Non-Strict Cache Coherence: Exploiting Data-Race Tolerance in Emerging Applications

Siddhartha V Tambat   Sriram Vajapeyam

IISc-CSA-TR-2001-3
http://archive.csa.iisc.ernet.in/TR/2001/3/

Computer Science and Automation
Indian Institute of Science, India

July 2001

A shorter version of this document was published in the 29th International Conference on Parallel Processing (ICPP-2000) held in Toronto, Canada, August 21–24, 2000.

Non-Strict Cache Coherence: Exploiting Data-Race Tolerance in Emerging Applications

Siddhartha V. Tambat
Dept. of Computer Science & Automation

Sriram Vajapeyam
Supercomputer Education & Research Centre and Dept. of Computer Science & Automation

Indian Institute of Science
Bangalore 560012 INDIA

{svtambat,sriram}@csa.iisc.ernet.in

July 31, 2001

Abstract

Software distributed shared memory (DSM) platforms on networks of workstations tolerate large network latencies by employing one of several weak memory consistency models. Data-race tolerant applications, such as Genetic Algorithms (GAs), Probabilistic Inference, etc., offer an additional degree of freedom to tolerate network latency: they do not synchronize shared memory references, and behave correctly when supplied outdated shared data. However, these algorithms often have a high communication-to-computation ratio and can flood the network with messages in the presence of large message delays. We explore the benefits of designing a DSM with non-strict cache coherence for such applications. We study the performance of controlled asynchronous implementations of these algorithms via the use of a previously proposed blocking Global_Read memory access primitive. Global_Read implements non-strict cache coherence by guaranteeing to return to the reader a shared datum value from within a specified staleness range; synchronization primitives are thereby avoided. As compared to fully asynchronous implementations, controlled (i.e. partial) asynchrony, implemented using Global_Read, reduces the overall amount of computation done with stale data by a process, thus controlling the amount of shared updates (and thereby the network traffic) generated. Experiments on an IBM SP2 multicomputer with an Ethernet interconnect show significant performance improvements for controlled asynchronous implementations. On a lightly loaded network, most of the GA benchmarks see 30% to 40% improvement over the best competitor across configurations ranging from 2 to 16 processors, while two of the Probabilistic Inference benchmarks see more than 80% improvement on a 2-node configuration. As the network load increases, the benefits of non-strict coherence and partial asynchrony increase significantly. Overall, non-strict cache coherence is indicated to be significantly beneficial over both the data-race-free based weak consistency memory models and fully asynchronous models that have no guarantees regarding coherence.

{A shorter version of this document was published in
the 29th International Conference on Parallel Processing (ICPP-2000)
held in Toronto, Canada, August 21-24, 2000.}

1. Introduction

High performance networks of workstations are becoming increasingly popular as a parallel computing platform. Both message-passing [4, 10] and software distributed shared memory [8, 6, 24] paradigms (DSMs) have been developed on such distributed platforms. An important performance bottleneck in these systems is the effective message transmission latency, which is poorer than in high-speed parallel computer interconnection networks. Software DSMs have attempted to reduce both the quantity of data transferred and the number of messages by supporting weaker [2, 11, 12] forms of the shared memory paradigm such as weak consistency, release consistency, lazy release consistency [22], and entry consistency. The propagation to other nodes of a shared location update can be delayed in such memory models until the point of actual use of the data [22], which is indicated by the explicit synchronization present in the application program. Thus, the performance optimizations are aimed only at programs which have a synchronous model of computation, i.e. *data-race* free programs.

A significant number of application classes function correctly in the presence of data races, for example, iterative equation solvers, genetic algorithms, probabilistic inference in Bayesian belief networks, neural networks etc. These applications offer an additional degree of freedom to address large data transmission latencies. These algorithms do not synchronize accesses to shared memory locations, and behave correctly in the presence of losses and delays in the propagation of shared memory updates. This can give them a performance advantage over their synchronous counterparts [5] since synchronization costs are avoided and further computation can be overlapped with the communication of shared writes. The asynchronous nature of the communication allows the underlying DSM to tradeoff latency to (a) dynamically adapt better to network load, via techniques such as buffering [31], and (b) amortize message transmission overheads, by coalescing several updates of a single shared memory location when outgoing messages are blocked. The tradeoff of latency is done at a potential cost: typical asynchronous algorithms are iterative in nature and converge to a solution of acceptable quality, and the rate of convergence is critically dependent on the propagation delay of shared variables. Each additional iteration needed to reach convergence results in additional shared memory updates and thus additional network traffic. When running on an already heavily loaded network, uncontrolled asynchronous algorithms can potentially flood the network with messages, moving the network to unstable conditions and thus unboundedly increasing the communication delay.

We have previously proposed [20] a method of controlling asynchronous algorithms wherein the message generation rate is controlled by the receiver of shared updates rather than by the sender. The main idea is to enforce an upper bound on the *age* of shared values read by a node. This is done by the use of a system supported blocking read primitive, termed Global_Read, that is guaranteed to return a value of acceptable age of the specified shared location. The receiver process is throttled until its Global_Read is satisfied, thus implementing program-level flow control. The underlying DSM could use the invocation of Global_Read as a hint to temporarily give higher network or processor priority (as appropriate) to the sender process when necessary. Alternately, Global_Read could be implemented as a user-level library. In realistic scenarios, the use of Global_Read also prevents individual processes from straying too far ahead or behind the other processes of the program; if a process strays too far ahead, it blocks on executing a Global_Read and waits for the others to catch up; if a process strays too far behind, all other processes will block at Global_Reads and wait for the stray process to catch up. In essence, the use of Global_Read converts a fully asynchronous algorithm into a *partially asynchronous* algorithm [5].

A preliminary performance evaluation of Global_Read using a simple linear equation solver kernel as benchmark and the first order metric of the number of iterations needed for convergence was reported in [20]. The results showed that Global_Read enables us to prevent unstable network conditions, which is achieved by specifying an appropriate amount of asynchrony via the Global_Read parameters. In this paper, we present a detailed performance evaluation of non-strict cache coherence implemented via Global_Read for two important, emerging application classes: (i) Genetic Algorithms (GAs) [13], and (ii) Probabilistic Inference in Bayesian belief networks using an approximate search algorithm [29, 27]. We use program completion time as our primary metric, while also reporting the number of iterations needed for convergence and various other metrics.

Experiments on an IBM SP2 multicomputer with an ethernet interconnect show significant performance improvements for controlled asynchronous implementations. On a lightly loaded network, most of the GA benchmarks see 30% to 40% improvement over the best competitor across configurations ranging from 2 to 16 processors, while two of the Probabilistic Inference benchmarks see more than 80% improvement on a 2-node configuration. As the network load increases, the benefits of non-strict coherence and partial asynchrony increase significantly.

The rest of the paper is organized as follows. In the next section, we present the Global_Read primitive, and discuss related work. In section 3, we briefly describe Genetic Algorithms and Probabilistic Inference in Bayesian Belief Networks, which are used as the driver applications in this work. Section 4 describes our experimental setup. In section 5, we present performance results showing the effectiveness of non-strict coherence implemented via the Global_Read primitive. Finally, section 6 summarizes the paper.

2. The Global_Read Primitive

The Global_Read primitive [20] was proposed as a shared-memory access mechanism that enables the programmer to control/ limit the amount of runtime asynchrony in asynchronous applications. The Global_Read primitive is visible to the programmer and takes three arguments: the shared location to be read, the current iteration number of the reading process, and the maximum acceptable age of the shared datum to be read. The maximum age is specified as the maximum number of iterations prior to the current iteration number that the shared datum could have been generated. Thus, *Global_Read(locn, curriter, age)* returns a value of *locn* generated no earlier than in the $\mid curriter - age \mid$'th iteration of the process that is generating successive values of *locn*. This implies that if the local copy of *locn* is older than acceptable, the reading process is blocked until an acceptable newer value of *locn* becomes available. Alternately, when the local copy is within the age limit specified, the Global_Read degenerates to an ordinary read. (There is still some amount of overhead in detecting whether the local copy is of acceptable age, but this is often negligible considering the high communication-to-computation ratio of asynchronous applications.) The *age* is chosen by the programmer based on the desired convergence rate for the algorithm and the convergence rate features of the algorithm.

The implementation of the Global_Read primitive in a DSM involves the maintenance of age information with each local copy of a shared location. The age of each shared variable has to be updated locally at every write, and propagated along with every propagation of the variable. The increase in DSM message sizes due to the extra age information is not likely to have a detrimental effect on performance because message latency is determined more by message overheads than by message size. On a Global_Read, the DSM checks whether the age of its local copy meets the specified requirements. If not, it blocks the reading process until a value of suitable age is locally available, and either proactively broadcasts a request for a copy of suitable age, or just waits until the required update arrives. In the former method, the receipt of such a request for a new copy of a variable can be interpreted as a hint that the process is running slower than the process issuing the Global_Read. The hint can be used to increase the process's immediate network priority or local processor-scheduling priority, as appropriate. The simple blocking implementation will generate fewer messages, and is more efficiently implemented as a user-level library routine. We consider only the latter implementation since dynamic load-balancing, which is a key additional benefit of the former implementation, is beyond the scope of this paper.

2.1. Related Work

We proposed Global_Read as a mechanism for implementing non-strict cache coherence in early 1996 [20, 21]. Independently and around the same time, the *delta consistency model* was proposed [30] for applications that tolerate loose notions of synchronization and consistency. Interestingly, the use of Global_Read results in a memory consistency model that is very similar to delta consistency.

Delta consistency [30] is motivated by the assumption that a temporal programming model is more intuitive for the development of applications that need temporal correctness guarantees. This model embodies two mechanisms: *delta consistency* - a time based correctness criterion to govern the shared memory access guarantees that is very similar to Global_Read, and *temporal synchronization* - a mechanism for thread synchronization along the time axis. These mechanisms have been implemented on a cluster of workstations in a software distributed shared memory architecture called *Beehive*, and the efficacy of these mechanisms in meeting real-time requirements of applications has been shown using simulation of interactive virtual environments as the driver application.

The performance potential of asynchronous algorithms on workstation clusters had earlier motivated the development of the software DSM system Mermera [31], which supports non-coherent memory operations for usage by asynchronous algorithms. Mermera supports slow memory [19], a very weak consistency model which does not require much ordering constraints on the updates sent by a processor, and on which totally asynchronous algorithms are guaranteed to converge. The writes to slow memory return as soon as the writes have been submitted to the DSM system for propagation, and are hence asynchronous in nature. The asynchronous nature of the communication allows the underlying DSM system to reduce network load and amortize message overheads by buffering updates,and transmitting the multiple updates together in a single message. The asynchronous nature of the shared memory updates, supported in the system Mermera, allows uncontrolled asynchronous algorithms to issue updates as fast as the processing power of the CPU allows. This results in a high traffic generation rate. The performance problems related to the high message arrival rate on to the network is usually more acute in a DSM system, since potentially every shared update can result in a message. The performance of uncontrolled asynchrony deteriorates further when the network is already loaded.

A previously proposed method [17] for controlling asynchronous algorithms uses the Warp [28] flow control protocol to adaptively throttle message generation rate of the asynchronous algorithm as a function of the current estimate of network load. Throttling is implemented by either not submitting a shared location update for transmission until congestion is alleviated [17] or by entirely discarding the update and proceeding with further computation [16]. However, asynchronous algorithms also have the problem that a few lightly loaded nodes in the computation may run ahead and generate unnecessary message traffic due to non-receipt of updates from heavily loaded nodes which are slow in finishing their iterations. Adaptive throttling will kick in once the network gets heavily loaded in this scenario but cannot prevent the initial flooding.

3. Data-Race Tolerant Applications

3.1. Genetic Algorithms

A genetic algorithm (GA) [18, 13, 25] is a search procedure inspired by the "survival of the fittest" principle of natural evolution. In a GA, a constant-sized *population* of *individuals*, representing a possible solution to the given problem, is *iteratively* evolved from *generation* to *generation* until the solution is reached. Each individual is assigned a *fitness* score according to how good a solution to the problem it is. The genetic search proceeds across generations by replicating or weeding out individuals in a generation based on some evaluation of their fitness, and thus transforming to a new generation. Usually, these transformations involve two operators: *crossover* and *mutation*. The termination criterion of the GA can be based either on a prespecified number of generations, on the amount of variation of individuals between different generations, or on a prespecified level of fitness achieved by some individual. GAs are used in a variety of application domains including numerical function optimization, combinatorial optimization, image processing, design tasks, and machine learning.

Parallel GAs are increasingly being used since GAs may require a huge amount of time in order to find good solutions to various hard problems in state space search and optimization. In addition to reduction in the execution time, parallel GAs can also explore different regions of the search space simultaneously thus leading to a better quality solution. In this work, we consider a particular class of parallel GAs known as *coarse-grained parallel* or *"island"* GAs. In this case the population of the GA is divided into multiple sub-populations or *demes* [7] that evolve isolated from each other most of the time, but exchange individuals occasionally. This exchange of individuals is called *migration*, and it is controlled by several parameters: interval, rate, and topology. Two of the important issues that arise in the design of island GAs are the sub-population size and the migration of individuals between sub-populations. Empirical studies [7] have shown that a parallel GA with small to medium-sized demes, and one in which each processor broadcasts the best individual found in every generation to all other processors, found the global solution in fewer number of generations compared to other parallel GAs.

The communication penalty and the overall execution time of coarse-grained parallel GAs can be often substantially reduced by means of an *asynchronous* implementation. An asynchronous implementation of the island GA, in a message passing system, will not wait for migrants from the previous generation to arrive. Each processor will continue computing with the available individuals and incorporate migrants into its population as and when they arrive. However, our experiments indicate that since the message generation rate of asynchronous GAs is greater, they can create large network loads, resulting in large queuing delays, in turn increasing the message generation of the asynchronous GAs, i.e. a positive feedback loop leading to network overload. Moreover, the increase in the number of generations supersedes the gains from the removal of synchronization, especially when the communication penalties are high as in a low bandwidth Ethernet. In addition, the quality of the genetic search is also adversely affected by large message delays. A deme, on account of the small size of its population, may converge quickly to its local optimum in the absence of any migrants from other demes for a fairly long time. The migrants will not have a significant effect on the receiving deme if the demes remain isolated for a long time, and so the overall quality of the solution will be poorer.

We use Global_Read to implement partially asynchronous parallel GAs and thus alleviate all the above-mentioned problems with synchronous and fully asynchronous parallel GAs.

3.2. Probabilistic Inference in Bayesian Belief Networks

Probabilistic inference [23] is an important technique for reasoning under uncertainty in such areas as medicine, software fault diagnosis, speech recognition, and automated vision. It answers a question about event probabilities given information about other events. The events and the dependencies among them are usually represented by a directed acyclic graph, referred to as a *Bayesian belief network* [29]. Nodes in the belief network denote events. Edges denote the dependencies between events. Each node has a finite number of values, which corresponds to the possible number of outcomes of that event. The information about events and the dependencies among their outcomes is expressed as *conditional probabilities*. Real life Bayesian Networks tend to be large and complex, and their use is often restricted by the time required to draw inferences from such large networks [23], thus necessitating parallel implementations.

Figure 1 An example Bayesian network.

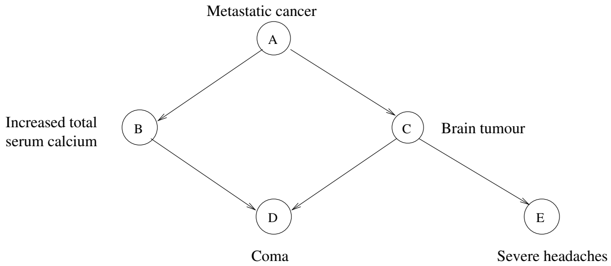


Figure 1 shows an illustrative belief network [29] which represents the following relationships:

Metastatic cancer is a possible cause of a brain tumour and is also an explanation for increased total serum calcium. In turn, either of these could explain a patient falling into a coma. Severe headache is also possibly associated with a brain tumour.

Each node takes on a set of values (here *false* or true) with probabilities that depend on the values taken by its parents. Associated with each node is a conditional probability, which describes the probability that the node takes each of its values given a combination of values for its parents. For example,

$$p(D = true \mid B = true, C = true) \quad = \quad 0.80$$

Probabilistic inference determines the conditional probability of a node or a set of nodes, called *query* nodes (e.g. the probability of brain tumour being absent or present in the patient), given the instantiation of evidence of other nodes (e.g. that the person suffers from a severe headache but has not fallen into a coma). Two types of approaches [29, 27] are taken to find the desired conditional probabilities: exact algorithms such as the Lauritzen-Spiegelhalter (LS) algorithm and Symbolic Probabilistic Inference (SPI), and approximate search algorithms or Monte-Carlo simulations. One of the approximate search algorithms that does probabilistic inference in a multiply connected network is the *logic sampling* algorithm [29], which we consider in this paper.

Logic sampling [29] is a method of computing probabilities by counting how frequently events occur in a series of simulation runs. For example, in the belief network of Figure 1, we can generate hypothetical samples of patients by the following procedure: We draw a random value a_1 for A, using the probability $P(a)$. Given a_1, we draw random values b_1 and c_1 for the variables B and C, using the probabilities $P(b \mid a_1)$ and $P(c \mid a_1)$, respectively. And so on, for the entire network. The combination of values for each of the nodes represents one sample of a patient scenario. The process is now repeated to obtain multiple patient scenarios. The probability of any event or combination of events is then computed by averaging the frequency of events over those cases in which the evidence variables agree with the data observed.

Though logic sampling takes much less time than exact methods, it becomes relatively more expensive with increasing demand for precision. Thus, there is a need to explore parallel implementations of logic sampling. An advantage offered by logic sampling is its inherent parallelism. If we associate a processor with each of the variables in the belief network, then the simultaneous occurrence of events within each run can be produced by concurrently activating the processors responsible for these events. (In practise, a subset of the network is assigned to each processor.) Each processor receives the values assigned to the parent nodes of the nodes in its own partition, and sends the values assigned to the nodes in its own partition to the child nodes that belong to different partitions. This procedurs *iteratively* converges to the posterior conditional probability distribution of the query nodes in the belief network. To alleviate the drawbacks of the synchronous implementation, we implement an *asynchronous* version of the algorithm using a modification of the *synchronization via rollback* [5] technique. To the best of our knowledge, there are no prior asynchronous implementations of logic sampling.

In the asynchronous implementation, each processor continuously updates the states of its nodes, and sends messages containing the values of the *interface nodes* [1] to other processors, without blocking for messages from other processors.

[1] A node whose adjacent nodes belong to different partition(s) is called an interface node

Figure 2 The C-style pseudo-code for the partially asynchronous program using Global_Read.

```
#define age DESIRED_UPPER_BOUND
t = 0;
do { /* Global Termination Loop */
    t++; /* Iteration Count */
    /* Process antimessages, if any */
    for (each node in partition) {
        for (each parent of node)
            Global_Read(&state, t, age);
        /* Compute new state for the node */
        for (each child of node)
            NonBlocking_Write(&state_of_node, t);
    }
    /* Check validity of current simulation run */
} until (global_termination()); /* iterate until
                    Global Termination */
```

Whenever a processor needs the value of some interface node and the corresponding message has not been received, the processor uses the default value π for that node. The default values for the interface nodes are determined on the basis of the conditional probability distribution of the nodes. For example, in the belief network of Figure 1, since $p(A = true) = 0.20$ and $p(A = false) = 0.80$, A will sample the value $false$ in four-fifths of a large number of samples, which is therefore used as the default value for A. The default values for the remaining nodes are determined in a similar fashion taking the dependencies in the network properly into account. In effect, each processor takes a gamble that the sampled value will be equal to the default value. If this is true for all the simulation runs and for all the interface nodes in the network, then it is evident that the simulation is completed in the least possible time, since each processor continues computing at full speed. When a processor receives a value from a node that differs from the default value for that node, the value of the child node and the values of all the nodes in the network that are dependent on this node and that have already been computed must be invalidated and recomputed. The processor then has to *roll back*. We use standard rollback techniques [], such as sending antimessages, to implement the rollback.

Figure 2 shows the fully asynchronous logic sampling algorithm modified to use Global_Read. The potential benefit of Global_Read in this scenario is to restrict the number of costly rollbacks by not allowing any processor to stray far ahead (or to lag far behind) of other processors. This results in a reduction in the number of iterations required for convergence, leading to an improvement in overall program completion time. Several optimizations are possible in this implementation. For example, Global_Reads are only required for the interface nodes in the belief network. This optimization can be especially useful in cases where the number of interface nodes (and hence the number of shared locations) is relatively less in comparison to the total number of nodes in the belief network.

4. Experimental Setup

4.1. Parallel Computing Platform

All our experiments were done on a multicomputer orchestrated by the PVM message passing library [], rather than on a real DSM implementation. A simple layer of software on top of PVM provided a shared-memory abstraction for use with Global_Read, without the optimizations inherent in a real DSM implementation. The applications considered in our studies allow for simple message-passing implementations of the shared-memory abstraction: since the readers of each value are known at compile time, direct sends and receives between processes suffice to implement shared location writes and reads. For locations accessed via Global_Read, a local user-level buffer at each node maintains the latest copies of the locations received from corresponding writers. Global_Read first checks this buffer before initiating a traditional receive(read) of the corresponding send (write) when necessary. Global_Read is implemented by simply waiting for the required update to arrive, since this method has lower message overhead than actively requesting a value of suitable age. In the experiments reported in this work, all the primitives that provide the Global_Read abstraction are implemented as user-level macros. (Macros are used instead of functions in order to avoid function call overhead.)

Table 1 Eight function test bed.

Fn	Function	Limits	$\min f(x)$
1	$f_1(x) = \sum_{i=1}^{3} x_i^2$	$-5.12 \le x_i \le 5.12$	0
2	$f_2(x) = 100(x_1^2 - x_2^2)^2 + (1 - x_1)^2$	$-2.048 \le x_i \le 2.048$	0
3	$f_3(x) = \sum_{i=1}^{5} integer(x_i)$	$-5.12 \le x_i \le 5.12$	0
4	$f_4(x) = \sum_{i=1}^{30} ix_i^4 + Gauss(0,1)$	$-1.28 \le x_i \le 1.28$	≤ -2.5
5	$f_5(x) = 0.002 + \sum_{j=1}^{25} \frac{1}{j+\sum_{i=1}^{2}(x_i - a_{ij})^6}$	$-65.536 \le x_i \le 65.536$	0.99804
6	$f_6(x) = nA + \sum_{i=1}^{20} x_i^2 - A\cos(2\pi x_i)$	$-5.12 \le x_i \le 5.12$	0
7	$f_7(x) = \sum_{i=1}^{10} -x_i \sin(\sqrt{\lvert x_i \rvert})$	$-500 \le x_i \le 500$	-4189.83
8	$f_8(x) = \sum_{i=1}^{10} x_i^2/4000 - \prod_{i=1}^{10} \cos(x_i/\sqrt{i}) + 1$	$-600 \le x_i \le 600$	0

The benchmark parallel programs used in our studies are written in in C and C++. The platform consisted of an IBM SP2 multicomputer having both a high-speed interconnect and a 10Mbps Ethernet. Each IBM SP2 node had an RS/6000 591 processor operating at 77 MHz. The average round-trip null message latency on the ethernet was found to be 2 ms. A large portion of this message-passing latency is due to system-software overhead. The latency measured is the sum of the times required for packing, sending, transmitting, receiving, and unpacking the message. It was obtained using a ping-pong program which sent packets back and forth between two nodes a large number of times. The wall clock time was measured at the beginning and the end of the ping-pong and the difference was used to compute the average message transmission delay. AIX 4.3, an IBM version of Unix, ran on each SP2 node. The C and C++ compilers, xlc and xlC respectively, available on AIX 4.3 were used for compiling the programs. All the programs used the -O2 optimization of these compilers.

We used the IBM SP2's LoadLeveler, which schedules user jobs in batch mode, to run our programs so that the nodes were ensured to be relatively free from background load during the experiments. For the applications available to us, the IBM SP2 connected by the ethernet provided the most suitable platform for illustrating the need for non-strict cache coherence when the network latencies are relatively high compared to the applications' communication demands. Hence we report most of our results for this platform. We expect that larger applications with higher communication requirements will see similar benefits from non-strict coherence even on faster interconnects such as the IBM SP2's high-speed switch.

4.2. Benchmarks

4.2.1. Genetic Algorithms. We use the function minimization problem to evaluate the performance of our GA benchmarks. Table 1 shows the test functions that we used in our experiments. Functions $F1$–$F5$ have been proposed by DeJong [9]. They are extensively used in the GA community. This suite includes functions which are non-convex($F2$), discontinuous($F3$), stochastic($F4$) and multi-modal($F5$). $F1$ is a unimodal function known to be easy for GAs. $F2$ is a harder multi-modal function. $F3$ is a discontinuous step ladder. $F4$ involves a large solution space (2^{240}) plus Gaussian noise. Because of the presence of noise, we consider $F4$ solved when the best string in the population reaches -2.5 [14]. $F5$ is characterized by the presence of several local minima.

Functions $F6$ through $F8$ are used in "mainstream" optimization [26]. $F6$ was proposed by Rastringin. It is highly multi-modal. The local minima are located at a rectangular grid with size 1. The global minimum is at $x_i = 0, i = 1, \ldots, n$, giving $f(x) = 0$. Grid points with $x_i = 0$ except one coordinate, where $x_j = 1$, give $f = 1$, the second best minimum. With increasing distance to the global minimum the fitness values of local minima become larger.

Function $F7$ was proposed by Schwefel. Here we use the function in arbitrary dimensions. The global minimum is at $x_i = 420.9687, i = 1, \ldots, n$. The local minima are located at the points $x_k \approx (\pi(0.5 + k))^2, k = 0, 2, 4, 6$ and $x_k \approx (\pi(0.5 + k))^2, k = 1, 3, 5$. The second best minimum is at $x_i = 420.9687, i = 1, \ldots, n, i \neq j, x_j = -302.5232$, far away from the global minimum. Therefore, the search algorithm may be trapped in the wrong region.

Function $F8$ has been suggested by Griewangk. The function has its global $f = 0$ at $x_i = 0.i = 1, \ldots, n$. The local minima are located approximately at $x_i(k) \approx \pm k * \pi\sqrt{i}, i = 1, \ldots, n, k = 0, 1, 2, 3, \ldots$. In ten dimensions, the function has four suboptimal minima $f(x) \approx 0.0074$ at $x \approx (\pm\pi, \pm\pi\sqrt{2}, 0, \ldots, 0)$. This function is the most difficult to optimize because it is non-separable.

In this work, our experiments are limited to a particular class of GAs characterized by the following six parameters [15]:

Table 2 Four Bayesian belief networks used to evaluate the parallel programs.

	A	AA	C	Hailfinder2.5
Nodes	54	54	54	56
Edges per node	2.2	2.4	2.0	1.2
Values per node	2	2	2	4
Edge-cut for 2 partitions	24	30	24	4
IBM SP2 uniprocessor inference time (seconds)	11.12	11.19	11.81	3.15

Population Size (*N*): The population size affects both the ultimate performance and the efficiency of the GAs. GAs generally do poorly with very small populations, because the population provides an insufficient sample size. A large population is more likely to contain representatives from all regions of the search space. Hence, the GAs can perform a more informed search. As a result, a large population discourages premature convergence to suboptimal solutions. On the other hand, a large population requires more evaluations per generation, possibly resulting in an unacceptable slow rate of convergence.
Crossover Rate (*C*): The crossover rate controls the frequency with which the crossover operator is applied. In each new population, $C * N$ individuals undergo crossover. The higher the crossover rate, the more quickly new individuals are introduced into the population. If the crossover rate is too high, high-performance individuals are discarded faster than selection can produce improvements. If the crossover rate is too low, the search may stagnate due to lower exploration rate.
Mutation Rate (*M*): Mutation is a secondary search operator which increases the variability of the population. After selection, each bit position of each individual in the new population undergoes a random change with a probability equal to the mutation rate *M*. Consequently, approximately $M * N * L$ mutations occur per generation, where *L* is the length of the binary string representation of the individual. A low level of mutation serves to prevent any given bit position from remaining forever converged to a single value in the entire population. A high level of mutation yields an essentially random search.
Generation Gap (*G*): The generation gap controls the percentage of the population to be replaced during each generation. That is $N * (1 - G)$ individuals of $P(t)$ are chosen (at random) to survive intact in $P(t + 1)$. A value of $G = 1.0$ means that the entire population is replaced during each generation. A value of $G = 0.5$ means that half of the individuals in each population survive into the next generation.
Scaling Window (*W*): When the objective is the minimization of some cost function $g(x)$, the cost-to-fitness transformation is commonly used. This transformation guarantees that the fitness $f(x)$ is positive regardless of the characteristics of $g(x)$. There are different methods of choosing the coefficient C_{max} based on a parameter called the scaling window *W*. If $W = 0$, then scaling is performed as follows: C_{max} is set to the maximum $g(x)$ value in the first generation. For each succeeding generation, those individuals whose objective function values are greater than C_{max} are ignored in the selection procedure. If $0 < W < 7$, then we set C_{max} to the greatest value of $g(x)$ which occurred in the last *W* generations. A value of $W = 7$ indicates an infinite window (i.e., no scaling is performed).
Selection Strategy (*S*): If $S = P$, a *pure selection* procedure is used, in which each individual in the current population is reproduced a number of times proportional to the structure's performance. If $S = E$, an *elitist strategy* is employed. First, pure selection is performed. In addition, the elitist strategy stipulates that the individual with the best performance always survives intact into the next generation. In the absence of such a strategy, it is possible for the best structure to disappear, due to sampling error, crossover, or mutation.

The work by DeJong [9] suggests parameter settings which have been used in a number of implementations of GAs. Based on DeJong's work, the parameter settings which we use in our experiments are: $N = 50$, $C = 0.6$, $M = 0.001$, $G = 1$, $W = 1$, and $S = E$.

The standard notion of speedup with increasing number of processors for a constant problem size is not a meaningful metric for parallel GAs. Since different processors in the parallel GA can be used to explore different regions of the search space simultaneously, it is interesting to determine any improvement in the quality of the genetic search as more processors become available. Therefore, in all the experiments with parallel GAs, we linearly scale the total population size with increasing number of processors. Each subpopulation in the parallel GA is initialized with individuals that are different from those in other subpopulations. Each processor broadcasts the best fit N/2 individuals found in each generation to all other processors. Each processor then replaces the worst individuals in its subpopulation with these migrants.

4.2.2. Probabilistic Inference in Bayesian Belief Networks. Table 2 lists the parameters of the four belief networks we used in our performance evaluations. The first three networks—A, AA, and C—are randomly generated [23]. That is, a completely interconnected graph of a given number of nodes was first built and then edges were removed randomly until it had a required number of edges. The inference time for these randomly generated networks was from 11.12 to 11.81 seconds with the sequential program. The remaining network is a belief network model of the diagnostic system Hailfinder developed at the Decision Systems Laboratory of the University of Pittsburgh [1] The uniprocessor inference time for this network is 3.15 seconds. Though this network is small like the randomly generated networks, we wanted to include this network because it was the only real network accessible to us. As mentioned in [23], most real, large Bayesian networks are proprietary and thus we have to make do with small, synthetic networks.

The table also lists the edge-cut of the partitions obtained by the graph partitioning program that we use. Since the edges of the belief networks do not reflect the true information exchange requirements of the underlying computations, this edge-cut is only an approximation of the true communication cost resulting from the partitioning.

4.3. Evaluation Metrics

The chief metrics of interest in all our experiments are program completion time and the number of iterations required for convergence. For all the programs, we use the total number of iterations across all the nodes as the performance measure since the number of iterations executed by the different processes of an asynchronous algorithm can be quite different.

An important metric for evaluating the performance of the parallel GA programs is the quality of the genetic search. Since the solution found by a GA is highly dependent on the choice of the initial population, we run all the programs for 25 runs, wherein the initial population in each run is initialized with a different set of individuals. The numbers reported are the average of the results of these 25 runs. The number of runs (out of 25) in which the global optimum is found and the average fitness of the population at the end of each of the 25 runs determines the quality of the genetic search.

For the probabilistic inference application, we run the programs to estimate the posterior conditional probability distribution of the query nodes in the belief network with 90% confidence intervals [3] to a precision of ± 0.01. The numbers reported are the average of the results for 10 runs.

Measurements of *warp* [2] [17] were done above PVM, for all the messages, to quantify the network load during the experiments. A particular measurement of *warp* at node i with respect to node j is given by the ratio of the difference in arrival times of two consecutive messages from node j to the difference in their sending times. *Warp* measures the rate of change of network load. The *warp* measured would be 1 when the network load is stable; *warp* values much higher than 1 indicate increasing load on the network. We also measure the null message latency over the network, during the runs of the application programs, to further quantify the network load.

We also studied the usefulness of Global_Read under heavy network traffic conditions by running the application programs for 4 nodes on the IBM SP2 along with a network loader program running on two other nodes. The network loader created intense traffic in the network. So, the results reported show the benefits of Global_Read under loaded network conditions.

5. Results

In this section, we report results from a systematic study of the synchronous, fully asynchronous, and partially asynchronous (i.e. Global_Read based) implementations of the Genetic Algorithms (GAs) and Bayesian Networks benchmarks discussed in section 4. Speedups for the parallel programs are reported with respect to corresponding sequential programs, which we optimized to a good extent. (e.g, for the sequential GA programs, we developed a software caching technique [32] to reduce the recomputation of fitness values of surviving individuals). Thus the speedups observed can be directly attributed to to the exploitation of parallelism.

All the results reported are for program runs on dedicated nodes (i.e. nodes with no other user processes) and hence the results do not reflect multi-programming effects. All experiments have realistic background network load, generated by other nodes on the network not allocated to the programs under study. But *warp* measurements have shown this network load to be quite low. We also report results for highly loaded networks by running a network loader program in parallel to generate desired network load levels.

[2] The time measurements required for computing *warp* were done at the granularity of microseconds

To separate out the benefits of Global_Read into benefits due to the elimination of synchronization overhead and benefits due to the tolerance of network loads and loadbalancing, we report speedups for Global_Read with age = 0. This setting removes the barrier synchronization overhead of the synchronous program but does not exploit any asynchrony in the algorithm, thus exposing the benefits of removing synchronization overheads alone.

We first present results for an unloaded network and then present results for heavily loaded networks.

5.1. Lightly Loaded Networks

Figures 3 and 4 show speedups of the different parallel implementations of the Genetic Algorithms benchmarks over corresponding serial implementations. Figure 6 shows corresponding results for the Probabilistic Inference benchmarks. We first discuss results for the Genetic Algorithms benchmarks.

5.1.1. Genetic Algorithms. Figures 3 and 4 show speedups over the serial programs for the synchronous, asynchronous, and different age settings (0, 5, 10, 20, and 30) of the partially asynchronous parallel programs. Also shown in Table 3 (and the last white bar in Figures 3 and 4) is the speedup of the best partially asynchronous program over the best competitor (i.e. the best age value for the Global_Read implementation versus the best of the synchronous, asynchronous, and serial programs). Results are shown for 2 to 16 processors.

We observe that the best partially asynchronous program is between 30% and 40% faster than the best competitor for six of the eight GA benchmarks, for different parallel platform sizes. The best case is 42% improvement for the first function, and four programs witness more than 35% improvement due to Global_Read. The base speedups of the parallel programs over the corresponding serial programs are significant and scale well with the number of processors.

We also observe that the age value 0 is typically not the best performer among the different partially asynchronous implementations. This indicates that most of the benefits due to Global_Read are because of the tolerance of network delay and load skew, and not just due to the elimination of synchronization overheads. We observe from the figures that both the synchronous and asynchronous versions do not scale well when the number of workstations rises above 8, whereas Global_Read scales significantly better.

Table 4 shows *warp* measurements as an indicator of the variation in network load during execution of the different parallel versions for one benchmark. Global_Read successfully reduces network load variation, as reflected by the reduced *warp* values compared to the asynchronous implementation.

In all the above runs, we ran the synchronous program for 1000 generations, and the asynchronous and controlled versions for enough generations so that the subpopulation converged further (i.e. better) than the synchronous version. Since the value reached by the asynchronous and controlled versions for any particular number of generations will differ across runs (since convergence rate is determined by run-time conditions also), the program trials were repeated 25 times and convergence beyond the required point was ensured for every trial.

One general observation that was made while running the experiments was the uncertainty in the results given by asynchronous runs. From the point of view of the user, who is running the program to get a result of a desired accuracy, there is not much certainty in the accuracy of the result he gets if he runs a certain fixed number of generations. Based on the run-time conditions, the values at different nodes will converge differently for different runs. It was observed that controlling the asynchronous program brings more stability and predictability to the result for any fixed number of generations [32]. This is another advantage of controlling asynchronous GAs.

Figure 3 Speedups of the synchronous, fully asynchronous, and partially asynchronous programs for the different GA benchmarks on the unloaded network.

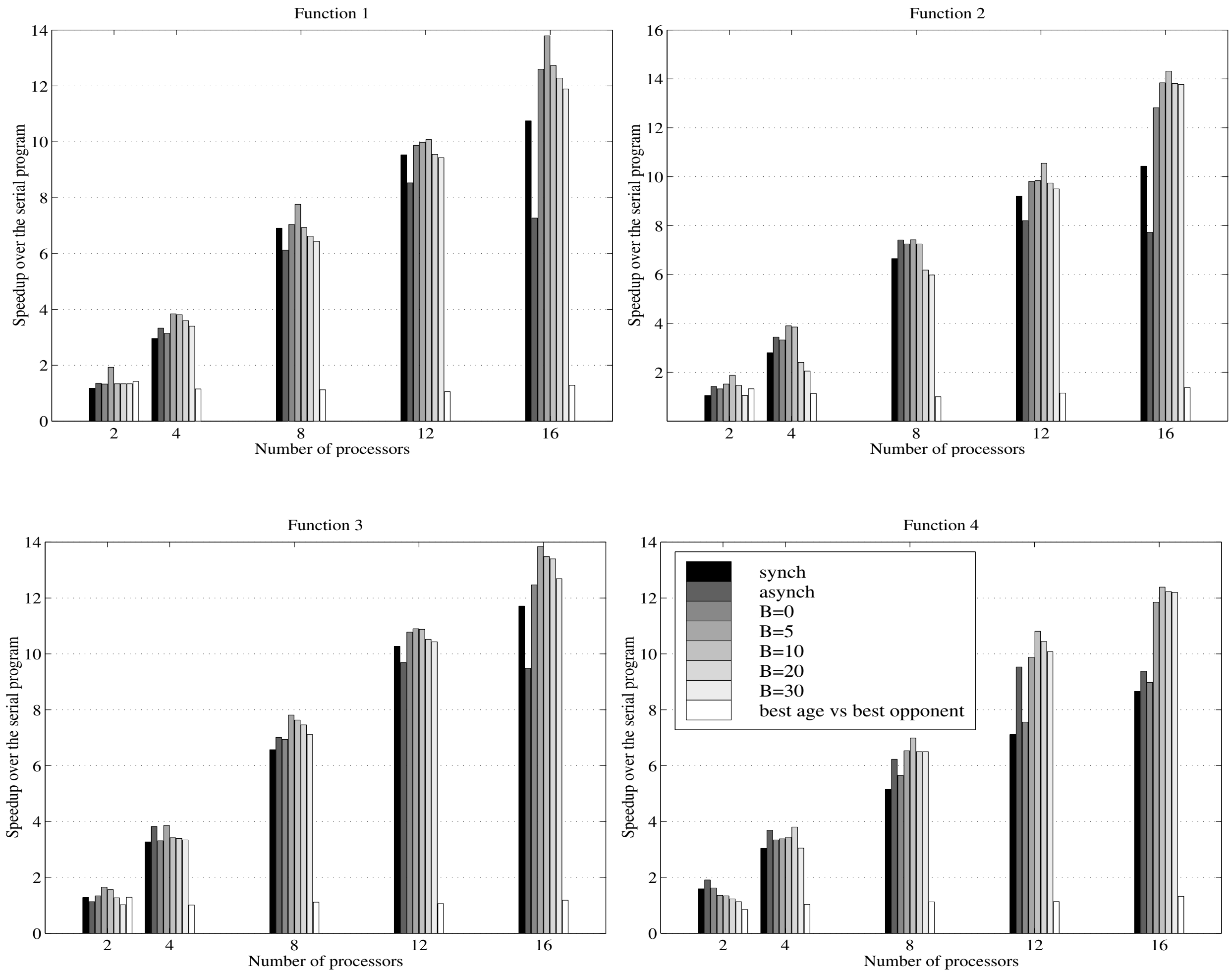


Figure 4 Speedups of the synchronous, fully asynchronous, and partially asynchronous programs for the different GA benchmarks on the unloaded network (contd.).

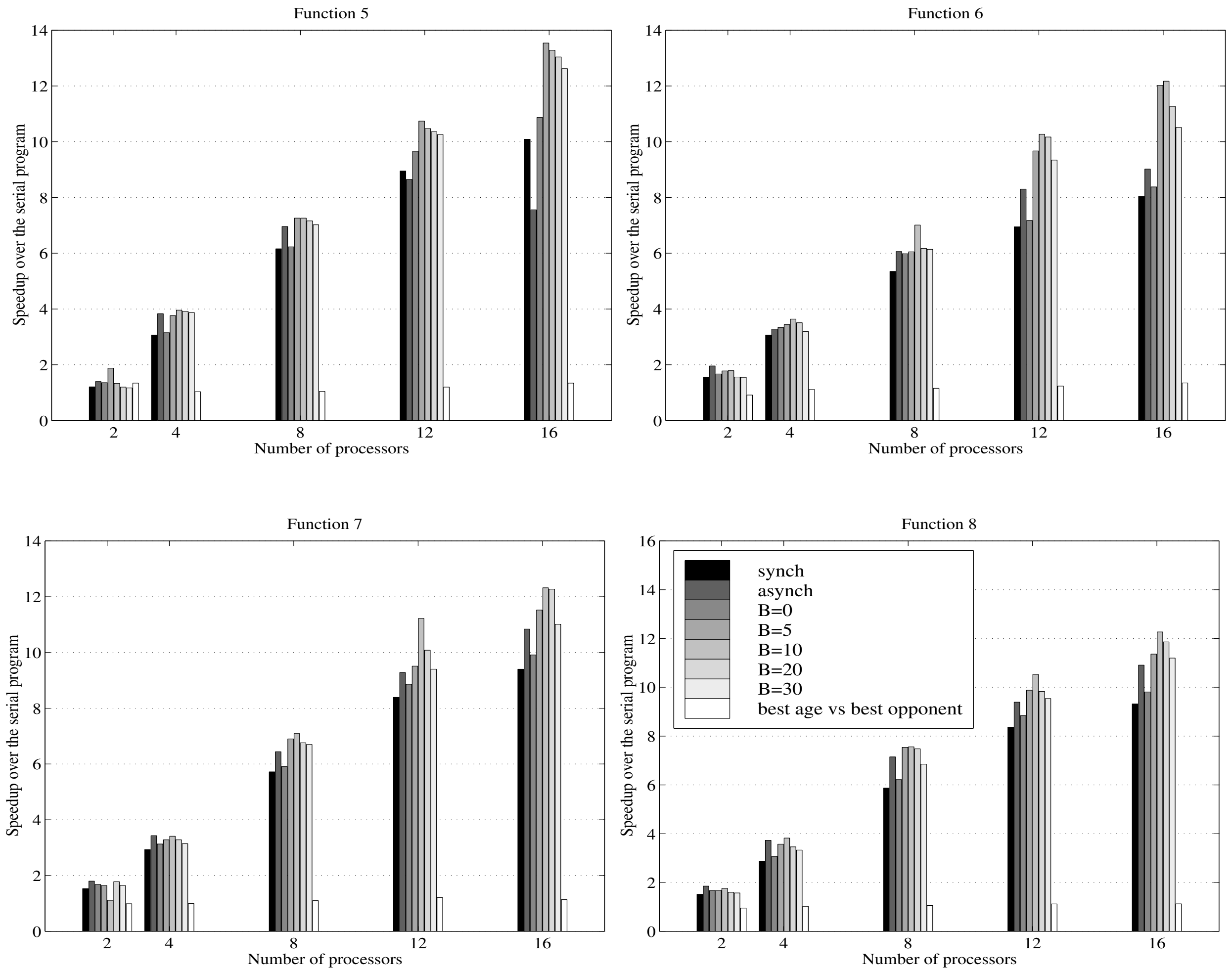


Figure 5 Average speedups of the synchronous, fully asynchronous, and partially asynchronous programs for the different GA benchmarks on the unloaded network.

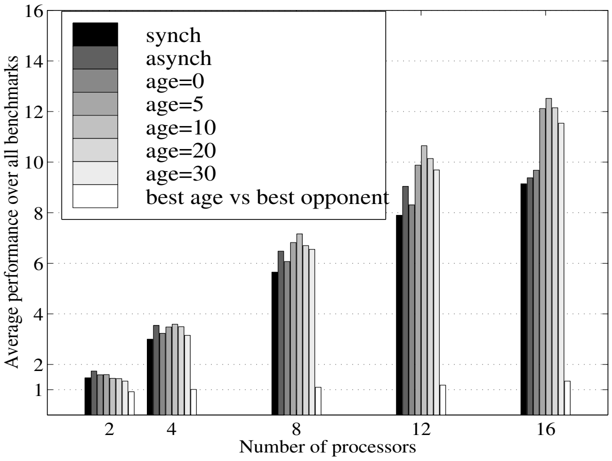


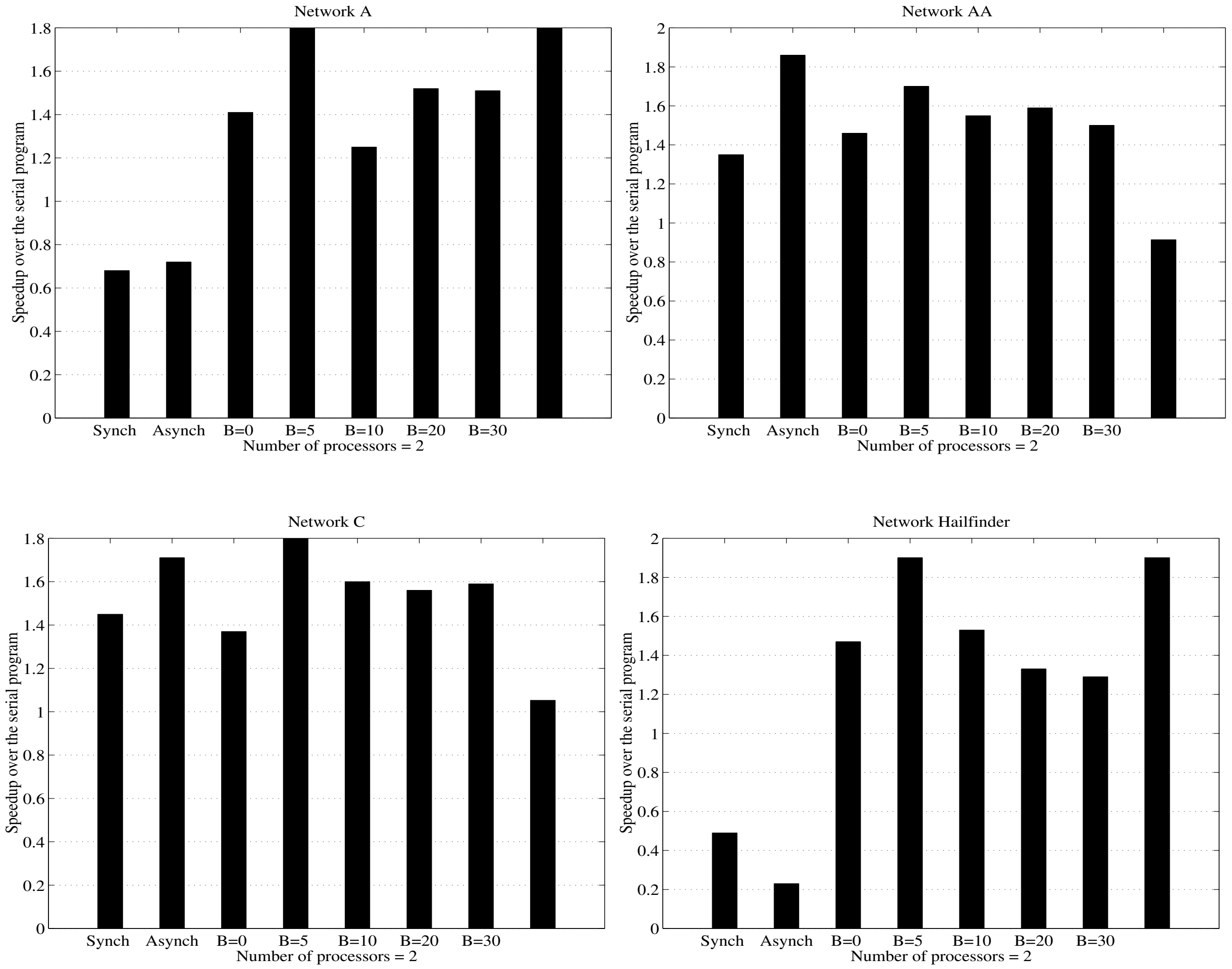
Table 3 Speedup of the best partially asynchronous program over the best competitor.

Function	Speedup of best Global_Read over best competitor	Number of processors
1	1.42	2
2	1.37	16
3	1.29	2
4	1.32	16
5	1.34	2
6	1.35	16
7	1.21	12
8	1.12	16

Table 4 Mean and standard deviation of *warp* measurements during execution of the different parallel GA versions for Function 2.

Program	Number of processors				
	2	4	8	12	16
Synch	1.01	1.01	1.01	1.01	1.01
	0.18	0.17	0.20	0.21	0.25
Asynch	1.13	2.74	11.74	15.21	21.14
	0.56	6.63	18.03	24.48	29.66
Age=0	1.31	1.20	0.62	1.03	1.01
	1.10	0.87	0.19	0.29	0.28
Age=5	1.05	1.30	1.57	2.30	2.07
	0.41	1.12	2.84	6.20	5.45
Age=10	1.25	1.67	2.24	2.34	2.39
	0.96	1.88	5.08	6.83	6.31
Age=20	1.11	1.80	3.28	2.43	3.10
	0.76	2.20	7.99	5.38	6.84
Age=30	1.23	2.18	4.57	3.73	4.89
	1.38	4.03	9.94	8.23	10.52

Figure 6 Speedups of the synchronous, fully asynchronous, and partially asynchronous programs for the different Bayesian belief networks on the unloaded network.



5.1.2. Bayesian Networks. Figure 6 shows speedups for the different parallel implementations of Probabilistic Inference on a 2-node configuration of the IBM SP2. The small networks available to us did not exhibit enough parallelism to be run on larger configurations. We use the small networks to predict the performance benefits of Global_Read for larger networks.

From Figure 6 we observe that the best partially asynchronous implementation is more than 80% faster than the best competitor for the real Hailfinder network and for the synthetic network A. While Global_Read does marginally better than the best competitor for Network C, it does worse than the asynchronous implementation for Network AA.

Again, we observe that the benefit due to removal of synchronization alone (Global_Read with age set to 0) does not account for all of the benefits of using Global_Read. An interesting observation is that for two of the networks, Global_Read with the age value setting of zero outperforms the fully asynchronous implementation, indicating the need for controlling asynchrony.

Figure 7 Average speedups of the synchronous, fully asynchronous, and partially asynchronous programs for the different Bayesian belief networks on the unloaded network.

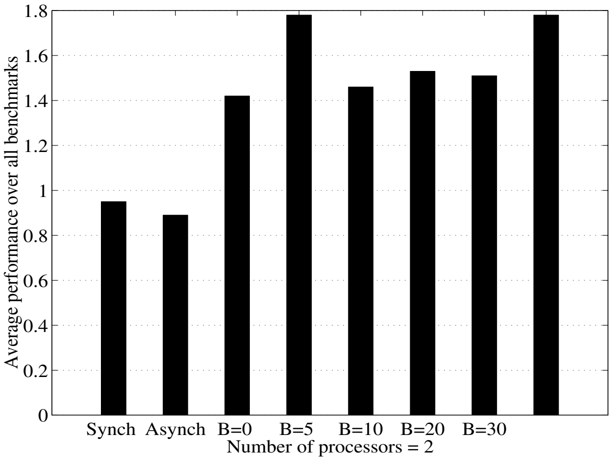


Figure 8 Speedups of the synchronous, fully asynchronous, and partially asynchronous programs for the different GA benchmarks on the loaded network.

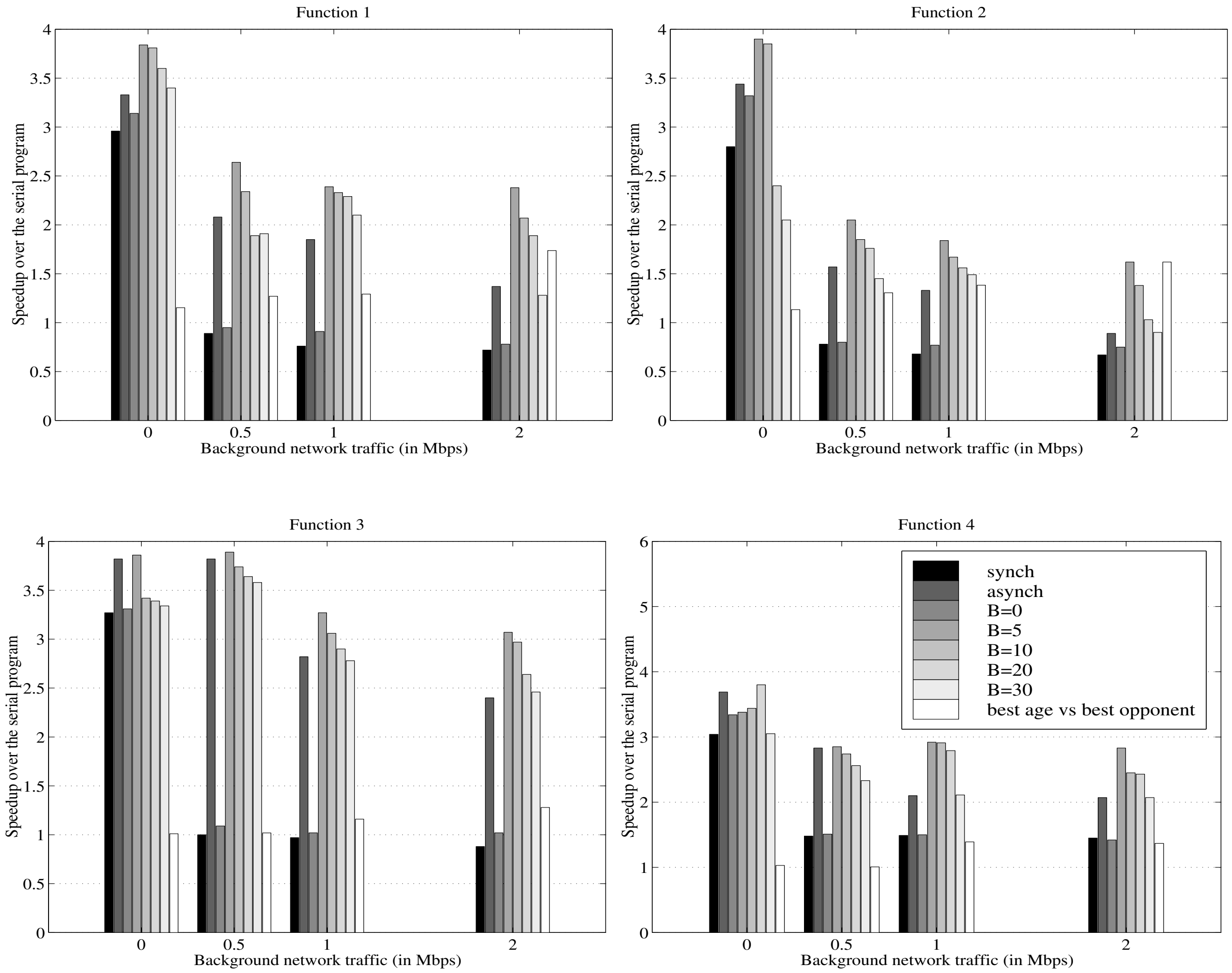
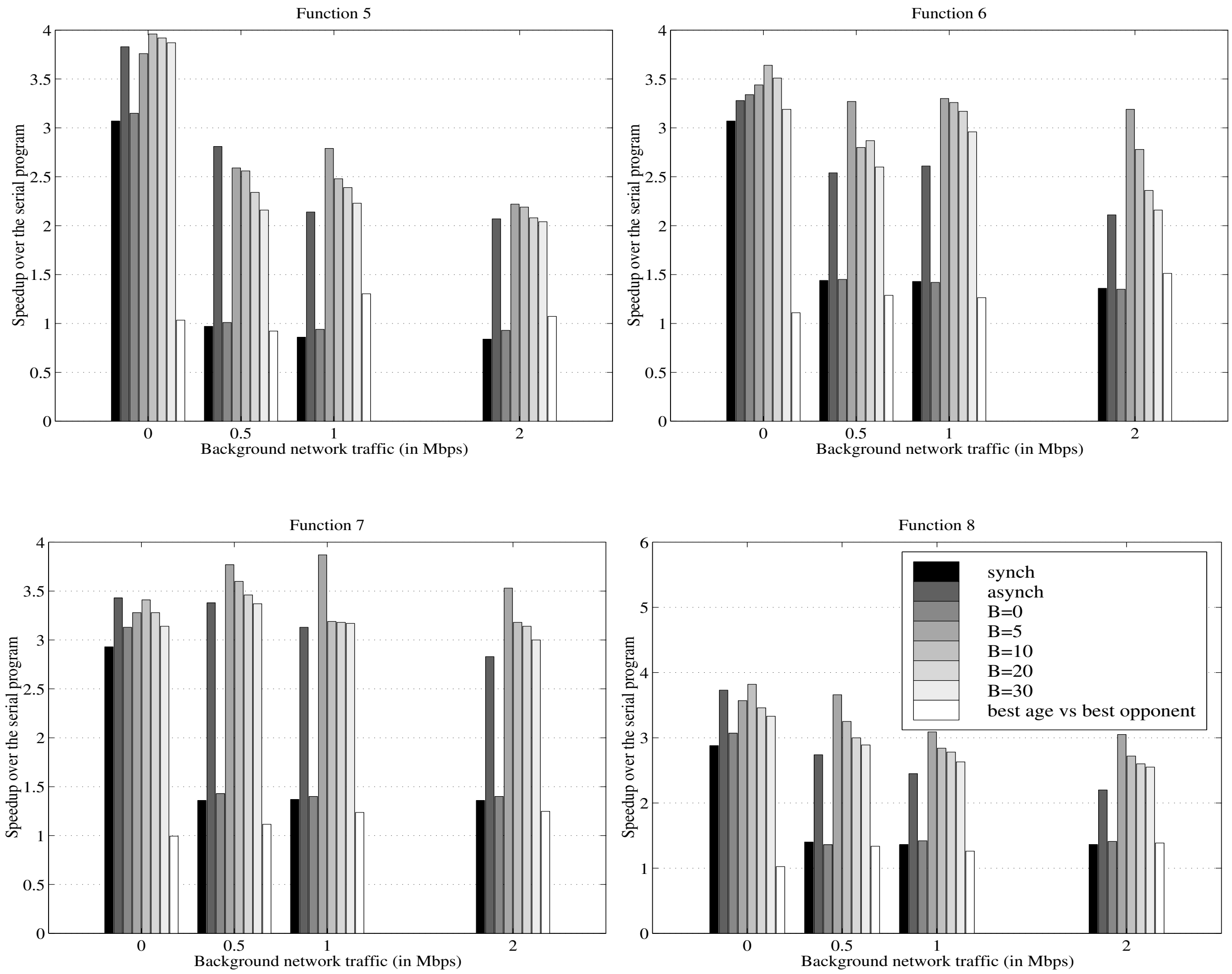


Figure 9 Speedups of the synchronous, fully asynchronous, and partially asynchronous programs for the different GA benchmarks on the loaded network (contd.).



5.2. Loaded Networks

Figures 8 and 9 show speedups for the Genetic Algorithms benchmark set for situations where the network is loaded. Network loads of 0.5 Mbps, 1 Mbps, and 2 Mbps are generated by a network loader program that runs in parallel with the benchmarks, on a separate pair of nodes. Due to node allocation policies, we were restricted to studying only a 4-node configuration (plus 2 nodes for the network loader program) for these experiments. Since the small Bayesian Networks available to us do not scale well to 4 processors, we do not report results for them.

Table 5 summarizes the benefits of the best Global_Read version over the best competitor for different levels of network load. We observe that the benefits of partial asynchrony are generally more when the network load is higher, touching as much as 70% for the GA benchmarks compared to 40% benefit on an unloaded network. As one goes from an unloaded network to a network with 2Mbps load, the benefits of Global_Read over the best competitor generally tend to increase. This buttresses our conclusion that Global_Read helps tolerate network delay for data-race tolerant applications.

Figure 10 Average speedups of the synchronous, fully asynchronous, and partially asynchronous programs for the different GA benchmarks on the loaded network.

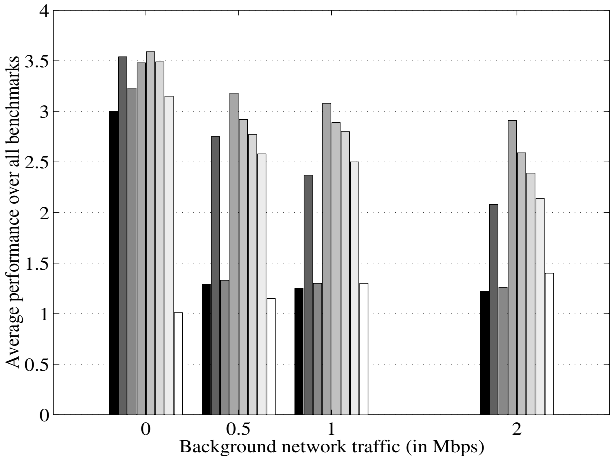


Table 5 Speedups of best Global_Read version over the best competitor for different levels of network load.

Function	Speedup of best Global_Read over best competitor	Network load (in Mbps)
1	1.74	2.0
2	1.62	2.0
3	1.28	2.0
4	1.39	1.0
5	1.30	1.0
6	1.51	2.0
7	1.25	2.0
8	1.39	2.0

6. Summary and Conclusions

We have studied the benefits of non-strict cache coherence for two emerging data-race tolerant application classes in this paper. We used benchmarks to represent the two important emerging applications - Genetic Algorithms (GAs) and Probabilistic Inference in Bayesian Belief Networks - in this paper. We evaluated the efficacy of the previously proposed memory access primitive, Global_Read, in implementing partially asynchronous parallel programs that exploit non-strict cache coherence. Partial asynchrony, i.e. controlled asynchrony, has been found useful in improving program performance, especially on a network of workstations where message delays are larger, and especially in the presence of large network loads and thus higher network delays. Naturally, programs with higher communication to computation ratio stand to benefit more from partial asynchrony.

On lightly loaded networks, the best partially asynchronous implementations had speedups between 30% and 40% over the best competitor for several of the Genetic Algorithms benchmarks. For two of the Probabilistic Inference benchmarks, the corresponding speedups were more than 80%. We also conducted experiments on loaded networks, with loads of 0.5 Mbps, 1 Mbps, and 2 Mbps. The results for the GA benchmarks indicate that as the network becomes more congested, the benefits of non-strict cache coherence and partial asynchrony increase significantly.

Controlled asynchrony trades off communication for increased computation to dynamically adapt to system load conditions, thus providing improved performance. We believe that such program level control of asynchrony promises to be superior to the previously proposed Warp control mechanism [17] that throttles message generation based on estimates of network load. The latter kicks in after the network load exceeds a threshold whereas program level control can prevent the initial setting in of congestion.

Future work includes a system-level implementation of Global_Read that allows the underlying system to exploit knowledge of staleness tolerance to dynamically adapt network and processor allocation policies. We also hope to study larger, real-life Bayesian Networks, and other emerging application classes such as neural-network based approaches. Also, to better understand and exploit the fact that different degrees of asynchrony are best for different programs and network loads, we are experimenting with dynamic (run-time) setting of tolerable age (staleness) levels when using Global_Read.

Acknowledgments

The work on genetic algorithms benefited from the many discussions the first author had with M. Narasimha Murthy. S. S. Ghosh, as ever, was an invaluable source of assistance in using the IBM SP2 multicomputer. P. J. Joseph provided helpful advice and comments on earlier drafts of this document.

References

[1] Decisions Systems Laboratory, University of Pittsburgh: The Hailfinder Project. Available on the WWW at http://www.sis.pitt.edu/ dsl/hailfinder/.

[2] S. Adve and K. Gharacharloo. Shared Memory Consistency Models : A Tutorial. *IEEE Computer*, December 1996.

[3] J. Banks, J. S. Carson, and B. L. Nelson. *Discrete Event System Simulation*. Prentice-Hall Inc., 1996.

[4] A. Beguelin, J. Dongarra, A. Geist, R. Mancheck, and V. Sunderam. A User's Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[5] D. P. Bertsekas and J. N. Tsitsikilis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[6] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In *Proc. of the Int'l Symposium on Computer Architecture*, June 1991.

[7] E. Cantú-Paz. A Survey of Parallel Genetic Algorithms. Technical Report 97003, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, May 1997.

[8] J. Carter, J. Bennet, and W. Zwaenopoel. Implementation and Performance of MUNIN. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[9] K. DeJong. *An Analysis of the Behaviour of a Class of Genetic Adaptive Systems*. PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, 1975.

[10] A. Geist, A. Begeulin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM : Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994.

[11] K. Gharacharloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[12] K. Gharacharloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990.

[13] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, Massachusetts, 1989.

[14] V. S. Gordon and D. Whitley. Series and Parallel Genetic Algorithms as Function Optimizers. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., 1993.

[15] J. J. Grefensette. Optimization of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 16(1), 1986.

[16] A. Heddaya and K. Park. Mapping Parallel Iterative Algorithms onto Workstation Networks. In *Proc. Int'l Symposium on High-Performance Distributed Computing*, August 1994.

[17] A. Heddaya, K. Park, and H. Sinha. Using Warp to Control Network Contention in Mermera. In *Proc.27th Hawaii Int'l Conference on System Sciences*, January 1994.

[18] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[19] P. W. Hutto and M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, June 1990.

[20] P. J. Joseph and S. Vajapeyam. Program-Level Control of Network Delay for Parallel Asynchronous Iterative Applications. In *Proc. 3rd Int'l Conference on High Performance Computing*, December 1996. Available at http://www.csa.iisc.ernet.in/publ/hipc96.ps.

[21] P. J. Joseph and S. Vajapeyam. Program-Level Control of Network Delay for Parallel Asynchronous Iterative Applications. Technical report, Computer Science & Automation, Indian Institute of Science, August 1996.

[22] P. Keleher, A. L. Cox, and W. Zwaenopoel. Lazy Release Consistency for Software DSMs. In *Proc. 19th Int'l Symposium on Computer Architecture*, May 1992.

[23] A. Kozlov and J. P. Singh. Parallel Implementations of Probabilistic Inference. *IEEE Computer*, December 1996.

[24] K. Li and R. Schaefer. A Hypercube Shared Virtual Memory System. In *Proc. of the Int'l Conference on Parallel Processing*, August 1989.

[25] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolutionary Programs*. Springer-Verlag, AI Series, New York, 1992.

[26] H. Mühlenbein, M. Schomisch, and J. Born. The Parallel Genetic Algorithm as Function Optimizer. In R. K. Belew and L. B. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., 1991.

[27] R. E. Neapolitan. *Probabilistic Reasoning in Expert Systems: Theory and Algorithms*. John Wiley and Sons, Inc., 1990.

[28] K. Park. Warp Control: A Dynamically Stable Congestion Control Protocol and its Analysis. In *Proc. ACM SIGCOMM*, September 1993.

[29] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, 1988.

[30] A. Singla, J. Hodgins, and U. Ramachandran. Temporal Notions of Synchronization and Consistency in Beehive. In *Proc. of the Symposium on Parallel Algorithms and Architectures*, 1997.

[31] H. Sinha. *MERMERA: Non-coherent Distributed Shared Memory for Parallel Computing*. PhD thesis, Boston University, Boston, MA, May 1993.

[32] S. V. Tambat. Study of Parallel Genetic Algorithms and Non-Strict Cache Coherence. M.E Project Report, Dept of Computer Science & Automation, Indian Institute of Science, January 1999.