Unified Stochastic Simulations for Vector Machines: Empirical Results

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Ling-Yu Chuang

Aditya P. Mathur
Purdue University, apm@cs.purdue.edu

Report Number:
92-030
UNIFIED STOCHASTIC SIMULATIONS
FOR VECTOR MACHINES:
EMPIRICAL RESULTS

Vernon J. Rego
Ling-Yu Chuang
Aditya P. Madur

CSD-TR-92-030
May 1992
UNIFIED STOCHASTIC SIMULATIONS FOR VECTOR MACHINES: EMPIRICAL RESULTS *

Vernon J. Rego † Ling-Yu Chuang, and Aditya P. Mathur
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

We investigate the efficacy of a recently proposed vectorization and concurrentization scheme known as Program Unification. The technique entails a source-to-source transformation of code designed for vector machines. As a pre-compilation technique, unification is applicable to programs which satisfy two criteria, namely, poor vectorizability and a need for repeated execution. In this study we present a set of experimental results on program unification in the stochastic simulation domain. The intent is to demonstrate how a variety of simulation applications including discrete-event simulations (e.g., queueing systems), Monte Carlo simulations (e.g., estimation of multidimensional integrals), and statistical simulations (e.g., computing distributions of statistics) lend themselves to unifying schemes with encouraging utilization and speedup characteristics. All experiments reported here were conducted on an Alliant FX/8.

---

*This research was supported in part by the National Science Foundation, under grant No. ASC-9002225.
†Supported in part by Mathematical Sciences Section of Oak Ridge National Laboratory under contract DE-AC05-84OR31400 with Martin Marietta Energy Systems, Inc.
1 Introduction

In recent years, the proliferation of more easily accessible high performance computing systems has been accompanied by varied and imaginative efforts towards creating algorithms that exploit the strengths of specific architectural characteristics of these computing systems. Issues governing these efforts involve certain key notions including the ease of use, cost of implementation, speedup to be had, and efficiency of a given algorithm on a given architecture. This phenomenon is particularly noticeable in the compute-intensive applications in the physical sciences, where vector machines like the CRAY Y/MP, NEC, Alliant FX, and SIMD machines like the CM-2 and MasPar are used as computational and modeling tools. In this context an application domain of particular importance is stochastic and numerical simulation, for example, Monte Carlo simulation [22, 38, 39]. As computing speeds continue to grow and processing costs continue to decrease, it is likely that large scale scientific experiments based on simulation will evolve into an important supporting paradigm supplementing theory and experimentation [4].

In this paper we are concerned with the class of SIMD machines which utilize vector instructions. In particular, we are interested in exploiting this vectorization feature for executing a large class of simulation models, known as **concurrent stochastic simulations** [32], with enhanced machine utilization and speedup. The underlying basis for improved performance is a technique known as **Program Unification** that was proposed and developed earlier [28, 31], though mainly in a theoretical view. The motivation for exploring the use of this technique empirically is two-fold. First, there will always be a need for code that works efficiently on vector multiprocessors and cost-effective mechanisms for obtaining such code. Hence, one exploratory aspect of this work involves a search for heuristics which can effect a beneficial program transformation. Given that the utility of program unification as a transformation has been demonstrated theoretically Mathur90,RegoOct90, such heuristics would be a valuable practical aid in automating the transformation. Second, because it is generally easier to demonstrate a concept in a well-understood framework, we decided to restrict ourselves to the manageable but rich and important class of applications known as stochastic simulations [33].

We define a stochastic simulation model to be **concurrentizable** if either independent or dependent replications of the model can be executed concurrently. As a simple example in the independent case, consider a simulated annealing model [1] operating on a domain $D$. If the domain $D$ can be partitioned into disjoint subdomains, say $D_1, D_2, \ldots, D_N$, where $D = D_1 \cup D_2 \cup \ldots \cup D_N$, replications of the same model can be made to execute independently on independent processors or sets of processors with the intention of determining that specific region $D_k$ containing the global optimum. Observing that each subdomain is being operated on independently, and in parallel, this kind of concurrency is aptly termed data parallelism [15].

Dependence between processors can be induced in the following manner. In looking for the global optimum in $D$, processors can be required to exchange information about their domains, converge on the “best” subdomain $D_j$, subdivide it for data parallelism, and then repeat the annealing algorithm on their respective subdivisions of domain $D_j$. This algorithm would terminate either when a certain cost has been exceeded or when the optimum has been found. In both the independent as well as the dependent case, the replicated simulations execute concurrently. This concurrentizing strategy works equally well for a variety of stochastic simulations, as can be seen in benchmarks presented here.
2 Related Work

A review of the literature suggests that there are currently two views of high-level parallelism for simulation applications, each view based on a separate, architecture-dependent paradigm. In one view, the simulation application is tailored for execution on a vector multiprocessor (such as the CRAY Y-MP), or a SIMD multiprocessor (such as the CM-2). Here, major emphasis is placed on the machine's compiler or supporting software libraries for enhanced performance. In the other view, the simulation application is mapped onto an arbitrary multiprocessor, with concurrency given more emphasis than hardware vectorization. This approach is not well understood in the general sense, since it entails a mapping of an arbitrary application to an arbitrary multiprocessor. Since model decomposition usually requires detailed application-dependent knowledge, there is an increasing amount of attention being paid to techniques for efficient decomposition.

In both views, the analyst uses simulation as a modeling tool mainly because a purely analytic solution is either unavailable or is computationally infeasible; and in both views, the two most important criteria for assessing the performance of the simulation model have traditionally been correctness and execution speed. In the recent past, however, the combination of improved machine accessibility and the complexities of newer and powerful architectures with hundreds or thousands of independent or dependent processors have brought another important performance criterion for simulation models to the analyst's attention, namely, the ease of model development. It is increasingly likely that a researcher who uses simulation as a modeling tool in some particular area of interest will also take the amount of time and degree of effort involved in developing his or her models on a particular architecture into account before finally choosing an underlying architecture as a computational workbench.

Simulation analysts tend to fall into either one of two broad categories: the Event-oriented analysts and the Sample-oriented analysts. The Event-oriented analysts tend to focus on discrete-event dynamic systems, using one of a variety of simulation techniques such as event scheduling, process interaction or activity scanning [12] in their studies. For example, performance analysts interested in computer network and computer system performance [27] routinely construct queueing network simulation models using one or more of the above techniques as a matter of course. The Sample-oriented analysts tend to focus on static systems, being primarily interested in the rapid generation of independent samples from a model dependent distribution. For example, a researcher in statistical pattern-recognition [21] who is interested in the distribution of a particular statistic in multi-dimensional space is interested in generating many samples in as short a time as possible, and in as effortless a manner as possible. Another well-known class of sample-oriented methods is the class of Monte-Carlo methods [22]. The Sample-oriented view can be regarded as a more general view of simulation since many discrete-event problems may be put in a sampling framework. A good example of this is regenerative simulation [8], where independent and identically distributed samples are constructed during the execution of a discrete-event model.

In the Sample-oriented view, the focus has traditionally been in the use of vector supercomputers to speed up the computation of primary estimates [5, 6, 38]. In essence, samples or sample-paths (trajectory) are generated repeatedly through a stochastic mechanism; vectorization aids in speeding up the execution of vector instructions and loops. Because of this, the technique is sometimes called vector simulation or stack simulation. The general tendency is to expect the machine's compiler to do a significant amount of optimization, thus relieving the analyst of programming details as long as he or she has attempted to
follow simple rules such as, for example, in the exploitation of loops and independence. The success of this approach can be seen in the proliferation of CRAY supercomputing sites, and in the growing number of CM sites in the country.

In the Event-oriented view, a simulated sample path develops stochastically as a sequence of discrete events from a finite set of event types [12]. Using one of a variety of protocols, this sequence is carefully decomposed and made to execute on a multiprocessor system so that speedups result. This approach is known as parallel discrete-event simulation, or distributed simulation [17]. While the vector simulation view largely ignores explicit use of multiprocessing power, except through loop-level scheduling, the parallel discrete-event simulation view largely ignores vector supercomputing power. In an effort to bridge this gap, Rego and Sunderam [32, 37] experiment with the concurrent stochastic simulation view, which encompasses both the event-oriented view and the sample-oriented view. While the experiments reported in [32] involve only fast scalar machines, they are just as appropriate for vector machines.

It is well known that Monte Carlo simulation programs perform inefficiently on vector processors or vector multiprocessors, unless the code is specially tailored for use on these machines [6, 40]. A considerable amount of effort has been expended in developing vector code or enhancements for Monte Carlo simulations [5, 6, 38]. In the discrete-event simulation area we are only aware of the effort by Chandak and Browne [7].

We are not aware of any tool that incorporates some or all of these techniques and performs an automatic transformation of the original Monte Carlo or discrete-event simulation code into vectorizable Monte Carlo or discrete-event simulation code, respectively. There is an apparent lack of a general methodology for obtaining improved performance through speedup and efficiency for both Monte-Carlo and discrete-event simulations in an architecture independent setting, particularly for vector/SIMD machines. Apart from a discussion on the use of vector machines for discrete event simulations by Chandak and Browne [7], we are unaware of research addressing discrete event simulations on vector multiprocessors.

In the following section we present the program unification method through a series of examples. In Section 3.1, we present a segment of code that can be used to estimate \( \pi \). To demonstrate the utility of unification, Section 3.1 also contains an elaboration on how this code segment can be unified to obtain the estimate more efficiently. Since an understanding of the method hinges on understanding speedup through vectorization, Section 3.1 introduces the notion of program block speedup with the aid of three different kinds of program blocks and a useful quantity that we call the inverse block speedup coefficient [31]. Sections 3.2 and 3.3 contain parallel developments for a random walk problem, and a queueing problem, respectively. We present some empirical results in Section 4, and a brief conclusion in Section 5.

### 3 Unification: Its Application and Analysis

Program unification [31] is a technique for source-to-source transformation of sequential programs. A given sequential program \( P \) is transformed, at the source level, into another program \( P' \) such that a single execution of \( P' \) generates the same results that are had by executing \( P \) serially on a number of distinct data sets. More importantly, such transformed programs often provide significant speedups on vector processors and vector multiprocessors with little or no additional cost. In contrast to techniques that attempt to detect parallelism within a program, unification exploits the natural parallelism that arises when multiple
instances of the same program are to be executed on simultaneously available data sets.

A necessary condition for program unification to be useful is that the program under consideration be inherently poorly vectorizable. It is not difficult to find such applications in common use on vector machines, typically due to an analyst’s dependence on the sheer speed of machines like the CRAY Y/MP. Many typical Monte Carlo or discrete-event simulation programs are known to contain some of the characteristics listed in 1 through 5 and thus belong to this class of programs with poor vectorizability [40]. Kuck [24] gives examples of programs that perform poorly on vector machines. In Section 4 we present benchmarks to show that unification can be used to improve the performance of such programs on vector multiprocessors.

The use of program unification for stochastic simulation applications is best described through the use of examples. Below we present three examples, each of which uses a random number generator for the generation of uniform variates. So as not to detract attention from explanation of the unification technique we will assume the existence of a readily available set of sequences of uniform random numbers. That is, a real valued function \textsc{Random} (\textsc{Seed}) takes as input a floating point variable \textsc{Seed} and returns a random floating point number in the interval (0,1). The variable \textsc{Seed} is changed in the process. Hence, given a seed \textsc{D}(1), say \textsc{D}(1) = 1234567, the function \textsc{Random} (\textsc{D}(1)) may return the value 0.3164093 while simultaneously changing the value of the seed \textsc{D}(1) to some other number, say 6186239. We will assume the existence of a set of seeds \textsc{D}(1), \textsc{D}(2), \textsc{D}(3) ... etc., which give rise to independent sequences of random numbers. Such sequences are obtained through parallel random number generators. A variety of techniques for generating independent random numbers efficiently in parallel are available in the literature [10, 18, 19, 25].

### 3.1 Estimation of \( \pi \)

In this example we demonstrate how the unification method is used in estimating the value of \( \pi \) through parallel sampling. A simple method to estimate the value of \( \pi \) entails randomly sampling some number of pairs of real numbers, say \( M \), from the interval [-1,1]. An estimate is then made of the fraction of pairs satisfying a specified property. It is important to note that the number \( M \) of samples is fixed prior to execution, and two random variates are required for the generation of each sample. Suppose that the random vector \((u(1), u(2))\) is uniformly distributed in a square of area 4 centered at the origin, as shown in Figure 3. Inscribed within the square shown in Figure 3 is a circle, with the origin as its centre. It can be seen that
1. \[ \text{SUM} = 0 \]
2. \[ \text{DO} \ 100 \ I = 1, M \]
3. \[ U(1) = \text{RANDOM}(D(1,2*I-1)) \]
4. \[ U(2) = \text{RANDOM}(D(1,2*I)) \]
5. \[ \text{POINT} = (2*U(1) - 1)^2 + (2*U(2) - 1)^2 \]
6. \[ \text{IF} (\text{POINT} \geq 1) \text{SUM} = \text{SUM} + 1 \]
7. \[ 100 \text{ CONTINUE} \]
8. \[ \text{AREA} = \text{SUM} / M \]
9. \[ \pi_{\hat{\text{HAT}}} = 4 * \text{AREA} \]

Figure 2a Nonunified code: Estimation of $\pi$

\[
P\{ (u(1), u(2)) \text{ is in the circle} \} = P\{ u(1)^2 + u(2)^2 \leq 1 \} = \frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi}{4} \quad (3.1)
\]

so that with the generation of a large number $M$ of pairs $(u_j(1), u_j(2))$ of uniform random variates representing points falling within the square, one can obtain the set

\[
A_M = \{ j \leq M | (2u_j(1) - 1)^2 + (2u_j(2) - 1)^2 \leq 1 \} \quad (3.2)
\]

of points falling within the circle. From the law of large numbers [23]

\[
\lim_{M \to \infty} |A_M| = \pi / 4 \quad (3.3)
\]

suggesting that for sufficiently large $M$, $\hat{\pi} = 4 |A_M|$ is a reasonable estimator of $\pi$. It can be shown that $\hat{\pi}$ is an unbiased estimator of $\pi$. A simple piece of code to estimate $\pi$ using the above strategy is outlined in Figure 3.1.

Given that $M$ samples are sufficient for estimating $\pi$ to some acceptable degree of statistical accuracy, the code segment in Figure 2a can be explained as follows. The loop in Lines 2 through 7 generates these $M$ samples by transforming the $M$ pairs of uniform random variates \{ $U(1)$, $U(2)$ \} from the interval $(0,1)$ into $M$ points inside a square of area 4, centered at the origin. A count is made of the number of such points falling within the unit circle, centered at the origin. The fraction of points falling within the circle gives an estimate of the area of the circle, relative to the area of the square; this estimate is made at Line 8. Finally, the required estimate $\hat{\pi}$ is made at Line 9. This simple sampling scheme works because each iteration of the loop in Lines 2 through 7 generates an instance of a Bernoulli random variable with parameter $\pi/4$.

It is not difficult to see that there is little in this code that can be vectorized by the smartest compiler. Except for statements 3 and 4, which can be executed concurrently, the remaining code is strictly sequential. Since the key factor in obtaining the estimate in Line 9 depends on $M$ independent samples, one will readily see that it would make sense to attempt to obtain a number $N$ of such samples in parallel on a vector machine, where $N$ is either the vector size or some multiple of the vector size of the host machine. Assuming that $M$ is divisible by $N$, the loop index will now only have to be $M/N$ instead of $M$. Thus, in order to enforce vectorization and still obtain an equivalent estimate for $\pi$, we can resort to
1. \[ \text{SUM} = 0 \]
2. \[ \text{DO 50 I = 1, N} \]
3. \[ \text{PSUM(J)} = 0 \]
4. \[ \text{50 CONTINUE} \]
5. \[ \text{DO 200 I = 1, M/N} \]
6. \[ \text{DO 100 J = 1, N} \]
7. \[ U(1,J) = \text{UNIFORM}(D(J,2*I-1)) \]
8. \[ U(2,J) = \text{UNIFORM}(D(J,2*I)) \]
9. \[ \text{POINT}(J) = (2*U(1,J) - 1)^2 + (2*U(2,J) - 1)^2 \]
10. \[ \text{IF (POINT}(J) \leq 1) \text{ PSUM}(J) = \text{PSUM}(J) + 1 \]
11. \[ \text{100 CONTINUE} \]
12. \[ \text{200 CONTINUE} \]
13. \[ \text{DO 300 J = 1, N} \]
14. \[ \text{SUM} = \text{SUM} + \text{PSUM}(J) \]
15. \[ \text{300 CONTINUE} \]
16. \[ \text{AREA} = \text{SUM} / (M/N) \]
17. \[ \pi\_\text{HAT} = 4 \times \text{AREA} \]

Figure 2b Unified code: Estimation of $\pi$

unifying N instances of the above program. Application of the unification algorithm \[31\] for repeated sampling of N pairs of samples is done by the piece of code shown in Figure 2b.

The unified code in Figure 2b is explained as follows. Instead of generating a single point at a time (as in Lines 2 through 7 of Figure 2a), the code in Lines 6 through 11 of Figure 2b generates N independent points concurrently. The outer loop, at Line 5, ensures that this procedure is repeated so that a total of N points is obtained. Because the unified program consists of N independent program components, a total of N partial sums must be available, initialized in Lines 2-4, and updated in Line 10. A grand sum is obtained in Line 14, where all N components pool their counts into a single count, and this is used to estimate $\pi$ at Line 17.

Block Speedup Coefficients

A reader familiar with vector multiprocessors will easily recognize the advantage of using the code in Figure 2b for obtaining the required estimate of $\pi$. For a reader less familiar with the use of such machines, understanding why the code in Figure 2b gives speedup over that in Figure 2a requires an appreciation of vector instruction execution \[36\]. Assume the existence of a mask vector (array) PV[ ] of k elements, \(1 \leq k \leq N\), where N is a large number, typically some multiple of the word size of the underlying vector machine. At any given step in a unified program's execution sequence, PV[I] = TRUE for any program component I belonging to a currently active set of executing components, and PV[J] = FALSE if component J is currently inactive, for \(I \neq J\). The array PV[ ] is used to control the distinct execution threads of a unified program with k components, allowing a component I with PV[I] = TRUE to execute while a component J with PV[J] = FALSE remains idle until such a time as the unified program's control sets PV[J] = TRUE. A detailed description of the mask vector can be found in \[31\].

In order to demonstrate the notion of program block speedup, assume a mask vector PV[ ] that remains fixed for the duration of a unified program's execution. For example we may assume that PV[I] = FALSE if I is odd, and PV[I] = TRUE if I is even, for \(1 \leq I \leq N\), meaning that only those program components with even indices will remain active throughout the unified program's execution. Correspondingly, all program blocks with
1. DO 100 I = 1, N
2. A = B * C * I
3. 100 CONTINUE

Figure 3a Nonunified block: Assignment with Multiplication

1. DO 100 I = 1, N
2. IF ( PV(I) ) A(I) = B(I) * C(I) * I
3. 100 CONTINUE

Figure 3b Unified block: Assignment with Multiplication

odd indices will remain inactive throughout the unified program's execution. It should be clear that masking out instruction execution for inactive blocks results in some inefficiency, possibly countering the gains to be had through vectorization. Alternately, if PV[I] = TRUE for each program component I in the unified program, the resulting code can be expected to be more efficient. We will elaborate on this point shortly, when presenting some benchmarks.

Define a program block to be a sequence of consecutive statements in which control flow enters via the first statement and leaves via the last without any possibility of leaving at any intermediate statement. Using this definition, a given program may always be partitioned into a distinct number of such blocks [3]. For example, the code segments in Figure 2a and Figure 3a are program blocks.

Consider the code shown in Figure 3a depicting a simple program block in which a variable A is obtained as a product of variables B, C, and I. In order to reproduce the effect of some number, say N, of such multiplications, with possibly different values for the three variables in each case, the multiplication is done within a loop. This code essentially simulates the serial execution of a program block comprising the single statement at Line 2. Note that the compiler directives $NOVECTOR and $NOCONCUR are used to ensure inhibition of vectorization and concurrentization. In concert with the use of scalar I, this is done to remove any optimizations performed by a smart compiler which may detect that A and B are unused program constants. Any such optimization would defeat our ability to show that unification yields speedup through improved vectorization and concurrentization.

Next, consider the alternative arrangement shown in Figure 3b, where the simple sequential multiplication with assignment is performed as the multiplication of vectors B[], C[], and a scalar I. While the code in Figure 3a represents the sequential execution of N programs, the code in Figure 3b represents the execution of an N-component unified program. A single execution of the latter program yields all the results to be had by the N serial executions of the former.

On a vector multiprocessor, compilation of the unified code in Figure 3b will allow both vectorization as well as concurrentization, while the sequential code in Figure 3a will yield neither. Prior to execution of the loop in Figure 3b, the vector PV[] allows the machine to set up a mask register. Each component I of PV[] satisfying PV[I] = TRUE will be enabled in the mask register, while the others will be disabled. Regardless of the
Figure 4a Nonunified block: Chaining

1. DO 300 I = 1, N
2. \( X = Y \times Z \)
3. \( W = X + Q + I \)
4. 300 CONTINUE

Figure 4b Unified block: Chaining

1. DO 300 I = 1, N
2. IF ( PV(I) ) THEN
3. \( X(I) = Y(I) \times Z(I) \)
4. \( W(I) = X(I) \times Q(I) + I \)
5. ENDIF
6. 300 CONTINUE

The contents of vector PV[], the product in Line 2 of Figure 3b is computed in a vector pipe for all program components, though the assignment to vector A[] is made only for those components enabled in the mask register. At this stage it is natural to question the benefit of performing the vector computation Figure 3b as opposed to the sequential computation in Figure 3a.

Let \( T_S \) denote the average time required to execute the assignment statement in Line 2 of Figure 3a, and let \( T_{U(N)} \) denote the average time required to execute the \( N \)-component unified code segment in Lines 1 through 3 of Figure 3b. The \textit{inverse block speedup coefficient} \( \alpha_Z(N) \) for a given program block is defined [31] as

\[ \alpha_Z(N) = \frac{T_{U(N)}}{NT_S} \]  

where \( N \) is the number of sequential executions of the assignment statement in Line 2 of Figure 3a, and \( Z = \text{Assignment with Multiplication} \) is used to identify the particular block under consideration. For reasonable values of \( N \), \( 0 < \alpha_Z(N) \leq 1 \), where smaller values of \( \alpha_Z(N) \) imply more efficient use of vector pipes and hence improved speedup performance. We define \( \gamma_Z(N) = 1/\alpha_Z(N) \) to be the block speedup coefficient obtained through the \( N \)-component unification of a given block \( Z \). In Figure 3c can be seen a pile of three plots of \( \gamma_Z(N) \) versus \( N \) for three different cases. For the graph on the bottom of the pile, \( PV[I] \) is TRUE for all program components I. For the graph in the middle of the pile, \( PV[I] \) is TRUE if I is odd; otherwise \( PV[I] \) is FALSE. For the graph on the top of the pile, \( PV[I] \) is not used. The assignment is done for all \( N \) program components without using a mask vector, as is done in Lines 3,7,8,9 etc. of Figure 2b. In Figure 3c, the graph at the top of the pile exhibits largest block speedup. This is to be expected because \( PV[I] \) is unused, and the vectorization is at peak performance. Because of the coupled execution of the Alliant's processors, speedup improves well with an increasing number of processors. The bottom two graphs depict a less readily apparent phenomenon. Whether only half the mask vector is true, (i.e., the graph in the middle), or the entire vector is true (i.e., the graph at the bottom), speedup remains roughly the same. This suggests that vectorization may work
well even when program paths diverge.

For completeness, we include code segments for two other kinds of program blocks and corresponding plots of $\gamma(N)$ versus $N$ for these. In Figures 4a and 4b are shown non-unified and unified code segments, respectively, for a program block using chaining, with corresponding speedup coefficient plots in Figure 4c. In Figures 5a and 5b are shown non-unified and unified code segments, respectively, for a program block using one-dimensional array references, with corresponding speedup plots shown in Figure 5c. Though the program blocks are different, the graphs in Figures 4c and 5c exhibit the same behaviour as those in Figure 3c, giving more evidence to suggest that program path divergence need not be detrimental to a vector machine’s performance.

### Speedup for Estimation of $\pi$

Let $t_A$ denote the average time required to execute an assignment statement including either a condition or some computation, as in Lines 6 or 8 of Figure 2a. Also, let $t_R$ denote the amount of time required for the generation of a uniform variate, as in Lines 3 or 4. The average time required for executing the code code in Figure 2a is given by $T_S(\pi) = 2M(t_R + t_A) + 3t_A$.

Observe that $N$ serial executions of an instruction that takes 1 time unit to execute would require a total of $N$ units when executed in nonunified mode. Using Eq. (3.4), if $\alpha(N)$ is the inverse block speedup coefficient for this instruction when executed in unified mode, the time required for unified execution becomes $N\alpha(N)$. Using this reasoning, the average time required by the unified code in Figure 2b is given by $T_U(\pi) = 2M\alpha(N)C + (2N\alpha(N) + 3)t_A$ where $C = t_R + t_A$, and $\alpha(N)$ is the inverse block speedup coefficient of the block in Lines 2 through 5 of Figure 2a. Speedup given by the unified code is expressed as

$$
S(\pi) = \frac{T_S(\pi)}{T_U(\pi)} = \frac{2MC + 3t_A}{2M\alpha(N)C + (2N\alpha(N) + 3)t_A} = \frac{MC}{\alpha(N)(MC + N)}
$$

(3.5)
Recall that $N$ is fixed, and is typically equal to the vector size or a small multiple of the vector size of the host machine. In stochastic simulations, the sample size $M$ is generally a fairly large number, especially when sample variances are large and reasonable statistical accuracy is required of the final estimate. For sufficiently large $M$, we obtain

$$S(\pi) \simeq \frac{1}{\alpha(N)} = \gamma(N)$$

(3.6)

implying that in the absence of architecture related overheads due to cache size, resource contention, etc., speedup is roughly equal to block speedup, in the absence of a partition vector. Unified programs without partition vectors are possible only if there is no scope for data-dependent program path divergence; this makes such applications rather special. In the next two sections we will see that using a partition vector will increase the range of applications, but may also decrease potential speedup.

### 3.2 A General Random Walk

Consider the situation where $M$ is determined at run-time, instead of being fixed prior to program execution. Simply put, this means that sampling continues until some prescribed statistical accuracy is met by the estimate; the prescription is available prior to execution, but $M$ is determined via this prescription and the particular trajectory taken by the simulation (which depends on the random deviates generated) only during program execution. Thus $M$ is the total number of samples generated when the simulation terminates.

Let $\{W_j; j \geq 0\}$ be a sequence of independent and identically distributed (i.i.d) random variables with

$$P\{W_j = i\} = a_i$$

(3.7)

for $i = 0, \pm 1, \pm 2, \ldots$ and all $j \geq 0$. Note that $\{a_i; i = 0, \pm 1, \pm 2, \ldots\}$ is a given sequence of probabilities whose sum is unity. Define the partial sum

$$S_n = \begin{cases} 0 & n = 0 \\ \sum_{j=1}^{n} W_j & n > 0 \end{cases}$$

(3.8)

and observe that $\{S_n; n \geq 0\}$ is a Markov chain describing a general random walk [23] with transition probability matrix $P = [p_{i,j}]$, where

$$p_{i,j} = \begin{cases} 0 & j < i \\ a_{j-i} & j \geq i \end{cases}$$

(3.9)

for all $i, j \geq 0$. Given $W_0 = 0$, let $H_L$ be the (random) hitting time to certain boundary states, or

$$H_L = \min_n \{n \geq 0 | W_0 = 0, |W_n| \geq L\}$$

(3.10)

describing the number of steps required by the random walk to move above state $+L$ or below state $-L$. Suppose that we are interested in obtaining certain characteristics of the random variable $H_L$, such as for example, the expected hitting time $E[H_L]$. In order to obtain an estimate $\hat{H}$ of $E[H_L]$ we must simulate a sufficiently large number $M$ of independent random walks each of which gives us a realization of the random variable $H_L$; the estimate $\hat{H}$ is obtained by averaging these realizations.
1. $H = 0$
2. $M = 0$
3. 100 CONTINUE
4. CURRENT = 0
5. NSTEPS = 0
6. 200 CONTINUE
7. NEXT = GETNEXTSTATE(CURRENT)
8. CURRENT = NEXT
9. NSTEPS = NSTEPS + 1
10. IF ( ABS(CURRENT) .LE. L ) GOTO 200
11. $H = H + NSTEPS$
12. $M = M + 1$
13. $H_{HAT} = H/M$
14. IF ( NOT (SUFFICIENT($H_{HAT}$)) ) GOTO 100

Figure 6a Nonunified block: Random Walk

Given an arbitrary discrete probability density $\{a_j; j \geq 0\}$, there are several ways of generating random variates from this density. One convenient mechanism for doing this is the alias method [34] which makes use of the fact that a $k$-point discrete density can be represented as an equally weighted mixture of $(k - 1)$ distinct two-point densities. Thus, with some preprocessing, one obtains the mixture representation of the given input discrete density function; random variates from the input density function are then had by generating only two uniform variates - the first used to determine the index of the two-point density in the equally weighted mixture representation, and the second used to select one of the components of the indexed two-point density.

A single realization of the random walk is obtained by initially placing the walker in State 0. The state occupied by the walker at the current time is called the current state. The state to be visited next by the walker is called the next state. Given that the current state is State 0 initially, the simulation program must generate the next state of the walker, update the current state to that next state, and repeat this generation/update procedure until either one of the two boundaries is crossed. We will assume that an integer function GETNEXTSTATE(CURRENT) is available to generate a next state when called with a current state as input. Also, a boolean function SUFFICIENT($H_{HAT}$) utilizes a user-defined prescription to decide if the present estimate $\hat{H}$ achieves a given statistical accuracy. It is important to note that though this is not made explicit, the function SUFFICIENT($H_{HAT}$) may utilize other information, such as the variance of the estimator, in the computation of the value it returns. A code segment which can be used to estimate $E[H_L]$ is shown in Figure 6a.

Instead of generating a single realization of the random variable $H_L$ at a time, we can take the approach used in the unified estimation of $\pi$ and generate $N$ random walks in parallel. Thus, each program component $I$ will start in initial State 0 and, in general, will move to a distinct next state which depends only on $I$. It follows that the entire trajectory of a random walk taken by component $I$ will depend only on $I$ and will, in general, be different from a trajectory of component $J$, for $I \neq J$. Nevertheless, the $N$ program components will generate $N$ independent random walks which move from distinct current states to distinct next states in unison, and vector instructions can be used to efficiently generate these moves.

A simple scheme for unifying the random walk code is shown in Figure 6b. Since $N$ random walks are generated concurrently, the variables current, next, and nsteps are now $N$-element vectors. We replace the integer function GETNEXTSTATE(CURRENT) by a procedure
VECTOR_GETNEXTSTATE(N, CURRENT, NSTEPS) which takes as input the variable N, and vectors CURRENT[ ] and NSTEPS[ ]. Only when all N components have obtained their realizations of $H_L$ will control return from the procedure to the calling program, at Line 9 of Figure 6b. Following this, an estimate is made by pooling together and averaging the cumulative realizations of all program components.

Since the random walks generated by some components will yield their realizations of $H_L$ earlier than other components, the unified execution of the N random walks will continue until all program components have generated their realizations. As pointed out earlier (e.g., Fig 3b), the partition vector PV[ ] can be used to exclude components whose walks have already terminated from a vector computation. Naturally, a question of efficiency now arises. If all components terminate with roughly the same number of steps for their realizations $H_L$, little vector underutilization due to masking results. On the other hand, if all but a single component have terminated, and the last component takes a long while to generate its realization of $H_L$, significant underutilization results.

Speedup is obtained for the unified random walk if the variance of $H_L$ is not large, so that components finish within a reasonably small number of steps of one another inside routine VECTOR_GETNEXTSTATE( ). Speedup also depends on the manner in which estimation is being done; for example, the unified code in Figure 6b constructs an estimate $\hat{H}$ of $H_L$ by using an equal number of samples (realizations) from each component to avoid sampling bias [20, 32]. As a result, the run-time of the unified simulation will depend on the run-time of the component which takes the largest amount of time to complete all its sampling, where this time is a function only of the particular trajectory this component happens to generate. Variations of this sampling strategy will yield improved speedup if they are not affected by asymmetry in termination times for the unified components [20].

**Speedup for Estimation of $H_L$**

Let $t_A$ and $t_R$ be defined as before. We will assume that in the non-unified code shown in Figure 6a, control passes through the statement at Line 3 a total of $K_S$ times, so that a total of $K_S$ samples are constructed. The average time required by Lines 1 and 2 is $2t_A$. Recalling that two uniform random variates are generated each time Line 7 is executed, the average time required by Lines 7 through 10 is $[2t_R + 3t_A]$. Clearly, control passes through the statement at Line 6 an average of $E[H_L]$ times. Finally, Lines 4-5, and 11-14 contribute
an average of \(6t_A\) units towards run-time. Thus, the average time required by the entire non-unified code segment in Figure 6a is

\[
T_S(K_S) = K_S(6t_A + E[H_L](3t_A + 2t_R)) + 2t_A
\]  
(3.11)
time units. It is now left to determine the average time required by the unified code in Figure 6b.

Assume that control passes through the statement at line 3 of Figure 6b a total of \(K_U\) times. Given that each pass through the unified code yields \(N\) independent samples, a total of \(NK_U\) samples is obtained upon termination. Because the prescription for termination is the same in both cases and the samples being computed are independent, it follows that \(K_S \approx NK_U\). Because samples are now had in parallel, various schemes for reduction of variance [26, 41] can be utilized to ensure that \(K_U < K_S/N\).

Recall that procedure \texttt{VECTOR\_GET\_NEXT\_STATE()} in Line 8 of Figure 6b simulates a total of \(N\) random walks in parallel, returning control to the calling program only when all walks have terminated. Let \(H_1, H_2, H_3, \ldots, H_N\) be i.i.d random variables with the same law as \(H_L\). The number of iterations performed within procedure \texttt{VECTOR\_GET\_NEXT\_STATE()} is given by

\[
H_L^* = \max\{H_1, H_2, H_3, \ldots, H_N\}
\]  
(3.12)
so that this procedure contributes an average of \(2E[H_L^*]N\alpha_V(N)t_R\) time units towards run-time, where \(\alpha_V(N)\) is the inverse block speedup coefficient of the code in \texttt{VECTOR\_GET\_NEXT\_STATE()}.

In Figure 6b, the code in Lines 1-2 require \(2t_A\) time units. Following the reasoning used in the unified speedup estimation for \(1r\), each iteration of the code in Lines 5-6 and 10 require a total of \(3N\alpha_Q(N)t_A\) time units, where \(\alpha_Q(N)\) is the inverse block speedup coefficient for a block consisting of Lines 5, 6, and 10. On each of the \(K_U\) passes through the unified code, Lines 12-14 contribute a total of \(3t_A\) time units. Therefore, the average time required by the unified code is given by

\[
T_U(K_U) = K_U(3N\alpha_Q(N)t_A + 3t_A + 2E[H_L^*]N\alpha_V(N)t_R) + 2t_A
\]  
(3.13)
using which we may now compute speedup. Since \(6K_ST_A > 3K_Ut_A\), the speedup ratio can be simplified and expressed as

\[
S(\hat{H}_L) > \frac{K_S E[H_L](3t_A + 2t_R)}{K_U \alpha(N)(3t_A + 2E[H_L^*]t_R)} = \frac{N E[H_L](3t_A + 2t_R)}{N\alpha(N)(3t_A + 2E[H_L^*]t_R)} > \frac{1}{\alpha(N)^{-1}E[H_L^*]}
\]  
(3.14)
where \(\alpha(N)\) is a weighted average of the quantities \(\alpha_V(N)\) and \(\alpha_Q(N)\). From Eq. (3.14) it becomes very clear that there is good potential for speedup as long as \(E[H_L]\) is reasonably close to \(E[H_L^*]\). Alternatively, speedup exceeds unity if the product of the block speedup coefficient and the expected \(N\)-maximum order-statistic for the random walk is less than the expected number of steps required by the random walk. For a large class of transition probability matrices it can be shown [30] that

14
which, when combined with Eq. (3.14) gives

\[ E[H^*_L] \propto L \log N \]  

(3.15)

for some constant \( C \). If \( C > 1 \), then the amount of speedup clearly depends on how fast \( \alpha(N) \) falls in relation to the growth of \( \log N \). Obtaining a value for \( C \) without an explicit consideration of the transition probability matrix \( \mathbf{P} \) is a difficult proposition. Indeed, this motivates the experiments conducted in Section 4.

3.3 Discrete event simulation: A Queueing System

Consider a single server queueing system with general independent arrivals and general independent service times (i.e., a \( GI/GI/1 \) queue). An event-scheduling based simulation of such a system, intended to yield some performance related characteristic of the queue such as mean queue length or mean queueing delay etc., is essentially made up of two kinds of events - arrival events (A-events) and end-of-service events (E-events). Instead of outlining the code for the queueing simulation, as was done in the previous examples, the flow of control for a non-unified simulation of a single server queue utilizing the event-scheduling paradigm is shown in Figure 7a. Program block A contains code that processes an arrival event, and program block E contains code that processes an end-of-service event.

In this queueing system, an event is represented by a two-tuple containing its type (i.e., A or E) and its time of occurrence. A data structure called a priority queue [2] is used to store a list of pending events while the simulation is in progress. The simulation is initiated in block I with an arrival event \((A, t_1)\) stored in the priority queue, and with a variable called \( \text{clock} \) initialized to zero. Control then passes to block G in which the priority queue
is accessed to obtain the event with highest priority. To maintain causality, an event \((.,t_i)\) is said to have a higher priority than an event \((.,t_j)\) if \(t_i > t_j\); we assume that ties are broken arbitrarily. If the tuple retrieved from the priority queue is \((C,t)\), then event \(C\) is said to occur at time \(t\). Hence the clock is updated by assigning it the value \(t\), and appropriate processing for the event \(C\) begins, where \(C\) is one of either \(A\) or \(E\). Assuming a queueing system operating at steady-state, let \(\beta\) denote the probability that program control branches off to block \(A\); correspondingly, \((1 - \beta)\) is the probability that program control branches off to block \(E\). That is, \(\beta\) is the steady-state probability that the highest priority event retrieved from the event queue is an arrival event.

It is known that for a system operating at steady state (assuming the queueing system is stable), the average number of arriving customers is equal to the average number of departing customers, so that \(\beta = 1/2\). Consider, for example, the situation when the customer arrival process is Poisson with rate \(\lambda > 0\) and the customer service times are independent exponentially distributed random variables with mean \(1/\mu\). Using \(\{A\}\) and \(\{E\}\) to denote the events corresponding to program control reaching blocks \(A\) and \(E\), respectively, the behavior of the simulation program over the space \(\{A, E\}\) can be described by a Markov chain \(\{X_n; n \geq 0\}\). Since the simulation must begin with a pre-scheduled arrival event (which occurs in initialization block \(I\)), the first branch from block \(G\) is made to block \(A\) so that \(X_0 = A\). The probability that the next branch from block \(G\) takes the model to block \(A\) (i.e., event \(\{X_1 = A\}\) occurs) is given by

\[
P(X_1 = A|X_0 = A) = P(I < S)
\]

where \(I\) and \(S\) are exponentially distributed random variables with mean \(1/\lambda\) and \(1/\mu\), respectively. It follows that

\[
P(X_1 = A|X_0 = A) = \int_0^\infty \mu e^{-\lambda y}dy - \int_0^\infty \mu e^{-(\lambda + \mu) y}dy = \frac{\lambda}{\lambda + \mu}
\]

and \(P(X_{n+1} = A|X_n = A) = \lambda/(\lambda + \mu)\) for all \(n \geq 0\), by using the fact that service times are memoryless. On the other hand, given that \(X_n = B\) for any \(n\), the event \(\{X_{n+1} = B\}\) will occur only if the queue is not empty and the service time of the first customer in the queue is less than the interarrival time of the next arriving customer. Once again, exploiting the memoryless property of customer interarrival times, and the fact that the event corresponding to a nonempty queue at time \(n\) is independent of event \(\{X_n = B\}\), we arrive at

\[
P(X_{n+1} = B|X_n = B) = \rho \cdot P(S < I)
\]

\[
= \rho [1 - P(I \leq S)]
\]

\[
= \frac{\lambda}{\mu} \cdot \left( \frac{\mu}{\lambda + \mu} \right) = \frac{\lambda}{\lambda + \mu}
\]

for all \(n\), so that the transition probability matrix for the chain \(\{X_n; n \geq 0\}\) is given by

\[
P = \begin{pmatrix}
A & E \\
\frac{\lambda}{\lambda + \mu} & \frac{\mu}{\lambda + \mu} \\
\frac{\mu}{\lambda + \mu} & \frac{\lambda}{\lambda + \mu}
\end{pmatrix}
\]

Observing that \(P\) is doubly stochastic, one readily concludes that
\[ \beta = \lim_{n \to \infty} P\{X_n = A\} = 1 - \lim_{n \to \infty} P\{X_n = E\} = \frac{1}{2} \]  

(3.21)

Using \( t(Z) \) to denote the average time required by the model to execute a given block \( Z \) each time it is entered, the average time \( T_S(N) \) required to sequentially execute the \( GI/GI/1 \) simulation model a total of \( N \) times, each time for a different value of \( \lambda \) and \( \mu \) is given by

\[ T_S(N) = N t(I) + \{ t(G) + t(S) + \frac{t(A) + t(E)}{2} \} \sum_{j=1}^{N} m_j \]

\[ = N t(I) + N t \bar{m} [t_G + t_S + \frac{t_A + t_E}{2}] \]  

(3.22)

where \( m_j \) is the number of times that the \( j \)th execution of the model moves from block \( G \) to block \( S \) prior to termination, and \( \bar{m} \) is the average of loop counts \( m_1 \) through \( m_N \). For a given block \( Z \), the quantity \( t_Z \) denotes the proportion of time required to execute block \( Z \) once, relative to the the time \( t \), where

\[ t = t(G) + t(A) + t(E) + t(S) \]  

(3.23)

is the average amount of time required to execute each block in the loop from \( G \) to \( S \) once.

In applying the unification algorithm [31], \( N \) instances of the program are made to execute in unison. In case it is required to execute the original model on parameters \((\lambda_j, \mu_j)\), for \( 1 \leq j \leq N \), it should be clear that a single unified program can serve to execute all \( N \) models in parallel. On the other hand, if it is required to execute the original model on a single parameter set \((\lambda, \mu)\), the unified version will simulate \( N \) sample paths of the same stochastic process in parallel and combine the results. It is only left to be seen if in both cases the unified program will yield enhanced performance in machine utilization and speedup.

In executing all \( N \) models together, the resulting code now vectorizes, as depicted in Figure 7b. Given that \( n \) models, \( 0 < n \leq N \), continue to execute within the unified program at a given time instant, with the rest already having terminated execution, the vector program must loop from block \( G \) to \( S \) repeatedly, until the model count \( n \) decreases to zero and the vector program terminates. Let \( m = \max\{m_1, m_2, \ldots, m_N\} \) denote the maximum number of times that any component of the unified program loops from block \( G \) to \( S \) prior to termination. Using \( T_U(N) \) to denote the time required by the unified program to complete execution,

\[ T_U(N) = N [\alpha_I(N)t(I) + m[\alpha_G(N)t(G) + \alpha_A(N)t(A) + \alpha_E(N)t(E) + \alpha_S(N)t(S)]] \]  

(3.24)

where \( \alpha_Z(N) \) is the inverse block speedup coefficient for a given block \( Z \). For \( m \) large, Eq. (3.24) can be written as

\[ T_U(N) \approx N m[\alpha_G(N)t(G) + \alpha_A(N)t(A) + \alpha_E(N)t(E) + \alpha_S(N)t(S)] \]

\[ = N m t \bar{m} \alpha(N) \]  

(3.25)
\[ S(N) = \frac{T_S(N)}{T_{ij}(N)} \]
\[ \simeq \frac{N \left[ t(I) + t_mC_1 \right]}{m \alpha(N)} \]  

where \( 0 < C_1 = t_G + t_S + \frac{1}{2}(t_A + t_E) < 1 \). Thus we obtain,

\[ S(N) \simeq C_1 \frac{\bar{m}}{m} \cdot \frac{1}{\alpha(N)} \]  

suggesting that speedup is largely determined by \( \bar{m}, m \) and \( \alpha(N) \). If \( m_1, m_2, \ldots, m_N \) are discrete random variables, it can be shown that under fairly general conditions [30]

\[ \frac{\sum_{j=1}^{N} E[m_j]}{N E[m]} = \frac{C_2}{\log N} \]  

for large \( N \), with \( E[m] = E[\max\{m_1, m_2, \ldots, m_N\}] \) and \( C_2 > 1 \). Hence, it follows that for some constant \( C > 0 \),

\[ S(N) \simeq \frac{C}{\alpha(N) \log N} \]  

implying that as long as \( \alpha(N) \) falls faster than \( \log N \) can grow, up to some threshold value of \( N \) for the vector multiprocessor, speedup will exceed unity. Once again, determining general conditions under which \( C > 1 \) is difficult without empirical work.
4 Empirical Results

In this section we present empirical results for four different applications of unified stochastic simulation. The experiments were performed on an eight-processor Alliant FX/8 at Argonne National Laboratory. We give a brief description of each simulation, and speedup results for the problem. Each experiment consists of a dozen runs of an $N$-component unified program, where $N = 1, 5, 10, 20, 30, 40, 50, 100, 200, 300, 400, 500$. Observe that even the case $N = 1$ represents a unified instance, since a unified program is designed to handle a variable number of unified instances, thus causing some execution overhead when $N = 1$. Also, experiments are repeated under different conditions, as will be made clear in the following discussion. In each case, the simulation terminates with a confidence interval for the estimate, based on a Student-t distribution, and a prescribed precision (i.e., ratio of the half-width of the confidence interval relative to the estimate) [26].

Let $T_U(N)$ denote the amount of time required to execute an $N$-component unified simulation which terminates only when some prescribed statistical accuracy is obtained for the simulation estimate. In each of the following simulations, we first compute $T_U(N)$ for the different values of $N$ mentioned above. For each value of $N$, we make careful use of random number seeds to execute the original (non-unified) simulation code in a consistent manner. That is, each of the $N$ sequential executions is made to take the same trajectory as its counterpart in the unified code. It is possible to ensure this consistency in most instances, though it is not strictly necessary. One could argue that the simulation results given by a unified program would be statistically correct even if this consistency were not preserved. Using $T_S(N)$ to denote the execution time of the sequential version, speedup is computed as the ratio $S(N) = T_U(N)/T_S(N)$, regardless of the number of vector processors involved in the computation. It is important to note that though the Alliant FX/8 provides eight processors, it is not possible to utilize these in MIMD mode. That is, the eight processors operate in coupled fashion, and it is not possible to program the processors to work on different pieces of code. Applications which do not vectorize perform poorly on such vector multiprocessors because they cannot make use of more than one processor. Indeed, poor vectorization makes for inefficient use of even this single processor.

Single-server queueing system

Using the unification scheme outlined in the previous section, we simulated the single-server queue both sequentially, and using unification. Note that in the absence of the unification idea, an analyst interested in computing some model dependent characteristic, such as mean queue-size versus traffic intensity $\rho (\rho = \lambda/\mu)$ will have to make repeated runs, for different values of the control variable $\rho$, where $0 < \rho < 1$. Through unification, the entire set of values used for $\rho$ can be input to the model simultaneously, and the result is typically available to the analyst sooner than through sequential execution.

In Figure 8 can be seen a set of four speedup curves, where the legends represent simulations done using the regenerative method [8] and the batch-means method [26]. In each case, the mean-queue length is estimated through a 99% confidence interval, with precision 0.15. The speedup results obtained in all four experiments are encouraging. While related experiments [32] suggest to us that the regenerative method will generally outperform the batch-means method in concurrent simulations where regenerative cycles have low or moderate variability, the relationship between the speedup curves obtained here for the two methods are unclear. In both cases, rotation sampling was used in the unified code.
to induce correlations and reduce variability. Under unification, both methods yield good speedups even when a single processor is used.

**Dijkstra's self-stabilization algorithm**

A good example of an algorithm in a distributed setting is the self-stabilization algorithm proposed by Dijkstra [9]. When loosely coupled processors cooperate by exchanging messages with one another, processors are capable of taking the system into an erroneous state. One of a number of algorithms proposed by Dijkstra is the \(-\)-state algorithm, which takes a system of processors from an erroneous state back into an error-free state.

Dijkstra's algorithm can very briefly be outlined as follows. Assume that \(M\) processors are arranged in a unidirectional ring, with processors capable of receiving from their clockwise neighbours. Initially, each processor possesses an arbitrary integer label from the set \(\{1, 2, \ldots, K\}\), where \(K > M\). Given a specific boolean function \(B(L(i), L(j))\) where \(j = i \mod M + 1\), and \(L(i)\) defines the label of processor \(i\), we say that processor \(i\) is in trouble if the function evaluates to true. The system is said to be in an erroneous state if one or more processors is in trouble. The \(-\)-state algorithm allows each troubled processor \(i\) to asynchronously obtain label information \(L(j)\) from its clockwise neighbour \(j\) with the intention of forcing \(B(L(i), L(j))\) to become false, thereby ridding itself of its trouble. In so doing, it may cause its downstream neighbour to acquire a troubled status. By progressing through a sequence of such label acquiring actions by processors, the \(-\)-state algorithm ultimately brings the system into an error-free state. What is of interest to the analyst is the average execution time of the algorithm, as a function of \(M\) and \(K\).

In this experiment, we execute Dijkstra's algorithm on a ring of \(M = 100\) processors, with \(K = 200\). Each run of the algorithm comprises two separate phases. A setup phase is initiated in which processors are assigned labels in such a way that initially 50 processors out of \(M\) are in a troubled state. Next, the execution phase that actually simulates the \(-\)-state algorithm is initiated. The entire run yields one sample path of the stochastic process that we are interested in studying. The runs are repeated so as to obtain a 99% confidence interval with a precision of 0.15. The speedup plot for this experiment can be seen in Figure 9.

Unlike the previous experiment, the components of the unified program in this experiment pool their results together during program execution in order to combine results across the components. The situation where each of the components in the unified program utilizes the same data (i.e., the same initial configuration of troubled processors) is indicated in Figure 9 by the legend 1, i.e., for identical initial configurations. The corresponding situation for different initial configurations on different components is indicated by D. The legend S indicates that simple sampling is performed; the legend R indicates that rotation sampling is used to induce correlations across components, and the legend C indicates that common random numbers are used on each program component. The latter technique is another device commonly used for variance reduction and perturbation analysis. Though speedup is positive, program path divergence is sufficiently large to keep speedup low.

**Belief Networks**

Belief networks are structures that are known to be useful for representing uncertain knowledge in a reasonably coherent, probabilistic form. A Bayesian belief network is defined [29] to be a directed acyclic graph in which nodes represent variables, links exhibit direct causal
Metalistic cancer

![Bayesian Belief Network Diagram]

**Increased total serum calcium**

**Brain tumor**

**Coma**

**Severe headaches**

\[
\begin{align*}
P(a) &= P(a=1) = 0.20 \\
P(b|a) &= P(b=1|a=1) = 0.80 \\
P(c|a) &= P(c=1|a=1) = 0.20 \\
P(d|b,c) &= P(d=1|b=1,c=1) = 0.80 \\
P(e|c) &= P(e=1|c=1) = 0.80 \\
\end{align*}
\]

Figure 10a: Bayesian belief network

influences between linked variables, and forward conditional probabilities contain information summarizing the strengths of these influences. Each node variable in a given network is initially either instantiated (i.e., fixed at some value), in which case it corresponds to some observed evidence concerning that variable, or it is left uninstantiated, in which case it is one of the variables responsible for information uncertainty. Thus under uncertainty, the variables represented by uninstantiated nodes are random variables. A primary goal of inference in Bayesian networks is obtaining posterior distributions of some or all random variables in the network, given certain priors and conditional probabilities.

Following Pearl [29], sequential stochastic simulation of a belief network can be described with the aid of an example borrowed from Spiegelhalter [35]. Consider the Bayes’ network shown in Figure 10a, along with its link matrix [29] of unconditional and conditional probabilities for each node variable. Given the information in this figure, it is of interest to determine the posterior probabilities of binary random variables from the set \( S = \{A, B, C\} \) of all uninstantiated variables. Observe that \( D \) and \( E \) are instantiated variables, with \( D = 0 \) and \( E = 1 \).

Pearl’s sequential algorithm, which is detailed in [29], is briefly outlined below. The algorithm proceeds through a sequence of simulation cycles, with \( X_n \) denoting the random variable representing node \( X \) in the \( n \)-th cycle. So in the case of Figure 10a, given some arbitrary initial state \( a_0, b_0, \) and \( c_0 \), where lower case letters are used to denote actual
values of the corresponding random variables, cycle \( n \) yields the set \( \{A_n, B_n, C_n\} \) of random variables, for \( n \geq 1 \).

Pearl's sequential algorithm is:

1. Select a variable \( Y \in S \) whose posterior distribution is to be estimated.

   Put each variable \( X \in S \) in an arbitrary initial state \( x_0 \).

   Set \( n \leftarrow 0 \).

2. **Repeat**

   \( n \leftarrow n + 1 \)

   For each variable \( X \in S \),

   2a. Compute the distribution of \( X_n \) conditioned on the current state of each of its neighbors.

   2b. Sample from this conditional distribution to obtain a realization \( x_n \) of \( X_n \), and put \( X_n \) in the state \( x_n \).

   2c. If \( X \) is the variable \( Y \) of interest, update the estimate \( \hat{P}(Y) \) for \( P(Y) \).

   until \( (\hat{P}(Y)) \) converges to a desired statistical accuracy.

Observe that starting from an arbitrary initial state for each of the random variables, the sequence of cycles represents a stochastic process which eventually reaches equilibrium. This means that the sequence \( \{(A_n, B_n, C_n); n \geq 0\} \) can be considered a strictly stationary stochastic process, for sufficiently large \( n \). This follows readily from the finiteness of the state space and the time independence of the transition probabilities in the link matrix of each random variable.

As in the previous experiments, this simulation was executed in unified mode. In Figure 10b is shown a set of four speedup results for six and eight processors. The required estimate was constructed by averaging passage times (i.e., times between returns to a given state) in the network. The simulation terminated with a 99% confidence interval with precision 0.1. Rotation sampling was used to induce correlation across the components of the unified program. Each component was made to work on different data by varying \( P(A) \) across components. The legend S (same) indicates a consistent experiment, i.e., the unified code was made to yield exactly the same results as the nonunified code. In contrast, the legend D (different) indicates an experiment where only statistical accuracy is required. That is, in the latter case the unified code terminates when a statistically correct result is obtained, though its precise value is different from that given by the nonunified code.

The results in Figure 10c show speedup obtained by varying the number of vector processors from one through eight. In this experiment, the required estimate was constructed by counting the number of times node A was found in state 1 relative to the total number of times it was visited, i.e., a frequency count. It is interesting to observe that speedups higher than five are obtained through program unification on a single processor. As in the previous case, the different components of the unified program were made to work on different data, but the experiment was consistent.

The results in Figure 10d show a less glamorous picture. In this case, the unified code is made to yield statistically correct results, though not identical to results of the nonunified code. Unlike the previous cases, all components of the unified program now work on the
same data, though initialized with different random number seeds to ensure that they take independent trajectories. However, the main reason that results are pessimistic is the use of a particular statistic combining strategy that we use to eliminate statistical bias. Heidelberger [20] shows that combining samples across samplers by taking an equal number of samples from each sampler is one way to eliminate order-statistic related bias. In this experiment, the combining routine introduces a relatively high overhead by making those components that have already computed a sample wait for others to complete. We expect that other statistic-combining methods will alleviate this problem to some extent. Observe that this is not a symptom of the unification scheme, but rather due to problems of bias elimination.

Multi-dimensional Integral Estimation

The computation of integrals in higher dimensions generally requires a tremendous amount of calculation. Monte Carlo methods are known [14] to be more efficient than analytical techniques when the number of dimensions is beyond seven.

A classical Monte Carlo technique for estimating multidimensional integrals is the sample-mean method. For ease of explanation, we assume the function \( h(x) \) to be integrated is bounded and non-negative over domain \( R_h \) of vector \( x \) in a \( (d - 1) \) dimensional space. In order to estimate

\[
I = \int_{R_h} h(x) \, dx
\]  

we begin by choosing a density function \( f(x) \) defined over \( R_h \). Then \( I \) can be expressed as

\[
I = \int_{R_h} \left[ \frac{h(x)}{f(x)} \right] \, f(x) \, dx = E \left[ \frac{h(X)}{f(X)} \right]
\]  

where \( X \) is a random vector whose density is \( f(\cdot) \), and \( E[\cdot] \) denotes expectation.

A sampling process selects a certain number \( n \) of points \( \{x^{(j)}; 1 \leq j \leq n\} \) randomly from \( R_h \), according to the density \( f(\cdot) \). An estimate \( \hat{I} \) of \( I \) is thus obtained as the sample mean of \( n \) observations of \( h(\cdot) \), where

\[
\hat{I} = \frac{1}{n} \sum_{j=1}^{n} \frac{h(x^{(j)})}{f(x^{(j)})}
\]  

In this experiment we constructed an estimate of a multi-dimensional integral of an exponential in the positive quadrant. The performance results for this experiment are shown in Figure 11 for one, four, and eight vector processors. The estimate of the integral is computed using a 99% confidence interval with precision of 0.05.

The legend “Indep” indicates that components of the unified program sample independently of one another, though rotation sampling is used across components. Each component of the unified program is made to work on different data, and hence the multidimensional integral is computed for several parameter sets (i.e., each component works on one parameter set) simultaneously. The unified code returns statistically correct results, though not identical to the results given by the nonunified code.

The legend “Comb” indicates that a statistic-combining strategy (i.e., same as that used in the belief network experiment) is used; here, each component of the unified program works on the same parameter set, and the combining strategy pools the results together. Samples returned by the components are guaranteed to be independent across components through
proper use of random number seeds. As in the belief network experiment, the combining strategy increases the overheads incurred by unified program execution, thereby decreasing speedups. However, in both cases, speedups for this experiment are positive.

5 Conclusion

From our experiences with unification on the Alliant FX/8 and the CRAY Y-MP, we are certain that program unification exhibits significant potential for obtaining speedup, and increased MFLOP and utilization ratings on vector multiprocessors. However, our experiments have been limited to applications in the simulation domain, and to code that is a few hundred lines long. In addition, unified code is generated manually, with painstaking attention to how loops are combined and how random numbers are used across components.

Our current work involves experiments with larger applications on the CRAY Y-MP vector multiprocessor, and the MasPar and CM SIMD multiprocessors. In the course of this work, we plan to automate (at least partly) the unification process, so that we may experiment with larger pieces of simulation code. It is our belief that the unification idea is still in its infancy as far as our understanding of its performance goes; there is much scope for experimentation with various levels of program unification.

Acknowledgement

The experiments reported here were conducted on an Alliant FX/8 at Argonne National Laboratory. We gratefully acknowledge this support from ANL.

References


Figure 3c
No partition vector

\[ \gamma(N) \]

\( N \) components active

\( \frac{N}{2} \) components active

Figure 4c
Figure 5c
Figure 8. Speedup for Single Server Queue
Figure 9. Speedup for Dijkstra's Self-Stabilization Algorithm
Figure 10b. Belief Network with Rotation Sampling
Figure 10c. Belief Network with Rotation Sampling and Frequency-based Estimation
Figure 10d. Belief Network with Combining Strategy
Figure 11. Speedup for Integral Estimation