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*

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to tolerate network latency: they do not synchronize shared memory references, and behave correctly when supplied outdated shared data.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 study the performance of controlled asynchronous implementations of these algorithms via the use of our 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. Experiments on an IBM SP2 multicomputer with an Ethernet show significant performance improvements for controlled asynchronous implementations. On a lightly loaded Ethernet network, most of the GA benchmarks see 30% to 40% improvement over the best competitor for 2 to 16 processors, while two of the Probabilistic Inference benchmarks see more than 80% improvement for 2 processors. As the network load increases, the benefits of non-strict cache coherence increase significantly.

1. Introduction

High performance networks of workstations are becoming increasingly popular as a parallel computing platform. Both message-passing and software distributed shared memory paradigms (DSMs) have been developed on such distributed platforms [16]. 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 and the number of messages transferred by supporting weaker shared memory models [16]. These models, however, are aimed only at programs which have a synchronous model of computation, i.e. *data-race* free programs.

A significant number of applications function correctly in the presence of data races, for example, iterative equation solvers, genetic algorithms, probabilistic inference in Bayesian belief networks etc. These applications offer an additional degree of freedom to address large data transmission latencies since they 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 albeit requiring more computation (iterations) to converge to the solution. This can give them a performance advantage over their synchronous counterparts since synchronization costs are avoided and communication can be overlapped with further computation. Typical asynchronous algorithms are iterative in nature, and their rate of convergence to a solution is critically dependent on the propagation delay of shared variables, though their correctness is not. The asynchronous nature of the communication allows the DSM to tradeoff computational efficiency (viz. the number of iterations executed) to (a) dynamically adapt better to network load, via techniques such as buffering [18], and (b) amortize message transmission overheads, by coalescing several updates of a single shared memory location. This tradeoff is done at a potential cost: 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 poten-

0-7695-0768-9/00 $10.00

tially flood the network with messages, moving the network to unstable conditions and thus unboundedly increasing the communication delay.

We have previously proposed [9] a method of controlling asynchronous algorithms wherein the message generation rate is controlled indirectly 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 since the receiver process cannot send its own messages. Eventually, this blocking of processes propagates to all nodes since in typical applications processes exchange messages rather than act exclusively as producers or consumers. In essence, the use of Global_Read converts a fully asynchronous algorithm into a *partially asynchronous* algorithm [2].

A preliminary performance evaluation of Global_Read was reported in [10]. 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 applications: (i) genetic algorithms (GAs) [5], and (ii) probabilistic inference in Bayesian belief networks using an approximate search algorithm [15]. 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 for 2 to 16 processors, while two of the Probabilistic Inference benchmarks see more than 80% improvement for 2 processors. As the network load increases, the benefits of non-strict cache coherence 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 [9] was proposed as a shared-memory access mechanism that enables the programmer to control 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. 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. 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 available, and either broadcasts a request for a copy of suitable age, or just waits until the required update arrives. In the former method, the receipt of a request for a new copy of a shared 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 [9]. Independently and around the same time, the *delta consistency model* was proposed [17] 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.

The performance potential of asynchronous algorithms on workstation clusters had earlier motivated the development of the software DSM system Mermera [18], which

supports non-coherent memory operations for usage by asynchronous algorithms. Mermera supports slow memory [8], 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 DSM system to reduce network load and amortize message overheads by buffering updates,and transmitting the multiple updates together in a single message.

A previously proposed method [7] for controlling asynchronous algorithms uses the Warp [14] 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 [7] or by entirely discarding the update and proceeding with further computation [6]. 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) [5] 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.

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 [3]. The population of the GA is divided into multiple sub-populations or *demes* 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 important issues that arise in the design of island GAs are the sub-population size and the migration of individuals between demes. Empirical studies [3] have shown that an island 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, finds the global solution in fewer number of generations compared to other island GAs.

The communication penalty and the overall execution time of island GAs can be often substantially reduced by means of an *asynchronous* implementation. An asynchronous implementation of the island GA 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 [20] 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 solution quality is also adversely affected by large message delays. A deme may converge quickly to its local optimum in the absence of any migrants from other demes for a long time, and so the overall solution quality 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 [15] is an important technique for reasoning under uncertainty. 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* [15]. 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 the dependencies among the outcomes of events 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 [12], thus necessitating parallel implementations.

Figure 1 An example Bayesian network

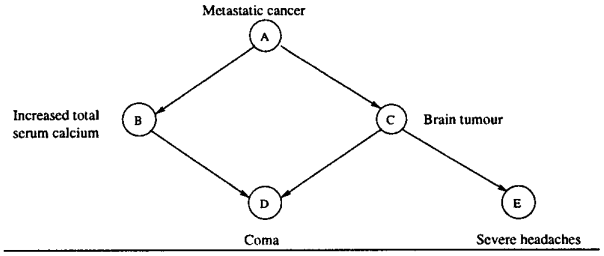


Figure 1 shows an illustrative belief network [15] for a case study in medical diagnosis. Each node takes on a set of values (here *false* or true) with probabilities that depend on the values taken by its parents. Each node has a conditional probability, which describes the probability that the node takes each of its values given a combination of values for its parents. For example, $p(D = true \mid B = true, C = true) = 0.80$.

Probabilistic inference determines the conditional probability of *query* node(s), given the instantiation of evidence of other nodes. Two types of approaches [15] are taken to find the desired conditional probabilities: exact algorithms, and approximate search algorithms. One of the approximate search algorithms is the *logic sampling* algorithm [15], which we consider in this paper.

Logic sampling 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 samples 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. This process is repeated to obtain multiple samples. 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.

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 practice, a subset of the network is assigned to each processor.) Each processor receives the values assigned to the parents of its nodes, and sends the values assigned to its nodes to the child nodes that belong to different processors. This procedure *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 logic sampling using a modification of the *synchronization via rollback* [2] 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. Whenever a processor needs the value of some interface node and the corresponding message has not been received, the processor uses a default value 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. In effect, each processor takes a gamble that the sampled value will be equal to the default value. If this is true for all iterations and for all the interface nodes, then it is evident that the simulation is completed in the least possible time.

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 [2], such as sending antimessages, to implement the rollback. 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.

4. Experimental setup

4.1. Parallel computing platform

All our experiments were done on a multicomputer orchestrated by the PVM message passing library [4], 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

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

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 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.

The benchmark parallel programs used in our studies are written 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. 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 with the Ethernet provided the most suitable platform for illustrating the benefits of non-strict cache coherence when the network latencies are relatively high compared to the applications' communication demands. Hence we report all our results for this platform. We expect that 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 [5, 13] that we use in this work. Our experiments are limited to a particular class of GAs characterized by the following six parameters [5]: population size (N), crossover rate (C), mutation rate (M), generation gap (G), scaling window (W), selection strategy (S). Based on DeJong's work [5], 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 a parallel GA can be used to explore different regions of the search space simultaneously, it is interesting to determine any improvement in the solution quality 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.

Table 1 Eight function test bed for GAs

No.	Function	Limits	min $f(x)$
1	$\sum_{i=1}^{3} x_i^2$	$-5.12 \leq x_i \leq 5.12$	0
2	$100(x_1^2 - x_2^2)^2 + (1 - x_1)^2$	$-2.048 \leq x_i \leq 2.048$	0
3	$\sum_{i=1}^{5} integer(x_i)$	$-5.12 \leq x_i \leq 5.12$	0
4	$\sum_{i=1}^{30} ix_i^4 + Gauss(0,1)$	$-1.28 \leq x_i \leq 1.28$	≤ -2.5
5	$0.002 + \sum_{j=1}^{25} 1/g(x)$, $g(x) = j + \sum_{i=1}^{2}(x_i - a_{ij})^6$	$-65.536 \leq x_i \leq 65.536$	0.99804
6	$nA + \sum_{i=1}^{20} x_i^2 - A\cos(2\pi x_i)$	$-5.12 \leq x_i \leq 5.12$	0
7	$\sum_{i=1}^{10} -x_i \sin(\sqrt{\lvert x_i \rvert})$	$-500 \leq x_i \leq 500$	-4189.83
8	$\sum_{i=1}^{10} x_i^2/4000 - \prod_{i=1}^{10} \cos(x_i/\sqrt{i}) + 1$	$-600 \leq x_i \leq 600$	0

Table 2 Four Bayesian belief networks

	A	AA	C	Hailfinder
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 (secs)	11.12	11.19	11.81	3.15

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 [12], i.e., 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 [12], 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 [11] 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.

An important metric for evaluating the performance of the parallel GA programs is the solution quality. 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 solution quality.

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 to a precision of ± 0.01. The numbers reported are the average of the results for 10 runs.

Measurements of *warp* [7] 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 studied the usefulness of Global_Read under heavy network traffic conditions by running the GA 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 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 [19] to reduce the recomputation of fitness values of surviving individuals). Thus the speedups observed can be directly attributed to the exploitation of parallelism. Due to space constraints, we do not report results for other metrics discussed in section 4.3 (these results are reported in [21]).

All the results reported are for program runs on dedicated nodes, and hence the results do not reflect multiprogramming 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.

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, 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.

5.1.1. Genetic algorithms. Figure 2 shows 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 for the best case (function 1), and the average performance over all the eight functions. (We measure the average performance by the ratio of the sum of the execution times for the serial program for all the benchmarks to that for the parallel programs.) The last white bar in Figure 2 shows 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 42% faster than the best competitor in the best case (function 1), and 34% faster on average. 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

Figure 2 Performance of the GA benchmarks on the unloaded network

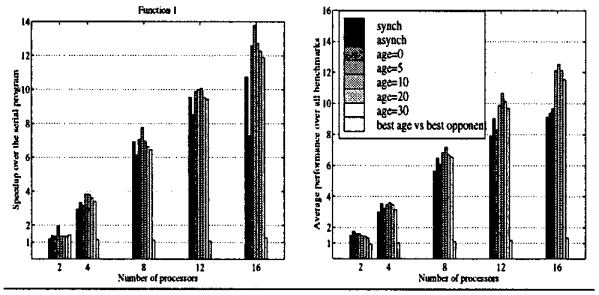
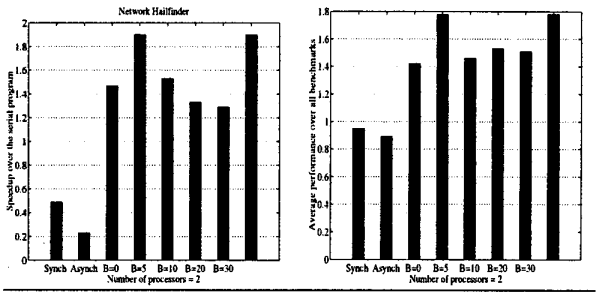


Figure 3 Performance of the Bayesian belief networks on the unloaded network



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.

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.

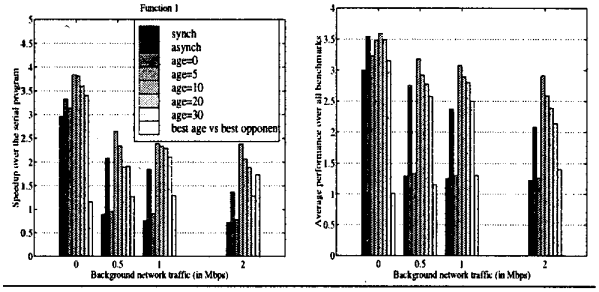
5.1.2. Probabilistic inference in Bayesian belief networks. Figure 3 shows speedups for the different parallel implementations of Probabilistic Inference on a 2-node configuration of the IBM SP2, and the average performance over all the belief networks. The small networks available to us did not exhibit enough parallelism to be run on larger configurations. We use the small networks to predict performance benefits of Global_Read for larger networks.

From Figure 3 we observe that the best partially asynchronous implementation is more than 80% faster than the best competitor for the real Hailfinder network, and 78% faster on average. 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.

5.2. Loaded networks

Figure 4 shows speedups for the GA benchmark function 1, and the average performance over all benchmarks when the network is loaded. Network loads of 0.5 Mbps, 1 Mbps, and 2 Mbps are generated by a network

Figure 4 Performance of the GA benchmarks on the loaded 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, we do not report results for them.

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 (for the best case). 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.

6. Summary and conclusions

We have studied the benefits of non-strict cache coherence for two emerging data-race tolerant 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. 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 GA 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 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 [7] 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 applications 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 (runtime) setting of tolerable age (staleness) levels when using Global_Read.

References

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

[2] D. P. Bertsekas and J. N. Tsitsikilis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall Inc., 1989.

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

[4] 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, 1994.

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

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

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

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

[9] P. J. Joseph and S. Vajapeyam. Program-Level Control of Network Delay for Parallel Asynchronous Iterative Applications. In *Proc. of the 3rd Int'l Conference on High Performance Computing*, December 1996.

[10] P. J. Joseph and S. Vajapeyam. Program-Level Control of Network Delay for Parallel Asynchronous Iterative Applications. Technical report, Dept. of Computer Science and Automation, Indian Institute of Science, August 1996. Available on the WWW at http://www.csa.iisc.ernet.in/~sriram/techrep/JoVa96-1.ps.

[11] G. Karypis and V. Kumar. METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices (Version 3.0.3). Technical Report 97-061, Dept. of Computer Science, University of Minnesota, 1997.

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

[13] H. Mühlenbein, M. Schomisch, and J. Born. The Parallel Genetic Algorithm as Function Optimizer. In *Proc. of the 4th Int'l Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., 1991.

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

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

[16] J. Protic, M. Tomasevic, and V. Milutinovic, editors. *Distributed Shared Memory Concepts and Systems*. IEEE Computer Society Press, August 1997.

[17] 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*, June 1997.

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

[19] 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.

[20] S. V. Tambat and S. Vajapeyam. A Case for Non-Strict Cache Coherence: Partially Asynchronous Genetic Algorithms on a Workstation Cluster. In *Proc. of the 8th Int'l Workshop on Scalable Shared Memory Multiprocessors*, May 1999.

[21] S. V. Tambat and S. Vajapeyam. Non-Strict Cache Coherence: Exploiting Data-Race Tolerance in Emerging Applications. Technical report, Dept. of Computer Science and Automation, Indian Institute of Science, August 2000. Available on the WWW at http://www.csa.iisc.ernet.in/~sriram/techrep/TaVa00-1.ps.