Title

Superboundary exchange : a technique for reducing communication in distributed implementations of iterative computations

Authors

Kuang, Hairong
Bic, Lubomir
Dillencourt, Michael B.

Publication Date

2000

Peer reviewed

ICS

TECHNICAL REPORT

SuperBoundary Exchange: A Technique for Reducing Communication in Distributed Implementations of Iterative Computations

Hairong Kuang, Lubomir Bic, Michael B. Dillencourt
Information and Computer Science
University of California, Irvine, CA 92697-3425
Email: {hkuang,bic,dillenco}@ics.uci.edu

TR 00-24

Information and Computer Science
University of California, Irvine

SuperBoundary Exchange: A Technique for Reducing Communication in Distributed Implementations of Iterative Computations

Hairong Kuang, Lubomir Bic, Michael B. Dillencourt
Information and Computer Science
University of California, Irvine, CA 92697-3425
Email: {hkuang,bic,dillenco}@ics.uci.edu

Abstract

We introduce a technique for lowering the communication cost in a certain type of distributed application, in which processors perform computation in each time step and must obtain boundary data from their neighbors before they can perform the next time step. A typical example of such an application is solving differential equations using the finite difference method. Our method, which we call *SuperBoundary Exchange*, consists of sending a larger boundary less often. This results in less frequent data exchange, trading off against larger messages and some redundant computation. We present experimental data showing that our method consistently results in significant speedup in communication-intensive applications, under varying assumption about the balance of computation load among the processors.

1 Introduction

As workstations become more powerful and less expensive, solving problems through distributed computation on a network of workstations, rather than on more specialized parallel processors, is becoming a much more popular and cost-effective approach. One obstacle to this approach is that communication across a network of workstation is more time-consuming than, for example, communication between processing units in a shared-memory parallel machine. For this reason, techniques

for lowering communication costs play an important role in the development of distributed algorithms.

In this paper we introduce a technique for lowering the communication cost in a certain type of distributed application, typified by solving differential equations using the finite difference method [Van94]. In the finite difference method, we have a discrete d-dimensional grid of element locations, and we want to compute a value $u_{\mathrm{X},t}$ at each grid location x and for each time step t. The value of $u_{\mathrm{X},t}$ is a function of the values of $u_{\mathrm{r},t-1}$, where r ranges over x and all neighbors of x in the element grid.

In a distributed implementation of the finite difference method, the element grid is partitioned, and each processor is assigned one of the partitions. In each time step t, each processor computes the values of $u_{\mathrm{X},t}$ for all grid elements x assigned to it. If a grid element x is on the boundary of the partition (i.e., it is assigned to a processor P_1, but it has a neighboring grid element assigned to a different processor P_2), then the computation of $u_{\mathrm{X},t}$ cannot be performed until P_1 has obtained from P_2 the values of $u_{.,t-1}$ for all these neighboring elements. Thus at each time step there is data exchange (communication) with neighboring processors as well as computation. The data that must be obtained by a particular processor from neighboring processor to allow it to perform its computation is sometimes called the *ghost boundary* of the processor [Van94].

The technique introduced here, which we call *SuperBoundary Exchange*, reduces communication by sending a larger ghost boundary less often. In essence, the increased overhead of sending larger messages and performing some redundant computation is traded off against the reduced overhead of less frequent messages, and the net result is an overall increase in performance. The advantages of both these tradeoffs are well-known in other contexts. For example, the advantage of combining multiple small data transfers into a single larger one underlies block transfers between memory hierarchies, and combining small packets into larger ones is a well known technique for efficient message passing in networks [Nag84]. The technique of replicating data to reduce message traffic is a classical technique in distributed data bases.

The organization of this paper is as follows. We describe the SuperBoundary Exchange technique in Section 2, and contrast it with another technique, send-ahead, which was previously introduced in [LF95]. In Section 3, we describe the results of several experiments demonstrating that performance advantages of SuperBoundary Exchange. Some conclusions and final remarks are presented in Sec-

tion 4.

2 The SuperBoundary Exchange Technique

We introduce the SuperBoundary Exchange Technique with the following simple example, illustrated in Figure 1. Suppose that we are solving a 1-dimensional finite difference equation

$$u_{x,t+1} = f(u_{x-1,t}, u_{x,t}, u_{x+1,t}),$$

on a network of processors. Suppose that the values of u associated with x in the range $1, \ldots, 100$ are computed on processor P_1, and that the values of u associated with x in the range $101, \ldots, 200$ are computed on processor P_2. In order to compute the value of $u_{100,t+1}$, P_1 must know the value of $u_{101,t}$; hence this data (the "ghost boundary") must be sent to P_1 by P_2 once it is known (and, symmetrically, P_1 must send $u_{100,t}$ to P_2 before P_2 can compute $u_{101,t+1}$). If each processor only stores the data for the x values directly associated with it, such an exchange of boundary data must occur at each time step in the computation. This is illustrated in Figure 1(a), which shows the computation from the end of time step 3 to the end of time step 6. The first row shows the computation at the end of time step 3. The next three double rows illustrate time steps 4, 5, and 6; each of these time steps is illustrated after the ghost boundary has been exchanged at the beginning of the time step (the dotted box indicates the ghost boundary), and then after the computation for the time step has been completed.

SuperBoundary Exchange is a technique for reducing the frequency of this boundary exchange. Suppose, at the beginning of time step $t+1$, P_1 sends the values $u_{98,t}$, $u_{99,t}$ and $u_{100,t}$ to P_2. Then at time $t+1$, P_2 has all data necessary to compute the values $u_{99,t+1}$, $u_{100,t+1}$, and $u_{101,t+1}$. At time $t+2$, P_2 has all data necessary to compute the values $u_{100,t+2}$, and $u_{101,t+2}$. It can then use this data to compute the value $u_{101,t+3}$. Hence no further boundary data must be exchanged until time $t+3$, and in general, data must only be exchanged every three iterations. This is illustrated for $t = 3$ in Figure 1(b).

More generally, the ghost boundary is defined as all the elements that are needed by neighboring processors to compute all their values at the next time step. In a regular boundary exchange, first each pair of neighboring processors exchanges ghost boundaries, and then each of the processors computes the new values for all of its elements. This process is repeated for every time step.

A *SuperBoundary* of width k is defined inductively: a ghost boundary has

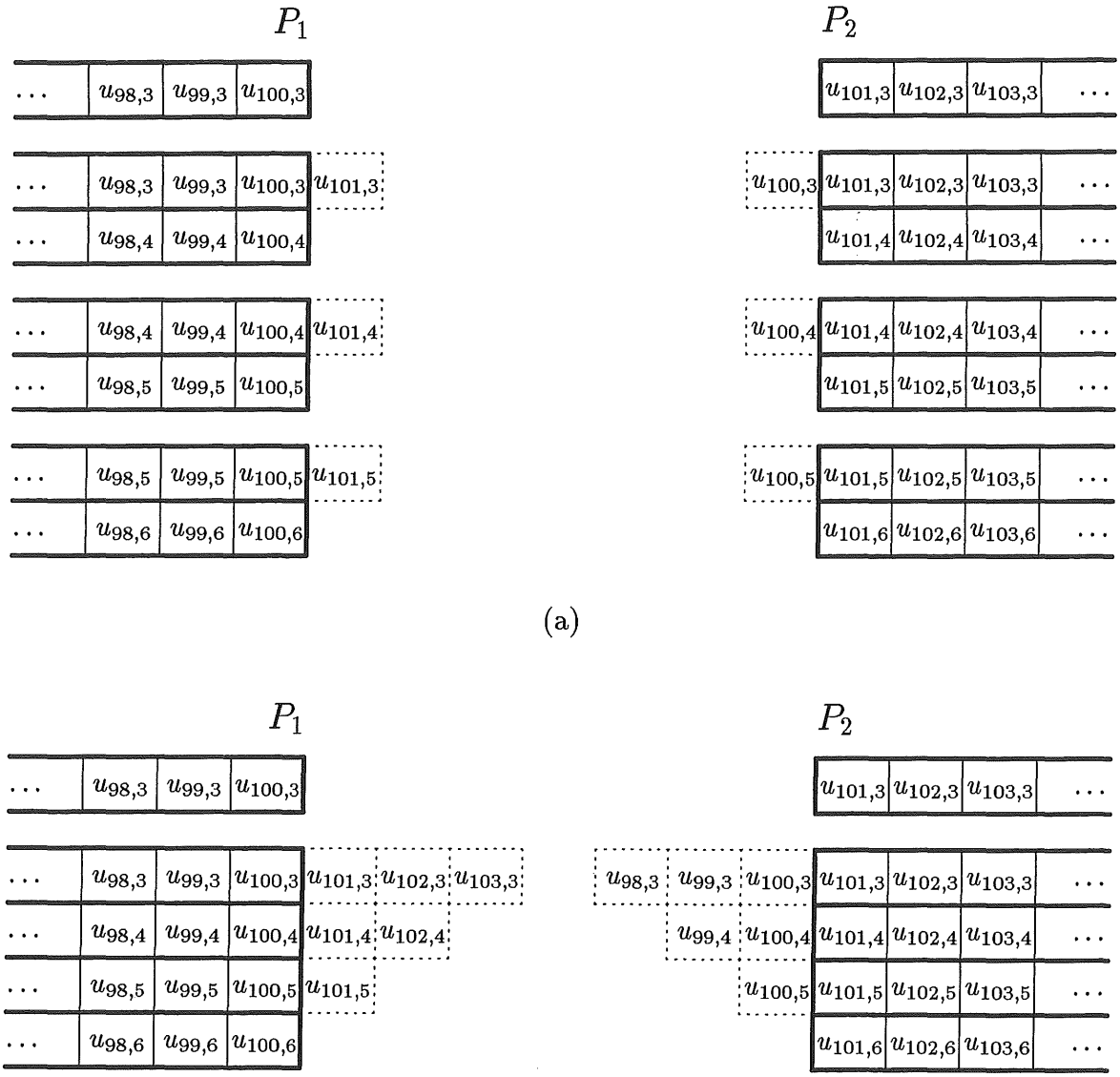

Figure 1: A simple example illustrating SuperBoundary Exchange. Both figures represent the interval from the end of time step 3 to the end of time step 6. (a) In the absence of SupeBoundary Exchange, the ghost boundary is exchanged at the start of each time step. (b) SuperBoundary Exchange: 2 additional boundary data items are exchanged at the start of time step 4, but no boundary information is exchanged at time 5 or at time 6.

situation after the data exchange at time t (t_e), after the computation at time t (t_c), and after the computation at time $t+1$ ($(t+1)_c$). At t_e, processor 0 has two "layers" of boundary data to work with. At t_c, the data for time t, including the data on the regular boundary has been computed. At $(t+1)_c$, the computation for time $t+1$ is complete, but there is no longer valid boundary data, so data must be exchanged again.

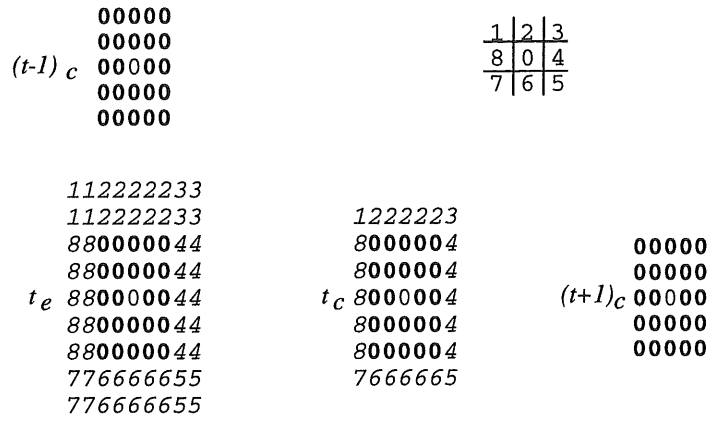

Figure 3: Schematic representation of 2-dimensional SuperBoundary Exchange, with a boundary width of 2.

Notice that in the 2-dimensional case just described, if the subgrid assigned to each processor is an $n \times n$ grid, and the boundary width is k, then the number of grid element values that must be exchanged between neighbors once every k time steps is $O(k \cdot n)$. Hence the amount of additional memory required goes up linearly with the boundary width, but only with the square root of the amount of memory required to store the subgrid associated with the processor. Hence, SuperBoundary Exchange requires only a small amount of additional memory. The generalization of SuperBoundary Exchange to d dimensions is conceptually straightforward, and we omit the details here.

The SuperBoundary Exchange technique improves performance in two ways: by reducing communication overhead and by smoothing out random bursts. Communication overhead is reduced because the frequency of communication (i.e., the number of messages) is reduced. Depending on the message size, the number of packets may be reduced as well, since several small messages are being combined into one larger one.

The effect of smoothing out random bursts is illustrated in Figure 4(a) and (b). Suppose that Processor 1 takes less time than its neighbor, Processor 2, to perform

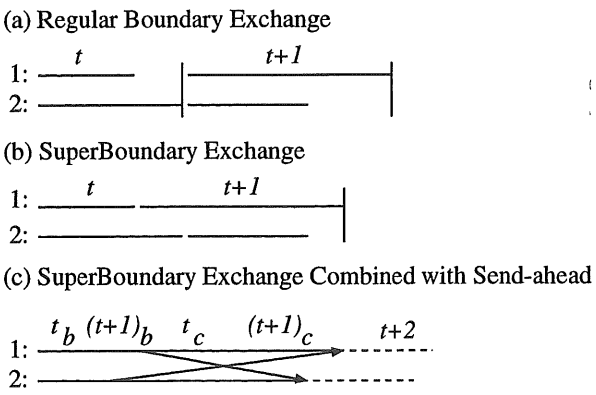

Figure 4: Smoothing out of random bursts using SuperBoundary Exchange and send-ahead.

the computation at time step t, but Processor 2 takes less time than Processor 1 to perform the computation at time step $t+1$. Under regular boundary exchange, the two processors must synchronize at every time step, and hence the time required for each time step is the time required by the slower processor. This is shown in Figure 4(a), where the horizontal lines represent the elapsed times for each time step and the vertical lines represent the synchronization points. Under SuperBoundary exchange with boundary width 2, the two time steps t and $t+1$ can be executed consecutively without synchronization as shown in Figure 4(b). As a result, the bursts are smoothed out and step $t+2$ can be started at an earlier time.

In [HCYA94] and [LF95], a different approach to reducing communication overhead, called "boundary precomputation" in [LF95], was proposed. We now briefly discuss this method and its limitations, and then show how it can be combined with SuperBoundary Exchange.

The boundary precomputation method proposed in [LF95] actually consists of applying two separate observations. The first observation is that it is not necessary to compute all values at time t before sending the ghost boundary; instead, a processor can compute its boundary values at time t, then send them to its neighbors while it is computing its interior values at time t. We refer to this part of the boundary precomputation as "send-ahead." The second observation is that if a process finishes its computation for time t while it is waiting for the time-t ghost boundary to arrive from its neighbors; it can continue processing by computing the values at time $t+1$ at cells of distance 1 from its internal boundary, values

at time $t + 2$ at cells of distance 2 from its internal boundary, etc. We refer to this as "internal precomputation." It is pointed out on [LF95, Page 197] that internal precomputation does not help overall performance. One reason for this is that if communication is the dominant factor, each processor will reach a point of saturation, where it has done as much internal precomputation as it possible can; after this, it will only be able to advance the time at each grid element by one in each cycle, just as in the absence of internal precomputation. Because of this, we discuss send-ahead, but not internal precomputation, in the remainder of this paper.

SuperBoundary Exchange and send-ahead can be combined as follows. Suppose that the starting time is t and the boundary width is k. Immediately after a data exchange, each processor computes the value of the data along the superboundary that it needs to send at time $t + k$ and sends it to the appropriate neighbors; it then computes the rest of its data (the inner portion). Figure 5 schematically illustrates the combination of these two techniques, for a 1-dimensional problem with a boundary width of 3. The line labeled t_e shows the status at time t after

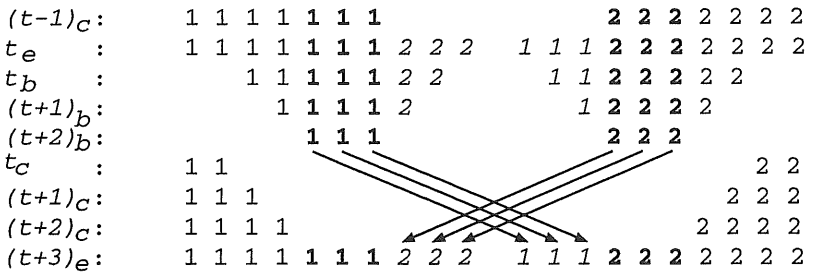

Figure 5: Schematic representation of the combination of SuperBoundary Exchange and send-ahead, for a 1-dimensional problem with a boundary width of 3. with a boundary width of 2.

data has been received: Processor 1 has received three boundary items from its neighbor, Processor 2. In the next three lines, it computes the values at time $t+3$ of the boundary values that it needs to send to processor 2, and sends them to processor 2 when this computation is complete (after the line labeled $(t + 2)_b$. It then computes its inner values for time $t+3$; this computation overlaps with waiting for the boundary values to arrive from Processor 2. If the boundary width is not too big, then the bulk of the computation (i.e., the computation of the inner values) is done while the processor is waiting for the boundary values from the neighboring processors, so the computation overlaps with the communication latency of the

boundary messages from neighboring processors. This can represent a considerable advantage, because it means that the combination of SuperBoundary Exchange and send-ahead can smooth out the random fluctuations in load as illustrated in Figure 4(c).

3 Experimental results

We present experimental results in this section. All the experiments were performed using nine Sparcstation 5's connected by a 100Mbit/second switch. The nine workstations were arranged as a 3×3 toroidal grid. This arrangement eliminated boundary effects, so the configuration and experimental results can be viewed as a portion of a much larger collection of workstations. All implementations used the MESSENGERS system [BFD96, FBD99], and speedup was computed relative to a MESSENGERS implementation using the same 3×3 toroidal grid as a logical network but running on a single workstation. To decrease the variability of the presented data, all the experiments describe here were run three times, and in each case the number presented here is the average over the three runs.

A distributed computation may be dominated either by the communication cost or by the computation cost. Because SuperBoundary Exchange is intended to reduce the effect of communication overhead, it is primarily useful in situations where the communication costs dominate. For this reason, the parameters for our experiments were chosen to make the communication-to-computation ratio relatively high. We experimented with three different workload-balance scenarios: (1) *even workload balance*, in which the computation time required by each processor at each step is about the same; (2) *systematic workload imbalance*, in which there are certain processors that require more time in each time step to perform their computation; and (3) *fluctuating workload imbalance*, in which case there is imbalance in the computation time, but the specific processors requiring more time vary from time step to time step.

The application we used is a 2-dimensional finite difference problem, with an $n \times n$ element grid with n varying as discussed below. The 2D $n \times n$ elements were evenly distributed on 9 workstations, logically connected as a 3×3 grid, with each workstation performing the computation for a subgrid $(n/3) \times (n/3)$ elements. In each workstation, the communication overhead is approximately $4a(n/3)$, where a is a constant, while the computation cost is approximately $b(n/3)^2$, where b is a constant. The communication to computation ratio is c/n, where $c = 12a/b$. There-

fore, when the problem size becomes smaller, the communication-to-computation ratio becomes bigger.

Our first experiment assesses the effect of the boundary width on the performance of a computationally balanced distributed application. Figure 6 shows the result of this experiment. The horizontal axis represents the boundary width, and

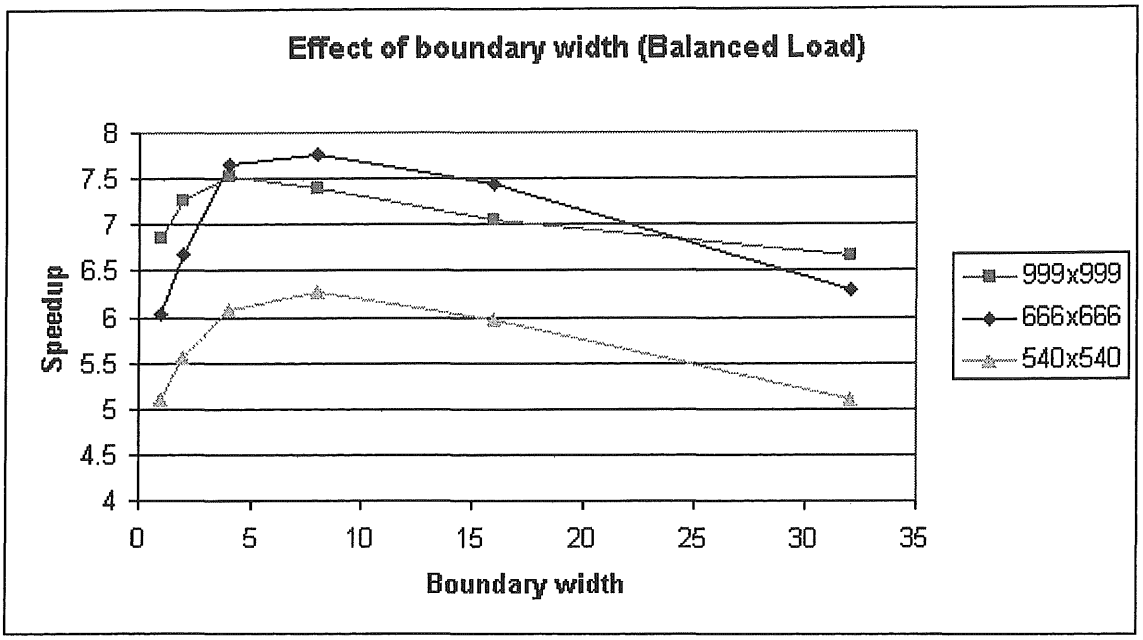

Figure 6: The effect of the boundary width on the performance of a computationally balanced application.

the vertical axis represents the speedup. A boundary width of 1 means that we are using regular boundary exchange; larger values mean that we are using SuperBoundary Exchange. We experimented with three problem sizes: 540×540, 666×666, and 999×999. In all three cases, SuperBoundary Exchange increases the speedup significantly for low values of the boundary width, and as the boundary width is increased, the speedup reaches a maximum and then falls off. This is because for low values of the boundary width, the decrease in communication overhead far outweighs the additional cost due to redundant computation of boundary values. For high values of the boundary width, most of the communication savings has been realized, the program becomes computation-intensive, and the increased computation starts to cause a performance decrease. But this latter effect is gradual: even when the boundary width reaches 32, SuperBoundary Exchange still beats the regular boundary exchange for the two smaller problem sizes.

The remaining experiments compare the SuperBoundary Exchange technique, the send-ahead technique, and the combination of the two for the three workload-balance scenarios discussed at the beginning of this section. Figure 7 shows the results for the balanced-workload case, for a problem of size 666×666. The gray line

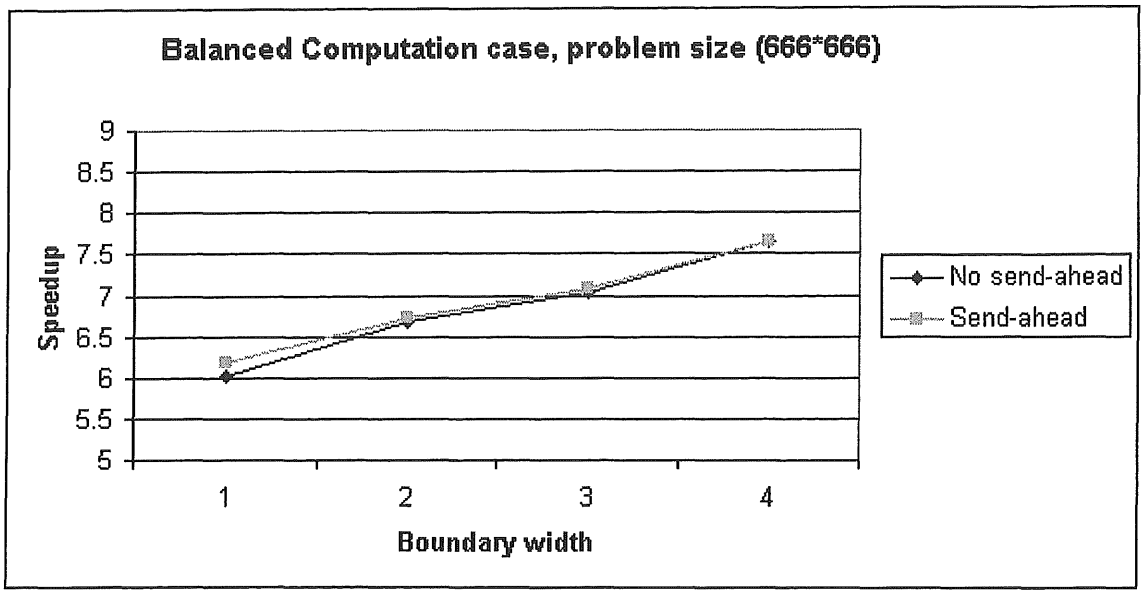

Figure 7: The effect of SuperBoundary Exchange, with and without send-ahead, on a balanced computation.

shows the performance when the send-ahead technique is used. From the figure, we can see that SuperBoundary Exchange improves the performance. Send-ahead helps slightly in the case of regular boundary exchange (i.e., for boundary width 1), but it does not help otherwise. Also, SuperBoundary Exchange speeds up the computation using regular boundary exchange much more than send-ahead does. There are several reasons explaining why send-ahead contributes so little in this case. First, the transmission time is negligible in our experiment network because workstations are connected by 100M/bit switch. Second, because the workload is evenly distributed, there is no idle waiting time to be eliminated by the send-ahead technique. In addition, send-ahead changes the computation order, and this may affect the hit rate on the memory cache.

Figure 8 shows the results for the case of systematic workload imbalance. In this experiment, we chose one processor and artificially increased its workload in each time step by 30%. As with the balanced case, SuperBoundary Exchange improves the performance, but the send-ahead technique doesn't help much. The

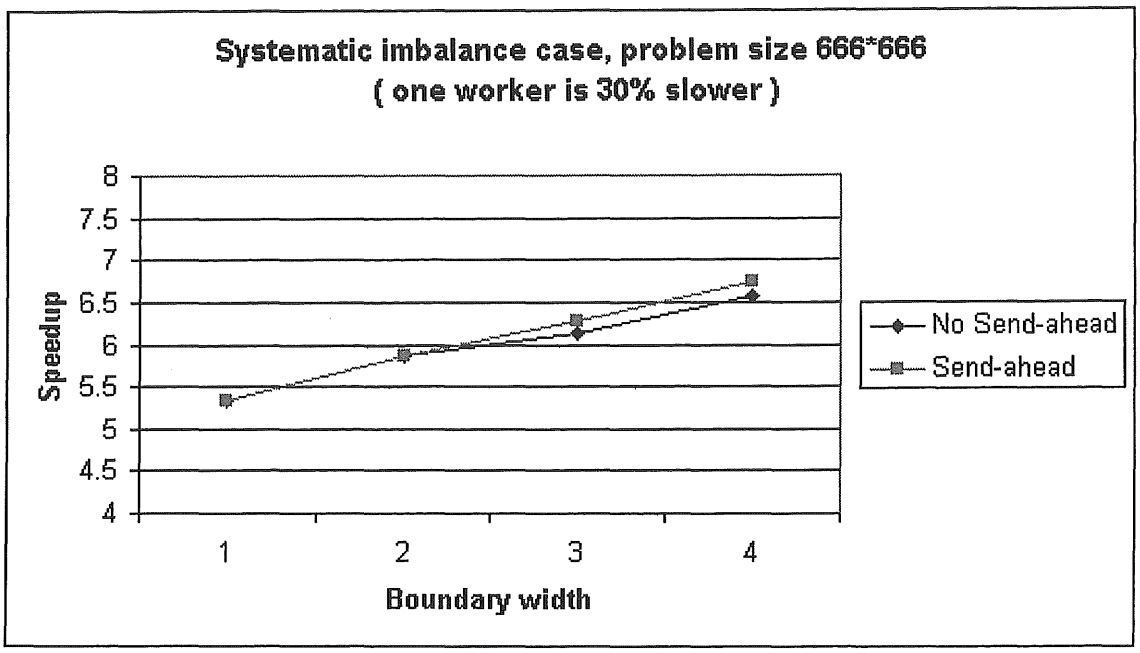

Figure 8: The effect of SuperBoundary Exchange, with and without send-ahead, on a systematically imbalanced computation.

explanation is the same as that of the balanced case: because the workload is systematically unbalanced, the idle waiting time can not be squeezed out.

Our final experiments address the case of fluctuating load imbalance. We ran the balanced experiment, but artificially increased the workload on each processor in each time step by a random amount. The random additional load was Gaussian, with a mean of 30% of the base load and a standard deviation of 10% of the base load. Figures 10 and 9 show the results for problem sizes of 999×999 and 666×666, respectively. If we compare the individual effects of send-ahead with the and SuperBoundary Exchange, we see that on the smaller problem SuperBoundary Exchange is better. On the larger problem, send-ahead is better than increasing the boundary width to 2, but for larger boundary widths SuperBoundary Exchange is better. The best results are obtained by combining the two methods: on the smaller problem, send-ahead together with SuperBoundary Exchange with a boundary width of 4 yields a 34% improvement in speedup. Send-ahead helps in this case because it helps smooth out the bursts due to the random fluctuations in computation time.

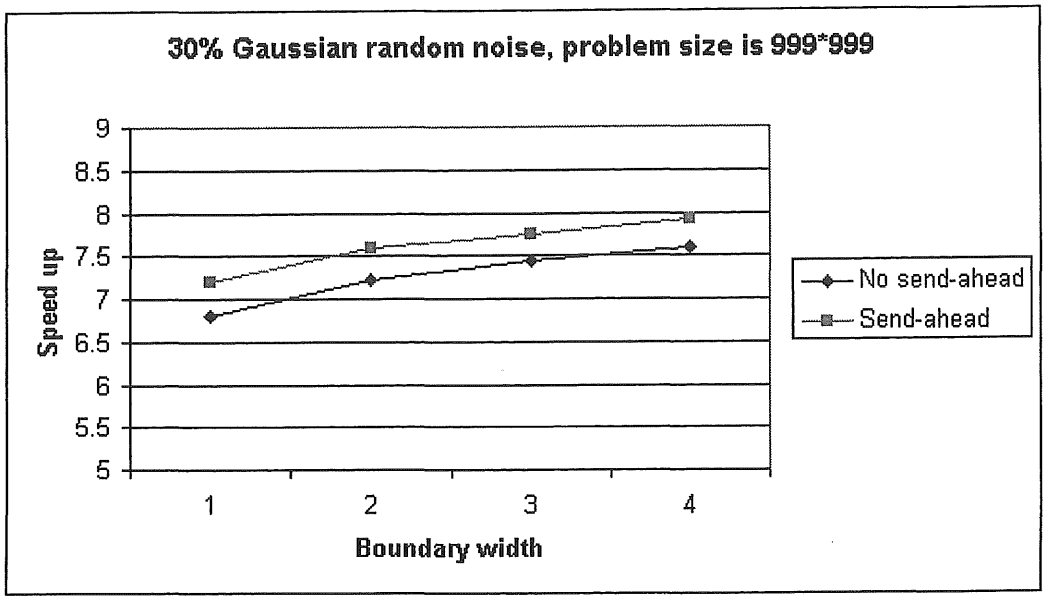

Figure 9: The effect of SuperBoundary Exchange, with and without send-ahead, on a computation of a 999×999 problem with fluctuating imbalance.

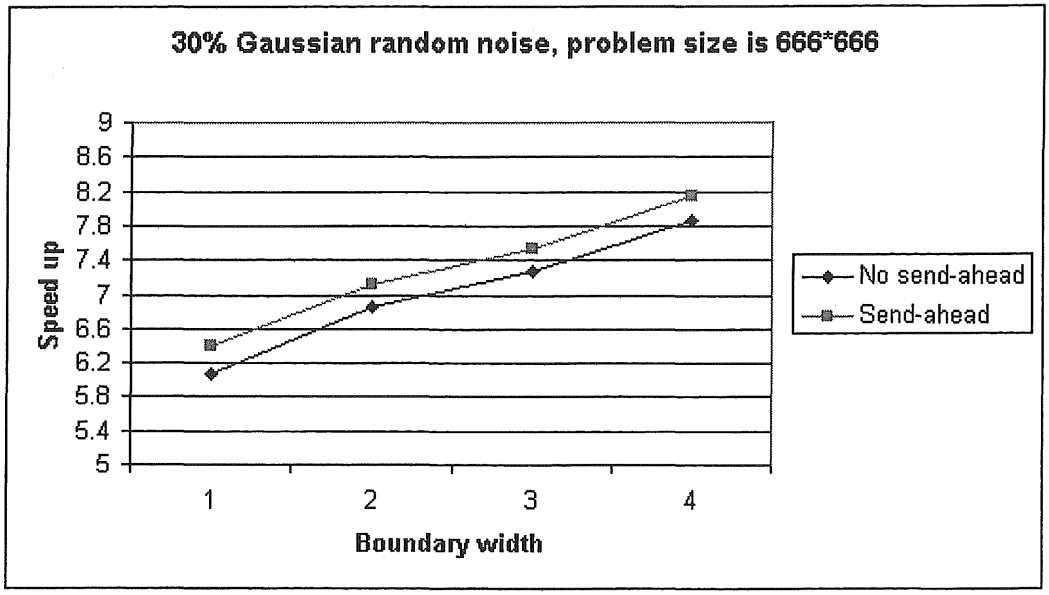

Figure 10: The effect of SuperBoundary Exchange, with and without send-ahead, on a computation of a 666×666 problem with fluctuating imbalance.

4 Conclusions and Final Remarks

In this paper we have introduced the SuperBoundary Exchange technique as a means of reducing communication overhead in distributed applications. The underlying tradeoff is fewer messages against potentially longer messages and some additional, redundant, computation. We have presented some experimental data comparing SuperBoundary Exchange with the send-ahead technique. The results of our experiments are summarized in Table 1. If the running time of the applica-

	Balanced Computation	Imbalance Computation	
		Systematic Imbalance	Fluctuating Imbalance
Computation Dominant	Better Hardware	Load Balancing	Load Balancing
Communication Dominant	SBX	SBX	SBX + Send-ahead

Table 1: Summary of the results of our experiments. SuperBoundary Exchange provided significant speedup improvement for all communcation-intensive scenarios. Send-ahead provided additional improvement in the case of fluctuating load-imbalance, but provided no incremental benefit in the other two cases.

tion is dominated by the computation time, then performance can be addressed by improving the hardware or by addressing load imbalance among processors. The technique introduced in this paper, SuperBoundary Exchange, reduces communication overhead, and hence is of use when the communication cost dominates. Our experiments indicate that this technique is useful whether or not there is load imbalance among the processors, and that it provides a significantly larger performance improvement than the send-ahead technique. In the case of fluctuating load imbalance, further improvement can be obtained by combining SuperBoundary Exchange with send-ahead. In the other two cases (balance, or systematic imbalance), send-ahead provides almost no incremental performance improvement beyond that provided by SuperBounday Exchange.

Although the SuperBoundary Exchange technique is of most use when the communication cost dominates, it can ultimately be applied even in a computation-dominated application. If computation dominates, appropriate ways to address

performance concerns would include fine-tuning the computation at each grid element, adding more processors, and upgrading the hardware. As these enhancements are made, a point of diminishing returns will be reached, because the computation cost will decrease to the point where it is dominated by the communication cost. When this point is reached, and the application becomes communication-intensive, SuperBoundary Exchange can then be used to further enhance the performance of the application.

While the only application mentioned is finite difference equations, the SuperBoundary Exchange technique can be applied to a wider class of problems. The technique can be applied to any application in which there is an underlying space, a well-defined notion of neighbor, and an iterative computation where the computation at a particular location in one time step depends on the state of that location and its neighbors in the preceding time step. For example SuperBoundary Exchange could be applied in individual-based spatial simulations in which the behavior of entities is based solely on their interactions with nearby entities (e.g., models of the flocking of birds or the schooling of fish [HW92, Rey87]). Here, the boundary data to be exchanged is the state of entities in nearby nodes, and SuperBoundary Exchange would require redundant computation of these behaviors after an exchange of data.

References

[BFD96] L. F. Bic, M. Fukuda, and M. B. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8), Aug. 1996.

[FBD99] M. Fukuda, L. F. Bic, and M. B. Dillencourt. Messages versus MESSENGERS in distributed programming. *Journal of Parallel and Distributed Computing*, 57:188–211, 1999.

[HCYA94] C.-C. Hui, G. K.-K. Chan, M. M.-S. Yuen, and I. Ahmad. Solving partial differential equations on a network of workstations. In *Proceedings of the Third IEEE Interational Symposium on High Performance Distributed Computing*, pages 194–201, August 1994.

[HW92] A. Huth and C. Wissel. The simulation of the movement of fish schools. *Journal of Theoretical Biology*, 156:365–385, 1992.

[LF95] J. Z. Lou and R. D. Ferraro. A parallel incompressible flow solver package with a parallel mutigrid elliptic kernel. In *Super Computing '95*, December 1995. `http://olympic.jpl.nasa.gov/Reports/flow_solver/pap.html`.

[Nag84] J. Nagle. Congestion control in TCP/IP internetworks. *Computer Commun. Review*, 14:11–17, October 1984.

[Rey87] C.W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, July 1987.

[Van94] E. F. Van de Velde. *Concurrent Scientific Computing.* Springer-Verlag, 1994.