High-Performance Computing: Past, Present and Future

Anne C. Elster[1]

Norwegian University of Science and Technology(NTNU),
NO-7491 Trondheim, Norway
elster@computer.org,
WWW home page: http://www.idi.ntnu.no/ẽlster

Abstract. Although the power of "yesterday's" supercomputers is now available on the desktop, our yearning for even more computational power to solve even larger problems continues to grow. This paper covers the highlights of some of the author's High Performance Computing (HPC) experiences dating back one of the first commercial supercomputers – Intel IPSC hypercube in the mid-80's – through today where her current HPC work focuses on cluster computing using MPI.
The author will also give some of her prediction on where she thinks the HPC field is heading in the future. Her "Ant Theory" (even sugar ants display more complex planning and data processing than any human-built system of today) tells us we have a long way to go. Through newer technologies such as nanotubes and Bose-Einstein condensates, our future should even prove Moore's law wrong, and let the HPC field tackle even bigger and more complex problems.

1 Introduction

Bell and Gray recently wrote an article [1] on what they think is next in HPC that includes a nice overview of the early days of HPC including pioneering work by the authors, Cray and others. My high-performance computing (HPC) experiences date back to when I joined Cornell University in New York State, USA, about the time they established their super computing center known as "The Theory Center" in the mid 1980's.

Although as an undergraduate I had envisioned focusing on computer architecture, I was a good match for parallel computing since back then it was especially useful to know a lot about hardware when developing parallel programs since the systems were so crude. I also took several computer science and mathematics classes in addition to my standard undergraduate curriculum in computer engineering. These classes included linear algebra with APL programming and writing a program simulating an HDLC layer protocol (acks and nacks) using Concurrent Pascal as part of one of my computer network classes, both which proved very useful when I later got into parallel programming.

My first task in graduate school, outside classes, was a Master's project [7] that involved porting a Parallel Pascal translator from a VAX 780 to an IBM

RX where I had to partition the code over several memory segments in order for it to fit. In retrospect, I think of this as my first experience with caching, only that it was on a slightly different scale.

The next section describes some of my graduate work on hypercubes. I also extended the work on hypercube algorithms in collaboration with Hungwen Li of IBM looking at a toroidal networks of processors [3].

My graduate work on HPC systems also included multitasking across several IBM3090s at IBM Yorktown Heights, [8] and as part of my dissertation work, writing particle simulations for Kendall Square Research (KSR) machines in the early 90's [9]. Some of the highlights from my PhD work will be covered in Sections 3-5.

Section 6 discusses current work related to cluster computing, MPI and numerical libraries, PETSc in particular. I discuss grid computing in Section 7 and Sections 8 and 9 includes several of my predictions for the future computing regarding where I thinks the HPC field is heading. Finally, I end with my "Ant Theory" of Section 10 that tells us we have a long way to go.

2 Hypercubes and PBLAS

The following summer (1986), I got to work on the first IPSC (Intel Personal Super Computer) hypercube outside the US at Christian Michelsen's Institute (CMI) in Bergen, Norway, where my first task was to write a matrix-vector multiplication routine for their computational library aimed at petrochemical related codes. [2]

The Intel hypercube was a distributed memory machine that had full hypercube connectivity (i.e. the number of connections per node increases logarithmically with the number of processors.) This was done so that communication costs could be minimized, but of course its physical wiring limitations for systems with more than a few hundred processors. My computer science training made me quickly realize that that matrix-vector multiplication could be done on a hypercube very efficiently with respect to communication by mapping orthogonal tree structures on to the hypercube links using Gray codes [2]. Good hardware and system knowledge was also important since our debugging tools were primitive (read:print statements and watching processor LED lights form tree patterns).

I also did some work jointly with Uyar and Reeves on how to redistribute data on hypercube networks when processor(s) fail and its impact on algorithm efficiency [4].

I then did a lot of work on parallelizing Basic Linear Algebra Subroutines (BLAS) for the hypercube with an aim towards a standard subroutine interface calling sequence for both the parallelized BLAS and their communication routines [5]. Like many of my colleagues, I was particularly careful to reduce communication overhead by using the extra communication links that hypercubes provided.

These libraries were later forwarded to Intel and also used in a parallelization effort of a charge transport simulation code jointly with Xerox [6]

Communication overhead was easily traced on early hypercube models since it did not overlap at all with computations. This, of course, changed when separate communication processors were introduced on later models, and one then had to see if one could "eliminate" communication costs by doing alternate computations as one waited for data from other processors. In practice, this was very hard to achieve since communication times were orders of magnitude larger than computations speed.

Unfortunately, this is even more true today both for distributed systems and with respect to accessing different levels of the memory hierarchy on single processor or shared memory systems.

3 Modularizing Large Codes

My dissertation work [9] involved modularizing and implementing a functional parallelized particle code. Particle simulations are fundamental in many areas of applied research, including astrophysics, plasma physics, semiconductor device physics, and xerography. These simulations often involve tracking of charged particles in electric and magnetic fields.

At the time of my dissertation, these simulations had, due to their high demand for computer resources (especially memory and CPU power), been limited to investigating local effects – typically using up to an order of 1 million particles. By developing novel algorithmic techniques to take advantage of modern parallel machines such as the Kendall Square Research (KSR) machine and employ state-of-the-art particle simulation methods, my PhD work targeted simulations with 10-100 million particles in order to study interesting larger-scale physical phenomena. Other numerical codes that target parallel computers would also benefit from many of the techniques developed in this work.

Our application fell in the category of collisionless systems, where each simulation particle, often referred to as a "superparticle", represents millions of physical electrons or ions in a collisionless plasma. The numerical techniques used usually involve assigning charges to simulated particles, solving the associated field equations with respect to simulated mesh points, applying the field solution to the grid, and solving the related equations of motion for the particles. Codes based on these numerical techniques are frequently referred to particle-in-cell (PIC) codes.

We used a 2-D FFT-solver (assuming periodic boundaries) as the field solver, but also considered other techniques, including finite difference (e.g. SOR). A leapfrog particle pusher was used to advance the particle positions and velocities. However, our choices of particular numerical methods were not the focus of my dissertation work; instead, we concentrated on general approaches of how to parallelize these methods. We also paid close attention, from the viewpoints of memory usage and speed, to **the manner in which these parallelized methods interact with other sections of the code.**

The primary goal was hence to investigate how best to incorporate parallel methods to invoke numerical algorithms with an eye towards maintaining their applicability to more sophisticated PIC methods to be developed in the future.

Our test code was implemented in C using Pthreads on the KSR1. Optimizations were guided by the methods dictated by our analytical and experimental results. The dissertation included several graphs depicting the performance of codes using replicated grids (fixed particle partitioning), a partitioned grid (particles move among processors), and adaptive methods. These results were then compared with our analyses.

3.1 Threads vs. Processes

One of the nice things about the KSR and other parallel systems in the 1990's was that they started to use threads (light-weight processes) rather than fork regular (heavy-weight) processes. This meant a lot of operating system (OS) kernel overhead was now avoided in parallel programs. A further discussion of thread vs. processes is discussed in the intro of my PhD thesis [9].

4 Testing Large Codes

In addition to the standard test one would do for each numerical module as a computer science/math person, I took the testing of my PIC code one step further by plugging in real physics constants and running the code with initial conditions leading to known phenomena.

One such test involved the plasma frequency which is are observed in real plasmas. To get a feel for where these oscillation stem from, one can take a look at what happens to two infinite vertical planes of charges of the same polarity when they are parallel to each other and occur repeatedly as in periodic systems.

In order to verify that our code actually would behave according to the physical laws, an in-depth analysis of one general time step was used to predict the general behavior of the code. The analytic derivations provided in Appendix B of the thesis proved that our numerical approach indeed produces the predicted plasma frequency which in turn was shown in our simulation results.

A detailed description of the physics behind plasma frequencies and how we set up our tests were described in the thesis [9].

4.1 Two-stream Instability Test

To further test whether our codes were able to simulate physical systems, we also performed a two-stream instability test [10]. In this case, two set of uniformly distributed particles (in our case 2D particle grids) are loaded with opposite initial drift velocities. Detailed knowledge of the non-linear behavior associated with such simulations was developed in the '60s.

Systems that simulate opposing streams are unstable since when two streams move through each other one wavelength in one cycle of the plasma frequency,

the density perturbation (bunching) of one stream is reinforced by the forces due to bunching of particles in the other stream, and *vice versa*. The perturbations hence grow exponentially in time. In order for this test to work, care must be taken in choosing the initial conditions. In our case, we chose to test each dimension separately using an initial drift velocity $v_{drift} = \omega_p * L_x$ for half the particles and $v_{drift} = -\omega_p * L_x$ for the other half of the particles.

Our code was able to capture the characteristic non-linear "eyes" associated with two-stream instabilities. Our time-step was set to $1.5 \cdot 10^{-7}$. The "eyes" appeared within 10 time-steps of the large wave appearing. Notice that these are distance *versus* velocity plots showing 1D effects. We hence also obtained similar plots for a corresponding test of x and v_x to check the second dimension.

5 Cell Caching

Many scientific codes access data grid cell (squares, cubes, etc) or stencils/templates rather than by vectors. However, these objects are typically stored and accessed either column or row-wise (or block-column or block-row-wise) using the standard array indexing features provided by the programming language.

My dissertation work showed that for systems with memory hierarchies, especially systems with caches, these storage schemes are not necessarily the best for these kinds of codes.

For instance, in 2-D PIC (Particle-in-Cell) Codes when the particles' charge contributions are distributed to the grid points (scatter) or field contributions from the grid points are collected at each particle (scatter), each particle will be accessing the four grid points of its cell (assuming a quadrangular grid). If the local grid size exceeds a cache line (which they invariably do since cache lines tend to be very small compared to system grids), and the traditional column or row oriented storage of the grid is used, each particle will need at least two cache lines to access the four grid points it is contributing to.

Since typically a significant overhead is paid for each cache hit, we proposed that one instead stores the grid hierarchically according to a cell caching scheme. This means that instead of storing the grid row or column-wise during the particle phase, one should store the grid points in a 1-D array according to small sub-grids that would fit into one cache-line. For 3-D codes, sub-cubes should be accommodated assuming the caches are large enough. In other words, the cache use should try to reflect the access pattern these codes use.

However, even on systems where the cache line is only 128bytes, and hence can hold a maximum of 16 64-bit floating point numbers, which means each cache-line can accommodate 8X2 or 4x4 sub-grids, one can still get significant improvements with respect to cash hits through cell cashing. Assuming one used the 4x4 sub-grids (which would minimize the border effects) the traditional column/row storage approaches would use 2 x 16 cache-hits whereas cell-caching would use [(3x3) * 1] + [(3+3)*2] + 4 = 25 cache-hits, an improvement of more than 25%!

To access array element A(i,j), when the cache size is CXC, we need:

$$A(i,j) = A[((i/C) * (C^2 * C^2)) + ((j/C) * C^2) + (jmodC) + (imodC) * C].$$

Although these index calculations require more operations than the typical $A(i,j) = A[i * N_x + j]$, most of these operations can be performed by simple shifts when C is a power of 2. Compared to grabbing another cache-line which accesses a different level of the memory hierarchy, this is still a negligible cost.

Notice that this cell-caching technique can be applied recursively to systems with a memory hierarchy. For larger caches or intermediate memory this technique may also extended to any finite dimension. Therefore, both parallel and "serial" cell codes for any system with a hierarchical memory would benefit from using this alternative block storage scheme.

6 Modeling Codes with Memory Hierarchies

When moving from sequential computer systems to high-performance workstations with caches (hierarchical memory) and parallel systems with layered distributed memory, data locality becomes a major issue for most application. Since intermediate results and data need to be shared among the processing elements, care must be taken so that this process does not take an inordinate amount of extra run-time and memory space. This is as true today as it was in the early 1990's.

In distributed systems, communication overhead is typically modeled as described by Hockney and Jessup [12]: $t_{comm} = alpha + N * beta$, where t_{comm} is the communication time for an N-vector. Here alpha is the start-up time and beta a parameter describing the bandwidth on the system.

In my dissertation, [9] this model was extended for hierarchical memory systems and systems with local as well as global memory so that one can better understand the impact of the layered memory and thereby take advantage of the combined speed and memory size offered by these systems.

For instance, on hierarchical memory systems, t_{comm} will be a function of which levels of the memory hierarchy are accessed for the requested data. Consider a model with only two hierarchical parameters, t_{lmem}, which denotes time associated with *local* memory accesses and t_{gmem}, time associated with *global* memory, then:

$$t_{comm} = t_{lmem}(N) + t_{gmem}(M).$$

Note that t_{lmem} here covers both items within a cache-line and items within local memory. For a vector of N local data elements, $t_{lmem}(N)$ is hence not a simple constant, but rather a function of whether the individual data elements are 1) within a cache-line, 2) within cache-lines in cache and 3) in local memory. Similarly, t_{gmem} is a function of whether a vector of M data elements accessed are all within 1) a communication packet, 2) in some distant local memory, and/or 3) on some external storage device such as disk.

We then used this model to determine the performance complexity of three 2-D Particle-in-Cell (PIC) algorithms:

1. a serial algorithm with cache,
2. a parallel implementation using particle partitioning with replicated grids and global sums, and
3. a parallel implementation using grids partition with automatic partial particle sorting.

Finally, we verified these theoretical results by comparing them to timing results obtained on a distributed shared memory machine.

Our results showed that given that one is limited in how much fast local cache memory one can have per processor, the serial code not only suffers from having only one processor, but also suffers when the data sets get large and no longer fit in cache. We also showed that the replicated grid approach would suffer from the grid-sum computation and communication overhead for requiring large grids on a large number of processors. We also showed that the grid partitioning approach would clearly be hampered for load-imbalanced systems when max(local no. of particles) >> No. of particles/No. of processors for a good portion of the time-steps.

Ways to compensate for this imbalance by re-partitioning the grid as well as how to apply this model to other large scientific codes were also discussed.

7 Cluster Computing, MPI and Numerical Libraries

This recent trend in HPC to use a network of workstations and/or PCs rather than traditional supercomputers is gaining momentum. Although the network speed is still slow compared to available processing speed, and implementing distributed memory programs is generally harder than shared-memory programs, the price and flexibility of such systems should not be underestimated. Standards like MPI and others also make these systems worth a serious look in the coming years.

This view was also recently echoed by Bell and Gray [1] in their Feb. 2002 article entitled "What's Next in High-Performance Computing".

7.1 Distributed Memory versus Shared Memory

The primary problem facing distributed memory systems is maintaining data locality and the overhead associated with it. This problem with parallel overhead also extends to the shared memory setting where data locality with respect to cache, is important. In my dissertation, I proposed that one view the KSR as a shared memory system where all memory is treated as a cache (or hierarchy thereof).

Shaw [13] pointed out that his experience with the SPARC-10s showed they had an interesting property which seemed highly relevant. In order to achieve their peak speed (17-19 MFLOPs), the data had to be be in what Sun called the SuperCache which was about 0.5 MBytes per processor. This implies that if you are going to partition a problem across a group of SPARC-10s, or one of today's PC clusters, you have many levels of memory access to worry about:

1. machine access on a network
2. virtual memory access on one machine on a network
3. real memory access on one machine on a network
4. cache access on one processor on one machine on a network

Hence, there is a great deal to worry about in getting a problem to work "right" on clusters.

A network of workstations consequently raises a lot of issues similar to that of the KSR in that they both possess several levels of cache/memory. To achieve optimum performance on any given parallel system, no doubt, a lot of fine-tuning is necessary.

7.2 MPI

My HPC experiences also include representing Cornell and Schlumberger on the MPI standards committee in the 1990's [11] [14]. Although not initially intended for threads, great efforts were made by the Committee to make the standard both thread safe and useful as a building block for library builders.

Unlike High-Performance Fortran (HPF), MPI has since evolved to become a favored standard for both cluster computing and shared memory machines (through OpenMPI). Although initially intended as a standard for message passing calls on commercial supercomputer systems, it now is often used on clusters of workstations. Indeed, my own work is currently focusing on cluster computing using MPI. One such effort is my work with Sack on speeding up MPI broadcast operations through multicasting which is described in another paper in this volume [17]. I've also used MPI on clusters successfully in several class settings when teaching students parallel programming.

7.3 PETSc

This section describes how PETSc, a versatile MPI-based software package from Argonne National Laboratory, is being effectively used to develop, implement and test several new iterative solvers. PETSc (Portable, Extensible Toolkit for Scientific Computation) [15] provides several modules for solving partial differential equations (PDEs) and related problems on high-performance computers.

PETSc is a large and complex package that is known to scare off several users for its apparent complexity and steep learning curve. However, it comes with an extensive User Manual [15] that is very well written once one "digs" into it. The support scripts for both the serial and parallel installations are fairly easy to modify and work well.

PETSc consists of a suite of data structures and routines that provide the building blocks for the implementation of large-scale application codes on both serial and parallel computers. PETSc is still under development and currently includes several parallel and non-linear equation solvers, unconstrained minimization modules, and lots of support routines. It provides many of the mechanisms needed within parallel application codes including parallel matrix and

vector assembly routines. Although PETSc also support Fortran, the designer as well as this author recommends that one codes ones routines in C or C++ for maximum flexibility and portability on high-end parallel systems.

By developing and testing new iterative methods using PETSc one can leverage the work put into these packages as well as easily compare ones routines to the Krylov subspace methods already provided by PETSc such as GMRES, Conjugate Gradient, CGS, Bi-CG-Stab, TFQMR, Richardson and Chebyshev. PETSc also provides several popular preconditoners including Additive Schwartz, Block-Jacobi, ILU, ICC, and LU (sequential only).

One of the new solvers we developed that was not previously available in PETSc was the Complex Chebyshev Acceleration method. This method is used when solving large linear system where the corresponding system matrices are non-symmetric and the eigenvalues can be found within an ellipse in the complex plane. The Chebyshev Acceleration procedure is a special case of polynomial extrapolation.

A non-optimized complex Chebyshev code using PETSc was developed with my help by one of my students in a graduate class in 1-2 weekend! Our codes are still under development, so more refined results will be presented in subsequent publications. Our related presentation at SIAM PP'01 [16] also highlights some of this work.

I hope that the reader gets inspired by this subsection to use PETSc as a tool to get started when developing their own implementations of research and production code for parallel PDEs, or other codes related to PETSc.

For a list of several current projects, products, applications and tools for scientific computing, check out the Webpage maintained by the PES (Problem Solving Environment) group at Purdue:

`http://www.cs.purdue.edu/research/cse/pses/research.html`

8 Heterogeneous Clusters and Grid Computing

As clusters become heterogeneous networks of a variety of PCs, high-end workstations and more traditional supercomputer systems, as well as special-purpose systems such as graphics and visualization caves, our challenges for interoperability continue to increase.

There is no doubt in my mind that the computational grid where several of such local area networks extends across the high-end backbones of the Internet will provide a lot of challenges and opportunities for HPC students and researchers. Some are already mentioned by Bell and Gray [1], others will be discussed throughout this volume and at other related conferences.

9 Future Challenges and Processor Technologies

As can be seen from the previous sections HPC has come a long way from its roots where monolithic computer systems sitting in a huge room and/or building

were used for physical simulations. The Grid and related technologies, including wireless networks, are changing the way we think about computing.

9.1 Some "Grander" Challenges

No doubt, we will continue to refine our physical simulations of weather systems and other "Grad Challenge problems", however I predict we will also see emerging problems we have yet not fully defined.

Like railway systems linked with GPS, future HPC system may be involved in controlling cars without input from drivers after input of desired destination. The challenges here are enormous as are the potential benefits considering how many accidents could be avoided.

Other areas that are already receiving increased attention are the use of the ultimate distributed processors: the Smart Cards and wireless network nodes. One idea that has been proposed is to store biometric information such as fingerprints on the cards which can then be used as a password to a wireless net, or for other real-world identification purposes.. This case is a great example were instead of accessing and comparing information from a huge database, one can instead provide the clients with enough processing power and storage to retrieve the data needed.

Security in itself also provides lots of challenges related to HPC and the Grid, since the more we protect things, the harder it will become to make them inter-operate.

9.2 The Wal-Mart Effect on COTS

Commercial forces such as Wal-Mart – a U.S. retailer that is bigger than Sears, Kmart and J.C. Penney combined, according to the "Fortune 500" – posess a control of the IT industry that should not be underestimated. Their $4 billion IT expenditure is predicted to influence $40 billion IT investments of suppliers [22] and has therefore impacted the IT market more than Microsoft and Cisco could ever hope to at this point. Wal-mart is, however, not driven by the latest and greatest technology, but rather by an improved business model. This is, unfortunately, not good news for HPC.

On the other hand we may see an interesting trend in high-end COTS processors. Assuming the Wal-marts of the world continue to embrace high-end gaming and entertainment systems, we may very well see that the processor market going from PCs to Video-game stations where Playstation chips may actually out-power Pentiums. In this scenario, I envision the Grid no longer as wired high-end PC's, but rather mostly networked video Playstation processors, possibly wirelessly connected.

9.3 Processing Break-Throughs

I also predict that we will continue to out-shine Moore's Law. Going from today's Pentium 4 chips with 42 million transistors using 130nm technology to

using emerging technologies such as **nanotubes** [20] made from strands of single atoms, each a few nanometers, to make chips crammed with billions of transistors, will no doubt continue to impact HPC.

Even further out we may see applications of Bose-Einstein condensates in quantum computing and nanotechnology. Bose-Einstein condensation involves cooling atoms to near absolute zero (0 Kelvin). The first success in this area was done by 2001 physics Nobel laurates Weiman and Cornell who in 1995 cooled rubidium atoms to about 50 nanokelvin (50 billionth of a degree above zero Kelvin). A few months later 3rd winner Ketterle succeeded similarly with a larger condensate of sodium atoms. [21]

10 The Ant Theory

Physicists thought at the turn of the 20th century that they had "solved physics", but then came quantum theory. Today's physicists no longer dare to make such finite sounding claims, however, many scientists are starting to think that solid state will be reaching its physical limits in the near future and hence bar further advances in computing.

However, to those who think that getting down to the quantum level will stop us from making any further progress in computing, I offer my "Ant Theory":

It stems from when I as a little girl, no more than 5 or 6 years old, used to sit in the grass and study the little critters making their way through the world of grass blades and debris. I was perpetually amazed at how well such tiny little creatures could navigate so well through a terrain that seems incredibly rough and large for their size.

Recently, I had the same exact thoughts studying some sugar ants on my bathroom floor navigating between water drops and other obstacles sometimes pulling object larger than themselves back to their hills: How can such a tiny little creature see and navigate so well processing all that information in a system so minute?

The answer to that question probably lies somewhere in biology and chemistry, and until we understand the connection between those fields and physics that enables such tiny organic creatures to perform so well, we have not come very far. However, I believe we are still a long way off from understanding what really makes organic life tick.

Moore's and other laws will continue to get broken.

References

1. Bell, G., Gray, J.: "What's Next in High-Performance Computing?" in *Communications of the ACM*, **45**, No. 2 (2002) 91–95.
2. Elster, A.C. and Reeves, A. P.: "Block-Matrix Operations Using Orthogonal Trees", Proc. of the Third Conf. on Hypercube Systems and Applications, January 19-20, 1988 in Pasadena, CA, Ed. G. Fox, ACM, pp 1554-1561.

3. Elster, A.C. and Li, Hungwen: "Hypercube Algorithms on the Polymorphic Torus", Proc. of the Fourth Conference on Hypercube, Concurrent Computers, and Applications, March 6-8, 1989 in Monterey, CA, **Vol. I**, Golden Gate Enterprises, pp 309-316.
   This paper was based on Cornell Computer Science TR 89-1003 and IBM Research Report, RJ 6775, 1989 (same title and authors).
4. Elster, A.C., Uyar, M.U., and Reeves, A.P.: "Fault Tolerant Matrix Operations on Hypercube Computers", Proc. of the 1989 International Conf. on Parallel Processing, St. Charles, IL, August 8-12, 1989, Ed. F. Ris and P. M. Kogge, Penn State, Vol. III, pp 169-176.
5. Elster, A.C:, "Basic Matrix Subprograms for Distributed Memory Systems", Proc. of the Fifth Distributed Memory Computing Conf. (DMCC5), in Charleston, SC, April 9-12, 1990, Ed. D. W. Walker and Q. Stout, IEEE Computer Society Press, pp 311-316. Received Student Paper Competition Award.
6. Elster, A.C. and Ramesh, P.S.: "Simulation of Charge Transport Using Parallel BLAS on the Intel Hypercube", Xerox Internal Report X9200084, Webster Research Center, NY, April 1992.
7. Elster, A. C.: "Porting of the Parallel Pascal Translator from VAX 11/780 to IBM-PC/AT". Master of Engineering Project Report, School of Electrical Engineering, Cornell University, Fall 1986.
8. Elster, A.C.: "Efficient Parallel Algorithms for Matrix Operations", Master of Science Thesis, Cornell University, Aug. 1988.
9. Elster, A.C.: "Parallelization Issues and Particle-in-Cell Codes", Ph.D. dissertation, Cornell University, August 1994. Abstract at:
   <http://www.englib.cornell.edu/thesesabstracts/August94/elster.html>
10. Birdsall, C.K. and Langdon, A.B.: *Plasma Physics via Computer Simulations*, Adam Hilger, Philadelphia, 1991.
11. Elster, A.C. and Presberg, David L.: "Setting Standards For Parallel Computing: The High Performance Fortran and Message Passing Interface Efforts", Theory Center SMART NODE Newsletter, May, 1993, **Vol.5**, No. 3, Cornell University.
12. Hockney, R.W. and Jesshope, C.R.: *Parallel Computers*, Adam Hilger Ltd., Bristol, 1981.
13. Shaw, J.G.: Personal communications.
14. The Message Passing Interface Standard (MPI):
   <http://www-unix.mcs.anl.gov/mpi/index.html>
15. PETSc (Portable, Extensible Toolkit for Scientific Computation):
   <http://www-fp.mcs.anl.gov/petsc/>
16. Elster, A.C. and Liang, C.: "Developing and Testing Linear Solvers Using PETSc", First SIAM Conference on Computational Science and Engineering, Washington, D.C., Sep 21-24, 2000. <http://www.siam.org/meetings/cse00/cp25.htm>
17. Sack, P. and Elster, A.C: "Fast MPI Broadcasts with Reliable Multicasting", PARA'02, in this volume of Springer Verlag's Lecture Notes on Computer Sciences, Fagerholm J. et al. (Eds), 2002.
18. "Top 500 Supercomputers" – Website maintained by Univ. of Mannheim, Germany and Univ. of Tennessee at Knoxville, U.S.A.. Yearly updates presented at the Supercomputing conferences. <http://www.top500.org>
19. Slides summarizing trend of Top 500 Supercomputers 1993-2001:
   <http://www.top500.org/slides/2001/11/>
20. Rotman, D: "The Nanotube Computer", *Technology Review*, **105**, No. 2, 36–45
21. "2001 Nobel Prizes", *Technology Review*, **105**, No. 2 (March 2002) 14–16
22. *Technology Review*, **105**, No. 2. <www.technologyreview.com>