Software Tools for High-Performance Computing: Survey and Recommendations

BILL APPELBE[1] AND DONNA BERGMARK[2]

[1]*College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280*
[2]*Cornell Theory Center, Cornell University, Ithaca, NY 14853-3801*

ABSTRACT

Applications programming for high-performance computing is notoriously difficult. Although parallel programming is intrinsically complex, the principal reason why high-performance computing is difficult is the lack of effective software tools. We believe that the lack of tools in turn is largely due to market forces rather than our inability to design and build such tools. Unfortunately, the poor availability and utilization of parallel tools hurt the entire supercomputing industry and the U.S. high performance computing initiative which is focused on applications. A disproportionate amount of resources is being spent on faster hardware and architectures, while tools are being neglected. This article introduces a taxonomy of tools, analyzes the major factors that contribute to this situation, and suggests ways that the imbalance could be redressed and the likely evolution of tools. © 1996 John Wiley & Sons, Inc.

1 INTRODUCTION

This article represents contributions from users, vendors, and tool builders; it is based on discussions among participants from the 1993 Workshop on Parallel Computing Systems in Keystone, Colorado, the 1994 Workshop on Debugging and Performance Tuning for Parallel Computing Systems in Chatham, Massachusetts, and the 1994 and 1995 Ptools consortium meetings in Mountain View, California. There have been several other workshops on this topic, such as the Pasadena Workshop on System Software and Tools for High Performance Computing documents [6], and proposals such as the National Software Exchange. This article is an attempt to summarize the conclusions of these meetings, and to suggest courses of action. The authors' experiences ranges from tool building and maintaining, to programming, advising users, and managing parallel systems.

Few scientists willingly undertake the arduous task of parallelizing their programs. Until fairly recently, the majority of scientists did not need to bother—few had access to paralllel computers, and the fastest computers (supercomputers or high-performance computers) were primarily vector versions of parallel architectures.

Today, all the high-performance computer vendors are marketing massively parallel computers, and for "grand challenge" problems parallel com-

Received November 1994
Revised October 1995

Many attendees at the May 1993 Workshop on Parallel Computing Systems, at Keystone, Colorado, contributed to this article, especially Gail A. Alverson from Tera Corp. and Wayne Smith from Intel. The article also includes discussions from the October 1994 Workshop on Debugging and Performance Tuning for Parallel Computer Systems, at Chatham, Massachusetts.

Scientific Programming, Vol. 5, pp. 239–249 (1996)
CCC 1058-9244/96/030239-11

puters are indispensable. The architecture of parallel computers varies widely, from parallel and distributed architectures to networks of workstations. To utilize the potential performance of such parallel computer systems, a range of new and enhanced software tools are needed. Parallel programming is not just more difficult, it is also "chaotic" in nature. Small changes in the size of the problem, the number of processors, and so on often lead to dramatic changes in performance, sometimes in the "wrong direction." Rerunning the same program with the same data often produces significantly different performance results.

Despite these challenges, many parallel computer users and vendors doubt the value of tools [8]. Users try to debug their programs with print statements, and they avoid run-time software monitoring of programs because of its overhead or side effects. Early high-performance computers came with almost no tools, and programmers often resorted to assembler language. For example, originally floating point systems thought that a Fortran compiler was unnecessary for their system.

In fact, it is almost always possible to develop programs without using tools. However, we believe that the lack of usable tools has greatly inhibited the effective use of modern high-performance computers: Either scientists are "turned off" from solving problems these computers are capable of, or shared resources are wasted.

The majority of scientists working on grand challenge problems are aware of the need for tools. The report from the Pasadena Workshop on System Software and Tools for High Performance Computing documents these needs in detail [6, chapters 3, 4].

Unfortunately, the current tools (other than compilers) are very limited in their capabilities and often do not meet the needs of scientists. Vendors supply few tools, and the tools available from research laboratories and universities are generally only prototypes.

Supercomputers have polarized the scientific and engineering programming community into two groups, each with its own set of unmet tool needs [7]:

1. "TeraGeeks": These are scientists, turned programmers, who will stop at nothing to achieve peak performance and solve bigger problems. They want to know exactly what is happening at the machine level, but disdain any tools that "get in the way," such as parallelizing compilers and multiuser operating systems. Most teraGeeks do insist on basic UNIX tools on all processors, such as **dbx** [6]. They regard squeezing the last MFLOP out of even the most recalcitrant architecture as another grand challenge. The Gordon Bell award has been established to reward these pioneers.
2. Scientists: Scientists first and foremost want to solve their problem and share their results with the community. They have no time to waste learning tools or rewriting and tuning their programs (which they may not have written themselves). They just want their "dusty decks" to run faster.

Of course, most high-performance computer users are not at these two fictional extremes. However, the polarization is real. The majority of potential users fit into the scientist category.

There is another important community that is often neglected, the tool developers. Tool developers build tools for other users (sometimes targeted at TeraGeeks, sometimes at scientists). Some tool developers are employed by vendors (which we call vendor tool groups), but the vendor tool groups are generally relatively small (we are not classifying compilers or operating systems or other essential software as tools).

As explained below, vendors generally do not supply any significant resources for tool builders because they see no economic incentive. Hence, the burden of experimental tool development falls almost entirely upon groups such as academic researchers and third-party tool vendors. Academic researchers often adopt the philosophy of "build a tool and they will come." Unfortunately, this approach almost never works. Academic researchers often understand little about the applications communities need and frequently never even try to use their tools for real applications. The successful tools from academia and research laboratories are invariably built by teams that include applications developers.

The needs of tool developers are similar to the TeraGeeks, except that the interface that tool developers need is at a library level rather than a tool level. For example, a TeraGeek might need a tool that will monitor the message traffic in a distributed memory multiprocessor. By comparison, a tool developer building a tool to visualize hot spots needs a library call that will return detailed information about message traffic.

The fundamental problem can be stated as follows:

> There is a serious lack of effective software tools, which leads to wasted computer resources and inhibits the use of high-performance parallel computers by scientists.

There are many factors contributing to this situation, some of which are listed below. Some of these, such as the limited market for high-performance computers and the limited financial resources of most vendors, are unavoidable. However, some of these problems are being tackled, and this article concludes with some realistic approaches that could significantly improve the overall situation.

Most programmers are used to the "edit, compile, run, debug" cycle of typical program development. The same cycle applies to parallel programming, but the process is more complex. Creating an efficient parallel program is far more difficult and machine dependent than creating efficient sequential programs. On some massively parallel systems, efficient programs can be hundreds of times faster than an inefficient program. Hence, new tools (that were not needed for sequential programs) are needed to help programmers transform and tune their parallel programs for efficiency.

The major reasons why parallel programming tools lag so far behind peak performance are discussed below. Not all the reasons are as important as others, and some reasons are largely mythical:

1. Typically, hardware vendors supply very limited tools.

   True. The reason is that there is very little economic incentive to supply better tools. For example, requests for proposals (RFPs) rarely specify tools. Most RFPs specify a bare minimum of software, such as a compiler. Emphasis on RFPs is on reaching peak performance levels on local benchmarks (to which manufacturers devote considerable effort). Software tools are not needed to sell machines. Of course, this could be viewed as a "chicken and egg problem." RFPs do not specify tools because nobody has them.
2. The majority of available tools are not production quality.

   This is unfortunately true. The majority of available tools crash readily, have poor user interfaces, and severe limitations. This is because the majority of tools are developed as prototypes by academic and research organizations (rather than provided as products by vendors). However, even vendor tools are often buggy. The reason for this is the limited budget of the vendor's tool groups, the push to market products ASAP, and the stress that large, high-performance applications put on software tools.
3. There is too much diversity, and too few standards, in high-performance computing.

   This is unfortunately true, but an improving situation. Diversity is not necessarily bad: Every different architecture or programming language is "best" for some application. However, diversity hurts tool availability. The race in the market is for maximum megaflops, which has led to a plethora of widely different architectures. Most tools are applicable only for a limited range of environments. Until recently, every vendor had its own parallel programming dialects. Now, finally, there are some emerging standards, such as PVM and MPI for distributed computing, and high-performance Fortran (HPF) and Fortran 90 for parallel computers. Unfortunately, standards such as HPF are complex and will require considerable tool development effort. A further problem is the continuing evolution of architectures. Widespread standardization is not going to happen until and unless "the dust settles." The recent shake out in parallel computing vendors, including Thinking Machine's and Kendall Square's bankruptcy, may result in a reduced range of architectures and better tool support for those that remain.
4. Users are uninterested in using the available tools.

   This is largely true, and is a basic problem. It is hard to blame the users for this situation, but ignoring tools is a basic human trait.* Users are often uninterested in learning to use tools, no matter how "friendly" the interface.

   Pessimistic tool builders and vendors claim

* Recall Aesop's fable of a wood-cutter furiously chopping away with a blunt axe.

that some scientists will not use a tool unless: (1) it is absolutely necessary, as in learning a parallel Fortran dialect; (2) it is obvious what the benefits or uses of the tool are; (3) the learning curve is outweighed by the perceived benefits; and (4) the tool is industrial strength.

Even if users can be convinced to try a tool, they soon abandon it when it is difficult to learn, or use, or crashes. Once users have abandoned tools they are very reticent to try them again even if later releases of the tool have fixes and improvements.

Unlike compilers, which are considered mandatory, most tools merely improve the productivity of scientists or resource utilization, so there is little incentive to use them. The obvious solution is to make these tools easier to use, and substantially higher quality. However, this requires substantial effort and a change in the way that tools are developed, as decribed later.

5. Available tools have poor user interfaces, or do not give users what they want.

   True. Unfortunately, many amateur and professional tool builders equate better user interfaces with "more and fancier windows." We recall a vendor at SuperComputing '92 proudly displaying a user interface so cluttered with overlapping windows, widgets, and controls that even compiler experts could not comprehend what the interface displayed. Users might be in awe, but they would hardly want to use the tool.

   It seems as though some tool builders want to cram every conceivable feature into their tools, without evaluating what users need. Most developers have little clue as to what users need or want, so they follow the strategy "when in doubt, add more features."

6. We do not know how to build the tools, or the tools are too difficult to build.

   This is largely a myth. For example, the software and human factor technology for building useful, effective performance monitors, analyzers, and debuggers is fairly well understood.

   There are a few tools that we do not have the current software technology to build, and may not in the foreseeable future. For example, we do not know how to automatically compile any sequential program into efficient code for a distributed memory multiprocessor [1, 11]. However, for alternative approaches such as user-supplied directives and interactive compilers, we have the software technology we need to build prototypes. In many cases, the tools that are needed are very simple (such as highly accurate timer library routines), yet unavailable.

2 A TAXONOMY OF PARALLEL PROGRAMMING TOOLS

Many different parallel programming tools have been developed or proposed, either with the goal of improving programmer productivity or computer utilization. The simplest classification of tools is by their functionality. The list below (adapted from [3]) gives the major classes of tools, together with some representative examples in parentheses:

Compilers: Most vendors of parallel systems supply compilers for parallel dialects of C and Fortran. Sometimes they are integrated with parallelizers: The input to the compiler can be either a sequential or a parallel program.

Program Restructurers and Parallelizers: Restructuring tools convert sequential, or partially parallel, programs into efficient parallel programs (comparable to that of a hand-parallelized program written by an expert programmer). Unlike compilers, parallelizers transform source code into source code rather than generate object code (KAP). There are several different classes of parallelizers, depending on the target architecture: instruction-level parallelism, vector and single instruction multiple data (SIMD) parallelism, and task-level parallelism. General task level parallelism is beyond the scope of production parallelizers.

Program Specification and Construction: Specification tools are used to construct parallel programs, usually by composing sequential code fragments (HeNCE).

Static Analyzers: Static analysis of a program can detect both potential bugs (such as race conditions) and poor resource utilization (e.g., processor, memory, or cache). Static analysis can include predicting, or simulating, the execution of a program.

Parallel Debuggers: Parallel debuggers extend traditional debuggers, such as *gdb*, with the ability to control and monitor execution of individual tasks. Parallel debuggers should be capable of detecting parallel errors at run-time, such as race conditions (Xmdb).

Execution and Performance Analyzers: Execution analyzers tell the user what happened during the execution of a program. Unlike debuggers, they are post-mortem and are consequently less intrusive. Performance analyzers determine resource bottlenecks, and may suggest source code modifications to remove these. Performance analyzers can include tools to automatically instrument programs to gather *trace data* for later analysis. Examples of such tools are xpvm (for PVM) and Pablo, from the University of Illinois.

Libraries: Canned parallel libraries and packages can greatly reduce development effort (BLAS).

The above list is not exhaustive. For example, tools that generally can be used equally well with parallel and sequential programs (such as configuration control) have been omitted. Also, the list excludes tools used primarily by systems administrators (such as load monitors).

Some tools provide functionality from different categories. For example, FORGE [10] provides both static analysis and profiling (performance monitoring). Some of these tools have graphical user interfaces (GUIs).

Tools can also be classified by other attributes, such as:

- Level of abstraction

  Does the tool work at the application, algorithm, language/program model, operating system/run-time library, or machine (instructions, cache, memory) level? Scientists usually want tools that work primarily at the application or program level. TeraGreeks and tool developers want lower level tools and interfaces.
- Portability/adaptability

  What range of environments or platforms can the tool operate in?
- Level of integration

  When tools provide several functions, an important issue is how well are these integrated (e.g., by using a program database and common interfaces).
- Level of presentation

  Level of presentation measures the presentation quality of the user interface. A few years ago, the majority of tools were batch and dumb terminal oriented. Interactive user interfaces and workstation clients were only slowly adopted by high-performance computer users. However, the use of such systems has become widespread. Hence, most recent tools have adopted X-Windows/Motif-based interfaces. Unfortunately, the usability of such interfaces by scientists tends to be low, as noted earlier.

2.1 Parallel Programming Tool Status

As noted earlier, the majority of current tools are:

- Limited in availability and applicability
- Not robust (crash easily)
- Generally only usable by the tool developer, or experts with a computer science background (which excludes the overwhelming majority of scientists) because they have poor interfaces
- Poorly integrated (do not work well with other tools)

For each of the classes of tools above we can summarize their status using the following metrics:

Quality:
Research, prototypes, or production.

What is the overall quality of the tool? Is it "production quality," implying good performance, high functionality, and/or ease of use?

Availability:
Good, fair, or poor.

Are there prototype tools that show promise of going into production fairly soon? Is the market push strong? Do we know how to build such tools?

Of course, Table 1 is subjective, but it does indicate where problems lie and where research and prototype development is needed. Following the table we discuss some of the reasons for the classification.

There are several bright spots in tools. Almost all parallel computers now come with parallel debuggers. Vendors of "classic" supercomputers (primarily IBM and Cray) have more mature tools (e.g., Cray's ATExpert for performance analysis [5]). However, good tools are less available for the massively parallel and distributed computers. SIMD computers need somewhat fewer tools, and the tools are simpler because SIMD computers are

Table 1. Parallel Programming Tool Status

Class	Quality	Availability
Compilers	Production	Fair
Program specification and construction	Prototypes	Fair
Program restructurers and parallelizers	Production/prototypes	Fair
Static analyzers	Prototypes	Fair
Parallel debuggers	Production	Good
Execution and performance analyzers	Prototypes/production	Good
Libraries	Prototypes/production	Good

deterministic. Unfortunately, SIMD computers have been largely relegated to specialized applications.

Most vendors have interactive performance analysis tools, with varying levels of maturity and capabilities. Tools that create, transform, and analyze source code (from compilers to static analyzers) are in the worst shape. Although most vendors have parallel Fortran and C dialects, there are no production tools that will consistently transform sequential to efficient parallel code. There are tools that will parallelize source code for some parallel computers, but they usually do not incorporate a performance model. Thus parallelizing tools tend to choose transformations in an ad-hoc manner. Again, traditional shared memory multiprocessors have the best parallelizers.

The situation is bleakest for distributed memory multiprocessors. The advent of HPF should improve this situation, but it is too soon to tell whether HPF compilers and parallelizers will be effective. Since HPF was announced a few years ago, there has been a lot of interest by programmers, but production compilers have only begun to emerge. HPF tools are in early development stages by various research groups. It remains to be seen whether HPF is too complex (for implementors and users) to be a successful standard.

By contrast, PVM is widely available, has been used for several years with C and Fortran bindings, and is accompanied by a growing range of tools. (MPI is now growing in popularity and availability as well.) Unlike HPF, PVM and MPI take the approach that the programmer must completely specify the distribution of data and all communication. The goal of HPF is to make distributed memory programs portable, yet specifiable in a high-level language. By contrast, the goal of PVM is to provide a simple, portable parallel programming *library* that can be used by a skilled programmer. Although the PVM library is portable, there is no implication that a PVM program's performance will be portable without redesign or reprogramming.

Because PVM is much simpler and more portable than HPF, PVM has a wide and growing user base, despite its performance problems for communication-intensive distributed applications. Message-passing libraries are probably at too low a level for most scientists, but it is being used as a target language by tools such as APR's distributed memory parallelizer.

The PVM vs. HPF debate hinges on a deep but unresolved question: *Is it practical to build a compiler that can automatically determine the optimal distribution of data and computation for most sequential (or HPF) programs for a specific distributed memory architecture?*

Experience has shown that some obvious compiler chores are impractical. For example, in the 1970s, there was considerable interest in compilers that did not just detect syntax and semantic errors, but also fixed them with sufficient accuracy to be able to generate a "correct program." Such error-correcting compilers have largely vanished, as the corrected programs rarely were what the programmer intended.

The HPF vs. PVM debate will largely depend on whether the HPF community can deliver usable HPF compilers that deliver good performance for a wide range of applications.

2.2 Tool Interfaces

A serious impediment to tool developers is the lack of support provided by manufacturers. Tool developers are often faced with building their tools from scratch, and reverse engineering the performance characteristics of parallel computers. Consequently, the prototype tools developed by universities and research laboratories are often severely limited in functionality. Alternatively, researchers are discouraged from even developing tools because of the immense cost of developing prototypes. For example, parallelizing tools are a fruitful research area (in program transformations, performance prediction, and so on). Yet there are only a few public domain tools capable of parallelizing full Fortran-77, and these are limited in functionality. To parallelize Fortran, tool developers must build yet another Fortran compiler front-end (scanning, parsing, semantic analysis) from scratch.

Because no vendor is going to document and deliver their compiler technology and source code into the public domain, other approaches are needed. One approach would be a national compiler infrastructure, a funded effort to develop an open, public domain, multisource/target parallelizing compiler (or at least a front-end for such a compiler). Indeed, a meeting was convened by funding agencies to address just this topic. However, the effort foundered before it got off the ground due to the difficulty of the task and the probability that it might end up as yet another government-funded large software disaster (as happened with the Department of Defense [DOD]-funded Ada programming environments, the ALS and AIE).

A much better, bottom-up approach, would be for compiler, tool developers, and vendors to agree on a minimal set of simple compiler interface standards that everyone could support for sequential programming languages such as C and Fortran. Common interface standards have proved their worth in other tool domains such as debuggers.

The reason why vendors do not supply interface support is both economic and competitive. The most effective way that vendors could support tool developers would be by providing applications programming interfaces to their compilers and tools, but this is a costly undertaking. Vendors fear that allowing tool developers access to performance characteristics and compiler internals could put them at a competitive disadvantage.

Another reason why vendors to not supply interface support is that they do not want to be bound to a documented interface. They fear that some customers will complain if locally developed tools break because an interface changed from out beneath the tool. This problem would be greatly ameliorated by standard interface declarations, because vendors would (presumably) become eager to supply the standard interface. This situation may also be changing due to pressure by developers and recognition by vendors of the usefulness of tools.

The following section summarizes the interfaces needed by tool developers. Many of these are also useful to users, as noted earlier.

Accurate Timers and Resource Statistics

Both tool builders and users need access to libraries that provide accurate information about the system resource utilization of program tasks, such as memory, central processing unit (CPU), and run-time library statistics. Tool builders typically obtain such information by using wrapper functions around library calls, and calls to the system clock or event counters. For timers, what is really needed is one call, something like

```
CALL TIME(ElapsedUserSec,ElapsedUserNanoSec)
```

which returns two floating-point numbers for elapsed user-based CPU time (i.e., time spent on THIS job). The first is seconds since the start of a job, and other is the nanoseconds remainder. This way you can count up hours without overflow, or fine-tune on the nanosecond basis with high resolution.

Another very important statistic is cache hit rate. Poor cache utilization is often the cause of anomalous performance results. On most computers it is almost impossible to determine cache statistics. In general, some standard interfaces to hardware performance monitors would be helpful.

Trace Data Collection

Once run-time libraries have been instrumented, it is often necessary to save this information in *trace files*. The I/O overhead of gathering traces can be prohibitive if it is not provided by the run-time library itself. Ideally, vendors should provide a standard, extensible trace file format and library calls to selectively enable tracing at relatively low overhead. If the overhead of tracing can be kept below 10%, then tracing can be the default.

Ideally, trace data formats can be standardized across platforms, or tools can use trace specifications. Some effort has been devoted to trace standardization, but it has not gone far [9]. We note, however, that there is a small trend toward converting trace files into SDDF format [2] for interchangeable tool use.

Syntax Analysis and Program Instrumentation

Tools such as performance monitors and execution analyzers are far more effective if their interface is at the program level. That is, performance is reported in terms of source program structure and constructs, and these also serve as the reference point for insertion of instrumentation. Any tool whose interface is at the program level requires syntax and semantic analysis of the source program. Ideally, this can be provided by interfaces supported by the vendor's compiler to data structures such as intermediate code (e.g., expression trees), symbol tables, and call graph and flow graphs. Ideally, each of a vendor's compilers should provide this information in an easily accessible format with efficient lookup. As noted earlier, vendors have generally not been willing to do this, and no standard format for this compiler information even if vendors were to supply access to it.

Currently, tool developers who want to provide a source program interface are thus usually faced with two unpleasant choices: (1) Simulate syntax analysis and parsing using simple tools such as *perl*, *yacc*, and *lex*. These are inaccurate, and hence only useful in a prototype. (2) Build or adapt a compiler front-end. This is a major undertaking, especially if the front-end is going to parse exactly the same programming language dialect as the vendor's compiler.

Even if compilers do not provide interfaces to their data structures, they should provide information about optimizations that were made, related to the source, so that subsequent tools can relate to source. In addition, debugging should be possible on optimized code. On some systems the debugging option for the compiler (which tells the compiler to generate information needed by the source debugger) cannot be used with the optimization option. This further discourages users from using the debugger.

Simulators and Performance Data

The ability to "predict" the performance of a program is critical for tools such as performance monitors and parallelizers. Performance monitors can use prediction to determine the causes of poor resource utilization, whereas parallelizers can use prediction to determine the quantitative benefits of source program transformations.

Predicting the performance of a parallel program, or progrm fragment, is extremely difficult. It requires a detailed model of the architecture along with instruction-level timings. Even then, effects such as cache misses and processor scheduling make accuracy difficult to achieve. Consequently, very few tool builders have attempted performance prediction. Even obtaining accurate timings for instructions and run-time libraries is difficult, although portable low-level benchmarks or training sets are useful.

The most accurate performance prediction usually comes from simulators. Although most vendors develop these when designing and prototyping their systems, they are typically not available to users or other tool builders.

Integrated Environments vs. Toolkits

Few will question the usefulness and effectiveness of integrated programming environments. For example, PC-based programming environments, such as Borland's C++ and Microsoft's Visual C++, put most workstation environments and all high-performance computing environments to shame.

Integration comes at different levels. Tight integration, typical of PC programming environments, implies tools that share a common program database and GUI interface. Loose integration implies that tools cooperate and interoperate.

Although integration is a laudable goal, there are several caveats that make toolkit integration unlikely or infeasible for high-performance computing in the near future:

1. Integration of tools can be helpful only if the tools are already mature with standard interfaces. From a user viewpoint, it makes no sense to be a guinea pig for a prototype integration of a prototype performance analyzer with a prototype debugger.
2. Integration must have clear-cut goals, and must start with well-defined interfaces between the components.
3. Synergy must result from the integration; tools should make use of each other so that the tools together are better than they are apart. At the very least, integration should mean that tools have a common interface.

Integration should not mean closed systems, especially in the evolving environment of high-performance computing. Ideally, an integrated environment would allow a plug and play" approach to tools: Combining the best performance analyzer with the most appropriate debugger, and so on.

This leads to a fundamental rule on tool integration: *Before commencing to integrate tools for a production quality toolkit, it is vital that the interfaces between the tools be well defined, well understood, and preferably standardized.*

2.3 Vendor Support

The overall lack of support by vendors for tool developers should not be taken as a blanket condemnation of the high-performance computer industry. As noted, they are merely responding to market demands and pressure. Vendors will supply and develop whatever tools a majority of potential buyers demand. Unfortunately, once a machine is delivered users generally have far less clout.

There are some examples of high-quality tools provided by vendors, such ATExpert from Cray Research (a performance analysis tool). Also, vendors occasionally provided interfaces and tools for developers, such as the trace facility provided by IBM's Parallel Environment for the SP1/2 [4]. However, as far as we know, there is no consistent or systematic attempt by a vendor to provide support for third-party tool developers or to make their systems easier to develop tools for.

3 IMPROVING THE STATUS QUO

Although the current state of production quality software tools for high-performance computing is dismal at best, the situation is not without hope. Some of the bright signs are:

1. Growing acceptance of workstation technology and standard GUIs among scientists.
2. Growing market for high-performance computers, and access to high-performance computers by potential tool builders.
3. Increasingly successful efforts at language standardization.
4. Emergence of user groups, such as Ptools, (WWW home page: `http://www.llnl.gov/ptools/ptools.html`).
5. The availability of high-quality software tools for commercial software development. PC-based tools such as *Visual Basic* and *PowerBuilder* put most high-performance computing tools to shame.

However, we believe that a lot could be done to improve the availability and effectiveness of production-quality software tools for a relatively modest investment of resources. Such an investment would be more than repaid by the improvements in utilization of high-performance computers.

3.1 Improved Funding for Prototype Tools

Universities and research laboratories develop dozens of interesting and potentially useful parallel programming tools every year. Yet there is no mechanism in place to convert these tools to useful prototypes and perhaps eventually to production software (adoption by vendors).

A major problem is that there is relatively little funding available from agencies such as the National Science Foundation for the purpose of converting prototype to production tools. A great deal of funding is available for research, but little for converting research into production. Given the current US political climate, it is more likely that government funding will decline, rather than expand, in the near future. Industry, on the other hand, is unwilling to invest in tools that are not proven to be useful.

Tools can be classified in three levels [6]:

Research prototypes

These are typically the product of a doctoral dissertation or research laboratory project. They are primarily "proofs on concept," are only usable by a skilled person, have very limited application, and little documentation. Usually the source code is freely available to encourage others to tinker with the tool.

Beta-version tools

These result from spending considerable time fixing and enhancing the research prototypes. Often little functionality needs to be added. The effort is devoted to making the tool more reliable, robust, easier to use, and so on. Nevertheless, these tools still often have many undocumented limitations, and have not been subjected to an intensive testing and evaluation process. Also, such tools have very little support (help and fixes available if something goes wrong).

Production tools

These are commercial-strength tools typically sold by vendors or third parties. They have support.

Typically it takes an order of magnitude more effort to go from one level to the next. Vendors are willing to make the effort to convert successful beta-version tools into production tools. What is needed is funding to convert research prototypes into beta-version tools. Part of the Federal HPC initiative should be spent to fund and support the conversion of promising research prototypes into beta-version tools.

3.2 Supercomputer Purchasers Insisting on Tools

No informed consumer would purchase a car based largely upon its maximum speed. Yet, high-performance computer purchases are largely based on performance (as opposed to usability). Until this changes, vendors have little incentive to devote resources away from benchmarks and toward tools. Buyers of parallel systems should insist on tools as a purchase requirement.

3.3 Improved Tool Interfaces Provided by Vendors

As noted earlier, vendors provide relatively little support for tool developers. There is no financial incentive to do so, and providing access to details of machine performance could be embarrassing if used by competitors.

Unfortunately, such a view is short-sighted, and the high-performance computer business is very much driven by market and short-term goals. As noted earlier, the interfaces that tool developers need overlap those needed by many users (such as more accurate resource statistics). More active user-groups and more support by vendors for internal tool development would improve this situation.

3.4 Improved Usability and User Interfaces

Currently, the human-computer interface (HCI) community has shown relatively little interest in parallel computing. Part of this is due to the relatively small size of the parallel computing community, and part due to the lack of understanding and interaction between the two communities. What is needed are more HCI experts within the vendor's tool groups, and the involvement of HCI facilitators between users and developers (at the stage of conversion of research prototypes into beta-version tools).

3.5 Standardization

The emergence of standards, and the insistence of the user-community upon vendors supporting these standards, will significantly improve the quality and portability of tools. As noted earlier, the situation with regard to standardization is improving, although the industry is still rapidly evolving.

4 CONCLUSION

In summary, it is fairly obvious that the lack of effective software tools has seriously impeded the effective use of high-performance computers by scientists and researchers. What is not so obvious is what, if anything, can be done to improve the situation. Market forces and maturity are gradually leading to better tools, but the status quo leads to a great deal of frustration among high-performance computer users and wasted resources. Each group bears some responsibility for the situation.

Funding agencies need a coordinated plan to carefully target funding at tool development (including transitioning research tools into industrial-strength tools), and studies of tool effectiveness, as discussed earlier. It is not enough merely to fund applications development and the development of new parallel computers, in the expectation that tools will either materialize or are not necessary.

Vendors need to support open interfaces to their tools, such as compilers, debuggers, and simulators, and devote more of their budgets to tools. Those who purchase high-performance computers need to place more emphasis on tools and tool quality in RFPs and contracts.

Tool developers need to focus on the usability of their tools rather than publishability.

Users need to let tool developers and vendors know what they want, and be willing to try new tools.

Everyone needs to be more aware of others' needs, e.g., by joining consortia such as Ptools.

REFERENCES

[1] B. Appelbe, C. McDowell, and K. Smith, "Start/Pat: a parallel-programming toolkit, *IEEE Software* Vol. 6, pp. 29–38, July 1989.

[2] R. Aydt, *The Pablo Self-Defining Data Format,* 1993, University of Illinois, Department of Computer Science.

[3] D. Bergmark, "Update on tools for parallel programming at the CNSF," Cornell University, Ithaca, NY, Tech. Rep., 1994.

[4] IBM, *AIX Parallel Environment Parallel Programming Operation and Use.* Austin, TX: IBM, June 1994.

[5] J. Kohn, and W. Williams, "ATExpert," *J. Parallel Distrib. Comput.*, vol. 14, May 1993.

[6] P. Messina, and T. Sterling, Eds., "*System software and tools for high performance computing environments,*" SIAM (Philadelphia), 1993, A report on the findings of the Pasadena Workshop April 14–16, 1992.

[7] C. Pancake, "Where are we headed?" *Commun. ACM* vol. 34, pp. 53–64,Nov. 1991.

[8] C. Pancake, and C. Cook, "What users need in parallel tool support: Survey results and analysis," in *Proceedings of Scalable High-Performance Computing Conference* (Knoxville, TN) (May 23–25 1994), IEEE Computer Society, pp. 40–47. http:/www.cs.orst.edu/ pancake/surveys/surveys.html.

[9] C. M. Pancake, P. S. Utter, D. Bergmark, and D. Gannon, "Supercomputing '90 BOF session on standardizing parallel trace for mats," Cornell Theory Center, Tech. Rep. TC91TR53, March 1991.

[10] A. P. Research, *FORGE Explorer User's Guide.* 550 Main Street, Suite I, Placerville, CA 95667, Jan. 1995.

[11] H. Zima, and B. Chapman, *Supercompilers for Parallel and Vector Computers.* New York: ACM Press, 1990.