LA-UR-89-2607

LA-UR--89-2607

DE89 016607

TITLE Vectorization of Monte Carlo Particle Transport: An Architectural Study Using the LANL Benchmark "GAMTEB"

AUTHOR(S) Patrick J. Burns
Mark Christon
Roland Schweitzer
Olaf M. Lubeck
Harvey J. Wasserman
Margaret L. Simmons
Daniel V. Pryor

SUBMITTED TO IEEE Supercomputing '89
Reno, Nevada, November 13-17, 1989

MASTER

Los Alamos Los Alamos National Laboratory
Los Alamos, New Mexico 87545

DISTRIBUTION OF THIS DOCUMENT IS

Vectorization of Monte Carlo Particle Transport: An Architectural Study Using the LANL Benchmark "GAMTEB"

Patrick J. Burns, Mark Christon And Roland Schweitzer
Colorado State University
Dept. of Mech. Engr. and Univ. Computer Center
Fort Collins, CO 80523

Olaf M. Lubeck, Harvey J. Wasserman, Margaret L. Simmons
Computing and Communications Division
Los Alamos National Laboratory
Los Alamos, NM 87545

Daniel V. Pryor
Supercomputing Research Center
17100 Science Drive
Bowie, MD 20715-4300

Abstract

Fully vectorized versions of the Los Alamos National Laboratory benchmark code *Gamteb*, a Monte Carlo photon transport algorithm, were developed for the Cyber 205/ETA-10 and Cray X-MP/Y-MP architectures. Single-processor performance measurements of the vector and scalar implementations were modeled in a modified Amdahl's Law that accounts for additional data motion in the vector code. The performance and implementation strategy of the vector codes are related to architectural features of each machine. Speedups between fifteen and eighteen for Cyber 205/ETA-10 architectures, and about nine for CRAY X-MP/Y-MP architectures are observed. The best single processor execution time for the problem was 0.33 seconds on the ETA-10G, and 0.42 seconds on the CRAY Y-MP.

Keywords: vectorization, parallel algorithm, branching, sparse operation, gather, scatter, Monte Carlo.

1. INTRODUCTION

1.1 General

Monte Carlo methods, introduced by Ulam and Von Neumann [1], have received intermittent but continuing attention from the research community [2-9]. Present supercomputers are sufficiently powerful to simulate "real" problems in some instances, but many problems are still CPU bound. The advent of gather/scatter hardware, available in supercomputers since 1981, has made the efficient vectorization of such problems possible. The random and *a priori* unknown branching implicit in such algorithms generally necessitates a full recasting of the algorithm before it can be vectorized, often an imposing task. Perhaps this is why this topic has been addressed only recently by the research community.

Brown [10] first developed a strategy to vectorize neutron transport, where the vectorization was over the particles in a single cell. He reported theoretical (not observed) speedups of up to 40 for an event-based algorithm on a Cyber 205 over an Amdahl 470V/8. Martin et al. [11] reported, for a vectorized zonal approach of tracing photons in axisymmetric enclosures, speedups of order 5 over optimized scalar code on Cray-1 and single processor Cray X-MP machines, and about twice that observed on a CDC 7600. A parallel processor photon tracing code (coarse-grained parallelism) has been implemented by Sequent Computer Systems [12], and is reported to have achieved nearly linear speedup when applied to 12 processors. Bobrowicz et al. developed vectorized codes for both photon [13] and neutron [14] transport based on the physics in the MCNP [15] production code. They accumulate particles in queues until some optimal number is reached. In this way, arithmetic operations are delayed until they can be done at efficient vector lengths, yielding speedups of approximately eight. Simmons and Wasserman have also developed a multitasking version of MCNP for the CRAY-2 and CRAY X-MP[16]. Pryor and Burns [17,18] have achieved speedups of 13 to 30 for two-dimensional photon tracing problems on the Cyber 205. Maltby [19] reported speedups of 11 to 16 for a three-dimensional photon tracing algorithm, also on the Cyber 205. In those studies, vector lengths were kept long by emitting new photons whenever old ones were either absorbed or transmitted.

1.2 Scope of Present Effort

The task we have undertaken is to vectorize the Los Alamos National Laboratory benchmark *Gamteb*, which is virtually unvectorized [20] in original form. The *Gamteb* algorithm simulates the axial emission of photons from a source adjacent to the face of a long, narrow cylinder, in the direction of the face at the other end. Photons are traced until they escape through the cylinder wall, are transmitted through the opposing face, or are backscattered through the adjacent face. Although the geometry is simple, the physical interactions are quite challenging to simulate, including the generation of new photons through collision processes and variance reduction techniques.

This paper describes implementations of the vectorized *Gamteb* on the ETA10 G and CRAY Y-MP supercomputers. The new implementation was initially designed for the CYBER-205 at Colorado State University, making substantial use of the CDC vector primitives. This code was then translated into a version that could run on the CRAY X-MP, following which, a new version, more suitable for the CRAY, was developed.

These machines have diverse architectural features that render the implementation of a vectorized Monte Carlo algorithm of interest to the research community. For conciseness, we will refer to an ETA implementation, which encompasses the ETA-10 and the Cyber 205, and a Cray implementation, which refers to the CRAY X-MP and Y-MP, although we also present results for a dynamic memory CRAY-2 (SN 2007).

It is useful to examine the present effort in the context of previous work. Previous applications to photon tracing between surfaces where particles do not interact [17-19] were algorithmically simple, but involved very complicated geometries. Substantial effort went into construction of a very efficient, vectorized surface intersection loop. A more complicated application to Molecular Aerodynamics [21-22] incorporated particle-particle interactions, and had the further difficulty of requiring temporal synchronization among grid cells. Yet the geometry was the simplest possible -- one dimensional. Various facets of that problem resulted in short vector lengths, and consequently, poor performance on the Cyber 205.

The present work represents a mix of features from these previous applications. The geometry is simple, and the physical particle interactions are moderately complex. Particles interact in a limited sense through selective production of pairs during certain collisions. No synchronization is required, and vector lengths should, on average, remain long (but will decay to zero as the simulation terminates). However, the present algorithm incorporates variance reduction techniques, including splitting and Russian Roulette (to be discussed in Section 2), which complicate the data motion. These features make the present effort an ideal logical extension to the previous work.

Our approach is conceptually very similar to that of Bobrowicz et al. [13], where particles were accumulated in multiple queues (a total of fourteen were required) until vector lengths become long enough for efficient computation. Our method differs in that we form only two queues. In this fashion, we defer the intermediate overhead of "handing off" particles to queues until all logical operations have occurred for the operation. Also, we incorporate the complexity inherent in the variance reduction strategies mentioned above. Finally, we explore in greater detail the performance of the algorithm over a wide range of vector lengths, and investigate random data motion, as it applies to Monte Carlo methods, on vector architectures.

The goal of this work is to compare the architectural and software features of the ETA and CRAY systems that play important roles in these types of problems. We will do so in terms of a theoretical framework that is general to any vector processor. We will also discuss some of the tradeoffs we encountered in implementing the codes and describe the vector algorithms in detail.

2. PHYSICAL MODEL

Gamteb is an implementation of a Monte Carlo photon transport algorithm. Photons are traced from birth to death, through intermediate collisions with the medium. As collisions occur, photon energies and numbers may vary. In Monte Carlo particle tracing algorithms, these situations are handled as follows. Each logical particle trace consists of a number of monoenergetic photons that are dealt with algorithmically as a single unit. This unit is loosely referred to as a particle, with two properties characteristic of the energy content - the energy of the photons (referred to as the energy value), and the statistical importance of photons (referred to as the particle weight). As photons are absorbed, the particle weight decreases. As the particle interacts with the medium, the energy value decreases. The product of these two quantities determines the total energy content of the particle. To distinguish physical from algorithmic quantities, we will refer to algorithmic quantities as particles and physical quantities as photons. We now proceed with a discussion of the physical model.

Gamteb calculates the transport of gamma rays in a carbon cylinder of radius 1 and length 30. The geometry is pictured in Figure 1, showing the cylinder wall (surface 2), the left end (surface 1), the right end (surface 4), and the internal partition (surface 3). The source is at Surface 1, with photons being emitted to the right along the major axis of the cylinder. Surface 4 is the target. Photons that exit through surfaces 1, 2 and 4 are referred to as "backscattered," "escaped" and "transmitted," respectively. We want to determine the combined weights of particles that exit each surface, as a function of energy level (35 energy levels are used). In this paper, for simplicity, we report only the total number of particles (summed over all 35 energy levels) that pass through these surfaces.

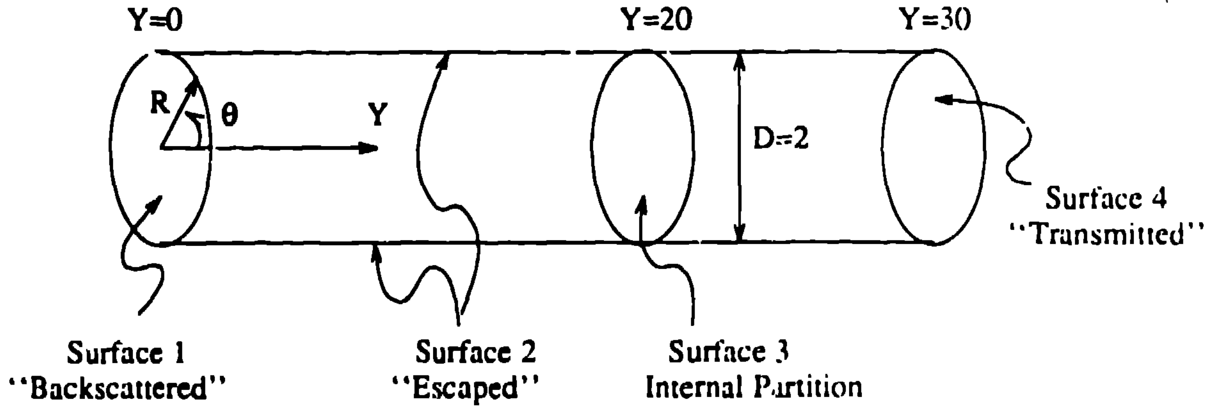


Figure 1
The physical system.

Surface 3 differs from the other surfaces in that it is only a logical surface, i.e., it partitions the geometry into an "uninteresting" portion (the left portion), and an "interesting" portion (the right portion). This artificial construct is used to improve the statistical properties of those particles transmitted (i.e., improve the accuracy of the answer), and simultaneously reduce the amount of (less useful) work to be done. Because the geometry is long and thin, few of the original particles will strike the target - resulting in poor statistical convergence. The convergence of the simulation varies as 1/sqrt(N), while the time is proportional to N, where N is the total number of particles tracked. An optimum balance between accuracy and runtime is done with biasing, the assignment of more importance to particles near the target. In *Gamteb* this is done by "splitting" those particles that cross the internal partition from left to right. Upon splitting, particle weights are halved, and then each particle is "cloned." (Note that this preserves the total number of photons, and ensures conservation of energy.) This requires that daughter particles be traced along with the parent particles. Heifetz has given a particularly lucid explanation of such variance reduction techniques [23].

Similarly, the strategy of Russian Roulette reduces the amount of work tracing "uninteresting" particles (i.e., those moving away from the target). Particles which cross the internal partition from right to left undergo Russian Roulette, where a random half (from a uniform distribution) "die," and the remainder each have their weights doubled. We rely on the fact that the random sample of that half of particles which "die," will, on average, have a total energy product equal to that of the survivors. Large numbers of particles are required for this assumption to be valid.

The physical model employed is relatively realistic for photons, although the geometry is very simple (few surfaces and only a single material). Photons may interact with the medium in a variety of ways. A particle may undergo: (1) energy exchange scattering with the medium into a direction dependent upon particle energy, (2) "pair production," where a pair of lower energy particles are produced that travel in random directions, or (3) absorption by the medium. All of these interactions are dependent on energy by virtue of energy-dependent cross sections for scattering, pair production, and photoionization.

The algorithm is thus illustrative of complex physical phenomena and not of complex geometrical aspects. This particular example was chosen as a representative test case for vectorization of the physical aspects, and as such is a reasonable first step in an effort to assess whether such methods should be further investigated through inclusion of arbitrary geometries and numbers of materials.

2.2 Approaches to Vectorizing Monte Carlo

The Monte Carlo particle transport model described above can be viewed as a binary decision tree (see Figure 2). A descriptive picture of a transport execution is to view a particle as being inserted into the top of this tree. As the calculation progresses, the particle drops toward the bottom bouncing randomly either left or right (depending on the physical process cross sections, the chosen random numbers, and the characteristics of the particle itself) much like a ball in a pinball machine. Some of the branches may consume the particle, others may unleash more particles. Almost all branches change some particle characteristic. In any case, the particle and any daughters either are consumed or drop to the bottom of the tree where they are then reinserted at the top with new characteristics.

If one particle at a time traverses the tree, the execution is scalar. If a continuous stream of particles is "dumped" into the top so that the particles make their way toward the bottom independently, the computation proceeds as asynchronous parallel processes. Finally, the more organized model of synchronous vector computation consists of a batch of particles dumped into the top that can undergo separation into smaller batches at each branch. The way in which this separation is carried out, as well as when the separations are carried out, are two important differences between the ETA and CRAY implementations of vector *Gamteb*.

We illustrate the operation of the vector code with a pictorial example tracing 500 particles. We have instrumented the code to obtain particle positions after each event. We frame our discussion around an example of relatively few particles. Five hundred particles are used in this test case, 250 of which are launched for the first event. This is

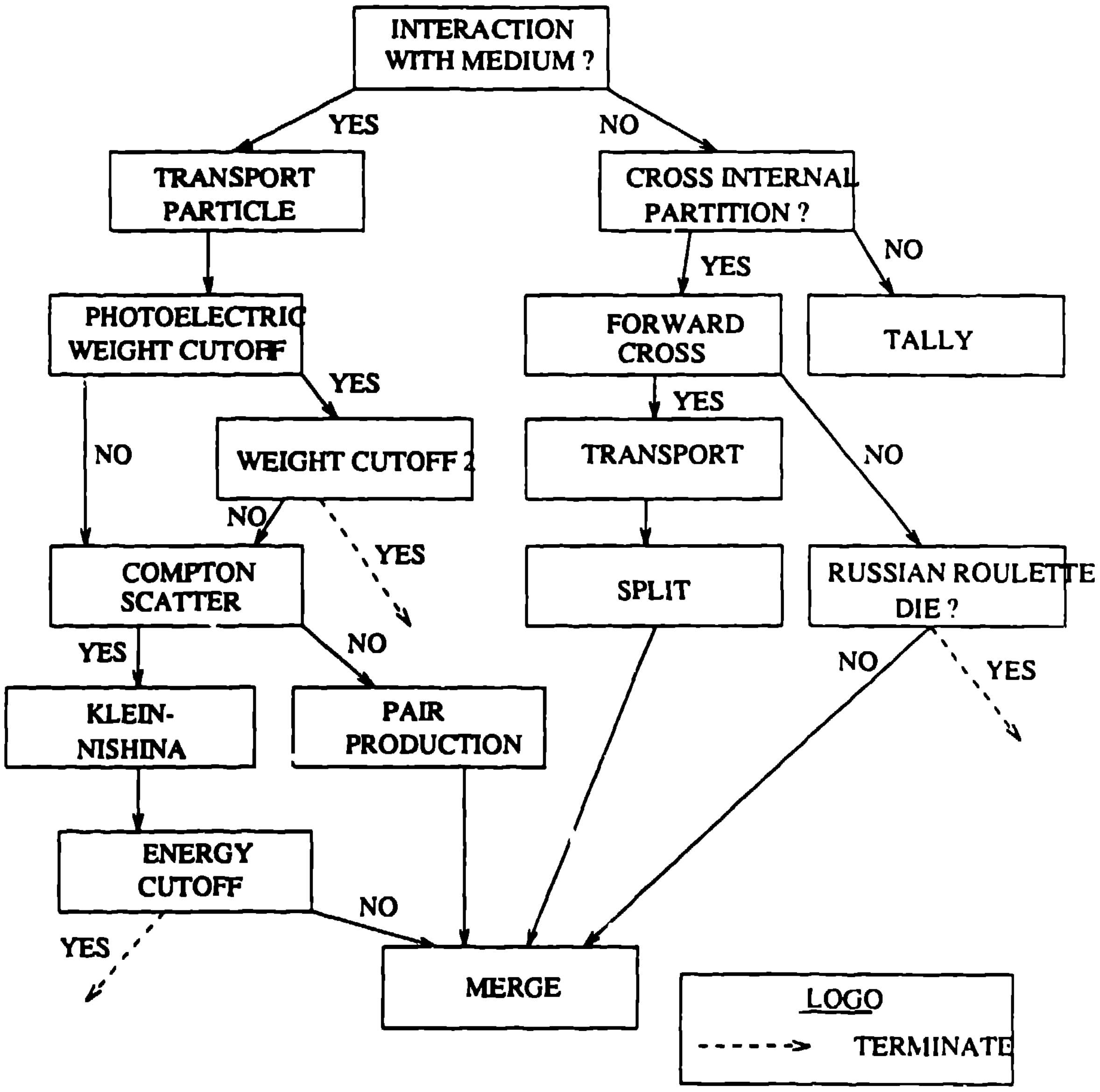


Figure 2

Binary Decision Tree

done to illustrate the operation of the algorithm for large numbers of particles since the remaining particles will be launched as storage space allows in subsequent event steps. We depict in Figure 3 particles at the end of each event step by particle position. Particle positions, even though calculated in three-dimensional Cartesian coordinates {X,Y,Z}, are displayed in two coordinates: radius - R, and position along the cylinder axis - Y (for illustrative purposes, we neglect the angular position). The dotted lines in the figures represent the internal partition, and the axis of the cylinder.

To begin the simulation, 250 particles are emitted from the source. Of these, 81 arrive at the internal partition and undergo splitting. Of the remaining 169, 149 undergo Compton scattering in the left cylinder, and the 20 remaining produce low-energy pairs.

(a) End of Event 1.

(b) End of Event 2.

(c) End of Event 3.

(d) End of Event 4.

(e) End of Event 5.

(f) End of Event 6.

Figure 3

Photons Positions at Ends of Events 1-6.

Two hundred fifty new particles are emitted at the beginning of Event 2; a total of 581 particles are then traced during this event. Of these: 227 strike an exterior surface and are tallied, 94 cross the internal partition to the right and are cloned, 241 undergo Compton scattering in the left cylinder, and 19 produce pairs.

In our example the third event proceeds much like the second, leaving 138 particles active to start the fourth event. During the fourth event, 1 particle crosses the internal partition to the right and splits, while 2 cross to the right and undergo Russian Roulette -- one dies, while the other lives. The fifth event produces one additional particle that undergoes Russian Roulette and dies.

After taking into account the particles that cross an external surface of the cylinder and the one that dies through Russian Roulette, only 7 particles remain active for event six. The surfaces of intersection and the travel distances are determined for these 7 particles. Six escape through the cylinder wall, and one is transmitted through surface 4. At this point, all original 500 particles and all of their children have been traced and undergone disposition. Final tallies are computed for the escaped, transmitted and backscattered particles; these are then divided by the number of emissions and the simulation is complete.

We now turn our attention to the random branching evident in the above example. The random branching occurring at each node of the decision tree may be dealt with in three ways. The first method physically divides the particle batch by copying all particle characteristics into separate contiguous storage regions. The advantage here is that subsets are easily tracked through the tree by their base addresses and lengths. The overhead associated with this involves the generation of two vectors of indices, two gathers, and a subsequent vector merge at the end of each event step. However, this overhead is amortized over the number of computations at each node, and the computations are performed at successively reduced loop lengths as the particle batch travels down the tree.

Alternatively, particles may be logically partitioned using two other strategies in which no actual movement of particle data occurs. The first of these uses a logical vector to track the particles on which computation is to be done. The use of logical vectors requires the composition of multiple vectors at each lower branch point. The length of any loop that comes after a branch is still the number of particles in the original batch. The other strategy uses two index vectors whose values are the particle indices belonging to each subset. We call this the "indirect method." The use of the indirect method results in loop lengths that are exactly equal to the number of particles in each subset. However, the memory access patterns are of the gather/scatter type and may not be suitable to some architectures.

In general, the higher the level in the decision tree, the greater advantage there is to physical partitioning. Specific advantages/disadvantages of each strategy must also be considered in the context of machine characteristics. The truth ratio of each branch is not important when two physical subsets are to be formed, but becomes quite important when subsequent computation is to be carried out on only one branch. This one-branch case is found in Monte Carlo situations where a "sieve" type technique must be used. In Section 5 we present some performance data that comment on the tradeoffs between all three methods.

3. A MODIFICATION OF AMDAHL'S LAW

We now develop an extended version of Amdahl's law [24] to illustrate the speedup to be gained by mapping a scalar problem onto vector hardware where the resulting algorithm has a large component of data motion. By data motion we mean either of the following: (1) Operations that rearrange elements in memory to improve performance of vector memory operations, i.e., physical partitioning. This can be either movement of elements from random locations into contiguous memory locations or scatter of resultant elements back into non-contiguous locations. (2) Excess memory fetches and computations that are ignored (not stored) so that the remainder of the computations may be done in the vector hardware (i.e., logical partitioning).

The modification of Amdahl's Law proceeds as follows. Let S be the scalar execution rate, V the vector execution rate, p the fraction of time spent in the vector hardware, F the ratio of time spent in the vector hardware due to memory latency to that time spent performing useful computations (i.e.. vector start-up), and D be the ratio of time spent in the vector hardware doing data motion to that time spent doing useful computations. Then the speedup, R, of the vector code over the optimized scalar code is given by

$$R = \frac{1}{(1-p)+p\left(\frac{S}{V}\right)[\,1+F+D\,]} \qquad (1)$$

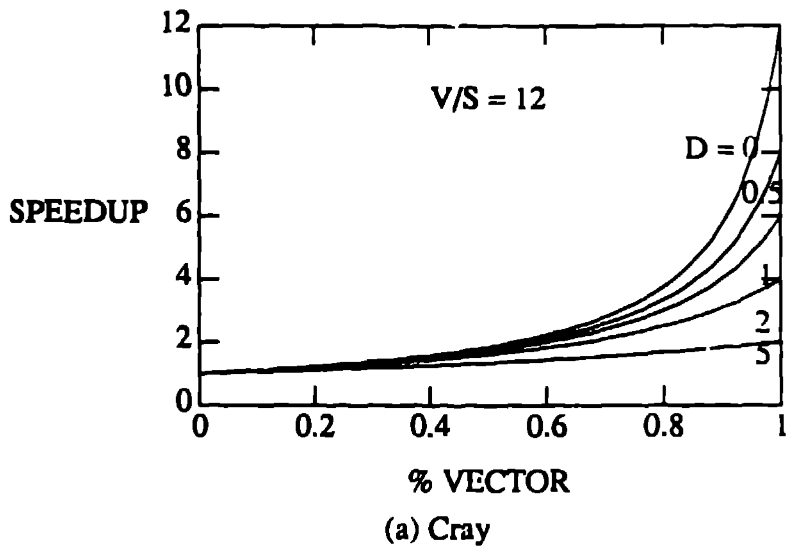
Figures 4(a) and 4(b) show plots of R versus p, with D varying parametrically. We have isolated portions of *Gamteb* and independently measured values of V/S in excess of 11 on the Cray Y-MP, and values of 17.18 and 24.05 on the Cyber 205. Thus, we select values of V/S of 12 and 20, representing, approximately, Cray and Cyber 205 hardware. (We demonstrate in section 5 that S for the ETA10 scales to about 30% less than the ratio of clock speeds, but that V scales approximately equal to the ratio of clock speeds - thus a value of $V/S = 25$ is appropriate for ETA10 architectures.) Note that the results are valid when $D \gg F$, or if D is viewed as the aggregate of D and F.

Note that Equation 1 defines a ratio between scalar and vector performance. The ratio may be large either as a result of high vector rates, V, or because of low scalar rates, S. In this case, the ETA architectures have both lower scalar rates as well as higher asymptotic vector rates than does the the CRAY X-MP, and therefore, larger values of V/S. Finally, note that for large values of V/S, p must be large to achieve substantial benefit, especially with the added penalty of D. Thus, for machines having smaller relative values of V/S, codes do not have to be as highly vectorized to achieve near-peak performance. Of course, absolute performance, i.e., the timings of the vectorized implementations, are the bottom line.

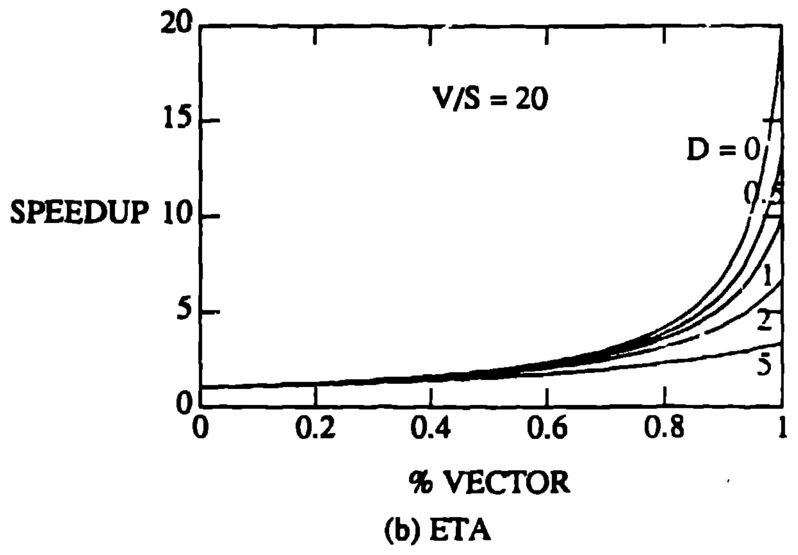
4. ARCHITECTURAL and COMPILER CONSIDERATIONS

4.1 General

ETA architectures are bit addressable [25], with memory-to-memory vector execution, and 256 scalar registers. The ETA-10 can access memory contiguously, with constant strides, or with scatter/gather. However, each memory access fetches a superword



(a) Cray



(b) ETA

Figure 4

Speedup vs. Percent Vectorization From Modified Amdahl's Law.

(eight 64-bit words), and so performance for non-contiguous fetches can degrade by up to a factor of eight [26,27]. Their scalar performance is weak, simply because of their relatively slow cycle times (except the liquid nitrogen cooled model G, which has a cycle time of about 7 nanoseconds). To offset this, more effort must be spent in vectorizing the code. To achieve good performance, we have explicitly vectorized the ETA algorithm using the Q8 Fortran extensions [28]. Implemented as in-line instructions, these allow the programmer to access the vector hardware with very low overhead -- an important feature in Monte Carlo simulation. We also make extensive use of bit vectors, for economy of storage, and to allow logical operations to proceed at approximately 8 times the rate per result as for full-word operations.

An inescapable penalty is lack of portability. Also, vector lengths cannot exceed 65,535 due to compiler limitations (longer vectors must be explicitly strip-mined). This is not an issue here, as we have designed the algorithm such that all vector lengths are bounded by this maximum. We also structure the data so that it fits entirely within local memory; thus no paging of the virtual system occurs.

The CRAY X-MP/Y-MP [29] is a shared-memory, word addressable architecture with local scalar, address and vector registers. Performance on short vectors is good, as is scalar performance. Memory access supported by Cray hardware includes contiguous, constant stride and scatter/gather. Cray performance for long constant-stride vectors is the same as for contiguous vectors[20], provided that bank-conflicts (power of 2 strides) are avoided. For a large class of (traditional) problems, goo ' performance can usually be obtained in a fashion almost transparent to the user. Cray FORTRAN implementations lack an explicit bit data type, and deny the programmer access to vector masks (even though they are used internally). However, this disadvantage is offset because scatter/gather operations are efficient on the X/Y-MP architectures [20,26].

Finally, we note that all Fortran compilers exhibit deficiencies in some key areas. Some can be easily circumvented, but others require complete redesign of algorithms. Currently, no compiler is sophisticated enough to vectorize "vanilla" scalar Monte Carlo code, and there is little expectation that such a feat will be accomplished soon.

4.2 Performance on Random Data Motion

A factor of great importance in Monte Carlo simulation is the ability of the hardware to perform data motion operations efficiently. This situation has not been fully explored for random data motion. In this section we attempt to clarify some of the issues that arise in vectorizing Monte Carlo algorithms. We look at the component operations involved in the data motion aspect of the problem, with a view toward quantifying the tradeoffs that will likely be encountered in general situations. Thus we consider the atomic operations of index list generation, logical vector generation, and the actual movement of data by gathering. As we attempt to show, the choice of method is dependent on input vector length, truth density, number of operations to be done on the partitioned set, and the number of associated vectors required for further computation. The issues are complex, but an understanding of these basic timings can lead to intelligent, if not always optimal, use of these machines.

To perform vectorized gather/scatter operations, a vector of indices is required. The generation of such a vector is not currently vectorized by Cray's CFT77 compiler (or for that matter, by ETA's compiler, but a variety of vector ETA Fortran extensions exist for such purposes). The standard Fortran is as follows (where RANDOM(I) is a vector of uniformly distributed random numbers):

```
K = 0
DO 100 I = 1, LENGTH
      IF (RANDOM(I) .LT. DENSITY) THEN
            K = K + 1
            INDEX(K) = I
      ENDIF
```

```
        100    CONTINUE
```

Note that this loop generates only the "true" index list of the physical partition -- an additional operation is required to obtain the "false" index list. Cray's library of scientific subroutines, SCILIB [30], does provide a set of subroutines (e.g., WHENFLT, WHENFGE) that generate the vector of indices in the vector hardware. However, subroutine overhead is incurred for each call. This could be a potential drawback in general, although in GAMTEB we have found the overhead to be small (≈ 0.5%).

If one adopts the logical partitioning approach, after generating the logical vectors (i.e., an integer is set to 1 when the condition is "true" and 0 when "false"), the SCILIB routines WHENEQ and WHENNE may be used to generate indices. Using this approach, one can with logical vectors do a series of operations encompassing one or several levels in the binary decision tree, then use the WHENEQ or WHENNE routines to generate lists of indices for a subsequent vector merge. As we show below, this avoids the substantial overhead of generating indices, and has been found to be the most effective strategy for both architectures.

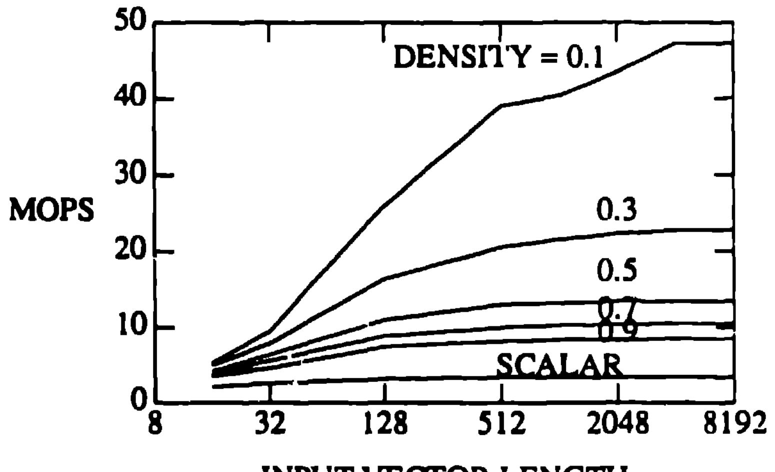
In the data below, we plot execution rates in millions of operations per second (MOPS) versus original vector length, with density varying parametrically. The density is specified in the loop above, with uniformly distributed random numbers in the interval (0,1). Where the random number is less than the density, those elements are gathered. Thus, elements in random locations are gathered.

4.2.1 CRAY Y-MP

Figure 5 presents CRAY Y-MP execution rates for generating the vector of indices, INDEX, versus input vector length, LENGTH. These are results from Cray's SCILIB routines WHENEQ, generated in the vector hardware. Execution rate in Figure 5 is the ratio of the number of generated indices to the time. The density varies from 0.1 to 0.9 in increments of 0.2 (n.b., the randomness occurs in location, not frequency of occurrence). Also included are results from the (scalar) FORTRAN code above for a density of 100% (execution rates for lesser densities scale as the density). Asymptotic vector execution rates are dependent upon density, ranging from 6 MOPS for a 10% density to 10 MOPS for a 90% density. The code for WHENEQ consists of both comparing and index incrementing. The time for comparison is constant over the entire range of truth ratios, while the time for incrementing increases with increasing truth ratios. Thus, lower rates of index production are observed at lowest density.

It is worth mentioning that often we require index lists for both branches of the conditional. As currently implemented, SCILIB has no routine to accomplish this efficiently, without making a call to, say, WHENEQ followed by a second call to WHENNE. In fact, one routine could easily produce two index lists, one for each branch, where the time would be less than the total time for two calls.

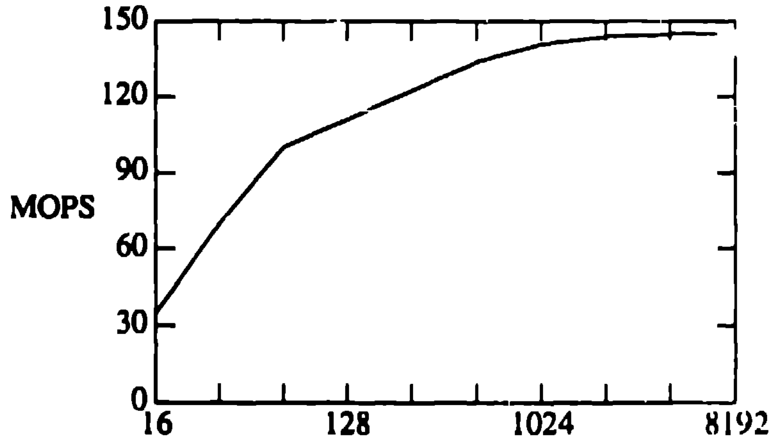
Figure 6 presents CRAY Y-MP execution rates for vector gather operations versus vector length. These data are independent of truth ratio because no unnecessary elements are gathered. The execution rates for these gathers are roughly an order of magnitude greater than those for index generation (Figure 5), showing that explicit generation of indices using SCILIB routines is a significant bottleneck. To gain good overall execution rates, the time for the SCILIB routines must be amortized over many floating point operations.



INPUT VECTOR LENGTH

Figure 5

Cray Y-MP Execution Rates for Generation of Index Vectors.



INPUT VECTOR LENGTH

Figure 6

Cray Y-MP Execution Rates for Random Gathers.

4.2.2 ETA10

In this section, we present results from runs performed on the Cyber 205 at Colorado State University. We use these as representative of ETA architectures, when appropriately scaled. However, there are slight differences in the architectures that must first be accounted for. In the ETA line, some scalar speed was sacrificed to yield decreased vector startup times. Therefore, short vector performance scales faster than the clock .ate from the CYBER to the ETA, while performance on long vectors has been observed to scale directly according tc clock rate. In any case, the differences between

ETA and Cray architectures are substantial, making the differences between the ETA and Cyber 205 architectures of lesser significance.

On ETA architectures, there are three ways to perform data motion, each with relative advantages within an operating scenario. First, we examine the rates at which vector "control" operations can be performed. These include, in Figure 7, the rates at which bit vectors can be generated and then used to either: (1) directly perform a vector compress, or (2) generate a vector of indices by performing a vector compress on an array of integers in sequence from 1 to 65,535. The compress preserves original order, and as such is a special case of a gather operation. The gather does so in this context as well. Execution rates are shown in Figures 8 and 9, for the: (1) compress, and (2) subsequent gather, respectively. Note that, in Figure 7, execution rate is based upon the number of logical bits constructed (both true and false), while in Figures 8 and 9 rates are based on the number of elements resulting from the compress or gather, respectively.

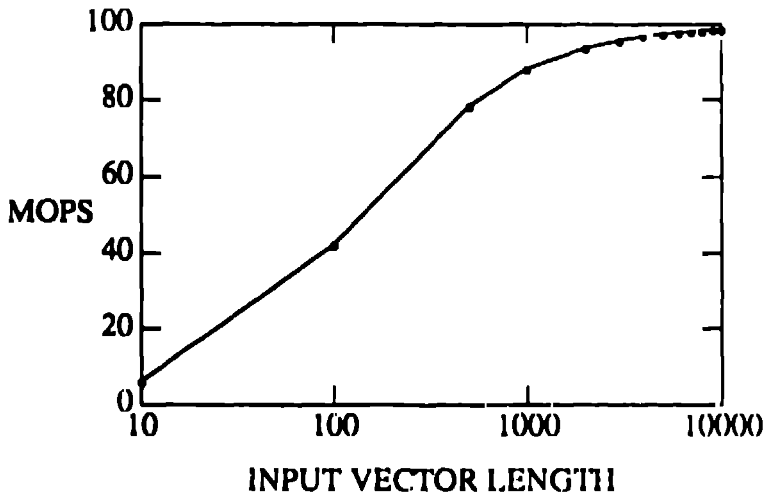


Figure 7

Cyber 205 Execution Rates for Construction of Bit Vectors.

These operations, as implemented, result in an explicit bit vector, which can then be used (multiple times) in subsequent operations for further partitioning. Such is useful when the algorithm contains nested data motion -- typical in Monte Carlo simulation. For example, logical operations on two logical bit vectors (e.g., for those photons which intersect surface 3, and for those which are traveling to the right) yield a third bit vector which can then be used for additional control (e.g., the logical AND of these two yields a control bit vector for splitting). These operations are very fast on the ETA because, as the stream rate from memory remains the same, multiple resultants are generated in each pipeline per clock tick. Execution rates for logical operations on bit vectors are shown in Figure 10. Use of bit vectors also substantially reduces memory requirements.

The generation of bit vectors proceeds independent of density, and approaches an asymptotic rate of about 100 MOPS. This is from two to ten times as fast as the generation of indices on the CRAY Y-MP. The asymptotic execution rate of the COMPRESS operation is nearly linear as a function of density, varying from about 10 MOPS at 10% density to 90 MOPS for full operations. GATHER operations, alternatively, have

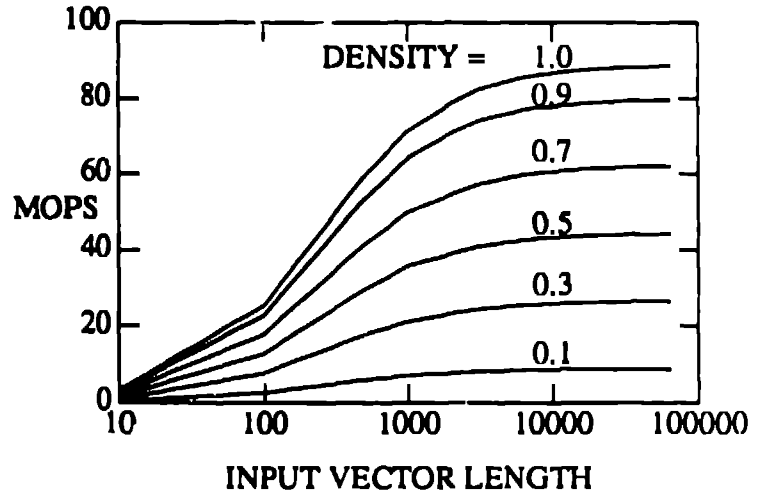


Figure 8

Cyber 205 Execution Rates for Random Compresses.

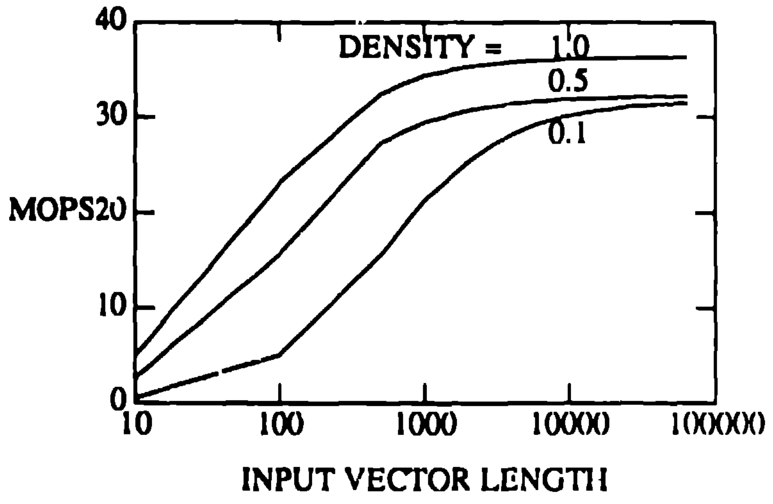


Figure 9

Cyber 205 Execution Rates for Random Gathers.

execution rates nearly independent of density, with asymptotic execution rates of approximately 35 MOPS. Thus for densities greater than about 35%, it is more economical to perform a COMPRESS, especially when including the additional overhead of index generation from the bit vector. Gathers on the Cray proceed at higher rates than on the Cyber 205, but generation of indices proceeds at higher rates on ETA architectures. When the Cyber 205 rates are scaled to the clock of the ETA10-G (7 ns vs. 20 ns), the asymptotic rate for gathers is approximately equal to that of the CRAY Y-MP. Note that Cray performance is good at low densities, and ETA performance is good at high densities.

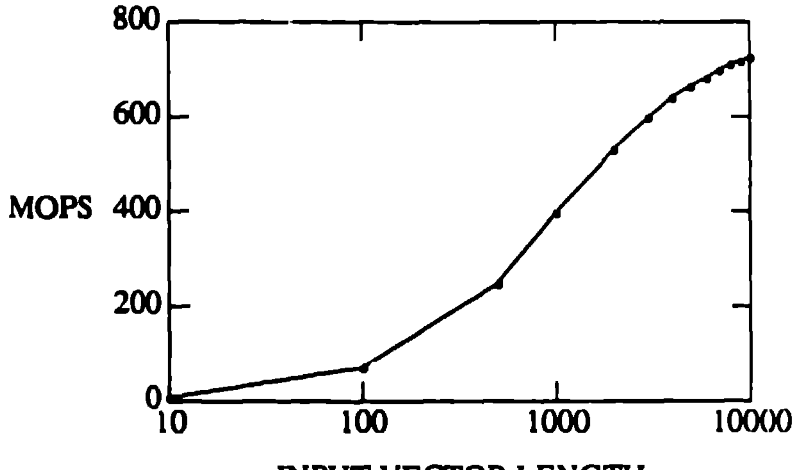


Figure 10

Cyber 205 Execution Rates for Bit Operations.

The single remaining strategy for data motion is the "WHERE" operation, for which calculations are performed using the entire vector, and results are stored only where the logical is set ("TRUE"). This operation can be done using an existing control bit vector, or by specifying a logical vector (e.g., vector 1 .LT. vector 2), in which case it displaces the generation of the bit vector and index list, the gather, and the subsequent scatter. "WHERE" operations incur a constant overhead of about 100 nanoseconds, independent of vector length and density. As such, they are very efficient for relatively full operations, and for locations near the bottom of the decision tree where there is little additional partitioning to be done.

We have chosen to combine the strategies of using bit vectors and WHERE constructs in the ETA implementation. This avoids the relatively slow gathers (here, we assumed *a priori* that the data motion would not be sparse; however some operations were). We use bit vectors that encompass several levels of the binary decision tree. This allows the intermediate logical operations to proceed at bit vector rate at the expense of vector computations of full (where "full" denotes the length at the beginning of the several levels of current interest) length. We believe this approach to be nearly optimal, however we have not coded the algorithm using alternative strategies and can only state this without proof.

5. RESULTS AND DISCUSSION

5.1 Timings and Speedups

Presented in Figure 11 are the CPU timings from just the solution phases of these runs for 40,000 emissions, as the vector length varies from 1,000 to 15,000. All runs use RANF pseudo-random sequences. A slight effect of vector start-up time is seen at vector lengths below 4,000. The ETA results at long vector lengths scale as the ratio of clock speeds.

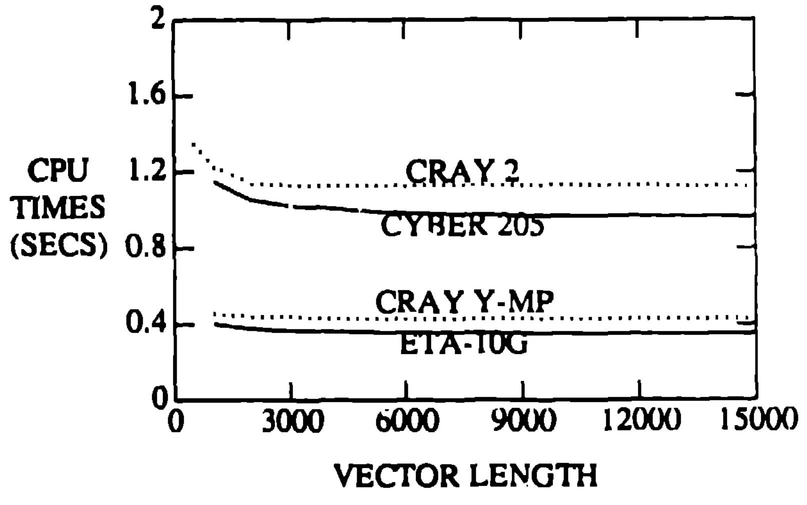


Figure 11

Solution Phase Timings of Vectorized *Gamteb* Algorithm.

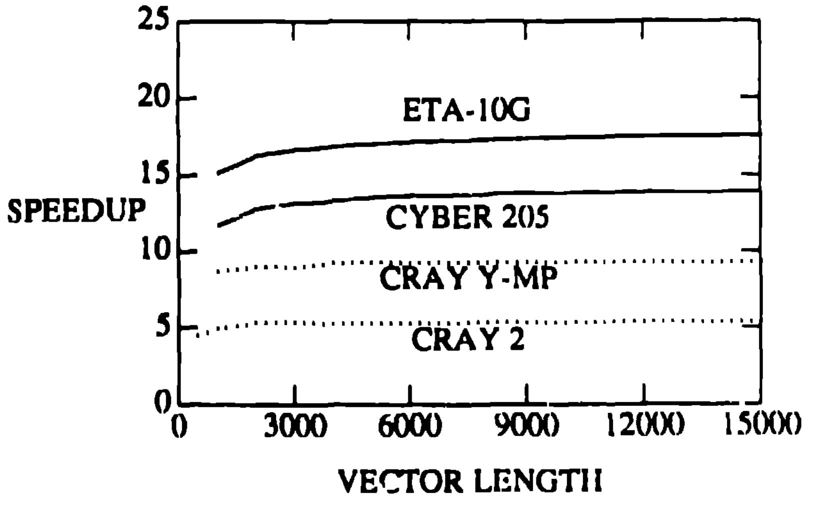


Figure 12

Solution Phase Speedup vs. Vector Length

To calculate speedups, we divide the scalar solution phase timings (asymptotic values are given in Table 1) by those timings in Figure 11 to yield Figure 12. Table 1 lists execution times for the original scalar algorithm and the revised vector algorithms. The execution times of the vector algorithm for the CRAY Y-MP use the SCILIB routines for index generation. Table 1 also shows the asymptotic speedups, calculated by dividing the solution-phase execution time of the original scalar code by the best times of the vector algorithms. The asymptotic vector times in Table 1 for the ETA10 and Cyber 205 machines are for a vector length of 40,000 - the largest run that would fit within the 1.7 million words of available memory on the Cyber 205.

Table 1. Execution Times and Speedups for the *Gamteb* Benchmark

Machine	Clock (ns)	Scalar Time (secs)	Vector Time (secs)	Speedup
ETA10-G	6.95	6.11	0.33	18.5
ETA10-E	10.5	9.20	-	
Cyber 205	20.0	13.40	0.90	14.9
CRAY Y-MP/832	6.17	3.9	0.42	9.20
CRAY-2	4.2	5.99	1.12	5.35

In the remainder of this discussion we concentrate entirely on the performance of the ETA-10G and CRAY Y-MP. Despite the 11% difference in cycle times between these two machines, the scalar code runs about 36% faster on the Y-MP. However, on the vector version of the code, the ETA-10G is the faster machine by 21%. On both machines, the times are nearly invariant with initial vector length, differing by less than 10% over the range of vector lengths from 1,000 to 15,000. Also, the execution times for both scalar and vector algorithms are strictly linear with the number of source emissions.

Note from Table 1 that there is an approximately 30.5% penalty in scalar performance of the ETA machines, when corrected for clock rate. Speedups are almost constant at about 14 and 18 for Cyber 205 and ETA architectures, respectively, and about 9 and 5 for CRAY Y-MP and CRAY 2 hardware, respectively. Due to their poorer scalar performance, the speedups are greater on the ETA machines than on the Cyber 205. Taking $V/S = 12$ as representative of CRAY Y-MP hardware, and $V/S = 25$ as representative of ETA hardware (ETA hardware, with relatively slower scalar performance, has a higher V/S ratio than for the Cyber 205), we have calculated "best fit" values in the modified Amdahl's Law of $p = 0.995$ and $D = 0.234$.

We have measured the vectorization levels of *Gamteb* on the ETA10-G and the CRAY Y-MP [31] using the hardware performance monitors. The percent vectorization from this method on the Cray is equal to the number of vector operations divided by the total number of operations. On the Y-MP/416, this value is 93.0%. However, the percent vectorization required by Amdahl's law is really based on time, not operation counts. Additionally, the Cray can overlap some scalar operations with some vector operations, so the percent vectorization measured by operation counting may not be accurate. We have also used another method [32], which is based on measuring, in scalar mode, the amount of time spent in vectorizable loops. When done in this manner, the percent vectorization on the Y-MP is >99%. On the ETA10-G, the algorithm spends 86% of the time in the vector hardware. At a speedup ratio of 25, this yields a value of $p = 0.993$.

Using these values of percent vectorization in Equation (1), we can calculate values of D based on the speedups we observed for *Gamteb*. The values of D range from 0.2 to 0.3 and are in good agreement with the values of D we obtained above. This suggests that the vectorized versions of *Gamteb* have effectively amortized the data motion over large numbers of floating point operations.

These speedup results are in the range of those reported in References 13 and 14, and 17 through 19, for vectorized photon tracing routines in evacuated enclosures. It is our belief, based upon speedup results for a variety of sizes (numbers of surfaces) presented in Reference 17, that this range of speedups would apply with some degradation to more complex simulations with greater numbers of surfaces.

6. CONCLUSIONS

Vectorization of Monte Carlo on the ETA-10 is best accomplished using bit vectors because their generation and manipulation is robust and effective. The disadvantage of this method is that the bit vectors must be repeatedly tested throughout the code and loop lengths do not decrease in inner nested conditionals. On the Cray systems, Monte Carlo vectorization is best carried out using gather/scatter operations, because vector lengths do decrease in inner loops. This is in spite of the overhead associated with index generation, which becomes ameliorated by amortizing the overhead over many floating point operations. The modified version of Amhdahl's Law effectively shows how the data motion needed to organize the computation in a Monte Carlo algorithm affects the overall performance.

Acknowledgement

We are grateful to ETA Systems and to Colorado State University for the provision of computer time. We also appreciate the vision of Mr. Jim McCauley who supplied the initial motivation for the effort, and Mr. Steve Simonds, for performing the runs on the ETA10-G.

References

1. N. Metropolis, "The Beginning of the Monte Carlo Method," *Los Alamos Science, 15*, pps. 122-143 (1987).

2. S. Ulam, R. D. Richtmeyer and J. Von Neumann, "Statistical Methods in Neutron Diffusion," LAMS-551, Los Alamos National Laboratory report (1947).

3. S. Ulam and N. Metropolis, "The Monte Carlo Method," J. Am. Stat. Assoc., 44, 335(1949).

4. J. H. Curtis, "Sampling Methods Applied to Differential and Difference Equations," *Proceedings IBM Seminar on Scientific Computation, IBM Corporation, New York, NY, pps. 87-109 (November 1949).*

5. J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods*, Meuthen, London (1964).

6. J. R. Howell, "Application of Monte Carlo to Heat Transfer Problems," *Advances in Heat Transfer, 5*, Academic Press, New York, NY (1968).

7. K. Binder, *Applications of the Monte Carlo Method in Statistical Physics*, Springer-Verlag, Berlin (1984).

8. N. Metropolis, "Monte Carlo: In the Beginning and Some Great Expectations," *Monte Carlo Methods and Applications in Neutronics, Photonics and Statistical Physics*, Cadarache Castle, France (1985).

9. A. Haji-Sheik, "Monte Carlo Methods," *Handbook of Numerical Heat Transfer*, John Wiley & Sons, Inc., New York, NY, pps. 673-722 (1988).

10. T. B. Brown, *Vectorized Monte Carlo*, PhD Dissertation, Department of Nuclear Engineering, University of Michigan (1981).

11. W. R. Martin, P. F. Nowak and J. A. Rathkopf, "Monte Carlo Photon Tracing on a Vector Supercomputer," *IBM Journal of Research and Development, 30*, No. 2 (March 1986).

12. Sequent Computer Systems, "Parallel Ray Tracing Study," TN-85-09 (rvp), Rev. 1.0 (1985).

13. F. W. Bobrowicz, J. E. Lynch and K. J. Fisher, "Vectorized Monte Carlo Photon Transport," *Parallel Computing, 1*, (1984).

14. F. W. Bobrowicz, K. J. Fisher, and R. G. Brickner, "Vectorized Monte Carlo Neutron Transport," Los Alamos National Laboratory report LA-UR-84-1269 (1984).

15. J. Briesmeister, ed., "MCNP: A General Monte Carlo Code for Neutron and Photon Transport," Los Alamos National Laboratory report LA-7396-M, Rev.2, September 1986.

16. M. L. Simmons and H. J. Wasserman, "A Performance Comparison of the CRAY-2 and CRAY X-MP/416 Supercomputers," Proceedings of the IEEE Supercomputing '88 Conference, IEEE Society, Washington, D.C., 1989.

17. D. V. Pryor and P. J. Burns, "A Parallel Monte Carlo Model for Radiative Heat Transfer," *Presented at the 1986 SIAM National Meeting*, Boston, MA (July 21-25, 1986).

18. P. J. Burns and D. V. Pryor, "Vector and Parallel Monte Carlo Radiation Heat Transfer," to appear in *Numerical Heat Transfer*.

19. J. Maltby, "Vectorization of MONT3D," *GS511 Final Project Report*. Department of Mechanical Engineering, Colorado State University, Fort Collins, CO (May 1987).

20. O. M. Lubeck, "Supercomputer Performance: The Theory, Practice and Results," *Adv. in Computers, 27*, 310(1988).

21. D. V. Pryor and P. J. Burns, "Vectorized Molecular Aerodynamics Simulation of the Rayleigh Problem," *Proceedings, Supercomputing 88*, Orlando, FL (Nov. 14-18, 1988).

22. P. J. Burns and D. V. Pryor, "Vector and Parallel Considerations for the Rayleigh Problem in Molecular Gas Dynamics," *Proceedings, 7th International Conference on Finite Element Methods in Flow Problems*, Huntsville, AL (April 3-7, 1989).

23. D. B. Heifetz, "Vectorizing and Macrotasking Monte Carlo Neutral Particle Algorithms," Princeton Plasma Physics Laboratory, PPPL-2427 (April 1987).

24. G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large-scale Computing Capabilities," *Proceedings of the American Federation of Information Processing Societies, Vol. 30*, Washington, DC, pps. 483-485 (1967).

25. Control Data Corporation, *CDC CYBER 200 Model 205 Computer System, Hardware Reference Manual*, Control Data Corporation Publications and Graphics Division, Sunnyvale, California (1981).

26. H. J. Wasserman, "Los Alamos National Laboratory Computer Benchmarking 1988," Los Alamos National Laboratory Report LA-11465-MS (1988).

27. R. W. Hockney and C. R. Jesshope, *Parallel Computers 2*, IOP Publishing, Ltd, Philadelphia, PA, (1988).

28. Control Data Corporation, *CDC CYBER 200 Fortran Version 2 Reference Manual*, Control Data Corporation Publications and Graphics Division, Sunnyvale, California (1981).

29. Cray Research, Inc., *CRAY X-MP Computer Systems Four Processor Mainframe Reference Manual*, Publication HR-0097, Mendota Heights, MN (1986).

30. Cray Research, Inc., *UNICOS Libraries, Macros and Opdefs Reference Manual*, Publication SR-2013, Cray Research, Inc., Mendota Heights, MN (1987).

31. R. Koskela, "Measurement of the Scientific Workload at Los Alamos National Laboratory Using the CRAY X-MP Hardware Performance Monitor," manuscript in preparation.

32. O. M. Lubeck and H. J. Wasserman, "Measurement of Vectorization Level in Fortran Programs," manuscript in preparation.

APPENDIX A

A Note on the Pseudo-random Number Generator

In this appendix we describe a problem encountered in converting this code to its vector version. This is really an aside to the central issue of constructing a good algorithm-architecture match, but an important aside nonetheless.

A critical factor in determining whether the scalar algorithm has been correctly mapped to the vector architecture is the accuracy and convergence properties of the answers. Indeed, we observed that the results of those bundles transmitted, backscattered, and escaped for the vector and scalar algorithm converged to different values. The results from the vector algorithm were close to those of the scalar but were definitely different -- especially in the values which are small. This was disturbing to us, and for some time we locked for an error in the vector algorithm. Unable to find an error, we eventually turned our attention to the random number generator.

In the vector algorithm, we duplicated, with fully vectorized code, the function of the random number generator supplied in the scalar code (referred to as the *Gamteb* random number generator). Our routine produced precisely the same sequence of random numbers as did the scalar random number generator. However, the vector algorithm accesses the random numbers in different order, as vector operations require a sequence of random numbers for a single purpose (e.g., determination of flight distance), whereas in the scalar algorithm, two successive random numbers are used for different purposes (e.g., determination of flight distance, then Russian Roulette).

To test the random number generator, we focused our attention solely on the scalar code with the *Gamteb* random number generator. We generated answers using the original sequence of random numbers for a variety of number of trials until we had achieved convergence. We next generated answers using every other random number in the sequence, and then every third. In Table A.1, the converged quantities of the ratio of bundles backscattered, escaped and transmitted is shown for these three cases along with the "true" answers (obtained from averaging the answers from the "corrected" vector and scalar codes for one million emissions).

Table A.1.
Converged Values for Problem Solutions

Case	Backscattered	Escaped	Transmitted
Every	0.0630	0.8067	0.2748
Every Other	0.0161	0.8368	0.2357
Every Third	0.0112	0.8489	0.2262
"True"	0.0118	0.8592	0.2173

The original random number generator in the scalar code is the following algorithm:

$$X_i = 3X_{i-1} \; mod \; 2^{22} \; ; X_0 = 2^{21} + 5.$$

This generator has the property of a full cycle: $2^{22} - 1$ numbers (see Knuth[A.1] for more discussion), but is deficient in the sense of having a high serial correlation. Consider the first few random numbers generated (seen as binary integers):

```
X0:     100000000000000000101
X1:     100000000000000001111
X2:     100000000000000101101
X3:     100000000000010000111
...     ...
```

Thus the first ten or so of the scaled random numbers are all approximately equal to 0.5. A similar "run" occurs when the scaled values get close to 1.0 and 0.0. In general, such behavior makes for a poor random number generator, and can lead to problems such as the one described above. (3 is simply not a good multiplier; it does not sufficiently shift and add the seed value.) A number of good random number generators are available [A.1],[A.2], and any of the linear congruential type can be fully vectorized.

We bring up this issue for two reasons. First, one must choose a random number generator with some care. The nature of Monte Carlo techniques is such that their "correctness" is very difficult to establish. (We speak in practical terms, since no computer program is provably correct.) For any given simulation, a large number of results will appear to be correct, and differences between the results of multiple trials will naturally be seen. Errors due to faulty generation of random numbers may easily masquerade as legitimate statistical difference. Second, the problem is compounded when one enters the vector and parallel arenas. In these situations, subtle errors may not show up until the problem is scaled up to a size which makes it impossible to "check" by any other method; or until some asynchronous (and nonrepeatable) sequence of events occurs. Given these difficulties in the vector and parallel regime, a faulty random number generator may be practically impossible to detect.

Appendix References

A.1 D. E. Knuth, *The Art of Computer Programming*, 2d ed., Vol. 2, Addison-Wesley, 1981.

A.2 S. K. Park and K. W. Miller, *Random Number Generators: Good Ones are Hard to Find*, Communications of the ACM, Vol. 31, No. 10, October, 1988.