Program Level Control of Network Delay for Parallel Asynchronous Iterative Applications

P.J. Joseph Sriram Vajapeyam

IISc-CSA-TR-1996-7
http://archive.csa.iisc.ernet.in/TR/1996/7/

Computer Science and Automation
Indian Institute of Science, India

October 1996

Program Level Control of Network Delay for Parallel Asynchronous Iterative Applications

P.J.Joseph
Dept. of Computer Science & Automation

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

Indian Institute of Science
Bangalore, India
{peejay,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. Fully asynchronous parallel iterative algorithms offer an additional degree of freedom to tolerate network latency: they do not synchronize shared references and behave correctly when supplied outdated shared data. However, these algorithms have a high communication to computation ratio and can flood the network with messages in the presence of large delays. We propose a method of controlling asynchronous iterative methods wherein the reader of a shared datum imposes an upper bound on its age via use of a blocking Global_Read primitive. This reduces the overall number of iterations executed by the reader, thus controlling the amount of shared updates generated. Experiments for a fully asynchronous linear equation solver running on a network of 10 IBM RS/6000 workstations show that the proposed Global_Read primitive provides significant performance improvement.

1 Introduction

High performance networks of workstations are becoming increasingly popular as a parallel computing platform. Both message-passing [2] and software distributed shared memory [5, 4, 11] paradigms (DSMs) have been developed on such distributed platforms. An important performance bottleneck in these systems is network latency, which is poorer than in high-speed parallel computer interconnection networks. Software DSMs have attempted to reduce both the quantity of data used in the transfer and the number of messages by supporting weaker forms [6, 7] of the shared memory paradigm for example, [10]. The propagation to other nodes of the write to a shared location can be delayed in such memory models until the point of actual use of the data [10] indicated by the explicit synchronization present in the application program.

Applications that employ *fully asynchronous* iterative algorithms [1, 3] 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 unbounded losses and delays in the propagation of shared memory updates between nodes. This can give them a performance advantage over their synchronous counterparts [3] 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 [14], and (*b*) amortize message overheads by transmitting several shared memory updates in a single message [12]. The tradeoff of latency is done at a cost: the rate of convergence of asynchronous iterative methods is critically dependent on the communication delay. Each additional iteration needed to reach convergence results in additional shared memory updates and thus network traffic. When running on an already heavily loaded network, uncontrolled asynchronous algorithms can flood the network with messages and thus cause unbounded increase in communication delay. A previously proposed method [9] for controlling asynchronous algorithms uses the Warp [13] flow control protocol to adaptively throttle message generation by 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 [9] or by entirely discarding the update and proceeding with further computation [8]. Asynchronous algorithms also have the problem that a few lightly loaded nodes used 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.

In this paper, we propose an alternate method of controlling asynchronous iterative methods that is initiated 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 updates read by a node and thus control the rate of convergence. We propose a system supported blocking read primitive, termed Global_Read, that is guaranteed to return a value of acceptable age of the shared location specified. 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. 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. This limits the number of shared memory updates generated and unnecessary computation by stray processes.

The rest of this paper is organized as follows. In the next section, we give a brief overview of fully asynchronous iterative methods and describe an example application from this class that we use in our studies. In section 3 we present the Global_Read primitive. Section 4 describes our experimental setup. We discuss the convergence behavior of baseline versions of the example algorithm in section 5, to motivate the potential usefulness of the

Global_Read primitive. In section 6 we present some performance results for the algorithm using the Global_Read primitive. We summarize the paper in section 7.

2 Fully asynchronous iterative methods

Many algorithms for the solution of systems of equations, optimization and other problems have the structure

$$X(t+1) = f(X(t)), t = 0, 1, ...$$

where each X(t) is an n-dimensional vector $(x_1(t), ..., x_n(t))$ and f is some function from $\Re^n$ into itself. Such algorithms, since they apply an operator repetitively to a set of data, are called iterative algorithms. The operator f is applied repeatedly until $X(t+1) = X(t)$, when the iteration is said to have converged to a value. If such a value is not reached, the iteration is said not to converge for the given input and execution conditions. Such algorithms can be parallelized in a straightforward manner, with a node computing certain components of the vector, the values computed by a node being made available at the other nodes at subsequent time steps. The most straightforward parallel algorithm will make sure that the value computed at one node at one time step is available to all the other nodes in the next time step. This is a synchronous algorithm. But it has been found that if $f()$ satisfies certain conditions (for detailed explanations refer to [3]), the iterations will converge even if the values of $x_i(t)$ at a node are temporarily outdated with respect to other nodes, provided the local copies are eventually guaranteed to get updated [3, 8, 14]. Convergence is guaranteed independent of the upper bound B on the delay between the generation of a new x_i at a node and its propagation to all other nodes, provided that delay is finite. Iterative algorithms which make use of this property are called *fully asynchronous* iterative algorithms.

```
Epsilon = 0.000001
do { /* Global Termination Loop */
    do { /* Local Termination Loop */
        new_xi = f(LocalRead(x1),
                 LocalRead(x2),...,LocalRead(xn));
        AbsoluteDiff = abs(new_xi - LocalRead(xi));
        NonBlocking_Write(xi , new_xi);
    } while (AbsoluteDiff > Epsilon); /* iterate
          until Local Termination */
} until(global_termination()); /* iterate until
          Global Termination */
```

Figure 1. An Asynchronous Iterative Algorithm.

Figure 1 presents the pseudo-code for a general asynchronous iterative algorithm in a distributed shared memory system. The LocalRead and NonBlocking_Write primitives together build up a weak memory system in which the shared memory updates eventually propagate, but there is no upper bound on the delay. The DSM system should have such a weak memory system in order to fully exploit the features of fully asynchronous algorithms. The LocalRead returns the local copy of a shared variable x_i. The NonBlocking_Write updates the local copy of the shared variable immediately, and queues the new value with the system for propagation to all other nodes. For example, in the Mermera [14] implementation that employs Warp [9] flow control, these writes are queued in a local buffer which is propagated to other processors at a time determined by network flow control issues [8]. We observe that a LocalRead of a shared variable updated by the same processor always returns the latest value of the variable.

A common special case of an iterative algorithm arises when the function $f()$ is of the form $f(X) = MX + n$, where M is a square matrix and n is a vector. Such an iterative equation arises in an iterative solution to a linear system of equations. Consider such a system of equations, $AX + b = 0$, where A is an $(m \times m)$ matrix. Let a_{ij} be an element of A and b_i an element of b. Iterative equations of the form

$$x_i(t+1) = -\frac{1}{a_{ii}}(b_i + \sum_{j=1}^{i-1} a_{ij}x_j(t) + \sum_{j=i+1}^{m} a_{ij}x_j(t))$$

can solve the system of equations. The set of iterative equations can be written as

$$X(t+1) = MX(t) + n$$

where

$$\begin{aligned} m_{ij} &= -a_{ij}/a_{ii} \quad i \neq j \\ &= \quad 0 \quad\quad i = j \\ n_i &= -b_i/a_{ii} \end{aligned}$$

This iteration will converge, when run asynchronously, if the spectral radius of $|M|$(maximum of the eigen values of $|M|$) is less than 1 [3]. This condition will be satisfied if A is *row diagonally dominant.* That is

$$\sum_{j \neq i} |a_{ij}| < |a_{ii}|, \qquad \forall i.$$

The pseudo-code for this asynchronous algorithm, to be run on a DSM, will be just like that in Figure 1 with the $f()$ function being replaced by the iterative equation given above. It is this particular iterative algorithm, for the solution of a system of equations, that we use in all our experiments.

Let us consider the execution of an asynchronous iterative algorithm for the solution of a system of 10 equations as a small example. Consider the algorithm to be running for solving $AX + b = 0$ on 10 nodes. The (10×10) matrix A and the vector b are given and A satisfies the conditions required for the convergence of the fully asynchronous iterative algorithm. Each node, P_i, computes one element, x_i, of the vector X using the iterative equation. In each iteration it makes use of the available values of x_j, $j \in (1, .., 10)$. The NonBlocking_Write to x_i queues up the new value to be sent to the other nodes and computation proceeds. If a node P_i receives no updates of any shared location between two iterations, it will result in the node continuing the computation with the old values. The slow arrival of updates will not affect the guarantee of convergence. Of course, the convergence rate will be affected and there will be wasted iterations which repeat computations done in the previous iteration. The increase in iterations, and as a result the unnecessary message traffic, can become a problem when asynchronous iterations are run in an uncontrolled fashion.

3 The Global_Read primitive

We introduce the Global_Read primitive as a mechanism to reduce the unnecessary iterations and updates generated when asynchronous algorithms are run in uncontrolled fashion. This primitive can be implemented over and above the buffering and Warp flow control mechanisms that may be existing in the DSM system and will prevent the unnecessary iterations those methods cannot prevent. 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 data that is read. The maximum age is specified as the maximum number of iterations prior to the current iteration number that the shared data could have been generated. Thus Global_Read(locn, t, B) returns a value of "locn" generated no earlier than in the (t-B)'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 when the Global_Read is issued, 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 negligible considering the high communication-to-computation ratio of asynchronous iterative methods). Figure 2 shows how the purely asynchronous algorithm given in Figure 1 will be like, when modified to use Global_Read. The constant B used here 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 will have to check whether the age of its local copy meets the requirements. If not, it can either broadcast a request for a copy that is no older than the specified value, and block the reading process or just block the reading process until the required update arrives. In the former method, the node which is updating the variable has to reply with an acceptable version of the variable. We observe that 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. In the latter method there will be fewer number of messages.

```
#define B DESIRED_UPPER_BOUND
#define Epsilon 0.000001
t = 0;
do { /* Global Termination Loop */
    do { /* Local Termination Loop */
        t++; /* Iteration Count */
        new_xi = f(Global_Read(x1,t, B),...
            ...,Global_Read(xn, t, B) );
          AbsoluteDiff = abs(new_xi -
LocalRead(xi ));
        NonBlocking_Write(xi , new_xi);
    } while (AbsoluteDiff > Epsilon); /*
iterate
          until Local Termination */
  } until(global_termination()); /* iterate
until
        Global Termination */
```

Figure 2. Example of the Use of the Global_Read Primitive.

4 Experimental setup

We simulate distributed shared memory on top of PVM [2] to study the effect of the Global_Read primitive. This is done on a set of IBM RS/6000 workstations connected by an ethernet. The message communication and buffering required in a DSM as well as the Global_Read primitive were implemented as a set of library routines to be linked in by applications. The iterative linear equation solver described previously (section 2) was used in all the experiments. Typically it was run in parallel on 10 workstations. The input matrices A and b required for solving the equation $AX + b = 0$ were randomly generated taking care that A met the conditions for convergence. In the experiments reported, A is a (1000×1000) matrix and b is a 1000 row vector.

The chief metric of interest in our experiments is the number of iterations required for convergence. The completion time of the program was also noted. 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.

The underlying ethernet as well as the workstations on which the experiments are done are not dedicated. Measurements of *warp*[1] [13] were done above PVM, for all the messages, to quantify the network load during the experiments. A particular measurement of *warp* on 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. The *warp* measured would be 1 under ideal network conditions;*warp* values much higher than 1 indicate a loaded network. It was also ensured that the load average on the workstations was nearly zero whenever the experiments needed dedicated workstations.

DSMs designed for asynchronous algorithms use buffer-size as a flow control knob, accumulating updates in buffers until the buffer is full before transmitting the buffer. In some of our experiments we use appropriate statically-chosen buffer sizes. Buffer sizes can be dynamically controlled as in [9]. We do not address the issue of dynamic-control of buffer sizes in this paper.

Global_Reads were implemented by simply waiting for the required update to arrive, since that method has lower message overhead.

5 Baseline convergence behaviour

We first examine the convergence rates of several baseline versions of the algorithm to obtain an understanding of the potential for further performance improvement. In the three sets of experiments reported in this section, we use the same input for all the experiments within a set. Figure 3 plots the results of all the sets of experiments. Two of the plots in Figure 3 show the total number of iterations[2] required for convergence as a function of communication delay when the communication delay is artificially fixed at some value B throughout a particular execution. This results in each iteration of the algorithm reading shared values generated exactly B iterations ago on all processors. The experiment for B set to 1 corresponds to the synchronous version of the algorithm, wherein each iteration reads the latest values, generated by the immediately preceding iteration on all processors. This experiment provides a lower bound on the performance of the asynchronous algorithm on a real DSM with an upper bound B on communication delay. We observe that performance degrades linearly with B after a certain value. Thus, in a real

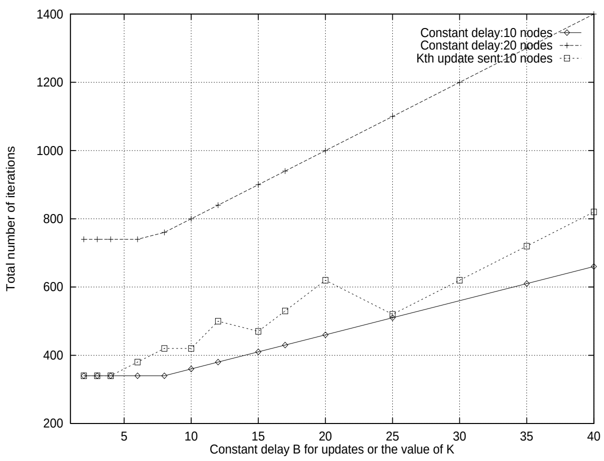

Figure 3. Performance for (i) various fixed communication delays B and (ii) shared updates transmitted only on every K-th iteration, with B=1.

[1]The time measurements required for computing *warp* was done at the granularity of milliseconds.

[2]It must be noted that all processes execute exactly the same number of iterations in the experiments reported in this section

DSM, controlling the delay B will improve performance beyond the lower bound on our algorithm. This will thereby reduce the overall number of shared memory updates generated. For the 20 node case the total number of iterations required across all the nodes is much larger than two times that in the 10 node case, for all values of B. This indicates that controlling the delay B will be of greater significance when we increase the parallelism. The third plot in Figure 3 reports results for an experiment where the communication delay B is fixed at 1 but each processor sends out shared memory updates to other processors once every K iterations rather than in every iteration. The iterations in-between update only locally. This experiment provides a lower bound on performance for a DSM where multiple updates (till a maximum of K) are not sent across the network because of system conditions or buffering. Here we observe that overall number of iterations increases with K. Thus, in real DSMs, prevention of update propagation could result in the generation of a larger number of shared memory updates.

6 Performance benefits of Global_Read primitive

Figure 4 shows a performance comparison between purely asynchronous iterations and iterations with Global_Read for various buffer sizes. The buffer size here represents the number of data values that was buffered. (Since the data values were of type double a buffer size of x means that $8 \times x$ bytes of space is required). In the iterations using Global_Read the buffer size was fixed depending on the value of the acceptable age, B. If B was large the buffer size was made large, but not so large as to create too many blocks at Global_Read. The plot compares results for the uncontrolled iterations and the controlled version which used the same buffer size. In this particular set of experiments alone we implemented a Global_Read which triggers requests across the network when the local copy of the variable does not satisfy the age limit. It must also be noted that the number of iterations plotted in Figure 4 are average values over four different trials. The numbers will vary depending on the workstation load and network load situations. The network load was relatively low in the four trials.

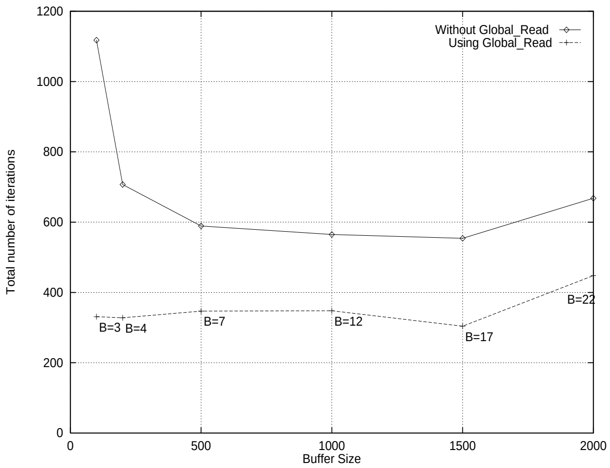

Figure 4. Performance of an uncontrolled fully asynchronous algorithm versus a controlled version.

Figure 4 shows that for any given buffer size the performance of uncontrolled purely asynchronous program is typically worse than the performance of a controlled version of the algorithm. The figure also illustrates another benefit of Global_Read. In realistic scenarios, it is possible that there is a skew in the load at each of the workstations in the network. It is also possible that process startup costs are different at different nodes. These factors lead to considerable asynchrony at the starting point of the different processes of the parallel program. When small buffer sizes are used, the uncontrolled algorithm executes a very large number of iterations (and thus a large number of shared updates and messages) because the processes that start early run ahead (without converging) while others are left behind. These processes send a large number of messages to the slower processes even before the latter have started up. When the slower processes eventually start up, they spend a significant amount of time processing the queued up messages rather than computing new values, thus further increasing the number of iterations executed by the faster processes. This is demonstrated by the large iteration counts in Figure 4 for small buffer sizes for the uncontrolled algorithm. The use of Global_Reads effectively squashes this problem.

We carried out *warp* [13] measurements for all the the above runs and it was found to be quite low, indicating low network load. We measured the average and standard deviation for the *warp* values computed for all messages in each trial. Mostly, average *warp* was between 1.1 and 2.0. For smaller buffer sizes the *warp* values were higher because of the traffic generated by the program itself. The standard deviation of the *warp* measurements in any trial was found to be between 0.9 and 2.5. The *warp* value of any message in any trial very rarely went above 25. These results show the network to be lightly loaded. In [9] it is reported that Warp control gives significant performance improvements only when *warp* values are very high. This shows that the performance gains reported for Global_Read are over and above what Warp control can provide.

The previous set of experiments does not explain whether the performance advantage of Global_Read is solely due to the initial skew in running the asynchronous algorithm. In order to prevent the initial skew we ran trials using a barrier just before the iterations started. We ran 20 different trials each for a purely asynchronous algorithm, a version with an initial barrier, Global_Read iterations with $B = 25$, and Global_Read with $B = 10$. A buffer size of 100 was chosen in all the experiments. (For the

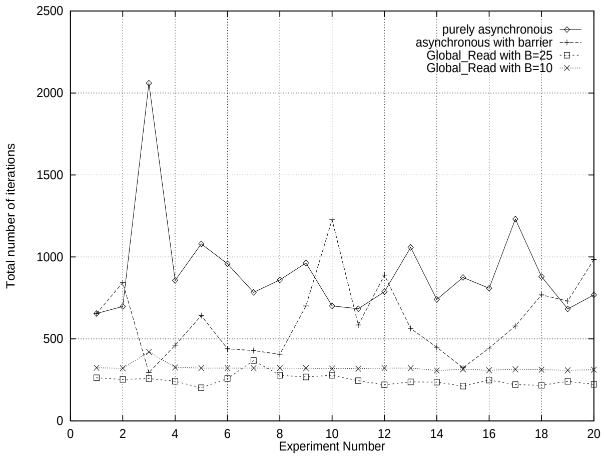

Figure 5. Performance variation across experiments.

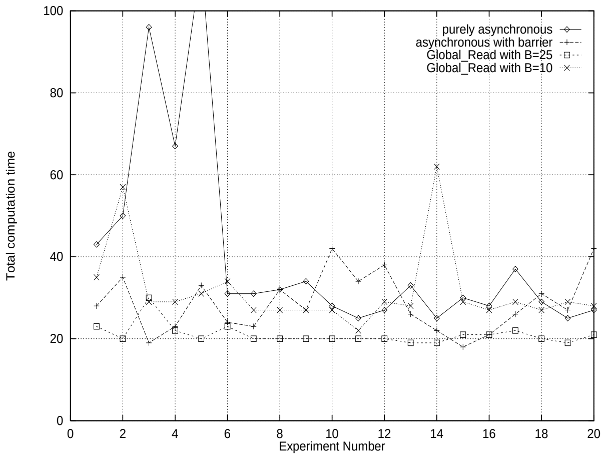

Figure 6. Time taken for the computation, across experiments.

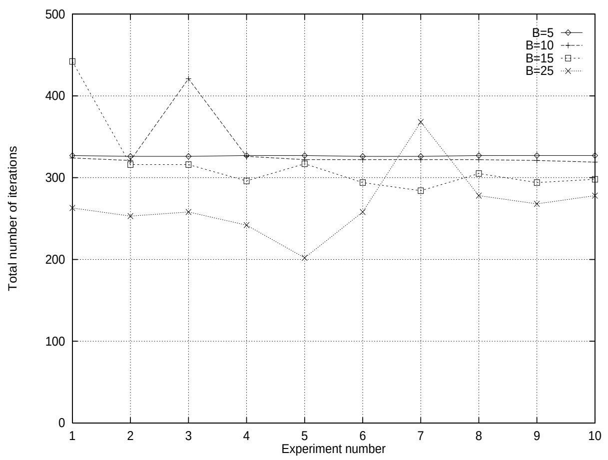

Figure 7. Variation in the number of iterations with B.

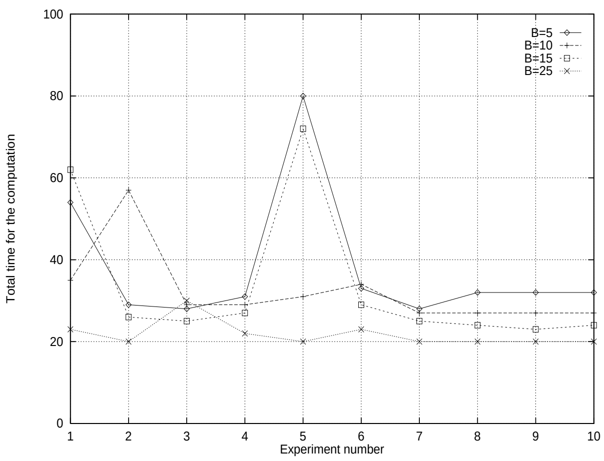

Figure 8. Time taken for the computation for different values of B.

input problem size to the program being run on 10 nodes, this resulted in one message every iteration). The total number of iterations observed in 20 independent trials of each of these sets is given Figure 5. The figure clearly shows that the controlled versions of the algorithm outperform the purely asynchronous most of the times even when there is no initial skew between the processes. Also, from the same figure, we see that the variations in the number of iterations, depending on workload situations, is much less for the controlled versions. Figure 6 plots the computation time taken for the 20 different trials of the four sets. It shows that usage of Global_Read does not adversely affect computation time and can even reduce it if the value of B is properly chosen.

6.1 Performance variation with B

We carried out a set of experiments to see the performance variation when we vary the value of B in the Global_Read primitive. We ran the Global_Read iterations with B=5 and B=15 in addition to the trials made in the previous set of experiments with B=10 and B=25. Figure 7 shows the total number of iterations taken for the various trials with the 4 different values of B, in the experimental setup described previously. Smaller values of B show lesser variations, in the number of iterations taken, across different trials. But using a larger value of B typically reduces the total number of iterations since more asynchrony is made use of there. Also, Figure 8, which shows the time taken for each of these experiments, indicates that larger values of B can reduce the computation time. This is because of lesser number of blocks at Global_Read, when the asynchrony is more. But the computation time will not decrease continually with B. For much larger values of B we can expect the time taken to more closely match that of purely asynchronous iterations since we are increasing the asynchrony. So the value of B has to be chosen appropriately in order to get the best performance.

6.2 Performance in a dedicated environment

We carried out all the previous experiments on a non-dedicated ethernet and set of workstations. So the workload affected the experiments as seen in the variations in the figures.

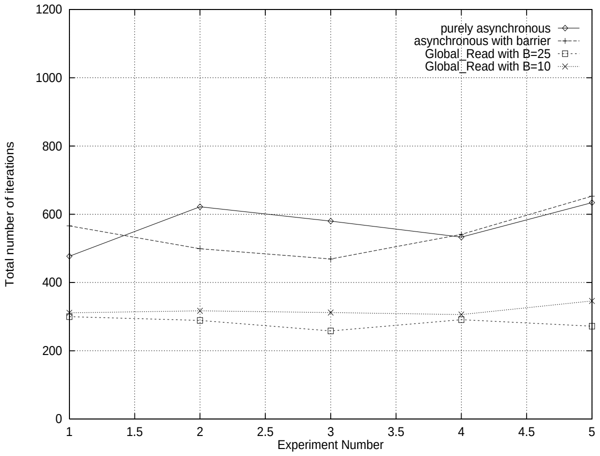

Figure 9. Total number of iterations under low workload conditions.

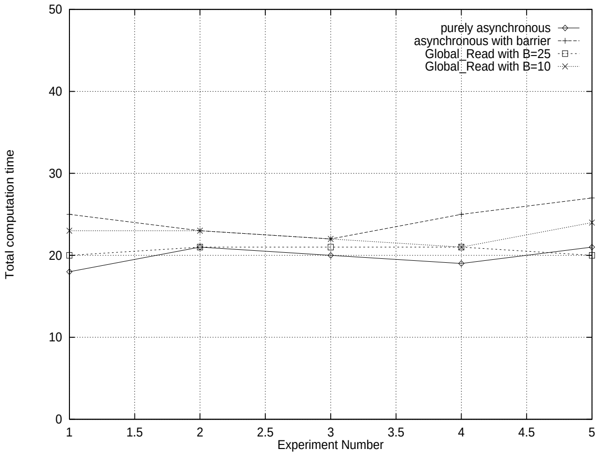

Figure 10. Time taken under low workload conditions.

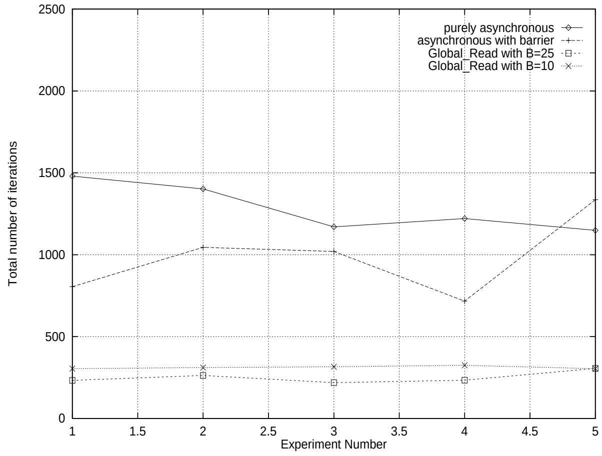

Figure 11. Total number of iterations under high network load conditions.

We ran a few number of experiments when the network load and the node workload were very low. The same four sets of trials done previously were repeated. But fewer trials were made. The total number of iterations taken and the time of computation are presented in the Figures 9 and 10 respectively. Global_Read clearly helps to reduce the number of iterations even in this case. The fact that the time of computation is more or less the same, even when node workload and network traffic are minimal, shows that Global_Read does not have much computational overhead.

6.3 Performance on a congested network

We studied the usefulness of Global_Read under heavy network traffic conditions by running the same four set of trials along with a program which created intense traffic in the network. These sets of trials were done when the network as well as the workstations were otherwise lightly loaded. We ran the traffic generation program on a different set of workstations, on the same network, from the ones which ran the trials. So the results reported show the effect of network traffic alone. The

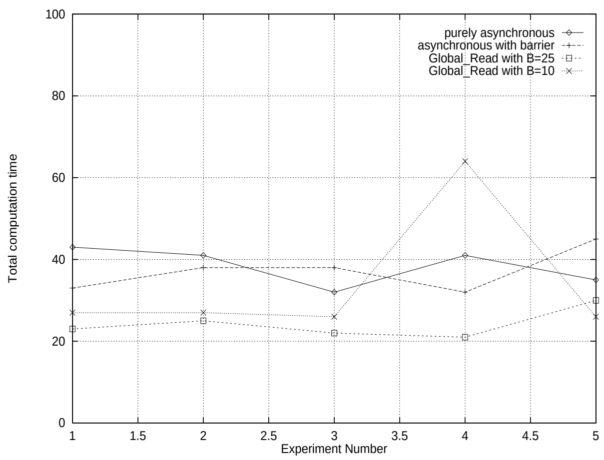

Figure 12. Time taken under high network load conditions.

Figures 11 and 12 show the tremendous usefulness of Global_Read under heavy network traffic conditions. The mean and the standard deviation of the *warp* values in a trial lay in the ranges 2.8-4.1 and 3.7-10.1 respectively. The number of iterations which is large when purely asynchronous iterations are run are kept controlled by Global_Read. It must be noted that even though *warp* measurements were made during the trials Warp flow control was not employed. This shows that Global_Read also works well in situations were Warp flow control has been found useful [9].

7 Summary and conclusions

We have proposed a program-level, message-receiver initiated method of controlling fully asynchronous iterations running on a network of workstations. The proposed primitive, Global_Read, allows the programmer to specify the maximum permissible age of a shared location being read. It has been found to be effective in reducing the total number of iterations required, and thus the number of shared memory updates (i.e,network traffic) generated by the applications. With the use of Global_Read the number of iterations required for convergence becomes much less dependent on network load and node workload conditions. Also, the method has been seen to be effective in preventing flooding of the network with messages when there is considerable skew in startup times of the different processes. All these improvements are without adverse effect on the computation time. In fact, with a proper choice of the age parameter B, the computation time can be improved.

The results reported indicate the potential usefulness of the proposed method in all the places where asynchronous iterations are used. Many long running scientific applications are currently parallelized using asynchronous iterations, in order to reduce synchronization penalty. The results reported clearly suggest that the use of Global_Read can bring significant improvements in the performance of such long running scientific applications which run on non-dedicated network of workstations.

Acknowledgments

We thank Matthew Jacob for several useful discussions and for comments on a draft of the paper.

References

[1] G. M. Baudet. Asynchronous Iterative Methods for Multiprocessors. *JACM*, 25(2), April 1978.

[2] 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.

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

[4] R. Bisiani and M. Ravishankar. PLUS: A Distributed Shared-Memory System. In

Proc. of the Int'l Symposium on Computer Architecture, June 1991.

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

[6] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. 13th Int'l Symposium on Computer Architecture*, June 1986.

[7] J. R. Goodman. Cache Consistency and Sequential consistency. Technical Report 1006, University of Wisconsin-Madison, February 1991. Originally appeared 1989.

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

[9] 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.

[10] 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.

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

[12] V. Nirmala. Distributed Shared Memory Implementation Across a Network of Workstations. M.E Project Report, Computer Science & Automation, Indian Institute of Science, Jan 1994.

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

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