NASA Contractor Report 182038
ICASE Report No. 90-33

ICASE

A SCHEME FOR SUPPORTING AUTOMATIC DATA MIGRATION ON MULTICOMPUTERS

Seema Mirchandaney
Joel Saltz
Piyush Mehrotra
Harry Berryman

Contract No. NAS1-18605
May 1990

To appear in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, South Carolina, April 1990

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, Virginia 23665-5225

Operated by the Universities Space Research Association

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

A SCHEME FOR SUPPORTING AUTOMATIC DATA MIGRATION ON MULTICOMPUTERS[1]

Seema Mirchandaney
Department of Computer Science
Yale University
New Haven, CT 06520

Joel Saltz
Piyush Mehrotra[2]
Harry Berryman
Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23665

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. We present data that suggests that the scheme may be a practical method for efficiently supporting data migration.

[1]Research supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18605 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23665.

[2]On leave from the Department of Computer Science, Purdue University, West Lafayette, IN 47907.

1 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. However, programming in this manner can be very time consuming, and can lead to programs that are difficult to debug. In this paper we describe a scheme which allows data to be automatically moved to the processor that needs it. In the remainder of this section, we provide an overview of our data migration scheme. Section 2 gives details of the hashed cache used in this scheme. The remainder of this paper describes the experiments that we have performed. Section 3 studies the overheads due to accessing the hashed cache. Section 4 outlines the results we have obtained using the model problem of sweeping over an unstructured mesh. These results were obtained with a synthetic workload along with an unstructured mesh used in aeronautical simulations. We finally present our conclusions in Section 5.

Overview of the Data Migration Scheme

In our scheme, data structures can be distributed across the set of processors. Such distribution of data structures across processors are supported in languages and systems such as [2, 3, 4, 8, 10, 11].

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. In a distributed memory machine, run-time procedures carry out the actual movement of data. Local copies of data are stored in a hash table. The hashing scheme for storing off-processor data in distributed machines will be called a *hashed cache*. There are many possible ways of organizing such a hash table, we experimentally examine a straightforward scheme for storing and retrieving off-processor data.

It is not realistic to expect that all loops can be parallelized or that it will be feasible to always map arrays so that computations taking place on a given processor use mostly local data. We must be able to handle the situation in which a single processor must access large numbers of off-processor array elements. This can happen either in a sequential code that executes on a single processor of a distributed machine or in a given processor's part of a parallelized loop computation. The hashed cache provides a mechanism for memory management on a distributed memory machine.

Coherency between the distributed data structures and the hashed caches must be maintained in software. It is particularly easy for compilers to generate code that guarantees coherency for (nested) parallel loops or sequential code. Although the hashed cache mechanism can be used in more general contexts, we will limit the discussion that follows to those two cases.

A parallel 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. Note that an inspector may have to be incremental subsets of the loop iterations when the amount of off-processor data to be stored exceeds the memory allocated to the hashed cache. The idea of splitting a parallel loop into an inspector and executor appeared in [5, 7, 9, 12]. In the context of sparse matrix computations, the idea was advanced in [1].

2 The Hashed Cache

We presume here that arrays have been distributed based on user annotations or directives. Once a distributed array is initialized, one 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 `O`.

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))`. At the end of each matrix vector multiplication in loop S5, the newly computed values `y(i)` are copied back into `yold(i)`.

Both arrays `y` and `yold` are distributed. When the loop is distributed, loop iterations may be assigned to processors in an arbitrary fashion. Consequently, long term storage of elements of `yold` may not be assigned to the processors that execute code referring to those elements. Due to the structure of the code, it is straightforward in this case to make sure that array `y` *is distributed so that each processor accesses only local elements of* `y`.

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.

2.1 The Inspector Phase

```
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

Figure 2 depicts the pseudo-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.

The location of a non-local distributed array element is determined by a hash function. Currently, we use a hashing function that, for a hash table of size 2^k, simply masks the lower k bits of the key. The key is formed by concatenating the processor-offset pair, (P, O), that corresponds to a distributed array reference. A linked list is used in case of collisions. To access the cache, the appropriate linked list must be traversed until the correct key is found. 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.
5. a pointer to the next data item that hashed to the same location.

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:

```
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 i

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 k

Perform  global communications to set up send and receive
pairs for the non-local data to be scattered and gathered
```

Figure 2: Inspector: Sparse Matrix Vector Multiply

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 in the 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 inserted in 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. As was the case with *process-global-read()*, this routine maintains a count variable containing the number of non-local elements to be scattered to P. It also maintains a list of the offsets of all the elements R to be scattered to P. At the completion of the

```
do iter=1, num

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

 Loop over local iterations i assigned to P

   do j = 0,m
       Perform calculation reading yold values or writing y
       values using hashed cache or local memory as is appropriate.
   end do

 End loop over local iterations i

 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 k

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

Figure 3: Hashed Cache Executor: Sparse Matrix Vector Multiply

inspector phase we precompute the communication pattern required to efficiently gather or scatter all the relevant non-local data referenced in the loop. This requires a global communication phase in which all processors participate. For a detailed description, see [5, 12].

2.2 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.

2.3 Enumerated Version of the Hashed Cache System

We have also implemented two modifications of the hashed cache system to allow us to quantify the cost of accessing the hashed cache and of transforming references to distributed array elements to locations in local storage. In these schemes, the non-local data is stored in the *hashed cache* as described in Section 2. Along with the hash table, we also have a list of pointers to all array elements referenced during the loop computation, on processor P. This enumeration list allows us to retrieve off-processor array elements during the executor phase without going through the hash table. Furthermore, locally stored elements of the distributed array can also be accessed without translating a global index to local offset in P's storage. We call this the *full enumeration* scheme. Note that the enumeration list can be easily generated by the inspector as it examines all the array references in the loop.

The fully enumerated version of the hashed cache can clearly require extremely large amounts of storage for storing its array of pointers, and is not a practical alternative for most applications. It is possible instead to employ an array of pointers that point only to hash table locations accessed. In this case, the number of pointers maintained is equal to the number of accesses to off-processor array references. This version is called the *partial enumeration* scheme.

3 Hash Table Overhead

We first present a set of simple experiments to illustrate the overheads associated with initializing and accessing the *hashed cache*. An array a is distributed between processors in blocks of size b. A loop executing on processor 0 (Figure 4) accesses off-processor data with varying strides. Recall from Section 2 that when multiple data elements hash to the same location we store the overflow in a linked list. By varying the stride of this simple

Table 1: Inspector and Executor timings as a function of the loop stride

loop stride	max links traversed	executor comm time (ms)	executor comp time (ms)	inspector comm time (ms)	inspector comp time (ms)	sequential time (ms)
1	0	30	82	10	49	20
2	1	31	93	10	62	20
4	3	19	112	11	86	20
8	7	15	154	12	130	20
16	15	14	238	15	217	20
32	31	17	381	21	391	20
31	0	18	82	22	49	20

loop, we are able to study the effect of link traversal through the *hashed cache* on the executor and inspector times. We fix the following parameters:

- hash table size = 4096
- numiter = 4096
- total non-local accesses = 4096
- b = 8192

Table 1 reports the experimental results we obtained by running the loop in Figure 4 with varying strides, all but one of which was a power of two. These experiments were carried out on a 32 node iPSC/2. With an odd stride, there were no hash table collisions and hence no links need to be traversed. This can be seen from the similar timings for strides 1 and 31. With power of two strides, we can have up to *stride* − 1 links in a hash bucket. The overhead of the executor increases as the possible number of links traversed for each hash table access increases. This is reflected in the increase of the executor

```
S0      if(my_processor .eq. 0)
S1       do j = b, numiter*stride + b, stride
S2         asum = asum + a(j)
         end do
```

Figure 4: Processor 0: Accessing non-local data

computation time as the number of links increases from 0 links to 31. The executor communication time, however, dropped by 50 %. This is because with a block size of b and stride 1 all the 4096 non-local elements accessed by the above loop on processor 0, reside on a single processor, processor 1. Thus processor 1 has to send all the 4096 array elements to processor 0. With stride 8, the 4096 elements accessed by processor 0 are distributed across four other processors. Thus, these four processors send their data concurrently to processor 0, reducing the overall executor communication time.

As the stride increases, the number of processors from which data is needed, increases. This is reflected in the inspector communication timing, which doubles as the the stride goes from 1 to 32. The stride used also has an effect on the inspector computation time. As noted above, an increase in the stride increases the maximum number of links in the hash table which is turn leads to an increase in the time required to access the hash table. Since the inspector computation time is directly dependent on the time required to access the hash table, we see a fairly rapid increase in this time with the increase in stride.

4 Unstructured Mesh Results

In this section we present the performance of the hashed cache system for the program depicted in Figure 1. This code exhibits greatly varying patterns of locality depending on how loop iterations are assigned to processors and the contents of the integer array nbrs. The integer array nbrs can be viewed as a representation of a sparse or unstructured mesh. We obtained these meshes in the following two ways :

- We used a synthetic workload to generate sparse matrices with differing dependency patterns.
- We used an unstructured mesh that was generated to carry out an aerodynamic simulation.

The next two sections describe the details of the synthetic workload and the unstructured mesh.

4.1 Synthetic Workload

The synthetic workload was defined in the following way. 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 row major or 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 q the link is replaced by a link to a randomly chosen point.
4. We use the rand() function available on Unix System V

Once generated, this connectivity information is stored in the integer array `nbrs`. We used this workload generator to obtain a set of matrices generated from 256*256 meshes. These matrices were used to perform a sequence of parallelized sparse matrix vector multiplications using our hashed cache data migration scheme for non-local array references to `yold`. We partitioned `yold` in various ways. First, let us define the terms:

- p = Total Number of Processors
- $MSize$ = matrix size = $n \times n$
- $BlockSize = MSize/p$

We partitioned `yold` as follows:

1. partition the array indices 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.

In the following sections, experiments involving matrices generated from 256 by 256 meshes in which `yold` is partitioned into *blocks* will be labeled by the descriptor *Blocked.* When `yold` is partitioned in an interleaved manner and the same sized mesh is used, the resulting experiments will be labeled by the descriptor *Interleaved.*

We now proceed to describe the details of the unstructured mesh.

4.2 Unstructured Mesh

We used an unstructured mesh that was generated to carry out an aerodynamic simulation involving a multielement airfoil in a landing configuration [6].

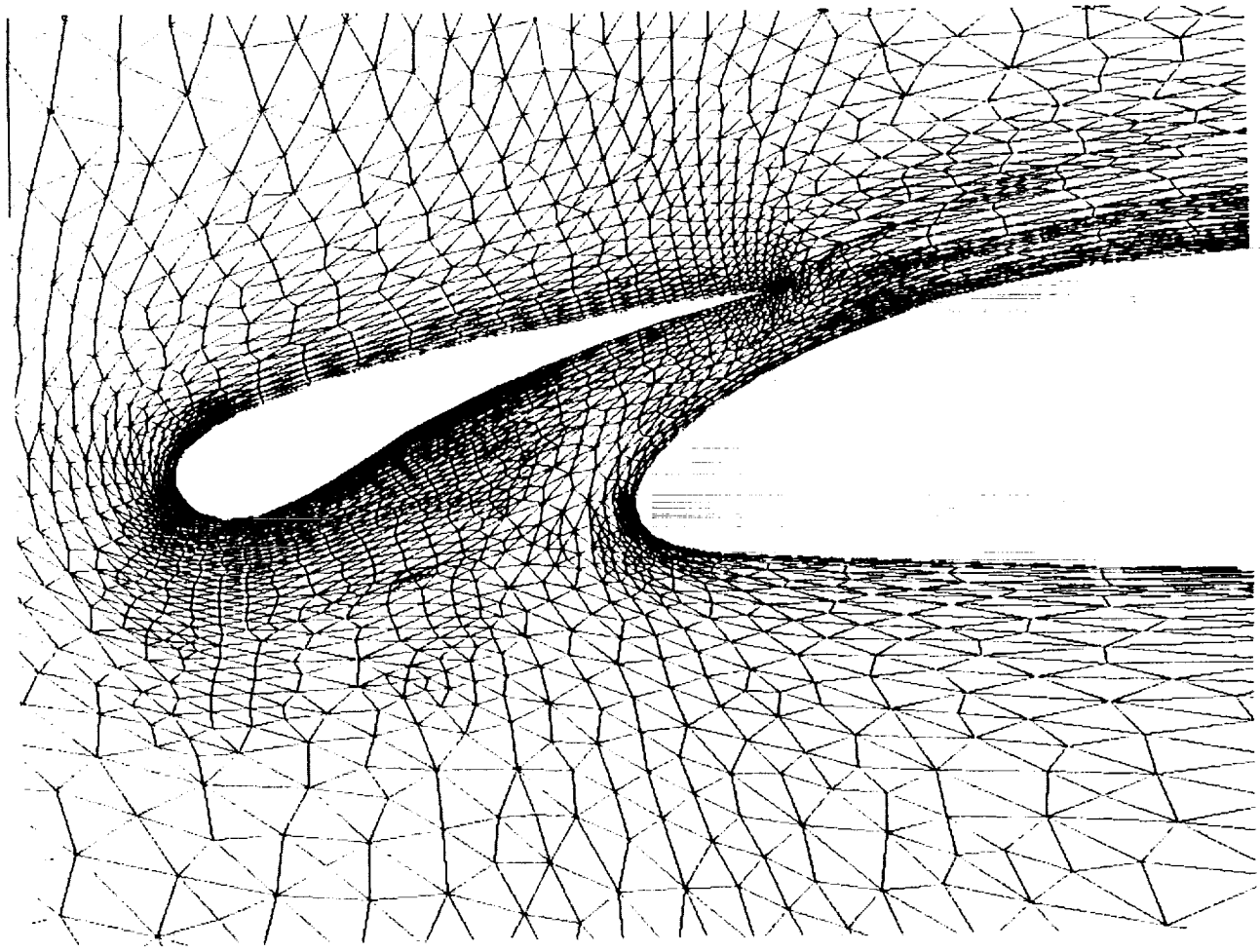

Figure 5: Unstructured Mesh for Multielement Airfoil

The unstructured mesh consists of a highly non-uniform scattering of mesh points joined together by line segments to form a set of triangular elements. The algorithm used is the Delaunay Triangulation algorithm [13]. The resulting mesh was then modified slightly in a postprocessing phase. Details of this mesh generation process can be found in [6]. An illustration of the mesh used is shown in Figure 5.

To obtain an experimental estimate of the efficiency of the executor and inspector on the Intel iPSC/2, we carried out a sequence of sparse matrix-vector multiplications using the unstructured mesh described above.

4.3 Performance of Off-Processor Data Access Mechanisms

In this section, we present data that gives an overview of some of the performance tradeoffs between different variants of the hashed cache data access mechanism. A major motivation for this set of experiments is to quantify the costs of accessing copies of off-processor data. There are clearly a tremendous number of different schemes that could be used to store and retrieve off-processor data, and we do not attempt to argue that the hashed cache method is in any sense optimal. In Section 2.3 we suggested building a list of pointers to locations in the hashed cache where off-processor data is to be cached; we called this mechanism *partial enumeration*. When computations are carried out, copies of needed off-processor data can be accessed by dereferencing pointers rather than by searching the hashed cache. We present in Table 2 parallel efficiencies obtained using the hashed cache and partial enumeration off-processor data access strategies. These parallel efficiencies only include the *executor* times, they do not include preprocessing overhead. As one might expect, the efficiencies obtained vary with the problem, but the use of partial enumeration led to efficiencies which ranged from roughly 3 to 19 % higher than those obtained from direct use of the hashed cache. The *number of parallel iterations per inspector* quantifies the required preprocessing overhead. Let T_{seq} represent the time required by an optimized sequential code on a single processor and P represent the number of processors. Then the *number of parallel iterations per inspector* is defined as the ratio between the time required to carry out a perfectly parallelized iteration (T_{seq}/P) and the time required for the inspector.

When we use only the hashed cache without enumeration, the number of parallel iterations per inspector varies from 1.08 to 5.10. The number of *executor* iterations required to carry out the inspector ranges from 0.66 to 0.99 when only the hashed cache is used, i.e., the inspector always takes less time to run than the executor. Since extra preprocessing is needed to assemble the enumeration list, it is to be expected that the number of parallel iterations per inspector is increased when partial enumeration is employed. The number of parallel iterations per inspector in this case ranges from 1.10 to 6.09.

In Section 2.3, we described another scheme called *full enumeration*, in which *all* memory accesses are enumerated. By enumerating all memory accesses, we can eliminate the costs incurred by

Table 2: Performances of Data Access Mechanisms

Problem	q	Hashed Cache		Partial Enumeration		Full Enumeration	
		effic	parallel iters / inspector	effic	parallel iters / inspector	effic	parallel iters / inspector
Blocked	0.0	0.61	1.08	0.63	1.10	0.80	1.36
Blocked	0.2	0.36	2.14	0.39	2.33	0.46	2.57
Blocked	0.4	0.30	2.80	0.35	3.00	0.39	3.27
Interleaved	0.0	0.23	4.30	0.27	4.44	0.30	4.60
Interleaved	0.2	0.21	4.57	0.25	5.20	0.27	5.67
Interleaved	0.4	0.19	5.10	0.23	6.09	0.25	6.04
Unstructured		0.43	1.81	0.51	1.90	0.63	1.98

1. determining whether a memory reference is to a locally stored array element and if it is locally stored

2. translating global array indices to locations in a processor's local storage

The storage costs incurred for this data access scheme are high enough to make the method impractical in many situations.

For the *Blocked* partitioning, with probability of deletion, $q = 0$, most array elements accessed are on the local processor. The hashed cache executor efficiency for this problem is 0.61. Use of partial enumeration leads to a small improvement in efficiency (0.63), full enumeration on the other hand leads to an efficiency of 0.80. For *Blocked* with q=0.4, approximately half of the array elements referenced are stored off-processor. The hashed cache executor efficiency in that case is 0.30, the use of partial enumeration increases this to 0.35 and full enumeration increases efficiency further to 0.39. The cost of accessing off-processor data in the hashed cache is clearly a much more important factor with $q = 0.4$ which has a larger number of off-processor references.

We also present in Table 2 the parallel efficiencies arising from the application derived unstructured mesh described in Section 4.2. This mesh was partitioned into 32 strips, and a strip was assigned to each processor. The strips were chosen so that the same number of floating point computations were required by each processor. The parallel efficiency obtained when we used only the hashed cache data structure was 0.43. Partial enumeration caused this efficiency to increase to 0.51 and full enumeration resulted in a further increase to an efficiency of 0.63. The number of parallel iterations per inspector varied from 1.81 when the hashed cache was used to 1.98 when full enumeration was employed.

In Table 3 we present the executor *computation* times for the hashed cache, partial enumeration and full enumeration data access methods. The executor computation times include the time required to:

1. copy off-processor data into hashed cache
2. distinguish between local and non-local references (for hashed cache and partial enumeration case)
3. obtain required locally stored data
4. access hashed cache (by searching hashed cache or by pointer dereference)
5. perform floating point computations
6. store off-processor writes to hashed cache
7. copy from hashed cache data needing to be scattered to other processors

. The executor computation time *does not* include the time required for carrying out interprocessor communication.

For a given sized mesh, all of the synthetic workload problems require the same number of operations. Consider first the *Blocked* partitioning generated by the synthetic workload generator. When the hashed cache was employed, as q increased from 0.0 to 0.4, the executor computation time increased 41 %. Both partial and full enumeration eliminate the need to search the hashed cache for each non-local memory reference. The increase in execution time as q went from 0 to 0.4 for the *Blocked* partitioning was consequently only 18 and 19 % for partial and full enumeration respectively

When `yold` is assigned in an interleaved manner, for all values of q most array references are non-local. Consider first the case in which the hashed cache is accessed directly. We note that for Interleaved, q equal to 0, we have a hashed cache executor computation time of 185 ms, while for q equal to 0.4 we have a time of 232 ms. This increase made sense when we noted that the number of *different* non-local off-processor references increased with q. Furthermore there is an increase with q in the number of hashed cache links that must be traversed in order to access copies of off-processor data (as we will show in Table 4). A more modest increase in execution time with q was noted in the partial enumeration and full enumeration cases; in these cases we expect only increases in the cost of copying elements to and from the hashed cache to play a role in the execution time increase. For the application derived unstructured mesh from Section 4.2, we measured executor computation times of 34 ms, 27 ms and 20 ms for hashed cache, partial enumerations and full enumerations respectively.

Further insight into the behavior of the hashed cache can be obtained from Table 4. This table categorizes array references in processor 0, it gives the percentage of references to locally stored data and to data accessed by traversing varying numbers of links in the hashed cache. For instance, for *Blocked* with q equal to 0, 97 % of array references were to local array elements and 3 % could be accessed by referring to the hashed cache without traversing any links. On the other hand, for *Interleaved*, only 3 % of array references were

Table 3: Executor Computation Times for the Off-Processor Storage Schemes Sparse MVM for 32 processors

Problem	q	hashed cache time (ms)	partial enumeration time (ms)	full enumeration time (ms)
Blocked	0.0	122	119	93
Blocked	0.2	148	131	101
Blocked	0.4	172	142	110
Interleaved	0.0	185	140	108
Interleaved	0.2	209	150	127
Interleaved	0.4	232	156	128
Unstructured		34	27	20

Table 4: MVM: Data Accesses by Processor 0 during the executor phase

Problem	q	% local accesses	% link 0	% link 1	% link 2	% link 3
Blocked	0.0	97	3	0	0	0
Blocked	0.2	79	18	3	0.25	0
Blocked	0.4	60	30	9	2	0.15
Interleaved	0.0	4	96	0	0	0
Interleaved	0.2	4	79	15	2	0.1
Interleaved	0.4	3	66	24	6	1

to locally stored data. 66 %, 24 % , 6 % and 1 % of references to copies of off-processor data required traversal of 0, 1, 2, and 3 links in the hashed cache respectively. In all cases, over two thirds of memory accesses did not require traversing even one hashed cache link. It should be noted that all table entries over 0.5 were rounded to the nearest integer so that row entries may not add to exactly 100.

5 Conclusion

In this paper, we have proposed a scheme for automatically migrating data across the processors of a distributed memory machine. The scheme uses a hash table for the easy access and modification of off-processor data. Using such a hashed cache allows us to support parallel loops accessing distributed data through a global name space.

We have investigated a set of model problems to characterize the performance of the hashed cache method. This model problem analysis employed a synthetic workload and an unstructured mesh that allowed us to perform a detailed examination of different rou-

tines required to implement the hashed cache scheme. The data presented here suggests that this scheme may be a practical method for efficiently supporting data migration on distributed machines.

References

[1] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley. An experimental study of methods for parallel preconditioned krylov methods. In *Proceedings of the 1988 Hypercube Multiprocessor Conference, Pasadena CA*, pages 1698,1711, January 1988.

[2] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for scientific programs on distributed memory machines. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, December 1989.

[3] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[4] A. Cheung and A. P. Reeves. The paragon multicomputer environment: A first implementation. Technical Report EE-CEG-89-9, Cornell University Computer Engineering Group, Cornell University School of Electrical Engineering, july 1989.

[5] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *2nd ACM SIGPLAN Symposium on Principles Practice of Parallel Programming*, pages 177–186. ACM SIGPLAN, March 1990.

[6] D. J. Mavriplis. Multigrid solution of the two-dimensional euler equations on unstructured triangular meshes. *AIAA Journal*, 26(7):824–831, July 1988.

[7] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. In *To appear in: Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, March 1989.

[8] P. Mehrotra and J. Van Rosendale. Parallel language constructs for tensor product computations on loosely coupled architectures. In *Proceedings Supercomputing '89*, pages 616–626, November 1989.

[9] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing , St. Malo France*, pages 140–152, July 1988.

[10] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Conference on Programming Language Design and Implementation*, pages 69–80. ACM SIGPLAN, June 1989.

[11] M. Rosing and R. Schnabel. An overview of Dino - a new language for numerical computation on distributed memory multiprocessors. Technical Report CU-CS-385-88, University of Colorado, Boulder, 1988.

[12] J. Saltz, K. Crowley, R. 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.

[13] N. P. Weatherill. The generation of unstructured grids using dirichlet tessalations. Report MAE 1715, Princeton, July 1985.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-182038 ICASE Report No. 90-33		

4. Title and Subtitle

A SCHEME FOR SUPPORTING AUTOMATIC DATA MIGRATION ON MULTICOMPUTERS

5. Report Date
May 1990

6. Performing Organization Code

7. Author(s)
Seema Mirchandaney
Joel Saltz
Piyush Mehrotra
Harry Berryman

8. Performing Organization Report No.
90-33

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

Final Report

To appear in Proceedings of the Fifth Distributed Memory Computing Conference, Charleston, SC, April 1990

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. We present data that suggests that the scheme may be a practical method for efficiently supporting data migration.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Distributed machines, data migration, cacheing	59 - Mathematical and Computer Sciences (General) 61 - Computer Programming and Software 62 - Computer Systems Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	18	A03

NASA FORM 1626 OCT 86