NASA Contractor Report 181987

ICASE INTERIM REPORT 9

A SCHEME FOR SUPPORTING DISTRIBUTED DATA STRUCTURES ON MULTICOMPUTERS

Seema Hiranandani
Joel Saltz
Harry Berryman
Piyush Mehrotra

NASA Contract No. NAS1-18605
January 1990

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

ICASE INTERIM REPORTS

ICASE has introduced a new report series to be called ICASE Interim Reports. The series will complement the more familiar blue ICASE reports that have been distributed for many years. The blue reports are intended as preprints of research that has been submitted for publication in either refereed journals or conference proceedings. In general, the green Interim Report will not be submitted for publication, at least not in its printed form. It will be used for research that has reached a certain level of maturity but needs additional refinement, for technical reviews or position statements, for bibliographies, and for computer software. The Interim Reports will receive the same distribution as the ICASE Reports. They will be available upon request in the future, and they may be referenced in other publications.

Robert G. Voigt
Director

A Scheme for Supporting Distributed Data Structures on Multicomputers *

Seema Hiranandani *Joel Saltz* *Harry Berryman*
Piyush Mehrotra
Institute for Computer Applications in Science and Engineering,
NASA Langley Research Center,
Hampton VA 23065

Department of Computer Science
Yale University
New Haven, CT 06520

January 3, 1990

1 Abstract

We propose a data migration mechanism that allows an explicit and controlled mapping of data to memory. While read or write copies of each data element can be assigned to any processor's memory, longer term storage of each data element is assigned to a specific location in the memory of a particular processor. Our proposed integration of a data migration scheme with a compiler is able to eliminate the migration of unneeded data that can occur in multiprocessor paging or caching. The overhead of adjudicating multiple concurrent writes to the same page or cache line is also eliminated. We present data that suggests that the scheme we suggest may be a practical method for efficiently supporting data migration.

*This work was supported by the U.S. Office of Naval Research under Grant N00014-86-K-0310, and under NASA contract NAS1-18605 while the authors were in residence at ICASE, NASA Langley Research Center.

2 Introduction

It is well known that data distribution and load balance play critical roles in determining the performance one can expect to obtain from distributed machines. Data must be moved from processor to processor in response to computational demands. One way of supporting data migration is to explicitly designate blocks of data that are to be prefetched into the memory of a given processor and to copy the data into customized data structures. Programs are written on each processor with intimate knowledge of the format used to store off-processor data. For some problems, this approach to data distribution can be extremely efficient. Programming in this manner can be very time consuming, and can lead to programs that are difficult to debug.

Mechanisms have been proposed to allow data to migrate in an automatic fashion. The physical memory of a multiprocessor is viewed as a single logical memory. Data migrates to the processors that refer to particular logical memory addresses. The following methods have been proposed to support data migration [1], [4], [3], [5]

- Multiprocessor paging: A logically shared memory is divided into *pages* which are contiguous, equal sized ranges. Processors store copies of required pages in their local memories. A page table is used to find the page corresponding to a given address in logical memory.
- Multiprocessor Caching: Each processor stores copies of the contents of address ranges in a logically shared memory. A subset of address bits are used to determine the location of data in physical memory.

Before describing our methods, we will outline some of the well known shortcomings of each of these data migration mechanisms. One of the principal difficulties with multiprocessor paging is the problem of false sharing. In a given portion of code, most locations in logical memory may not be accessed by multiple processors. Since pages consist of ranges of contiguous memory locations, different portions of a given page may be accessed by different processors. At best, false sharing will cause processors to waste physical memory to store data that will not be used. False sharing has the potential for causing particularly severe performance degradation if multiple processors attempt to concurrently write to different memory locations on a given page. Decreasing page size will tend to reduce false sharing. However when page size is reduced, page table storage requirements and communications latency overheads increase.

In multiprocessor caching, we may also see false sharing when a large cache line size is chosen or alternately experience significant communications latency effects when small cache lines are employed. Furthermore, since the cost of obtaining off-processor data may be very high compared to the cost of fetching data from local memory or cache , we want to maintain an extremely high hit ratio. It is well known that it is easy to find patterns of data access that make very poor use of cache memory. It may not be possible to maintain an extremely high hit ratio in a data cache.

Finally, in both caching and paging schemes, maintaining coherency in large scale multiprocessors may be associated with very high overheads.

2.1 Overview of Hashed Cache Data Migration Scheme

We support distributed data structures in a way that allows an explicit and controlled mapping of data to memory. While read or write copies of each data element can be assigned to any processor's memory, longer term storage of each data element is assigned to a specific location in the memory of a particular processor. A processor needing to read or write an array element gets a copy of that element. We constrain the form of programs so that only parallel loops or sequential code can be specified. Each data element can be written to by at most one processor during the course of a single set of (nested) parallel loops. This constraint eliminates the need for hardware coherency support. At the end of a set of nested parallel loops, all modified data is copied back to its home location.

A loop is transformed into two parts by a compiler. One part is called an *inspector*, the other is called an *executor*. The inspector is responsible for determining what data elements are required by a loop, the executor carries out the actual computations.

In a distributed memory machine, execution time procedures carry out the actual fetching of data. Local copies of data are stored in a hash table. Access to the table must be very fast. Location the in hash table is determined by taking low order bits of a quantity that is analogous to an address in logical memory. Unlike a traditional cache, we cannot afford to allow data elements to be thrown away just because too many required addresses have the same low order bits. We instead use a linked list when more than a single data element hashes to a given location.

3 The Hashed Cache System

3.1 Support of Distributed Arrays

We support distributed multidimensional arrays in a way that allows an explicit and controlled mapping of data to memory. As stated above, long term storage of each data element is assigned to a specific location in the memory of a particular processor. Users are able to specify the following attributes in their distributed array initializations:

- The topology of the processor array on which the data arrays are to be embedded
- The dimensionality of the data arrays
- Subset of processors used to store array elements
- Mapping of array elements onto the specified processors

Once a distributed array is initialized, we can use the specified partitioning information to find, for any distributed array element, a unique processor P along with a unique location in that processor P's storage. In each processor, contiguous memory locations are used to store local elements of a given distributed array. The unique location in P's storage can thus be expressed as an integer offset 0. Further details of our support for distributed arrays are beyond the scope of this note but can be found in [2], [6].

```
S1  do iter=1, num
S2    do i=1,n**2
S3      do j=0, m
S4        y[i] += values[i][j]*yold[nbrs[i][j]];
        end do
      end do

S5    do k=1,n**2
       yold[k] = y[k];
      end do

    end do
```

Figure 1: Sparse Matrix Vector Multiply

3.2 Details of the Hashed Cache System

The program in Figure 1 will be used as a running example in this discussion. This program performs a sequence of matrix vector multiplications. In order to compute `y[i]` at each iteration, we need `yold[nbrs[i][j]]`. Both `y` and `yold` are to be defined as distributed arrays. When the loop is distributed, loop iterations may be assigned to processors in an arbitrary fashion. Consequently long term storage of elements of `y` or `yold` may not be assigned to the processors that execute code referring to those elements.

The global arrays are initialized at the start of the program. We proceed to describe the primitives that support the *inspector* and *executor* phases of the *hashed-cache* system. To best understand the details of the *inspector* and *executor* phases we describe them in the context of the example presented in Figure 1.

Firstly, there are 3 different kind of references to global arrays.

1. Local: The address of the reference corresponds to the local memory of the processor. The reference may be a read or a write.
2. Non-Local Read: The address of the reference corresponds to the local memory of some other processor. The element is only read.
3. Non-Local Write: The address of the reference corresponds to the local memory of some other processor; the element is written to.

3.3 The Inspector Phase

Figure 2 depicts the psuedo-code of the *inspector* phase for the sparse matrix vector multiply. During the *inspector* phase we go through the inner-loop once to check for local and non-local

global array accesses. If an array reference is local we do nothing. However, if it is a non-local reference to a global array, we compute the processor on which the element resides and its offset. We need to store this information in such a way that accessing it is efficient. This is achieved by using a *hashed cache* scheme.

Initially, we allocate a certain amount of memory for a cache. We partition the cache into blocks, one for each globally defined distributed array. Each block is treated as a separate hash table. The location of a non-local distributed array element is determined by a hash function. Currently , we use a hashing function that simply masks the lower k bits of the key where k depends on the size of the hash table. The key is formed by concatenating the processor-offset pair, (P, O), that corresponds to a distributed array reference. Each entry in the hash table consists of the following:

1. a reference to the non-local data item, i.e., the data item's processor-offset pair,
2. whether the item is to be read (read flag),
3. whether the item is to be written (write flag),
4. the data value itself.

If the data item is a non-local read reference R, it is processed by the *process-global-read()* routine. The routine is described as follows:

process-global-read()

1. Search for the reference R in the *hashed-cache*.
2. If R exists and the read flag is set, do nothing.
3. If R exists and the read flag is not set, set read flag.
4. If R is not found the in hashed cache, create an entry with read flag set and enter it in the *hashed-cache*.
5. In the latter two situations, increment a count variable that contains the number of non-local elements to be gathered from the processor P on which this element resides. The offset of this element is written to a list containing the offsets of all the elements to be gathered from P.

Non-local array references R that are written to, are processed by the *process-global-write()* routine described below:

process-global-write()

1. Search for the reference R in the *hashed-cache*.

```
Loop over local iterations i assigned to P

  do j = 0,m

      Compute processor, offset pair for element of yold

      If yold reference is to non-local array element,
            process-global-read()

  end do

End loop over local iterations

Loop over local iterations k assigned to P


  If yold reference is to non-local array element,
        process-global-write()

End loop over local iterations
```

Figure 2: Inspector: Sparse Matrix Vector Multiply

2. If R exists and the write flag is set, do nothing.
3. If R exists and the write flag is not set, set write flag.
4. If R is not found in hashed cache, create an entry with write flag set and enter it in the *hashed-cache*.
5. In the latter two situations increment a count variable containing the number of non-local elements to be scattered to P. The offset of R is written to a list containing the offsets of all the elements to be scattered to P.

At the completion of the *inspector* phase we precompute the communication pattern required to efficiently gather or scatter all relevant non-local data referenced in the loop. This requires a global communication phase in which all processors participate. For a detailed description, see [6], [2].

3.4 The Executor Phase

Figure 3 depicts the pseudo-code of the *executor* phase for the sparse matrix vector multiply.

The non-local data required by the inner loop is first obtained from other processors and stored in the hashed-cache by the *process-gather-data* routine. We now proceed to execute the doall loop.

During the execution of the inner-loop we check each distributed array reference to decide whether it resides in the local array or not. If it does, we compute the offset of the element in the local array and fetch the data item from the appropriate memory location. If it does not, we fetch it from the hashed cache. If the array reference occurs on the left hand and it is non-local, we enter the new value in the *hashed cache.* At the end of the execution of the inner-loop, each processor calls the *process-scatter-data()* routine. This routine goes through the list of non-local offsets of elements to be scattered, searches for these elements in the *hashed cache* and writes the value to a list containing the new values to be written to the distributed memory. The data is then scattered to the distributed memory.

The operations for computing processor number and offset are computationally very cheap since we assume the distributed array may be partitioned in a block or block wrap fashion. The size of each block is a power of 2 and thus we need to perform simple integer operations such as shifts to compute the offset and processor number of a distributed array element.

4 Experimental Results

The program depicted in Figure 1 exhibits greatly varying patterns of locality depending on how loop iterations are assigned to processors and depending on the contents of the integer array nbrs. Integer nbrs can be viewed as a representation of a sparse matrix. We used a synthetic workload to generate a number of sparse matrices with differing dependency patterns. A square mesh in which each point was linked to four nearest neighbors was incrementally distorted. Random edges were introduced subject to the constraint that in the new mesh, each point still required information from four other mesh points.

Our workload generator makes the following assumptions:

1. The problem domain consists of a 2-dimensional mesh of points which are numbered using their natural ordering;
2. Each point is initially connected to its four nearest neighbors
3. Each link produced in the above step is examined, with probability Pr the link is replaced by a link to a randomly chosen point.

Once generated, this connectivity information is stored in integer array nbrs.

To obtain an experimental estimate of the efficiency of the executor on the Intel iPSC/2, we carried out a sequence of sparse matrix-vector multiplications using a 128*128 matrix generated from a square mesh using the workload generator described above.

```
do iter=1, num

 process-gather-data() - obtain needed yold values from
                         other processors,
 put in hashed cache

 Loop over local iterations k assigned to P

   do j = 0,m

       Perform calculation reading yold values or
       writing y values using local memory or
       hashed cache as is appropriate.

   end do

 End loop over local iterations

 Loop over local iterations k assigned to P


   Perform assignment reading y values or
   writing yold values using local memory or
   hashed cache as is appropriate.


 End loop over local iterations

 process-scatter-data() - scatter yold values from
          hashed cache to appropriate
                          processors

end do
```

Figure 3: Executor: Sparse Matrix Vector Multiply

We define

- p = Total Number of Processors
- $BlockSize = (128 \times 128)/p$
- $ArraySize = 128 \times 128$

We partitioned yold in the following ways:

1. partition the array in contiguous blocks of size *BlockSize*, i.e. *processor i* is assigned indices $i\times$ *BlockSize* through $(i+1)\times$ *BlockSize* -1

2. partition the indices in an interleaved fashion i.e. *processor i* is assigned indices $i, i+p, i+2p, \ldots, i+$ (*BlockSize* - *1*) $\times$p.

We first present the results obtained by partitioning yold in contiguous blocks. Table 1 depicts the time required to carry out the inspector and executor loops along with the optimal time. We define the *optimal* time as the sequential time divided by the number of processors. The inspector took a time roughly equal that required by one or two optimally parallelized iterations. Since the inner loops of most scientific codes consist of many repetitions of loops with invariant dependency patterns, this inspector overhead is not expected to be a serious performance bottleneck in many programs of practical interest.

We define parallel efficiency as $T_{parallel}/(T_{sequential} * P)$ where $T_{sequential}$ is the time taken by a sequential program to run on a single processor, P is the number of processors and $T_{parallel}$ is the time required to run the parallelized program on P processors. The parallel efficiencies were 0.76, 0.73, 0.67 and 0.56 for problems run on 4, 8, 16 and 32 processors respectively. We obtained relatively high efficiencies because most of the required data resided in the local memory of the processor. We also ran the sparse matrix vector multiply on 32 processors with probabilities Pr equal to 0.2 and 0.4 that an edge is randomly displaced. As we increased Pr, we encountered more non-local references. Efficiencies dropped from 0.56 to 0.30 when the probability was 0.2 and to 0.21 when the probability was 0.4.

We next present results obtained using an interleaved partition of yold. When an interleaved partition is employed, most of the data required by each processor is non-local. We need to fetch large amounts of data from the *hashed cache*. Moreover, there are also non-local write data accesses to yold. Thus we have to write the new values to the distributed memory. The parallel efficiencies obtained using an interleaved partition of yold are depicted in Table 2. As we increased the probability Pr, the number of edges displaced randomly increases. Due to the nature of the array partitioning, it is likely that the main effect of of increaseing Pr is to increase the number of processors from which data needs to be fetched. We see a decrease in efficiency as we increase the probability of randomly displacing an edge.

Single processor timings of an optimized version were compared with the parallel code run on one processor. The sequential code required $T_{sequential} = 64.4$ milliseconds and the one processor parallel code required $T_{parallel} = 81.4$ milliseconds. The overhead due to the executor

Table 1: Matrix-Vector Multiply - Blocked Partitioning

Processors	Inspector time (ms)	Executor time (ms)	Optimal time (ms)
4	11.8	21.3	16.1
8	7.3	11.1	8.05
16	4.8	6.0	4.03
32	4.3	3.6	2.01

Table 2: Matrix-Vector Multiply - Interleaved Partitioning

Processors	Pr = 0.0 efficiency	Pr = 0.2 efficiency	Pr = 0.4 efficiency
4	0.36	0.29	0.25
16	0.23	0.18	0.17
32	0.19	0.11	0.10

is approximately 26 percent. We expect substantially lower overheads on RISC based multiprocessors both because of the high prevalence of shift operations in hashed cache calculations and because of the potential for being able to concurrently schedule executor floating point and integer operations.

5 Conclusion

We propose a data migration mechanism that allows an explicit and controlled mapping of data to memory. While read or write copies of each data element can be assigned to any processor's memory, longer term storage of each data element is assigned to a specific location in the memory of a particular processor. Our proposed integration of a data migration scheme with a compiler is able to eliminate the migration of unneeded data that can occur in multiprocessor paging or caching. The overhead of adjudicating multiple concurrent writes to the same page or cache line is also eliminated. We present data that suggests that the scheme we suggest may be a practical method for efficiently supporting data migration.

References

[1] Mark Holliday. Page table management in local/remote architectures. Report CS-1988-2, Duke, 1988.

[2] C. Koelbel and P. Mehrotra. Supporting shared data structures on distributed memory architectures. Technical Report CSD-TR 915, Purdue University, West Lafayette, IN, October 1989.

[3] Kai Li. A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing, Penn State, University Park, Penn*, August 1988.

[4] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the Fifth Annual ACM Symposium on Distributed Computing, Calgary Alberta Canada*, August 1986.

[5] Janak H. Patel. Analysis of multiprocessors with private cache memories. *IEEE Transactions on Computers*, 31-4:296–304, 1982.

[6] J. Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines, to appear in journal parallel and distributed computing, april 1990. Report 89-7, ICASE, January 1989.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181987 ICASE Interim Report 9		

4. Title and Subtitle

A SCHEME FOR SUPPORTING DISTRIBUTED DATA STRUCTURES ON MULTICOMPUTERS

5. Report Date

January 1990

6. Performing Organization Code

7. Author(s)

Seema Hiranandani
Joel Saltz
Harry Berryman
Piyush Mehrotra

8. Performing Organization Report No.

ICASE Interim Report 9

10. Work Unit No.

505-90-21-01

9. Performing Organization Name and Address

Institute for Computer Applications in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665-5225

11. Contract or Grant No.

NAS1-18605

13. Type of Report and Period Covered

Contractor Report

12. Sponsoring Agency Name and Address

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665-5225

14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to 17th Annual Symposium on Computer Architecture

Final Report

16. Abstract

We propose a data migration mechanism that allows an explicit and controlled mapping of data to memory. While read or write copies of each data element can be assigned to any processor's memory, longer term storage of each data element is assigned to a specific location in the memory of a particular processor. Our proposed integration of a data migration scheme with a compiler is able to eliminate the migration of unneeded data that can occur in multiprocessor paging or caching. The overhead of adjudicating multiple concurrent writes to the same page or cache line is also eliminated. We present data that suggests that the scheme we suggest may be a practical method for efficiently supporting data migration.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Data Migration, cache, paging, inspector, executer	59 - Mathematical and Computer Sciences (General) Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	14	A03