HIGH PERFORMANCE COMPUTING: ARE WE JUST GETTING WRONG ANSWERS FASTER?[1]

Mark A. Stadtherr
Department of Chemical Engineering
University of Notre Dame
Notre Dame, IN 46556 USA

In the first part of this presentation we will take a brief look at the tremendous growth in computational power that is ongoing, from desktop machines to high performance computing hardware. It will come as no surprise to this audience that computational power has grown very quickly and continues to grow very quickly. The availability of all this power allows us to solve problems much faster. This means we can solve larger problems involving more complex and more realistic models. It allows us to solve problems we probably would not have even considered trying to solve years ago, because of their computational demands. All of this has made possible significant advances in many fields of science and engineering. But, another thing we can use all this computer power for, that is often overlooked in the quest to solve problems faster and faster, is that we can use it to solve problems more reliably—in fact to actually provide mathematical and computational guarantees of reliability. So in the second part of this presentation we will focus on issues of reliability—what good is high performance computing if we are just computing the wrong answer faster?

Growth in computational power.

The historical trend is that computational performance increases by about two orders of magnitude a decade, and this is a trend that we should expect to see continue into the foreseeable future. What is happening is that we have been riding the rapid growth stages of a continuing series of technological advances. As growth in one type of advance levels off, rapid growth due to some other advance kicks in, and thus growth continues at a rapid pace.

Looking at single-user PC performance in Figure 1, the two orders of magnitude per decade growth rate is apparent, putting us over 100 MFLOPS (millions of floating point operations per second) today. This is based not on a theoretical peak rate, but the rate on solving a 100 x 100 system of linear equations (the LINPACK-100 benchmark). For reference, Figure 1 also shows a couple of old machines that I used to work with: the CDC Cyber 175 mainframe and the Cray X-MP supercomputer. Clearly, we now have come a long ways since then, especially in price/performance ratio. The Cray X-MP cost around 8-10 million dollars; a comparably performing machine today would run around 1000 dollars, and by Christmas shopping time in 1999 we should expect this to drop to around 200 dollars. The PC is rapidly becoming a "disposable" applicance.

---

[1] Presentation given at CAST Division Awards Banquet, November 17, 1998, Miami Beach FL

The growth in performance of multi-user workstations (Figure 2) has followed a very similar pattern, again about two orders of magnitude a decade. However, the growth here, at least based on single processor machines, has not been quite as fast as in the PC case, leading to the increasing overlap today in the low-end workstation and high-end PC markets.

Looking at high performance computing hardware (Figure 3), the situation is somewhat different, but in the end still the same. If we look at the growth in single processor performance using vector processing technology (based on the Cray T94), it is not that impressive—less than an order of magnitude per decade. This is based on performance solving a 1000 x 1000 linear system (the LINPACK-1000 "Toward Peak Performance" benchmark). If we allow for multiple vector processors (based on the Cray T932), we can do much better. However, to really see the growth here we need to compare state of the art then (Cray 1-S) with state of the art now (Intel Sandia Red) and look at performance on the largest problem solved (the R_{max} benchmark). R_{max} for the Cray 1-S is little different than its performance on the LINPACK-1000 benchmark, so again we see the two orders of magnitude per decade growth rate, which, if anything, in recent years has actually been accelerating, due to the use of massively parallel architectures.

Looking at the current situation in parallel computing (Table 1), the latest hardware gives a peak performance of around 4 teraflops (million MFLOPS), with a likely R_{max} of over 3 teraflops. At Notre Dame, I am currently working with a group of people that is already working on the technology and algorithms for petaflop machines—that is, 10^{15} floating point operations per second (a billion MFLOPS). One trend that we can see here is that these high end machines use massively parallel computing; and they do this with commodity level chips. For example, the Intel Red machine is based on Pentium Pro processors.

On a more commercial mainstream level, the trend is toward shared-memory machines, from 4 to 32 to 64 processors. These are very easy to use machines, and the word "mainstream" should be emphasized here. These machines are very widely used in the business world, as servers and for various other applications. For example, looking at the June 1998 listing of the top-500 supercomputing sites, one will find several banks, several telecommunication companies, and various other companies such as American Airlines, Raytheon, Federal Express, Rubbermaid, Allstate, etc. So we should no longer think of parallel computing as some kind of exotic technology—it is much more widespread than most people probably think.

Another trend is toward the use of network-based systems. Essentially, if you have a number of computers connected in a network, you have a parallel computer. The rapid growth in networking and its projected future growth, including all the new wireless technology, has led people to think about a concept called "metacomputing". Metacomputing refers to the use of a heterogeneous network of computing resources, that may range from simple microprocessors to high-end multiprocessors. The analogy that is frequently used is that plugging into a metacomputer would be just like plugging into the electrical power grid. You would get as much computational power as you need, when you need it and where you need it, by grabbing compute cycles off the network. And just like I do not know or necessarily care where the electricity running my workstation has been generated, you would not necessarily know where the compute cycles you are using are actually coming from. This is a still developing concept, with much research still needed

in algorithms, software and applications. In fact, there is a metacomputing group at Notre Dame that I currently interact with, looking at process engineering applications.

One interesting development along these lines is Jini. This is a system developed by Sun Microsystems, and announced in Summer 1998. It is based on Java, and is designed to allow machines ranging from computer-controller appliances to supercomputers to talk to each other and share computing power across a network. The idea that a lot of excess processing power resides today in appliances, such as refrigerators, is something that was mentioned in July 1998 at the FOCAPO meeting in Snowbird by Larry Smarr, who was the keynote speaker. This led to some joking about the potential power of a "Refrigerator-Net". Something like Jini makes that a little less far fetched. In fact it may not be far fetched at all. On September 30, 1998 a story appeared on the New York Times web site proclaiming "Refrigerator, Computer Combined." The story went on to say that this product is being marketed as an "Internet Refrigerator," and that it has a Pentium II microprocessor, a huge hard drive—and separate compartments for fruits and vegetables.

What does this growth in computational power mean in process engineering? This is a question that will be discussed in concluding this presentation. For now, I want to emphasize that a problem has been, in this area and in others, that existing problem solving strategies were developed under a serial computing paradigm, and thus may take little advantage of advanced computing architectures, such as vector and/or parallel computing. So, there really is a need to be rethinking the way we solve problems. This is an area that I have been very interested in over the years, and I will provide one example to demonstrate the point.

This example (Figure 4) involves a dynamic simulation run using Aspen Technology's SPEEDUP package on a Cray C90 vector machine not too many years ago, and shows what happens when you change sparse matrix solvers in order to try to take better advantage of vectorization. This was a comparison done by Steve Zitney in collaboration with people at Bayer [1]. What this shows is that with the conventional sparse matrix solver of the time, MA28, the simulation took about 12 hours, which, since this was a simulation of a much shorter period of actual plant time, was not a good thing. By changing to the FAMP solver, which was developed by Steve Zitney in my group, and which takes advantage of the vector computer architecture, the simulation time was reduced by an order of magnitude, and the time to solve a single linear system by two orders of magnitude.

Reliability in Computing

Now shifting gears, consider the question: With all this computing power, can we in fact reliably compute the right answers? To explore this issue, we will look at some examples. The first example is the relatively well-known problem due to Rump [2]. Here we are asked to evaluate the expression

$$f(x,y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/2y$$

for $x = 77617$ and $y = 33096$. All numerical inputs in this calculation are exact machine numbers, so any errors we get in the result are due to the computation. Looking at the computed results from

a Fortran program, which Rump did on a IBM S/370, and others have repeated on many other machines, we see that when using single precision the result is

$$f = 1.172603...$$

when using double precision, the result is

$$f = 1.1726039400531...$$

and when using extended precision, the result is

$$f = 1.172603940053178...$$

The fact that the answer does not change with increasing precision is often taken as confirmation that the correct answer has been obtained. However, the correct answer is, in fact,

$$f = -0.827396059946...$$

So we didn't even get the sign right!

The problem here is due to rounding errors, combined with other difficulties, such as cancellation errors, that are inherent in the use of floating point arithmetic. A frequent reaction when people see this example is "so what, this will never happen to me" and "even if it does happen to me, it will be no big deal." So consider now a couple of real world examples.

On February 25, 1991, during the Gulf War, an American Patriot missile battery fired at an incoming Scud missile but failed to intercept it. The Scud missile struck an American Army barracks and 28 soldiers were killed. During the Gulf War, the U. S. Army had been claiming a successful intercept rate by Patriot missiles of 80% in Saudi Arabia. This estimate was scaled back to 70% shortly after the war. However, in a later congressional investigation, testimony indicated that "the Patriot's intercept rate could be much lower than ten percent, perhaps even zero."

So, what was the problem? It turns out that the computation of time in a Patriot missile, which is critical in tracking a Scud, involves a multiplication by a constant factor of 1/10. The number 1/10 is a number that has no exact binary representation, so every multiplication by 1/10 necessarily causes some rounding error. In the case of the Patriot missile, the accumulated rounding error was sufficient to cause it to mistrack incoming Scuds and thus miss them, with deadly consequences—and all due to bad computer arithmetic.

The European space agency spent 10 years and 7 billion dollars to develop the Ariane-5 rocket. On June 4, 1996, the first Ariane-5 was launched. At 39 seconds after liftoff it exploded, destroying the rocket and cargo valued at half a billion dollars. So what happened? It turns out that the explosion was caused by activation of the self-destruct mechanism built into the rocket. The self-destruct was triggered by unusually large aerodynamic forces that were ripping off the boosters. These forces were due to an abrupt course correction made by the on-board steering computer, which was in compensation for a wrong turn off course that in fact *never took place*.

The inertial guidance computer had told the steering computer that the rocket had gone way off course, when in fact it was not off course at all.

What caused this turn of events? It seems that what happened was that in the computations done by the inertial guidance computer it was converting a 64-bit floating point number into a 16-bit signed integer number. At about 36 seconds into the flight, a number was encountered that was larger than 32768, which is the largest possible 16-bit signed integer, so the conversation failed. Thus, erroneous numbers were sent to the steering computer, causing it to think the missile was off course and leading to the explosion at 39 seconds into the flight. Again a very costly disaster due to bad computer arithmetic.

Difficulties like this have caused some in the computing industry to suggest a rethinking of computer arithmetic paradigms. Originally computers used fixed point arithmetic. However, while fixed point arithmetic continues to be used in some special applications, there was a major paradigm shift in the mid-1950s to floating point arithmetic. At the time, this shift was the cause of some controversy. Accuracy was one main concern, since error analysis is much more complicated under the floating point paradigm. Alston Householder reportedly pronounced that he would never fly in an aircraft designed with the help of floating point arithmetic. The biggest drawback to floating point, however, was that it was very much slower than fixed point, and given the computers of the time, this was a very serious concern. But it was much easier to write programs in floating point arithmetic and so that paradigm won out.

Today, at least one major computer hardware and software company is seriously considering another computer arithmetic paradigm—namely, interval arithmetic. This is slower than floating point, so in that sense presents an issue similar to what had to be considered in moving from fixed to floating point in the 1950s. However, today we have ample computing power to deal with this issue. What is the advantage of interval arithmetic relative to floating point? Mainly it is an issue of reliability. In floating point arithmetic, if we add two numbers, say $c = a + b$, even if a and b have exact binary representations, the result c in general will not, and so the result of the computation will have rounding error, which may then continue to propagate. In interval arithmetic, if we add two numbers, we actually add two degenerate intervals, $[a,a] + [b,b] = [(a+b),(a+b)]$. Then the lower bound of the result is rounded down to $(a+b)^-$ and the upper bound rounded up to $(a+b)^+$. In this way, the computed result $C = [(a+b)^-,(a+b)^+]$ is a very narrow interval that is known to contain the correct result c.

The use of interval arithmetic has some interesting implications when it comes to problem solving. For instance, just consider the problem of solving $10x = 1$. Mathematically the answer is 1/10, but as we have already seen, this has no exact binary representation. So, in fact, solving the equation $10x = 1$ on a binary computer is not possible—you cannot find the correct solution because the number 1/10 does not exist in a binary computer. However, if we use interval arithmetic to solve $10x = 1$ we will come up with a narrow interval enclosure than is guaranteed to contain the correct solution.

Consider now some more difficult equation solving problems, and what the role of interval mathematics might be. One at the core of many chemical engineering problems is that of computing phase equilibrium. To do this we could solve the equifugacity equations. But these

frequently have multiple solutions, so to be sure that we have the right solution we really need to be able to find *all* the solutions. Another way to compute phase equilibrium is do a minimization of the Gibbs energy. But this may have multiple local minima, so we need a reliable way to be sure that we get the *global* minimum.

Problems like this, involving issues of the existence and uniqueness of solutions, are difficult ones, but there are some misconceptions about how difficult they really are. For example, in Dennis and Schnabel's classic book [3] it is said that "In general, the questions of existence and uniqueness—does a given problem have a solution and is it unique?—are beyond the capabilities one an expect of algorithms that solve nonlinear problems." This, however, is not entirely true, as we shall soon discuss. In a more recent textbook, Heath [4] says "It is not possible, in general, to guarantee convergence to the correct solution or to bracket the solution to produce an absolutely safe method" [for solving nonlinear equations]. Again this is not quite right.

In fact, there do exist methods, based on interval mathematics, in particular interval-Newton methods, that can, given initial bounds on the variables, enclose any and all solutions to a nonlinear equation system, or determine that there is no solution, or find the global optimum of a nonlinear function [5]. These methods provide a *mathematical* and also *computational* guarantee of reliability. The latter is important since mathematical guarantees can be lost once things are implemented in floating point arithmetic. In my group at Notre Dame, we are actively involved in developing algorithms and applications using these concepts [e.g., 6,7]. So why isn't everyone using these methods? A primary reason is that they can be significantly slower than standard local point methods. However, my feeling on this and on other issues of reliability is that we have lots of computing power, so why not use it to solve problems more reliably? The use of interval mathematics is one potential approach for doing this.

Now consider briefly another question. If we cannot be sure that we are getting the right answers, are we in danger of relying too heavily on computing power? Again we will explore the question by looking at a couple examples.

The USS Yorktown is a guided missile cruiser, and the first in the Navy to be outfitted with so-called SmartShip technology, which would allow reducing crew levels by computerizing many ship functions. (This is reminiscent of the Starship Enterprise's ill-fated encounter with Dr. Daystrom and the M-5 Multitronic computer system in "The Ultimate Computer" episode of the original Star Trek series.) In September of 1997, the Yorktown suffered a complete propulsion system failure and was dead in water for about two hours and 45 minutes. The subsequent investigation determined that "the Yorktown lost control of its propulsion system because its computers were unable to divide by the number zero." Apparently a crew member entered a zero into a field of some application program, leading to a complete crash of the system and leaving the ship dead in the water.

Now if I write a computer program, run it on the Unix workstation in my office, and it mistakenly divides by zero, about the worst that will happen is that the program will stop and I will see some message on my monitor saying "overflow error." It will not lead to a complete shut down of every computer on the Notre Dame campus network—which is the analog of what happened on the Yorktown. There is still some controversy about why this seemingly simple error could have

such severe consequences, but a popular theory attributes it to the use of the Windows NT operating system. A report from the Atlantic Technical Fleet Support Center concluded that "Using Windows NT ... on a warship is similar to hoping that luck will be in our favor."

Sleipner A is an offshore drilling platform in the North Sea. Such platforms are constructed on shore in two parts, a concrete base and the platform itself. These are then mated in a deep water area near shore (a fjord typically) and then floated out to the desired position in the North Sea. Thus the concrete base has a number of large buoyancy cells allowing it to float. The process of mating the platform to the base is the most critical part of this process. During mating, the concrete base is lowered, so that the support pillars are just under water, allowing the platform to be properly positioned over it. At this time, the buoyancy cells are deeper than they will ever be, and thus subject to the highest water pressure they will ever see. The cells must thus be designed with this in mind. On August 23, 1991 while the original concrete base for Sleipner A was being lowered for mating, it sprang a leak and sank, causing a seismic event registering 3.0 on the Richter Scale, and an economic loss of about 700 million dollars.

So what went wrong? It seems that the concrete base structure was designed using a well known and quite sophisticated finite element algorithm and code, and one that had been successfully employed before in this same type of application. There was great trust placed in this particular algorithm and code, and a sophisticated design was produced. Later investigation, using a different finite element algorithm, showed however that the algorithm used initially made a poor finite element approximation of a critical area in the cluster of cells, resulting in an underestimate of stresses by about 50% and a design in which the cell walls were too thin in critical places.

After the original base sank, the operator was faced with an economic loss of production of about a million dollars a day. And they no longer trusted the computer analysis. So what could they do to get this project moving? What they did was to make a decision "to proceed with the design using precomputer sliderule era techniques" [8]. The resulting design was not as sophisticated as the first, and reportedly somewhat more costly to build, but it *did not sink*. One of the investigative reports later concluded with a simple lesson [8], namely that "relatively simple hand calculations ... should always be done, both to check the computer results and to improve the engineers' understanding of the critical design issues." This is a point that many of us make in teaching the senior design class in which students may make extensive use of simulation packages. However, in my experience this is a point that does not take easily with students and has to be repeatedly pounded in.

These two examples suggest that, without good algorithms and software, putting too much trust in computing power may be downright dangerous. Perhaps more importantly, these examples show we must always keep in mind that, no matter how powerful the computer or sophisticated the software, results must be viewed with sound engineering judgement.

Concluding Remarks

Now that I have too long played devil's advocate, I want to conclude on a very positive note. The fact is that chemical engineers today are using high performance computing, and computing at all levels, to break computational barriers and truly expand the frontiers of process

engineering. For the chemical process industries, effective and appropriate use of computing technology has much to offer: cleaner, safer, more efficient and less costly manufacturing processes, new and better products, faster times to market, and faster responses to changes in economic, regulatory, and technological environments. This adds up to a bottom line of enhanced competitiveness in the global marketplace.

Acknowledgements

I want to give particular thanks to my early mentors, namely Skip Scriven, whom I worked with as an undergraduate at the University of Minnesota, and Dale Rudd, who was my Ph.D. advisor at the University of Wisconsin. Thanks also to all my graduate students, and others who have worked in my group. I think it is well known that it is these students who do most of the work I also want to thank the various funding agencies and others who have supported this work, both in terms of dollars and computing time. These include the National Science Foundation, the ACS Petroleum Research Fund, the Environmental Protection Agency, the Department of Energy, the Army Research Office, the Dreyfus Foundation, Sun Microsystems, Cray Research, IBM, Dow Chemical, Du Pont, Shell Oil, the University of Illinois at Urbana-Champaign, and the University of Notre Dame.

References

[1] S. E. Zitney, L. Brüll, L. Lang and R. Zeller. *AIChE Symp. Ser.* **91**(304), 313-316 (1995).

[2] S. M. Rump. In *Reliability in Computing* (R. E. Moore, ed.), pp. 109-126, Academic Press (1988).

[3] J. E. Dennis and R. B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall (1983).

[4] M. T. Heath. *Scientific Computing: An Introductory Survey*, McGraw-Hill (1997).

[5] R. B. Kearfott. *Rigorous Global Search: Continuous Problems*, Kluwer (1996).

[6] J. Z. Hua, J. F. Brennecke and M. A. Stadtherr. *Ind. Eng. Chem. Res.* **37**, 1519-1527 (1998).

[7] R. M. Maier, J. F. Brennecke and M. A. Stadtherr. *AIChE J*. **44**, 1745-1755 (1998),

[8] M. P. Collins, F. J. Vecchio, R. G. Selby, P. R. Gupta. *Concrete International* **19**(8), 28-35 (1997).

Table 1. Leading Parallel Computers in Late 1998.

	R_{max} (GFLOPS)	R_{peak} (GFLOPS)
Cray/SGI Mountain Blue (1999)		~4000
IBM Blue Pacific (5800 processors)		3880
Intel Sandia Red (9152 processors)	1338	1830
Cray T3E-1200E (1080 processors)	891.5	1296
IBM SP/604e (1900 processors)	547	1262
SGI Origin 2000/250MHz (512 processors)	195.6	256

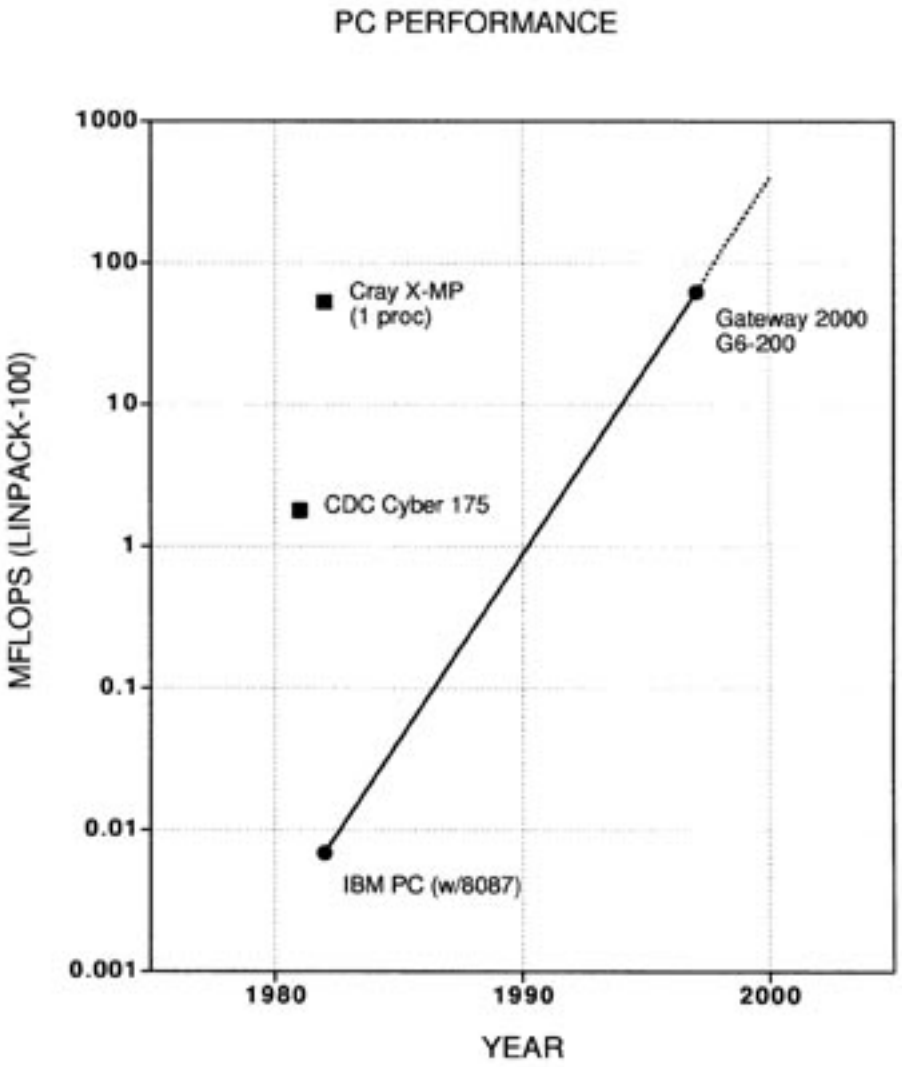

Figure 1. Growth in PC performance. See text for discussion.

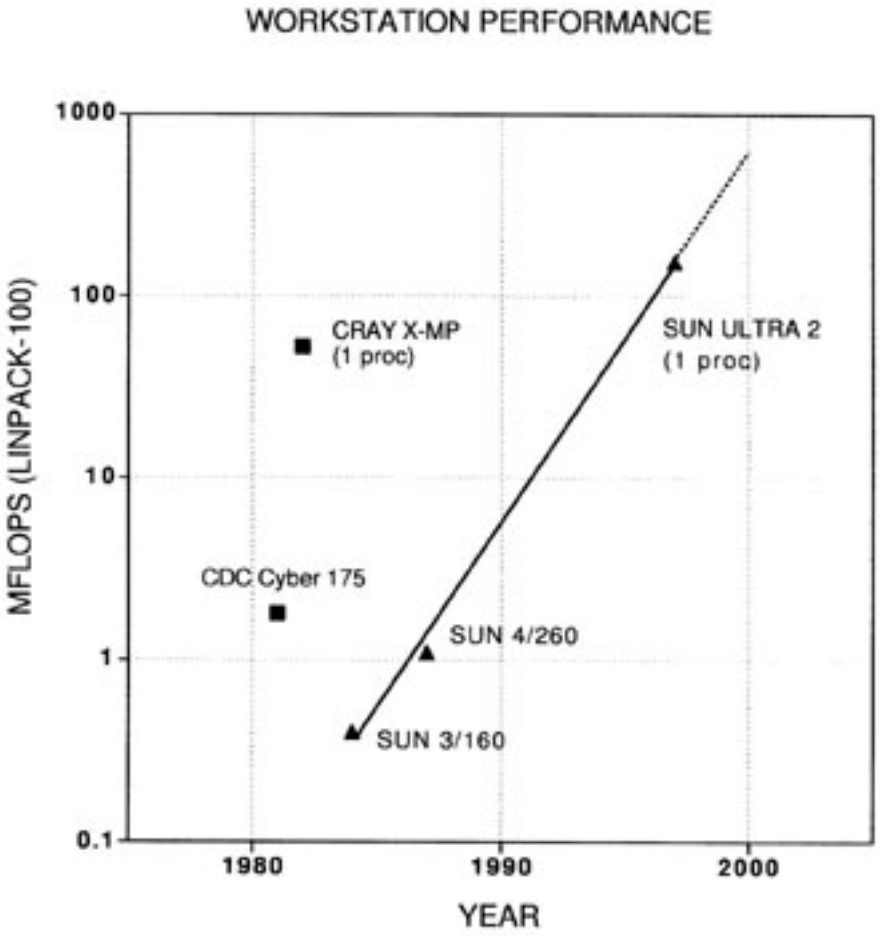

Figure 2. Growth in workstation performance. See text for discussion.

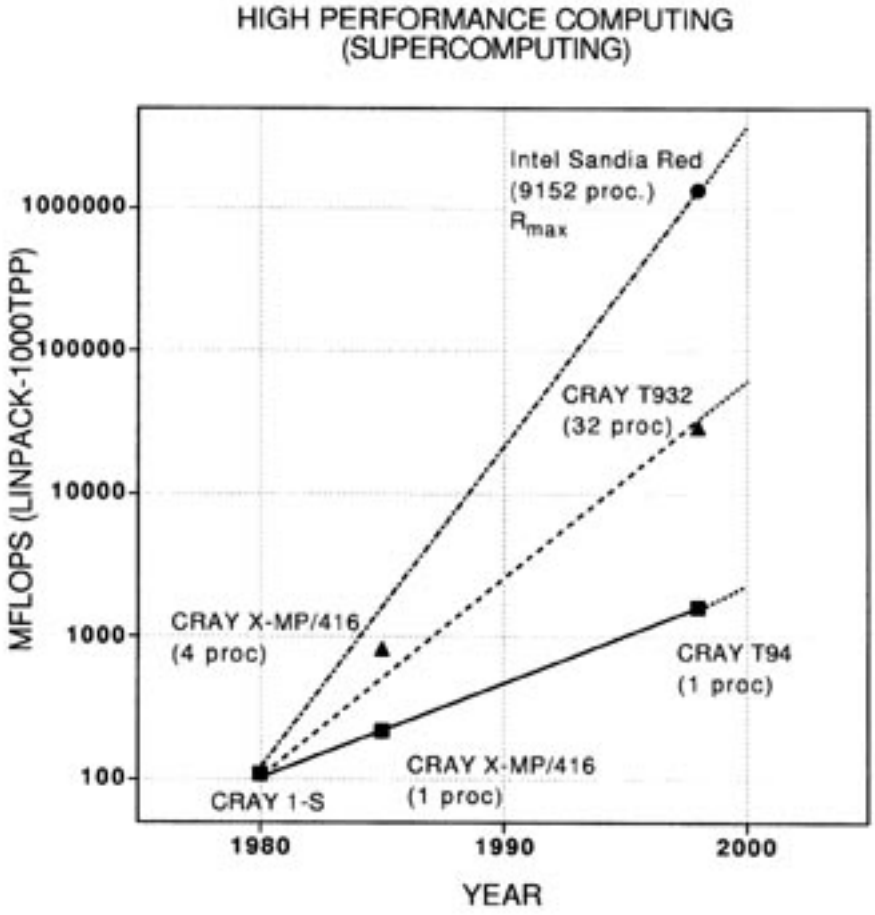

Figure 3. Growth in high performance computing. See text for discussion.

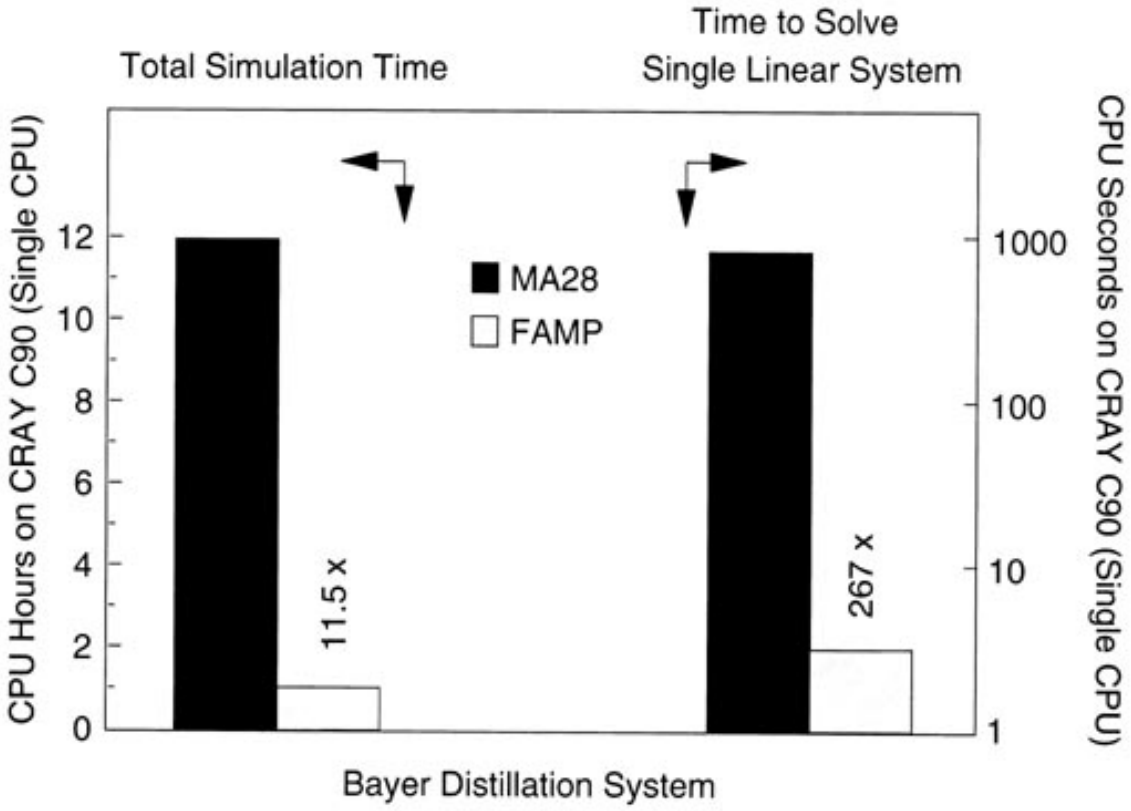

Figure 4. Comparison of sparse matrix solvers [1]. See text for discussion.