Proc. Second Aizu Int. Symposium on Parallel Algorithms/Architectures Synthesis, Aizu-Wakamtsu, Japan, March 17-21, 1997, IEEE Computer Society Press, Los Alamitos, CA , pp. 42-49

Run-Time Reference Clustering for Cache Performance Optimization

Wesley K. Kaplow
Boleslaw K. Szymanski
Peter Tannenbaum
Department of Computer Science
Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, N.Y. 12180-3590, USA
{kaploww,szymansk,tannenp}@cs.rpi.edu

Viktor K. Decyk
Physics Department, UCLA
Los Angeles, CA. 90024, USA
and
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA. 91109, USA
vdecyk@pepper.physics.ucla.edu

Abstract

We introduce a method for improving the cache performance of irregular computations in which data are referenced through run-time defined indirection arrays. Such computations often arise in scientific problems. The presented method, called Run-Time Reference Clustering (RTRC), is a run-time analog of a compile-time blocking used for dense matrix problems. RTRC uses the data partitioning and re-mapping techniques that are a part of distributed memory multi-processor codes designed to minimize interprocessor communication. Re-mapping each set of local data decreases cache-misses the same way re-mapping the global data decreases off-processor references. We demonstrate the applicability and performance of the RTRC technique on several prevalent applications: Sparse Matrix-Vector Multiply, Particle-In-Cell, and CHARMM-like codes. Performance results on SPARC-20, SP-2, and T3-D processors show that single node execution performance can be improved by as much as 35%.

1. Introduction

One of goals of parallel program optimization is to keep data references as low as possible in the memory hierarchy. However, this is a difficult task for irregular computations that make array references via indirect indices. The indices are data-dependent, and thus it is impossible to determine at compile-time a static distribution of the data that will minimize inter-processor communications. Heuristic techniques, such as spectral bisection, simulated annealing, *etc.*, are used to create an irregular distribution of data in an attempt to minimize inter-processor communication. However, these methods introduce the problems of determining the run-time dependent remote access requirements of the application, and providing efficient facilities to perform the communication. These problems are addressed in the CHAOS/PARTI run-time and compilation methods [4, 12, 10]. The essential technique is the *inspector/executor* model in which the inspector is used to determine which references are required for execution, and the executor performs the communication and the actual computation.

Cache optimization for irregular problems share the same problems of multi-processor irregular data distribution: the reference pattern is not determined until run-time, and may vary during execution. During compilation of such programs it is impossible to determine a loop structure that will confine references to the current contents of the cache. It is exactly this type of problem that this paper addresses.

Several methods have been explored for improving cache performance of irregular computation. Multithreading [6] attempts to hide memory latency by creating many parallel threads of computation that can be finely scheduled with respect to the availability of data to process. Another approach is to modify the reference order to improve locality. For certain applications, such as finite-element methods, the performance of the cache can be improved by applying algorithms that narrow the bandwidth of the sparse-matrix constructed increasing reference locality [3]. However, this does not address the details of reuse, nor is it generally applicable to irregular problems. To our knowledge, [11] is the only paper that applies data repartitioning explicitly for cache optimization. They determine the size of a sub-domain of the local cache, based on an analysis of the data structures and algorithms of a problem, and then use a domain decomposition scheme at run-time to reorganize data to fit into these local cache regions. Their paper also examines software prefetching. When applied to KSR-1 (a Cache Only Memory Architecture) both of these techniques reduce the number of cache misses. The domain re-mapping decreases the execution time by 8%-19%, but prefetching does not reduce the execution time because of the extra cost of prefetching instructions.

In this paper we introduce a novel method of cache performance optimization for parallel programs based on the concept of Run-Time Reference Clustering (RTRC). The references of an irregular program are clustered in such a way that the data referenced in each cluster fits within the processor's cache. For static irregular codes, such as sparse matrix-vector multiplication, the number of clusters and their sizes are determined statically (once during runtime), similar to [11]. For dynamic scientific applications, such as Particle-In-Cell [1] and CHARMM [2], the clusters' scopes can change. Run-time management used to maintain the reference clusters is similar to local memory data-reallocation done for load balancing.

The remainder of the paper is structured as follows. Section 2 describes the RTRC model itself and application types that are amenable to it. Performance results for Sparse Matrix-Vector multiply, Particle-In-Cell simulation, and a CHARMM-like kernel are given in Section 3. Conclusions and final remarks are given in Section 4.

2. The RTRC Model and Target Applications

RTRC Model. Scientific computations often involve evaluating the interactions of each object from one collection with a specific subset of objects from another collection. (The two collections are not necessarily distinct.) Examples include:

- *Particle-in-Cell (PIC) codes,* in which each particle in a simulation interacts with its neighboring grid points.
- *Molecular Modeling (CHARMM) codes,* in which each simulated particle interacts with its neighboring particles.
- *Sparse Matrix-Vector Multiplication codes,* in which non-zero elements of the sparse matrix reference the corresponding elements of the dense vector.

The first two examples above are *dynamic* problems; particles in the physical simulations change their positions and therefore interact with different sets of objects during different simulation steps. Matrix multiplication is a *static* problem; matrix elements always interact with the same elements of the vector (although the vector's values change). All three examples share the following two properties: (i) the specific set of objects with which a given object of the first collection interacts is not known at compile time, and (ii) the order of interaction evaluations does not affect the result.

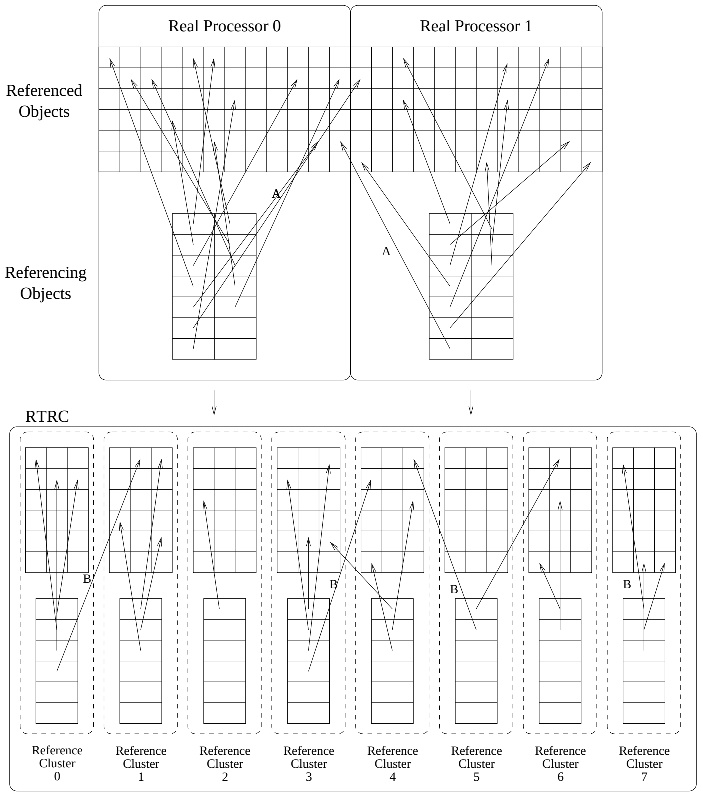

Figure 1. Run-Time Reference Cluster Model

The Run-Time Reference Clustering model optimizes this kind of computation. For simplicity, RTRC assumes that collections of interacting objects are represented as arrays. The specific nature of the interaction is immaterial in this model. Rather, the focus is on the pattern of interactions: is there an ordering that provides better performance when evaluating interactions? Intuitively, an ordering that guarantees a high degree of locality should provide optimal performance. Any type of interaction will involve data references for each element. If successive pairs of interacting elements are stored near enough to each other in memory, then the data references will enjoy a high cache-hit rate, thereby decreasing the effective memory access time. With static codes, a single repartitioning step can achieve an optimal ordering. For dynamic codes, the repartitioning must occur often enough to provide the desired degree of locality. Repartitioning at each step ensures optimal locality, but at the cost of increased overhead.

The optimal ordering scheme is application dependent. (Each specific ordering, regardless of the application, will also be data dependent.) In the case of CHARMM, each particle interacts only with the particles that lie within a given radius. With this in mind, clusters of particles in physical space should also be clustered together in memory. The PIC codes are similar, in that particles interact only with the nearest grid points. Again, clusters of particles in physical space should be clustered in memory. In the sparse matrix-vector multiply codes, the locations of the non-zero elements in the matrix determine the elements of the vector with which to interact. Here, an optimal ordering rearranges the interactions to achieve locality with respect to data references in the vector.

Each cluster of elements should fit within the cache. Finding an optimal ordering is a data partitioning problem: just as the global data is mapped to processors with the intent of minimizing inter-process communication, the data on each processor is partitioned, and thereby ordered, into reference clusters (RCs) with the intent of minimizing inter-RC communication. The RCs are made to fit within the cache, and therefore any interactions evaluated within the RC will enjoy a high cache-hit rate.

The target machine is described by the set of cache parameters that are critical in determining the performance of the application code. An object array A is divided into sub-arrays (RCs) depending on application code and the cache parameters.

Figure 1 shows an example of the RTRC method applied to two processors. Each processor stores the sub-arrays with referencing and referenced objects. In the figure, each referencing object points to exactly one referenced object. In general, the references will be scattered randomly over the upper array of referenced objects. In the bottom part of the figure each processor's data are partitioned into four RCs. Each cluster contains a section of both arrays.

The key feature of the RCs is that the referencing objects contain pointers to local sections of the array of the referenced objects. However, in dynamic codes, the referenced objects' effective addresses change during execution, requiring their reallocation between RCs.

An important observation can be made from Figure 1. At the upper part of the figure there are referencing objects that point to objects belonging to another processor. Such pointers (labeled A in the figure), are frequent in multi-processor execution. To provide access to off-processor data, a message will be created that moves such objects to the processor to which its reference points. A completely analogous situation exists in the Reference Clusters (RCs). The pointers labeled B originate at objects that, due to the execution of the program, need to access data in another RC. In exactly the same way as above, a referencing object is reallocated from the first RC to the second RC to maintain good locality of data references.

RTRC Application Structure. The following steps are used to apply the RTRC method to an application:

1. Determining the number of reference clusters, which can be done heuristically, analytically, or experimentally at compile time.

2. Creating reference clusters, On each processor, several clusters can be created depending on the data structure and cache sizes.

3. Modifying the code to execute on a reference cluster (RC) basis at run-time. In most cases the modification follows the changes introduced to support load balance of multi-processor execution.

```
SUBROUTINE PMULT(...)
   DO IRC = 1, NRC
     IBR = (IRC-1)*(NCOL+1) + 1
     IER = IBR + NCOL - 1
     ICOL = 0
     DO I = IBR, IER
       ICOL = ICOL + 1
       IBGN = COLPTRA(I)
       IEND = COLPTRA(I+1)-1
       SUM = 0.0D00
       IF (IBGN .LE. IEND) THEN
         DO J = IBGN,IEND
           JAJJ = ROWIND(J)
           SUM = SUM + VALUES(J)*X(JAJJ)
         ENDDO
       ENDIF
       W(ICOL) = W(ICOL) + SUM
     ENDDO
   ENDDO
   DO I = 1,NCOL
     X(I) = W(I)
   ENDDO
RETURN
```

Figure 2. ITPACK Matrix-Vector Multiplication Code With RTRC

Compile-Time Determination of RC Size. The creation of Reference Clusters (RCs) on a processor requires a compile-time estimation of the ranges of the arrays that should be allocated to an RC. There are several methods that can be used to determine the size of the arrays allocated to the virtual processors. An *execution* or *benchmark* method can be used, or a cache performance estimation technique can be employed. We describe a method in [7] that can be used to determine quickly the optimal range values by partial simulated execution of the application code on an architecturally correct model of the target processor's cache.

Reference Cluster Creation and Code Modification. The data structures used to support RC creation can be the same as the structures required to support the movement of program objects from one real processor to another in a multi-processor implementation of an application. The code modifications required can also follow this pattern. One way this can be accomplished is via the specification of multiple *virtual* processors on each real processor when setting compiler directives for automatic parallelizing languages such as

HPF. We currently have a preliminary implementation of a language pre-processor that assists in this process.

Run-Time Component. Reallocation of referencing objects between RCs must be performed at run-time in response to the changing reference pattern of each of such objects. The overhead associated with reallocating reference objects between reference clusters must be small enough so as not to negate the performance improvement resulting from better cache hit-rates. For example, testing each object in every iteration may be too costly. Since we are dealing with the statistical nature of the cache, it is clear that some number of non-local RC references will not significantly impact the performance of the RC. Therefore, we can define two types of run-time systems: *strict* when during execution each object making non-local references is reallocated to its proper RC, and *non-strict* otherwise.

A *non-strict* method could be designed according to one of these possible schemes:

1. Count non-local RTRC requests. When a threshold is exceeded, a reallocation of the referencing objects would be made to restore RC locality.

2. A periodic reallocation could be used in which the referenced objects are changed every $freq$ interaction evaluations.

Target Application Characteristics. In general, the RTRC method is useful if the program reference pattern is defined at run-time. In scientific applications such programs use index arrays to reference data arrays. For some applications, like sparse matrix-vector multiply, these indirection arrays do not change during execution. In many programs, such as finite-element solvers, sparse matrix-vector multiplication is a key algorithm that is used to iteratively solve a linear system. The references are static in the sense that the reference pattern is a function of the sparse matrix contents which do not change during iterations. Since dense matrix optimizations for improving cache performance rely on subscript analysis, the use of indirection in subscript expressions in codes using sparse matrices makes these methods useless. Techniques for improving the performance of sparse matrix-vector multiplication on parallel architectures have focused on improving processor partitioning to reduce remote processor communication costs [13]. Figure 2 shows the code for sparse matrix-vector multiplication taken from ITPACK [8] with RTRC applied (The code multiplies A by X to find W.) In general, a multi-processor implementation of this algorithm would assign to each processor a contiguous number of rows of A,X, and W. Since each sparse row of A is stored in contiguous addresses (in *VALUES*), its cache line reuse is optimal, as are accesses to W. However, access to the X array follows the sequence of non-zero elements in a row access. Usually, the number of non-zeros is below a few percent of the total number of elements in the matrix, and therefore accesses to X will be widely spaced. Unless all elements of X fit into cache, there will be no cache reuse on those references. This data locality can be improved by applying the RTRC method to partition the rows into reference clusters. Due to space limitations the code to create the reference clusters from the initial sparse matrix could not be included.

```
SUBROUTINE PUSH(PART, FX, FY, NX, NY, NOP,IDIMP,
      nRC,mxperRC,mxsndRC,
      nsRC, nsRCy, ndRCx,
      sendlist, destlist, npsend, iphole
DIMENSION FX(NX, NY), FY(NX, NY)
DIMENSION PART(IDIMP, mxperRC,nRC)
dimension npRC(nRC)
dimension sendlist(idimp,mxsndRC,nRC)
dimension destlist(mxsndRC,nRC)
dimension npsend(nRC),iphole(nRC)

DO iRC=1,nRC
      DO I=1,npRC(iRC)
      N = PART(1,J,iRC) + 0.5
      M = PART(2,J,iRC) + 0.5
      DX = function(FX,N,M)
      DY = function(FY,N,M)
      PART(1,J,iRC) = function(DX)
      PART(2,J,iRC) = function(DY)
      if(freq(nsratio) .eq.  1) then
          ipdest = RCMAP(col1,col2)
          if (ipdest .ne.  iRC) then
              call MoveOpReq(J,RC,ipdest,...,
                  PART,sendlist,
                  destlist,npsend,iphole,...)
              endif
          endif
      END DO
END DO
call DoMoveReq(PART,npRC,nRC,sendlist,
      destlist,npsend,iphole,...)
```

```
SUBROUTINE NONBOND(PART,NPART,X, Y, DX, DY,
  NOP,MAXPART)
DIMENSION X(NOP), Y(NOP),DX(NOP),DY(NOP)
DIMENSION PART(MAXPART, NOP),NPART(NOP)

DO iRC=1,nRC
  DO I=1,npRC(iRC)
  use variables X(I,iRC),Y(I,iRC)
        DO J = 1,NPART(I,iRC)
            P = PART(J,I,iRC)
            use variables X(P,iRC),Y(P,iRC)
            perform force calculation
                use variables DX(P,iRC),DY(P,iRC)
        ENDDO
        update variables DX(I,iRC),DY(I,iRC)
  ENDDO
```

Figure 3. Example PIC and CHARMM Code Templates with RTRC Application

The reference pattern of dynamic irregular codes is created by indirect array references that change values during program execution. The example procedure $PUSH$ in Figure 3 illustrates a typical template for sections of Particle-in-Cell codes. The plasma Particle In Cell simulation model integrates in time the trajectories of millions of charged particles in their self-consistent electromagnetic fields. Particle interactions are not modeled directly, but through the fields which they produce. Particles can be located anywhere in the spatial domain; however, the field quantities are calculated on a fixed grid. The General Concurrent Particle in Cell (GCPIC) Algorithm [9] partitions the particles and grid points among the processors of the MIMD (multiple-instruction, multiple-data) distributed-memory machine. A secondary decomposition partitions the simulation space evenly among processors, which makes solving the field equations on the grid efficient. As particles move among partitioned regions, they are passed to the processor responsible for the new region.

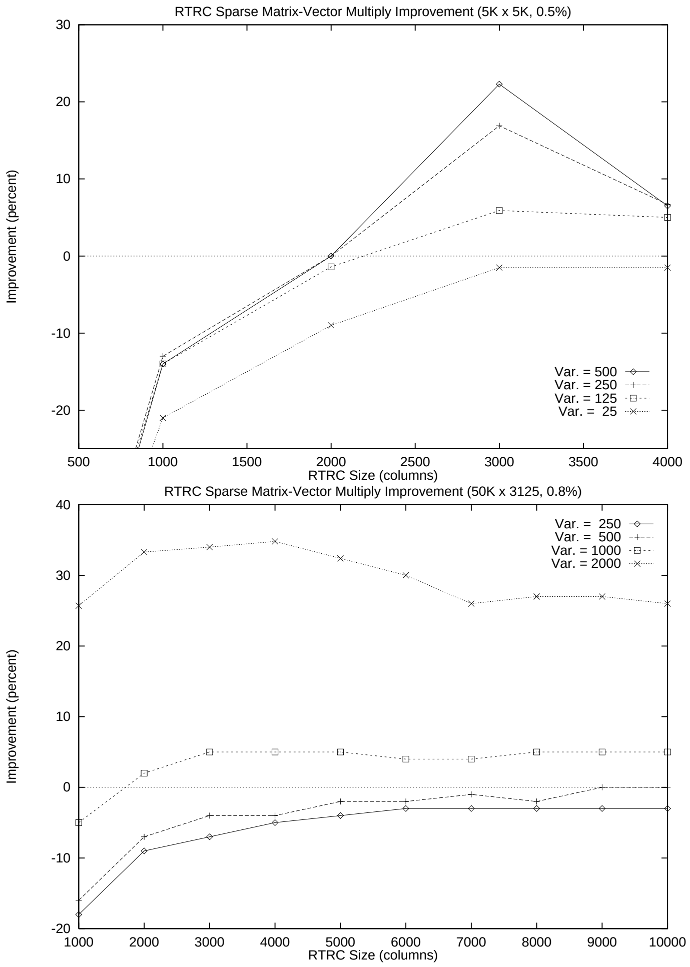

Figure 4. RTRC Performance Improvement vs. RC Size and Distribution on a single Node and Multiprocessor SP2 Sparse Matrix-Vector Multiplication

Figure 3 shows and example application of the RTRC method to a PIC code template. There are three principle arrays in the PIC code. The array *PART* is used to store the *X* and *Y* positions of all of the simulated particles (there are NOP total particles). The *FX* and *FY* arrays represent the electric field in a 2D simulation. This computation is repeated at the *particle push* and *charge deposit* stages of PIC codes and it consumes the majority of the execution time of a simulation. The order of traversal of the *PART* array is irrelevant for the results.

Cache performance can be poor because references to the *FX* and *FY* arrays are dependent on the run-time values of the *PART* array which change during execution. Over time, the reference pattern to *FX* and *FY* become essentially random, leading to as much as a 40% performance decrease [5] over an initially sorted order.

The creation of RCs in the PIC code directly follows the approach that supports multi-processor execution. In this case the same data structures that provide the mechanism for interprocessor communication are reused to supply inter-RC object movement. Lines 20 thru 27 represent the run-time support for the periodic reallocation of objects in the reference clusters using the non-strict policy. The $MoveOpReq$ assigns an object to the new reference cluster, and, when necessary calls the relevant inter-processor run-time procedure to move this object between processors. $DoMoveReq$ is an RTRC procedure that effects the object movement between RCs on the same processor.

Another application suitable for RTRC are CHARMM-like codes that model molecular dynamics in macromolecular systems. The computationally intensive parts of CHARMM are the molecular dynamics simulation routines. In particular, the *non-bonded* force calculation represents up to 90% of the total execution time because each modeled atom needs to account for its interaction with all other atoms in the system. The computation is of $O(N^2)$ time complexity, but it is simplified in CHARMM by ignoring all interactions with atoms that are greater than a certain cutoff radius away. Thus, each atom in the simulation maintains a dynamically changing list of all the atoms present within its cutoff radius. Figure 3 shows a template for the non-bonded force calculation ($NONBOND$) characteristic of CHARMM codes. The *PART* array in this case holds the adjacency information for each atom.

As stated before, to create the reference clusters, the RTRC method reuses the partitioning method (*e.g.*,RSB, Recursive Spectral Bisection) included in the original code to create an irregular data distribution for multi-processor execution. However, now RSB is applied to an individual processor partition to subdivide it into pieces fitting into cache. The resulting sub-partitioning reduces the number of inter-RC reference for the same reason that the the original partitioning minimizes inter-processor communication. As atoms move between RCs (operation performed in another phase of the CHARMM algorithm), the run-time support of both RTRC and inter-process load balancing have to maintain the quality of the partitions. Our current implementation is limited to a single initial RC partitioning and a periodic complete repartitioning to maintain reference locality.

3. Results

In this section we present the performance results of applying the RTRC method to codes described in the previous section.

Sparse Matrix-Vector Multiplication. Many problems that are solved using large sparse matrices have most of their elements located close the the diagonal. Therefore, in order to provide a realistic test of the RTRC modified matrix-vector multiplication, a sparse matrix generation program was created with the following user-defined parameters: the number of rows and columns, density, and variance. The density determines the number of non-zero elements in the array. A normal distribution is used to cluster non-zeroes around the diagonal with the given variance. Raising the variance widens the band. There is little or no improvement when the variance is small because all non-zero elements are close to the diagonal and accessed with good locality. As the variance increases the performance improvement grows, reaching 32% for Sun SPARC-20.

The top graph in Figure 4 shows results for a single IBM SP2 node. In this case a diagonal sparse matrix is used and both the variance and cluster size are varied. As in the SPARC-20 results, the performance is proportional to the variance. Also, there is an optimal value for the cluster size. If the size it too small then the extra overhead of managing the reference clusters causes a significant performance degradation. If the size to too large, then the locality that the Reference Cluster defines is larger than the cache.

The bottom graph in Figure 4 shows the performance results for a single processor in a multiprocessor configuration. The total size of the matrix is $50,000 \times 50,000$ with $1,250,000$ non-zeros. With 16 processors, 3125 rows are assigned to each processor. The results show, as in the uniprocessor case in Figure 4, that the variance determines whether using RTRC is beneficial or not.

Particle-in-Cell. Performance results of applying the RTRC method to PIC codes are presented. The x-axis of all of the graphs represents the iteration step of the simulation. The y-axis represents either the time of the particle push and deposit charge routine, or a ratio indicating performance improvement. Three processor architectures show the application of the method: the Sun SuperSPARC processor, a Cray T3D node, and an IBM SP2 node.

The top of Figure 5 shows a performance comparison between the original PIC code and RTRC optimized code. The

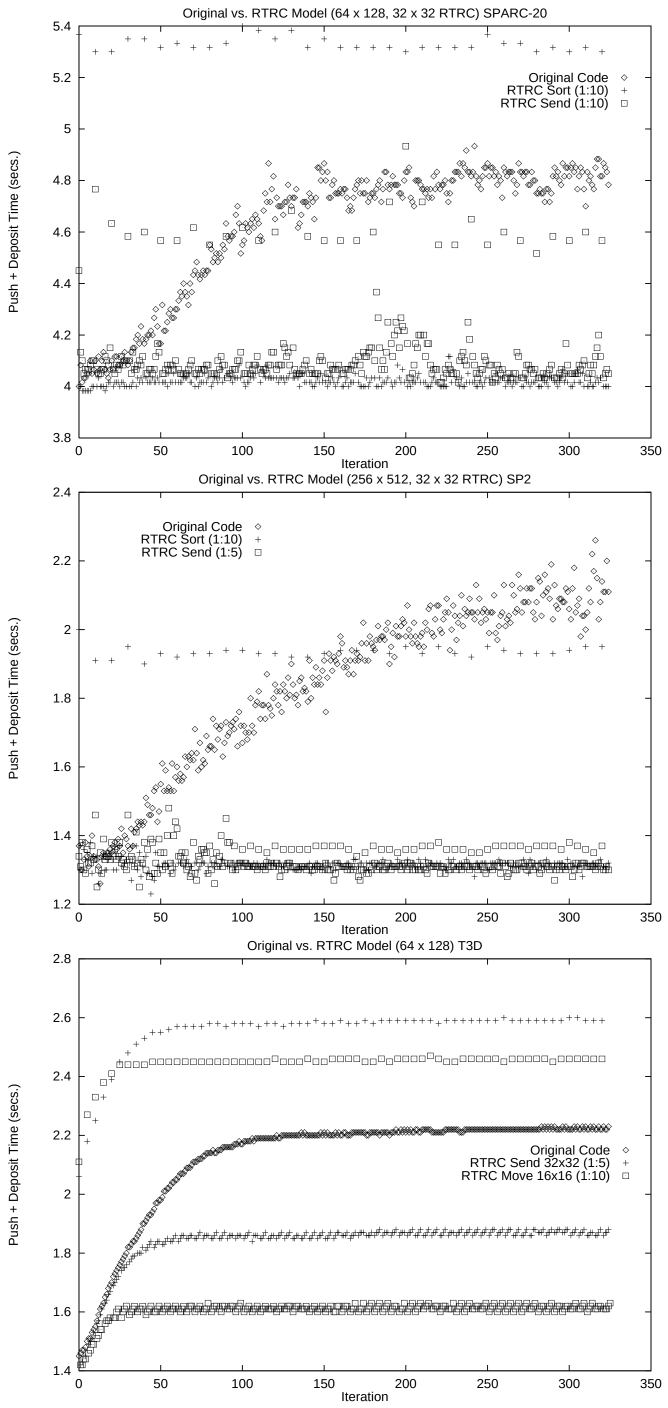

Figure 5. Performance Improvements: SunSPARC-20, SP2 and T3D Processors

PIC code example included a *GRID* array of size 64×128 and *PART* array with $327,680$ entries. The reference cluster size is 32×32, and, therefore, there are eight reference clusters. In both cases the performance of the original code deteriorates as the simulation proceeds. Initially, the *PART* array contains particles sorted by their positions and consecutive reference to *GRID* are made to adjacent values in the cache. However, as the particles move over time and change their referenced grid points, this locality of reference is lost. For example, by the 175^{th} iteration, cache misses decrease the performance by $13 - 20\%$. This figure shows two different methods to maintain the reference clusters during execution. Both methods maintain performance that is within a few percent of the optimum but differ in the overhead required to maintain strict RC order. The *sort* method represents a complete periodic local data repartitioning and has a 5.3 second overhead. The *send* method represents periodic restoration of the reference cluster by moving particles from one reference cluster to another with a 4.6 second overhead. Of course, the repartitioning method must move every reference in the particle array twice, and therefore we would expect that this would take longer than the send method. However, by allowing non-strict operation of the RTRC run-time we can amortize this cost over a number of iterations. In this case the run-time attempts to make a strict ordering once every 10 iterations, and the average overhead time of these methods become approximately 0.5 seconds per iteration on a SPARC-20.

The SP2 results in Figure 5 shows a PIC problem size where the *GRID* array has been increased to be 256×512. The size of the RC is 32×32. The performance of the original code degrades from 1.3 to 2.2 seconds, a 41% performance drop. The average cost is 1.72 seconds for the repartitioning method, and 1.31 seconds for the send method, when the repartition is performed every 10^{th} iteration and the send every fifth. This maintains performance within approximately 15% of optimal.

The last architecture compared here is the Cray T3D. This processor has a small direct mapped cache, and therefore we would expect that the RC size would have to be much smaller. In the bottom Figure 5 the performance of the original code decreases by 36%. Two sizes of RCs are also shown. The first size represents approximately 50% of the entire cache capacity, while the second size (16×16) is $\frac{1}{8}^{th}$ the capacity. The performance of the send code represents 11% and 34% improvement respectively.

The top of Figure 6 shows the results of a comparison of different RC sizes. As expected, over some threshold size, the performance of too large an RC decreases as the simulation proceeds. This effect is caused by the deterioration of locality from the initial sorted order, and the self-interference misses caused by having too large a locality for the cache.

Selecting an RC smaller than necessary does not have a large impact on performance. The cache locality provided by RTRC is not influenced by a small RC, but the RTRC run-time overhead does increase as the number of RCs increase.

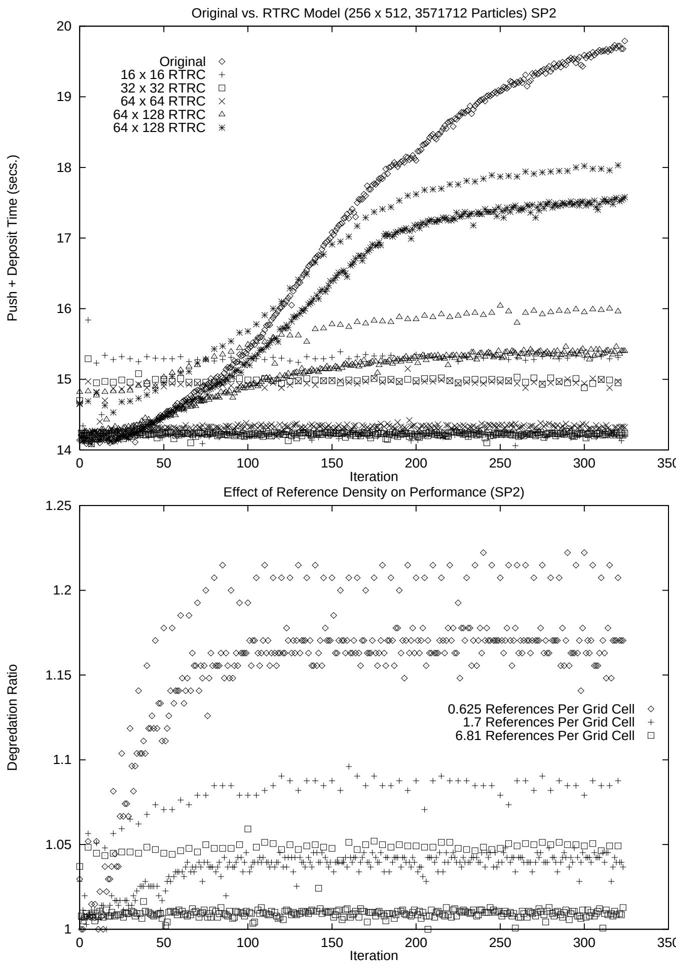

Figure 6. Impact of RC Size Selection and Reference Density on SP2 Performance

The second graph in Figure 6 shows the effect of reference density on performance. If the number of references per grid cell is too small then there is not enough reuse of data in the cache to amortize the cost of bringing in the cache line from memory. The figure shows that for a reference density of less than one there is a degradation of approximately 18% from the optimal time. A reference density of 1.7 improves this to only a 5% degradation. A density ratio of nearly 7 provides the needed cache reuse to maintain optimum performance.

CHARMM Performance. A benchmark was derived from the CHARMM code to study the effect of applying RTRC techniques. In this case, the benchmark consisted of the non-bonded force calculation in two dimensions. A initial population of atoms was distributed randomly over a field. Reference clusters were created by applying domain decomposition and reordering of atoms in the *PART* array. Results are measured with respect to the *worst* case, which is when the locations of atoms in the various arrays used in the calculation are not correlated with their partners for the entire array. Figure 7 shows the performance improvement for both a SPARC-20 processor and a single node of an SP2 each for different reference cluster sizes. With an RC size of approximately 600 the performance improves by 25% and 34% respectively.

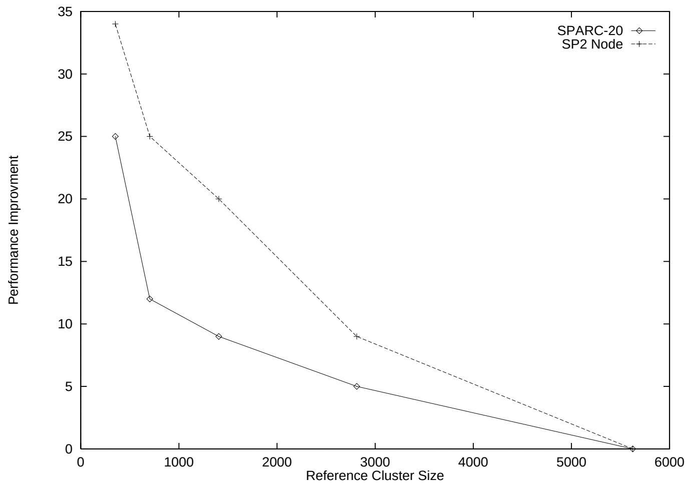

Figure 7. CHARMM Performance Improvement

4. Conclusions

The RTRC model presented here provides a viable framework for optimizing the use of the cache and memory hierarchy for irregular static and dynamic codes. As described in this paper, there are two components that are integrated to form the model. The first is to determine a close to optimal reference cluster size, and use an irregular data partitioning method to distribute the data to each reference cluster. The second is a method for keeping the references local to each reference cluster during execution. We have shown that the overhead of the RTRC method is low enough to enable significant performance improvements. We have a preliminary version of a compiler that will automatically apply the RTRC method to PIC codes. We intend to add the additional capability of integrating a user supplied data partitioning function to support other applications, as well as looking at the possibility of using the RTRC method to direct fine-grained thread scheduling and data prefetching in an attempt to further reduce effective memory access times.

Acknowledgments The authors thank Charles Norton for his comments and assistance. This work was supported in part by NSF Grant CCR-9527151. The content does not necessarily reflect the position or policy of the U.S. Government.

References

[1] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. McGraw-Hill, New York, 1981.

[2] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[3] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy. Design and implementation of a parallel unstructured euler solver using software primitives. *AIAA*, 32(3):489–496, March 1994.

[4] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communiction optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22:462–478, 1994.

[5] V. K. Decyk, S. R. Karmesin, A. de Boer, and P. C. Liewer. Optimization of particle-in-cell codes on risc processors. *Computers in Physics*, 1995. Submitted for Publication.

[6] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill, 1993.

[7] W. K. Kaplow and B. K. Szymanski. Program optimization based on compile-time cache performance prediction. *Parallel Processing Letters*, 6(1):173–184, 1996.

[8] D. R. Kincaid, J. R. Respess, D. M. Young, and R. G. Grimes. Itpack 2c: A fortran package for solving large sparse linear systems by adaptive accelerated iterative methods. Technical report, University of Texas at Austin, 1992.

[9] C. D. Norton, B. K. Szymanski, and V. K. Decyk. Object-oriented parallel computation for plasma simulation. *Communications of the ACM*, 38(10), October 1995.

[10] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, and G. Fox. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems*, 6(8):815–831, August 1995.

[11] K. A. Tomko and S. G. Abraham. Data and program restructuring of irregular applications for cache-coherent multiprocessors. In *8th ACM International Conference on Supercomputing, Manchester, England*. ACM, July 1995.

[12] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Transactions on Computers*, 44(2):737–753, 1995.

[13] L. H. Ziantz, C. C. Ozturan, and B. K. Szymanski. Run-time optimization of sparse matrix-vector multiplication on simd machines. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE 94 Parallel Architectures and Languages Europe, Athens, Greece*, volume 817 of *Lecture Notes in Computer Science*, pages 313–322. Springer-Verlag, July 1994.