Proc. of 37th IEEE International Computer Conference, COMPCON '92, San Francisco, CA, February 1992

Is SIMD Enough for Scientific and Engineering Applications on Massively Parallel Computers ?

Steve Plimpton, Sudip Dosanjh, and Randy Krall*

Sandia National Laboratories, Albuquerque, NM 87185

Abstract

Some basic issues involved in matching an application to a distributed memory parallel machine are addressed in this paper. In particular, data communication and processor synchronization as they relate to MIMD (multiple instruction - multiple data) and SIMD (single instruction - multiple data) architectures are discussed. To illustrate the differences, we describe the implementation and performance of several engineering and scientific applications that have been coded for both kinds of machines. We find that many problems are well suited to both architectures. However, when the natural parallelism in a simulation requires a loose synchronization between processors, the MIMD paradigm offers a greater programming flexibility than SIMD.

1: Introduction

The current generation of massively parallel (MP) computers are very competitive with (and sometimes outperform) traditional vector supercomputers on many engineering and scientific problems. They provide Gbytes of memory and, for many applications, Gflops of computational power. Further, as has been widely discussed [1,2], they are a good match for many scientific applications because the parallel operation of the processors mimics the way nature operates simultaneously on all elements of a physical domain. MP computers still must mature to the point they offer the same flexible, high-speed I/O and robust, general purpose computing environment that users of vector supercomputers have come to expect. Then they will be the machines of choice for a wide range of problems.

To date, the fastest MP computers (e.g. Connection Machine, Touchstone Gamma and Delta, nCUBE 2) have been designed with distributed memory with a local address space, as opposed to a shared memory with a global address space. They have been of one of two types: MIMD - multiple instruction/multiple data or SIMD - single instruction/multiple data. (The recent announcement of the CM-5 serves to blur the distinction somewhat; we discuss it briefly in the conclusion.) The features of the two competing architectures are well-known; in this paper we relate some practical experiences with coding applications for one or both kinds of machines. In particular we address the question of deciding if MIMD or SIMD is a better choice for a particular application and the trade-offs involved in implementing a model on one architecture versus the other. At Sandia we face this issue frequently since we have a 16K-processor (512 floating point processor) CM-2, a 64-processor Intel Touchstone Gamma, and a 1024-processor nCUBE 2. In practice we code for the nCUBE and Intel almost identically. Thus our comments regarding MIMD programming issues apply to both machines as well as the Touchstone Delta (at CalTech), to which we have ported some of the codes discussed here.

For many applications, including PDE simulations on regular grids, dense matrix solvers, iterative sparse matrix techniques (such as those arising from finite difference or finite element models), and computational fluid dynamics simulations, we have seen good performance on both our MIMD and SIMD machines. However, for some applications, we have found the MIMD paradigm is a more natural and robust choice for implementing the physical model on the computer. In this paper we focus on those applications and their common attributes. Our purpose is to emphasize the need for there to be a MIMD programming capability in any and all parallel supercomputers.

In the next section we develop some simple "rules-of-thumb" that have been helpful to us in matching applications to machines. In the remainder of the paper we illustrate our points by briefly discussing several applications where the issue of MIMD vs. SIMD was considered or directly tested by dual implementation. Applications include simulations of charged particle beams for accelerator design, image processing for synthetic aperture radars, shock physics for high speed collisions, molecular dynamics for material modeling, and electron trajectories for electron microscopy studies. For each case we highlight the physics of the simulation, identify the parallel nature of the computation and its partitioning to different processors, and discuss the MIMD vs. SIMD trade-offs for implementing the mod-

* Randy is an employee of Thinking Machines Corporation, working at Sandia. The work described here was performed at Sandia National Laboratories which is operated for the Department of Energy under contract number DE-AC04-76DP00789.

el. Where appropriate we also give performance numbers for actual implementations on our SIMD and MIMD machines.

2: MIMD vs. SIMD

The first step in implementing a simulation on a parallel machine is to clearly identify where the parallelism is in the computation. This requires a sound understanding of each operation in the computational algorithm, how the data structures are modified by each operation, and how data flows from one operation to the next. This understanding will indicate how data should be distributed to the processors, what computations they can do simultaneously, and what communication of data between processors will be necessary. For either MIMD or SIMD machines, data structures that keep needed data local to processors should be used in order to minimize communication.

At this point, we find it useful conceptually to parameterize the simulation in terms of two metrics - the frequency of data communication and the level of processor synchronization. Each metric can vary from low to high. By "frequency of communication" we mean how finely interspersed the communication is with the computational statements. A section of code requiring messages (data) to pass between processors in every line of Fortran code would have frequent (high) communication. Conversely, an algorithm where whole subroutines could execute locally on a processor before data needed to be exchanged would have infrequent (low) communication. The "level of synchronization" metric refers to the execution of program code on each processor. A code where each and every statement could execute simultaneously on all the processors is tightly (high) synchronized. Conversely, a loosely (low) synchronized code is one with lots of branching or where the program flow resulting from a processor's data might dictate different subroutines executing at the same time on different processors. We note that in an actual application, the measure of either communication or synchronization will likely fall somewhere in between the extreme cases given here.

We now assign the synchronization metric to the x-axis and the communication metric to the y-axis, as shown in Figure 1. By knowing the communication and synchronization implicit in a particular code, the application can be plotted as a (fuzzy) point in the 2-d plane. The point will fall in one of the four quadrants (I-IV).

Two examples serve to illustrate these ideas. For example, an image processing code that does simple arithmetic identically on individual pixels and frequently requires neighboring pixel values to do the operations, is a code in quadrant I. On the other hand, an explicit finite element code using an irregular mesh where different physics is done in different cells with communication of neighboring cell information only needed once a timestep, is a code in quadrant III.

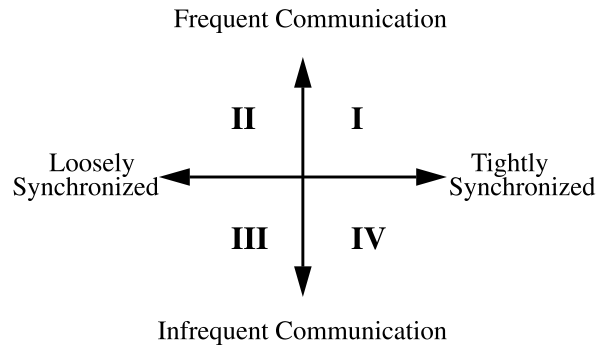

Figure 1: Partitioning the "universe" of parallel applications into regions of a 2-d plane parameterized by communication frequency and processor synchonization.

The utility of this partitioning is in the following observations:

(1) SIMD machines (and vector machines) perform well on applications in quadrants I and IV.

(2) MIMD machines perform well on applications in quadrants III and IV. Note that this implies a MIMD machine can execute a code essentially synchronously and perform well, so long as the message passing can still be done infrequently. It also implies that the best performance of a MP machine relative to a conventional supercomputer like the CRAY will be on applications in quadrant III.

(3) Both MIMD and SIMD architectures can have trouble with applications in quadrant II - particle-in-cell (PIC) codes, for example, where large amounts of data must be passed between two different data sets (field and particle) several times each timestep.

(4) The more realism one adds to a simulation in terms of special cases, boundary conditions, geometry complexity, etc. the less synchronized (more MIMD-like) it becomes.

With these observations, we hasten to add a couple of caveats. First, we do not imply that the density of interesting codes in this artificial 2-d universe is uniform. Since the CM-2 performs well on so many different applications, it may well be that most physical processes can be expressed synchronously and lie on the right side of the plane in Figure 1. Second, and more importantly, one simulation can be expressed by different algorithms and end up in different quadrants. A classic example is finite difference stencil operations on a regular 2-d mesh as in Figure 2. A SIMD implementation would put one mesh cell per (virtual) processor and achieve high efficiency by shifting data synchronously in different directions on the mesh. Conversely, an efficient MIMD implementation would be to partition the mesh into non-overlapping blocks of mesh cells, one block per processor. Each processor could then compute on interior points independently and only need to exchange block boundary information with its neighbors occasionally.

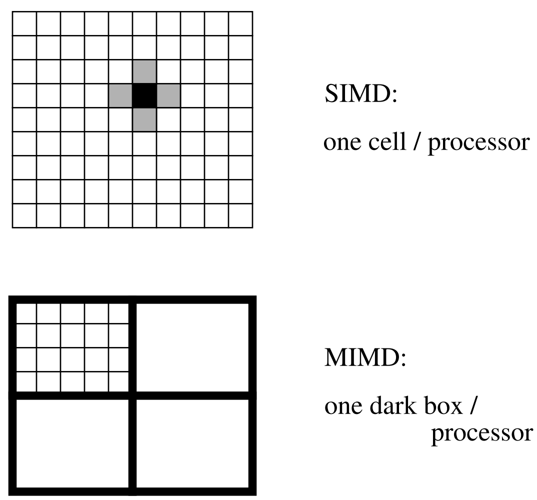

Figure 2: Typical data layout of a 2-d finite difference mesh on SIMD and MIMD machines. A finite difference stencil is shown shaded in the SIMD diagram.

In our experience, the chief drawback of the MIMD machines is the cost one pays to send a message, particularly a small one (message start-up time). This is why the MIMD machines are at a disadvantage in quadrants I and II (frequent message passing). However, in many cases this can be worked around by special coding and data layout so that messages are buffered and sent less frequently. This amortizes the cost of message start-up. On the other hand, SIMD machines by definition execute code in a tightly synchronized fashion (quadrants I and IV). No amount of clever coding will work around this limitation if a simulation requires a large amount of asynchronous operations. Of course, the price one pays for the additional flexibility of the MIMD paradigm is more effort for the programmer. The CM is well-known for its ease of programming primarily because the user can think of his or her data structures as global constructs and typically need not worry about individual processors.

To illustrate the points made in this section, we now briefly discuss the MIMD and SIMD implementation of several applications. We have chosen these particular codes to compare the implementation styles and because they make a case for the added flexibility MIMD offers.

3: Charged Particle Beams

The BUCKSHOT computer model [3,4] calculates the motion of an intense electron beam traveling at velocities close to the speed of light. Researchers use the model to examine physical effects that conspire to produce defocusing and beam spreading. The goal is to develop experimental techniques to propagate an intense beam over long distances while keeping the beam focused to within a few centimeters. Such a beam might be used in an accelerator or as a SDI weapon.

The simulation geometry is shown in Figure 3. The beam is emitted from an accelerator (at the left) and propagates to the right. As the beam electrons move forward they interact with a background (stationary) plasma channel (ions) which serves to focus the beam. Hence, at the front the beam is focused as shown in the figure. However, wave-like oscillations and instabilities can be induced at the back of the beam because of channel perturbations and phase mixing that takes place as the beam electrons oscillate back and forth across the plasma channel. The final result can be a diluted, spread-out beam.

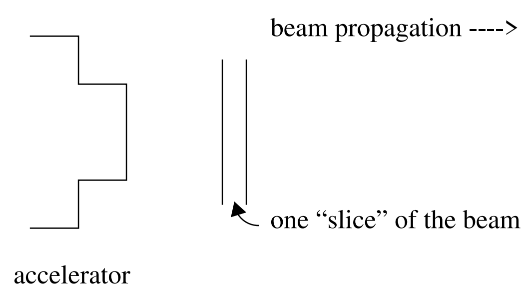

Figure 3: Relativistic electron beam propagating from left to right after being emitted by an accelerator. The computation in different slices of the beam can be done independently during one timestep.

To model these effects, the simulation uses ~10^6 particles of several species: beam electrons, channel ions, channel electrons, etc. Because the beam is moving at velocities close to the speed of light and the wavelengths of interest are much longer than the diameter of the beam (the figure is not to scale), a relativistic approximation can be invoked whereby at each timestep the particles in one transverse slice of the model (see the figure) only interact with each other. These forces are computed for all beam-beam, beam-channel, and channel-channel interactions in each slice and the particles moved via the relativistic Maxwell equations. At the next timestep the beam particles slide forward relative to the channel and a new set of interactions is computed. Thus, a natural decomposition of this model onto a parallel machine is to have each processor own one or more slices and compute all the interactions in those slices. The only communication then required is to pass beam particles to another processor between timesteps.

MP versions of BUCKSHOT have been developed for the nCUBE and CM-2 using this decomposition strategy [4], since the force interactions can be computed in a synchronous fashion for all slices simultaneously. The performance of the code on the two machines and of the original single-processor CRAY-YMP version is given in Table 1. The YMP version vectorizes well and runs at about 120 Mflops.

Buckshot Simulation	YMP 1 proc	nCUBE 2 1024 proc	CM-2 16K
Time (sec)	59.5	6.6	31.0

Table I: CPU time (in seconds) for a timestep of a 500-slice charged particle beam with 1024 particles/ slice.

The nCUBE code is the fastest MP implementation (~ 1 Gflop) for two reasons. The first is that when the simulation begins as the beam is first emitted from the accelerator, there is only one slice. At the next timestep there are two, then three, etc. The beam reaches its full length after a few hundred timesteps; it subsequently propagates forward through new channel particles for the remainder of the simulation. Thus, at early times in the simulation, one wants to assign multiple processors to a slice; at later times there may be one or more slices per processor. This growing domain presents load-balancing problems for the SIMD machine, but the MIMD environment allows processors to be mapped and remapped to the domain more flexibly, since one has direct control over what processor owns what slice.

Secondly, when the number of particles per slice is large, there is a faster NlnN force solver that can be used within a slice instead of the N^2 pairwise solver described earlier. These timings are about a factor of 2-4x faster for the nCUBE and CRAY than those listed in Table 1. It is most easily implemented when one processor owns all the particles in a slice. On the CM-2 this data layout will not perform well until the beam grows to the point there are four times as many slices as floating point processors. Since typical runs are at most a few thousand slices in size, this means a slower solver must be used on the CM for a large portion of the simulation. With the data layed out in a 2 dimensional array (particles,slices), more communication between processors is also required for even the N^2 solver.

4: SAR Image Processing

Synthetic aperture radar (SAR) is a commonly used tool in reconnaissance and remote sensing. It combines high angular resolution with day-night and all-weather imaging capability. A SAR system transmits and receives coherent broadband signals that are stored, combined, and processed to synthesize a large aperture [5]. Important applications of SAR include arms control and treaty verification.

SAR data, as collected, suffers from many kinds of errors which produce blurry images. One of the most significant is due to uncompensated movement of the radar platform (the airplane) during aperture synthesis. An iterative algorithm known as phase-gradient autofocusing (PGA) [6] can correct these effects to produce highly focused images and has the advantage of being a naturally parallel computation [7]. A large image (16K by 16K for example) can be corrected quickly by first calculating a phase correction factor for a small subpatch (1K by 1K) of the image. This involves identifying the strongest point scatterers in each row of the image, isolating their support (masking off all but nearby pixels), and performing 1-d FFTs (along rows) to convolve the calculated factor with the original image prior to the next iteration. These operations can be done in parallel most quickly if the data is layed out as one or more rows per (floating point) processor. Once the phase correction factor is calculated it can be applied to all other subpatches in the full image very quickly via 1-d convolution FFTs. This is again very fast on both machines if entire rows are stored in the local memory of a processor. Timings for the two parts of the PGA algorithm on the MIMD and SIMD machines are listed in Table 2.

PGA Implementation	nCUBE 2 1024 proc	CM-2 16K
Correct (1024x1024)	3.4	21.0
Apply (256x)	16.0	16.6
Correct (2048x2048)	9.91	42.0
Apply (64x)	17.3	18.1

Table 2: Timings (in seconds) of the phase-gradient autofocus algorithm on a 16K by 16K image. The first set of timings is for correcting a 1024x1024 subpatch and applying the phase correction factor to the other 256 subpatches. The second set of timings is using a 2048x2048 subpatch size.

The nCUBE performs the "correct" portion of the algorithm at about 600 Mflops, while the CM-2 is somewhat slower, primarily because there are a few asynchronous components of the code (the search for and isolation of point scatterers) that slow down the SIMD implementation. On the other hand, the "apply" portion runs at 1.8 Gflops on the nCUBE and at an (estimated) 6 Gflops on a full 64K CM-2. This is because the CM allows the image to be stored as a 3-d data set (planes of subpatches) to fill available memory and can then perform 1-d in-memory FFT's for the convolution very quickly. Thus PGA is a good match for both MIMD and SIMD architectures allowing large images to be focused in tens of seconds. We note that the chief issue keeping the MP computers from competing with dedicated hardware systems for this task is that of I/O. The MP machines can process Gbytes of image data in seconds, but cannot read or write them out to any form of permanent storage nearly so quickly.

5: Shock Physics

The ability to model shock physics is critical when studying high speed collisions between objects. Simulations often provide information that is not available from experiments and can help design and analyze experiments. The CTH code was developed to investigate a wide spectrum of shock physics problems ranging from armor/antiarmor performance to weapons safety [8]. This code has recently been used to analyze potential damage to a NASA space station from collisions with space debris.

Currently, even 2-d calculations in this field can take many hours on the fastest vector computers. Order of magnitude increases in computer speed are needed to solve three-dimensional problems in reasonable time frames. Large memories are also a requirement to adequate 3-d grid resolution. MP computing is being pursued to overcome these hurdles and develop a credible three-dimensional capability.

A shock physics code called CTH has been partially ported from the CRAY-YMP to our MIMD and SIMD machines [9,10]. The original CTH code solves mass, momentum and energy conservation equations for multiple materials by explicit time integration methods. A high-resolution interface tracker is used to prevent distortion of material interfaces. High temperatures and multi-phase equations of state (solid, liquid and vapor) are considered in the model. Elastic-plastic behavior, high explosive detonations and fracture are also treated. An Eulerian mesh is used to solve the partial differential equations and second-order differencing is employed on most approximations to the governing equations. For the MIMD implementation the 2-d grid is allocated as block of grid cells to each processor; for the SIMD machine a (virtual) processor owns each grid cell.

The portion of the port that has been completed (for which timings are listed in Table 3) is the hydrodynamics part of CTH, which includes multiple material handling and the interface tracker. The timings are promising because they indicate the MP machines can compete with the CRAY for even the part of the code that is highly vectorized.

CTH Mesh Size	YMP 1 proc	nCUBE 2 1024 procs	CM-2 16K
128 x 128	20.0	7.9	11.5
256 x 256	16.0	4.1	7.1
512 x 512	15.0	3.1	5.6

Table 3: Grind times (cpu time in microseconds per mesh cell per timestep) for a shock physics simulation on various machines for different sized meshes.

What still needs to be added to the MP versions of CTH are the more complex multi-phase equations of state. Since these portions do not vectorize as well, it is anticipated that the MIMD implementation will perform better than the SIMD. This is because a MIMD processor can more readily branch to a different physics routine depending on the temperature, pressure, or phase state of the material(s) in a particular cell it owns.

A second option that the MIMD implementation will allow for is more sophisticated domain decomposition of the blocks of mesh cells to the processors. As the materials flow through the mesh, the computation can become unbalanced (varying amounts of work to do in different cells and regions). Load balancing, so that processors do not work on empty portions of the mesh can potentially be implemented on the MIMD machine, but would be more difficult for the SIMD case.

6: Molecular Dynamics

Molecular dynamics (MD) simulations are commonly used to calculate static (thermodynamic) and dynamic (e.g. transport) properties of liquids and solids. Each of the N atoms (or molecules) in the model is treated as a point mass and Newton's equations of motion are integrated to compute trajectories. From the motion of the ensemble of atoms a variety of useful microscopic and macroscopic quantities can be calculated. The physics of the model is contained in a potential energy functional for the system from which force equations are derived for each atom.

Two parallel algorithms have been developed for a general class of MD problems [11,12]. These problems are characterized by short-range forces (with a cutoff distance

r_c) and by atoms undergoing large displacements over the duration of the simulation (each atom's neighbors change rapidly in time). The chief computational task in short-range MD is to efficiently calculate the force on each atom at every timestep. This requires knowing, at any instant in time, the small subset of nearby atoms that will interact with a given atom.

The first parallel algorithm assigns each processor a fixed set of N/P atoms for the duration of the simulation, where P is the number of processors (nodes). This algorithm has the advantage of simplicity and is fastest for small problems (N < 4000 atoms). Each node updates the positions and velocities of its atoms by calculating the forces on them at every timestep. This requires knowing the positions of (potentially) all other atoms in the simulation domain; consequently, a global exchange of atom positions is performed at every timestep. Thus the computation in this algorithm is local (completely parallel) but the communication is global. To efficiently calculate forces, each node creates a neighbor list for its atoms which contains all the neighbors of an atom that are within a sphere of radius r_s, where $r_s > r_c$. The list is used for a few timesteps to calculate all possible pairwise interactions; it is rebuilt before any atom moves from a distance $r > r_s$ to $r < r_c$. Each node builds the lists for its N/P atoms by first binning all N atoms into boxes with side length $s \sim r_s$.

The second parallel algorithm takes greater advantage of the local nature of force interactions and is faster for larger simulations. Each node is assigned a fixed region of space (a small box) and calculates forces and updates the positions of all atoms within its box in a given timestep. Between timesteps, atoms may move from one processor to another as each node gathers atom information from its 26 neighboring boxes. Both the computation and communication portions of this algorithm are local in nature. If the box size s is less than r_s, each node accumulates information from several boxes in each direction to compute forces. If, however, $s > r_s$, which is the case for large problems, each node only needs to exchange atoms near the faces of its box. As in the first algorithm, neighbor lists are formed for each atom (these lists must also be passed from node to node as atoms move throughout the simulation domain).

Both of these algorithms have asynchronous aspects that make them MIMD-like. In particular, the binning of atoms within a processor, the selective exchange of atoms and neighbor lists in certain sub-regions, and the sorting of atoms depending on different cutoff lengths, all are difficult to implement efficiently on a SIMD machine. The MIMD architecture allows each node to take full advantage of these and other computational "tricks" (such as computing only half the forces - Newton's 3rd law) commonly used in serial MD.

Table 4 shows timings for the parallel algorithms on a MD simulation with Lennard-Jones potentials that has been widely benchmarked. It has a cutoff distance ($r_c = 2.5\sigma$) and particle density such that every atom interacts with ~55 other atoms at each timestep. YMP (single processor) timings are also given for an implementation of the fastest vector algorithm reported in the literature []. The CRAY times scale very linearly with N, so that for the largest problem the Intel Delta is about 25x faster than the YMP would be if there was enough memory to do a 1,000,000 atom simulation with the vector algorithm.

The fastest published work on a SIMD machine is that of Tamayo, et.al. [13] on a CM-2. They have implemented a variety of algorithms, the best of which is within a factor of two of the YMP timings in Table 4. However, just recently, they have begun work on algorithms [14] for the new CM-5 (programming it in a single-node MIMD style) which are similar in many respects to the 2nd parallel algorithm discussed here. They have promising initial results which show the MIMD algorithms to be faster than their previous SIMD implementations. Thus it is possible that in the future, the fastest large-scale MD algorithms for all parallel supercomputers will employ similar techniques and be able to take advantage of MIMD's programming flexibility for selective message passing and neighbor-list construction.

N	YMP 1 proc	nCUBE 2 1024 procs	Intel Delta 512 procs
500	.00930	.0083	.0035
2048	.0369	.0256	.0082
6912	.106	.0406	.0108
16384	.250	.0775	.0215
50000	.733	.182	.0533
250000	---	.556	.170
1000188	---	---	.550

Table 4: Timings (secs/timestep) for a Lennard-Jones MD simulation with ~55 neighbors/atom for varying simulations sizes (N = # of atoms).

7: Electron Microscopy

Analytical electron microscopy (AEM) is a widely used tool for performing atomic-scale analysis of materials. It enables atomic resolution chemical information to be characterized. In many cases, it is these compositional details that determine the useful physical properties of materials.

The challenge facing the microscopist is to determine the actual composition and spatial distribution of different materials in a sample from the raw microscope data. Since even small spot size beams undergo considerable broadening due to scattering once they enter a material, the data at a point (or pixel) in the microscope image is actually a convolution of X-ray signals from a large spatial region of the sample with the compositional profile that exists in the specimen. If the makeup of the sample is complex (and unknown), it can be an impossible task to deconvolve the data analytically. A simpler, yet effective method for accomplishing this goal is via Monte Carlo (MC) simulation [15].

In this technique, illustrated in Figure 4, the sample composition and geometry as well as the microscope characteristics are input parameters to the model. Random electron trajectories are followed through the sample (consistent with this input) and a resulting X-ray signal is the output. The signal can be compared to the experimental data and the input parameters adjusted if there is not good agreement. By iterating this procedure, coupled with the microscopist's physical intuition, a sample can be quickly and accurately characterized.

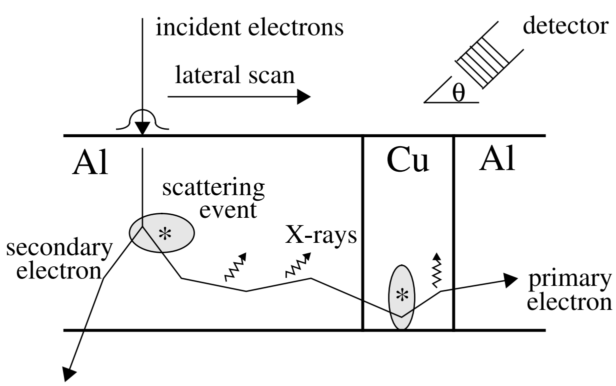

Figure 4: Diagram of the trajectory of an electron incident on an Al-Cu thin film with attendant production of secondary electrons and X-rays. Typically 10^5 electrons are followed for each impact point and dozens of impact points are used to simulate a lateral scan across the Al-Cu interfaces.

In practice, a limitation to more widespread use of a MC trajectory simulation is the large computation time needed to generate accurate statistics, particularly if the events of interest are rare (e.g. X-ray production from very small concentrations of impurities). For example, to simulate an electron beam scanning across an interface, dozens of beam impact points each with ~10^5 electrons might be modeled for one sample geometry and composition. If the entire process is done repeatedly for different sample and microscope characteristics to deconvolve real data, ~10^8 electron trajectories may be needed.

Because individual electron trajectories can be computed independently, this simulation is a natural candidate for parallel computing. On a MIMD machine, the decomposition is very simple: each processor follows N/P of the trajectories (P is the number of processors) and sums up statistics of interest at the end. This allows the basic physics routines to remain unchanged from the serial version of the code. Only minimal communication of data between processors at the beginning and end of the simulation is required. We implemented our CRAY MC electron scattering code for the nCUBE and Intel in this fashion with excellent performance results [16]. For thin film simulations the nCUBE is a factor of ~30x faster than a single processor of the YMP. For simulations of bulk samples which have much more lengthy trajectories the nCUBE is over 100x faster than the YMP, allowing many physical effects that were previously inaccessible computationally to be studied.

The reason the performance vs. the YMP is so good is because the code, as written, is inherently serial. The only loop of interest is over trajectories and it is at the very "outside" of the entire program. While it is possible to recast such codes for vectorization and achieve good speed-ups (8-10x vs. the original serial code is typical), it requires a complete rewrite of the physics routines and data structures to process vectors of particles. This is what would also have to be done for a SIMD implementation. But for a complex MC trajectory simulation this has inherent inefficiencies. At each step of an electron's path, the code must check for interface crossings within the sample, test for backscattering out of or transmitting through the sample, compute different energy loss equations depending on the electron's energy, test for secondary electron production and potentially follow that trajectory, etc. All of these imply branching in the code, often to different subroutines, and would be difficult to do efficiently in SIMD, especially since some cases occur only very occasionally.

Conversely, the MIMD implementation (N/P trajectories per processor) is straightforward and requires minimal modification of the serial code. It also load balances well so long as N/P is a reasonably large number.

8: Conclusions

We have described the implementation on SIMD (CM-2) and MIMD (nCUBE and Intel) machines of several applications. SIMD machines require, by their very nature, a code to execute synchronously on all its processors which is not flexible enough for some applications. However, if the data exchange is also synchronous, SIMD machines can handle large message and large numbers of data exchanges very efficiently. They also provide a simpler programming model since data structures are global and can be thought of as a unified whole by the programmer.

Conversely, MIMD machines do well on problems regardless of their synchronization level, allowing for very different physics to be simulated simultaneously on each processor. But the cost of message-passing overhead causes them to perform poorly on simulations requiring small (or large) amounts of data to be frequently exchanged between processors. Additionally, they require more effort on the part of the programmer to explicitly partition data across the processors, pack and unpack data into message buffers, and exchange messages between processors.

The recently introduced CM-5 has components of both a SIMD and MIMD architecture within it. In fact, while it retains the data parallel model [2] for SIMD applications with users allocating global arrays across all the processors, it also allows users to write (single) node-level code. In this mode all data references are to local memory and explicit MIMD-style message-passing can be done between nodes. The issues of data locality and infrequent communication discussed in this paper will be important for high performance in this mode of programming the machine. It will thus be possible to write high performance MIMD code for the CM-5, providing the single-node code will vectorize well for the node's floating point hardware. What is less clear is what the performance of unvectorizable MIMD applications (such as the one discussed in section 7) will be.

We find it encouraging that the new generation of parallel machines (e.g. CM-5 and Intel Paragon) are supporting a MIMD programming environment. This will make the task of porting codes between different MP machines easier and provide more flexibility to the programmer to attack a wider range of scientific and engineering problems.

Acknowledgments

The authors thank their many collaborators at Sandia who have worked with them on these applications. Some have also provided useful suggestions for this paper. However, since views on these issues are often held with near "religious" fervor, the authors must claim sole responsibility for the final product.

References

[1] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, Solving Problems on Concurrent Processors, Vol. 1, (Prentice-Hall Inc., Englewood Cliffs, NJ) 1988.

[2] B. M. Boghosian, "Computational Physics on the Connection Machine", Comp. in Phys., Jan/Feb 1990.

[3] I. R. Shokair and J. S. Wagner, "Analysis and Benchmarking Calculations for the Buckshot Code," Sand87-2015 Report, Sandia National Laboratories, Albuquerque, NM (1987).

[4] S. J. Plimpton, I. Shokair, J. Wagner, "Load Balancing and Performance of a Gridless Particle Simulation on MIMD, SIMD, and Vector Supercomputers", proc. of 5th SIAM Conference on Parallel Processing for Scientific Computing", Houston, TX, March 1991.

[5] D. A. Ausherman, A. Kozma, J. L. Walker, H. M. Jones, and E. C. Poggio, IEEE Trans. Aerospace Electronic Systems, AES-20 (1984), 323.

[6] P. Eichel, D. Ghiglia, and C. Jakowatz, Jr., "Speckle processing method for synthetic-aperture-radar phase correction", Optics Letters, 14 (1989), 1.

[7] S. J. Plimpton, G. Mastin, and D. Ghiglia, "Synthetic Aperture Radar Image Processing on Parallel Supercomputers", proc. of Supercomputing '91, (published by IEEE), p. 446.

[8] J. M. McGlaun, S. L. Thompson, and M. G. Elrick, "CTH: A Three-Dimensional Shock Wave Physics Code," Int.J. Impact Engin. 10 (1990).

[9] H. E. Fang, A. C. Robinson, and Kah-Song Cho, "Hydrocode Development on the Connection Machine", proc. of the 5th SIAM Conference on Parallel Processing for Scientific Computing, Houston, TX, March 1991.

[10] A. C. Robinson, C. T. Vaughan, H. E. Fang, C. F. Diegert, and K. Cho, "Hydrocide Development on the nCUBE and the Connection Machine Hypercubes", proc. of 1991 APS Topical Conf. on Shock Compression of Condensed Matter (1991).

[11] S. J. Plimpton, "Molecular Dynamics Simulations of Short-Range Force Systems on 1024-Node Hypercubes," Proceedings of the Fifth Distributed Memory Computer Conference, (published by IEEE), pp. 478-483.

[12] S. J. Plimpton, G. A. Geist, "Fast Parallel Algorithms for Molecular Dynamics", submitted for publication.

[13] P. Tamayo, J. P. Mesirov, and B. M. Boghosian, "Parallel Approaches to Short-Range Molecular Dynamics Simulations", proc. of Supercomputing '91, (published by IEEE), p. 462.

[14] personal communication with P. Tamayo.

[15] A. D. Romig Jr., S. J. Plimpton, J. Michael, R. L. Mykelbust, and D. E. Newbury, "Application of Parallel Computing to the Monte Carlo Simulation of Electron Scattering in Solids: A Rapid Method for Profile Deconvolution", proc. of XIIth International Congress for Electron Microscopy, Seattle, WA, August 1990.

[16] S. J. Plimpton, J. R. Michael, A. D. Romig, Jr., "Parallel Simulation of Electron-Solid Interactions for Electron Microscope Modeling", submitted to J of Supercomputing.