AT&T Bell Laboratories Technical Journal
Vol. 63, No. 6, July–August 1984
Printed in U.S.A.

Program Transformations for Data Access in a Local Distributed Environment

By J. D. DeTREVILLE* and W. D. SINCOSKIE†

(Manuscript received March 23, 1983)

This paper presents a set of program transformations that are useful in transforming certain sequential program schemas for use in a local distributed environment. The environment is considered to be a set of processors connected by a local area network with broadcast capability. Examples of transformed program schemas are given that implement shared data, maximization, and abstract queues in a distributed environment.

I. INTRODUCTION

The use of program transformation has been frequently proposed as an aid to program development and program structuring.[1,2] Transformations that preserve program correctness can be used to convert a clearly written program that is unfortunately inefficient or otherwise unsuited to its environment into an equivalent implementation that is more directly usable. Moreover, if transformations are performed mechanically (although, as typically proposed, guided by the user), the developmental relationship between the original program and the transformed program can be retained explicitly, aiding in later understanding of the transformed program and simplifying further changes.

1. Typical simple transformations involve operations such as moving invariant computations outside of loops, or eliminating recursion

*AT&T Bell Laboratories. †AT&T Bell Laboratories; present affiliation Bell Communications Research, Inc.

by the use of explicit stacks; these can achieve greater efficiency at a cost in simplicity.

2. Another family of transformations involves exploitation of certain dualities to transform a program into its dual. For example, Wall notes a duality between sites in a distributed environment and the messages they exchange and proposes that certain classes of programs (in particular, those programs whose structure is related to the topology of the network) be written from the point of view of the messages themselves.[3] Useful insights can be gained through the use of this approach, and since such programs can be mechanically transformed into programs written from the point of view of the processors sending the messages, the new process is easier to execute.

This paper presents a set of program transformations that can be used for programs in a distributed environment to convert references to local data relations into references to the data over a network. We first note a duality between certain looping constructs and a particular distributed communication structure. Programs iterating over a global data relation stored locally are shown to be equivalent to programs making explicit requests to external processors for data; programs written in the first form are easy to understand, but must be converted to the second form to be executed. We then present a further set of transformations that can be performed on programs in the second form to make their communications more efficient.

II. THE DISTRIBUTED ENVIRONMENT

Consider the distributed environment shown in Fig. 1, with some number of similar sites connected over a local area network. These sites are loosely coupled architecturally, with all communication achieved via messages exchanged over the network.

The assumption of a local area network suggests high-speed operation. It also suggests the ability for any one site to broadcast a message to all other sites.

Within this architectural model, assume that we wish to provide some set of shared data, accessible to all the sites and containing information related to the sites themselves. For example, consider the case of a distributed telephone system in which each site controls

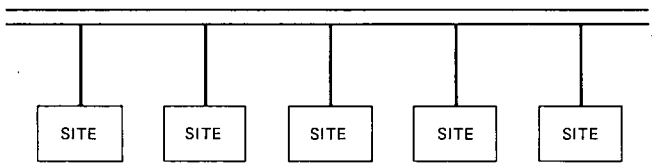

Fig. 1—Distributed environment.

some small number of telephones. A data relation mapping telephone numbers to site addresses could be used as a directory to determine which site is associated with a given telephone number. The question arises of where to store these data. There are at least three distinct approaches:

1. The data could be redundantly stored at every site, requiring quadratic total space (linear at any site); reliability would be very good. Accessing the data would be simple. Updating them could entail significant complexity and cost. For the telephone example above, each site would contain a complete directory, which would be bulky and difficult to update.

2. The data could be stored at some central site, at low cost but with poor reliability. Service would be lost if the central site were to fail. Accessing as well as updating the data would involve communication with the central site. For the telephone example, there would be a central directory server, giving good space efficiency but not allowing telephone calls to be made if the server were unavailable.

3. The data could be stored across sites, with each site holding the data pertaining directly to it. Accessing the data would typically involve broadcast communication with all sites, as could updating the data. Reliability could be very good, although this approach requires quadratic total time (linear at any site). For the telephone example, each site would know only its own telephone numbers and its own site address. Mapping a telephone number to a site address would involve a broadcast message to all sites, followed by a reply or replies. If a site were down, only calls involving it would be affected. For a moderate number of sites, this approach should be reasonably time-efficient.

This paper assumes that the third case is chosen. Thus, programs accessing shared data will need to communicate with the various sites where the data are actually stored.

We show that a program written as though these distributed data were available locally, as in the first case, can be mechanically transformed into one with explicit communication, as in the last case. We also show that this communication can often be made more efficient through the use of further transformations. The increases in efficiency occur with a reduction in the number of messages transmitted. Depending on the network being used and other particulars, there may or may not be a significant advantage to doing this.

III. TRANSFORMING SIMPLE LOOPS

The basic transformation presented here transforms looping structures over data relations into a distributed message-passing structure. The original control structure is a simple iteration over data, while the network topology is a simple iteration of sites. The transformed

structure is a simple broadcast to the sites, followed by their iterative replies. We note that, as in Wall's approach, the original control structure corresponds to the topology of the network and is transformed into the communication structure of the resulting program. Consider the pseudocode program fragment:

```
loop for tuple in relation do
    "perform operation on tuple"
end loop
```

If the tuples of the data relation are distributed across sites, this can be transformed into:

```
broadcast (this_relation_id);
loop until all_replies_received do
    receive tuple;
    "perform operation on tuple"
end loop
```

where a separate process at each site performs:

```
loop do
    receive request;
    case request.type in
    ...
    this_relation_id:
        loop for tuple in relation_here do
            reply tuple
        end loop
  ...
  end case
end loop
```

Here, an iteration over all tuples in a local relation is transformed into a broadcast request for all sites (including this one) to transmit as replies those tuples of that relation that they remotely store (held in *relation_here*; the constant *this_relation_id* names which relation is being requested), followed by an iteration over the replies. (It is assumed that the order of iteration is unspecified for the original looping construct.)

Since all sites receive the broadcast request almost simultaneously, they could be ready to transmit their replies at about the same time. On contention networks, such as *Ethernet*,*[4] this could lead to a low transmission efficiency due to collisions and retransmissions. On such networks, we could have each site choose an appropriate random delay time to wait before transmitting its tuples.

This transformation covers most accesses to data stored in relations,

*Trademark of Xerox Corporation.

in which the particular tuple or tuples to be used are not known beforehand. If they are known and their home site is also known, we may apply the obvious additional transformations to communicate directly with that site.

In the simplest case, the access to the tuples is read-only. If a tuple is to be updated within the loop, an additional message must be returned to the site from which the tuple was sent.

The problem of synchronization of multiple processes at multiple sites accessing shared data is not considered here. Algorithms for mutual exclusion in a distributed environment are presented by Ricart and Agrawala,[5] including certain approaches similar to those later in this paper.

It may be difficult for the requesting site to determine when all replies have been received. Although the number of sites may be known, some of these may currently be inactive and thus may not send replies. The use of time-outs seems the only workable solution to this problem within the distributed framework assumed here. Since the relative speeds and response times of the sites may vary, this time-out might need to be fairly large, possibly limiting the range of applications of this approach to systems with infrequent accesses to shared data. In the case of a telephone system such accesses would be required only during the call setup, which occurs relatively infrequently and which only needs to proceed at human speeds. We note again that the failure of sites to reply when they are unavailable can be viewed as perfectly appropriate if the tuples being accessed relate to the sites themselves. If a site is down, it can be viewed as nonexistent and so its data should not be seen. Thus, the semantics of the original programs have been (unavoidably) extended to deal with site or communications failures.

In certain cases (where, for instance, there is only one tuple per site), the tuple information may be stored implicitly and regenerated on each request. If this is done, the nature of a data relation will have undergone a substantial shift. Where previously it would have been a data structure in its own right, now it would exist only as an effect of the control structure of the transformed program.

IV. MOVING FUNCTION EVALUATION TO REMOTE SITES

Consider the program fragment:

```
loop for tuple in relation do
    "perform operation on f(tuple)"
end loop
```

where f is some side-effect-free function applied to the tuple. The function can be computed at the remote sites and the value returned instead of the tuple. The program fragment above becomes:

```
  broadcast (this_relation_id);
  loop until all_replies_received do
       receive f_of_tuple;
       "perform operation on f_of_tuple"
  end loop
```

while the separate process at each site is transformed to:

```
  loop do
       receive request;
       case request.type in
       ...
       this_relation_id:
            loop for tuple in relation_here do
                 reply f(tuple)
            end loop
    ...
     end case
  end loop
```

Here, all sites combine the application of f to the tuples in parallel, which can decrease the total running time of the application by parallel computation and may reduce the length of messages if f maps a tuple into less total data. The sites may be heterogeneous and the computation of f may depend on some characteristic of the site to which the data refer. Having each site compute its own f does not require the requester to know how to compute f for the data of other sites. Each site need only know how to compute its own f, which could be useful in a heterogeneous network.

V. MOVING TESTS TO REMOTE SITES

The transformation of loops over tuples in a relation, turning a program that accesses data as though it were local into one explicitly requesting remote data, is the basic transformation considered here. After this is done, there are further transformations possible that can make the communication more efficient.

The first additional transformation allows us to perform certain tests remotely. For example, assume that we have transformed the program fragment:

```
  loop for tuple in relation do
       if tuple.key = value then
          "perform operation on tuple"
    end if
  end loop
```

into:

```
  broadcast (this_relation_id);
  loop until_replies_received do
```

```
        receive tuple;
        if tuple.key = value then
          "perform operation on tuple"
      end if
    end loop
```

with appropriate remote server processes receiving the broadcast messages and returning their tuples. We can now invoke a further transformation of the program fragment into:

```
  broadcast ((this_relation_id, this_test_id), value);
  loop until all_replies_received do
        receive tuple;
        "perform operation on tuple"
  end loop
```

with the remote server process transformed to:

```
  loop do
        receive request;
        case request.type in
        ...
        (this_relation_id, this_test_id):
          loop for tuple in local_relation do
                if tuple.key = request.value then
                  reply tuple
                end if
              end loop
          ...
          end case
        end loop
```

Here, we have moved a test on the tuples to the sites where the tuples are stored. The constant *this_test_id* is used to identify which test the remote sites should perform. Since the remote sites will transmit a reply only if the test succeeds, this transformation can sharply reduce the number of messages transmitted. We note that, since the variable *value* appears free in the test condition, its value must be transmitted to the remote sites for their evaluation of the test.

In the telephone example, we would broadcast a request containing the telephone number to all sites, and, assuming uniqueness of telephone numbers, the site replying would be the one controlling the telephone with that number.

VI. MOVING LOOP TERMINATION TO REMOTE SITES

Consider the program fragment:

```
found := false;
loop for tuple in relation do
```

```
        if tuple.key = value then
          found := true;
          break;
        end if
    end loop
```

Using the transformation in the previous section, we can move the test condition to the remote sites. Moreover, we can then make the loop termination (in this case *break*) operate remotely by having the remote servers, while they are waiting to transmit their own replies, watch the network for any prior reply. If one is seen, the site should abort its own reply (and if it had more than one reply possible, it should transmit only one).

Due to race conditions, the requesting site may still receive multiple replies to its requests. Any late replies should be identified and discarded. Of course, this was already the case with replies received after a time-out and can be implemented through the use of serial numbers on requests and their corresponding replies.

VII. MOVING MAXIMIZATION AND MINIMIZATION TO REMOTE SITES

Just as loop termination can be moved to remote sites, as shown in the last section, so can certain ongoing loop computations. One important example is maximization (or minimization) of a quantity over a relation.

Consider the program fragment:

```
    max := negative_infinity;
    largest_tuple := nil;
    loop for tuple in relation do
        if max < tuple.value then
          max := tuple.value;
          largest-tuple := tuple;
        end if
    end loop
```

Applying the transformation to this fragment would give:

```
    max := negative_infinity;
    largest_tuple := nil;
    broadcast (this_relation_id);
    loop until all_replies_received do
        receive tuple;
        if max < tuple.value then
          max := tuple.value;
          largest_tuple := tuple;
        end if
    end loop
```

The remote process would wait for requests and respond with the maximum-value tuple existing in the remote machine.

This transformation can be improved if one takes advantage of the broadcast nature of many local networks. The process at the remote site can delay its reply, while watching the network for other replies larger than its own. If a larger value is seen, the reply is aborted. In general, if the sites reply in random order, the expected value of the number of replies is reduced from N to $\log_2 N$, since about half of the sites will drop out on each reply seen. An even greater improvement can be realized, if the distribution of values is well understood, by having each remote site's delay be inversely correlated with its value, letting the sites with the larger values transmit first.

For the telephone example, there are many times when it is necessary to maximize or minimize some quantity over a relation. For example, when calling a hunt group (a set of telephones with the same number), one wants to find the nonbusy telephone with that number (a pair of conditionals, which can be distributed) that is the earliest in the list (the one that has been idle the longest, or that has not rung for the longest time).

In a distributed computer system, resource allocation is often a maximization problem. For example, if one site finds itself overloaded, it could find the site with the maximum available resource, such as available processor time or main memory, and off-load part of its work to that site. Independent derivations of this approach have been reported earlier: Farber and Heinrich applied it as a "bidding" mechanism for load sharing in a distributed computer system.[6]

Certain abstract queues can be implemented using maximization. A site would enter a queue of sites simply by setting an internal flag. The operation of finding the head of the queue would involve finding the site that has been in the queue the longest time. Here, as before, the queue itself has disappeared. Sites are in the queue if and only if they think they are in the queue; the queue is only an effect of their control structure. Such a scheme can be quite robust, since if a site in the queue fails, then it is no longer in the queue (since it will not reply to queries). Queues of any objects closely coupled to sites can be created in a similar fashion.

VIII. CONCLUSION

A number of transformations have been presented that turn some common sequential program schemas into distributed program schemas. These transformations operate by taking a data structure that exists at a single site and distributing it among a number of sites in a distributed environment. An effect of the distribution is that the

explicit data structure "disappears" from the program, and thereafter exists only as an effect of the control structure in the distributed environment.

The transformations presented here have been applied manually to implement initially sequential algorithms in certain experimental distributed environments. Useful distributed program schemas included locating resources in a distributed environment and performing distributed maximization.

REFERENCES

1. R. Balzer, N. Goldman, and D. Wile, "On the Transformational Implementation Approach to Programming," Proc. Second Int. Conf. Software Eng. (October 1976), pp. 337–44.
2. M. S. Feather, "A System for Assisting Program Transformation," Trans. on Programming Languages and Systems, *4,* No. 1 (January 1982), pp. 1–20.
3. D. W. Wall, "Messages as Active Agents," Ninth Annual ACM Symp. on Principles of Programming Languages (January 1982), pp. 34–9.
4. R. M. Metcalf and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Commun. ACM, *19,* No. 7 (July 1976), pp. 395–404.
5. G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," Commun. ACM, *24,* No. 1 (January 1981), pp. 9–17.
6. D. J. Farber, and F. R. Heinrich, "The Structure of a Distributed Computer System—The Distributed File System," Proc. Int. Conf. Computer Commun. (October 1972), pp. 364–70.

AUTHORS

John D. DeTreville, B.S. (Mathematics), 1970, University of South Carolina; S.M. (Computer Science), 1972, Massachusetts Institute of Technology; Ph.D. (Computer Science), 1978, The Massachusetts Institute of Technology; AT&T Bell Laboratories, 1978—. Mr. DeTreville's doctoral thesis was in applied Artificial Intelligence. His first work at AT&T Bell Laboratories was on 5*ESS*™ switching equipment; he moved to Murray Hill in 1980, where his work has incorporated topics in distributed systems and program synthesis. Member, ACM.

W. David Sincoskie, B.E.E., 1975, M.E.E., 1977; Ph.D. (Electrical Engineering), 1980, University of Delaware; AT&T Bell Laboratories, 1980–1983; present affiliation Bell Communications Research, Inc. While at AT&T Bell Laboratories, Mr. Sincoskie was performing research in distributed computing, computer networking, and operating systems. Since 1982, he has been working with integrating voice and data switching on local area networks. In 1983, Mr. Sincoskie was appointed District Research Manager, Computer Communications Research, at Bell Communications Research, Inc. Member, Tau Beta Pi, Eta Kappa Nu, IEEE, ACM.