Congestion Control for Asynchronous Parallel Computing on Workstation Networks*

Abdelsalam Heddaya
Computer Science Department
Boston University
Boston, MA 02215
heddaya@cs.bu.edu

Kihong Park†
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
park@cs.purdue.edu

Abstract

Asynchronous parallel computing can result in high message generation rates, thus triggering network congestion. This paper investigates the network congestion problem that can result from asynchronous parallel programs' high message generation rates. First, we characterize the communication requirements of a large class of supercomputing applications falling under the category of fixed-point problems amenable to solution by parallel iterative methods. In particular, we concentrate on asynchronous iterative algorithms whose communication/computation ratio is especially high resulting in degraded effective throughput if communication is not managed properly. Second, we show the effects of network contention and asynchrony on application performance in a local-area network environment and investigate methods of solution. Our approach is based on a congestion control algorithm called Warp Control whose adaptive properties are exploited to yield significant performance enhancements when network contention is high. Although tested in a LAN environment for experimental control purposes, our solution follows the end-to-end paradigm and refrains from exploiting special MAC-layer properties to achieve applicability to general WAN environments. Third, we provide a framework wherein efficient congestion control can be facilitated, encompassing methods acting at the application layer as well as the transport/network layer, with emphasis on application-driven control. We conclude with a discussion of our experimental results and special issues arising in high-bandwidth ATM networks.

Keywords: Distributed parallel computing, congestion control, parallel iterative algorithms, communication architecture.

*This research was supported in part by NSF grants IRI-9041581 and CDA-8920936.
†This author was additionally supported by NSF grant CCR-9204284.

1 Introduction

With the advent of high-speed networks linking together a large number of high-performance workstations via local-area (LAN) and wide-area networks (WAN), harnessing their collective power for parallel computing 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.

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. These include dynamic programming, systems of linear equations, network flow problems, genetic algorithms, and ordinary differential equations, to name a few [2, 3]. This paper deals with the issue of how to control network congestion when mapping such applications onto workstation clusters linked by local and wide-area networks. In particular, we focus on asynchronous iterative methods that admit non-blocking, unreliable communication primitives thereby inducing a high communication/computation ratio which can result in increased susceptibility to network congestion.

Performance studies of LAN-based [7, 8, 15, 24] and WAN-based systems [30] have shown the importance of controlling network delay and throughput for improving application performance. This is even more pronounced for parallel asynchronous iterative algorithms due to their high communication/computation ratio, 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 an important factor in determining the speed of convergence is the average delay experienced by messages. For synchronous algorithms, the average delay manifests itself as synchronization penalty which can be shown to be "exponentially sensitive" in the number of nodes participating in the computation [14]. For asynchronous algorithms, processors do not need to wait for each other to proceed with the next iteration, and outdatedness of the messages is the main factor which impedes convergence. That is, the longer the time-lag between the creation of a message and its arrival at the destination, the less valuable it is in contributing to convergence. At the same time, such algorithms' tolerance of message delay and even loss, presents an opportunity for controlling network congestion by slowing the message transmission rate, independently from the application.

Although both synchronous and asynchronous algorithms are subject to network contention when the degree of parallelism is high, their individual realization is different. In the synchronous case, over the course of a single iteration, network contention is at its peak at the beginning of the communication phase (i.e., at the end of the computation phase) leveling off thereafter as more and more messages get delivered. In the asynchronous case where there is no notion of *global* iteration, the message generation rate stays constant at its maximum mean rate exerting a sustained burden on network bandwidth. Thus, synchrony has the side effect of throttling the application-generated mean arrival rate experienced by the network vis-à-vis the asynchronous case. From the perspective of congestion control, the bandwidth demand of synchronous iterative algorithms is subsumed by the more severe situation faced by asynchronous iterative algorithms, and in subsequent sections we will concentrate on the asynchronous case.

Congestion control in heterogenous, high-speed networks is a difficult problem aggravated by the high bandwidth-delay product and quality of service (QoS) requirements of real-time, multimedia traffic [1, 9, 10, 25]. One of the advantages of asynchronous iterative algorithms is that they do not require reliable message transmission. Hence, conditional on the severity of the packet loss rate, they may do without the access to reserved, high-priority connections advocated for constant bit rate (CBR) and variable bit rate (VBR) traffic in asynchronous transfer mode (ATM) networks [31]. Instead, access to best-effort or available bit rate (ABR) service of the ATM framework may suffice, very much like how it is today with IP. For synchronous iterative algorithms, reliable message transmission is a requirement. This can be achieved either by directly accessing QoS guaranteed VBR/CBR service or by implementing reliable service over ABR traffic through the use of higher level protocols such as TCP. A method for facilitating efficient, cost-effective communication channels via ABR service for parallel distributed computation is described in [14, 26].

In an ideal world, congestion control is applied at the network interface layer, providing pre-negotiated/default levels of service priority and fairness, achieving maximal transparency with respect to the application layer. This also allows for inter-application congestion control ensuring that all traffic flows remain well-behaved. Such unified and standardized computing environments are still some years away, and one is faced with the problem of application-driven end-to-end congestion control when implementing parallel computing applications over workstation networks. In the present approach, we construct a set of communication primitives with built-in congestion control features that reside above the transport/network layer and do not require special operating system support. The interface can be accessed by the application directly for message-passing-based communication, or it can be further encapsulated to provide a software shared-memory environment. The interface functionality and specification is modular and generic, and it can be straightforwardly incorporated into other distributed computing environments (e.g., PVM [11]).

This paper is an extension of previous work [15, 13] where the benefit of congestion control for solving parallel applications across an Ethernet LAN environment has been shown. We formalize, integrate, and extend the previous results, putting application-oriented congestion control on an implementation-independent platform applicable to both message-passing and shared-memory distributed systems. We propose a generic interface design represented as a set of protocol requirements sufficient for the efficient implementation of parallel iterative algorithms.

The rest of the paper is organized as follows. First, we give a brief overview of fixed-point problems, the class of applications we aim to support. This is followed by an analysis of the communication requirements associated with parallel iterative algorithms for solving 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, increasing the range of useful application parallelism. We describe the implementation set-up followed by an analysis of the performance measurements. We conclude with a discussion of issues related to high-speed ATM networks and the extrapolation of our results to many-node workstation clusters.

2 Parallel Iterative Algorithms

2.1 Fixed-point Problems

Many problems in mathematics, science, and engineering can be formulated as fixed-point problems. As an example [28], to solve the scalar equation $f(x) = 0$, it suffices to find the fixed point of $F(x) = x$ where $F(x) \equiv f(x) + x$. In general, F acts on a multi-dimensional space, say $\mathbf{R}^m$, with a suitable norm $|| \cdot ||$, and the problem can be formulated such that F has a unique fixed point x^* to which $F^t(x)$ converges in the limit under discrete iteration,

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

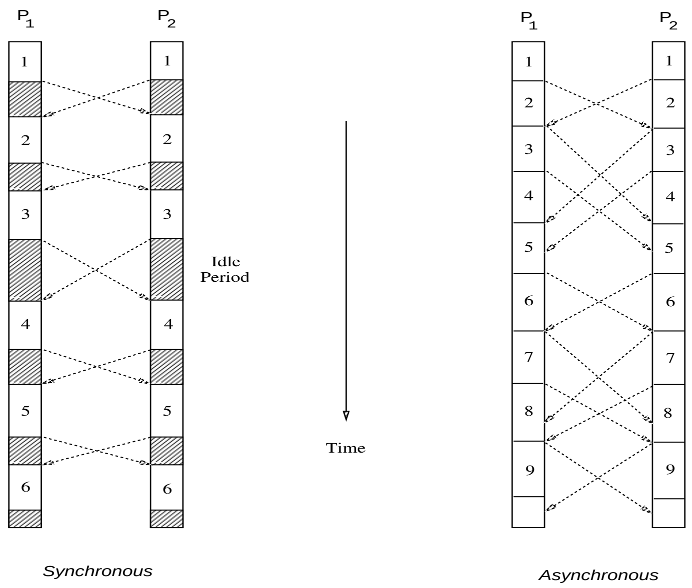

Figure 1: Synchronous (left) vs. asynchronous (right) communication in a 2-processor parallel iteration.

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

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

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 and t,

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

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 [22], convergence may not be guaranteed, and when both converge, the asynchronous iterative method often converges faster than the synchronous one. Nonetheless, a large class of problems have been *proven* to be amenable to solution by asynchronous iterative methods [3]. For a comparative analysis of these two methods, see [2, 3].

The importance of asynchronous methods lies in the elimination of the *synchronization penalty* which can be severe in large-scale implementations. In fact, under simplifying assumptions on the delay terms, it can be shown that synchronization penalty is "exponentially sensitive" in the number of nodes participating in the computation [14]. Thus asynchronous iterative algorithms are able to execute more iterations resulting in faster convergence other things being equal. Under certain conditions it can be proved that this is indeed the case [3]. It is not always the case, however, that executing more updates based on outdated information benefits convergence. The reduction in synchronization penalty does not come for free. The main drawback of asynchronous iterative methods over their synchronous counterpart is the increase in message generation rate. For applications with high communication/computation ratios running on workstation clusters with shared network resources, communication bottlenecks can arise counteracting the benefit of asynchrony. Thus, when transiting from synchronous to asynchronous algorithms for a given fixed-point problem—assuming this is possible—harvesting the benefits of asynchrony entails that network congestion be controlled so as to achieve as many iterations as possible without incurring a high delay penalty. Figure 1 illustrates the communication pattern of a 2-processor system implementing synchronous versus asynchronous iterations. Within the depicted time period, the synchronous algorithm is able to achieve only 6 iterations while the asynchronous algorithm executes 9.

Equation (1), in the asynchronous case, implies a degree of "fault-tolerance" which presents an additional degree of freedom for optimizing network communication. With suitable assumptions on F and the τ's, it can be interpreted to mean that the loss of messages carrying updated values can be tolerated without violating the correctness of the computation. As the gap between effective *reliable* throughput and *unreliable* throughput widens in B-ISDN ATM networks with high bandwidth-delay products, the benefits of unreliable communication may be further amplified.

To fix notation and also serve as our main reference point, let us consider the problem of solving a system of linear equations $Ax + b = 0$ where $A = (a_{ij})$ is an $(m \times m)$ matrix. A parallel 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.

2.2 Parallel Asynchronous Iterative Algorithms

Let m represent the problem size[1], 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 $k = m/n$ variables. We will ignore the issue of load balancing in this paper. A generic parallel iterative algorithm can be described as follows. At each node i,

```
ITERATIVE FIXED-POINT ALGORITHM:
  repeat
    receive x_1, x_2, ..., x_m
    update x_{i·k}, x_{i·k+1}, ..., x_{i·k+k-1}
    send x_{i·k}, x_{i·k+1}, ..., x_{i·k+k-1}
  until termination condition
```

where `send` and `receive` are *asynchronous* message passing primitives. That is, the function calls are non-blocking and no assumption is made on reliable message transmission.

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 of `send`. The termination condition cost is ignored here. Let M be the total number of messages produced per iteration at each node. For each node i, its message generation rate is given by $\lambda_i = M/C$, and the total message generation rate for the entire system is $\lambda = \sum_i \lambda_i = n\bar{\lambda}$ where $\bar{\lambda}$ is a fixed mean rate assuming fairness is preserved[2].

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

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

With broadcasting, M and C_s change to

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

[1] The actual problem size may be a function of m as in the linear equation example (quadratic).

[2] A scheme that achieves fair allocation of bandwidth given multiple competing sessions is described in [25].

$M \propto m/n$ represents the most optimistic scenario with respect to broadcast cost such as when all nodes are connected to a single Ethernet. For general routing networks, the broadcast cost will be greater than $O(1)$ depending on the topology. For some 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 note that the least computation needed for an iteration at a single node includes the task of inspecting the m values it receives. Thus, $C_u = \Omega(m)$. The finest granularity is achieved when $n = m$. Since $n \leq m$, in the point-to-point case we have

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

For the broadcast case,

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

From the previous analysis we can reach the following conclusions. First, in the point-to-point case, increasing the number of nodes to speed up the application by a factor approaching $1/n$ carries the potential of inflating the network load proportional to n. When λ exceeds the network's service rate μ, it may trigger network congestion resulting in high message delays and decreased effective throughput slowing down the completion time of the application. When mapping parallel algorithms onto distributed systems, we consider n to be large, hence working in the realm where $\mu < \lambda$ is possible, and controlling λ becomes an important objective.

Second, 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 case when $M \propto m/n$, increasing the number of workstations participating in the asynchronous iterative computation does not increase the mean rate of message flow to the network. However, the broadcast mechanism incurs the same total receive cost, which means that nodes spend a larger fraction of their time processing incoming packets, relative to the time consumed in preparing packets for transmission.

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 into the network. When the degree of parallelism relative to the available bandwidth is small, controlling communication may not

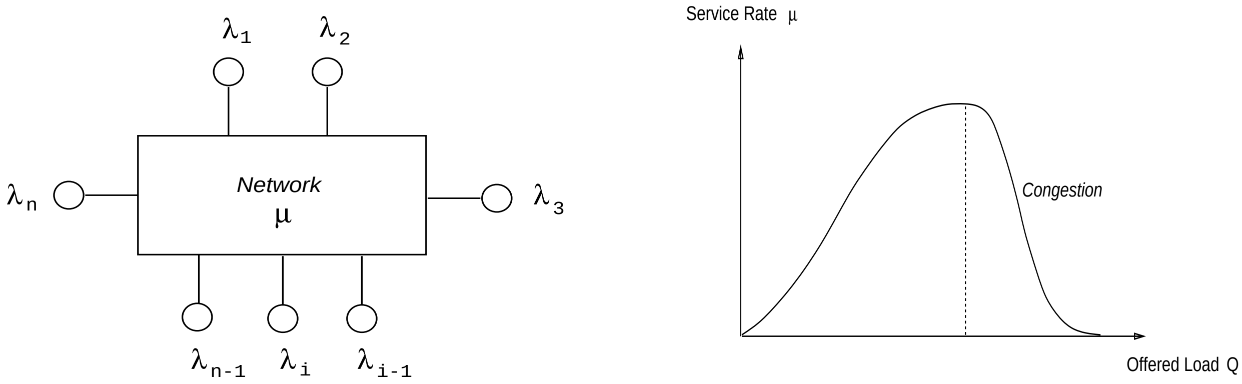

Figure 2: n-node network model (left). Unimodal load-throughput function (right).

be a problem. Nonetheless, in situations when packet size is variable, the issue of minimizing the context-switch time between `receive`, `send`, and `update` by packing an adequate number of application messages into a single network packet becomes more dominant.

3 Congestion Control

3.1 Congestion Model

In the previous section, we have analyzed the communication requirements of asynchronous iterative algorithms and their effect on aggregate message generation rate. In this section, we investigate the congestion problem experienced by the network and formulate methods for its solution. Let N be a network with n nodes, service rate μ, and arrival rates $\lambda_1, \lambda_2, \ldots, \lambda_n$. The total arrival rate is given by $\lambda = \sum_{i=1}^{n} \lambda_i$. N is a packet-switched local-area or wide-area network, and we make no assumption on its topology. The network is shared by applications distributed over the n nodes, and their message generation rates are reflected via $\lambda_1, \ldots, \lambda_n$. Figure 2 (left) is a schematic depiction of the network model.

There exist several characterizations of network congestion and its causes [23, 27]. In a packet-switched network, if the network is over-extended by the volume of incoming traffic, its resources become tied up in unproductive work servicing packets that may be ultimately dropped due to buffer overflow and time-outs. If, in addition, *reliable* transport is implemented at higher layers of the protocol stack (e.g., TCP), this may result in a positive feedback loop eventually leading to *congestion collapse* [18]. One way of modeling congestion phenomena is via a load-throughput function which relates effective throughput μ to offered load Q [12, 25]. Treating the effective bandwidth of a network as a service process with mean rate μ, the service rate $\mu = \mu(Q)$ can be viewed as a function of load Q possessing a *unimodal* shape capturing the congestion effect [25]. Figure 2 (right) depicts a unimodal load-throughput function with congestion kicking in if Q exceeds a certain level.

Two tasks need to be accomplished by the congestion control algorithm to optimize overall system performance. First, in the short run, $\lambda = \sum_{i=1}^{n} \lambda_i$ must be matched to μ to achieve high utilization. Second, in the long run, due to the variability of the arrival process, the total load in the system may accumulate resulting in a decrease in μ if the network becomes overloaded. Thus the system needs to "monitor" the overall load, throttling λ when Q approaches a critical level. These two mechanisms are denoted *rate matching* and *load matching*, respectively [25]. In a local area network consisting of few nodes, the store-and-forward feature of general packet-switched networks becomes diminished, and network state is more directly related to instantaneous total arrival rate $\lambda(t)$. Thus load matching is directly tied to rate matching and its short-term effects.

In the context of the cost-analysis framework of Section 2.2, congestion affects a decrease in the receive cost C_r, thus increasing λ_i and triggering a positive feedback loop:

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

The *slow start* component of TCP Reno is a congestion control feature that attempts to break this cycle by dampening the dependence of the message generation rate λ_i on C_r, but at the cost of overly conservative bandwidth utilization [19]. We note that there is an alternative approach to congestion control called link-based control which tries to achieve reliable packet transport at the link level through the use of credit tokens that conserve traffic flow in a hop-by-hop fashion [20]. This paper follows the rate-based framework with congestion control affected at the end points of the network connections.

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 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.

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 assuring maximum transparency. In the absence of operating system support for inter-application congestion control, per session/application implementations must be sought. We will show that in spite of the high level at which control is applied, significant performance gains are nonetheless obtained.

3.2 Warp Control

Warp Control [25] is a distributed, end-to-end congestion protocol that uses a time-stamp based scheme to throttle arrival rates for achieving optimal utilization. Let $\rho = \lambda/\mu$ denote network utilization. The protocol itself, excluding the control law, consists of a pair of encoding/decoding procedures which are 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 from k. At time t_2, the following procedure is executed at the receiving node i:

Message decoding protocol (MDP):

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

It can be shown under simplifying assumptions on N that the quantity $warp_{ij}$ as used in MDP approximates network utilization ρ [25]. Notice that $warp_{ij}$ is a function of both i and j. Rate matching is achieved by the following control law

$$\frac{d\lambda_{ij}}{dt} = \epsilon\,(1 - warp_{ij}) \tag{2}$$

where ϵ is a parameter that determines the rate of change, and λ_{ij} denotes the flow from node i to node j. Since $\lambda_i = \sum_{k \neq i} \lambda_{ik}$, the change in the total message generation rate at node i is given by

$$\frac{d\lambda_i}{dt} = \sum_{k \neq i} \frac{d\lambda_{ik}}{dt} = \epsilon \sum_{k \neq i} (1 - warp_{ik}).$$

Implementation details when channels are asymmetric are straightforward and omitted here. The control law given by (2) is called *rate adjustment protocol* (RAP). It can be shown [25] that the system is asymptotically stable achieving maximum utilization if the following condition holds:

$$0 < \epsilon < \frac{c\bar{\mu}}{\tau}.$$

where τ is the average network delay, $\bar{\mu}$ is the mean service rate, and $c > 0$ is a fixed constant. Thus, if network delay is high, the rate parameter ϵ has to be sufficiently small to maintain stability. For a detailed analysis and treatment of related issues including load matching and fairness, see [25].

4 Experimental Set-up

In this section, we describe performance measurements that capture the behavior of a representative parallel iterative application implemented synchronously and asynchronously on a workstation network. The experiments were carried out in the context of a shared-memory environment called *Mermera* [15, 16] in which Warp Control was incorporated as part of the message passing interface. The application tested was a parallel iterative linear equation solver, both synchronous and asynchronous.

We conducted our measurements on a network of six dedicated Sun Sparc 1+ workstations and a server, connected by a private 10 Mbps 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[3]. Although the communication/computation ratio of

[3] Congestion, in the context of an Ethernet LAN, arises from packet collisions and buffer overflow.

this application is relatively high, we found that seven workstations were not enough to saturate the network if all were engaged in the application. The remaining five workstations were dedicated to running the parallel linear equation solver on top of Mermera and Warp Control. The traffic generators ran at near maximum rate producing cross-traffic which were not subjected to congestion control.

4.1 Mermera

Mermera [16] is a software shared memory system that provides a general-purpose environment for parallel computing on workstation networks with special support for asynchronous iterative programs. Processes comprising a parallel program reside on a specified group of nodes and they communicate with each other via shared-memory read/write primitives provided by Mermera. Several types of memory behavior are supported, one coherent (equivalent to sequential consistency), and two non-coherent: Pipelined Random Access Memory [21] and Slow Memory [17]. Mermera provides one `read` operation and three types of write operations: `co_write`, `pram_write`, and `slow_write`. The `slow_write` and `read` primitives essentially provide for asynchronous, unreliable (or best-effort) communication.

Our test application is implemented so that it can run synchronously or asynchronously. When run synchronously, it employs coherent memory primitives, i.e., `co_write` and `read` plus barrier synchronization at the end of every iteration. The asynchronous version invokes the non-coherent `slow_write` and replaces per-iteration barrier synchronization with a more sophisticated termination detection routine at the end of its execution.

Our implementation of Mermera incorporates an update-based protocol for propagating new values written to the shared memory of all the processors which fits the application at hand since every node needs to read the values written by every other processor. The version of Mermera used to obtain the main experimental results is built on top of version 2.2.5 of the Isis toolkit for group communication [4] which provides several multicast protocols that support different message ordering properties. This particular version of Isis employs point-to-point messages for multicasting.

4.2 Warp Control Implementation

Warp Control is implemented by controlling the message submission mechanism of a buffer interposed between Mermera and Isis that is flushed if, and only if, its size B exceeds $\Lambda_i B_{max}$ where

B_{max} is the maximum buffer size. The quantity Λ_i can be viewed as an "inverse" of λ_i which is adjusted in the opposite direction from what is dictated by the right-hand-side of the control law (2) in Section 3.2. Expressed in terms of its individual components,

$$\Lambda_{ij}^{new} \;:=\; \Lambda_{ij}^{old} - \epsilon\,(1 - warp_{ij}).$$

Due to the homogeneity of the network configuration, the following simplified control is executed

$$\Lambda_{i}^{new} \;:=\; \Lambda_{i}^{old} - \epsilon\,(1 - warp_{i})$$

where $warp_i$ is a running mean of the $warp_{ij}$'s. Thus, the larger the value of Λ_i, the smaller the "message generation rate," and vice versa. The present buffer flushing scheme does not affect the mean arrival rate to the network since application messages are only aggregated into larger network packets. In general, to affect the mean arrival rate of the network,

$$\min\{B, \lambda_i B_{max}\}$$

of the current buffer content needs to be flushed where λ_i is the original message generation rate of node i obeying the control law of Section 3.2. Generally, B is maintained as a FIFO queue, but in the case of asynchronous iterative algorithms, a LIFO discipline is preferable which, in addition to dequeueing last-in-first-out, drops early arrivals first in case of buffer overflow thus promoting recency. We note that there are several schemes for implementing a rate-controlled congestion control, the above methods just being two.

In the context of Ethernet and the CSMA/CD protocol, maximum throughput is achieved when Ethernet frames are large [6]. Our particular implementation of Warp Control represents a customization reflecting the underlying LAN data link technology. We will use "message generation rate" interchangeably with the aggregation mechanism effect described above, keeping in mind that in the Ethernet context they are only qualitatively related. The aggregation mechanism, however, is not totally without cost since it has the effect of increasing the variance of inter-message spacing of application messages in time, reducing their utility and adversely affecting the convergence rate. We will see in the next section that over-aggregation results in the deterioration of application performance when network contention is high.

4.3 Conditional Send

To enhance responsiveness of congestion control and reduce unnecessary protocol stack overhead, a message passing primitive called *conditional send* (`c_send`) is implemented. `c_send` is a `send` that has access to information about network state, and if the network is congested, it degenerates into a null operation. In its simplest realization, `c_send` is equivalent to the macro

$$\texttt{c\_send}\,(\vec{x}) \quad \Longleftrightarrow \quad \textbf{if}\ \neg\texttt{b}\ \textbf{then}\ \texttt{send}\,(\vec{x})$$

where `b` be a Boolean variable that is `true` if the network is congested using a binary criterion, and `false`, otherwise. Supporting a data structure such as `b` does not necessarily entail involvement of the operating system since detection of congestion can be performed by a user-level process acting on information filtered through the protocol stack. In fact, in the current implementation, `b` was a user-space global variable accessible by the application process. Nonetheless, responsiveness may be further enhanced if selective, read-only network state was maintained by the operating system and made visible to user processes without requiring a context-switch. All `slow_write`'s by the application are treated as `c_send`'s, making this mechanism both high level and transparent to the programmer, albeit with the understanding of `slow_write`'s semantics.

The effect of `c_send` is threefold. One, it circumvents the CPU overhead incurred when going through the protocol stack to reach congestion control. The otherwise wasted cost C_s is thus 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. Three, `c_send` can be used by an application independent of the presence of lower level congestion control, complementing existing control features by enhancing responsiveness.

5 Performance Measurements

We measured completion time, throughput, and other variables of network state when solving a system of linear equations in 1000 variables (i.e., $m = 1000$). In the following subsections, we investigate the effect of network congestion on application performance and analyze the dynamics of Warp Control and its efficacy at controlling congestion.

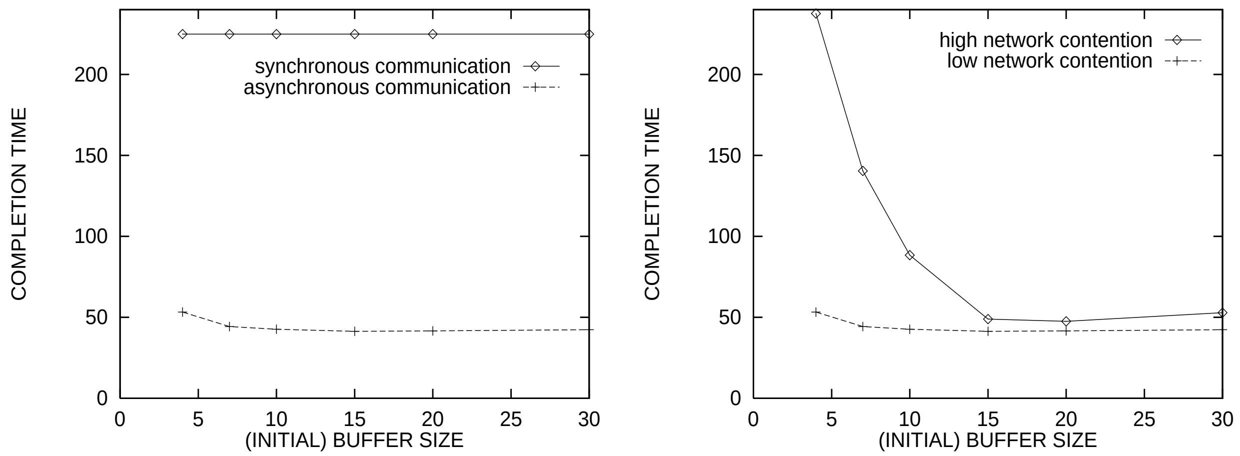

Figure 3: Removing synchrony drastically improves the application's completion time (left). However, when network contention is high, excessive asynchrony can amplify congestion resulting in poor performance (right).

5.1 Application Performance and Synchronization Penalty

Figure 3 (left) shows the completion time of the benchmark application when run both synchronously and asynchronously under low network contention (i.e., in the absence of external cross-traffic). Asynchrony speeds up application completion time by a factor exceeding 4, brought about by the reduction in synchronization penalty. The application-driven increased message generation rate is not enough to saturate the otherwise silent network, carrying with it a negligible communication penalty. When overall network contention is high, however, Figure 3 (right) shows that "excessive asynchrony" can amplify congestion thus resulting in poor application performance. In the Ethernet context, not enough memory updates are aggregated into a single network packet, resulting in a high collision rate (near 20%) accompanied by reduced effective throughput and increased message delays. The aggregation level is represented by the abscissa labeled `(INITIAL) BUFFER SIZE` which is a *static* buffer flushing scheme corresponding to the *dynamic* (i.e., adaptive) method described in Section 4.2.

5.2 Congestion Effect

Whereas Figure 3 (right) depicts the high communication penalty associated with insufficiently controlled network congestion (note that Isis' native congestion control [5] was still active), Fig-

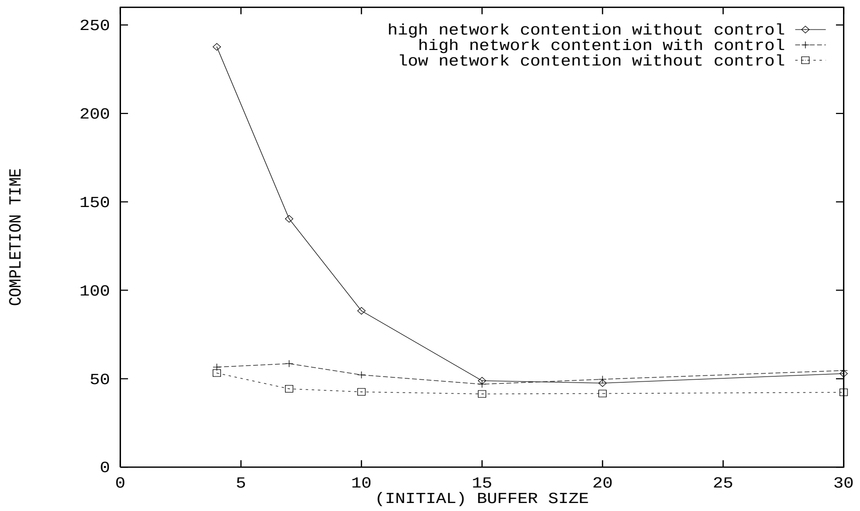

Figure 4: Effect of Warp Control under high network contention—reduces the communication penalty caused by excessive asynchrony.

ure 4 shows the efficacy of Warp Control at regulating message generation rate achieving improved application performance approaching the level of the low network contention case. Figure 5 (left) shows the stable *warp*[4] values plotted as a function of application message count fluctuating around 1 which indicates that messages are evenly spaced out in time and message delay is well-behaved. The high spikes toward the end of the run are due to barrier synchronization for checking a global termination condition. Figure 5 (right) shows the erratically fluctuating *warp* values with Warp Control absent, indicating high variation in message delay which is a sign of severe network congestion. As a consequence, many more messages need to be sent to achieve convergence resulting in a significant increase in application completion time. Figure 6 shows the corresponding runs when Warp Control is active. When network contention is low (left), Warp Control is superfluous and it does not exhibit noticeable effects. Under high network contention (right), after an initial transient period, Warp Control is able to regulate the message generation rate thus stabilizing the network and improving effective throughput. This is also reflected in the smaller number of messages needed to achieve convergence, resulting in a faster application completion time.

We note that the distribution of data to the workstations during the initialization phase can be

[4] Intuitively, the quantity *warp* (see Section 3.2) is a measure of the relative change in network delay. A more detailed discussion can be found in [25].

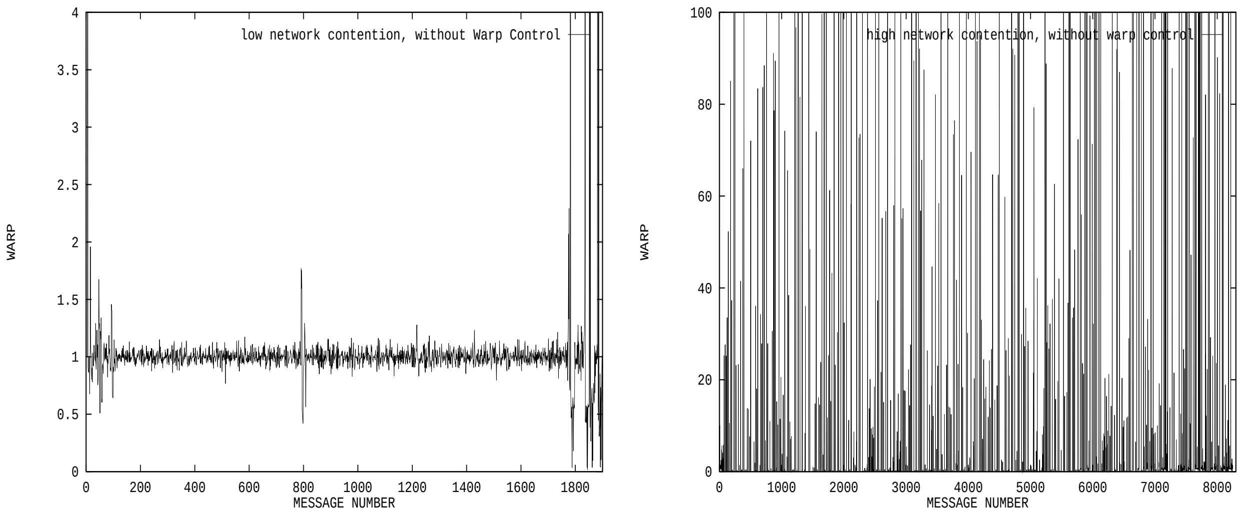

Figure 5: Under low network contention (left), *warp* values indicate that the network is well behaved. Under high network contention (right), absence of adequate congestion control results in high fluctuation of *warp* values indicating that the network is ill-behaved,

significant (20%–40%) depending on the degree of parallelism. We have approached the initial data loading problem in two ways. In the first method, we have used the fact that the 7 workstations use NFS to share the file system exported by one of the attached workstations (i.e., the file server) and let NFS handle the data distribution problem. For practical purposes, this slowed down the overall completion time considerably, and since our goal was to measure the performance of the congestion-controlled system during the dynamic asynchronous evaluation phase, we have not used this method for performance evaluations. Instead, in the second method, on which the present results are based, we have utilized each workstation's local file system to predeposit the initialization data thereby reducing the data loading cost to local reads. The issue of minimizing congestion during the data distribution process has not been studied separately. The collection of results after termination involved only a $O(n)$ communication and was insignicant compared to the computation cost and initialization cost.

5.3 Warp Control Dynamics

Figure 7 gives a more detailed picture of the dynamics of Warp Control showing its ability at stabilizing the network and maintaining message generation rate at a "desirable" level. The abscissa

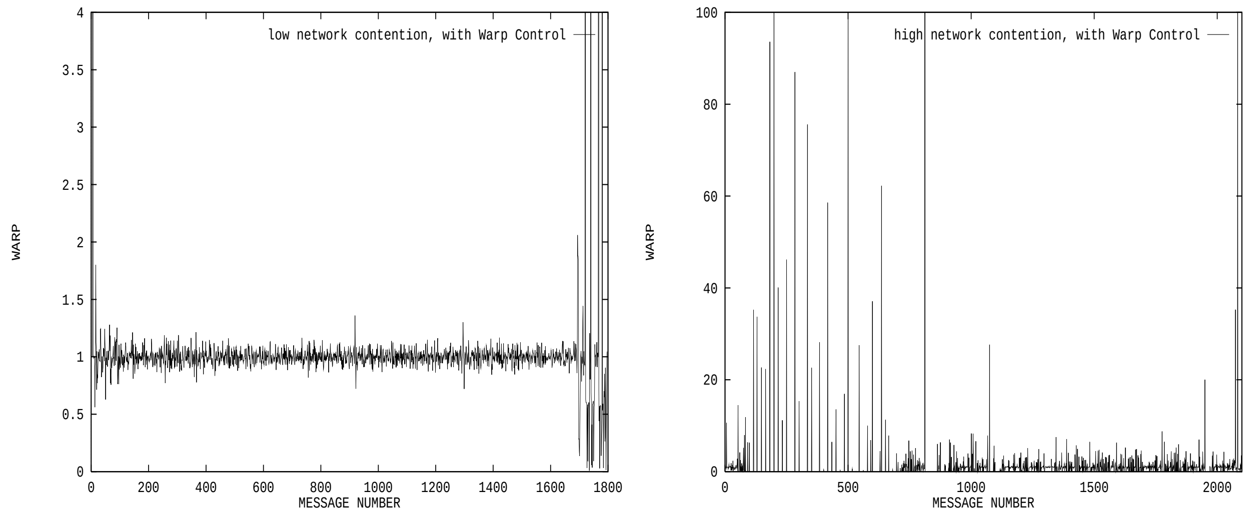

Figure 6: Under low network contention (left), *warp* values indicate that the network is well behaved. Under high network contention (right), Warp Control is able to stabilize the network after an initial transient period restoring the network to a well-behaved state.

has been rescaled from the message count of Figure 6 to actual time measured in seconds starting from a fixed reference point. The first 15 seconds constitute a fixed set-up period unrelated to the computation of the benchmark application during which prepatory steps including the start-up of the two extraneous traffic generators are performed. Figure 7 (top) is a binary impulse plot which shows the arrival time of successive network packets sent from a fixed source to a fixed receiving node[5]. The bottom figure shows the time evolution of the Λ value at the receiving node with initial value 0.1. Λ stabilizes near 0.75 after a transient period that lasts for about 10 seconds. We remark that the larger the value of Λ, the larger the aggregation of application messages into a single network packet resulting in an increased utilization of Ethernet bandwidth. Thus Λ is "inversely" related to message generation rate (notice that message generation rate is used loosely here subject to the caveat mentioned in Section 4.2). The impulse plot shows that during the transient phase message arrivals are interspersed with lengthy silent periods during which messages from the given sender do not arrive. This is also reflected in the sharp spikes of *warp* values (Figure 7 (middle)) which indicate that variation in message delay is high. The high *warp* values impart an upward

[5]The overhead of keeping run traces was found to be minimal and traces were kept for all five application nodes, this particular run showing the interaction of just two.

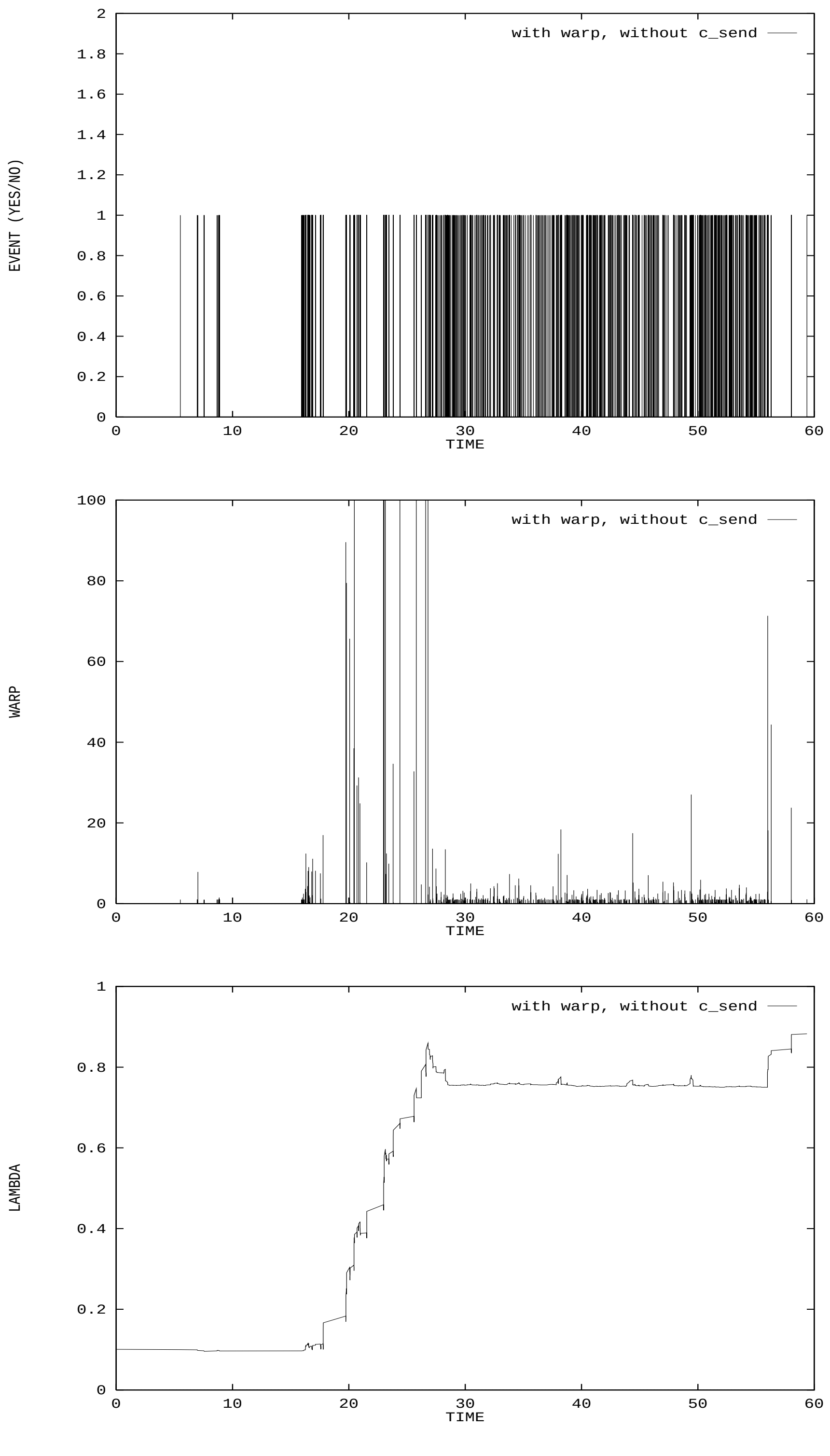

Figure 7: The dynamics of warp control under high congestion. The top two graphs represent successful send events, while the bottom two show $warp$ and the corresponding Λ values as a function of time. Λ is proportional to buffer size (with constant $B_{max} = 40$), and "inversely proportional" to message generation rate.

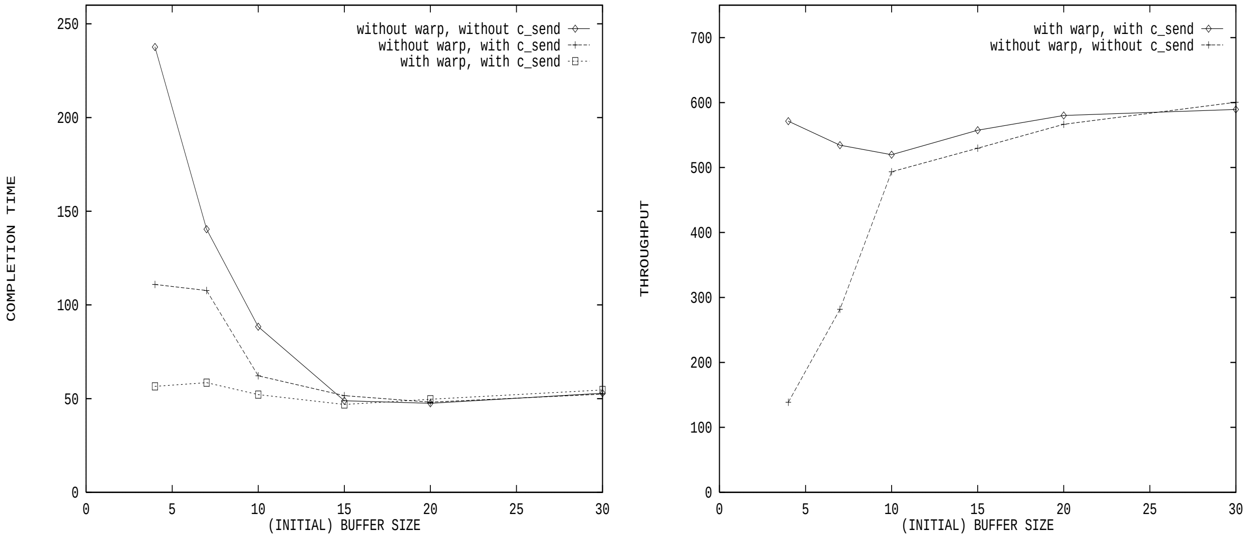

Figure 8: Application completion time in seconds (left), and throughput in application messages per second (right), as a function of the initial buffer size. Dynamic congestion control achieves almost the same completion time as the best *a priori* selection of static buffer size. Application-level congestion control, in the form of `c_send`, can be benefitial in the absence of Warp Control.

drift on Λ, thus reducing the message generation rate which eventually stabilizes the system at $\Lambda \approx 0.75$. This is accompanied by a more dense impulse plot as well as correspondingly smaller *warp* values within the same regime. We remark that the larger the problem size m, the more amortized the cost of the initial transient period becomes.

5.4 Conditional Send and Effective Throughput

Figure 8 (left) shows the effect of `c_send` when Warp Control is not applied. When message generation rate is very high (buffer size = 4), `c_send` enhances application performance by a factor of 2. It is not surprising that conditional send alone is insufficient as a comprehensive congestion control mechanism. The same figure shows that Warp Control is the main factor in achieving effective congestion control. In fact, with Warp Control active, `c_send` exhibits only a marginal performance improvement (not shown here) since the linear equation solver generates "regular" constant rate traffic which obviates the need for increased responsiveness as would be the case for

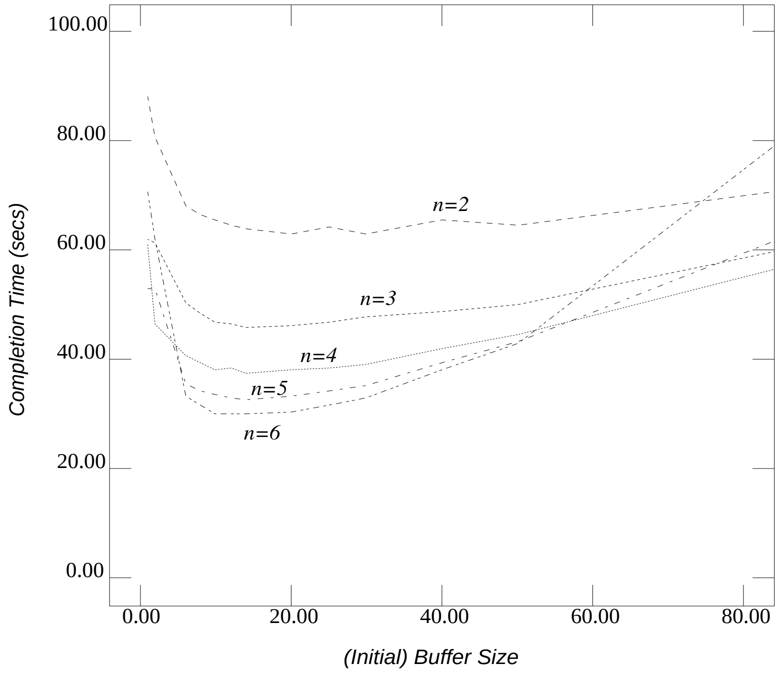

Figure 9: Completion time depends more acutely on buffer size as the number of nodes n (i.e., degree of parallelism) is increased.

variable rate traffic[6] or sudden changes in overall background traffic.

Figure 8 (right) depicts the application throughput when both Warp Control and `c_send` are applied as compared to when they are inactive. For excessive message generation rates (buffer size < 10), the gap in effective throughput widens by a factor approaching five, producing a communication bottleneck which directly impacts application completion time (left). It should be noted that asynchronous iterative algorithms, independent of the existence of communication bottlenecks, continue to maintain their computation rate updating shared memory variables at an undiminished rate. However, when overall network contention is high, the benefit of continuous updating facilitated by asynchrony is cancelled out by the reduction in effective bandwidth induced by excessive influx of messages.

5.5 Degree of Parallelism and Sensitivity of Control

Figure 9 shows the effect of increasing the number of nodes participating in the computation on application performance in an otherwise silent network (i.e., without extraneously generated cross-traffic). As the number of nodes is increased, completion time decreases. The most interesting feature of Figure 9 is not the "downward shift" of the performance graphs but the associated trend of increasing curvature as the number of nodes is increased. The optimal operating point—"bottom of the valley"—becomes more sharply defined, and the cost of over-aggregation in the Ethernet LAN setting begins to weigh in as a nonnegligible factor. In other words, the control problem of finding the optimal operating point becomes more challenging as the degree of parallelism is increased. Although in restricted, dedicated local area network environments the optimal operating point may be established a priori through controlled experiments—a form of open-loop control—in shared dynamic network environments such a brute-force strategy is no longer feasible and it points toward the importance of adaptive, on-line congestion control.

6 Discussion

We have empirically shown that network congestion can be severely deterimental to parallel application performance, and that its effect can be minimized by congestion control. In this particular implementation, congestion control is effected close to the application layer yet it still produces significant performance enhancement when network contention is high. Our study shows that in the absence of adequate transport/network layer congestion control, high-level congestion control can still be effective, facilitating portable, application-driven traffic control. With standardized, inter-application traffic management network interfaces still several years away, and parallel distributed computing platforms steadily gaining in popularity (e.g., PVM), a high-level control strategy may serve as an interim solution for per-application traffic control to avoid self-defeating excessive traffic generation.

When customized to the case of asynchronous iterative algorithms, our message passing primitives take the form of non-blocking, unreliable, network state-triggered protocols (`c_send`) where conditional send provides a direct link between network state and the application layer, improv-

[6]This may be affected by the characteristics of the application itself, but in the context of parallel iterative algorithms, the more likely culprit is the interference of other applications that share common network resources.

ing responsiveness while reducing unnecessary protocol stack overhead by aborting unproductive communication events at the earliest possible moment. Operating system support for `c_send` and "`c_send`-like" features, when implemented such that the need for context-switching is obviated, may further improve system performance. The revised version of Mermera built on top of UDP and running Warp Control in its interface represents an effort in the direction of achieving improved efficiency and user-level transparency at the cost of reduced portability.

In the context of B-ISDN ATM networks, the potential for tolerating unreliable message transmission through the use of asynchronous iterative methods (whenever possible) looms as a significant feature to be exploited since the gap between *reliable throughput* and *unreliable throughput* is expected to widen due to the high bandwidth-delay product. We believe that large-scale switched network environments and ATM networks will further increase the curvature of the performance graphs in Figure 9 since the abscissa will be replaced by a more faithful measure of message generation rate[7], buffer overflow and queueing delay being the main congestion effects.

Lastly, synchronous computations, although less prone to producing self-induced congestion, are nonetheless highly sensitive to the overall level of network contention, and our observations—excluding non-blocking unreliable message transmission—are equally valid in the strictly synchronous context. Synchronous computations are "self-regulating" with respect to traffic generation in the sense that a congested network causes applications to reduce their message generation rate by virtue of its effect on synchronization delay penalty which stretches each synchronization phase in time. Nonetheless, techniques for hiding the latency of synchronous communication such as multi-threading may enable applications to generate more messages taxing a network's resources when the degree of parallelism is high. Congestion should, therefore, be considered a basic communication problem that can have severe detrimental effects on many forms of parallel distributed applications sharing a resource-bounded communication network.

Acknowledgements

We would like to thank several parties for their contributions and help. Arif Bhatti implemented the UDP version of Mermera, and with Sulaiman Mirdad, participated in its design. Figure 9 is

[7] In the simplest configurations (e.g., low fan-in/fan-out ATM switch), SONET frames may exert an aggregation effect.

derived from Himanshu Sinha's Ph.D. thesis [29]. Hewlett-Packard's Metrix Network Systems, Inc. provided us with free use of their NetMetrix 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.

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

[3] D. Bertsekas and J. 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] K.P. Birman and T. Clark. Performance of the Isis distributed computing system. Technical Report TR-94-1432, Cornell Univ., Computer Science Dept., June 1994.

[6] D.R. Boggs, J.C. Mogul, and C.A. Kent. Measured capacity of an Ethernet: myths and reality. In *Proc. ACM SIGCOMM '88*, pages 222–234, 1988.

[7] C. Cap and V. Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.

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

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

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

[11] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine.* The MIT Press, 1994.

[12] M. Gerla and L. Kleinrock. Flow control: a comparative survey. *IEEE Trans. Commun.*, COM-28:553–574, 1980.

[13] A. Heddaya and K. Park. Mapping parallel iterative algorithms onto workstation networks. In *Proc. 3rd IEEE International Symp. on High-Performance Distributed Computing*, pages 211–218, 1994.

[14] A. Heddaya and K. Park. Parallel computing on high-speed wide-area networks: a pricing policy for its communication needs. In *Proc. 3rd IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, pages 188–191, 1995.

[15] 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.

[16] A. Heddaya and H. 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.

[17] 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.

[18] Van Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM '88*, pages 314–329, 1988.

[19] H. Kim and D. Farber. The failure of conservative congestion control in large bandwidth-delay product networks. In *Proc. INET '95*, 1995.

[20] H. T. Kung, Trevor Blackwell, and Alan Chapman. Credit-based flow control for ATM networks: credit update protocol, adaptive credit allocation, and statistical multiplexing. In *Proc. SIGCOMM '94*, pages 101–114, 1994.

[21] 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.

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

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

[24] 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.

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

[26] Kihong Park. Self-organized multi-class QoS provision for ABR traffic in ATM networks. In *Proc. 15th IEEE International Phoenix Conference on Computers and Communications*, pages 446–453, 1996.

[27] 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.

[28] Yu Shashkin. *Fixed Points*, volume 2 of *Mathematical World*. American Mathematical Society, 1991.

[29] Himanshu Sinha. MERMERA*: Non-coherent Distributed Shared Memory for Parallel Computing*. PhD thesis, Boston University, Computer Science Department, 111 Cummington Street, Boston, MA 02215, May 1993.

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

[31] M. Wernik, O. Aboul-Magd, and H. Gilbert. Traffic management for B-ISDN services. *IEEE Network*, pages 10–19, September 1992.