of them had attended various two-day mini-courses offered by PicSCiE on MPI. However, they still had many complaints like "it is hard," "looks complex," "big learning curve," and "implementation time is too high."

  Of the few researchers who wrote MPI code, a significant fraction faced enormous problems debugging. Their experience using debuggers like gdb and TotalView was not good. All of them uniformly complained about the complexity of understanding and using such "unintuitive" tools on clusters. Researchers had significant difficulty exploiting parallel I/O on clusters. Quite often, they resorted to using a single process to do all the prints, even when it increased communication costs.

- **Thread based CPU Parallelization.** Despite the fact that most desktops today have at least two cores and almost all nodes in a cluster have multiple processors operating on a shared memory address space, only 7% of researchers leveraged any form of thread based shared mem-

ory CPU parallelism. A total of only eight researchers admitted any knowledge of threads or had any experience using explicit thread based parallelism in their research.

- **GPU based parallelism.** Around 9% of researchers leveraged GPUs. Of these, there was one expert per research group who wrote the GPU based parallel code and the rest of the researchers were users of this code. For instance, research staff studying collective animal behavior in ecology and evolutionary biology included postdoctoral researchers with relevant computer science background who wrote the initial version of the code. Other researchers used GPU based open source versions of well known programs (for example, hoomd [17]).

- **Loop Based Parallelism.** Only 11% of researchers utilized loop based parallelism, where programmer written annotations enable a parallelizing system to execute a loop in parallel. The most common form of loop based parallelism was the use of `parfor` construct in MATLAB, which enables execution of multiple iterations of a loop in parallel and requires that the iterations access/update disjoint memory locations.

Only one researcher exploited pipeline parallelism, even though pipeline parallelism is a natural way of extracting performance for many "simulate, then analyze" kind of applications. In these applications, a simulation run outputs a lot of data (typically on the order of gigabytes), which is then analyzed by another piece of code after the simulation ends. Instead of sequentializing the simulation and analysis codes, by pipelining the data to the analysis code as and when it is available, significant overlap between the simulation and analysis can achieved. This in turn would reduce the feedback cycle time drastically.

4. ANALYSIS OF RESULTS

This section presents the conclusions from analyzing the survey results, together with suggestions for future directions for improving the state of scientific computing ecosystems.

Scientists need software tools that are easy to use and require little training.

The gap between the theory and practice of computational science is wide, but education can bridge the gap. Offering formal training courses on software engineering basics as part of the science curriculum at the graduate and undergraduate levels might be one possible option. However, given the current outlook of scientists on software and programming in general ("scientists are not interested in software as a first order concern," as noted by one researcher), education by itself might not be a very effective solution.

A more promising approach is to develop solutions that are customized to the requirements of scientists. Scientists spent considerable time in trying to reinvent existing technologies. For example, several researchers implement matrix multiplication algorithms. These implementations are cache-naïve, sequential, and are not formally tested. There are numerous free Basic Linear Algebra Systems available to scientific programmers [1, 4, 26, 61]. These implementations are cache-optimized, parallel, and well-tested. Since for many researchers, it is often a case of "What is limiting us is not processor speeds but knowledge about programming the machines," it might be well worth the trouble to hire professional programmers to create reusable software solutions even if it works against the economics in the short term.

In the fields of sociology, politics, music, and astrophysics, the use of domain specific languages (DSLs) was nearly universal. DSL programmers report higher productivity and satisfaction compared to scientists who primarily use general purpose, numerical, or scripting languages. They also learned programming skills more easily. Domain specific languages are usually simple to learn due to their goal of targeting specific tasks within each scientific discipline. Although DSLs are not as general as other programming solutions, their higher level specialized programming constructs are best suited for tasks that are repeated sufficiently often within each scientific domain. Interactive Data Language (IDL) [6] is an example of a DSL that is hugely popular amongst astrophysicists. BUGS [50], a DSL for describing graphical models, is almost the *de facto* standard amongst researchers studying politics. Chuck [59] and Max/MSP [7] are DSLs that are heavily used by researchers in Music for audio and video processing. The true challenge for tools developers is to design and implement DSLs that are not only user friendly and have a support system in form of testing and validation tools, but also are able to achieve performance parity with general purpose language implementations.

Many scientists write programs to solve infrastructure problems unrelated to their research. Some scientists had to write utilities to translate between file-formats expected by different tools. For example, many geophysics applications each use a different data format for representing the model of the earth, making them difficult to interact with each other. Adopting uniform standards for data formats maybe one solution, but might impose restrictions on the flexibility of scientific software developed by scientists from wide variety of backgrounds. Tools for automatic data conversion, and automatic generation of documentation [2] based on programmer comments can go a long way in promoting the development of software that is easily reusable.

Most scientific programmers deal with several programming languages. Unfortunately, dealing with many languages forces programmers to tediously write and maintain wrapper code. In well-designed programs, interfaces are stable and change very infrequently. However, many scientific programs are built in an ad-hoc manner with frequently changing interfaces. Tools [19] for automatically generating wrapper code between MATLAB, FORTRAN, C, C++, and many other languages exist, but none of the researchers interviewed were aware of this resource. We suggest prominently integrating such tools into IDEs. Ideally, such tools should be an invisible part of the build process, hiding unnecessary technical details from programmers and allowing them to focus on the underlying science. Additionally, multi-language testing and validation frameworks [36, 57] can be employed to guarantee safety properties at external interface sites.

Scientists should release code to their peers.

The survey reveals that very few scientists release code to their peers. Publicly releasing source code has numerous benefits for scientists and should be encouraged. Publish-

ing source code allows other scientists to reproduce prior work and compare new contributions on an equal footing. Released source code benefits from additional peer review. Other scientists have the opportunity to review the code, fix bugs, improve portability and performance, and extend functionality. Furthermore, the scientific community benefits when source code is freely available for re-use and each scientist does not have to spend time rewriting the same software.

Survey responses indicate that software frameworks for scientific computation allow researchers who are just getting started on research to get up to speed faster. Scientists surveyed used frameworks such as Sleipnir [31], AFNI [21], Quantum Espresso [27], and Dynare [35]. These domain specific scientific coding frameworks provide a variety of builtin features that enable easier implementation and testing of new ideas. However, care has to be taken to ensure frameworks are as close as possible to what scientists actually do, to prevent disuse due to "software bloat." [42]

Releasing scientific code that performs well, is portable, and extensible takes a lot of programming effort. It took many years for the programs listed in Table 2 to reach the level of portability and performance they are at today. Given the interdisciplinary expertise needed for writing robust scientific code, it is desirable to have many theses focused on scientific tools. Unfortunately, scientists are not rewarded for developing and releasing robust scientific software. As noted by two prominent scientists during the survey, faculty generally believe that development of software tools "does not make for a scientific contribution." Similar sentiments echoed included "If you are not going to get tenure for writing software, why do it?" Alarmingly, even "funding agencies think software development is free," and regard development of robust scientific code as "second class" compared to other scientific achievements.

Scientists can benefit immensely from faster computation.

Currently, the research conducted by 85% of researchers would profoundly change with faster computation. Nothing evoked stronger reactions during the interviews than questions regarding impact of faster computation. Responses included "more papers, more quickly," "I'll graduate in 3 years instead of 5," "can get you out of school earlier," "if it is 2x faster, life will be five times better." Several patterns regarding the potential areas of research improvement emerged during the survey, which can be concretely categorized as follows.

- **Accurate scientific modeling.** Across several disciplines, researchers approximate scientific models to reduce running-times. Faster computation enables accurate scientific modeling within time scales previously thought unattainable. Across disciplines, an order of magnitude performance improvement was cited as a requirement for significant changes in research quality. For instance, researchers simulating precipitation/evaporation of earth science processes (Civil and Environmental Engineering), can adopt the use of finer resolution models which are currently avoided due to prohibitive communication costs between fine-grained cells and consequent increased time for convergence. A similar pattern holds for molecular dynamics simulations (Chemical Engineering), where increasing computation speed would not require researchers to relax error thresholds/step sizes to allow for faster convergence. The survey data reveals that 34% of the interviewees would directly benefit from accurate models.

- **Speeding up the scientific feedback loop.** Scientific computation is typically not performed in isolation, but as part of a three step feedback loop: (a) Evolve scientific models (b) Perform computation using models (c) Revise models based on computational results. For 30% of researchers, slower computation in Step (b) leads to an overall slow feedback loop. An example instance observed was in Computational Biology, where different machine learning techniques are iterated over genome/protein data to predict gene interactions/protein structure. Faster computation shortens this feedback loop which in turn results in faster availability of prediction data to the larger scientific community.

- **Wider exploration of parameter space.** Currently, many researchers fit their scientific models to only a subset of available parameters for faster program runs. For instance, psychologists studying human decision making build models that fit only a sparse subset of parameters, despite the potential of obtaining accurate information about human subject by choosing a larger set of parameters. Other cases included the use of a faster but more approximate heuristic for determining shortest path problems in a stochastic network (Operations Research and Financial Engineering). Around 23% of researchers fall into this category. For these researchers, faster computation translates into better heuristics and in eventual broadening of research scope to be more general and realistic.

- **Others.** For 12% of researchers, faster computation leads to better sensitivity analysis to data (lower normalized errors with repeated runs in parallel), faster post-processing of huge amount of experimental data, and reduced effects of queuing time on simulation runs.

Over the past few decades, generations of newer parallel hardware have met the need of faster computation for scientific applications [28]. The continuing doubling of transistors as per Moore's Law [45], even in the presence of power and thermal walls for uniprocessor design, has meant that parallel hardware is now ubiquitous. Faster hardware will no doubt enable accurate scientific modeling and wider parameter space exploration for free in many applications. However, for an increasing number of applications, it is software and not hardware that is on the critical path of performance [62]. Rethinking current software solutions to sustain scalable performance over successive generations of parallel architectures in a cost-effective way remains vital to the success of computational science.

Scientists need tools for managing accuracy-performance tradeoffs in order to create performance portable applications.

Nearly all scientist-programmers in the survey make accuracy-performance tradeoffs. However, researchers often fix the choice of flexible thresholds like error bounds, radius of influence in a simulation algorithm [49], search subspaces, and so on, *statically once* in their program. Often, these static choices lead to programs without performance portability, i.e., programs whose performance is worse when run

on newer parallel architectures, while retaining the same quality of approximation.

Rewriting code manually by taking advantage of the specifics of each parallel architecture is one way to create programs that perform optimally on each architecture. However, the amount of programmer effort involved for scientists in learning new parallel programming paradigms is enormous. As was observed in the survey, very few scientists had knowledge of various parallelization paradigms (Section 3.4) and most admitted a steep learning curve for adopting parallel programming. This situation is further compounded by the emergence of heterogeneous parallel architectures that presuppose knowledge of the underlying hardware [60] on the programmer's part to deliver optimal performance. Consequently, computer scientists should focus on creating tools that provide performance portability without requiring expert programmer involvement.

Recent work [18, 52] within the programming languages and compiler community addresses the accuracy-performance tradeoff problem by stipulating it as a first order language concern. Within this framework, programmers declaratively specify various parameters and customized policies involved in the accuracy-performance tradeoff. These specifications are provided by means of a few annotations to existing sequential programming languages, and do not require programmers to learn a completely new language. The declarative specification gives sufficient freedom for the compiler and runtime to synthesize a performance portable program, with decisions regarding choice of a performance strategy conforming to a required quality of service/approximation delayed until late in a program's life cycle. Typically, these decisions are not taken until complete details of a target parallel architecture and the environment are available.

Scientists need performance analysis and enhancement tools that are an integral part of the software development and build process.

Performance optimizations should focus on parts of code that are most time consuming. Unfortunately, even very experienced programmers may not be able to identify performance bottlenecks without profiling. In the survey, 18% of scientists profile their code. To encourage profiling, performance analysis tools should be interactive, integrated with the IDE, and be made a part of the build process. For instance, profiling should execute automatically after compilation within an IDE and suggestions for optimization offered to the programmer. The design of intuitive graphical user interfaces for visualizations of these suggestions, with cross-references to source code, could go a long way in making programmers use tools as well.

Overall, scientists' approaches to performance and optimization is extremely puzzling. Although 85% of scientists say increasing the performance is important, nearly half of the scientists surveyed never *attempt* optimization. Only about 10% compile with a non-default optimization level and consequently do not benefit from high-benefit low-cost compiler optimizations. Presently, programming tools are designed for professional programmers, and need to be carefully tailored to meet the needs of scientists.

Even though researchers understand the importance of parallelism in accelerating their research, the predominant perception of parallel programming is that of black art ("heard it is notoriously hard to write" on MPI, "scared of it" on shared memory parallelism). The emerging heterogeneity in parallel architectures and explicit programming models is not helping either. Even though researchers seem excited at the potential of newer parallel architectures, they seem overwhelmed by current programming techniques for these machines ("GPUs can give 25x instantly instead of waiting for generations of Moore's Law, but I don't know how to program them," said a prominent scientist). The parallel computing landscape seems too complex for any single solution to be a panacea. Parallel libraries [16, 40], domain specific languages [58], parallel computing toolboxes [46, 54], implicit and interactive parallelization [32, 5], and automatic parallelization [25] are some promising directions that can might be easily adoptable by scientists due to higher level abstractions provided.

5. RELATED WORK

Hannay et al. [30, 43] conducted an online survey of 2000 scientists with a goal of studying the software engineering aspects of scientific computation, from a correctness and program development perspective. The survey was carried out anonymously and conclusions drawn without knowledge of the subject's computing environment. Cook et al. [20, 48], present a survey to exclusively understand the programming practices and concerns of potential parallel programmers. The subjects involved in this survey were attendees of supercomputing conference (SC 1994), and the majority of subjects were computer scientists. By contrast, the survey presented in this paper involved scientists working in diverse fields, and was conducted within the framework of a known (university) scientific computing ecosystem. This survey was carried out in the form of a detailed interview between two of the authors and the subjects. While covering the correctness and software engineering aspects of computational science, the survey presented in this paper also addressed the need for computational performance, and the practices followed by researchers in enhancing performance in their research.

6. CONCLUSION

Overall, the survey reveals that current programming systems and tools do not meet the needs of computational scientists. Most tools assume the programmer will invest time and energy to master a particular system. By contrast, scientists tend to want results immediately. Nevertheless, the survey discovered that virtually all scientists understand the importance of scientific computing, and many spend enormous time and effort programming. Despite this effort, most scientists are unsatisfied with the speed of their programs and believe that performance improvements will significantly improve their research. In many cases, scientists said that increased performance would not just improve accuracy and allow for larger experiments, but would enable fundamentally new research avenues. Overall, we believe the needs of computational scientists are underserved. New tools and techniques are urgently needed to unlock the potential of high-performance computing and accelerate the pace of scientific advance.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. This material is based on work supported by National Science Foundation Grants 1047879, 0964328, and 0627650, and United States Air Force Contract FA8650-09-C-7918.

7. REFERENCES

[1] AMD Accelerated Parallel Processing Math Libraries. http://developer.amd.com/gpu/appmathlibs/pages/default.aspx.

[2] Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[3] Gordon Bell Prize for Peak Performance 2003. http://www.sc-conference.org/sc2003/tech_awards.html.

[4] Intel Math Kernel Library. http://software.intel.com/en-us/articles/intel-mkl/.

[5] Intel Parallel Advisor. http://software.intel.com/en-us/blogs/tag/intel-parallel-advisor/.

[6] Interactive Data Language Online Help. http://idlastro.gsfc.nasa.gov/idl_html_help/home.html.

[7] Max/MSP: An interactive graphical dataflow programming environment for audio, video and graphical processing. http://www.cycling74.com.

[8] Princeton Plasma Physics Laboratory. http://www.pppl.gov/.

[9] School of Engineering and Applied Science (SEAS). http://www.princeton.edu/engineering/.

[10] Terascale Infrastructure for Groundbreaking Research in Engineering and Science. http://tigress.princeton.edu/.

[11] The Carnegie Classification of Institutions of Higher Education. http://classifications.carnegiefoundation.org/.

[12] The Lewis-Sigler Institute for Integrative Genomics. http://www.princeton.edu/genomics/.

[13] The Princeton Institute for Computational Science and Engineering. http://www.picscie.princeton.edu/.

[14] The Princeton Institute for Science and Technology of Materials (PRISM). http://www.princeton.edu/prism/.

[15] S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand, et al. Geant4-a simulation toolkit. *Nuclear Instruments and Methods in Physics Research-Section A Only*, 506(3):250–303, 2003. http://www.geant4.org/.

[16] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: A standard template adaptive parallel C++ library. *International Workshop on Advanced Compiler Technology for High Performance and Embedded Processors*, 10, 2001.

[17] J. Anderson, A. Keys, C. Phillips, T. Dac Nguyen, and S. Glotzer. HOOMD-blue, general-purpose many-body dynamics on the GPU. *Bulletin of the American Physical Society*, 55, 2010.

[18] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, pages 198–209, 2010.

[19] D. Beazley and P. Lomdahl. Feeding a large-scale physics application to Python. In *Proceedings of the 6th International Python Conference*, volume 6. Citeseer, 1997.

[20] C. R. Cook, C. M. Pancake, and R. A. Walpole. Are expectations for parallelism too high? a survey of potential parallel users. In *SC*, pages 126–133, 1994.

[21] R. W. Cox. AFNI: Software for analysis and visualization of functional MRI. *Computers and Biomedical Research*, 29:162–173, 1996. http://afni.nimh.nih.gov/afni.

[22] M. de Hoon, B. Chapman, and I. Friedberg. Bioinformatics and computational biology with Biopython. *Genome Informatics Series*, pages 298–299, 2003.

[23] P. J. Denning. Computer science. In *Encyclopedia of Computer Science*, pages 405–419. John Wiley and Sons Ltd., Chichester, UK.

[24] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, July 1999. http://hmmer.janelia.org/.

[25] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the automatic parallelization of four Perfect-Benchmark programs. *Languages and Compilers for Parallel Computing*, pages 65–83, 1992.

[26] M. Frigo and S. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216 –231, 2005. http://www.fftw.org/.

[27] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. Dal Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, and R. M. Wentzcovitch. QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502 (19pp), 2009. http://www.quantum-espresso.org.

[28] S. Graham, M. Snir, and C. Patterson. *Getting up to speed: The future of supercomputing*. National Academy Press, 2005.

[29] W. Gropp, E. Lusk, and A. Skjellum. Using MPI-Portable Parallel Programming with the Message-Passing Interface. *Sci. Program.*, 5:275–276, August 1996.

[30] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? *Software Engineering for Computational Science and Engineering, ICSE Workshop on*, 0:1–8, 2009.

[31] C. Huttenhower, M. Schroeder, M. D. Chikina, and O. G. Troyanskaya. The Sleipnir library for computational functional genomics. *Bioinformatics*, 24(13):1559–1561, July 2008. http://huttenhower.sph.harvard.edu/sleipnir/.

[32] W.-m. Hwu, S. Ryoo, S.-Z. Ueng, J. H. Kelm, I. Gelado, S. S. Stone, R. E. Kidd, S. S. Baghsorkhi,

A. A. Mahesri, S. C. Tsao, N. Navarro, S. S. Lumetta, M. I. Frank, and S. J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 754–759, New York, NY, USA, 2007. ACM.

[33] C. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *Computer*, 39(4):93–95, 2006.

[34] E. Jones, T. Oliphant, and P. Peterson. SciPy: Open source scientific tools for Python. 2001. `http://www.scipy.org/`.

[35] M. Julliard. Dynare: A Program for the Resolution and Simulation of Dynamic Models with Forward Looking Variables Through The Use of Relaxation Algorithm. *CEPREMAP*, 2005. `http://www.dynare.org/`.

[36] V. Karakoidas and D. Spinellis. J%: Integrating domain-specific languages with java. In *Panhellenic Conference on Informatics*, pages 109–113, 2009.

[37] W. J. Kaufmann and L. L. Smarr. *Supercomputing and the Transformation of Science*. W. H. Freeman & Co., New York, NY, USA, 1992.

[38] D. Komatitsch, J. Labarta, and D. Michéa. A simulation of seismic wave propagation at high resolution in the inner core of the Earth on 2166 processors of MareNostrum. *High Performance Computing for Computational Science-VECPAR 2008*, pages 364–377, 2008. `http://www.geodynamics.org/cig/software/specfem3d`.

[39] D. Komatitsch, S. Tsuboi, C. Ji, and J. Tromp. A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the earth simulator. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, SC '03, pages 4–, New York, NY, USA, 2003. ACM.

[40] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[41] X. Liang, D. P. Lettennmaier, E. F. Wood, and S. J. Burges. A simple hydrologically based model of land surface water and energy fluxes for general circulation models. 99:14415–14428, July 1994. `http://www.hydro.washington.edu/Lettenmaier/Models/VIC/`.

[42] J. McGrenere. Bloat: the objective and subject dimensions. In *CHI'00 extended abstracts on Human factors in computing systems*, pages 337–338. ACM, 2000.

[43] Z. Merali. Why scientific programming does not compute. *Nature News*, 467, 2010.

[44] C. Moler. *Numerical computing with MATLAB*. Society for Industrial Mathematics, 2004.

[45] G. E. Moore. Readings in computer architecture. chapter Cramming more components onto integrated circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[46] J. Mullen, N. Bliss, R. Bond, J. Kepner, H. Kim, and A. Reuther. High-productivity software development with pmatlab. *Computing in Science Engineering*, 11(1):75 –79, jan.-feb. 2009.

[47] T. Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing, 2006. `http://numpy.scipy.org/`.

[48] C. Pancake and C. Cook. What users need in parallel tool support: Survey results and analysis. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 40–47. IEEE, 2002.

[49] S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, (1):1–19, March. `http://lammps.sandia.gov/`.

[50] M. Plummer. JAGS: Just Another Gibbs Sampler. `http://www-ice.iarc.fr/~martyn/software/jags/`, 2011.

[51] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical recipes in Fortran: the art of scientific computing*. Cambridge university press, 1993.

[52] M. Rinard. Probabilistic accuracy bounds for perforated programs: a new foundation for program analysis and transformation. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 79–80, New York, NY, USA, 2011. ACM.

[53] A. Sameh, G. Cybenko, M. Kalos, K. Neves, J. Rice, D. Sorensen, and F. Sullivan. Computational science and engineering. *ACM Comput. Surv.*, 28:810–817, December 1996.

[54] G. Sharma and J. Martin. MATLAB®: a language for parallel computing. *International Journal of Parallel Programming*, 37(1):3–36, 2009.

[55] D. E. Stevenson. Science, computational science, and computer science: at a crossroads. In *Proceedings of the 1993 ACM conference on Computer science*, CSC '93, pages 7–14, New York, NY, USA, 1993. ACM.

[56] J. Stone, T. Gardiner, and P. Teuben. Athena MHD code project. `http://trac.princeton.edu/Athena/`.

[57] G. Tan and G. Morrisett. Ilea: inter-language analysis across java and c. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 39–56, 2007.

[58] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36, June 2000.

[59] G. Wang, P. Cook, et al. ChucK: A concurrent, on-the-fly audio programming language. In *Proceedings of International Computer Music Conference*, pages 219–226. Citeseer, 2003.

[60] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 156–166, New York, NY, USA, 2007. ACM.

[61] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. *SC Conference*, 0:38, 1998.

[62] N. Wirth. A plea for lean software. *Computer*, 28(2):64–68, 1995.