International Journal of Parallel Programming, Vol. 22, No. 1, 1994

What Do Users of Parallel Computer Systems Really Need?

David J. Kuck[1]

Received date: May 1993

High performance computers have played key roles in many scientific and engineering advances over the past 40 years, and many more may be expected in the future. However, unless practical parallel systems can be produced in this decade, a performance crisis will arise by 2000 across the spectrum of systems from workstations to supercomputers. There is widespread confusion today about how best to proceed with future parallel systems because so many different approaches have been taken and the performance results have been so spotty. A fundamental flaw in our approach to parallel computing, as a nation, is the poor understanding we have obtained about delivered performance. This paper analyzes the situation and suggests fundamental changes that are necessary to achieve practical parallelism in this decade. A great deal of money is now being spent and more is planned, to advance the field, but money is not so much the problem as shortages of qualified people and a sharp focus for their work. Our national goals for the end of this decade must be the creation of an infrastructure for understanding performance, and its natural consequence, the development of practical parallel systems.

KEY WORDS: Computational science and engineering; practical parallelism tests; parallel performance; parallel software.

1. INTRODUCTION

High performance computing is a key technology in many scientific and engineering disciplines; this has been true for the past several decades, and its scope and importance are increasing over time. Parallelism has developed remarkably in the past decade, but its utility is still lacking in

[1] Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 465 C & SRL, 1308 West Main St., Urbana, Illinois, 61801. (kuck@kai.com).

828/22/1-7

many respects, relative to sequential computing. On the other hand as traditional clock speed increases fade, parallelism is becoming a more important technology. We have already seen supercomputer clocks stagnate over the past decade, and the same appears likely to happen for CMOS microprocessors in the next decade. Thus, parallelism will become a necessity for building faster computers in the 21st century.

After a decade of fast-track development, we reached the 1990's with the parallel processing imperative on everyone's mind: use parallel processing now or fall behind in R & D activities. Dozens of different commercial systems have been built, sold and then collapsed in the marketplace; but we still have not converged on architectures that are well regarded or broadly used.

The U.S. government has launched a national program, the *High Performance Computing and Communications Initiative* (HPCCI), and the European Community seems about to do the same, to apply massively parallel processing to various grand challenge computations. While there is great potential in these efforts, there are also substantial risks. There are risks in promising usable high-performance, but not delivering it through several generations of parallel systems. Already, some companies are turning away from massively parallel systems after trying one or more of them and then deciding that they are too difficult to use.[1] There are also risks in announcing the goal of achieving teraflops computing by 1996, but not having a clear idea of what that goal really means. The risks include wasting money in following too many paths, and more importantly, not having enough money to follow the key ones, or even to determine rationally what they are.

There is more confusion in high performance computing and parallel processing today than there has been in many years. Computer companies have lost their way concerning what kind of systems to build, and computer users do not know what to ask for in new systems. A by-product of this confusion is the pursuit of short-term goals that have little long-term benefit. When such pursuits consume much of the time of the few experts in this field, it is important to question both the current state of high performance computing and the future goals of our field. We believe that the national effort will be most successful if it focusses on the *greatest grand challenge* (GGC) of designing highly effective, practical parallel computing systems in the coming years. In this way, grand challenge computations will continue to benefit forevermore.

1.1. Two Eras

Figure 1 presents a century-long view of computing technology divided into two eras: the Sequential Computing Era and the Parallel

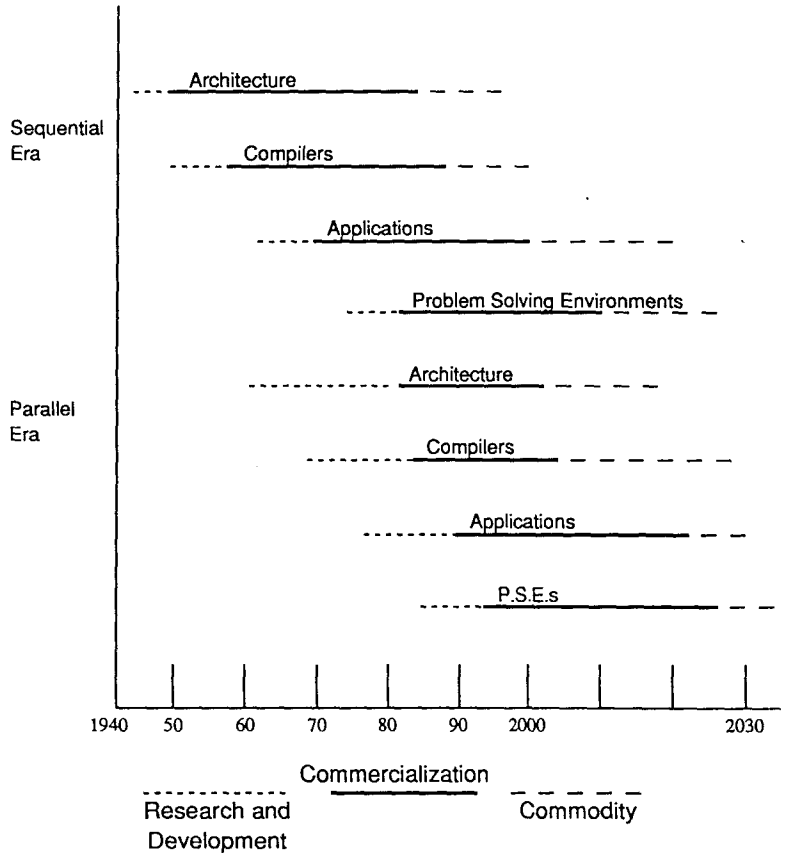

Fig. 1. Two computing eras.

Computing Era. Each era is further divided into four phases that are oriented toward usability and the user's view of systems. The architecture phase refers to the hardware systems alone, and to this are added phases of compilers which translate high level languages and optimize user-written programs for machine execution; applications software packages that free users from writing certain standard pieces of code; and problem-solving environments that integrate compilers, applications packages, and other software into "Do what I mean" software systems that free users from most programming chores. Finally, each of these phases in Fig. 1 is broken into three segments: the first segment is denoted by a dotted line and refers to a period of research and development efforts (R & D segment); the second segment, denoted by a solid line, refers to the release of commercial products as well as continuing R & D (commercialization segment); and

the third segment, denoted by a broken line, refers to a period when, although there remain a number of open questions, the topic is no longer regarded as a "hot research area," products are easy to produce, and prices and advertising direct the market (commodity segment).

The Sequential Computing Era began in the mid-1940's with the construction of a number of computer systems in research settings, as shown by the dotted architecture line in Fig. 1. By about 1950, computers were available commercially as denoted by the solid architecture line. Compiler research and development began in the early 1950's and commercial compilers became available by the late 1950's, as the dotted and solid lines show. By the late 1980's, we show the solid architecture and compiler lines giving way to broken lines as the merger of uniprocessor architecture and compiler ideas in RISC systems reached wide commercial acceptance.

In the early 1970's, commercial applications software became available on uniprocessors and by the early 1980's complete problem-solving environments (PSEs) emerged in the form of CAD systems, word-processing, and spread-sheet software. In fact, the low cost of microprocessor system hardware and the convenience of PSE software led to the personal computer and workstation revolutions of the 1980's. Note that we project the research phases of applications and PSE software to extend beyond the year 2000 for uniprocessors.[2]

Computing in the 1980's was made tremendously exciting for laymen and computer professionals alike by the emergence of lowcost personal uniprocessors as sketched earlier, as well as the introduction of commercial parallel systems. Figure 1 shows that the Parallel Computing Era began at just about the same time that uniprocessor PSEs made sequential computing affordable by the layman. The same four phases are shown unfolding for the Parallel Computing Era in the 1980's and through the mid-1990's.

The phase shift between the introduction of each of the four phases of the Sequential Computing Era is about ten years, and we estimate that the commercialization segments of this era each lasted about thirty years. This leads to two key questions for parallel computing: How great a time delay will there be between each of the four phases in the Parallel Computing Era, and how long will the commercialization segment of each phase last before high-quality practical parallel systems become easy to produce as a commodity segment is reached?

The optimistic answer is that because we have learned so much from the past fifty years of sequential computing, the phase shift will be reduced to, say, five years and each commercialization segment will drop to ten or fifteen years. The pessimistic answer, however, is that parallel computing is so much more difficult to understand and the design problems are so much more challenging, that these numbers will be much higher, perhaps exceed-

ing the corresponding times for developing the Sequential Computing Era. There are important policy issues for government, industry and academia here, which we will discuss throughout the paper.

The true answer will be revealed over time as research and development efforts advance parallel computing and as sequential computing speeds run their course. In the next decade it is likely that the rate of speed increase of microprocessors will fall off as it has for superprocessors in the past decade. Thus, users' attention may further be torn between the potential speed advantages of parallel systems and the existing usability advantages of sequential systems.

1.2. A Brief History of Parallelism

Figure 2 gives a brief history of parallel computing in terms of performance and technology levels. For each of three decades we show that relative to the time period, parallel computing offered the highest performance levels available.

Furthermore, the hardware technology has changed dramatically over these three decades. In the 1960's, parallel systems used the highest level of technology available, so the systems were difficult to manufacture, expensive and hence, rare. In the 1970's, a second generation of parallel systems arose that used simpler technology (e.g., bit-serial processors) and so were easier to manufacture, less expensive, and more commonly available. By the 1980's, with the advent of standard microprocessors and busses, and

		Relative to era		
		Peak Speed	Hardware Technology Level	Software Quality
1960's	U. of Ill. Illiac 4 Bell Labs PEPE	high	high	low
1970's	Goodyear AerospaceSTARAN ICL DAP	high	medium	low
1980's	Hypercubes SIMDs	high	low	low

Fig. 2. Brief history of massively parallel machines.

increasingly larger memory chips, the hardware aspects of parallel processing had become low tech, and many companies as well as university research projects were able to build parallel systems. On the other hand, in each period, it was difficult to exploit the parallelism and achieve the high potential performance levels because the software remained a very high technology item; i.e., the software did not perform as well as desired and was difficult to use.

Thus, the history of parallel computing systems can be viewed as one in which the building of larger and larger systems has become easier and easier in practice. But as their size has grown and, to some extent, their architectures have been weakened by lower-tech hardware approaches, the software challenges have remained enormous. So the appeals of peak speed have remained high, the appeals of low cost have grown (relative to fast scalar supercomputer processors), but the software appeal has remained low. In this climate entrepreneurial companies have introduced all manner of new products, but few end users have achieved production-level practical results from parallel computing to date.

2. SYSTEM DESIGN

The problem of designing a good parallel computer system is a constrained optimization problem. Each end user measures the system payoff using some criterion function that is proportional to system performance/cost. The system components are many, but we will consider five major ones (excluding OS to simplify the discussion):

- Hardware Units
- Architecture
- Algorithm Library
- Compiler
- Languages

The constraints provided by the real world include issues arising from:

- Hardware Technology
- Computational Science and Engineering
- System Buyers and Users

Because the cost and performance results for a parallel system have very complex nonlinear interactions, we cannot attempt a mathematical formulation of this optimization problem. However, a *logical* formulation is presented in Fig. 3. The arrows there show causal relationships in a given

system; much more complex relationships may be found, especially if one considers a sequence of computer systems over time.

2.1. Faulty Reasoning

Naive observers of Fig. 3 often reach misleading conclusions. The most common of these are reached by looking along the left-most path (Hardware Units, Architecture, System Performance) alone and observing the following:

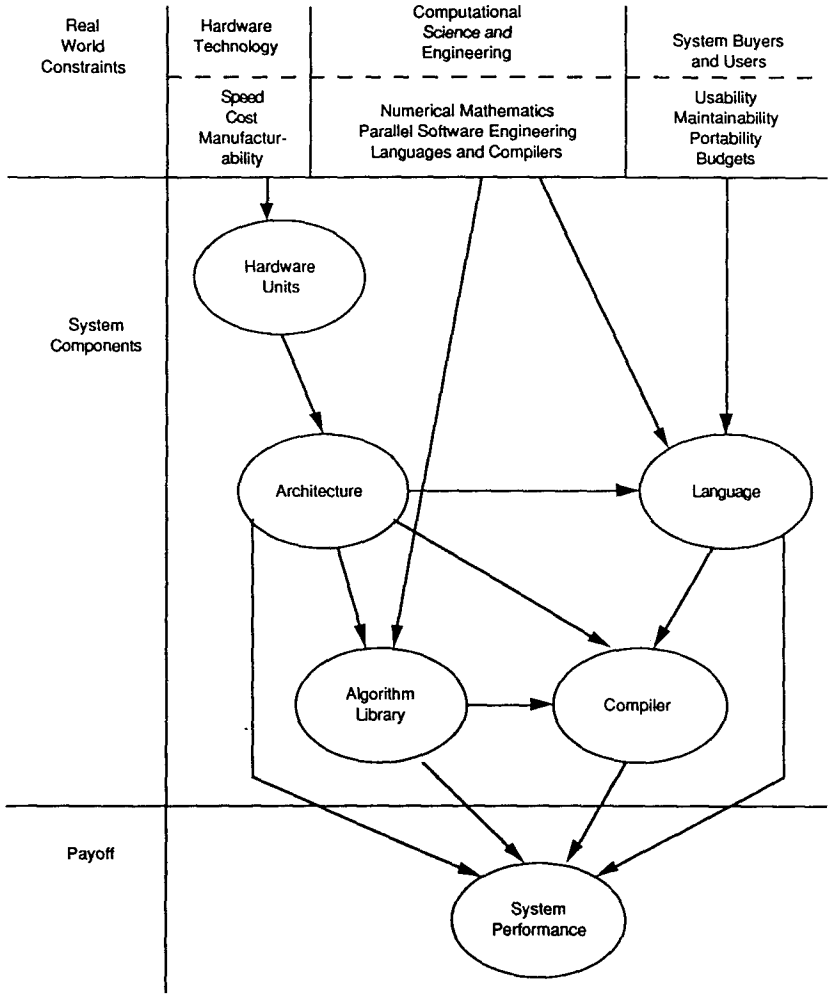

Fig. 3. The logic of system design.

- Peak speed (proportional to clock speed and number of processors) is roughly equivalent to delivered system speed.
- Higher speeds are just around the corner because of recent basic technology breakthroughs.
- System costs are coming down in proportion to the costs of certain basic technology.

Faulty reasoning along these lines leads to the erroneous conclusion that high-speed, low-cost parallel systems will continue or accelerate traditional performance increases over time. In addition to serious misconceptions about system speed's dependence on hardware and architecture, this reasoning ignores the very important subject of how performance and cost depend upon software. But examining the software issues is even more difficult than the architecture and hardware issues.

Another common set of misleading conclusions is reached by examining the right-most paths alone, and observing the following:

- A language that reflects the architecture will yield good system performance.
- Powerful compilers can exploit any program on any architecture.
- At a higher level, the software development process can be helped by placing more information and responsibility in the users' hands, e.g., through interactive compilation or the display of performance bottlenecks.

Faulty reasoning along these lines leads to the belief that new programming languages and tools can fill any gaps that may appear in the software for parallel computing. In fact, direct contributions to system performance are made by all of the system components shown in Fig. 3 (including language use), and users should not be burdened with much responsibility for performance in any case. Furthermore, architectures with serious design flaws cannot be "improved" with any amount of software.

2.2. Design Needs

To achieve good system performance, all of the components must be designed into the system properly, and they must be matched and balanced with one another. The designers' problem is to make each component sufficiently strong to support overall system performance, and sufficiently low cost to yield an affordable system. Overdesigning any part of the system can greatly increase cost, and underdesigning any part of the system can ruin performance.

In this paper, we focus on the perspective of end users of machines; what do they want and what do they need. Users know that they *want* better software and better system performance, but because they have embraced such a wide variety of new software and parallel architectures in the 1980's, it is obvious that they are confused. In the early 1990's, they are still searching for "the right stuff." This indicates that they do not really know what they *need*, and we will attempt here to separate their expressed *wants* from their real *needs*.

Simply put, we will summarize some of today's basic technology gaps in MPPs, and then translate this into action plans that address user needs. There are a number of obvious problems with today's parallel systems, and the magnitude of the problems may be regarded as proportional to the number of parallel processors used. We will focus on what we regard as the most serious of these:

- Too much latency in accessing the overall system memory,
- Too little compiler power for existing languages, and
- Too little performance and functionality in numerical libraries.

As a nation we have conducted a great parallel computing experiment during the past decade. A very wide range of systems was built and sold using tremendous amounts of venture capital, large company R & D funds and government contract funds. Many successful projects were completed and many projects failed. And yet, as a nation, we have learned very little from the billions of R & D dollars spent. Little performance data was collected about individual system's performance, and much less was collected about comparative systems' performance. Virtually none of this data is publicly available or is being used to develop better systems.

One of our strongest beliefs is that this situation should be changed immediately. A network accessible performance database should be developed that contains codes, broken into algorithms, together with performance information at various levels.[1,3] This would allow people including algorithm designers, compiler writers, system architects,[4] and potential system buyers or users to gain insight about the state of the field and to make plans for the future.

2.3. Performance Limitations and Potentials

Parallel system performance is limited by a number of system architecture and software factors, as well as the computations being run. The computations being run involve particular data and code structures as well as a given data size, each of which has obvious performance implications.

Together, the data and code structures and the data size help determine γ, the percentage of a program's sequential running time that can be executed in parallel. Exactly how these factors determine parallelism is a difficult subject; for many years the interrelationships affecting delivered parallelism have been core questions in the study of program restructuring, parallel programming languages, parallel algorithms, etc.

A very simple relationship between performance and γ was presented by Amdahl,[5] and is often referred to as Amdahl's Law. Using speedup ($Sp(P) = T(1)/T(P)$ where $T(i)$ is the best i-processor time) to represent performance and assuming that a fraction γ of $T(1)$ can fully exploit P processors, while the remainder of $T(1)$ runs on just one processor, we have

$$Sp(P) = \frac{T(1)}{\gamma T(1)/P + (1-\gamma)\, T(1)} = \frac{P}{\gamma + (1-\gamma)\, P} \leqslant \frac{1}{1-\gamma}$$

We plot the relationship between speedup and γ in Fig. 4. Even though it is an obvious relationship, this equation (and variations of it) has caused much discussion and confusion over the years, probably because of its two idealized assumptions. The assumption that the parallel part of a computation executes fully on P processors is optimistic, while the assumption that the remainder executes on just one processor is pessimistic. A more complete model was presented in Ref. 6, which assumed that fractions γ_p of the sequential time executed in parallel on p processors, $1 \leqslant p \leqslant P$. However, experimentally determining γ_p values is very difficult to do before the fact, and after the fact they are of marginal interest since the performance is known. Thus, although one can imagine that the optimistic and pessimistic assumptions approximately balance each other, real parallel system performance could be better or worse than the curve of Fig. 4 if more parameters were used. Nevertheless, the model is inescapably correct, following the assumptions made in its derivation.

Historically, Fig. 4 has been used to argue that parallel processing is a difficult, if not impossible alternative to faster sequential systems. In particular, in the 1960's and 1970's, the slope of the curve presented a formidable challenge to parallel processing enthusiasts. As Fig. 5 shows, when γ was in the neighborhood of 0.5 and the inequality gave $Sp \leqslant 2$, a reasonable change in $\Delta\gamma$ made only a negligible improvement ΔSp, in speedup. This was the situation when the curve was first presented and it held for many years. However, over the past twenty years, parallel processing has made progress on many fronts, and today computations frequently operate with γ well above 0.9, as shown in Fig. 4. Now, the same $\Delta\gamma$ that made a negligible performance difference twenty years ago, can cause a 16-

or 32-processor system to double or quadruple its performance. Still, for any size of P, the $Sp \leqslant [1/(1-\gamma)]$ bound of Ref. 5 holds, so even for 1 K or more processors, $\gamma = 0.99$ with this model upper bounds speedup at 100.

Figure 5 shows that the magnitude of P has increased substantially over the past three decades, as have γ values for compiled programs and for the best hand-tuned parallel programs. The top delivered parallel speedups have shot ahead in past decades far greater than average performances would indicate. Indeed, as parallelizing compilers have improved and good parallel programming style and know-how have been disseminated, compiled performance has risen dramatically for nonexpert

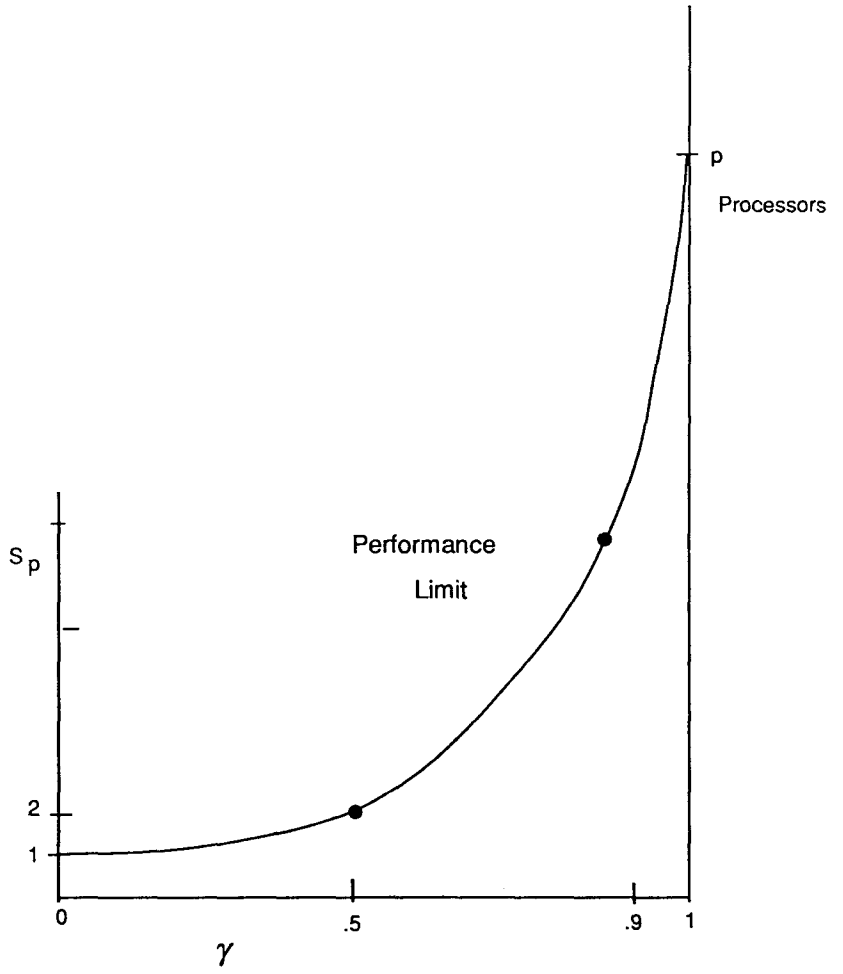

Fig. 4. Amdahl parallel speedup limit.

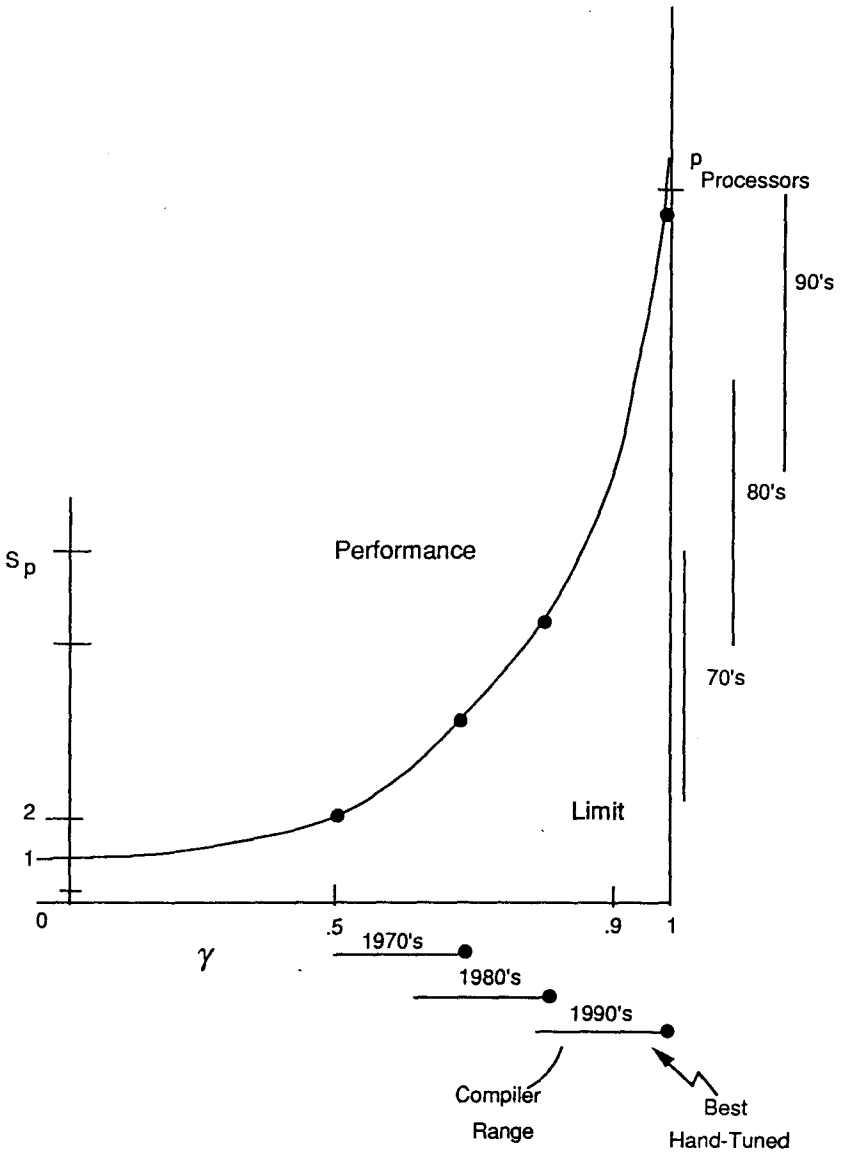

Fig. 5. Historical parallel speedup limit.

users with certain classes of codes. Thus, an outside observer of only the three data points marked on Fig. 5, which correspond to the best code running on the largest system available at each time, would correctly conclude that parallel processing has made tremendous progress over the past three decades, but might misunderstand that this is not across-the-board, general purpose progress. These experiences have helped fuel the parallel processing imperative of the 1990's, but they do not imply that practical parallel processing has arrived for ordinary users.

Taken at its face value, Amdahl's Law has served as a cautionary, even threatening guidepost to parallel processing system designers from the 1960's through 1980's as they proceeded across the low- γ portion of the curve. For the 1990's and beyond, however, this simple model may serve as a motivating stimulus to the next generation of parallel system designers whose incentive is to climb the steep, high- γ portion of the curve and make practical parallel processing a reality. Overall, this model offers us the optimistic possibility that when sufficiently many problems have been solved, progress toward practical parallel processing will accelerate.

3. WHAT REALLY MATTERS TO USERS IN HPC

To address the question of the paper's title, we will first ask the same question for users of workstations. In the workstation marketplace over the past decade, there has been a continuous cycle of performance enhancement and new applications software. Users obviously want to move toward the ability to do more complex computations in less time. Faster systems reduce the time for a given computation or accommodate more complex computations in a given allotment of time. New applications software expands a machine's functionality but usually requires more hardware performance for its support. Thus performance and functionality are the yin and yang that drive computer systems forward.

Superficially, it seems obvious that workstation users and supercomputer users are similarly motivated. It can be argued that supercomputer users have come to expect much less software than workstation users have, while at the same time reaping the rewards of much higher performance. On the other hand, the very purpose of HPCCI has come to be regarded as advancing a number of grand challenge computations.[7] By definition, solving the most challenging computational problems will require the joint efforts of many people and the joint development of new codes (as outlined later). Thus, on close analysis it must be agreed that the idea of supercomputer users as "super users" who can get by without much software support is anachronistic, so in the end, workstation users and supercomputers both want and need more performance *and* more functionality.

3.1. Large Code Development

As one example of the software needs in high performance computing, consider the problems of combining one code that computes the flow around an airplane, with another code that computes the airplane's structural dynamics, and a third that simulates fluid flow through the engines. By combining these three codes we might get a rather complete simulation of an airplane in flight. It is obvious that the use of parallel software engineering techniques would be of great help here, both in structuring the three original codes and in the process of combining them.[8]

An example that is even more dependent on careful parallel software engineering would be the combining of *parts* of several codes. Suppose that we want to create the world's best general circulation model of the global atmosphere. A good strategy should be to start with the best code available and then enhance it with better parts obtained from other codes, e.g., the best subroutines for cloud formation, ocean-atmospheric interaction, etc.

These two examples would challenge the state of the art in sequential software engineering as applied to physical modelling, and are currently unthinkable for parallel codes. Since the nation is attempting to drive HPCCI by such grand challenge computations, this work should be based on the most well-established and best-engineered codes available. Thus, one must conclude today that the best machines to use for such activities would be those shared-memory systems with the best established software systems. A good deal of time and effort seem necessary before distributed-memory MPPs can provide the necessary software support to become competitive with sequential systems or even shared-memory parallel systems. A crucial underlying problem is the effective memory speed of distributed memory systems (see Section 4.1). Nevertheless, the HPCCI effort has distributed memory MPPs as its cornerstone architectural component.

3.2. Problem Size vs. Scope

One of the recurring debates about the effectiveness of MPPs vs. shared-memory systems, concerns data size and memory size, as well as memory access time. It is argued that MPPs can easily provide massive, low-cost memory to handle the very large problems that will be necessary in grand challenge computations. In fact, the *scope* rather than the mere *size* of a computation is what matters in grand challenge computations. By data size we mean the total amount of data, but by scope we refer to the total amount of data as well as code, tohgether with the complexity of their interactions. The previous section gave an example of combining three

codes to get an accurate airplane simulation. While the data size and code size might increase by about 3X when combined, the scope might increase by an even greater factor. This would be caused by the fact that each of the codes would have to be made more complex to accommodate the others by adding coupling factors, more reality (physics, chemistry, etc.) more mesh points, etc., and this would cause the running time of each of the three modified codes to be much longer in the combined case than when run separately.

The interaction of various parts of a code is a crucial issue in parallel system performance for several reasons:

1. Parallelizing a code becomes much more difficult as loop structure complexity and the depth of procedure call nesting increases. These tend to increase as the scope of a code increases, because of interactions between the various program components.
2. To manage the memory of a parallel system, *locality* of computation is important, since nonlocal memory references are relatively expensive on any parallel system. As the scope of a computation increases, locality tends to decrease simply because the probability of different access patterns increases. For example, with a single algorithm we may find a highly localized data partitioning, but this partitioning is not likely to hold for the next algorithm required by a computation. Thus, it can be expected that as a code's scope increases, the inherent difficulties of obtaining good parallel performance increase substantially.

Of course, most individual codes become more complex in scope as time passes, and new reality and more data points are incorporated. On the other hand, some problems may actually decrease in size as time passes; oil reservoir simulation may focus on smaller domains as they are depleted by pumping, but increase in scope as more chemical and geological code is added.

3.3. System Software Design Goals

System software is the glue that binds applications codes to hardware systems, and its design has major performance consequences. At the bottom level, the operating system causes the hardware to start and stop parallel tasks, it decides which tasks can run with others, and it generally manages the hardware and software system resources. One level up, the compiler translates user-language programs into machine-language programs and can restructure the program and data structures in the

process. The compiler as well as users depend on libraries which provide functionality that is common to many programs, but that no user has the need to write. Each of these system software components can substantially affect performance. Nevertheless, no amount of software can overcome architectural flaws, for example, to reach a theoretical peak speed that is fundamentally unreachable because of architectural bottlenecks.

A fourth software issue is the programming language or environment through which users express themselves to the computer. Problem solving environments (PSE) are evolving rapidly in which users do not write programs but rather specify problems to be solved.[9] Computer-aided design (CAD) systems which have been in use in various engineering disciplines since the 1970's, and spreadsheet and word processing software that swept the world in the 1980's, can be regarded as the prototypes for future PSEs. A PSE can relieve the user of most programming burdens, and can also ensure high performance if the system itself is well-implemented. Thus, such systems are likely to be used in more and more applications areas in the future (recall Fig. 1).

3.4. Languages and Performance

Today we still must use programming languages for high performance computing and because of the difficulty of obtaining good performance from parallel systems, for HPCCI, language design issues currently remain at center stage. Languages must be sufficiently expressive and easy to use that users will easily accept them. Furthermore, languages (and other software tools) must allow old codes to be ported and new codes to be maintained on various machines, including MPPs, over long periods of time and from one set of people to the next.

It is obvious that languages can be designed that give users sufficient flexibility to provide top performance in every case. Unfortunately, such languages e.g. assembly language, have long since been rejected by most people as too difficult to use. Since the 1960's, the generally acceptable computer language level has risen from the machine level to the point where PC users now have problem-solving environments that do not require users to be programmers at all, but instead users may express themselves in terms of their own disciplines. Parallel processing cannot succeed by attempting to reverse this historical market force.

Suppose, though, that we attempt to design high-level languages which are easy to use and still reflect sufficiently much of the architecture that by using them people can easily obtain good parallel performance. The tradeoffs that must be considered in such a language design are:

- Usability will suffer, at least to some degree, as the language level must "drop," by definition, to accommodate performance-enhancement features.
- Portability will suffer as users "optimize" the code they write for a given machine.
- System design effort is misplaced because more work should be done on better algorithm libraries, compilers and architectures to provide what we really need: better performance that is obtained in a user-transparent manner.

The logical conclusions that we reach from these are:

A. Programming languages are very important in satisfying user-generated, system-design constraints concerning *system usability* (see Fig. 3). They mst provide expressiveness in a good programming model for new programs as well as portability and maintainability for existing codes. Software engineering principles must be used in writing large programs, and languages can be designed to encourage the use of these principles.

B. If A. is satisfied by a language, then system performance cannot be much of an issue for users as they write programs. Delivered performance on each intended target machine is important (see Fig. 3) and usability must suffer if user-invokable performance enhancement becomes a language design criterion. In practice, compilers must provide performance on each target machine that a user chooses.

A simple way of summarizing this is to say that language design is constrained by all of the arcs touching "language" in Fig. 3 and that languages should primarily be designed to satisfy system buyers and users. If some architectural features (e.g., vectors) are a natural part of the users' discipline, then they should be included in the source language. Performance, however, should be designed into the architecture, algorithm library, and compiler so that users' concern with it is minimal.

Two prominent Fortran language extension committees have developed designs in the past five years. For shared memory machines, *Parallel Computing Forum* (PCF) Fortran has evolved into ANSI committee X3H5,[10,11] and more recently the High Performance Fortran Forum has been evolving *High Performance Fortran* (HPF) for distributed memory workstation networks and MPP systems.[12] At this point, however, HPF reflects sufficiently much architecture-oriented detail, e.g., about memory management, that it appears to be better thought of as a target language for compiler output than as a user-oriented language.

As the field evolves, we must compromise to succeed. Furthermore, items that are given up in early compromises ay be taken black in later designs. Thus, language extensions by which users can easily obtain large performance boosts, which are (more or less) machine-independent, and which can be COMMENTed out as time passes are good early compromises. On the other hand, poor ideas for compromise would include those that do not match these earlier ideas or that seduce users into thinking that their use will lead to performance increases but do not because of compiler weaknesses or architectural flaws.

Thus, in the short run a given approach can satisfy certain users, but raise a number of questions about the long run. First, even for current machines, to maximize users' performance tuning possibilities, vendors may have to provide dialect extensions (e.g. to HPF) that allow exploitation of unique aspects of their own systems. Second, if parallel processing is to emerge from its current niche market and become a practical technology, it is essential that architectures be improved (e.g., in communication delays—see next section), and at the same time compilers must be improved substantially. Finally, since current architectures must change, the code that was written for earlier systems will need modification for new systems.

The forward thinking users who decide now to drop their old codes and rewrite them from scratch in HPF, for example, may find that as architectures and compilers evolve, they will be forced to repeat the rewriting process a number of times, before practical parallel processing arrives. Some industrial users have already dropped MPP projects due to their current lack of cost-effectiveness. This is not being written as a warning to users so much as a challenge to system designers. New systems must be designed nd built to deliver advances in the cost-effectiveness of parallel processing, in contrast to the too-frequent changes of the past that provided higher peak speeds with only the promise that "they might deliver better performance to some users."

4. PRACTICAL PARALLEL PROCESSING

4.1. Fast and Slow Memory Computations

We shall use the terms "fast-memory parallel computation" and "slow-memory parallel computation" to divide the world of parallel computation. We use the terms "fast" and "slow" to express the operational capabilities of a hardware system to meet the demands of a compiled code. From the point of view of the code running in each processor, if its demands are met on time, then the memory system appears "fast" to that processor, so a

fast-memory parallel computation is one for which each processor's memory demands are met on time throughout an entire computation; if a parallel computation is not a fast-memory parallel computation then we call it a *slow-memory parallel computation.* (An operational definition of "on time" here can be understood to be that the processor does not notice significant degradation to its cache's "average" uniprocessor performance.)

In parallel computers data must be communicated between processors, and the time required to pass this data is a crucial issue. If one processor is to access data in another processor's memory, it clearly must pay at least the time that a sequential machine would require for memory access, but there are substantial added penalties in a parallel machine. See Ref. 13 for a recent tutorial collection of parallel architectures. An interconnection network is required and this adds two kinds of penalties. First are the hardware delays along wires and through the switching devices used to route the data to its destination. As modern microprocessors are appearing with clock periods of 10 nanoseconds or less, propagating a signal across a printed circuit board and through silicon requires a significant fraction of a clock. Secondly there are operational delays caused by conflicts between two or more data items that are directed along the same path by a computation. On top of these delays, for actually transmitting the data, there is the address translation time which can be accomplished through hardware (as in most shared-memory systems), or can require software intervention (as has been typical of message-passing systems).

Because they are relatively easy to design and build, distributed-memory message-passing parallel systems became very popular in the 1980's. Using off-the-shelf components, they appeared to be inexpensive, and using simple interconnection networks, they appeared to be scalable up to large numbers of processors. But due to the message-passing paradigm, their performance was poor whenever there was much network traffic in a computation. In fact, the generation of MPPs produced in the early 1990's have system-wide memory access delays on the order of 100 μs and, using processors whose clock periods are a few tens of nanoseconds, the processor experiences a thousand-fold delay over a cache access. Thus, unless a computation largely avoids use of the system-wide memory, generating mostly local memory addresses, it will run as a slow-memory parallel computation. Of course, in a P-processor system whose total main memory size is M words, such computations effectively have a memory size M/P words.

On the other hand, shared-memory systems have tended to use custom parts which appeared expensive, and because they tend to use more complex networks, their scalability has been more questionable than distributed-memory systems. Today's shared-memory systems with a few tens

of processors have shared-memory access times that are not substantially worse than the factor of ten degradation experienced by uniprocessors for cache misses. Thus they typically can run jobs as fast-memory practical parallel computations, and each processor has access to the full M words of system main memory.

Overall, shared-memory systems have system memory access times that are one or two orders of magnitude smaller than distributed-memory systems, but the existing systems have one or two orders of magnitude fewer processors than distributed-memory MPP systems. Whether or not these shared-memory architectures can be scaled up substantially, without degrading shared-memory access time is one of today's important architecture questions. If parallel processing prevails in the 21st century, it will almost certainly be because system designers have been able to achieve fast-memory practical parallel computation for most users. This can be expected to happen only through the solution of hardware and software problems at the system level, while fully accounting for the applications demands of users.

4.2. (CLEAR) Compiler-Library Engineered ARchitecture

What is needed in parallel processing today is an integration of existing ideas that leads to more usable practical parallel systems. It took many years to integrate uniprocessor design and compiler technology and produce RISC processors, which led to a new set of difficulties but has delivered short-term performance/price breakthroughs. Although the analogous parallel processing problem is not yet solved, we offer an acronym that captures the key requirements for its solution: CLEAR, for Compiler-Library Engineered ARchitecture. A CLEAR parallel system would offer users good performance with existing languages (perhaps modified a bit) and library routines that had sufficient parallel power and broad functionality. Users would find the language-compiler easy to use and would find that the library contains familiar functionality that fits their applications. Note the relationship between architecture and compilers in Fig. 3. The system would have been designed as an integrated whole to *deliver* parallel performance to users, not merely to have overwhelming peak speed.

A CLEAR parallel system would *not* offer users language extensions or software tools that were advertised to help their parallel thinking (unless users wanted that for their applications domain), or to allow them more control over their data, or to allow them to examine the parallel static structure or execution characteristics of their code. In short, CLEAR parallel systems must allow users to think about their problem domains

and not about the parallel system being used, i.e., the systems solve problems but are not part of the problem to be solved.

The reader may object to this, regarding our desires as well-motivated but unachievable. Some may say that these have been the objectives of the field since the beginning and therefore do not need restatement now. Our point here is that the parallel computing field has developed well in the past decade, but it has not yet matured. We must not err now toward believing that it has matured with current MPP architectures and that we will have parallel systems software that can compete with sequential systems software by some fixed date in the next 5 years. Nor should we be drawn into the anachronistic approach that attempts to deliver systems now and lets users add software as they go.

Vector supercomputing developed in that way because the Cray-1 was the fastest scalar processor available in 1976. People were happy to use it for its scalar performance, and then tried to do better by vectorizing their codes. Today's MPPs require uch software working together correctly to deliver high performance, otherwise users get workstation performance for supercomputer prices. Again, we should not deliver weak systems to users with the notion that we may have gone as far as necessary, and the users can do the rest.

We believe that our objectives are achievable, at least for certain classes of computations. The field must evolve practical parallel hardware and software systems to meet these objectives.

4.3. Parallel Processing Needs and the Practical Parallelism Tests

The term "practical parallelism" can be used to refer to the eventual goal that we have been discussing for parallel processing technology. In this section we shall define five practical parallelism tests that allow us to discuss the subject's parts. The first three tests should be clear following the discussion of this paper; before listing the tests we present background material explaining the last two tests.

The parallel systems that we seek must have a certain performance robustness to qualify as practical. In particular we must be able to run our codes on a certain range of processor counts, not just on the whole machine or some fixed segment of it. This is the case because a user may want to pay for different performance levels at different times, the O.S. may face scheduling constraints when the system is heavily loaded, etc. We will refer to this as *code scalability* on a given parallel system. Additionally, it is necessary that the system possess *architectural scalability*. That is, systems with various processor counts must be implementable using the

same architecture but without varying the underlying hardware technology. This allows a range of system performance levels to be implementable.

A final test that parallel systems face, which sequential ones do not, is that of technology *reimplementability*. The issue here is that a successful architecture cannot depend upon a given technology, so that when a new technology mix becomes current, the system must be reimplementable. One way of failing this test is by using very fast hardware in one part of the system, relative to the remainder of the system, so that if the remainder of the system is reimplemented in a fast, new low-cost technology the one part, having no new faster technology base, becomes a bottleneck.

Practical parallelism has not yet been demonstrated; in fact, no standard definition of it exists. It seems clear that there should be "laboratory level" and "commercial level" criteria for judging practical parallelism, and we will now propose five criteria that add up to a Practical Parallelism Test.[14]

At the laboratory level, we will use as our criterion for the success of parallelism.

4.3.1. The Fundamental Principle of Parallel Processing (FPPP)

Clock speed is interchangeable with parallelism while:

A. maintaining delivered performance, that is
B. stable over a certain class of computations.

There are really three statements in the FPPP: first, the well-established point that high peak speeds are possible through parallelism, and then two important constraints that we shall use as Practical Parallelism Tests (PPTs).

Practical Parallelism Test 1: Delivered Performance

The parallel system delivers performance, as measured in speedup or computational rate, for a useful set of codes.

Practical Parallelism Test 2: Stable Performance

The performance demonstrated in PPT1 is within a specified stability range across a useful set of codes as the computations vary with respect to program structures, data structures, and problem sizes.

Next we discuss two additional tests that must be met if one has demonstrated the FPPP and wants to use it in a commercially viable product.

Practical Parallelism Test 3: Portability and Programmability

The computer system is easy to port codes to and to program, for a general class of applications.

Practical Parallelism Test 4: Code and Architecture Scalability

The computer system effectively runs each code/data size on a range of processors, and each code can be scaled up or down with respect to architecture.

Finally, if the first system is a success and the company is to survive over time, the system must demonstrate.

Practical Parallelism Test 5: Technology and Scalability Reimplementability

The system architecture must be capable of being reimplemented in new, faster or less expensive technologies as they emerge.

It is important to realize that, despite the great enthusiasm for parallel processing today, not even the Fundamental Principle of Parallel Processing has been demonstrated beyond rather narrow classes of computations. Substantial amounts of work will be required before the remaining three PTTs are passed. Inasmuch as this may require a number of years to achieve, it is important to consider the training of new people relative to these points. The scope of the practical parallelism problem can be reflected across a wide academic spectrum, as we shall discuss next.

5. THE FUTURE ROLE OF ACADEMIC CSE IN ADVANCING THE GGC

5.1. Computer Science

The future role of academic computer science and engineering is currently under increasing debate.[9,15,16] Simply put, the field arose a few decades ago because computers were being built and used to solve a growing range of real-world problems. Initially, these problems and the computer systems themselves provided a tremendous stimulus for the field. Subsequently, computer science has slowly turned its back on its roots. There were probably several reasons for this, including:

- The desire of computer scientists to free themselves from the subservient role of machine builders and programmers for other well-established academic fields that used computers and wanted help.
- The fact that new, interesting and indeed useful problems were discovered which could occupy the attention of many researchers; these

included the theory and practice of algorithms, artificial intelligence, databases, languages and operating systems, etc.

- The crush of new students in the 1980's whose curiosity was probably stimulated by home PC's and was fueled by the excitement of these new fields of research.
- The growing importance of computers in the world, adding to the budgets, publicity, and importance of the field. This led computer scientists to feel confident that they could chart their own courses.

In the 1990's, we face a crisis in computer science education.

A. Enrollments are down, probably partly because of demographics and partly because of a cooling of student interest in the field.

B. Funding is in question as the economy, peace, and the interplay between academic disciplines ebb and flow.

C. Future research directions are debated as the field matures on the one hand and new technical problems arise on the other.

Our purpose here is not to deal broadly with these issues, but rather to focus on one important piece of the whole puzzle.

5.2. Computational Science and Engineering

Computational science and engineering (CSE), along with theory and experimentation, is now being called the third branch of science and engineering.[17] We view CSE as the direct extension of those subjects that started the academic computing field in the 1940's and 1950's and, beginning in the 1960's, led to the formation of academic computer science departments and caused many electrical engineering departments to become electrical and computer engineering departments.

We define *computational science and engineering* as the study of the *whole* computational process of solving problems in science and engineering. On the one hand the name is a generalization of "computational chemistry," "computational electronics," "computational physics," "computer aided design," etc. On the other hand it is a variation on "computer science" and "computer engineering" that refers to the application of scientific and engineering principles (rather than the current, often-used intuitive approach) to the whole computational process of solving problems. It does not include the theoretical and experimental aspects of the discipline to which computing is being applied. Nor does it include those parts of modern theoretical or experimental computer science that are completely divorced from the principles of designing computer systems to solve

problems, (e.g. artificial intelligence as a theory of human thought processes, or abstract complexity theory).

There are several subjects that have not been of the greatest interest in computer science and computer engineering in the recent past, which we now see as essential to the future of high-speed computing. The advent of commercially available parallel computer systems in the 1980's must be regarded as a major milestone in the history of computing. Parallel systems are being built of necessity, to achieve high performance; the fastest clock speeds have improved very little in the past decade, so parallelism is the architect's only hope. This means that for the first time in history, computer architecture is not just an interesting academic subject, but rather the future of high speed computing seems to depend on it. Architecture of the future must become a discipline that is deeply rooted in the performance analysis of earlier systems. (See Section 2.2.) Furtherore, compiling for parallel machines is a major challenge. Whereas compiler research for sequential machines was pretty much a closed subject by the early 1980's, parallel compilation has blossomed in the past decade, and is currently a very important subject.

Finally, parallel algorithms need substantial development and implementation in useful libraries. As architectures become more complex, algorithms to exploit them become more difficult to understand. Unfortunately, numerical library research and development has not kept pace. Much attention has been placed on reimplementing and repackaging algorithms for traditional problems (e.g., LAPACK from EISPACK, and LINPACK), but little effort has gone into sparse algorithm libraries, or any libraries for parallel machines.

These basic subjects together with material that integrates various aspects of large parallel codes should form the core of CSE. Integrating material might include:

1. The structure of large programs with performance implications, e.g., program structures, data structures, data generation and analysis, visualization.[18]
2. Great Equations and Their Solution Techniques, e.g., Maxwell's, Navier-Stoke's, Boltzmann's, etc.; Monte Carlo, Linear Algebra, Table-Lookup, etc.
3. Parallel Software Engineering, e.g., structure of codes for high performance parallel computation, and data for parallel memory; good programming style.
4. Performance Evaluation and Improvement, e.g., comparative system performance analysis; system component performance improvement techniques.[3]

5. Problem Solving Environments, which form a direct software link between traditional computer science and many other departments, as outlined next.

5.3. PSEs' in CSE

An excellent starting point for cooperative research and development between CS, CE and other engineering and science departments would center on existing CAD systems. Consider the benefits and problems of adapting an existing CAD system to automate the figures and expand the homework in an engineering textbook. There would be manifold benefits in allowing students to see dynamic, 3D, color graphics rather than static, 2D, black and white textbook figures. Furthermore, the students could manipulate the "figures" by changing the load on a beam or the input voltage to a circuit. Similarly, all homework could consist of "machine problems" that were based on the figures and text as built into a CAD system.

The first challenging problem here would be to use the CAD systems to build up simple designs (discipline-oriented work) and then to develop new user interfaces for interacting with the figures and for solving the homework exercises (computer science-oriented work). This use of CAD systems makes them effectively *Computer-Aided Simulation* (CAS) systems. The potential payoff of this should be enough to entice the CAD system companies to cooperate by releasing (parts of) their source code for academic enhanceent. This would allow academics to build software on top of the existing CAD systems for user interfaces, to interface two CAD systems, and to study their structures for adaptive use. The real goal of computer science research would be in developing tools, software engineering methods and frameworks to allow this kind of activity to proceed easily. It could also open the CAD companies to academic cooperation in developing parallel versions of their systems for higher performance.

This scenario could also open the door to building *Computer-Aided Research* (CAR) systems which differ from CAD systems as follows. Given an input specification, a CAD system produces answers that users desire and can generally believe. On the other hand, a CAR system produces answers for which, if they appear plausible, the user seeks to study their derivation. In other words, a CAD system is treated like a black box, whereas a CAR system is a transparent box full of smaller transparent boxes, each of which bears examination and enhancement in building better models. Examples here include modeling a weather system, the folding of a protein, or the heart pumping blood. Notice that a CAR system will spin off CAS systems to be used in accurate simulators of engineering design or of the physical world.

5.4. CSE Summary

Much work is needed in curriculum development for CSE, and it is necessary that computer engineers, computer scientists, and a wide range of applications people be involved in the effort. The subject is controversial today as it seems to attack and compete with current academic computer science and computer engineering activities in terms of A, B and C.[19] It must be realized that CS and CE will be revitalized by a cooperation with CSE. Developing CSE will have immediate payoff in helping to solve a number of the problems discussed earlier. It will also have long-term benefits in educating a new generation of people to think in new ways that will, in the end, solve the greatest grand challenge.

6. CONCLUSION

We conclude with some overall recommendations which seem most important for the future of high performance parallel computing.

- In order to succeed, parallel computing must deliver the basics: good architectures with low communication delay and high bandwidth, high performance algorithm libraries with broad functionality, and powerful parallelizing compilers.
- The greatest grand challenge (GGC) is to develop a methodology for improving parallel systems generally. Progress on the GGC will have wide benefits in the future and will aid all of the applications grand challenges. For this reason, the GGC should be our central focus.
- A national performance metacenter in the form of a network accessible database is an urgent need to allow people to attack the grand challenge. It would contain grand challenge codes together with global performance data and detailed performance information about the basic algorithms used. With it, comparisons could be made between all machines.
- Short-term software bandaids are misplaced efforts that cannot overcome architectural defects, misdirect highly talented software people, and confuse users. These include reimplementing sequential libraries and developing software that forces users to struggle (more or less) with performance.
- Computational Science and Engineering must be seen as a means of revitalizing and enriching Computer Science and Computer Engineering, not as competing with them. Many of the ideas being

developed in these academic disciplines are useful in pushing parallel processing toward its maturity and advancing the GGC. A CSE curriculum enhancement in the short-term is an absolute necessity.

These five points contain the essence of what we feel is necessary to minimize the time to success for parallel computing. They are evolutionary steps that do not force radical changes on computer researchers, designers, or users. We feel that high performance computing would significantly benefit from each point in the short run.

ACKNOWLEDGMENTS

I am greatly indebted to many colleagues at CSRD for help in developing these ideas over the years, especially George Cybenko, David Padua, and Ahmed Sameh. The strong opinions stated here, of course, are mine and not theirs.

REFERENCES

1. George Cybenko and David J. Kuck, Revolution or Evolution? *IEEE Spectrum Special Issue: Supercomputers*, **29**(9):39–41 (September 1992).
2. E. Gallopoulos, E. Houstis, and J. R. Rice, Future Research Directions in Problem Solving Environments for Computational Science, *Report of a Workshop on Research Directions in Integrating Numerical Analysis, Symbolic Computing, Computational Geometry, and Artificial Intelligence for Computational Science*, NSF, Washington, DC, April 11–12, 1991. CSRD Report Number 1259, Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois (October 1992).
3. David Kuck and Ahmed Sameh, A supercomputing Performance Evaluation Plan, *Proc. of First Int'l. Conf. on Supercomputing, Athens, Greece, Lecture Notes in Computer Science*, T. S. Papatheodorou, E. N. Houstis, C. D. Polychronopoulos (eds.), Springer-Verlag, New York, **297**:1–17 (1987).
4. Lyle D. Kipp and David J. Kuck, Newton: Performance Improvement through Comparative Analysis. CSRD Report Number 1286, Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois (February 1993).
5. G. Amdahl, The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, *AFIPS Proc. SJCC*, Vol. 30 (1967).
6. Utpal Banerjee, Speedup of Ordinary Programs. CSRD Report No. 222, UIUCDCS-R-79-989, Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois (October 1979).
7. Committee on Physical, Mathematical, and Engineering Sciences; Federal Coordinating Council for Science, Engineering, and Technology (FCCSET); White House Office of Science and Technology Policy, *Grand Challenges 1992: High Performance Computing and Communications. The FY 1992 U.S. Research and Development Program. Supplement to the President's Fiscal Year 1993 Budget*, Washington, DC, Chapter 3, pp. 25–39 (1993).
8. David Kuck, A User's View of High-Performance Scientific and Engineering Software Systems in the Mid-21st Century, *Expert Systems for Scientific Computing*, E. N. Houstis, J. R. Rice, and R. Vichnevetsky (eds.), North-Holland, Amsterdam, pp. 69–87 (1992).

9. John R. Rice, Academic Programs in Computational Engineering and Sciences, *Computing Research News*, 3(1):11–12 (March 1991).
10. Mark D. Guzzi, David A. Padua, Jay Hoeflinger, and Duncan H. Lawrie, Cedar Fortran and Other Vector and Parallel Fortran Dialects, *The Journal of Supercomputing*, 3:37–62 (1990).
11. Bruce Leasure, Walt Rudd, Ross Knippel, Andrew Ingalls, and Cherri Pancake, Parallel Processing Model for High Level Programming Languages, Document No. X3H5/91-0023-G, X3H5 Technical Committee on Parallel Processing Constructs for High Level Programming Languages, American National Standards Committee on Computers and Information Processing (X3), p. 19 (March 1992).
12. High Performance Fortran Forum, *High Performance Fortran Language Specification*, David Loveman (ed.), Rice University, Houston, Texas, p. 150 (January 1993).
13. Dharma P. Agrawal, *Advanced Computer Architecture*, IEEE Computer Society Press, Washington, DC, p. 383 (1986).
14. D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C.-Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U. M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner, The Cedar System and an Initial Performance Study. To be presented at the *Int'l. Symp. on Computer Architecture*, San Diego, California (May 1993).
15. David L. Parnas, Education for Computing Professionals, *Computer*, **23**(1):17–22 (January 1990).
16. Nathaniel S. Borenstein, Colleges Need to Fix the Bugs in Computer Science Courses, *The Chronicle of Higher Education*, pp. B3–B4 (July 1992).
17. Robert Pool, The Third Branch of Science Debuts. *Computing in Science*. A special section of *Science*, **256**:44–47 (April 1992).
18. Cybenko, George, Lecture Notes for ECE 371, Topics in Electrical and Computer Engineering, Section GC: Large Scale Scientific and Engineering Computations (unpublished), University of Illinois, Urbana, Illinois (January 1992).
19. Juris Hartmanis and Herbert Lin, (eds.), *Computing the Future: A Broader Agenda for Computer Science and Engineering*, Committee to Assess the Scope and Direction of Computer Science and Technology; Computer Science and Telecommunications Board; Commission on Physical Sciences, Mathematics, and Applications; National Research Council, National Academy Press, Washington, DC, p. 272 (1992).

Printed in Belgium
Verantwoordelijke uitgever:
Hubert Van Maele
Altenastraat 20 – B-8310 St.-Kruis