DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

On the Transportation and Distribution of Data Structures in Parallel and Distributed Systems

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

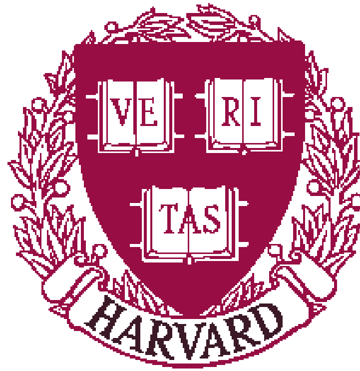
Citation	Fahmy, Amr F. and Robert A. Wagner. 1995. On the Transportation and Distribution of Data Structures in Parallel and Distributed Systems. Harvard Computer Science Group Technical Report TR-27-95.
Citable link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:26506440
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

On the Transportation and Distribution of Data Structures in Parallel and Distributed Systems

Amr F. Fahmy
and
Robert A. Wagner

TR-27-95

March 1995



Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

On the Transportation and Distribution of Data Structures in Parallel and Distributed Systems

Amr F. Fahmy*
Aiken Computation Lab.
Harvard University
amr@das.harvard.edu

Robert A. Wagner
Department of Computer Science
Duke University
raw@cs.duke.edu

March 29, 1995

Abstract

We present algorithms for the transportation of data in parallel and distributed systems that would enable programmers to transport or distribute a data structure by issuing a function call. Such a functionality is needed if programming distributed memory systems is to become commonplace.

The distribution problem is defined as follows. We assume that n records of a data structure are scattered among p processors where processor q_i holds r_i records, $1 \leq i \leq p$. The problem is to redistribute the records so that each processor holds $\lfloor n/p \rfloor$ records. We solve the problem in the minimum number of parallel data-permutation operations possible, for the given initial record distribution. This means that we use $max(mxr - \lfloor n/p \rfloor, \lfloor n/p \rfloor - mnr)$ parallel data transfer steps, where $mxr = max(r_i)$ and $mnr = min(r_i)$ for $1 \leq i \leq p$.

Having solved the distribution problem, it then remains to transport the data structure from the memory of one processor to another. In the case of dynamically allocated data structures, we solve the problem of renaming pointers by creating an intermediate name space. We also present a transportation algorithm that attempts to hide the cost of making a local copy for the data structure which, is necessary since the data structure could be scattered in the memory of the sender.

*Contact author. Address: Aiken Computation Lab., Harvard University, Cambridge, MA 02138. Phone: 617 495-5841.
Research supported in part by ARPA contract no. F19628-92-C-0113 and by NSF grant CDA-9308833.

1 Introduction

Programming distributed memory systems is not an easy task. In such systems data located in some processor's memory is not reachable by all the processors. This gives rise to the extra problem of moving data between memories. Such data movement is present in both message passing and shared memory systems. In message passing systems, the burden is on the programmer to issue commands to do these movements while in shared memory systems the operating system is responsible for fetching the data or keeping the caches coherent. Keeping the load balanced is another problem that is faced in parallel systems. One method of achieving load balance is to distribute data in an even fashion. Most of the time it is required to move structured data such as arrays or trees. This paper addresses the problem of even distribution of data structures and the transportation of these data structures in parallel and distributed systems. By even distribution we mean that each processor gets an equal portion of the data structure. In section 2 we present an algorithm that redistributes records of a data structure to achieve this balance.

The transportation problem is the problem of copying some data structure from some processor's memory to another. Some of the problems that arise are discussed in section 3, and include the problem of pointer renaming since the memories of the two processors may have independent address spaces. Another problem that arises in message passing systems especially is the problem of making a local copy of the data structure before it is being sent. The local copy is necessary since the data structure could be scattered while message passing functions require the message to be in contiguous space. An algorithm that hides this extra cost is discussed in section 3.4.

2 Even Distribution of Data Structures

In order to obtain good load balance in parallel and distributed systems it is necessary to distribute data structures in such a way that each processor receives an equal portion of the data structure assuming a uniform network of processors. In this section we develop an efficient algorithm to distribute data structure evenly among the processors. We call this the distribution problem.

We assume that n records of a data structure are scattered among p processors where processor q_i holds r_i records, $1 \leq i \leq p$. The problem is to redistribute the records so that each processor holds $\lfloor n/p \rfloor$ records. We solve the problem in the minimum number of parallel data-permutation operations possible, for the given initial record distribution. This means that we use $max(mxr - \lfloor n/p \rfloor, \lfloor n/p \rfloor - mnr)$ parallel data transfer steps, where $mxr = max(r_i)$ and $mnr = min(r_i)$ for $1 \leq i \leq p$.

Our solution consists of three stages, an information collection stage, a directive computation and distribution stage and finally the actual transportation stage. The information gathering stage is necessary to compute the number of records necessary to send/receive by each processor. Following this, it is necessary to compute, for each processor, the set of partner processor(s) it will communicate with in the actual transmission stage. Directives will be sent to each processor to indicate the members of this set. The methods developed in the next section will be used for the last stage.

2.1 Information Collection

Each processor will either send or receive a number of records to possibly many processors so that each processor will hold an equal number of records. The computation of the number of records a processor must send/receive will be computed by all the processors in a distributed fashion. The processors are configured into a binary tree where information will travel upwards towards the root. Each processor will receive two lists from each child. Entries of these lists are of the form (q_i, r_i) where r_i is the number of records held by processor q_i. The first list contains all the entries such that $r_i < \lfloor n/p \rfloor$ and the second list contains all the entries such that $r_i \geq \lfloor n/p \rfloor$. On receiving the lists, a processor creates two new lists with the same property by appending the appropriate lists it has received from its children, adding its own entry to the proper list and sending the two new lists to its parent. The root adds its own entry to the proper list and starts to compute the directives. This stage takes $logp$ steps and $2p$ messages to construct the two lists.

2.2 Directive Computation and Distribution

Directives are going to be of the form "send d records to processor q". The directive computation is done by the root processor using the two lists that it has received. Let $L1$ be the list of all processors holding less than $\lfloor n/p \rfloor$ records, $L2$ be the list containing the rest of the processors, $L[i].r$ be the number of records of the ith entry in list L and $L[i].p$ be the processor number of the ith entry in L. The following procedure computes the set of directives D in linear time in the number of processors. Elements of D are of the form (p,directive) where processor p is to execute the directive.

```
D = empty set;
i = 1;
j = 1;
while((i ≤ | L2 |) and (j ≤ | L1 |)) {
  while(L2[i].r == ⌊ n/p ⌋)
    i++;
  d1 = ⌊ n/p ⌋ - L1[j].r;
  d2 = L2[i].r - ⌊ n/p ⌋;
  d = min(d1,d2);
  append((L2[i].p,"Send d records to processor L1[j].p"),D);
  L1[j].r += d;
  L2[i].r -= d;
  if(L2[i].r ≤ ⌊ n/p ⌋)
    i++;
  if(L1[j].r ≥ ⌊ n/p ⌋)
    j++;
}
```

After computing the directives the root processor sends the directives to their destination processor using a downward pass in a tree scheme similar to the information gathering process. The root processor sends the directive destined to processor p to its child that is an ancestor of p. This process continues until processors at the leaves of the tree receive their directives.

3 Transportation of Data Structures

3.1 Introduction

We consider the problem of transporting a dynamically allocated linked data structure from the memory of one processor to the memory of another. Such a problem, we anticipate, will arise frequently in programming parallel and distributed systems. We note that the algorithms described here are analogous to compacting algorithms for garbage collection, see [CN83], except for section 3.4.

Consider a linked data structure such as a tree or a graph that resides in the memory of some processor. Usually such data structures do not reside in contiguous memory due to their dynamic nature. Their size is not known at the time they are created and thus one cannot reserve memory space for them. Most message passing functions take as arguments, a pointer to the beginning of a contiguous memory region and a number of bytes to send to the destination processor. Thus to send a data structure that does not reside in contiguous memory requires a gathering process to make a contiguous copy of the data structure. Message passing functions also perform better with messages of particular sizes; if the message size is too small the time needed to send routing data, and prepare the receiver for a message (collectively, the per-message latency time) exceeds the time needed to transmit bytes of the message, and the network total traffic load is increased. A message which is too large can exceed limits imposed by the message passing function. The first problem we consider is thus: How should the gathering process be done and how large should the messages be, in an environment where the data structure can be sent in multiple messages (segments). One way to send a data structure without a gathering phase is to send the nodes of the structure one at a time.

Another important problem arises with pointers. Consider a data structure consisting of one node that resides in the memory location p1. If this node is copied to the memory of another processor, location p1 can be already in use in the destination processor's memory and thus the single node must be located in some other free location p2. All pointers in the data structure which refer to p1 must then be changed to refer to p2 in the copied structure. With linked dynamically allocated data structures, such a problem will always arise.

Cyclic data structures and data structures that have shared nodes will constitute another problem that we shall address also.

We argue that the solution of gathering and segmenting the data structure is a function of the time needed to make a local copy of the data structure and the cost of transporting segments of certain sizes. These numbers are properties of the particular system being used and an efficient solution must take these numbers into consideration. We also give algorithms that address the pointer problem outlined above. In our solution, the data structure is packaged in the sender's memory for transportation so that the receiver is able to unpack it and allocate memory for it. The algorithm is fault tolerant in the sense that messages can be received out of the order in which they were sent while allowing the receiver to reconstruct the data structure correctly.

3.2 Pointer Conflicts Between Processors' Memories

Suppose that the nodes of a linked list are sent as is, without modification, to the destination processor, in arbitrary order. Each node, except the last one, will have a pointer to the next node in the list. However, it is impossible to reconstruct the linked list in the memory of the remote processor, since each node lost its identity by being moved from its original location.

We need a method by which we can identify nodes; one method is to give names to nodes. The sender replaces the pointer in the next field of each node with the name of the node it points to and sends each node along with its name as a tuple. The destination processor can now reconstruct the list by allocating space for each node and replacing node names with their new addresses. A table mapping pointers to node names will be maintained at the sender and a table mapping node names to pointers will be maintained at the destination processor. The table at the sender is maintained just in case of sharing; there are two nodes, n_1 and n_2, pointing to the same node, n_3. Once a name is given to n_3, it must be used in the pointer fields of both n_1 and n_2. At the destination processor, the table is maintained so that once n_3 is assigned a location, the pointer fields of n_1 and n_2 will point to the same location. The table at the sender is not needed where there is no sharing, as in the case of a linked list or a tree.

There is a variety of name spaces to choose from in assigning unique names to nodes. One possibility is to use the old pointer as a name of a node. Another is to assign an integer to each node etc. Each one of these name spaces requires sending the name and the node together as a pair. One solution that does not require sending the name with the node is to use the location of the node in the message as a name. If the data structure is being sent in several segments, with each segment containing a number of nodes, then a pair of integers consisting of a segment number and an offset within the segment can identify each node.

3.3 Transporting Data Structures Without Gathering

Copying large data structures to contiguous memory is expensive. A parallel program can do such a copy many times during its execution, thus substantially increasing its runtime. On the other hand sending a data structure one node at a time is the only other alternative if copying is too expensive. An alternative that does not exist at the time of writing this paper is a message passing function that takes multiple addresses and sends the contents of these addresses. So, for example, an array of N pointers can specify N nodes to be sent.

In this section we describe algorithms for sending and receiving data structures without copying the nodes. In this case each segment (message) will contain only one node, and thus message identification is enough to identify nodes. In most message passing functions in existing systems, when a message is sent, a tag identifying the message is sent with it to identify the message. We shall use this tag to identify nodes also.

We distinguish between two different types of data structures; the first is the class of acyclic *and* sharing-

free data structures. Sharing [HL82] means that there are two different nodes that point to the same node. Trees and lists are examples of acyclic sharing-free data structures. The second class is the class of data structures that are cyclic or permit sharing such as general graphs. We distinguish between the two cases because it is necessary for the sender (and the receiver) to know whether a node has been sent (received) or not. In the case of acyclic sharing free data structures a depth first traversal is sufficient to ensure that a node will never be visited twice. If sharing is permitted, a record must be kept at each node, indicating that the node has/or has not been "visited". This record is not needed, if no sharing occurs.

3.3.1 Sending Data Structures

To send an acyclic sharing-free data structure, the sender performs a depth first traversal of the structure sending each node as it is visited. Children of a node are sent before the parent so that references to the childrens' old names can be changed in the parent node.

Notice that it is not necessary to know the size of the data structure prior to sending it. This is not the case when the data structure has to be segmented prior to sending. A traversal of the data structure, just to determine its size, is necessary before sending can start. We note that the size can be maintained as the data structure is being changed.

As an example, consider a binary tree node, T, containing some data field and pointers to the left and right subtrees. To send T, its left subtree is sent first followed by the right subtree followed by T. The new name of each node is simply the order in which that node was visited during the depth first traversal. Notice that a node will never be sent twice in this scheme.

In the case of cyclic data structures and ones that permit sharing, a table mapping old names of nodes to new names must be maintained since nodes can be visited more than once. If there is a method to *mark* nodes, such as extra fields in the node, then during the depth first search these marks have to be checked before a node is sent and the new name of the node must be found from the mapping table.

3.3.2 Receiving Data Structures

Once a node, T, has been received at the destination processor, a check is made in a table, called the node table, mapping names of nodes to locations in memory, to see if space has already been allocated for that node. The reason for this step will be clear shortly. Space is allocated for the node if there is none allocated for it already.

To insert the received node into the local data structure, all the names of nodes referenced by T must be replaced by pointers to these nodes. The locations of these nodes can be found from the node table. Two situations can occur; the first is that the address of each node is known from the table. In this case the appropriate fields of T are replaced by pointers to the nodes. The second case is that one or both of these nodes have not yet arrived. In this case, space is allocated for the missing node(s) and the their locations recorded in the node table and in the appropriate fields of T. For this reason when a node arrives the node table is checked first to see if there is space already allocated for the new node.

Finally, it is necessary to have a fast method to find nodes in the node table. If an upper bound on the size of the data structure is known then an array can be used to implement the table. Otherwise a hash table will be necessary.

3.4 A System Dependent Efficient Solution

In this section we present a solution to the gather and sending problem that is adaptive to the system being used. We make the following assumptions:

- To make a local copy of 1 K-byte takes C time units,
- To send a message of size S K-bytes from one processor to another takes $\alpha + \beta S$ time units, where α is the latency of sending a message and β is the time needed to transmit one K-byte of data.

We compare two algorithms. The first gathers all the data and sends it in one message and the second partitions the data into m segments and sends each segment in a separate message.

3.4.1 The Single Segment Algorithm

This algorithm copies all the data into a contiguous place in memory and sends it to the destination processor in a single message. The time it takes to gather and send the message and receive it by the destination processor is thus the sum of these two operations which is:

$$SC + \alpha + S\beta$$

where S is the size of the data structure, in K-bytes.

3.4.2 The Multiple Segment Algorithm

This algorithm sends the data structure in M segments to the destination processor. It gathers the first segment and sends it. While the first segment is on its way, it gathers the second segment etc. Thus the operations of gathering and sending of segments are overlapped. Such an algorithm is only valid when overlapping of computation and communication is possible.

Gathering each segment takes SC/M time units and sending each segment takes $\alpha + S/M\beta$ time units. The total cost of this process is thus

$$SC/M + (M-1)max(SC/M, \alpha + S\beta/M) + \alpha + S\beta/M$$

but on all systems $\beta \geq C$ thus $max(SC/M, \alpha + S/M\beta) = \alpha + S\beta/M$ and the cost is

$$SC/M + M\alpha + S\beta$$

To find the optimal segment size we differentiate with respect to M and set the derivative to zero and we get

$$M = \sqrt{SC/\alpha}$$

and the cost is

$$2\sqrt{SC\alpha} + S\beta$$

4 Related Work

The problem of transporting and distributing data is fundamental to Computer Science. As we mentioned earlier the problem of gathering a data structure for transmission is very similar to the problem of compacting the data structures for garbage collection [CN83]. Sollins [Sol79] investigates the problem of moving complex data structures in a distributed system. Semantics and contexts of computation were the main issues explored. In Thor [DLMM94], a distributed object-oriented system, objects migrate between object repositories. Two different schemes to support object migration are presented, one using location-dependent names and the other using location independent names. Herlihy and Liskov [HL82] describe a technique for communicating abstract values between processors by defining a call-by-value semantics. The emphasis there is on "meaning of transmissions".

The two problems of transportation and even distribution of data structures arise in an implementation of the Barns Hut algorithm for the n-body problem within the Bulk Synchronous Parallel model of computation see [Val90] and [CFSV95]. The BSP model describes any parallel system using three parameters, p, the number of processors, g the ratio of computation to communication and L, the cost of synchronization. In the Barns Hut algorithm, a tree is constructed and a decision must be made, based on the parameters of the system, whether to rebuild the tree or to transfer parts of the tree to processors with less work to do [SV95].

5 Conclusions

In this paper the problems of the even distribution and the transportation of data structures were discussed. Algorithms that allow the creation of functions for this purpose were presented. These algorithms are expected to be useful in practical distributed systems when large data structures involving pointers are employed.

References

[CFSV95] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk synchronous parallel computing– a paradigm for transportable software. *Proceedings HICSS95*, II, 1995.

[CN83] J. Cohen and A. Nicolau. Comparison of compacting algorithms for garbage collection. *ACM transactions on programming languages and systems*, 5(4), 1983.

[DLMM94] M. Day, B. Liskov, U. Maheshwari, and A. Myers. References to remote mobile objects in Thor. 1994.

[HL82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM transactions on programming languages and systems*, 4(4), 1982.

[Sol79] K.R. Sollins. Copying complex structures in a distributed system. Master's thesis, MIT Laboratory of Computer Science, 1979.

[SV95] D. Stefanescu and L. Valiant. Personal communication. 1995.

[Val90] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), 1990.