Mapping parallel iterative algorithms onto workstation networks*

Abdelsalam Heddaya		Kihong Park
Computer Science Department
Boston University
Boston, MA 02215

Abstract

For communication-intensive parallel applications, the maximum degree of concurrency achievable is limited by the communication throughput made available by the network. In previous work [10], we showed experimentally that the performance of certain parallel applications running on a workstation network can be improved significantly if a congestion control protocol is used to enhance network performance.

In this paper, we characterize and analyze the communication requirements of a large class of supercomputing applications that fall under the category of fixed-point problems, *amenable to solution by* parallel iterative methods. *This results in a set of interface and architectural features sufficient for the efficient implementation of the applications over a large-scale distributed system. In particular, we propose a direct link between the application and network layer, supporting congestion control actions at both ends. This in turn enhances the system's responsiveness to network congestion, improving performance.*

Measurements are given showing the efficacy of our scheme to support large-scale parallel computations.

1 Introduction

With the advent of high-speed networks linking together an ever increasing number of high-performance workstations via local area (LAN) and wide area networks (WAN), harnessing their collective computing power for parallel computations has become a viable goal. *Supercomputing applications*, the prime benificiaries of such a venture, span a diverse spectrum of computational problems, ranging from partial differential equations to global weather simulation models and molecular sequence analysis. The methods of attack are not always uniform, some requiring only local interaction among processing elements whereas others are inherently global in nature.

*This research was supported in part by NSF under grants IRI-8910195, IRI-9041581 and CDA-8920936.

A large subclass of such applications falls under the category of *fixed point problems*, a class that is amenable to parallel iterative methods, synchronous or asynchronous [2, 3]. These include dynamic programming, systems of linear equations, network flow problems, genetic algorithms, and ordinary differential equations, just to name a few. This paper deals with the issue of how to map such applications to large-scale workstation clusters linked by local or wide area networks. In particular, we concentrate on asynchronous iterative methods that admit non-blocking, unreliable communication that can can be exploited to yield fast convergence.

Performance studies on LAN-based [5, 6, 16] and WAN-based systems [20] have shown the importance of controlling network delay for improving application performance. This is even more pronounced for parallel iterative algorithms since their communication/computation ratio tends to be relatively high, leading to the flooding of network resources if not managed properly. Parallel iterative algorithms possess a uniform communication pattern (*i.e.*, everyone needs to talk to everyone else with high frequency), and the single most important factor in determining the speed of convergence is the delay experienced by messages. For synchronous algorithms, it is obvious why this is so. For asynchronous algorithms that do not require waiting among processors, it is the outdatedness of the message itself which impedes convergence. That is, the longer the time-lag between the creation of a message and its reception at the destination, the less valuable it is in contributing to convergence.

Congestion control in heterogenous, high-speed networks is a difficult problem aggravated by the high propagation delay–bandwidth product and the different characteristics of multi-media traffic [1, 7, 8, 17]. One of the advantages of asynchronous iterative algorithms is that they do not require reliable message transmission. Therefore, they need not have access to reserved, connection-oriented channels advocated for delay sensitive ATM traffic [21]. Instead they can use the variable bit rate channels subject to statistical multiplexing. As such, the actual service rate available to the application distributed across a LAN/WAN is a dynamically changing variable, very much like how it is today. The retransmission cost of this class of traffic remains negligible due to the tolerance of message loss.

The positive effect of congestion control for solving systems of linear equations across an Ethernet LAN has been demonstrated in [10]. In this paper, we formalize and extend the results to an implementation-independent platform, applicable to both shared-memory and message passing distributed systems. We propose a generic design, represented as a set of protocol requirements, sufficient for the efficient implementation of parallel iterative algorithms. It is easily incorporable into existing distributed computing environments. In the context of B-ISDN systems, several papers have addressed inefficiencies associated with the overhead incurred by current network protocols [9, 16, 18]. One feature of our design, the *conditional send* (or **c-send**), establishes a direct link between network layer congestion control and application layer **send**, by making information about the network state available to the application. This eliminates wasteful protocol stack overhead when the network is congested, improving the performance of the system.

This paper is organized to mirror the structure of our proposed communication architecture. First, we give a brief overview of fixed-point problems, the class of applications that we aim to support. This is followed by an analysis of the communication costs associated with a generic parallel iterative algorithm that solves fixed-point problems. Next, we specify the problem of congestion, and show how a congestion control protocol can operate the communication network near its capacity, thus increasing the range of useful application parallelism. Section 4.2 presents the collection of application interface and operating system design requirements whose confluence suffices for the efficient iterative solution of fixed point problems on workstation networks. We end with a discussion of experimental results supportive of our conclusions, and describe work in progress.

2 Fixed-point problems

Many problems in mathematics, science, and engineering can be formulated as fixed-point problems. As an example, to solve the scalar equation $f(x) = 0$, it suffices to find the fixed point of $F(x) = x$, where $F \doteq f(x) + x$. In general, F acts on a multi-dimensional space, say $\mathcal{R}^m$, with a suitable norm $||\cdot||$, and the problem can be formulated such that F has a unique fixed point x^* to which $F(x)$ converges in the limit under discrete iteration,

$$x^* = \lim_{t\to\infty} F^t(x).$$

This is usually done by showing that F is a contraction mapping. Thus F is essentially a discrete dynamical system, and its iterative process can be expressed as

$$x_i(t) = F_i(x_1(t-\tau_1^i), x_2(t-\tau_2^i), \ldots, x_n(t-\tau_m^i)) \quad (1)$$

for $i = 1, 2, \ldots, m$, where x_i and F_i are the components of x and F, respectively. The τ's, themselves function of time, are delay terms reflecting the communication delay imposed by the system evaluating F. If, for all i,

$$\tau_1^i(t) = \tau_2^i(t) = \cdots = \tau_n^i(t),$$

then equation (1) defines a ***synchronous iterative algorithm***. Otherwise, it is said to define an ***asynchronous iterative algorithm*** for solving fixed-point problem F.

The iteration of F behaves differently under synchronous and asynchronous methods. As with classical iteration methods such as Jacobi and Gauss-Seidel [14], convergence may not be guaranteed, and when both converge, the asynchronous iterative method often converges faster than the synchronous one. For a comparative analysis of these two methods, see [2, 3]. Nevertheless, a large class of problems have been ***proven*** to be amenable to solution by asynchronous iterative methods [3].

The importance of asynchronous methods lies in the elimination of the ***synchronization penalty*** which can be very high in large-scale implementations. This enables them to execute more iterations, resulting in faster convergence. It is not always the case that executing more updates based on outdated information is beneficial to convergence, but under certain conditions, it can be shown that this is indeed the case [3]. The main drawback of asynchronous iterative methods over their synchronous counterpart lies in the increase in message transmission rate. For applications with high communication/computation ratios running on large-scale workstation clusters with shared network resources, this can overload the network, resulting in severe communication delays. Thus the goal lies in achieving as many iterations as possible without incurring a high delay penalty.

The fault-tolerance implied by equation (1) presents an additional degree of freedom in optimizing the network. This equation, with suitable assumptions on F and the τ's, can be interpreted to mean that the loss of messages carrying updated values can be tolerated without violating the correctness of the computation. In section 4.1, we exploit this feature to maximize the advantages of congestion control.

To fix notation and have a reference point to fall back on, consider the problem of solving a system of linear equations $Ax + b = 0$, where A is an $(m \times m)$ matrix. An asynchronous iterative algorithm to solve the above equation is given by the fixed-point iteration

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

With a suitable bound on the spectral radius of A, this iteration can be shown to converge when run asynchronously [2, 3]. In the implementation, we use this particular example to demonstrate the efficacy of congestion control.

3 Parallel asynchronous iterative algorithms

Let m represent the problem size (the actual problem size may be a function of m as in the linear equation example), and let n denote the number of nodes. A straightforward way to partition a problem of m variables over n nodes is to assign each node $k \doteq m/n$ variables. A generic parallel iterative algorithm can be described as follows. At each node i,

```
Iterative fixed-point algorithm:
  repeat
    receive $x_1, x_2, \ldots, x_m$
    update $x_{i \cdot k}, x_{i \cdot k+1}, \ldots, x_{i \cdot k+k-1}$
    send $x_{i \cdot k}, x_{i \cdot k+1}, \ldots, x_{i \cdot k+k-1}$
  until termination condition
```

where **send** and **receive** are both *asynchronous*.

The total cost of an iteration, measured in units of time, is given by

$$C = C_r + C_u + C_s$$

where C_r is the cost of **receive**, C_u is the cost of **update**, and C_s is the cost **send**. The termination condition cost is ignored here. Let M be the total number of messages produced per iteration at each node. node. For any node i, its message generation rate $\lambda_i = M/C$, and the total message generation rate for the entire system is $\lambda = \sum_i \lambda_i = n\bar{\lambda}$ (assuming fairness).

The costs, expressed as functions of m, n, depend on whether point-to-point or broadcast messages are employed. In the point-to-point case,

$$M \propto nk = m; \quad C_r \propto nk = m; \quad C_s \propto nk = m.$$

With broadcast, M and C_s change to

$$M \propto k = \frac{m}{n}; \quad C_s \propto k = \frac{m}{n}.$$

$M \propto m/n$ represents the most optimistic case, for example, when all the nodes are connected to a single Ethernet. For general routing networks, M will remain proportional to m. For positive constants K_1, K_2, and K_3, we have

$$\lambda_i \propto \frac{1}{K_1 + C_u/m}, \quad \text{and,} \quad \lambda_i \propto \frac{1}{K_2 n + K_3 + nC_u/m},$$

for the point-to-point and broadcast cases, respectively,

C_u is a function of both the problem size m, and the number of nodes n. For our purposes, it suffices to notice that the least computation needed for an iteration at a single node must include inspecting the m values it receives. Therefore, $C_u = \Omega(m)$. Note, $n \leq m$, and the finest granularity is achieved when $n = m$. Hence, for the point-to-point case,

$$C_u/m = \Omega(1) \Rightarrow \lambda_i = O(1) \Rightarrow \lambda = O(n),$$

and for the broadcast case,

$$nC_u/m = \Omega(n) \Rightarrow \lambda_i = \frac{1}{\Omega(n)} \Rightarrow \lambda = O(1).$$

From this analysis, we reach the following conclusions:

One, in the point-to-point case, increasing the number of nodes in the hope of speeding up the application in proportion to $1/n$ also carries the potential of inflating the network load proportional to n. When λ exceeds μ—the network's service rate—it triggers network congestion, resulting in severe message delays and slowing down the convergence of the algorithm. When mapping parallel algorithms onto distributed systems, we consider n to be large, hence we work in the realm of $\mu < \lambda$. Adjusting λ such that $\lambda \approx \mu$ becomes a central goal.

Two, other things being equal, broadcasting is superior to point-to-point communication with respect to reducing the likelihood of network congestion. In the most optimistic situation when $M \propto m/n$, increasing the number of workstations participating in the asynchronous iterative computation does not increase the input rate to the network. The imbalance created by the broadcast mechanism on the receive cost forces the nodes to spend an increased fraction of their time processing incoming packets.

Third, the time-complexity of the update cost C_u determines the communication/computation ratio, and directly influences the rate at which data is pumped to the network.

In the limited context when the structural aspects are all favourably aligned, controlling communication may not be a problem. Even then, the issue of minimizing the context-switch time between **receive**, **send**, and **update** processes by packing an adequate number of application messages into a single network packet looms as a potential variable to be controlled.

In the next section, we discuss the general case where congestion control is needed, the communication artchitecture appropriate for it, and measurements in support of our proposed architecture. As a side-effect, the congestion control procedure also carries the benefit of reducing excessive communication-induced context-switch cost, made possible by a positive correlation of the latter with network contention.

4 Controlling communication

4.1 Congestion control

For a fixed problem size m, given the trade-off between higher parallelism and increase in communication penalty for parallel iterative algorithms, is it possible to have the best of both worlds? That is, increase n without paying a high communication penalty?

The main penalties in the case of $\mu < \lambda$ stem from queueing delays and network congestion, the latter defined as a decrease in effective throughput, *i.e.*, a drop in μ, caused by tying up network resources in an unproductive way [15, 19]. This in turn aggravates queueing delays and triggers a positive feedback loop that worsens congestion. Congestion manifests itself in a decrease in C_R, hence increasing λ_i and establishing a positive feedback loop. That is,

$$\text{congestion} \Rightarrow \mu\downarrow \Rightarrow C_r\downarrow \Rightarrow \lambda\uparrow \Rightarrow \text{congestion}\uparrow$$

A *congestion control algorithm* may be viewed as trying to achieve two things [17]:

1. *Rate matching*, defined as $\lambda(t) \approx \mu(t)$, where $\lambda(t)$ and $\mu(t)$ are viewed as functions of time.
2. *Load matching*, a sufficient condition for long-term rate matching, where load is defined as the number of messages in transit. One way to model congestion is to assume an optimal load, Q^*, associated with a network, and view $\mu = \mu(Q)$ as a unimodal function of Q, reaching its maximum service rate, $\mu^* = \mu(Q^*)$, at Q^*. So, the goal is to keep $Q \approx Q^*$.

A distributed, end-to-end congestion control algorithm that captures the above aspects is Warp Control [17], and it is used in our experiments. The ideal place for congestion control to reside is at the network layer in the ISO-OSI reference model, and at the ATM adaptation layer (AAL) in the B-ISDN protocol model. This relieves communication-intenstive applications from having to worry about network issues. In the absence of operating system support for congestion control, implementation even at the applications layer can yield noticeable performance gains [10]. We argue that the decision not to send a message on a congested network should be made ***as high as possible*** in the system software architecture, *i.e.*, as close as possible to the point of an application's request to send.

Under asynchronous, unreliable communication, congestion control throttles the arrival rate to the network by discarding excess traffic. In essence, this has the same effect as increasing the unit message cost C/M. If, for example, messages produced from every other **send** are discarded due to heavy traffic, then

$$C/M = 2(C_r + C_u + C_s)/M$$

since effective generation of accepted traffic is done only at every other call to **send**. Unless discarded messages are ignored without any processing whatsoever, such messages incur an unnecessary overhead cost C_s. To eliminate this cost, some operating system support is desirable.

4.2 Design requirements

We have argued that, to harness the full power of speed-up for fixed-point problems solved by parallel iterative algorithms, we must apply congestion control to throttle the arrival rate to the network. Furthermore, such action should be taken at the earliest possible time, before wasted message processing overhead is incurred. This section sets forth the architectural features that suffice for such control to be affected, and specifies how they can be put together to achieve early discarding of messages without directly involving the application. Note, it is not necessary for the **receive** and the **send** to occur once per iteration, and in bulk, as is shown in section 3; these features serve only to simplify our presentation. The four design features are:

1. Asynchronous **send** and **receive**. That is, communication is non-blocking.
2. Unreliable transmission. (*E.g.*, UDP.)
3. Congestion control. It must be sophisticated enough to enable the network to maintain a sustained service rate close to its effective capacity.
4. Conditional send (**c_send**). In the same spirit as reducing the distance between application layer and communication layer [9].

Traditional message passing systems often provide an asynchronous, unreliable communication interface, and recently shared memory systems have begun supporting similar behavior [11].

We propose a new *conditional send* communication primitive which we call **c_send**. It is a **send** that has access to information about the network state, and if the network is congested, it degenerates into a null operation. In its simplest realization, let **b** be a Boolean variable that is **true** if the network is congested, and **false**, otherwise. Then **c_send** is equivalent to the following macro:

$$\texttt{c\_send}\, \vec{x} \;\; \equiv \;\; \textbf{if}\ \neg\textbf{b}\ \textbf{then}\ \texttt{send}\, \vec{x}.$$

Supporting a data structure such as **b** may require involvement of the operating system. The effect of having **c_send** is twofold. One, under heavy network traffic, by circumventing the overhead of having to go through the communication layers to reach congestion control, the otherwise wasted cost of C_s is eliminated (replaced by a cost of $O(1)$) for calls that would have resulted in the discarding of their payload. Two, by effectively increasing C_u/M and C_r/M, faster convergence and more efficient use of network bandwidth is achieved.

5 Implementation and experiments

In this section, we describe experiments of an implementation on a shared-memory environment called Mermera [10, 11]. All four design features (non-blocking **send/receive**, unreliable transmission, congestion control, and conditional send) were incorporated. The congestion algorithm employed is Warp Control [17], and the application being tested was the linear equation solver described earlier.

5.1 Mermera

Mermera [11] is a software shared memory system that provides a general-purpose environment for parallel computing on workstation networks. Processes comprising a parallel program reside on a specified group of nodes, and they communicate with each other via shared-memory read/write calls provided by Mermera. Several types of memory behavior are supported, one coherent (equivalent to sequential consistency), and two non-coherent: Pipelined Random Access Memory [13], and Slow Memory [12]. Mermera provides a **read** operation and three types of write operations: **co_write, pram_write,** and **slow_write**, the last of which is the one used by the linear equation solver during most of its computation. The **slow_write** and **read** operations capture the first two design requirements: asynchronous **send** and **receive**, and unreliable transmission.

There exist two versions of Mermera that incorporate the beforementioned design features. The first version is built on top of version 2.2.5 of the Isis toolkit [4] of multicast protocols that support different message ordering properties. That version of Isis employs point-to-point messages for multicasting. The second version of Mermera was redesigned to replace Isis with a communication interface based on UDP. This was necessitated in part by the need to apply congestion control at the lowest possible level, circumventing the mechanisms employed by Isis itself. It also enabled us to exploit Ethernet's hardware broadcast capability through UDP, thus reducing network contention. The data reported here is based on the first implementation using Isis, primarily because network contention can be made more severe due to the lack of broadcasting.

5.2 Warp Control

Warp Control [17] is a distributed, end-to-end congestion protocol that uses a time-stamp based scheme to throttle arrival rates for achieving optimal network utilization. Let N be the network characterized by two quantities, its service rate μ and its load Q. Let $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the message generation rates (i.e. arrival rates) of the n nodes. Let $\rho = \lambda/\mu$ be the network's utilization. The protocol itself, excluding the control, can be described as follows. Assume at time t_1, node j wants to send a packet m_{ij} to node i. At node j,

Message encoding protocol (MEP):

1. Create a header containing i, j, and time stamped with t_1. Denote the latter operation by $m_{ij}.time\_stamp := t_1$.

2. Attach the header to the data section of m_{ij} and submit to network N.

Let t_2 be the time at which m_{ij} arrives at node i. For all $k \in \{1, 2, \ldots, n\} \backslash \{i\}$, node i maintains a data structure $hist[k]$ containing two fields $hist[k].last\_in$ and $hist[k].last\_out$. In $hist[k].last\_in$ is recorded the time at which the last message from node k arrived at i, and $hist[k].last\_out$ records the time at which it was sent out from k. At time t_2, the following protocol is executed at the receiving node i:

Message decoding protocol (MDP):

1. $warp := (t_2 - hist[j].last\_in) \,/\, (m_{ij}.time\_stamp - hist[j].last\_out)$.
2. $hist[j].last\_in := t_2$.
3. $hist[j].last\_out := m_{ij}.time\_stamp$.

It can be shown under certain assumptions on N that the quantity *warp* as used in MDP approximates network utilization ρ [17].

Rate matching is achieved by the following control:

$$\frac{d\lambda_i}{dt} = \epsilon(1 - warp),$$

where ϵ is a parameter that governs the rate of change. This is called the rate adjustment protocol (RAP).

It can be shown that the system is asymptotically stable reaching its maximum utilization $\rho = 1$, if $0 < \epsilon < k/\bar{\tau}$, where $\bar{\tau}$ is the average network delay, and k is a positive constant of no concern here. Thus, if the network delay is high, then the rate constant has to be small to keep the system stable. For a detailed analysis and treatment of other related issues such as load matching and fairness, see [17].

5.3 Experiments

We conducted our measurements on a network of six dedicated Sun Sparc 1+ workstations and a server, connected by a private 10 Mbit Ethernet, and running SunOS version 4.1.1. We ran NetMetrix version 3.10, on the server and one of the workstations, to generate background traffic in order to increase contention for the Ethernet[1]. Although the communication/computation ratio of this application is relatively high, it was found that seven workstations were not enough to saturate the network. The remaining five workstations were dedicated to running the parallel linear equation solver on top of Mermera.

We measured the completion time and the state of the network when solving a system of linear equations in 1000 variables (*i.e.*, $m = 1000$). Figure 1 illustrates the level of improvement, in overall application performance, achieved via dynamic control over static buffering. The horizontal axis represents the buffer size for the case in which no congestion control is applied, and for the case in which only **c_send** is used. For the case when Warp Control is activated, it represents the initial buffer size (equivalently, the initial value of Λ_i).

[1]Congestion, in the context of a dedicated Ethernet LAN, corresponds to the collision rate on the network and buffer overflow.

Warp Control is implemented by modifying the message submission mechanism to interpose a buffer between Mermera and Isis that is flushed if and only if its size $B > \Lambda_i B_{max}$. The quantity Λ_i is maintained as an "inverse" of λ_i by modifying it in inverse proportion to the right-hand-side of the above control law. Thus, the larger the value of Λ_i, the smaller the message generation rate, and vice versa.

The **c_send** primitive was implemented by providing a direct link between congestion control and the Mermera "send" (*i.e.*, `slow_write`). All Slow Memory writes by the user are treated as **c_send**'s, making this mechanism both high level and transparent to the application program.

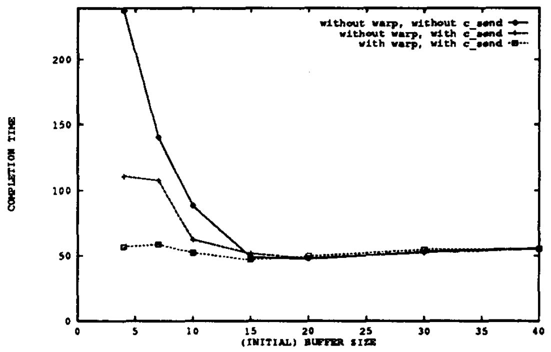
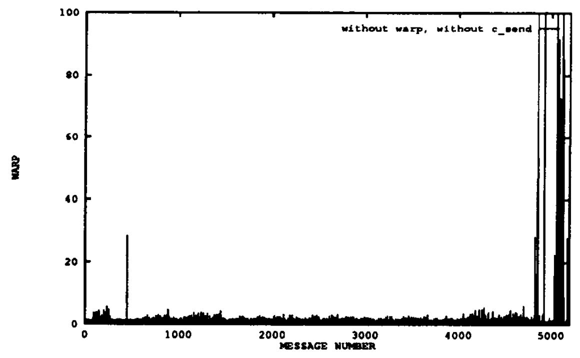


Figure 1: Time to solve linear equations in 1000 variables. Completion time improves dramatically when dynamic congestion control is applied, in comparison to static buffering.

We observe in figure 1 that conditional send enables the application to progress considerably faster on a congested network than it does otherwise, despite the crudity of our current binary implementation of **c_send**. However, further significant improvement is achieved by applying Warp at a lower level in conjunction with **c_send**. Although not shown here, Warp Control, even without **c_send**, is almost as effective. Also note that control is critical at smaller buffer sizes, and as the static buffer size is increased, its detrimental effect grows only slowly. We believe the sensitivity will be more pronounced in a workstation environment with many more nodes.

Figures 2 and 3 offer insight into the network dynamics, without (figure 2), and with (figure 3) **c_send** and warp control. The former shows how *warp* changes over time, represented indirectly via the message number shown as the horizontal axis. A value of 10 for warp at a particular message number means that the corre-



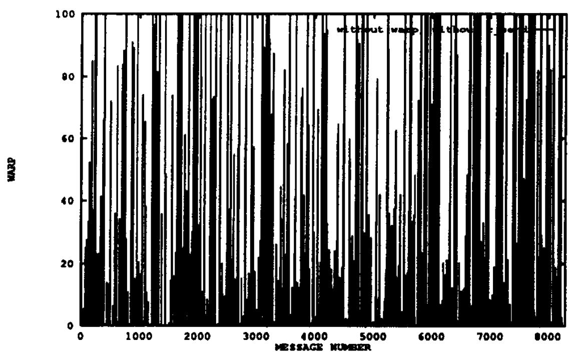


Figure 2: Congestion in the absence of control. Warp versus message number when only the equation solver uses the network (left), contrasted with the case when high artificial background traffic contends for the network (right). Case shown is for a static buffer size of 4 updates per message.

sponding message took 10 times longer to arrive at its destination, than the previous message from the same sender. Similarly, a value of one means that there has been no change in message delay. Figure 2-right indicates the dramatic contention that takes place as a result of injecting artificial background traffic to contend for the network with that generated by the application.

Thus, figure 2-right shows the network to be extremely congested before applying congestion control (figure 3-top), which squelches the wild fluctuations in delay. That warp's smoothness and closeness to a value of 1 causes a reduction in average message delay, is evidenced by the decrease in the total number of messages needed by the equation solver to complete: from 8250 down to 2103. Moreover, the effective network bandwidth in terms of the number of updates that are successfully transmitted (not shown) increases from 139 updates/sec to 571 updates/sec.

Figure 3-bottom indirectly traces the changes in message generation rate effected by warp control. The graph gives the value of Λ, which is proportional to the message size, for consecutive messages transmitted through Isis. Assuming a constant update rate by the equation solver, Λ is inversely proportional to the message generation rate. Thus, a change in Λ from 0.2 to 0.4, corresponds to the halving of the message generation rate (with the *caveat* that the horizontal axis does not directly represent time, so the rate of change in the message generation rate cannot be correctly deduced from the graph).

6 Conclusion

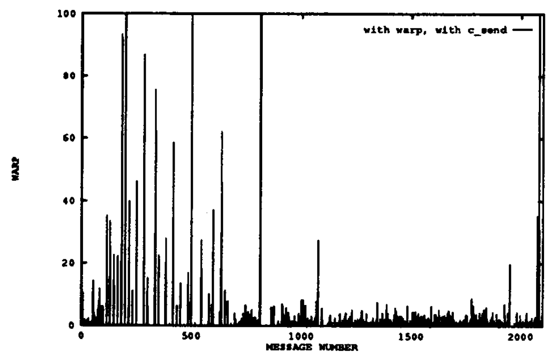
We have presented a framework for analyzing parallel asynchronous iterative algorithms for solving fixed-point problems, and its consequent system support requirements for efficient execution of such applications across LAN/WAN workstation networks. Experimental data from a seven workstation network connected via a single Ethernet support the efficacy of our system environment for enhancing application performance under heavy network traffic. The potential benefit of network control, inclusive **c_send**, can be extrapolated to large-scale workstation clusters involving hundreds of workstations where the need for control will be even more pronounced.

We remark in passing that our conclusions, in an overall sense, apply to other applications that do not admit nonblocking interfaces and loss of messages. In this case, the issue of network control should be totally divorced from the application, leading to control in a system-wide sense.

Acknowledgements. Arif Bhatti implemented the UDP version of Mermera, and participated with Sulaiman Mirdad in its design. A forthcoming technical report will describe that implemenation. We also thank Metrix Network Systems, Inc., for free use of their software.

References

[1] K. Bala, I. Cidon, and K. Sohraby. Congestion control for high speed packet switched networks. In *Proc. IEEE INFOCOM '90*, pages 520–526, 1990.



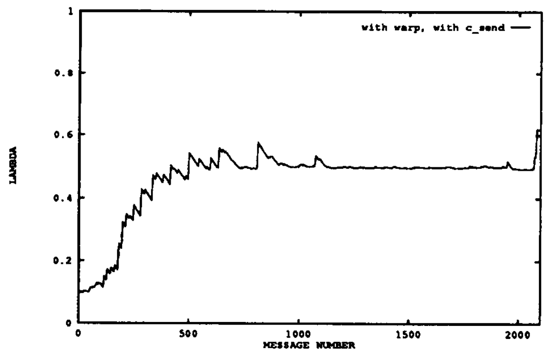


Figure 3: Warp control and c_send squelch congestion. Warp (top) and corresponding Λ (bottom) as a function of message number, under high congestion. A larger value of Λ corresponds to a bigger buffer, and a slower message generation rate λ. Initial Λ corresponds to initial buffer size of 4 updates per message.

[2] Gerard M. Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the Association of Computing Machinery*, 25(2):226–244, 1978.

[3] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and distributed computation: numerical methods.* Prentice-Hall, 1989.

[4] Kenneth P. Birman. The process group approach to reliable distributed computing. *Comm. ACM*, 36(12):37–53 and 103, Dec. 1993.

[5] Clemens Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.

[6] Alex Cheung and Anthony Reeves. High performance computing on a cluster of workstations. In *Proc. First International Symp. on High-Performance Distributed Computing*, pages 152–160, 1992.

[7] R. Dighe, C. J. May, and G. Ramamurthy. Congestion avoidance strategies in broadband packet networks. In *Proc. IEEE INFOCOM '91*, pages 295–303, 1991.

[8] A. E. Eckberg. B-ISDN/ATM traffic and congestion control. *IEEE Network*, pages 28–37, September 1992.

[9] Zygmunt Haas. A communication architecture for high-speed networking. In *Proc. IEEE INFOCOM '90*, pages 433–441, 1990.

[10] A. Heddaya, K. Park, and H. Sinha. Using warp to control network contention in Mermera. In *Proc. 27th Hawaii International Conference on System Sciences, Maui, Hawaii*, pages 96–105, 1994.

[11] Abdelsalam Heddaya and Himanshu S. Sinha. An overview of MERMERA: a system and formalism for non-coherent distributed parallel memory. In *Proc. 26th Hawaii International Conference on System Sciences, Maui, Hawaii*, pages 164–173, 1993.

[12] P.W. Hutto and M. Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *Proc. 10th IEEE Intl. Conference on Distributed Computing Systems, Paris, France*, June 1990.

[13] R.J. Lipton and J.S. Sandberg. PRAM: a scalable shared memory. Technical Report CS-TR-180-88, Princeton Univ., Dept. of Computer Science, Sep. 1988.

[14] Melvin J. Maron. *Numerical Analysis: A Practical Approach.* Macmillan, 1982.

[15] A. Mukherjee and J. Strikwerda. Analysis of dynamic congestion control protocols - a Fokker-Planck approximation. In *Proc. ACM SIGCOMM '91*, pages 159–169, 1991.

[16] M. Parashar, S. Hariri, A. Mohamed, and G. Fox. A requirement analysis for high performance distributed computing over LAN's. In *Proc. First International Symp. on High-Performance Distributed Computing*, pages 142-151, 1992.

[17] Kihong Park. Warp control: a dynamically stable congestion protocol and its analysis. *Journal of High Speed Networks*, 2(4):373–404, 1993.

[18] Thomas La Porta and Mischa Schwartz. Architectures, features, and implementation of high-speed transport protocols. *IEEE Network Magazine*, pages 14–22, May 1991.

[19] G. Ramamurthy and R. S. Dighe. Distributed source control: a network access approach to integrated broadband packet networks. In *Proc. IEEE INFOCOM '90*, pages 896–907, 1990.

[20] Volker Strumpen. Parallel molecular sequence analysis on workstations in the Internet. Technical Report 93.28, Department of Computer Science, University of Zurich, 1993.

[21] Marek Wernik, Osama Aboul-Magd, and Henry Gilbert. Traffic management for B-ISDN services. *IEEE Network*, pages 10–19, September 1992.