

Productivity in HPC

David J. Kuck
Parallel and Distributed Solutions Division
Intel Corporation
Dec. 2003

Abstract

HPC has been a popular acronym for decades, and has been applied to many types of architectures, software and applications. The “P” has recently been overloaded to mean both performance and productivity. We present a survey of today’s performance and productivity situation relative to the constituent HPC HW & SW components, and provide some analysis of current controversies and open issues. Since “HPC” will continue to be applied to whatever is happening at the leading edge of computer architecture, system & development software, and algorithms & applications, there is no hope or need to define or clean up terminology (this paper uses HPC to denote hiperc and hiproc, with context determining meaning). Instead, it is important to clarify the whys and wherefores of the state of the art, in order to focus on new work that will maximize future benefits.

This paper gives a broad discussion of HPC productivity in terms of effective architectures, runtime system software, and applications development tools. There are costs and tradeoffs associated with each of these, and in fact multiple marketplaces consume these products. The range of demands placed on HPC by owners and users of systems ranging from public research laboratories to private scientific and engineering companies, enrich the topic with many competing technologies and approaches. Rather than expecting to eliminate each other in the short run, these HPC competitors should be learning from one another in order to stay in the race. It seems clear that the dynamics between “commodity” and “custom” building blocks will remain at the center of HPC debates for some time, and indeed these competing forces form the engine of improvement for overall HPC cost/effectiveness.

1. Introduction

For the high performance computing book that I was writing about 10 years ago [Kuck96] to summarize our Cedar system experiences at CSRD, I designed a dust jacket cover that was a maze, entered at “demand” and exited at “productivity.” To solve the maze one had to successfully pass “systems,” “policy,” “performance” and “parallelism.” In the book I discussed the performance of parallel systems in great detail. As a metaphor, the maze captures the essence of the problems that must be solved in each architectural generation to produce high-productivity high-performance computer systems. In this paper, I will focus on productivity from each of these viewpoints, using the problems of today as drivers.

In the early 1990s, the HPC community was in a state of shock over the fact that ECL circuits had run their course and were being replaced by CMOS RISC processors. Clusters of non-vector processors were appearing and people were acting on the old observation that even if it was slower, one might be better off owning an inexpensive clustered system oneself, than sharing a very expensive vector system with hundreds of other users. Some felt that the post-ECL-Cray era would be catastrophic for HPC. In the past decade, however, the DOE ASCI program has devoted itself to procuring a series of HPC computers for nuclear stockpile stewardship by its three major labs, the Japanese have introduced the earth simulator system, and many HPC sites have installed thousands-of-processor clustered systems.

At the low end of the computer architecture hierarchy people began thinking in the 1980s about clustering the lowest priced processors, which were relatively very weak but “almost free;” in the middle there were a variety of exotically interconnected NUMA RISC-based systems; and multi-vector systems dominated at the high end. Programming the vector systems had a decade earlier been very difficult, beginning in the mid-1970s with total reliance on assembly language. But in the 1980s, a combination of more powerful automatically-vectorizing compilers and the tendency

to rewrite old Fortran programs in a vector style, made the vector systems quite effective for many technical computing applications. Improving the combination of algorithms and SW development tools had proven the productivity of vector systems.

Since the early 1990s, clusters of low-end processors have become increasingly popular. COTS (commodity off the shelf) clusters, driven in the mid-90s by Beowulf clusters, have grown in popularity and performance as microprocessor-based clusters have been enhanced by architectural ideas that were pioneered by earlier custom systems. Programming distributed systems remains difficult, however. Message passing is the universal programming model and MPI has evolved as the dominant library, but enormous, assembly-language style effort is required to develop MPI programs.

That HPC has made a tremendous impact and that computational science and engineering have entered the mainstream, should not be overlooked at this point. It is only necessary to open an issue of any engineering or scientific journal, not just those devoted specifically to computational matters, to observe that HPC has really become the third branch of science and engineering. Decades of HPC R&D are paying excellent returns, but many important open problems remain.

The next three subsections sketch HPC background areas that are discussed in more detail throughout this paper. In Section 1.1, productivity is defined intuitively and discussed in terms of costs and expectations by the HPC community; Section 6 summarizes the paper's productivity discussion with a proposal of four metrics for measuring productivity in concrete terms. Section 1.2 outlines aspects of the conflicting demands in the HPC community, which make progress uneven, as limited resources are applied that improve the field, nonuniformly over time. Section 1.3 provides a high-level view of how market forces and technical progress interact, to satisfy one or another of the many productivity demands in the HPC community.

1.1 Productivity

What do we mean by productivity in HPC? Some people have begun *defining* HPC to mean "high productivity computing," a nice, but somewhat ambiguous, turn of phrase. We should only accept one definition for productivity enhancement in computing: ***increasing productivity means that users are able to do better work, more quickly and easily than before.*** This requires several contributing technical points relative to parallel computing, the first three from the computing industry, and the fourth from users:

1. better parallel architectures for efficient individual program execution,
2. better run-time hardware and software support for system-level control, performance, & reliability,
3. easier parallel software engineering methods for developing innovative applications programs, and
4. a series of larger or more complex problems to solve.

The simpler productivity enhancement definition of "getting solutions faster" can be satisfied simply by providing higher performance systems, but overall productivity enhancement using computers should incorporate the notions of easier development for, and use of, computers, as well as solving "harder problems" or "doing better work." Only with all four of the above points in place can we truly suggest that parallel computing has achieved high productivity status.

In this paper we discuss points 1, 2 and 3 in detail; all improve over time, with successes and remaining problems. Point 4 is easily achievable in high-end computing as the world moves forward and people develop new parallel mathematical models and algorithms to solve ever-harder problems. This is a natural phenomenon of the HPC world as new demands are made on existing computational science and engineering systems, but it is not so in general. For example, an ultimate word processing system could be built in principle, such that no further computing capability would be needed. Since people only type and think at a fixed rate, correctly parsing, fixing syntax and spelling, formatting, etc. could be done ideally (up to inherent ambiguities) with

an ultimate system. However, we are still far from that goal despite well-funded companies having been in the business for years, so corporate motivation plus money are requisites to success.

To elaborate point 4, note that scalability is the issue here as it is in points 1 through 3. In 4, an issue is that some users are mainly interested in solving a fixed-size problem faster. This *strong scalability* is technically more challenging than the *weak scalability* implied by solving larger problems. For example, consider the scalability of a given <parallel program, problem size> pair. On a suitable architecture, scalability initially depends on using systems with more and more processors; when scaling fails [Kuck96], a better architecture may sustain the scalability increase. This process continues at a given time/technology state until the sequence of “better systems” runs out. At that point scalability enhancement can be sustained by changes to program structure or to the basic algorithms used, but eventually the best achievable algorithm/program/compiler combination for each current system reaches its limit, thus exhausting strong scalability options.

Most users move immediately in the weak scalability direction by increasing problem size over time, as dictated by new science/engineering questions. Of course, algorithm, program, and compiler improvements appear continuously, so real world progress moves simultaneously along both strong and weak scalability dimensions. In point 4, the focus is on weak scalability, independently of the strong scalability drivers 1-3.

The *costs of productivity* can be expressed in terms of points 1, 2 and 3 as:

TCP = Total Cost of Purchase,

TCOM = Total Cost of Operation and Maintenance, and

TCAD = Total Cost of Applications Development,

where these terms sum to Total Cost of Ownership:

TCO = TCP+TCOM+TCAD.

Table 1 shows relations between these productivity dimensions (or their related costs) and the viewpoints of system purchasers/owners vs. system users. The last column lists some providers of these goods and services as well as tradeoffs encountered in making decisions. A clear set of tradeoffs surrounds open-source/commodity vs. proprietary/custom-provided software and hardware. Some of these topics will be discussed in more detail throughout the paper, and Table 1 serves as a roadmap to overall issues (space limits prevent a detailed discussion here).

Productivity Dimension and Related Costs	System Owner's View of Expectations	User's View of Expectations	Providers, Tradeoffs
System: Total Cost of Purchase (TCP)	3 year budget, technology cycle	Needs met, easy access	OEM/SI – Custom Differentiation vs. COTS
	Good system architecture/HW @ low cost		
TC Operation & Maintenance (TCOM)	Effective RT SW, High throughput Autonomous, Adaptive,	No Interruptions or Errors, Just Amount Needed (JAN)	ISV/OEM SW – Open Source + Differentiation

TC Application Development (TCAD)	Low cost to users, fullest use of system by maximum community	Low \$/solution with wide apps coverage	ISV/OEM Parallel SE tools – Open Source + Differentiation
Effective Apps Enabled on System		Hands-free usage model Most industry uses ISV, govt. acad. still develop	OEM ISV-enablement & services or self-developed apps

Table 1. Productivity and Cost Tradeoffs

While Table1 is only a sample of the tradeoffs at work in moving HPC forward, it captures enough about the field to show why progress is nonuniform over time. It also motivates how strenuous debates within the HPC community can arise and be sustained over periods of years, as breakthroughs allow one or another aspect of the field to race ahead.

1.2 Current Demands within the HPC Community

Today, at the high end, Intel-based OEMs sell multi-thousand processor systems regularly, using nodes with either 2- or 4-processor SMPs, so the number of nodes is large. This can put major demands on interconnect fabric and message-passing programming. In the commercial markets, large high-throughput clusters are common with many transactions sharing common databases and usually making low inter-processor communication demands. On the technical HPC side, however, inter-processor communication demands can be severe. At the ASCI labs, where clustered systems run up to 16K processors, the norm in processor demand for single runs is large (1K processors for 40% of the jobs in a recent unpublished survey at Los Alamos National Lab).

Thus, for computer companies, technical computing raises one set of demands, but commercial database systems raise another set, and in a much larger market than technical computing. This leads to business-policy issues within manufacturers about where to focus attention in order to favor the technical computing market, while not ignoring the commercial clustering needs. For example, IBM has introduced BlueGene and its OnDemand Supercomputing program, and Intel has recently established an Advanced Computing Program to help address these issues

While there is demand for a wide range for clustered systems, the market for large SMP and NUMA systems continues and many manufacturers have 16- and 32-way systems as well as larger ones. The delivered cost per processor is much higher than for clustered systems, and programming tools are more advanced. But in the overall sense of TCO, debates still rage on whether or not the advantages of larger SMP systems justify their higher initial costs. We will say more below about these issues, including the point that influential public HPC sites often seek maximum *peak* systems rather than maximum *productivity* systems, as their motivations and reward systems are quite different from commercial technical or business support teams and users; see Table 1.

1.3 The Innovation Cycle

The concepts of open, standardized software and hardware are often at odds with commercial interests in computing, and many arguments have been advanced about what the best model for SW development is. For HW development, the *public* arguments are less intense, because (unlike SW) most HW development projects require enormous financial investment, but there has been a perpetual debate about custom vs. commercial (now COTS) hardware, that goes back decades to well-publicized systems like IBM's STRETCH HARVEST and continues from public QCD systems to modern classified systems.

When whole computer systems, together with productive applications software, are considered, the following model captures much of the engine of change for computer systems over time. Innovation arises from privately or publicly supported research, it is commercialized in differentiated products by ISVs, OEMs or others, successful ideas lead to standards for human and technology interfaces, and standardized products attract the attention of innovators. This cycle, as illustrated in Fig. 1, allows for great diversity in each step, but basically drives the march toward price reduction and high volume (i.e. COTS). The tradeoffs between standardization (for user acceptance) and differentiation (for competitive advantage) are pervasive, as was pointed out in the tradeoffs column of Table 1.

The arrows of Fig. 1 are meant to be read as “follows from, over time” relations, not as logical implications. Thus, when standards have been established, and especially as sales flatten for market-saturated COTS products, if innovation is possible it arises to generate improved products. These provide differentiation in the marketplace for producers, but generally inhibit market penetration, until standardization paves the way for the least adventurous system owners to adopt what the early adopters may already be tiring of. Thus, in the real world, various market segments pass through the stages of Fig. 1 at different times, adding conflict among demands in the field.

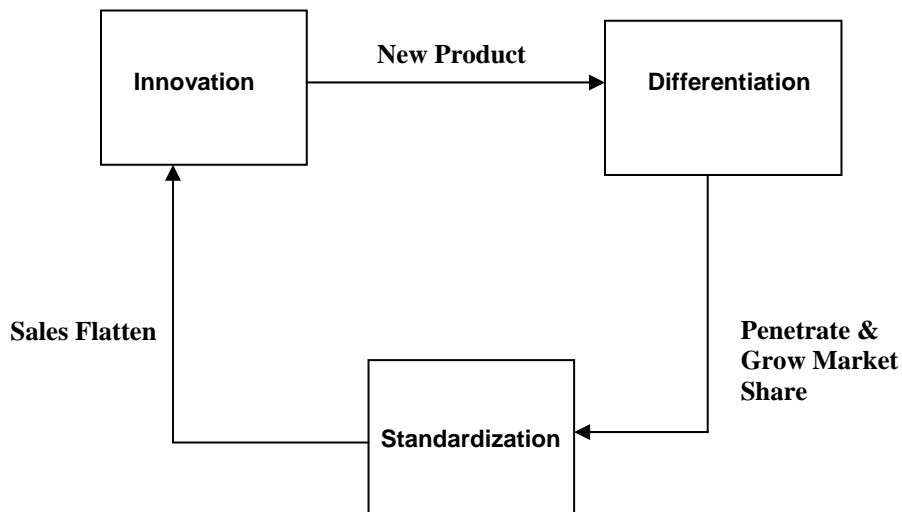


Fig. 1 The Innovation Cycle

2. Productivity in Parallel Application Development

End users care very little about this topic; they are interested simply in getting problems solved using fast, effective, reliable and robust software. Most ISVs (independent software vendors) are similarly oriented toward their users, and expect OEMs and software ISVs to provide tools that help them produce high quality application software for the industry.

2.1 Parallel Software Engineering

Three issues dominate developer productivity in producing parallel applications:

- Programming model and language
- Correctness
- Performance

First, it must be easy to write and update parallel programs, second one must get the program to run, and finally – the *raison d’être* for HPC parallel programming – the program must perform well. Unfortunately, each of these must be considered for SMP as well as DMP systems, and none of the topics is easy for most real application developers.

Who are these developers? In most areas of computing today, they are employed by ISVs who make a business of providing software that solves particular problems for a class of end users. Some large organizations still develop their own internal applications; this is the case at most government labs and in diminishing numbers of large industrial firms it remains true. Furthermore, most ISV employees are focused on the domain specific aspects of their software, e.g. mechanical CAE ISVs hire mostly mechanical engineers, not computer scientists. Thus, application developers want software engineering solutions that work out of the box.

For the past decade, those developers who are focused on workstation and SMP solutions have threaded their programs, while those interested in supercomputing solutions have switched from vector to message passed programming. In both cases, parallel programming leads to a new class of difficult bugs beyond those found in sequential programming: after a program works correctly, high performance is often very difficult to obtain. On 1K and higher processors, single digit efficiencies dominate the field today for deployed applications software, with rare cases getting above 25%, and many applications are regarded as not parallelizable beyond, say 4-8 processors.

2.1.1 Programming model and language

Automatic parallelization of sequential language programs has been pursued for several decades. For vector systems it became quite effective, for SMP systems it is used in some cases (often supplemented with program directives), but it is not generally used for distributed memory programming. Research in the area has diminished due to the difficulties encountered with analysis speed, aliasing, global analysis of millions of lines in a single application, and the conflicting desires of ISVs to exert control over where restructuring is applied. As programming languages and styles evolve with architectures, and as higher productivity is demanded, automatic parallelization techniques are likely to be reinvigorated for various locally analyzable cases in future systems that use high degrees of chip-level parallelism.

Many explicit and semiautomatic parallel programming methods exist (see Section 2.2). Explicit programming methods are the most popular because, despite lower productivity than automatic methods might provide, developers know exactly what they are doing – or they think they know, and that is what decides matters. We will outline some details of these below. It is important to point out that ease of parallelism expression in high-level language terms has a rough correlation with the ability to provide power tools for correctness checking and performance analysis. For example, for OpenMP (which is usually an easy model to use), current tools can very accurately locate threading correctness problems, making it easy for developers to correct them and enhancing developer productivity. Similarly, OpenMP performance profiling can be done more easily than for performance profiling of a program making manual calls to a threads library.

At the level of distributed programming, Intel is researching cluster OpenMP to allow the use of OpenMP-style correctness and performance tools for distributed programs, based on a new directive beyond the OpenMP standard [OpenMP]. This is based on a much-modified version of the Treadmarks Distributed Virtual Shared Memory system [KCDZ94]. This style of programming is easier than MPI programming in that it allows one to add directives incrementally to a serial program and even have the tools generate a list of needed fixes to get a correct program. Furthermore, it can dynamically accommodate programs in which static domain decomposition is not possible, potentially providing better performance for some applications. On the other hand, it will only succeed for programs that have appropriate compute/communicate time ratios. If deep algorithm change is necessary, then the work required may be equivalent to rewriting an application for MPI. At this time there are many open questions about how powerful tool availability will expedite use of a cluster OpenMP programming model.

2.1.2 Program Correctness

Debugging sequential programs is a difficult task that in practice still relies on intuition and break-point debuggers. Once basic debugging is over, Quality Assurance testing to locate bugs, ensure

path coverage, check for regressions, etc. is the normal practice for maintaining correctness using a standard test suite.

Assume that we have a correct sequential program (in the above sense), and use parallelism to develop an equivalent higher-performance version. This means that we may be able to test the correctness of the parallel version against the normal Quality Assurance procedure used for the sequential program. For most ISVs, sequential correctness implies a standard set of real-world problems that is validated each time a new version of the application is released. Using this approach, we can test the various parallel programming models for correctness with a range of validities. That is, while we cannot find every error, we can find more and more of the errors, as we increase the time and space used in the analysis process. The process depends on generating comparable sequential and parallel memory traces from a program for analysis. In each case, when errors have been found they must be fixed, and in the worst cases, break-point debuggers still play a role in tracking down details. Notable threads aware debuggers include gdb, the cross-platform GUI-based TotalView [Etnu04], and offerings from several OEMs.

O Threading

Threaded programs can be developed by directly calling a threads library (usually Posix or Winthreads), by using language extensions embedded in C++ or Fortran (usually OpenMP), or by using a language that implicitly supports threads (e.g. Java or C#). Standard threads libraries have been in use longer than OpenMP, Java, or C# and so are found in many deployed applications today. However, since calls can be used in arbitrary ways to spawn new threads, for a given hand threaded program, the intended memory semantics of an inferred unthreaded sequential program may be difficult to analyze. For OpenMP, the parallelism is so structured that if the directives are ignored, the original sequential program remains, and parallel correctness can be inferred very accurately relative to the sequential program's execution. The Java and C# threading models both allow unstructured threading, so only weaker validity testing can be done by comparison to parallel execution.

For OpenMP, KAI introduced Assure in 1997, and the product has been used on many ISV codes with excellent results; this was followed by an effective KAI Java product. Compaq introduced Visual Threads and Quest has Threadalizer. In 2003 Intel introduced Thread Checker [PeSh03] which has the power of Assure for OpenMP, but also handles manually threaded programs, as well as combined manual threads calls and OpenMP, and handles programs containing source and binary code (e.g. binary DLLs).

These tools work by instrumenting a given program, running it against given data sets, and analyzing the resulting states of memory. To approximate complete program coverage, multiple data sets, as in an ISV's QA suite, must be run. Thus the tools can be used incrementally throughout the development process and as a QA tool at product shipment time.

O Message passing

This is a much more difficult model to analyze than the threaded case. An arbitrary MPI program may not have much resemblance to a sequential program from which it was derived, because the requisite domain decomposition causes major rewriting of the program, and indeed there may not be a sequential program. Currently, much weaker forms of correctness tools are in research states for MPI programs, a leading example is [VeSu00]. As mentioned above, however, for the Cluster OpenMP programming model it is possible to develop very accurate correctness tools.

2.1.3 Performance

Once a program is defined to be acceptably correct (e.g. shippable by an ISV), performance tuning is required to enhance end-user productivity. Profilers are the basic tools used for sequential performance analysis. For parallel programs, profiling can take several forms. First consider the case of SMP programming using threads. If a structured parallel run-time library is used (e.g. the OpenMP directives), instrumenting the library calls can provide a profile of the

parallel program structure (this was provided in GuideView for OpenMP by KAI, where views by thread, by program segment, etc. were provided).

If a program is hand threaded (Posix or Winthreads calls), the structure of the parallelism, in general, is impossible to derive (as there may be nothing regular about the call graph). The obvious starting point for performance analysis is the critical (longest time) path through the program. This is not difficult to find, but is likely to contain so many threads related calls (fork, join, lock, etc.) that merely displaying a critical path is of little value. The current Intel Thread Profiler [] shows the user performance views based on a critical path analysis, to promote insight into performance improvement issues.

For message passed programs, the same issues exist, although for high performance there must usually be more parallelism, and at a higher level. The fact that there may be thousands of parallel processes introduces a scalability problem in storing or analyzing the data, as well as displaying it. A number of tracing systems have been developed that allow users to view time series across processors and attempt to discover performance hot spots beginning with [Couc89] through current products in wide use [BHNW01]. A combined OpenMP and MPI cluster tool is described in [HKNP01].

2.2 Commentary on success and innovation in tools

Besides the difficulty of solving technical problems in parallel software engineering, there are many difficulties in getting innovations widely adopted, because the ISVs for whom they are intended are focused on their applications, not software engineering and tools. Even superior ideas need to be sold, and sometimes lose out in the marketplace. The history of natural languages shows that in the long run technical advantages tend to be adopted (e.g. written and spoken vs. spoken only, alphabets vs. characters only), and that languages follow military and economic power (Greek, Latin, English) more than tradition or familiarity. An example in computing is the defeat of proprietary operating systems by Unix in the 1980s based on price and openness, but it promoted the spread of C, which offered disadvantages with its advantages and in some ways reduced productivity (weak arrays, memory leaks).

Many parallel language ideas have been proposed and rejected by disuse. Improving on currently accepted methods should be a high priority for the community, but it is very difficult to make progress across the board. For example, OpenMP is a major productivity improvement over hand threading, but offers little relative to asynchronous threading. MPI is portable and allows user control, but requires much expert work to use. HPF was offered as a high-level language alternative for Fortran, but was too weak for wide acceptance. UPC [CDCY99] and Coarray Fortran [Numr99] offer some useful features, but have had low levels of adoption by the community. Such languages provide a uniform address space view of DMP and insert communication automatically. The *shmem* library for the put/get/synch model has proven very effective for applications based on large blocks of data. The active object programming model (e.g. Corba) is another domain-specific model that maps easily to DMP systems.

For debugging and performance, simple procedures are often preferred to more powerful innovative tools that are unusual to use. Users initially found KAI's Assure amazingly powerful, but often asked for consulting help to use it as they were unfamiliar with its usage details. While users do adopt new procedures that offer sufficient benefits, the SW development process changes slowly (following a long learning/comfort curve) so innovative methods require much training, assimilation, and confidence building.

The enhancement of parallel programming productivity is moving ahead through the introduction of innovative tools, but the tools need time to mature and be widely adopted. This can be aided by open standards that allow experimental add-ons. Cooperation among multiple companies in developing and using tools helps their adoption just as military and economic events have historically driven the migration of natural languages.

But the adoption process is difficult and slow, and success usually does not occur on the first attempt. For example, unification of SMP directives spanned 2 decades: by the mid-1980s industrial directives were becoming diverse, the late-1980s attempt via ANSI X3H5 to unify directives was followed by industrial backsliding in the early 90s, leading to universal adoption of OpenMP in the late 90s. In this case the pain of the ANSI process defeated the energy of the industry-wide X3H5 committee, but the market pull insured the success of OpenMP.

Message passing followed a similar course through various implementations, until a self-authorized, industrial/academic, standards group defined MPI and popularized it via open implementations and books. Roughly speaking, success seems to require the pain of many users as reflected through computer company marketing departments, and seems not to need the endorsement of formal standards bodies.

3. Productivity in Parallel Execution Support

This is a much more wide-ranging topic than the subject of Section 2, so we will discuss it mostly from a high level perspective.

3.1 Open vs. proprietary software choices

The open-source movement as embodied in Linux-based Beowulf clusters and follow-on activities has led to a host of choices for setting up and operating clusters of COTS hardware. There is a dichotomy here in that for academics and government labs, which typically have a lot of help available, challenging SW projects are eagerly accepted. On the other hand, in a for-profit enterprise, there is little tolerance for a new system that is not productive on day one. This has led to many mismatches in expectations among suppliers and potential users in the clustering community.

Cluster Platform Software Components		
Run time Software	Cluster System Software	Linux, OS installation, Configuration management, Clustering control, Management package
	Job Control	Batch queue, Schedulers, Cluster monitoring, Scalable Unix tools
	Middleware	Communication Libraries (MPI, DVSM, etc), Fabric interface SW, Management support
Development SW	Software Development Tools	Compilers, Math libraries, Debuggers, Correctness tools, Performance tuning tools

Fig. 2 The Hierarchy of Cluster Software

The positive view of this pigeon-holing is that various computer manufacturers (OEMs) and system integrators (SIs) have distinct approaches and distinct constituencies for product sales. Those OEMs who can provide a full range of software and services often have custom-written software that supports parallel execution, while those OEMs and SIs who specialize in the lowest priced systems will gladly encourage the use of free-ware. The results allow a range of choices for the range of customers mentioned above, but there are also conflicts between open and proprietary approaches that are offered in a given technology area, as described next.

3.2 Software Development Styles

Within full-range OEMs, open source projects create a mixed reaction. From the management point of view, such a project forgoes intellectual property claims, relieves the need to allocate as much internal money for development (as the community is sharing the work), and may focus management priorities elsewhere. In general, the drive to innovation is weakened in a community effort, but for re-implementations of software that has wide circulation, that is not a problem (e.g. Linux in a world that has had Unix for decades). Improvements (if not breakthrough innovations) are easy to achieve as there is a large base of users, and years of experience to guide the open source community. Thus, with Beowulf or Oscar consumers get low cost, widely familiar software from the open source community when projects succeed, although many projects do not succeed; note that SourceForge hosts more than 65K open source projects in total, so only a few can possibly reach the support levels necessary for great success.

If high risks, innovative directions, and initially low volumes dominate, focused academic or proprietary R&D efforts are more likely to succeed. For example, IBM is pursuing its wide-ranging Autonomic Computing [KeCh03] strategic vision and products, while HP has announced an Adaptive Enterprise [IDC03] software initiative. While these visions are targeted at the whole IT world, they include very specific research topics that impact high performance computing, and vice versa. In other words, the high productivity computing activities within the high performance and advanced computing research communities will have an opportunity for influence that goes well beyond their traditional grasp, as various elements of clustered system software converge across technical and non-technical computing.

3.3 Software specifics

System managers focus on three basic issues; these are easy, effective methods to:

1. Set up new clustered systems above the raw HW level,
2. Control running systems, and
3. Deal with systems running incorrectly operating HW and SW.

This paper does not go beyond the philosophical aspects of these topics because there are so many potential subjects of interest to various user communities. Figure 2 describes a four level hierarchy of software that spans Sections 2 and 3 of this paper, with this section covering the top three levels. Broad topics that need much attention from the productivity point of view include:

- o Initial configuration of clusters
- o System monitoring and management
- o Schedulers for throughput and turnaround time
- o Fault tolerance and recovery
- o Reconfigurability

In HPC, an open source basis for this work seems desirable to many system owners, but there has been much debate about whether or not an add-on tool strategy would be cost effective, given widely used basis software. One such basis is OSCAR, the first project by the Open Cluster Group, and a widely downloaded software basis (hosted at [SourceForge](#)) for open cluster construction.

As stated in their website [OCG]: "OSCAR version 2.3.1 is a snapshot of the best-known methods for building, programming, and using clusters. It consists of a fully integrated and easy to install software bundle designed for high performance cluster computing. Everything needed to install, build, maintain, and use a modest sized Linux cluster is included in the suite, making it unnecessary to download or even install any individual software packages on your cluster."

While OSCAR, NPACI Rocks, and other open source software, as well as ISV proprietary solutions e.g. ScaliManage, etc., and research initiatives such as Score (Japan), SciDAC (U.S. DoE), etc. provide the basics as well as advancing the field, open questions in this arena will continue indefinitely, as user demands vary and evolve. OEMs will have the continuing goal of

differentiating their products, and the areas for differentiation will continue to change as common software standards advance and particular markets are further developed by the OEMs.

Innovation, from wherever it arises needs wide circulation and use to mature, so basing proprietary advances on a widely used open source base seems the best strategy for innovative full range OEMs. It can provide their users with both a familiar open source basis and some innovative, productivity enhancing proprietary features.

3.4 Execution Support Conclusions

As the use of an open source basis gets wider circulation, innovations and differentiations can be built on top of the basis. Standardization draws innovations into an open source basis and feeds a new cycle of innovation. Thus, referring to Fig. 1, innovations and differentiated proprietary products tend to drive a new cycle toward standardization and wide use.

Full range OEMs (or dominant ISVs) often fret about the erosion of their markets for established products (i.e. defend their cash cows), when low-overhead system integrators or limited-range OEMs undercut their prices. However, until HPC productivity has advanced far beyond its current level, those OEMs who can demonstrate intellectual leadership will have a wide range of opportunities for innovation, differentiation, and continuing their drive to lead market segments. Subsequent sections explore various aspects of these points in more detail. In the end, this benefits productivity in each market segment, because full-range OEMs are able to invest sufficient resources to work closely and effectively to solve real-world problems.

4. Productivity in parallel architectures for cost/efficient program execution

A wide variety of architectures remain in common use today, as was the case when [Kuck96] was written, but the range is decreasing. Fewer new vector systems are being sold, fewer custom processor types are appearing, and there is a trend toward COTS systems. Large, full-range OEMs as well as small system integrators have joined the academic/lab pioneers in combining separately manufactured building blocks to construct clustered systems; separation points in the architecture hierarchy come at the levels of SMP nodes, interconnect fabrics, and I/O systems. The cost-performance benefits of these systems are now widely acknowledged across technical and commercial computing. The latter point is illustrated with succinct and aggressive optimism in the following Oracle advertisement in the Dec. 3, 2003 *Financial Times*, quoted in its entirety: "The Oracle grid: Turn 64 small servers into a giant mainframe. It's cheap ... and it never breaks."

4.1 SMP nodes

Shared memory processing arose decades ago when 2 or 4 large processors were hooked to a single memory for economies of scale. In the near future, chip multiprocessor (CMP) systems will incorporate more than 4 processors on a single die. The modern motivation is a combination of the mere fact that the real estate exists, and of the need to surmount the memory and power walls that have frustrated uniprocessor chip designers; as processors become more complex and clock speeds increase, power densities have grown to unsustainable levels. Two large wins are achieved by partitioning a given amount of chip real estate into several processors that are not the fastest possible (and do not have the largest possible cache): reducing power density and providing large shared caches.

From the productivity point of view, larger SMP systems provide several advantages over DMP systems. Their programmability is superior to DMP systems, and their manufacturability and engineering design is becoming superior to the fastest uniprocessors or DMP systems as they become the focus of single die designs. These designs should be able to incorporate good cache coherence, fault tolerance, and scalability to multichip SMPs.

The scalability of SMP systems will always have a limit, so hierarchically clustered SMP nodes forming scalable DMP systems will continue to be the norm. In this case, the central architectural

design problems concern providing good bandwidth across the SMP chip boundary and good interconnection fabric among the SMP nodes.

4.2 Interconnect fabric

Choices here range from Ethernet to much higher bandwidth and more expensive options (e.g. Myrinet, Quadrics, or Infiniband). Depending on the bandwidth and latency requirements of the applications to be run on a system, great cost savings or performance improvement result from correct choices. This has benefits to manufacturers in configuring system classes for different market needs, but it also leads to unfortunate anomalies. If an ISV chooses to port and develop an application on a given fabric, it may also lead to choosing SW associated with that fabric, e.g. a given message-passing library may be highly tuned only on particular fabrics. This in turn can force large enterprises to operate several different clustered systems, each based on a distinct switching fabric.

The software stack that supports applications running on COTS systems has grown very deep in the past decade, with standards emerging for many different functions. To enhance performance, hardware is often designed to replace software layers, but unless it is easily programmable, productivity is eventually cramped.

Current trends are to RDMA (remote direct memory access) approaches that free computational processors from almost all involvement in the fabric's operation, and partitioning of busses to allow processing and communication to proceed independently. Infiniband is an industry-standard RDMA fabric that allows multipurpose communication (node communication, disk I/O, network connection) to operate in a highly scalable manner. Layering software stacks on such a fabric, which are sufficiently abstracted so that multiple, hardware switching-fabrics can be supported just by replacing drivers, should enhance cost/performance and minimize the need to rewrite system or application software. This enhances programming productivity and frees the market from being locked into proprietary solutions at too low a level.

4.3 System level issues over time

The speedup of a given program over calendar time is a metric of basic interest that is poorly understood. If speedup equals processor count (p) X efficiency (E_p), this can be easily reduced to two well-known principles: Moore's Law drives p , and Amdahl's Law drives E_p . Moore's observation has been robust over several decades, but has inevitability problems, while Amdahl's is an unavoidable inequality that has many workarounds. The latter include using better tools to develop more-parallel applications, reducing run time delays by better support software (and this extends to throughput via better scheduling, partitioning, fault recovery), better cache coherence and management to reduce data access times, better interconnect fabric to reduce communication delays, as well as increasing problem sizes for a fixed p (weak scaling).

The basic question is, what function f , can be observed to constrain delivered performance:

$$Sp(t) = p(t) E_p(t) < f(t),$$

where t represents calendar time spanning years of computing the same class of applications? Further, how can we improve $f(t)$ through hardware and software innovation? It is widely understood that massively parallel systems are often employed even when they deliver less than 10% efficiency on real applications. If this efficiency falls faster than processor counts increase over time, then we have a serious problem. Indeed, if we are not eventually able to improve SW tools and system architecture sufficiently to drive $E(t)$ and $f(t)$ at the traditional Moore's law rate, then when the growth of $p(t)$ hits serious impediments, the benefits of parallelism will disappear.

4.4 Total Cost of Ownership (TCO)

Recall from Section 1.1 that we express:

$$TCO = TCP + TCOM + TCAD.$$

One of the anomalies of current HPC usage is that there is a drive for public organizations to be “leaders” in computing and Top-N lists are kept based on metrics not much more discriminating than peak speed (e.g. dense linear system solvers). Thus, budgets in public research Labs tend to be spent mostly on system purchase (TCP) in order to place high on Top-N lists. In contrast, real applications do not scale nearly as well as Linpack. So dSp/dp is usually minimal when thousands of processors are applied to a single complete real application. Thus, by cutting the number of processors acquired by a factor of 4 to 8, say, real applications might achieve cost/performance gains of 2 to 5X. This would make the COTS advantage much clearer, over traditional vector machines, but ruin managers’ ability to announce Top N rankings.

4.4.1 Total Cost of Applications Development (TCAD)

Another HPC anomaly remains in the difficulty of programming high quality parallel applications. Programmer productivity directly affects the overall HPC productivity equation through weaknesses in parallel software engineering. Most people must engage in deep program restructuring and assembly language level work (for effective MPI programs), using tools that lack sufficient power. Some running applications (MPI and threaded) are said to be parallelized, as a cachet and because so much effort was applied to parallelization, even though they yield embarrassingly little performance gain as a result.

Three conceptual approaches exist for improving the situation:

- Automatic generation of parallel programs from sequential ones
- Incremental (automatic or manual) refinement of parallel programs
- Manual development of parallel programs from scratch

As pointed out earlier, real gains in productivity require new algorithms and applications development work. The gains occasionally are months-to-weeks levels of reduction in end-user time to problem solution. Such work requires expensive development cycles, and yet, even at HPC centers, most of the budget and focus remains on buying faster systems and not on the harder (than getting larger budgets) issues of effective programming and using the systems to genuinely enhance productivity. This is true at many government labs where ISV SW is little used, so projects move ahead based on labor-intensive, long-running internal applications development.

4.4.2 Total Cost of Operation and Maintenance (TCOM)

In a sense, these are naturally and logically the last problems to get attention. Hardware comes first, then applications, and finally fine tuning of the system and its operation. In fact, hardware vendors with motivation to maximize their HW revenues have little inherent motivation to assure that it is used efficiently. Of course, full-range OEMs with substantial service organizations are asked to help in this dimension, causing it to become a major topic for distinguishing themselves from other OEMs, ISVs, and the open source community.

There are complex relations in the marketplace among open source low cost SW, proprietary differentiated SW, and services for sale vs. systems management demands. These multiple approaches by different types of providers seem to imply that there will continue to be opportunities for many kinds of providers for a long time. Success can be achieved by small companies catering to very narrow bands of the market, and at the other extreme, full range OEMs are likely to be able to succeed by selling proprietary SW and services to productivity-oriented organizations who simply want computers to work for them (the ISV SAP, has succeeded elsewhere with this model).

4.5 Proprietary and Differentiated Systems

While low-cost COTS clusters that include standardized hardware and software are emerging, and provide many productivity-enhancing features, there is much unfinished business. The monitoring and management of these systems remains an area of open-ended issues needing attention, as they are used in environments with many distinct needs. Reliability and fault tolerance of hardware and software are in a similar state. Relative to such issues, there is

enormous opportunity for OEM and ISV differentiation with proprietary value-added solutions that enhance productivity.

In other words, after a decade of COTS development work in cluster HW and SW, these systems are now being widely adopted and are available from a wide variety of sources. Some system integrators provide the lowest cost systems – perfect for one class of users. Other systems are provided by full-range OEMs who add value with custom software and support. As cluster deployment spreads, this customization and differentiation will continue, providing revenue growth for OEMs and ISVs, while enhancing productivity in more and more market segments.

The debate between COTS and custom advocates has a transient aspect, as volume parts will not forever lack the features needed for high performance. One only needs to observe the changes over the past decade to anticipate that eventually, most key architectural features will trickle down to low-priced hardware systems. This does not mean that innovation in custom parts will ever end, but in terms of cost effectiveness at the system level, it is arguable whether or not a custom market will continue forever. From the beginning of computing it has been observed that custom logic can be designed that is far faster for any given application (cf. Section 1). It is *cost* that has usually prevented such designs from being fabricated. FPGA technology is a current approach that may eventually lead to cost-effective do-it-yourself custom designs.

5. Open Problems

The HPC field has advanced substantially in the past decade, but there are many remaining problems that must be solved, in order to lift HPC productivity to a point where TCO is reduced to much more reasonable levels (i.e. to the point where an HPC purchase becomes matter of fact, not requiring special funding as it has been for many years). As has been pointed out above, there are so many types of consumers that fruitful opportunities exist for many types of solutions.

Government and Industrial Policy

The number of problem solutions generated per month or year should be a key metric for HPC facilities, and this should include a focus on up-to-date applications. In some cases older algorithms and applications are run inefficiently on new systems, long after the introduction of the new systems, because of the difficulties of developing and tuning on new parallel platforms.

The use of large budgets to pay for new systems that place high on the Top N list vs. developing state of the art problem solutions is an issue at government labs and universities around the world. The former is a “no-brain” use of funds leading to instantaneous image enhancement, whereas hard work on applications development can take years and be a much harder path to gain public recognition. Government policies that allow or tend to encourage lab expenditures to buy into the Top N, without being accountable for improvements in end-user productivity, are a serious current problem.

The current epitome of this position is the quest for petaflops systems. The position of this paper is that a far better goal than petaflops systems would be truly productive teraflops systems, in particular ones that deliver on the High Productivity Computing Indicators of Section 6.1. Besides raising user productivity, HW and SW suppliers would benefit from higher value, higher margin sales, over time.

Industrial users, while not generally appearing on Top N lists (because they have no motivation to report such numbers), nevertheless have problems similar to government labs. In many cases, managers do not want to risk innovative approaches on the grounds that it is too difficult to get new software to produce results that match exactly the results of well-established computational methods. Thus, old software is sometimes sustained as long as possible, because risking the loss of current productivity in attempting to greatly improve productivity, is not judged reasonable in view of the known difficulties of harnessing parallelism.

5.1 Application Development Software Issues

5.1.1 Algorithmic Performance Contributions

At the most basic level, the number of algorithmic steps required to produce a problem solution determines the productivity of a computation. Over time, progress has been made on a number of fronts. Reductions in the number of sequential operations and overall memory requirements, as well as time reduction by parallel computations, including data communication time, have had major impacts. Examples of time step reductions include sequential and then parallel FFTs or parallel linear algebra algorithmic changes. New algorithmic approaches to old problems (often reducing data communication time) include spectral methods for PDEs, fast iterative linear system solvers with optimal preconditioners for specific applications areas, fast multipole algorithms, etc.

These and many other methods have yielded tremendous speed and productivity gains in particular applications areas. For example, a time reduction by serial algorithm improvement or parallelization from $O(N^2)$ to $O(N \log N)$, for problems of size $N=1000$, yields speed gain factors of 100, which are equivalent to 14 years of processor gain (using 10X per 7 years) or of employing 1000 parallel processors with a 10% efficiency.

It is difficult to capture the importance or effectiveness of specific algorithmic breakthroughs, as real applications usually employ combinations of methods, and whole codes are incrementally changed over time. Furthermore, standard libraries usually do not have the breadth or currency to capture all of the crucial algorithms used in real applications. However, a table of major applications areas vs. the key algorithms used as major time consumers in each area, was proposed by Ahmed Sameh [DKLS87], and is reproduced as Table 2.

Application Area	Lattice Gauge (QCD)	Quantum Chemistry	Weather Simulation	C F D	Geodetic Network Adjustment	Inverse Problems	Structural Mechanics	Electronic Device Simulation	Circuit Simulation
Sparse linear system solvers	X			X	X		X	X	X
Linear least squares algorithms					X	X			
Nonlinear algebraic system solvers				X			X	X	X
Sparse eigenvalue solvers	X	X					X		
FFTs			X	X		X			
Rapid elliptic solvers			X	X				X	
Multigrid schemes				X				X	
Stiff ODE solvers		X							X
Monte Carlo schemes	X							X	
Integral transforms		X							

Table 2 Applications areas and Algorithms

Such a table for each market segment, if kept up to date and combined with overall program profile information (see below), could capture the essence of algorithmic impact on applications, and track changes over time. Of course, as algorithm effectiveness has a strong architectural component, the effective *implementation* of innovative algorithms on each system is crucial.

Again, keys are progress on matching algorithms and HW for system-wide communication, including cache coherence and global interconnection network bandwidth/latency.

5.1.2 Programming Model and Language

These questions center on human usability and wide language acceptance, and are really a social as well as technical issue. Furthermore, this is intended to be a sampling of issues, not an exhaustive list.

SMP

How can we express less-well-structured programs in high level languages, as OpenMP does to avoid hand threading, say for GUI programming [SHPT00].

DMP & SMP

There are many MPI programs that deliver good performance, but because the time required to develop those programs is often measured in years, application development productivity levels are often very low. Can we rise above the assembly language-level of MPI (say by bulk synchronous programming [Vali90])? Can Coarray Fortran [Numr99], UPC [CDCY99] or more innovative ideas be adapted to effective use? Cluster OpenMP (Section 2.1.1) is another approach. A major difficulty beyond getting the right idea, is having a high performance, portable implementation, and this requires major investment – which tends to lead to a chicken and egg problem with many “great ideas.”

5.1.3 Correctness Tools

The correctness problem for SMP programs, relative to sequential programs is in theory a solvable problem as we basically only need to compare the serial and parallel memory states at each computational step. For message passing programs without serial counterparts, the problem is much harder.

SMP

How can the performance and space requirements of threaded program correctness checkers be enhanced?

DMP

Can we find effective ways of automatically checking message passed programs? As mentioned in Section 2.1.2, it is possible to do an effective job of this for the Cluster OpenMP programming model.

5.1.4 Performance Tools

Unlike the parallel correctness problem, the performance tuning process only ends if one accepts some definitions, as those presented in [Kuck96], to stop working on performance. Moreover, the problem is harder in principle to solve than the above. Furthermore, some people are experts in specific kinds of tuning, while most developers are application domain-specific experts in need of HW/SW help, e.g. what to do about low cache locality in SMP systems.

SMP and DMP

How do we build tools that help human performance experts more easily find the most important performance bugs to work on, i.e. those code segments where improvement will most materially improve overall performance? In other words, can we list out key points at which expert effort should best be spent to maximum the effect of manual performance tuning?

Can we build prescriptive tools for performance enhancement, that minimize human effort in making basic contributions to parallel performance improvement, even when the humans are not performance experts?

In each of the above architectural cases, companion questions can be asked about combining the SMP and DMP functionality for the two architectural models.

5.2 Hardware issues

Flexibility in architectural configuration for application areas holds great potential for enhancing the cost and productivity of HPC systems. If one is trying to design just one system cost/effective configuration to suit all cases, one faces the impossible problem of optimizing parameters to satisfy the constraints of all applications. But when we partition applications into processing-bound and I/O-bound, say, we immediately see how to enhance systems in two dimensions, thereby matching the constraints of each application area.

With clustered COTS systems that will include CMP (chip multiprocessor) nodes in the next generation, the dimensionality of the parameter space over which we may consider optimizing grows dramatically. This will allow “customization” of COTS systems to a wide range of application types, or more accurately to collections of applications with parameters that make sufficiently similar architectural-demands. The trick then, is to determine in advance what these parameters are, to be able to identify them in applications, and to enable system designers to deal with them independently in producing future HW systems.

For example, given a fixed amount of semiconductor real estate, in designing a CMP, we could have as parameters: number of processors vs. processor complexity, cache size vs. more arithmetic or more processor connectivity bandwidth, and so on. Since interconnection fabric is becoming a complex system itself, similar issues could be parameterized relating to path bandwidth, connectivity topology, conflict resolution, and other parameters capturing the types of demands that can be placed on networks for various workload types.

Across all levels of the system there are issues of fault tolerance and recovery, schedulability, and control of the complex system to ensure its productivity under varying workloads. These issues span HW and SW co-design, and will not be discussed here further.

5.3 Run-time Support Issues

The reliability and quality of service delivered to users are major points of importance for high productivity computing. It is important to quantify these in high level terms that could be compared across systems and over calendar time. For example, unscheduled down time (UDT) and average time to recover (ATTR) from breaks in productive solutions, and total down-time events (TDE), i.e. total number of crashes that break user runs, produce negative impact on user productivity. Thus, reporting on the quantity:

$$\text{User QoS} = \text{TDE} \times \text{ATTR}$$

over time, could provide a useful metric for capturing an image of lost user time. Inasmuch as these issues arise for current implementations of MPI, runtime schedulers, hardware faults, and more, such a metric would provide a summary overview for users and system managers, especially if attribution of causes were included.

Partial system failures with recovery via checkpoint rollbacks and dynamic job rescheduling could make quantification complex. On the other hand, such techniques are largely lacking in this area, currently. So as they become available, metrics could be refined to account for the number of processors available per job, as the number is adjusted over time.

5.4 Comprehensive Metrics

How much productive work does a system compute in the course of a year? How much improvement is there from year to year in productive work? These are challenging and central questions to the discussion of High Productivity Computing. There are no metrics currently in use that attempt to capture these notions.

Standard benchmarks have been difficult to introduce for HPC systems. The Perfect Club codes [BCKK89] showed instabilities from application to application and from system to system for a given application. They have been adapted to become the SPEC HPC codes [EiHa96], but large

applications are difficult to run, and there is much more variety among HPC users than there is among transaction processing users, say, making universal adoption difficult. While continuing to run industry-standard benchmarks provides useful performance information, perhaps there are other metrics that could be introduced, which would reflect whatever applications a given HPC center runs, but in addition would allow productivity comparisons across systems, centers, and over calendar time.

An important concept in parallel computing is the use of many processors over time to solve one problem. Consider a definition that helps capture this notion:

$$\begin{aligned}\text{Jumbo Job} &= \text{total operations produced by 1000 current microprocessors in 24 hours} \\ &= 10^5 \text{ sec/day} \times 10^9 \text{ operations/processor second} \times 10^3 \text{ processors} \\ &= 100 \text{ petaoperations}\end{aligned}$$

If we could agree on such a unit, and this includes defining an “operation” (something like a floating-point operation is implied by this definition) major computing centers could report on the number of jumbo jobs run per year, as an indication of their ability to serve large users. In conjunction with metrics about quality of service (Section 5.3), and the basics of what the jobs are doing (Section 5.1.1), this would give a high level overview of progress.

Another indicator of annual productivity gains would be a plot of solutions per year in each major applications area for each system. This should be a clear indication of *solutions* per year, not total runs per year, and is therefore subjective and somewhat difficult to define exactly. Further refinement of this would be to indicate whether gains from year to year came from system improvement (e.g. cluster I/O systems), the use of parallel software engineering tools, or better algorithms.

To track productivity improvement over time, profiles of performance could be shown, indicating for each major contribution to overall execution time & performance level, the most recent date of code update. This would indicate how long poorly performing segments of code were allowed to remain in the application.

6. Conclusions and Recommendations

The following four points are broad summaries of much of the discussion in this paper. Taken together they form a broad core-indicator set for HPC productivity growth over time. If a collection of major computing facilities were to report such results annually, the HPC community would be able to assess the state of its art and make solid decisions about where R&D support was most needed for the future. A repository of such measures would, over time, also show productivity progress and longer term trends.

The propensity of human nature to compete and to desire simple metrics argues for simplicity, but the complexity of productivity issues in HPC demands that we develop and use at least simple measures for each of these four dimensions. Many past decisions about next steps in HPC have used word of mouth-based intuition and “industry trends.” The idea is to enhance these with measures representing concrete facts about the present as well as historical trends.

Following each point is a set of numbers referring to the productivity contributors listed in Section 1.1. Listed first with a * is the highest priority contributor today, and an unordered set of other major contributors follows. Note that the only primary contributor listed twice is 2, which is “better run-time hardware and software support for system-level control, performance, & reliability.”

6.1 High Productivity Computing Indicators

Individual Application Performance

The Linpack benchmark or an augmentation with Stream, represents peak performance of part of a system, as would a cluster/parallel I/O benchmark. SPEC HPC or SPEC parallel are application based benchmarks that give integrated performances for all parts of a system running these long-debated codes. SPEC HPC actually includes whole applications and large data sets, for limited

markets; such numbers provide good talking points. It is necessary that each specific system provide its owners and users with performance for their own codes and their own data sets, to support productivity in the sense of Table 1. {1*, 3, 4, algorithms - Section 5.1.1}

Jumbo Job Throughput

Section 5.4 defined this, and the number of such jobs per month over several years would indicate increases in large scale computations. This is an indicator that large jobs are indeed making effective use of a system. It presumes that the users and owners of the system are committed to running reasonable applications, and providing useful problem solutions. Furthermore, this is an implicit answer to the issue raised in the previous paragraph, in that each user is presumed to be getting good performance on jumbo jobs; the obvious next step would be to report on performance for these jobs, a more demanding task discussed in great detail in [Kuck96]. Finally, small job throughput is important on most systems, and must be balanced with jumbo jobs, according to owners' priorities. {2*, 1}

Applications Breadth

A matrix in the style of Table 2, Section 5.1.1 would indicate the breadth of applications and algorithms being run at an HPC facility, and a time series view would indicate productivity change. This is a diagnostic measure of how a facility is providing "better solutions" over time, if one assumes that productivity is enhanced as an HPC facility runs a richer mix of algorithms and applications types, together with an increase in jumbo job throughput and QoS. The matrix would reflect the specific applications areas of each given facility, including new algorithms being used. {3*, 4, algorithms - Section 5.1.1}

Quality of Service

Since the failure of a single processor can bring down a thousand processors that have been running for days, it is very important to have HW/SW error detection, prevention and recovery mechanisms in place (e.g. HW monitoring and reporting, checkpoint/restart, dynamic resource scheduling). This is currently a rather weak aspect of most HPC systems, so reporting on one or two QoS metrics, and reporting on sources of failures are important. They are, indeed, necessary conditions for improving productivity in HPC HW/SW systems. {2*, 1}

Finally, the subject that underlies many of the debates about HPC is money. The above measures should be used to signal where to spend money on research and development projects most effectively. This would tend to drive down prices and increase effectiveness most in the highest priority areas.

Acknowledgements

I am indebted to Rudi Eigenmann, Flash Gordon, Don Harbert, Bruce Leasure, David Lombard, and Ahmed Sameh, as well as three very insightful referees, for valuable comments about earlier drafts, and contributions to the final contents of the paper.

References

[BCKK89] M. Berry, D. Chen, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schnieder, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, and F. Seidl, "The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers," *Int'l Jour. Of Supercomputer Applications*, 3(3), pp. 5-40, Fall 1989

[BHNW01] Holger Brunst, Hans-Christian Hoppe, Wolfgang E. Nagel, Manuela Winkler, "Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach," *International Conference on Computational Science*, (2) 2001: 751-760, 2001

[CDCY99] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Waren, *Introduction to UPC and Language Specification*. Technical Report CCS-TR-99-157, May 1999

- [Couc89] Alva L. Couch, "Problems of scale in displaying performance data for loosely coupled multiprocessors." In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, March, 1989
- [DKKLS87] E. Davidson, D. Kuck, D. Lawrie, and A. Sameh, "Supercomputing Tradeoffs and the Cedar System," in *High Speed Computing*, Robert B. Wilhelmson, Ed., Univ. of Illinois Press, 1987
- [EiHa96] Rudolf Eigenmann and Hassanzadeh, Siamak, "Benchmarking with Real Industrial Applications: The SPEC High-Performance Group," *Computing in Science and Engineering*, pp. 18-23, 1996
- [Etnu04] www.etnus.com/Products/TotalView/
- [HKNP01] Jay Hoeflinger, Bob Kuhn, Wolfgang E. Nagel, Paul Petersen, Hrabri Rajic, Sanjiv Shah, Jeffrey S. Vetter, Michael Voss, Renee Woo, "An Integrated Performance Visualizer for MPI/OpenMP Programs," *WOMPAT 2001*, pp. 40-52, 2001
- [IDC03] IDC White Paper sponsored by HP, "Enabling Business Agility: HP's Adaptive Enterprise Strategy," www.idc.com, May 2003
- [KCDZ94] Peter J. Keleher, Alan L. Cox, Sandhya Dwarkadas, Willy Zwaenepoel, "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *USENIX Winter 1994*, 115-132, 1994
- [KeCh03] Jeffrey O. Kephart and Chess, David M., "The Vision of Autonomic Computing," *IEEE Computer*, Jan. 03, pp. 41 -50, 2003
- [Kuck96] David J. Kuck, *High Performance Computing: Challenges for Future Systems*, Oxford Univ. Press, New York, 1996
- [Numr99] Robert W. Numrich, "An Algebraic Theory of Parallel Algorithms with Examples from Co-Array Fortran," *PPAM'99/Kazimierz Dolny*, Poland, September 14-17, 1999
- [OCG] <http://www.OpenClusterGroup.org/>
- [OpenMP] <http://www.openmp.org/specs/index.cgi>
- [PeSh03] Paul Petersen and Sanjiv Shah, "OpenMP Support in the Intel® Thread Checker," *WOMPAT 2003 Proceedings*, June 2003
- [SHPT00] Sanjiv Shah, Grant Haab, Paul Petersen, and Joe Throop, "Flexible control structures for parallelism in OpenMP," *Concurrency Practice and Experience 2000*; 12, 1219-1239, 2000
- [Vali90] L. G. Valiant, "A Bridging Model for Parallel Computation," *CACM*, vol. 33, no.8., pp. 103 – 111, 1990
- [VeSu00] Jeffrey S. Vetter and Bronis R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," *Proceedings SC 2000*, Nov. 2000