Index Translation Schemes for Adaptive Computations on Distributed Memory Multicomputers *

Bongki Moon Mustafa Uysal Joel Saltz

Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742
{bkmoon, uysal, saltz}@cs.umd.edu

Abstract

Current research in parallel programming is focused on closing the gap between globally indexed algorithms and the separate address spaces of processors on distributed memory multicomputers. A set of index translation schemes have been implemented as a part of CHAOS runtime support library, so that the library functions can be used for implementing a global index space across a collection of separate local index spaces. These schemes include two software-cached translation schemes aimed at adaptive irregular problems as well as a distributed translation table technique for statically irregular problems. To evaluate and demonstrate the efficiency of the software-cached translation schemes, experiments have been performed with an adaptively irregular loop kernel and a full-fledged 3D DSMC code from NASA Langley on the Intel Paragon and Cray T3D. This paper also discusses and analyzes the operational conditions under which each scheme can produce optimal performance.

1 Introduction

Distributed memory multicomputers have been widely used to solve many structured and unstructured problems. Most of the performance gain from the distributed memory multicomputers can be obtained by data distribution and load balancing. In such multicomputers, each processor owns a separate local memory and is connected to an interconnection network. Processors communicate with each other by sending and receiving messages across the network. However, since most of the applications are written assuming a single global index space, the programming task on distributed memory multicomputers often requires a substantial amount of effort. Thus, current research in parallel programming is focused on closing the gap between globally indexed algorithms independent of the underlying distribution of data and the separate address spaces of processors.

Compilers for various languages such as Fortran [8] and C++ [4] have been developed to give the illusion of a shared address space on distributed memory multicomputers. For structured problems, such compilers as Fortran D [8] use distribution directives to partition computation across processors. Using the directives, the compilers can statically determine the processor that owns a data item and the processor that requires the value of the data item. The compilers can then generate message passing calls to directly pass this value from the owner processor to the processor that needs it. Another

*This work was supported by NASA under contract No. NAG-11560, by ONR under contract No. SC 292-1-22913 and by ARPA under contract No. NAG-11485. The authors assume all responsibility for the contents of the paper.

```
L1: do n = 1, n_steps
L2:     do i = 1, n_edges
            y(ia(i)) = 0.85 * x(ia(i)) + 0.42 * x(ib(i))
            y(ib(i)) = 0.88 * x(ia(i)) + 0.44 * x(ib(i))
        enddo
L3:     do i = 1, n_grids
            x(i) = y(i)
        enddo
    enddo
```

Figure 1: An example code segment of an irregular loop

approach called Distributed Shared Memory (DSM) enables an application's user-level code to support shared memory and message passing efficiently [12]. Distributed shared memory is typically supported by the processor's address translation hardware.

This paper describes and evaluates a set of index translation schemes for implementing a global index space across a collection of distributed memories. These schemes can be incorporated into a runtime support library so that calls to the library functions can be invoked by manually parallelized programs or can be generated by compilers [14]. These schemes can also be incorporated into distributed shared memory systems such as the one used in the Wisconsin Wind Tunnel project to support user-level shared memory [12].

To illustrate the need of runtime support for address translation, consider the Jacobi iterative method for solving a partial differential equation on an irregular numerical grid, which arises in molecular dynamics codes and sparse linear solvers. A typical example loop of such an irregular computation is presented in Figure 1. The update of each grid point depends only on the values at neighboring grid points from the previous iteration. Since the grid structure of such irregular problems is determined only at runtime, compilers cannot fully analyze and translate globally indexed memory accesses. For instance, in Figure 1, the data access patterns to the arrays `x()` and `y()` are determined at runtime via the indirection arrays `ia()` and `ib()`. Thus communication patterns between processors should be determined at runtime and accordingly the globally indexed data accesses should also be translated at runtime.

The rest of the paper is organized as follows. Section 2 describes a distributed translation scheme which allows us to map a globally indexed distributed array and how this scheme can be used in parallel computation. Section 3 introduces new adaptive translation schemes which offer reduced overhead of index translation by using software caching techniques. Experimental results performed with an irregular loop kernel and a direct particle simulation application are presented in Section 4. We compare the performance of the software-cached translation schemes and discuss the operational condition under which each scheme can produce optimal performance in Section 5.

2 Distributed Translation Table

This section describes a *distributed translation table* which allows us to map a globally indexed distributed array onto processors in an arbitrary fashion. Briefly outlined is how the distributed translation table can be used in the preprocessing stage of the *inspector/executor* model of parallelization [15, 7].

On distributed memory machines, large data arrays may not fit in a single-processor's memory, hence they are divided among processors. Also computational work is divided among individual processors to

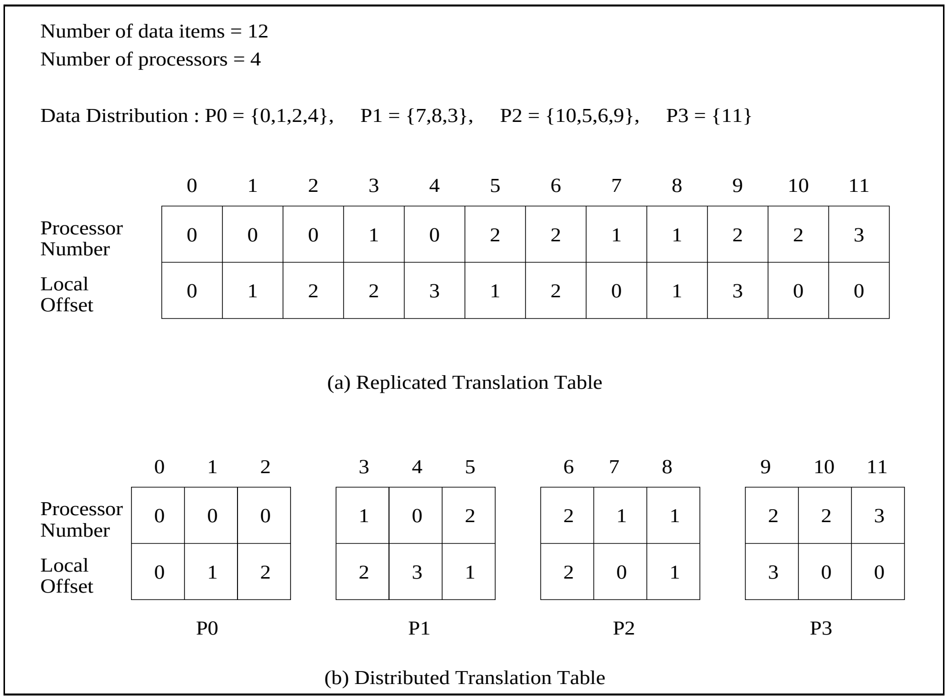

Figure 2: Examples of Translation Tables

achieve parallelism. Once distributed arrays have been partitioned, each processor ends up with a set of globally indexed distributed array elements. Each element in a distributed array A of size N is assigned to a particular home processor. In order for any processor to be able to access a given element, $A(i)$, of the distributed array, the home processor and local address of $A(i)$ must be determined.

Generally, unstructured problems solved with irregular data distributions perform more efficiently than with regular data distributions such as BLOCK. In the case of irregular data distribution, a *translation table* is built that, for each array element, lists the home processor and the local offset. If the data is distributed in a BLOCK or CYCLIC manner, the translation table can be simulated with an analytic function. Otherwise, a full-fledged translation table needs to be built. This translation table is used for *dereferencing*, the process of finding the processor home of a global element and the local offset within the processor. Figure 2(a) illustrates a replicated translation table for a given data distribution of 12 globally indexed data items over 4 processors. For instance, a data item with a global index `3` is stored in the third memory location within the processor P_1. Thus, its home processor (`1`) and local offset (`2`) are stored in the fourth entry of the translation table.

Memory considerations make it clear that it is not always feasible to replicate a copy of the translation table on each processor, so the translation table must be distributed across processors. This is accomplished by distributing the translation table by blocks, i.e., putting the first N/P elements on the first processor, the second N/P elements on the second processor, and so on, where P is the number of processors and N is the number of globally indexed data items. Figure 2(b) illustrates a distributed translation table obtained by partitioning the replicated translation table given in Figure 2(a).

When an element $A(i)$ of a distributed array A is accessed, the home processor and local offset are found in the portion of the distributed translation table stored in processor $\lfloor \frac{i \times P}{N} \rfloor$. A dereferencing operation using the distributed translation table requires communication between processors to exchange the information stored in each processor's portion of the distributed translation table.

Figure 3 presents an example of the Jacobi iterative loop parallelized with the CHAOS runtime library [13]. Each processor passes the procedure `build_translation_table` a list of global indices of

```
I1: ttable = build_translation_table(index,n_local_grids)
I2: call dereference(ttable,ia,offseta,proca,n_local_edges)
    call dereference(ttable,ib,offsetb,procb,n_local_edges)
I3: call CHAOS functions to generate communication schedule
L1: do n = 1, n_steps
E1: call CHAOS functions to gather off-processor data elements
L2:    do i = 1, n_local_edges
          y(ia_local(i)) = 0.85 * x(ia_local(i)) + 0.42 * x(ib_local(i))
          y(ib_local(i)) = 0.88 * x(ia_local(i)) + 0.44 * x(ib_local(i))
       enddo
E2: call CHAOS functions to scatter off-processor data elements
L3:    do i = 1, n_local_grids
          x(i) = y(i)
       enddo
    enddo
```

Figure 3: An irregular loop parallelized by CHAOS runtime library

```
L1: do n = 1, n_steps
L2:    do i = 1, n_edges
          y(ia(i)) = 0.85 * x(ia(i)) + 0.42 * x(ib(i))
          y(ib(i)) = 0.88 * x(ia(i)) + 0.44 * x(ib(i))
       enddo
L3:    do i = 1, n_grids
          x(i) = y(i)
       enddo
S:     if (mesh redefined) then regenerate ia() and ib()
    enddo
```

Figure 4: An example code segment of an adaptive irregular loop

array elements for which it will be responsible. To create a translation table, for example, in the first inspector step `I1` of Figure 3, each processor passes an array `index(1:n_local_grids)` to the runtime function. The array `index(1:n_local_grids)` stores a set of global indices owned by the processor which is determined by the current distribution of data. If a given processor needs to access a data item that corresponds to a particular global index i for a specific distributed array, the processor can consult the distributed translation table to find the owner processor and location of that item within the local memory of the owner processor. The next inspector step `I2` carries out the dereferencing operation. Though this step inherently incurs communication overhead, the cost of dereferencing will be amortized over loop iterations as long as the indirection arrays `ia()` and `ib()` are not changed.

3 Software-Cached Adaptive Translation Schemes

In static irregular problems such as sparse linear systems and unstructured mesh codes, data access patterns are determined via a level of indirection and the access patterns remain static. Thus, the dereferenced data accesses of globally indexed data items may be reused over loop iterations by storing the index translation information in local memory (for example, the arrays `offseta()` and `proca()` returned from the `dereference()` function in Figure 3). In adaptive irregular applications which can be found in direct particle simulation and molecular dynamics simulation, however, the data access patterns

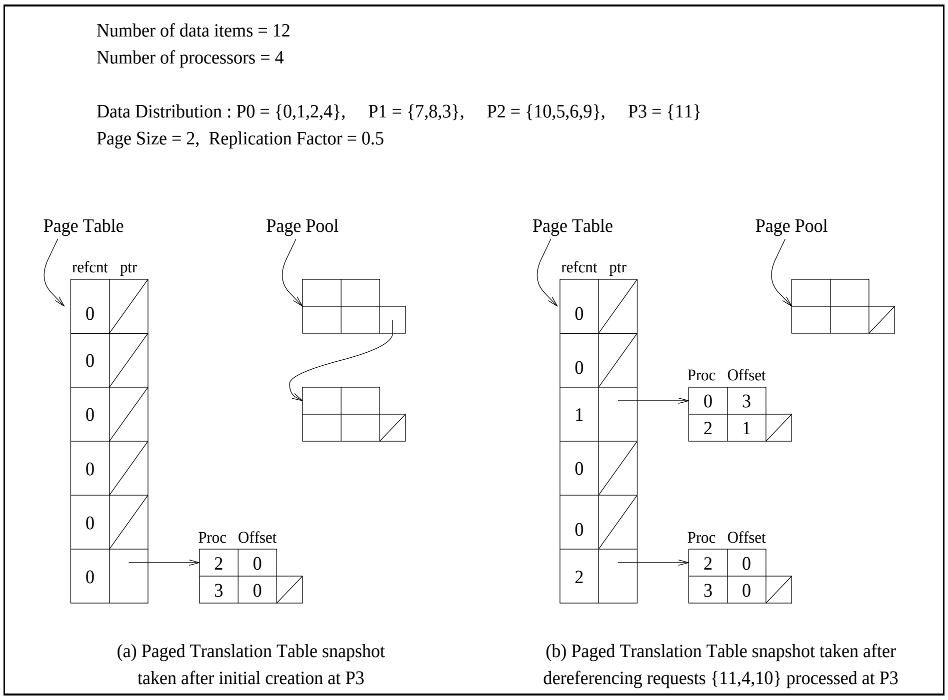

Figure 5: Paged Translation Tables

may change during the processing of loop iterations. Figure 4 shows the computational structure of such adaptive irregular applications. The data access pattern in loop `L2` changes whenever the indirection arrays `ia()` and `ib()` are regenerated in the conditional statement `S`. Then, since the index translation information stored in local memory can not be reused, the globally indexed data items should be dereferenced whenever the access patterns change.

In adaptive applications such as DSMC [3] and CHARMM [5], data access patterns change frequently and irregular data distribution is preferred for better performance over regular data distribution. Thus, minimization of the dereferencing cost is crucial for efficient processing of such applications on distributed memory multicomputers. In such cases, the distributed translation table described in Section 2 tends to be too costly to use. There are three main reasons. First, the dereferencing operation inherently requires communication between processors to exchange the translation information. Second, the distribution of the translation table across processors is fixed and bears no particular relationship to the distribution of dereferencing requests. Third, even though a nonlocal global index is dereferenced in several loop instances, the translation information obtained in the previous loop instance can not be reused in the subsequent loop instances unless it is stored explicitly in local memory.

In many cases there is enough memory to partially replicate the translation table. The distributed translation table is not able to replicate portions of the translation table in order to trade memory for improved performance. This section introduces two variations of the distributed translation table which offer reduced overhead by using extra memory: *paged translation table* and *hashed translation table*. These translation schemes use software caching techniques so that the extra memory can be exploited adaptively for changeable data access patterns and communication latency can be avoided.

3.1 Paged Translation Table

The *paged translation table* is composed of a *page table* and a set of *page frames*. Followed here is the convention found in the virtual memory literature where the memory location associated with each page is called a page frame. The process of generating the paged translation table is governed by two adjustable parameters, a page size $\mathcal{S}$ and a replication factor $\mathcal{R}$. The replication factor $\mathcal{R}$ is defined as the fraction of the maximum number of pages for which extra frames are allocated by each processor. In this scheme, the translation table is decomposed into fixed-sized pages. Each page lists the home processors and offsets associated with a set of $\mathcal{S}$ contiguously numbered global indices. Suppose N is the size of a distributed array. Then, each processor maintains a page table which has $N/\mathcal{S}$ entries, and stores up to $\frac{N\times\mathcal{R}}{\mathcal{S}}$ pages. Each page table entry contains a *reference counter* and a *page pointer* which points to a page for $\mathcal{S}$ consecutive global indices if the page pointer is not null.

Translation table information for each index must be stored somewhere. In this current paged translation table implementation, a distributed translation table is built up as a back-end data structure to make it simpler to dereference global indices which are not currently available in local memory. When a paged translation table is initially created, each processor stores only the pages which include dereferencing information of global indices assigned to the processor. Specifically, if a processor owns a global index i, the $\lfloor\frac{i}{\mathcal{S}}\rfloor$-th entry of the processor's page table points to a page containing the home processors and offsets for global indices $\lfloor\frac{i}{\mathcal{S}}\rfloor\times\mathcal{S}, \lfloor\frac{i}{\mathcal{S}}\rfloor\times\mathcal{S}+1, \ldots, \lfloor\frac{i}{\mathcal{S}}\rfloor\times\mathcal{S}+\mathcal{S}-1$.

Figure 5(a) depicts an initial paged translation table for a given data distribution which is identical to the one shown in Figure 2. Since $\mathcal{S}=2$ and $\mathcal{R}=0.5$, each processor allocates a page table with $\frac{N}{\mathcal{S}}=6$ entries, and a page pool with $\frac{N\times\mathcal{R}}{\mathcal{S}}=3$ available page frames. On processor P_3, for example, there is only one global index `11` owned by it. Thus, the processor P_3 fetches a page frame from the page pool, fills the page frame with home processors and offsets, and attaches it to the sixth ($\lfloor\frac{i}{\mathcal{S}}\rfloor=5$) entry of its page table.

When a processor receives a dereferencing request $\{j\}$, it looks at its $\lfloor\frac{j}{\mathcal{S}}\rfloor$-th page table entry. If the pointer field of the page table entry points to a valid page frame, then the home processor and offset of the global index j can be fetched from the ($j \bmod \mathcal{S}$)-th record in the page frame. The reference counter of the page table entry is incremented. If the pointer field of the $\lfloor\frac{j}{\mathcal{S}}\rfloor$-th page table entry is null, then a page fault occurs. When a page fault occurs, the distributed translation table is referenced to translate the global index which caused the page fault. Then, a page frame is fetched from the page pool and the home processor and offset of the global index is stored in it. Figure 5(b) shows a paged translation table snapshot taken after a set of dereferencing requests $\{11,4,10\}$ is processed.

In many cases where a replication factor is chosen to be less than one, page faults may occur while no unused page frames are available in the page pool. There are basically two options in handling the situation: the information of home processors and offsets obtained by referencing the distributed translation table may be returned without being stored in the paged translation table, or a page may be evicted to make room for an incoming one. The latter was chosen, since it adapts to the variation of data access patterns. A replacement policy governs the choice of the victim when eviction of pages is in order. Since implementation of the well-known page replacement algorithm *LRU (Least-Recently-Used)* imposes too much overhead to be handled by software alone, implemented here is the *NRU (Not-Recently-Used)* page replacement algorithm, one of the approximations of LRU, using the reference counters in the page table [9].

3.2 Hashed Translation Table

The structure of a *hashed translation table* differs from that of a paged translation table in that a hashed translation table consists of a *hash table* and a set of *hash nodes* instead of a page table and a set of page frames. There is only one operational parameter, replication factor $\mathcal{R}$; it is defined as the fraction of the maximum number of indices for which hash nodes are allocated by each processor. Each hash node includes a global index, its home processor and offset as well as a pointer field. Suppose N is the number of globally indexed data items. Then, each processor stores up to $N\times\mathcal{R}$ hash nodes. In this

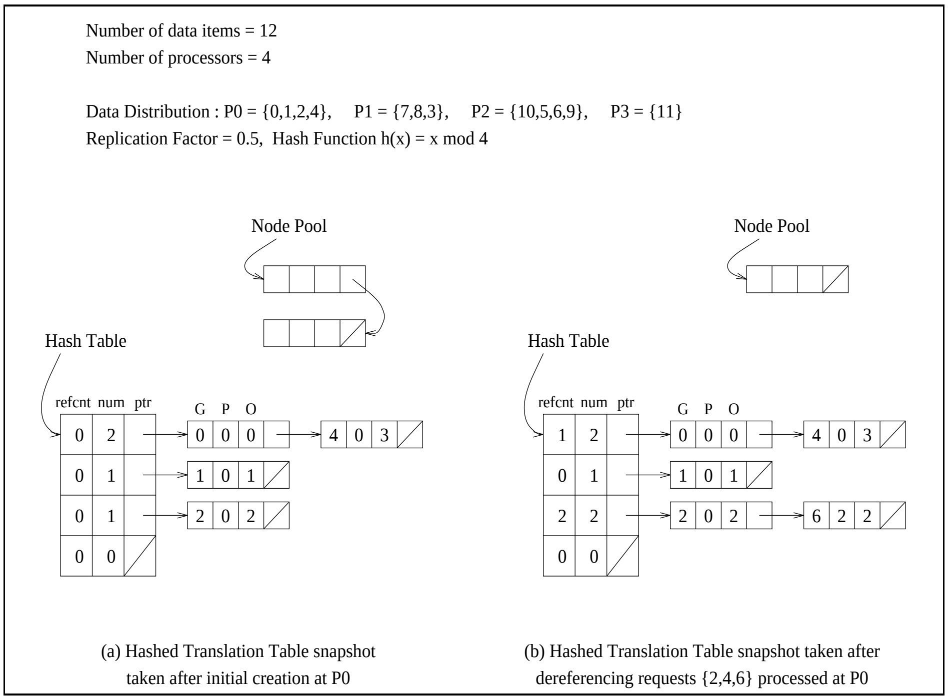

Figure 6: Hashed Translation Tables

current implementation of hashed translation table, the size of hash table $\mathcal{H}$ is determined in such a way that $\mathcal{H} = 2^k$ where k is the smallest number such that $2^k \geq \lceil N/P \rceil$. Each entry of the hash table includes a *reference counter*, a *node counter* and a *node pointer*. The reference counter field is used by the NRU replacement algorithm, and the node counter and node pointer fields are used to maintain a list of nodes hashed into the same entry. Since the hash table is of size 2^k, a hash function $h(x)$ is currently used which simply masks the lower k bits of the global index x. Since a linked list is used to store colliding indices, the simple choice of hash function has the potential to incur the high overhead of traversing a long list in discovering the correct index. This issue shall be further addressed in Section 5.

As with the paged translation table, a distributed translation table is built up as a back-end data structure. When a hashed translation table is initially created, each processor stores only the translation information for globally indexed data items which the processor owns. Specifically, if a processor owns a global index i, the processor adds a hash node to the $h(i)$-th entry of its hash table.

Figure 6(a) depicts an initial hashed translation table for the same data distribution given in Figure 2 and Figure 5. Since the number of processors P is 4 and replication factor $\mathcal{R}$ is 0.5, each processor creates a hash table of size 4 and a node pool with 6 unused hash nodes. Then, a list of (global index, processor, offset) triplets for locally owned data items is stored in the hashed translation table. For instance, the processor P_0 fetches four hash nodes from the node pool, fills each hash node with a global index, its home processor and offset, and attaches it to the proper hash table entry. This step does not require extra communication between processors because the information about the home processors and offsets of locally owned data items can be obtained on the fly.

Dereferencing a global index with a hashed translation table is to essentially search through a hash table. When a dereferencing request $\{j\}$ arrives, the $h(j)$-th hash table entry is looked up and the linked list attached to the hash table entry is traversed to find a hash node having a matching key value (i.e., a global index). If a matching hash node is found, then the home processor and offset of the node is returned, and the reference counter of the hash table entry is incremented by one. Otherwise, the distributed translation table must be referenced to translate the global index. This requires information

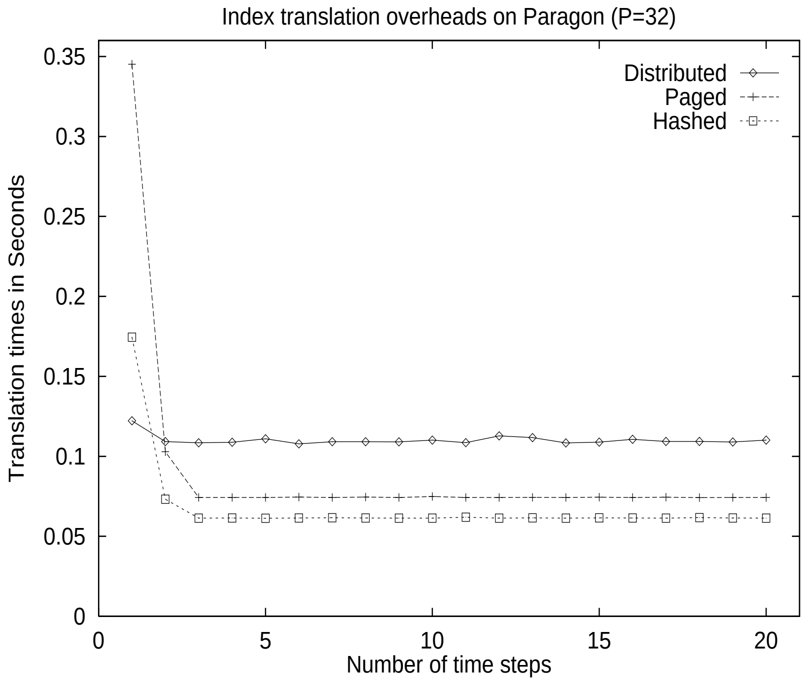

Figure 7: Index translation times with $\mathcal{R}$=0.3, $\mathcal{S}$=8 on the 32-node Intel Paragon

exchange between processors. Since the index translation information obtained by referencing the distributed translation table is not in the hashed translation table, an unused hash node is fetched from the node pool and is used to store the translation information. If there are no unused hash nodes available in the node pool, the NRU replacement algorithm is applied using the reference counters in the hash table.

4 Comparison of Experimental Results

This section describes the experiments performed and discusses the results. A number of different experiments were carried out using (1) an irregular loop kernel which has a similar structure to an irregular Jacobi iteration, and (2) a real world application, 3-dimensional Direct Simulation Monte Carlo (DSMC) code. The remainder of this section presents and compares results from distributed translation table, paged translation table and hashed translation table with various operational parameters.

4.1 Experiments with an irregular loop kernel

The sample adaptive loop described in Figure 4 was run with an irregular mesh with 100,000 grid points. To emphasize the effect of index translation operation, the assumption that the structure of the mesh is redefined every time step by the statement `S` in Figure 4 was used. Thus, the CHAOS function `dereference`() must be invoked every time step to run the code on distributed memory multicomputers. The same experiments have been performed on the Caltech CCSF Paragon with 512 processing nodes and the Jet Propulsion Laboratory (JPL) Cray T3D with 256 processing nodes.

Figure 7 and Figure 8 compare the costs of three index translation schemes (i.e., the processing costs of `dereference()` function invocations) at each of 20 time steps on the 32-node Paragon and on the 32-node T3D, respectively. The same replication factor ($\mathcal{R}$=0.3) has been used for both the paged and hashed translation schemes, and the same page size ($\mathcal{S}$=8) for the paged translation scheme.

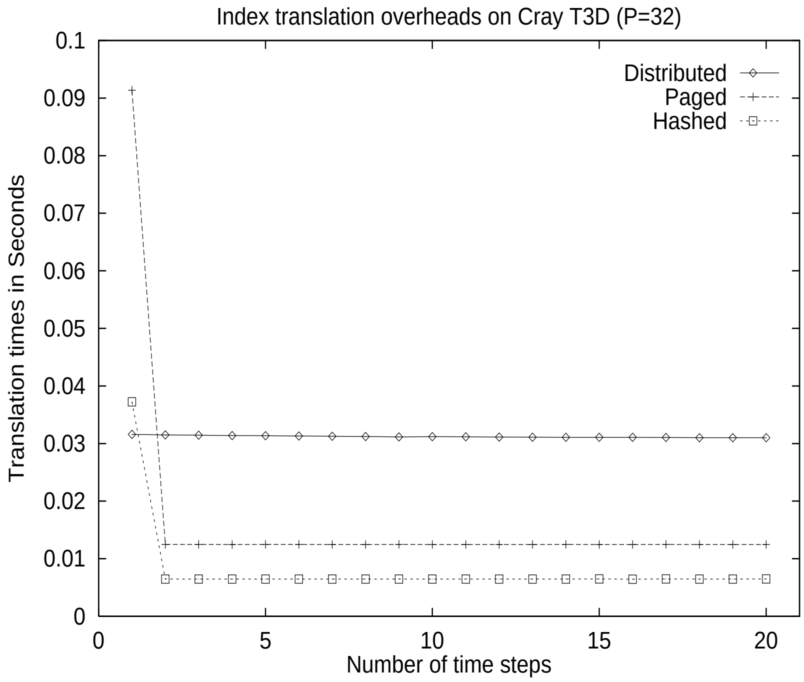

Figure 8: Index translation times with $\mathcal{R}$=0.3, $\mathcal{S}$=8 on the 32-node Cray T3D

Though the actual costs of the three index translation schemes differ by an order of magnitude on different machines, the index translation schemes show common characteristics. While the performance of the distributed translation scheme is almost invariant during all the time steps, the costs of the other schemes are much higher in the initial time steps and lower in the remaining time steps than that of the distributed translation scheme. This is due to the fact that a number of nonlocal global indices are translated and cached into the paged or hashed translation table in the initial time steps, and most of the global indices are translated locally in the subsequent time steps. It is also observed that the cost of the paged translation scheme is much higher than that of the hashed translation scheme in the initial time steps. This is due to the coarse-grained memory management of the paged translation scheme. In other words, the paged translation scheme needs to translate and cache all the indices in the page frames that should be brought in local memory. This property increases the number of dereferencing requests beyond the required number of indices.

The effect of page size on the performance of the paged translation scheme is shown in Figure 9 and Figure 10. The replication factor was 0.05 in the experiments performed on the 512-node Paragon, and it was 0.10 in the experiments performed on the 128-node T3D. The results shown in both of the figures indicate that the performance of the paged translation scheme is very sensitive to the choice of page size. The paged translation scheme with relatively small page sizes significantly outperformed the distributed translation scheme. However, when larger page sizes were chosen, the performance of the paged translation scheme became even worse than that of the distributed translation scheme. Such performance degradation is mainly due to page thrashing. It is more likely the page thrashing happens with page frames of larger size because the larger the page size the higher the ratio of page faults.

4.2 Experiments with a direct particle simulation

This section presents experiments which were carried out with a 3D Corner Flow Direct Simulation Monte Carlo (DSMC) code from NASA Langley. DSMC is a well-established technique for modelling rarefied gas dynamics via direct particle simulation on a grid [3]. It has been widely used in aerospace

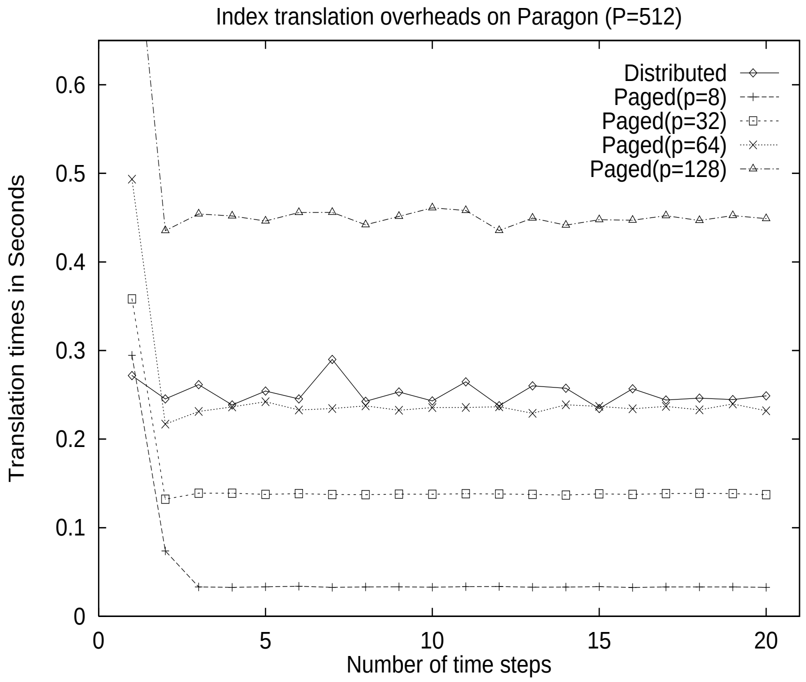

Figure 9: Index translation times with varying page sizes on the 512-node Intel Paragon ($\mathcal{R}$=0.05)

Table 1: Effects of replication factor in DSMC computation
(the numbers in the parentheses are the percentage of the translation overhead)

Times in seconds	Translation Schemes				
	Distributed	Paged		Hashed	Hashed/T3D
Rep. Factor		pagesz=16	pagesz=256		
0.025	11.3 (33)	17.5 (40)	22.9 (47)	17.0 (40)	37.1 (58)
0.050		9.5 (27)	19.5 (43)	9.8 (27)	10.1 (28)
0.075		8.9 (25)	13.1 (34)	8.3 (25)	6.3 (19)
0.100		8.2 (24)	10.7 (29)	7.1 (21)	3.4 (11)
0.125		7.5 (22)	10.0 (28)	6.4 (20)	2.7 (09)
0.150		6.7 (20)	9.3 (26)	6.0 (19)	2.4 (08)
0.175		6.5 (20)	8.5 (25)	5.9 (18)	2.4 (08)
0.200		6.6 (20)	7.8 (23)	6.1 (18)	2.4 (07)

applications such as upper-atmosphere flows for hypersonic cruise vehicles and rocket plumes, and in vacuum-related technologies for the semiconductor industry modelling plasma etching or chemical vapor deposition [2].

The DSMC method includes movement and collision handling of simulated particles on a spatial flow field domain overlaid by a Cartesian mesh. The spatial location of each particle is associated with a Cartesian mesh cell. The key concept of the DSMC method is that particle movement is decoupled from particle collisions. That is, the computation of a time step can be split into the calculation of physical quantities of collided particles and the relocation of moved particles. Furthermore, since the computations associated with performing probabilistic chemistry and collisions can be distributed across processors cell by cell, the DSMC method in principle is a good match for parallel processing on distributed memory multicomputers [16, 10].

Changes in position coordinates may cause the particles to move across cell boundaries. In the

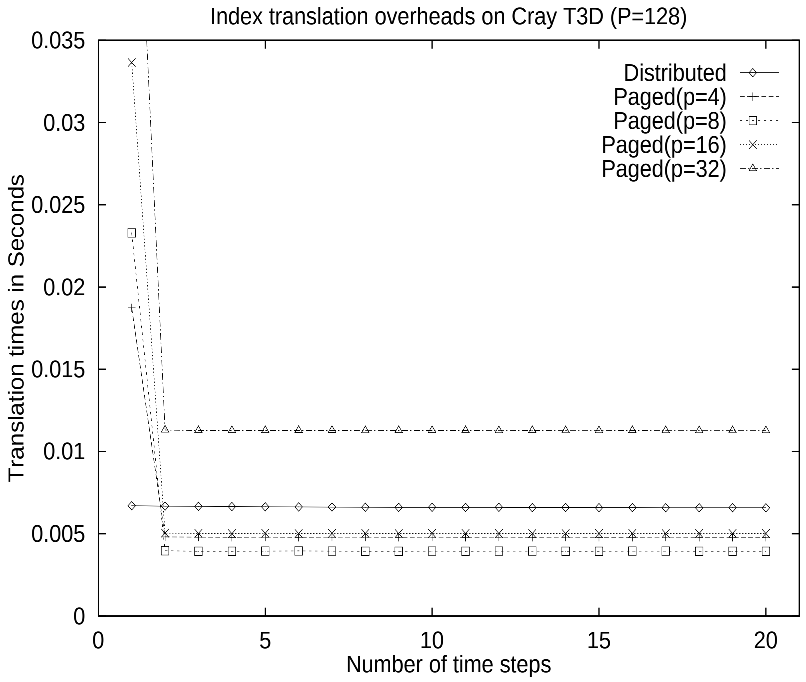

Figure 10: Index translation times with varying page sizes on the 128-node Cray T3D ($\mathcal{R}$=0.10)

particular corner flow DSMC code presented here, about 30 percent of the particles change their cell locations every time step. However, particle movements are local enough that particles only move between neighboring cells. The relocation of particles must be done every time step to move them to their new cells. Thus, the index translation must also be done every time step to find the particles' new owner processors. The corner flow DSMC code simulates a 3-dimensional flow field with 77,760 cells and about 600,000 particles.

Figure 11 shows index translation costs of three translation schemes measured at each time step on the 53-node Paragon. The experiments discussed here focused on the first 80 time steps of transient phase. During the transient time steps, the number of particles keeps increasing because the number of entering particles is greater than that of leaving particles. This explains the fact that the cost of the distributed translation scheme increases as the computation proceeds.

Another key point of the experiments is that the problem domain (that is, cells) of the DSMC code is repartitioned across processors periodically to balance the work load. In these particular experiments, the domain was repartitioned every 20 time steps. If the problem domain is repartitioned, a translation table must be regenerated and the cached information of nonlocal global indices must be invalidated. Thus, the costs of paged translation and hashed translation schemes are far higher in the time steps after domain repartitioning because a number of nonlocal global indices are translated and cached into the paged or hashed translation table.

Table 1 shows the performance of the translation schemes with varying replication factors. The numbers in the parentheses represent the ratio of the translation time to the total elapsed time. The experiments shown in this table were carried out with the same corner flow DSMC code simulating 9,720 cells and about 50,000 particles. The performance numbers were measured in seconds for the first 200 time steps on the 32-node Paragon except the last column which was obtained using the hashed translation scheme on the 32-node T3D. The replication factor has a significant effect on the performance of the paged and hashed translation schemes. However, when the replication factor becomes large enough to avoid frequent page or node replacements, the performance is almost invariant with

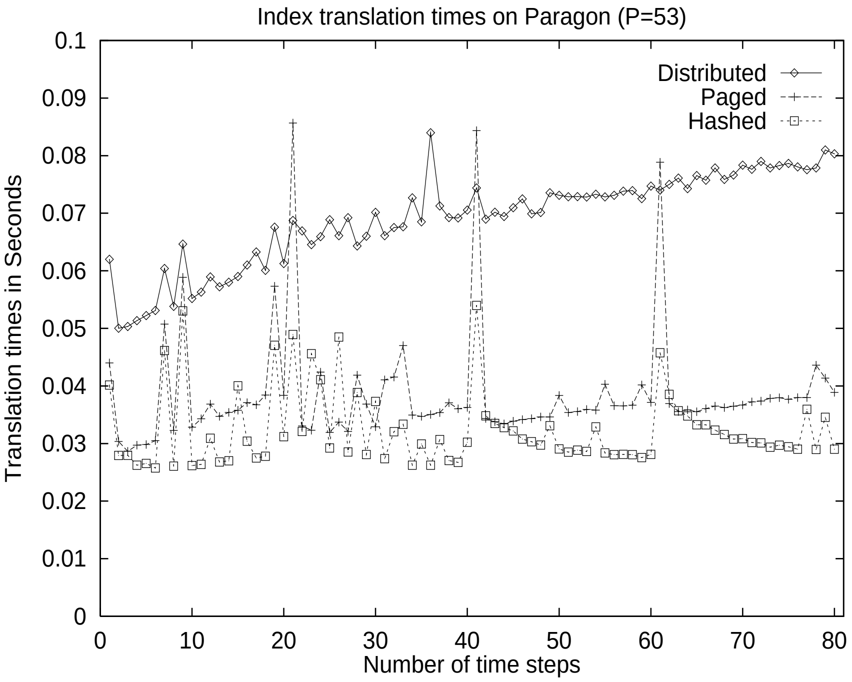

Figure 11: Index translation times of the 3D DSMC code ($\mathcal{R} = 0.2$ and $\mathcal{S} = 16$)

respect to varying replication factors. It is also observed that large page frames use up replicated memory fast and may cause severe performance degradation due to page thrashing.

5 Discussion

Through the experimental results presented in this paper, it has been demonstrated that both the paged and hashed translation schemes significantly outperform the distributed translation scheme. When comparing the results from the paged and hashed translation schemes, the hashed translation scheme slightly outperformed the paged translation scheme in most of the cases. This is due mainly to the difference in the granularity of the replicated memory management. That is, the finer-grained memory management of the hashed translation table adapts better to the highly random access patterns encountered in both experiments with the irregular loop kernel and with the NASA Langley DSMC code.

However, it is anticipated that the paged translation scheme will outperform the hashed translation scheme in other applications where the access patterns change slowly and bear high locality. In dereferencing a global index, if the global index has already been cached into the local memory, the paged translation scheme guarantees a constant translation cost. On the other hand, the hashed translation table may suffer from skew built in the hash table. That is, if a particular choice of a hash function generates long lists of collided hash nodes, then the overhead of traversing a long list of hash nodes in discovering the key index may be high. Consequently, to stabilize the performance of the hashed translation scheme, it is necessary to choose a good hash function which does not entail such a hash skew.

When the hash skew hurts the performance of the hashed translation scheme, one of a collection of randomly generated hash functions can be selected to ensure a good performance. This is done by simulating the use of the individual functions with the globally indexed data items owned by each processor. For this purpose, the current implementation of the hashed translation scheme allows the option of choosing a hash function from a $universal_2$ class of hash functions H_1 defined in [6]. It is experimentally shown that for a given set of keys, by choosing functions at random from the class H_1, the theoretically predicted performance of the hash functions can be achieved in practice, independent of the key distribution [11].

These translation schemes have been implemented as a part of the CHAOS runtime support library on various distributed memory multicomputers such as Intel Paragon, IBM SP-1/2, Thinking Machine CM-5 and Cray T3D. The current implementation has used vendor-supplied message passing libraries. It should be noted, though, that the translation schemes have been further optimized using the low latency shared memory functions on the Cray T3D [1]. The shared memory functions copy blocks of data directly from one processor's memory to another. These shared memory functions remove a substantial amount of overhead for synchronization. The last two columns in Table 1 demonstrate the optimized performance obtained from Cray T3D over that from Intel Paragon.

Another issue of the translation schemes is memory requirement. Suppose that N is the total number of global indices, P is the number of processors, $\mathcal{S}$ is a page size, and $\mathcal{R}$ is a replication factor. Then, the memory complexity of the paged translation scheme is given by $\mathcal{O}(N \times (\frac{1}{\mathcal{S}} + \mathcal{R}))$. In order to keep the amount of replicated memory scalable with large numbers of processors and large problems, it is desirable to make the page size $\mathcal{S}$ proportional to the number of processors P. However, the need for a large page size may result in severe performance degradation due to page thrashing. Thus, it may be a complicated process to choose an optimal page size under various situations. On the other hand, the hashed translation scheme requires $\mathcal{O}(N \times (\frac{1}{P} + \mathcal{R}))$ memory, which makes the hashed translation scheme ideally scalable. Accordingly, the hashed translation table may be more desirable in a situation where the memory constraint is tight.

6 Conclusions

This paper has presented a set of index translation schemes for implementing a user-level global index space across a collection of local index spaces on distributed memory multicomputers. These schemes have been incorporated into the CHAOS runtime support library so that calls to the library functions can be generated by compilers.

For unstructured problems with irregular data distributions, a distributed translation table can be built to list the home processor and offset for each globally indexed data item. Cached translation schemes use software caching techniques to reduce the dereferencing costs for adaptive irregular applications which require frequent index translations. Experiments have been performed with an adaptively irregular loop kernel and a 3-dimensional NASA Langley DSMC code. It has been observed that the software-cached translation schemes significantly outperform the distributed translation table for such problems with changeable data access patterns. For example, the hashed translation scheme achieved about 46 percent improvement with the DSMC code on the 32-node Paragon.

The performance of the software-cached translation schemes is sensitive to the choice of parameters which these schemes are governed by. Future work may include the extension of these schemes so that automatic selection of the parameters can be done using runtime information such as the amount of available memory and the fraction of locally accessed global indices.

Acknowledgments

The authors would like to thank Richard Wilmoth at NASA Langley for the use of DSMC production codes. Access to the Caltech CCSF Paragon and the Jet Propulsion Laboratory Cray T3D was provided by the Center for Research on Parallel Computation.

References

[1] Ray Barriuso and Allan Knies. Shmem user's guide. Report, Cray Research, Inc., April 1994.

[2] T. J. Bartel and S. J. Plimpton. DSMC simulation of rarefied gas dynamics on a large hypercube supercomputer, AIAA-92-2860. In *Proceedings of the 27th AIAA Thermophysics Conference*, Nashville, TN, June 1992.

[3] Graeme A. Bird. *Molecular Gas Dynamics and the Direct Simulation of Gas Flows.* Clarendon Press, Oxford, 1994.

[4] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel system. In *Proceedings Supercomputing '93*, pages 588–597. IEEE Computer Society Press, November 1993.

[5] B. R. Brooks and M. Hodoscek. Parallelization of CHARMM for MIMD machines. *Chemical Design Automation News*, 7, 1992.

[6] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[7] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. The design and implementation of a parallel unstructured Euler solver using software primitives, AIAA-92-0562. In *Proceedings of the 30th Aerospace Sciences Meeting*, January 1992.

[8] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Department of Computer Science Technical Report TR90-141, Rice University, December 1990.

[9] Milan Milenkovic. *Operating Systems Concepts and Design.* McGraw-Hill, Inc., 1987.

[10] Bongki Moon and Joel Saltz. Adaptive runtime support for direct simulation Monte Carlo methods on distributed memory architectures. In *Proceedings of the Scalable High Performance Computing Conference (SHPCC-94)*, pages 176–183, Knoxville, TN, May 1994. IEEE Computer Society Press.

[11] M. V. Ramakrishna. Hashing in practice, analysis of hashing and universal hashing. In *Proceedings of the 1988 ACM SIGMOD*, pages 191–199, June 1988.

[12] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 325–336. IEEE Computer Society Press, April 1994.

[13] J. Saltz et al. A manual for the CHAOS runtime library. Technical report, University of Maryland, Department of Computer Science and UMIACS, 1993.

[14] Joel Saltz, Harry Berryman, and Janet Wu. Multiprocessors and runtime compilation. Technical Report 90-59, ICASE, NASA Langley Research Center, September 1990.

[15] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.

[16] Richard G. Wilmoth. Direct simulation Monte Carlo analysis of rarefied flows on parallel processors. *AIAA Journal of Thermophysics and Heat Transfer*, 5(3):292–300, July-Sept. 1991.