Communication Support for Adaptive Computation*

Ali Pınar[†] *and Bruce Hendrickson*[‡]

1 Introduction

In this work we address two problems associated with redistributing data amongst processors. The first problem is that of determining the inter–processor communication pattern necessary to perform a calculation like matrix–vector multiplication. Consider the situation when a calculation is first described or when it is repartitioned after dynamic load balancing. Processors do not know what communication operations to perform to enable the matrix–vector multiplication to proceed. Assuming the matrix is partitioned by rows, looking at its own domain allows each processor can determine what it wants to receive, but it does not know which processor owns these desired data. We propose a *distributed directory* algorithm to efficiently determine the communication pattern (i.e., what a processor needs to receive from and send to every other processor). Our experiments show that the proposed algorithm performs efficiently on large numbers of processors.

The second problem is that of actually migrating data in the case of limited memory. Although a number of algorithms and software tools have been developed to repartition the work amongst processors, the mechanics of actually moving large amounts of data has received much less attention. If sufficient memory is available, each processor can allocate space for its incoming data, post asynchronous receives, and then send its outgoing data. Memory for outgoing data can be deallocated only

*In Proc. 10th SIAM Conf. Parallel Processing for Sci. Comput., 2001. This work was funded by the Applied Mathematical Sciences program, U.S. Department of Energy, Office of Energy Research and performed at Sandia, a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the U.S. DOE under contract number DE-AC-94AL85000.

†Department of Computer Science, University of Illinois, Urbana, IL 61801–2589, e-mail: `alipinar@cse.uiuc.edu`

‡Parallel Computing Sciences Department,Sandia National Laboratories, Albuquerque, NM 87185–1110, e-mail: `bah@cs.sandia.gov`

after the send is complete. This requires each processor to simultaneously have space for both its outgoing and its incoming data, which is not always possible. To overcome this problem, instead of sending all the data at once, we can send in phases. In each phase only a fraction of the data is migrated, so less memory is required to receive messages. After each phase, processors can free up the memory of the data they have sent. That memory is now available for the next communication phase. We discuss efficient algorithms to exchange messages in minimum number of phases. We also discuss practical issues about implementation of these algorithms to exchange data under memory constraints.

2 Determining Communication Patterns

This section considers the problem of determining the communication pattern after each processor is assigned a portion of a problem. We assume that each processor can determine what data it needs to receive, but does not know which processor owns that data. This problem can arise in various applications. As a concrete example we will consider the very important case of matrix–vector multiplication, where the matrix is partitioned by rows. To perform matrix–vector multiplication a processor can examine its rows, find all the columns for which it has a nonzero and thus identify the components of the vector that it needs. But it does not know which processors own these components. To perform the communication, each processor has to know what it will receive from and what it will send to every other processor. The purpose of this work is to design efficient algorithms to determine this communication pattern.

A solution to this problem is useful for two reasons. First, the alternative is to have the partitioner (or some other code) determine this information and to have it included in the problem description. This alternative burdens the routines which set up the problem and adds additional complexity to software interfaces. By providing an efficient parallel solution to this problem, we simplify the use of parallel computers. Second, dynamic load balancing redistributes tasks amongst processors, and so changes the communication pattern. Although the repartitioning tool could carry some data along with it which facilitates the determination of the new pattern, a good solution to the general problem will obviate the partitioner of this responsibility.

A simple solution to the problem of determining the communication pattern is to have each processor exchange messages with all others. For instance, each processor can create a list of the items it needs and the processors can pass these lists around a ring. When a processor receives a list it checks to see if it owns the requested items. Unfortunately, this all–to–all communication will be slow and not scalable. It is possible to employ higher-dimensional variants of the ring algorithm like a 2D-torus algorithm where we first run the ring algorithm on subsets of processors to aggregate lists, and then exchange aggregated lists with the other processors. But these variants still perform all–to–all communication, only in a somewhat more efficient way. They also require more memory on each processor. In this work, we propose the *distributed directory* algorithm, which is a rendezvous algorithm, which avoids excessive communication while still requiring only a small amount of memory.

2.1 Solution Methods

In this section we will discuss the *ring* algorithm, its extensions to higher dimensions, and the directory algorithm we are proposing.

The ring algorithm circulates queries among processors in a ring fashion. At each step i, a processor p
(1) receives a list from processor $p-1$
(2) processes the list to determine what it has to send to $p-i$
(3) sends the remainder of this list to processor $p+1$.
For P processors, after $P-1$ steps, every processor will have seen every other query, and complete sending information – what each processor has to send to every other processor – will be generated. We also need to generate receiving information as well, which is easy to generate given the sending information. This algorithm requires each processor to have memory to store two lists at a time.

To decrease the number of steps, we can use higher-dimensional variants of the ring algorithm. For instance, in 2D-torus algorithm, we can think of the processors as being assigned to a 2D grid (even if the actual network topology is different). We can run the ring algorithm on each row of the grid, aggregating lists. These larger lists are then circulated via a ring algorithm in each column of the grid. This requires two ring algorithms of $\sqrt{P}-1$ steps, however the lengths of queries in the second step can be $\sqrt{P}$ times larger, since it is a collection of queries from all processors in a row. We can further increase the dimensions, and go all way up to hypercube or even all-to-all-communication in one step to further trade increased message size for fewer number of steps. In a k-dimensional torus algorithm each processor,
(1) Uses the ring algorithm to aggregate lists from processors that have the same k dimension as itself.
(2) Process these lists to generate sending information.
(3) Make a recursive call by replacing dimensions as $k-1$ and replacing its own list with the collective list.
In general, a k-dimensional torus algorithm requires k ring operations, where each ring has $P^{1/k}$ processors, so the total number of steps is $kP^{1/k}$. Unfortunately, the final steps involve an aggregation of $P^{(k-1)/k}$ individual lists. Going to higher dimensions increases not only the memory requirement, but also sizes of messages being transferred as well.

To avoid moving excessive volumes of data, we propose using a distributed *directory*, which is a rendezvous algorithm. In this algorithm each processor will be responsible for maintaining directory information for a subset of the components. The algorithm is presented in Figure 1.

The algorithm assumes that each item has some unique identifier associated with it which can be used to determine a directory processor which will be informed about the item. This directory processor is the meeting place where the owner and the requesters can *rendezvous*. Throughout the algorithm, processors work both

Directory Creation:
As a client:
Identify directory owner for all my items which might be queried.
Submit these items to the directory.
As a directory:
Receive Submissions;
Build a hash table of submitted lists.

Query Resolution:
As a client:
Identify directory owner of items I need.
Submit requests to the directory.
As a directory:
Receive queries.
Check hash table to determine owners of requested items.
Inform requesters of the results.
As a client:
Receive query results.

Figure 1. *The Distributed Directory Algorithm*

as clients, who have queries to be resolved, and directory servers that answer the queries.

The algorithm begins by creating a distributed data structure for answering queries. The directory is distributed across the same set of processors as those that want to figure out the communication pattern. First, each processor identifies its items that might be queried by another processor. For each such item, it determines which directory processor should own it via some kind of hash function. After sending its data to directory processors, each processor works as part of the directory, collecting submitted lists and organizing them into a data structure to answer queries.

With this directory, query resolution is straightforward. Each processor identifies the items it needs to know the owner of. For each such item, it determines the directory processor which owns it via the same hash function used to create the directory. As a directory processor, it then receives queries, and looks up the owner of the requested data, and returns the answer to the requester.

One of the critical issues for performance is to limit the size of the directory. We want to maintain the directory for only those items that will be queried. For symmetric matrices (or symmetric communication in general) rows that need to be queried are those rows that require communication for themselves (equivalently, the boundary vertices on the graph of the matrix). Thus, only items that will actually be queried need be placed in the directory.

If communication is not symmetric, it is not possible to determine the minimal set of items that need to be submitted to the directory. An easy solution would require processors to submit their entire domain to the directory. However, typically only a very small portion of a processor's domain will be queried, so submitting the whole domain will be inefficient. Notice that the ring algorithm and its variants do not suffer from this problem, and they can be adopted without modifications. However, in practice unsymmetric matrices often have a lot of symmetry in them. We can exploit this observation to reduce the size of the problem by first using the directory algorithm. The directory algorithm will resolve queries associated with the symmetric matrix structure communication. We can use a ring–variant algorithm to resolve any remaining queries (i.e., items with unsymmetric communication). As the experiments in the next section reveal, the time to run the directory algorithm is much smaller than all ring variants. So we can afford a preprocessing step that reduces the problem size to improve the efficiency of the ring algorithm.

2.2 Experimental Results

We have tested our algorithms for determining communication pattern for the *ocean* matrix, a symmetric matrix of dimension 143,437 with 962,623 nonzeros. We ran our experiments on the ASCI Red parallel computer at Sandia National Labs. Our results are presented in Fig. 2 for 4, 16, 64, 256 and 1024 processors. The matrix was partitioned using the multilevel algorithm in Chaco. For these experiments the items (matrix rows) had a global numbering which we used as the identifier. For a hash function, we simply assigned the first N/P items to the first directory processor. The number of the rows of the matrix contains some implicit locality structure which this hash function exploits. Thus, a processor probably only communicates with a subset of directory processors. A random hash function might improve load balance in the directory, but it would likely increase the number of messages involved in constructing and querying the directory.

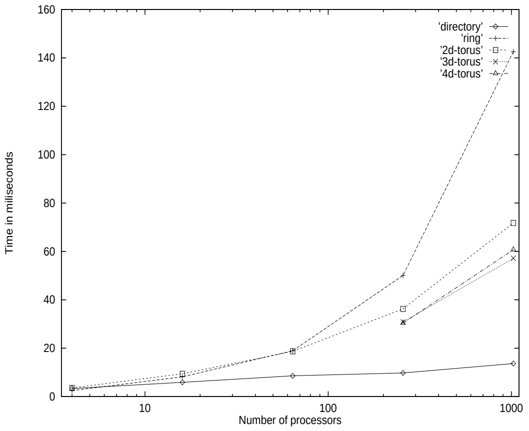

Figure 2. *Performance of Different Algorithms*

As expected, the directory algorithm scales well with increased number of processors whereas performance of the ring algorithm deteriorates. The 2D algorithm performs better than the 1D ring, but still is not scalable. Further increasing the dimensions of the ring algorithm slightly improves performance, but the gain that can be achieved by increasing number of processors is limited, as proved by the similarity between the performances of the 3D and 4D versions for 256 and 1024 processors. Recall that these ring variants also require more space than the distributed directory.

3 Communication with Memory Constraints

Although a number of algorithms and software tools have been developed to repartition work among processors (see, for example, [1, 3] and references therein), the mechanics of actually moving large amounts of data has received much less attention. If sufficient memory is available, the simplest way to transmit the data is quite effective. Each processor can execute the following steps.

(1) Allocate space for my incoming data
(2) Post an asynchronous receive for my incoming data
(3) Barrier
(4) Send all my outgoing data
(5) Free up space consumed by my outgoing data
(6) Wait for all my incoming data to arrive

The barrier in step (3) ensures that no messages arrive until the processor is ready to receive them, so no buffering is needed.

This protocol requires a processor to have sufficient memory to simultaneously hold both the outgoing and the incoming data since incoming messages can arrive before outgoing data is freed. An alternative way to view this issue is that for a period of time the data being transferred consumes space on both the sending and receiving processors. A protocol that alleviates this problem is desirable for three reasons. First, since many scientific calculations are memory limited, reserving space for this communication operation limits the size of the calculations which can be performed. Second, the amount of memory required by this protocol is unpredictable, so setting aside a conservative amount of space is likely to be wasteful. And third, a general purpose tool for dynamic load balancing should be robust in the presence of limited memory. It was the construction of just such a tool which inspired our interest in this problem [2].

To address these problems, we propose a simple modification to the above scheme. Instead of sending all of the data at once, we will send it in phases. After each phase, processors can free up the memory of the data they have sent. That memory is now available for the next communication phase. Since each phase can be expensive, it is important to limit the total number of phases.

We have studied the combinatorial aspects of this problem in our earlier work [4]. We proved that the problem of finding a minimum-phase schedule is NP-Complete, and presented efficient algorithms with tight bounds on performance. First, we will briefly explain our solution techniques below. Then, we will address more practical aspects of this problem.

3.1 Combinatorial Techniques for Minimum-phase Remapping

We have proposed two algorithms. The first algorithm depends on continuous relaxation, which relaxes the constraint that the amount of data transferred in a phase must be an integer. The second algorithm depends on *parking*, which tries to utilize memory that would otherwise be wasted. Both algorithms have tight bounds on their performance. Let T be the total volume of data to be moved, and M be the total available memory in the parallel machine. Note that M does not change between phases, even though its distribution might change. The minimum number of phases is $\lceil \frac{T}{M} \rceil$. We will use this lower bound to give bounds on the performances of our algorithms.

Continuous Relaxation

The approximation algorithm to be described in this section relaxes integral constraints on the volume of data transfers to allow continuous values. Naturally, the volume of transfer between two processors in a phase must be an integer. But integer solutions near the continuous ones can be used as heuristics. Note that the unit of data transfer is only a byte, whereas the volume of data being transferred is often in the order of megabytes. So, conversion from a continuous solution to an integer solution will often be a small perturbation. However, bad cases for this heuristic exist as discussed in [4].

The essence of the algorithm is to divide each message into $L = \lceil \frac{T}{M} \rceil$ equal pieces, and send one piece at each phase. Clearly, such a schedule does not necessarily satisfy memory constraints. To make this idea work, we use pre– and post–processing phases to ensure feasibility. In the pre-processing phase, receive assignments are redistributed to equalize the volumes of incoming and outgoing data on each processor. This ensures that each processor will have enough space for the next phase to receive, because the volume it needs is exactly equal to how much it ships out at the current phase. Each processor also needs to have enough space to receive for the first phase. For this purpose we reassign equal amounts of send and receive assignments to open up enough space at each processor for the first phase. The two steps can be merged into one phase, since we are only trying to balance numbers, thus send and receive operations can cancel each other. In the next L phases, we transfer $1/L$-th of each data transfer at each phase. We also need a post-processing phase to make up for the reassigned receives in the first phase. Altogether this algorithm gives a solution with $\lceil \frac{T}{M} \rceil + 2$ phases for the continuous approximation to the problem. A more detailed description and analysis of this algorithm can be found in [4].

Greedy Algorithms

We will describe the basics of a family of efficient algorithms that provide solutions in which the number of phases is at most 1.5 times that of an optimal solution. The algorithm is motivated by some simple observations. As discussed above, the minimum number of phases in a solution is $\lceil \frac{T}{M} \rceil$. This bound can only be achieved if available memory is used to receive messages at each phase. So free memory is wasted if it resides on a processor that has no data to receive. Our algorithm works by redistributing free memory to processors that can use it. Equivalently, data is *parked* on a processor with free memory it can't use, thereby freeing up memory on processors which can use it. We will only park data that needs to be transferred eventually.

Parking aims to utilize memory that would otherwise be wasted. Consider a processor that received all its data and still has available memory. This memory cannot be utilized in subsequent phases, decreasing the total memory which is usable for communication, thus potentially increasing the number of phases. Instead, another processor can temporarily move some of its data to this processor to free up space for messages. An example is illustrated in Fig. 3. In this simple example, the top two processors want to exchange 100 units of data, but each has only one unit of available memory. A simplistic approach will require 100 phases. However, the third processor has 100 units of free memory. By *parking* data on this third processor (i.e. transferring free memory to another processor), the number of phases can be reduced to three.

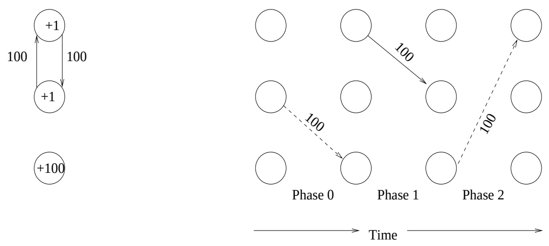

Figure 3. *Example of the utility of parking.*

In our algorithm, we merely store data in a parking space, and then forward it to its correct destination, when the destination processor has available memory. Note that it is inconsequential which processor owns the parked data. In other words, parking spaces are indistinguishable. What potentially effects performance is which processors shunt their data to parking space.

Below, we describe an algorithm that obtains a solution with at most 1.5 times the optimal number of phases. The algorithm is quite generic and allows for a number of possible enhancements.

Basics of a greedy $\frac{3}{2}$-approximation algorithm:

- A processor receives as much data as it can in each phase (i.e., if a processor has available memory at the end of a phase then this processor does not have any more data to receive).
- If the transfer request cannot be completed in the next phase, then park as much data as possible (i.e, park the minimum of the total parkable data and the total available parking space).

The second condition in this algorithm guarantees that each parking operation is followed by a direct transfer from the sender to receiver. This means at most half of the data being transferred can go through parking, thus total volume of data transfer cannot exceed $\frac{3T}{2}$. Also note that this algorithm uses all available memory in all phases with the exception of last two phases. These two observations imply an upper bound of $\frac{3T}{2M}+1$ phases, where the $+1$ term is due to not using all available memory in the last two phases. A formal proof of this argument can be found in [4].

Note that many details about the algorithm are unspecified: If I have more incoming data than free memory, which messages should I receive in the current phase? If several processors want to park data, but limited parking spaces are available, which should succeed? Intelligent answers to these questions could be used to devise algorithms with better practical (or perhaps theoretical) performance.

3.2 Practical Considerations

Our theoretical work concentrated on the total volume of available memory, ignoring how the available memory, and data to be sent reside in the memory. Our theoretical model can be applied directly if the available memory and the data to be sent are stored in a contiguous block of memory. In this case, all our results are exactly applicable without any caveats. First, we don't need to construct messages separately, because they are already in a contiguous block of memory, and ready to be sent. Second, since the available memory is contiguous, we can chunk it up into pieces as we like. If this was not true (memory was available in discontinuous blocks of memory), we might have to receive messages in smaller pieces. However, even if data and memory are contiguous to start a phase, at the end of a phase the some data will have been freed, so the available memory will no longer be contiguous. We can address this compacting the messages, shifting to have a continuous block of available memory. Proceeding this way, we will replace the data being sent by the data being received using the same block of memory.

In practice, outgoing messages and available memory are rarely available in a contiguous block, and organizing data for this purpose might be a challenging problem by itself. However, if data structures are designed to store data on a contiguous array, then a simple in-place permutation of the data is sufficient to achieve the desired arrangement of the data. As will be discussed below, this organization enables a more efficient and more aggressive operation while processors are exchanging their data.

An arbitrary data structure will bring its burdens. First, we will have to buffer messages to be sent (and received). This will decrease efficiency, since we are already short of memory. Second, available memory is likely to be scattered as small blocks in the memory. This will force the use of smaller messages, and available memory in subsequent phases will be unpredictable (pieces might be too small to be utilized), disabling the use of algorithms like continuous relaxation that rely on pre-scheduling the messages. In Fig. 4 we sketch an algorithm which is robust under these challenges. In this algorithm, the neighbor set of a processor refers to those processors it wishes to exchange data with.

Notice that in Fig. 4 parking is not included. Parking by its definition requires

> **While** data exchange is not complete **do**
>   Determine my send buffer size and my priority
>   Exchange buffer size and priority information with neighbors
>   **While** there is available memory and data to receive **do**
>   Choose a sender s depending on its priority
> $x \leftarrow \min($buffer size of s, remaining volume to be received from $s)$
>   Try to allocate up to x units of memory
>   Let y be the volume of memory allocated
>   Post asynchronous receive
>   Inform s that it can send y units.
>   **for** each sending assignment received
>   Construct message on the send buffer and send.
>   Free up the memory for data sent.

Figure 4. *Template of a robust algorithm for memory-constrained inter-processor communication*

a global operation among all processors, because any processor can park to any other processor, whereas data exchanges are localized, since each processor is likely to communicate with only its neighbors. Without parking, processors can exchange messages without any synchronization, but when we include parking, we have to use a global operation at each phase which slows down the overall process. Besides, in our experiments only a very small percentage of data is being parked, and the gain to due fewer number of phases is usually not worth the cost due to global operations in each phase. However, it is important to keep parking as a safeguard against cases where memory constraints are extremely tight.

4 Conclusion

We addressed two problems associated with redistributing data among processors. The first problem is that of determining the inter-processor communication pattern when each processor knows which components it wants to receive, but does not know which processor owns them. In a simple solution to this problem, processors will exchange messages in a ring fashion, however this will require all-to-all communication and will be unscalable. We described a rendezvous algorithm for an efficient solution for this problem, and our experiments verified its effectiveness. The second problem is that of actually migrating data in the case of limited memory. A processor can deallocate memory for outgoing data only after its send is complete. This requires each processor to have sufficient space both for its outgoing and incoming data, which is not always possible. We proposed a protocol to alleviate this problem and discussed implementation details.

Both of these problems arise from our collaborative efforts to build general purpose libraries to support complicated parallel applications. Both of the features described here are now being added into Zoltan, a public–domain dynamic load balancing tool [2]. Our distributed directory algorithm is also being added to the Aztec linear solver package [5].

Acknowledgements

We are indebted to Mike Heroux for introducing us to the problem of determining communication patterns.

Bibliography

[1] G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, J. Parallel Distrib. Comput., 7 (1989), pp. 279–301.

[2] K. D. Devine, B. A. Hendrickson, E. G. Boman, M. M. St.John, and C. Vaughan, *Zoltan: A dynamic load-balancing library for parallel applications – user's guide*, Tech. Rep. SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 1999. http://www.cs.sandia.gov/Zoltan/.

[3] B. Hendrickson and K. Devine, *Dynamic load balancing in computational mechanics*, Comp. Meth. Appl. Mech. Eng., 184:2–4 (2000), pp. 485–500. Invited paper.

[4] A. Pinar and B. Hendrickson, *Interprocessor communication with memory constraints*, Proc. of ACM Symp. Parallel Algorithms and Architectures, 2000.

[5] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid, *Official Aztec user's guide: Version 2.1*, Tech. Rep. SAND99-8801, Sandia National Labs, 1999. http://www.cs.sandia.gov/CRF/aztec1.html.