Data Parallel Performance Optimizations Using Array Aliasing

DATA PARALLEL PERFORMANCE OPTIMIZATIONS USING ARRAY ALIASING

Y. Charlie Hu
S. Lennart Johnsson

TR-13-97

July 1997

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

To appear in Rob Schreiber, Mike Heath and Abhiram Ranade, editors, *Algorithms for Parallel Processing*, IMA Volumes in Mathematics and its Applications. Springer–Verlag, 1997.

DATA PARALLEL PERFORMANCE OPTIMIZATIONS USING ARRAY ALIASING

Y. CHARLIE HU* AND S. LENNART JOHNSSON†

Abstract. The array aliasing mechanism provided in the Connection Machine Fortran (CMF) language and run–time system provides a unique way of identifying the memory address spaces local to processors within the global address space of distributed memory architectures, while staying in the data parallel programming paradigm. We show how the array aliasing feature can be used effectively in optimizing communication and computation performance. The constructs we present occur frequently in many scientific and engineering applications, and include various forms of aggregation and array reshaping through array aliasing. The effectiveness of the optimization techniques is demonstrated on an implementation of Anderson's hierarchical $O(N)$ N–body method.

Key words. Data parallel programming, array aliasing, hierarchical N–body methods.

AMS(MOS) subject classifications. 68N15, 68N20, 70–08, 70F10

1. Introduction.

Data parallel programming provides an effective way to write maintainable, portable, and scalable parallel codes. The proprietary data parallel programming language Connection Machine Fortran (CMF) [9], with many characteristics in common with the emerging data parallel programming language, High Performance Fortran (HPF) [3], has been successfully used in solving many structured and unstructured problems in science and engineering. The array aliasing mechanism provided in CMF and its run–time system provides a unique way of identifying the local memory address space as part of the global address space, while staying in the single–threaded control and a global address space. This feature was frequently used in optimizing the performance of many applications by offering a technique at the language level for managing memory references.

This paper presents a set of optimizations for both communication and computation actions using the array aliasing mechanism in CMF. The particular constructs we present aim at

- minimizing local memory moves in global communication operations,
- avoiding the need for general communication primitives when specialized, and on most architectures more efficient, communication primitives such as CSHIFT are specified,

* Aiken Computation Lab, Harvard University, 33 Oxford Street, Cambridge, MA 02138. The first author was supported by the Air Force Office of Scientific Research through grants F49620-93-1-0480 and F49620-96-1-0289.

† Department of Computer Science, University of Houston, Houston, Texas 77204-3475. The second author was supported by the Air Force Office of Scientific Research through grants F49620-93-1-0480 and F49620-96-1-0289.

- aligning and operating on nonconforming arrays without growth in memory requirements and excessive communication,
- arithmetic performance enhancement through aggregation.

The effectiveness of our techniques is demonstrated on an implementation of Anderson's hierarchical $O(N)$ N–body method [1] for the Connection Machine system CM–5/5E. Of the total execution time, communication accounts for about 10–20% of the total time, with the average efficiency for arithmetic operations being about 40% and the total efficiency (including communication) being about 35%.

Section 2 describes the array aliasing feature of CMF. Section 3 presents briefly the computational structure of hierarchical methods and the computational elements in Anderson's method applied to the evaluation of gravitational or Columbic N–body interactions. The optimization techniques using the array aliasing mechanism in CMF are presented in Sections 4 – 6. Section 7 reports some performance results of our implementation of Anderson's method using the optimization techniques, and Section 8 summarizes the results.

2. High Performance and Connection Machine Fortran.

Below we briefly summarize the new features in HPF. We then present the array aliasing mechanism in CMF, which provides an elegant way to avoid excess data motion, and compare it to the use of extrinsic procedures for the same purpose.

2.1. High Performance Fortran.

HPF consists of Fortran 90 with extensions mainly for data management. The main extensions are:

1. Data distribution directives, which describe data aggregation, such as cyclic and block aggregation, and the partitioning of data among memory regions;
2. Parallel `FORALL` statements and constructs, which allow specifications of parallel computations on fairly general array sections;
3. Extrinsic procedures (local procedures), which define interfaces to procedures written in other programming paradigms, such as explicit message–passing SPMD style;
4. A set of extended intrinsic functions, including mapping inquiry intrinsic subroutines that allow a program to know the exact mapping of an array at run–time.

HPF supports data parallel programming with a global address space. Programs can be written without any knowledge of the architecture of the memory system. HPF has inherited the Random Access Memory model used in most programming languages, with the data distribution directives providing a mechanism for the programmer to manage data layout indirectly. In a hierarchical distributed memory system, such as in *distributed memory* and *distributed shared memory* architectures, efficient use of the memory system and communications facilities require complex analysis and optimization, largely beyond state–of–the–art compiler and run–time system technology for such architectures. The consequence is that most compilers for such architectures often generate excess data movement, and frequently fail to invoke the most efficient communication mechanism. For instance, performing a circular shift on an array causes most compilers at best to issue instructions that move every element as specified by the shift instruction. At worst, a call to general communication facilities is generated. A more efficient way of dealing with shift instructions is to move data between processors as required, but to eliminate the local memory moves by modifying subsequent local memory references to account for the specified move (that was not carried out). One sensible way of avoiding excess data movement is to restructure the program in such a way that even a not–so–sophisticated compiler is able to generate efficient code. This goal can be achieved by exposing the local memory and processor address spaces and giving a programmer explicit control over data allocation and data references.

2.2. Connection Machine Fortran. In CMF, separation of the local and processor address spaces is elegantly achieved through *array aliasing* within the global programming paradigm. The array aliasing mechanism allows a user to address memory already allocated for an array, as if it were of a different type, shape, or layout. No data motion occurs.

Example 1. Let A be an n–dimensional array with shape $L_1 \times ... \times L_n$. In the mapping of A onto the physical machine, assume that there are p_i processors used for axis i, resulting in a subgrid of length s_i for axis i within each processor, i.e., $L_i = s_i \times p_i$. Using array aliasing, an array alias A_{alias} with shape $s_1 \times ... \times s_n \times p_1 \times ... \times p_n$ can be created, with the first n axes local to each processor, and the last n axes fully distributed.

Example 2. Let A be a two–dimensional array with shape $L_1 \times L_2$. Assume A is mapped onto the physical machine such that there are p_i processors used for axis i, resulting in a subgrid of length s_i for axis i within each processor, i.e., $L_i = s_i \times p_i$, just as in Example 1. Now, using array aliasing, an array A_{alias} with shape $s_1 \times s_2 \times p_1 \cdot p_2$ can be created such that the first two axes are local to each processor and the last axis fully distributed.

In the above two examples, we have explicitly identified the local address space as part of the global address space. The subgrid equivalencing feature in CMF provides a means of managing memory accesses similar to that of the EQUIVALENCE statement in Fortran 77.

In CMF, the array alias variable is declared to have type array descriptor, and the aliasing is created by calling the CMF utility library procedures [11]. The aliased array is then passed to a subroutine in which it is declared to be an array with the desired new type, shape, or layout. The following code segment[1] illustrates the programming concept:

[1] All the code examples in this paper will be in CMF.

```
      SUBROUTINE CALLER(L1,L2)

      INTEGER L1,L2,S1,S2,P1,P2
      REAL*8 A(L1,L2)
CMF$LAYOUT A(:,:)
      INTEGER A_ALIAS(CMF_SIZEOF_DESCRIPTOR)

C     1. determine s1,s2,p1,p2 by calling array inquiry procedures
C     2. creat the array alias in Example 2 by calling the utility
C        library procedures
      ...
      CALL CALLEE(A_ALIAS,S1,S2,P1*P2)
      STOP
      END

      SUBROUTINE CALLEE(A,S1,S2,P1P2)
      INTEGER S1,S2,P1P2
      REAL*8 A(S1,S2,P1P2)
CMF$LAYOUT A(:serial,:serial,:)

      ...
      RETURN
      END
```

In the current version of HPF, the separation of local and processor address spaces can only be achieved through the use of extrinsic (local) procedures. Within a local procedure, a program can access directly only the memory local to a processor. Access to other parts of the global memory must either be made through explicit message passing, or by returning to the global program. Hence, within HPF, optimizations based on separation of address spaces cannot be achieved within the language itself, but only by mixing programming models (data parallel and message passing). Moreover, mixing programming models and the use of procedure calls increase the difficulty of many forms of compiler optimizations.

3. Hierarchical methods. Hierarchical methods are often considered a serious challenge with respect to performance of data parallel programs. In addition to having many of the same challenges as nonhierarchical methods with respect to programming, they also introduce issues associated with poor parallelism close to the root of the hierarchical decomposition as well as low memory and arithmetic efficiencies in operations involving nonconforming arrays. Hierarchical N–body methods encompass yet another issue, namely, the interaction between two different data structures; one for the discrete particles, one for the discretized fields. We used an implementation of Anderson's hierarchical N–body method [1] to evaluate the effectiveness of the techniques we propose. However, only the techniques described for *multigrid–embed* and *multigrid–extract* addresses issues unique to hierarchical methods. All other techniques apply to non-

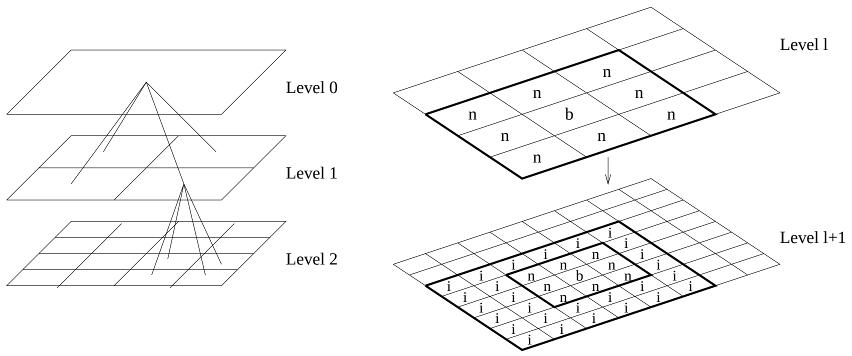


FIG. 3.1. *Recursive domain decompositions, the near–field, and the interactive–field in two dimensions.*

hierarchical methods as well.

All code fragments are extracted from our N–body code. Therefore, we here present briefly some essential characteristics of hierarchical N–body methods. Almost all constructs can be understood with no understanding of such methods, however. The hierarchical N–body methods [1,4,12] applied to potential field evaluation partitions the field into two parts:

$$\phi_{total} = \phi_{near-field} + \phi_{far-field}, \tag{3.1}$$

where $\phi_{near-field}$ is the potential due to nearby particles and $\phi_{far-field}$ is the potential due to faraway particles. The near–field is evaluated through the classical N–body technique of pairwise interactions, while the far–field is evaluated hierarchically. The $O(N)$ hierarchical methods differ in the computational elements they use, but share the same computational structure. The hierarchical domain decomposition is illustrated in Figure 3.1. Mesh level 0 represents the entire domain. Mesh level $l+1$ is obtained from level l by subdividing each subdomain at level l (*parent domain*) into four (in two dimensions) or eight (in three dimensions) equally sized subdomains (*child domains*). Subdomains that are not further subdivided are *leaves.*

In Anderson's method, Poisson's formula is used to represent solutions of Laplace equation. Let $g(x, y, z)$ denote potential values on a sphere of radius a, and Ψ denote the harmonic function external to the sphere with these boundary values. Given a sphere of radius a and a point $\vec{x}$ with spherical coordinates (r, θ, ϕ) outside the sphere, let $\vec{x_p} = (cos(\theta)sin(\phi), sin(\theta)sin(\phi), cos(\phi))$ be the point on the unit sphere along the vector from the origin to the point $\vec{x}$. Then, the potential value at $\vec{x}$ is approximated by (equation (15) of [1])

$$\Psi(\vec{x}) \approx \sum_{i=1}^{K} \left[\sum_{n=0}^{M} (2n+1)(\frac{a}{r})^{n+1} P_n(\vec{s_i} \cdot \vec{x_p}) \right] g(a\vec{s_i}) w_i \tag{3.2}$$

where P_n is the nth Legendre function, K the number of integration points on the sphere, $\vec{s_i}$ their location and w_i their weights. This approximation is called an *outer–sphere approximation*.

The approximation used to represent potentials inside a given region is (equation (16) of [1])

$$\Psi(\vec{x}) \approx \sum_{i=1}^{K} \left[\sum_{n=0}^{M} (2n+1)(\frac{r}{a})^{n+1} P_n(\vec{s_i} \cdot \vec{x_p}) \right] g(a\vec{s_i}) w_i \tag{3.3}$$

and is called an *inner–sphere approximation*.

The outer–sphere and the inner–sphere approximations define the computational elements in Anderson's hierarchical method. Outer–sphere approximations are constructed for clusters of particles in the leaf–level subdomains. During the upward pass, outer–sphere approximations of subdomains are combined into a single outer–sphere approximation of the parent subdomain. In the downward pass, the effects of the subdomains marked i in Figure 3.2 on the subdomain marked X and its descendants are evaluated. Lastly, the leaf–level computations involve the evaluation of the effects of the hierarchically evaluated far–field on the particles, and the direct evaluation of the effects of particle interaction in the near–field.

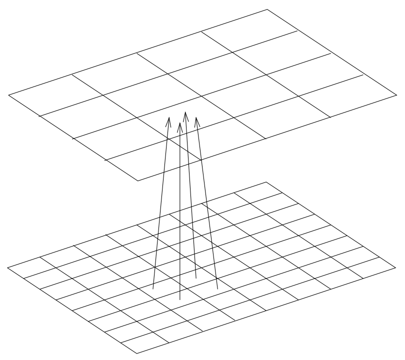
4. Data structures and data distribution.

We start the discussion of our optimization techniques using array aliasing with the type of data structure used for the representation of persistent variables in the hierarchy.

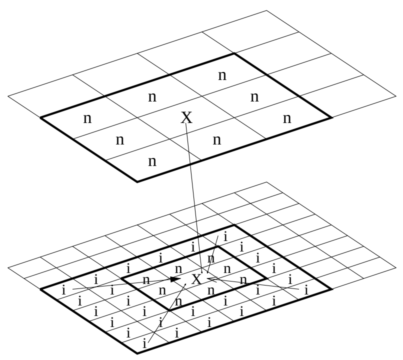
In most hierarchical methods, it is important to assure that data for a subdomain is allocated to the same processor as data for its parent domain, whenever there is a sufficient number of parent domains to cover all processors. To accomplish this form of locality we represent persistent data for the hierarchy by five–dimensional (5–D) arrays that effectively consists of two 4–D arrays with the same layout as shown in Figure 4.1. Three of the axes represent the organization of the subdomains in the three spatial dimensions, while a fourth axis is used to represent data local to a subdomain. The declaration of the hierarchy of subdomains (for the far–field potential) in CMF is:

```
      REAL*8 FAR_POT(2,K,L,M,N)
CMF$LAYOUT FAR_POT(:SERIAL,:SERIAL,:,:,:)
```

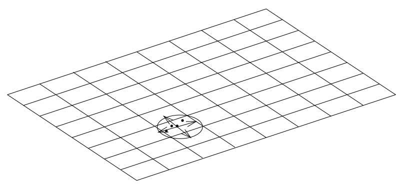
The compiler directive CMF$LAYOUT specifies that the rightmost three axes of array FAR_POT are parallel axes and that the two leftmost are local to each processor (specified through the CMF attribute `:SERIAL`, or * in HPF). The rightmost three axes represent the subdomains at the leaf–level of the hierarchy along the z–, y–, and x–coordinates, respectively. The local axis of extent K is used to store field values local to a subdomain. The leaf–level subdomains are embedded in FAR_POT$(1, :, :, :, :)$, while levels $(h-i)$ are embedded in FAR_POT$(2, :, 2^{i-1} : L : 2^i, 2^{i-1} : M : 2^i, 2^{i-1} :$



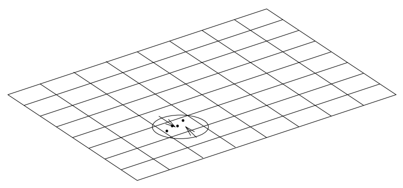
2. Upward pass: for far-field potential



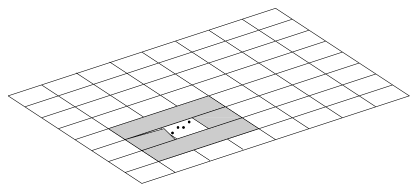
3. Downward pass: form local-field potential



1. Leaf level: form far-field potential



4. Leaf level: evaluate local-field potential at particles



5. Leaf level: direct evalution in the near-field

FIG. 3.2. *Computational structure of a generic $O(N)$ hierarchical N–body method.*

$N : 2^i$) (see Figure 4.1). The embedding preserves locality between a subdomain and its descendants in the hierarchy. If at some level, there is at least one subdomain per processor, then all descendants for each such subdomain are allocated to the same processor as the subdomain itself.

The 5–D array representation is quite effective with respect to memory utilization, yet guarantees locality in traversing the hierarchy. The 5–D array representation is easy to use for any depth of the hierarchy; only the extent of the three spatial axes depend on the depth of the hierarchy. Representing each level of the hierarchy as a separate array can clearly be made more memory efficient, but the number of arrays depends on the depth of the hierarchy. Adjusting the number of arrays allocated according to the depth of the hierarchy at run–time is a difficult problem in Fortran and HPF since arrays have to be named at programming time. Using

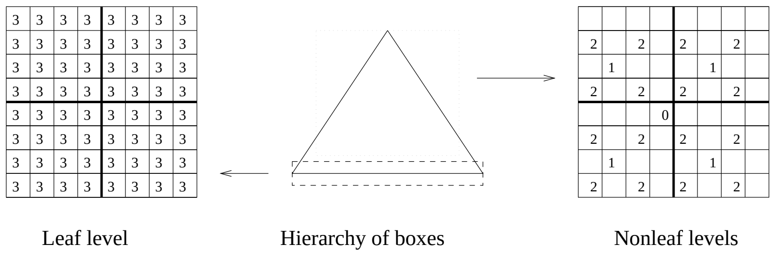


FIG. 4.1. *Representation of persistent hierarchical data in a 5–D array.*

axis	extent	processor address $b_{p+n-1}b_{p+n-2}...b_n$			local memory address $b_{n-1}b_{n-2}...b_0$				
0	2				1				
1	`K`					b..b			
2	`L`	b..b					b..b		
3	`M`		b..b					b..b	
4	`N`			b..b					b..b

FIG. 4.2. *The allocation of the* `FAR_POT` *array.*

arrays with one of the axis representing the levels of the hierarchy would require support for ragged arrays for space efficiency. But, ragged arrays are neither supported in HPF nor in CMF.

The (default) allocation of the array FAR_POT to local memory and processor addresses is illustrated in Figure 4.2. First, we note that the extent of the leaf–level subdomain axes, `L`,`M`, and `N`, are powers–of–two. Second, on the Connection Machine systems, the number of processors assigned to a task is also a power–of–two. Moreover, the Connection Machine run–time system as a default attempts to factor the set of processors assigned to a task such that the set of subdomains assigned to a processor has as small a surface area as possible, within the constraint that the number of processors assigned to an array axis is also a power–of–two. Thus, the allocation of parallel array axes can be described entirely in terms of address bits. Local axes, declared as `SERIAL`, do not allocate local memory in powers–of–two.

5. Optimizing communication.

In this section we present four constructs using array aliasing to reduce (minimize) communication. The four constructs are

- comunication in a global address space, illustrated through the CSHIFT intrinsic,
- assignments between nonconforming arrays in the form of *multigrid embed* and *multigrid–extract*,
- array reshaping to allow nonconforming arrays to conform without communication or memory growth, illustrated through the re-

shaping of 1–D particle arrays to conform with arrays for leaf–level subdomain data,

- exploiting symmetry in all–to–all communication, illustrated for the direct (all pairs) N–body algrorithm.

5.1. Avoiding excessive data motion in global communications. Many compilers, in particular in their early stages of development, assume a fixed mapping between the index space and memory addresses, rather than reevaluating the index map according to specified communication actions. For computations that exhibit locality of reference in the physical space, properly identifying the part of the address space local to processors combined with the use of "ghost regions" with respect to the specified computations on the local address space, can significantly reduce both the number of communication actions and the amount of data moved between processors as well as within local memory. The technique described below attempts to minimize the number of shift operations and the amount of data moved in the followed assignments through array sectioning on aliased arrays.

As an example of how data motion can be reduced through the explicit specification of local and global address spaces in data parallel programming, we consider the gathering of data from neighboring subdomains through the use of CSHIFT, a common communication primitive in implementing convolution operations. Such operations tend to occur in inner loops in the implementation of finite difference methods and in signal processing applications. In the nonadaptive N–body methods, the interactions conveniently specified through CSHIFT imply a "stencil"that involves up to 875 subdomains in 3–D (the domains marked i in Figure 3.1).

In the context of the three–dimensional arrays of our N–body example, we assume that the shape of the leaf–level subdomains assigned to each processor is $S1 \times S2 \times S3$, with the parallel extents being $P1 = L/S1$, $P2 = M/S2$, and $P3 = N/S3$, respectively. Explicitly identifying the subdomains local to processors through the aliasing mechanism can be made as follows:

```
      REAL*8 POT(K,L,M,N)
CMF$LAYOUT POT(:SERIAL,:,:,:)
      REAL*8 POT_ALIAS(K,S1,S2,S3,P1,P2,P3)
CMF$LAYOUT POT_ALIAS(:SERIAL,:SERIAL,:SERIAL,:SERIAL,:,:,:)
```

With a ghost region d subdomains deep on all sides of the subdomain of shape $S1 \times S2 \times S3$, a total of $(S1 + 2d) \cdot (S2 + 2d) \cdot (S3 + 2d) - S1 \cdot S2 \cdot S3$ subdomains must be fetched. Fetching them all individually incurs an unnecessarily high communication overhead, in particular when general communication primitives or CSHIFTs with large offsets must be used for the individual fetches. Instead of fetching individual subdomains through independent shift operations, we gather all subdomains required in a certain direction through a single shift of the maximum distance required.

Combining this fairly obvious idea with assignment on aliased arrays yields high efficiency in gathering the subdomains in the ghost region.

The ghost region consists of 26 subregions: six regions adjacent to the six faces of the subdomains local to each processor, 12 regions adjacent to the edges of the local subdomain, and eight regions adjacent to the corners. The regions adjacent to corners are of shape $d \times d \times d$. For the regions adjacent to edges, four are of shape $S1 \times d \times d$, four of shape $S2 \times d \times d$, and four of shape $S3 \times d \times d$. Note that each subregion may cover more than one processor when $d > min(S1, S2, S3)$. In fact, the entire ghost region cover $(1+2 \cdot \lceil \frac{d}{S1} \rceil) \cdot (1+2 \cdot \lceil \frac{d}{S2} \rceil) \cdot (1+2 \cdot \lceil \frac{d}{S3} \rceil)$ processors. Fetching the 26 subregions may in fact be executed as $54 + (2 \cdot \lceil \frac{d}{S1} \rceil - 1) \cdot (2 \cdot \lceil \frac{d}{S2} \rceil - 1) \cdot (2 \cdot \lceil \frac{d}{s3} \rceil - 1)$ CSHIFTs, each parallel to an axis. For $d < min(S1, S2, S3)$, this amounts to 54 CSHIFTs, since fetching each of the 12 edge regions requires two axis–parallel CSHIFTs, while fetching each of the eight corner regions requires three axis–parallel CSHIFTs. Since the mechanism used for the implementation of CSHIFTs on many architectures is more efficient than general communication, converting CSHIFTs not parallel to an axis into a sequence of axis–parallel shifts may in fact yield better performance than invoking general communication facilities. The conversion is usually made by the run–time system and not under user control.

The interprocessor data motion in the above approach is minimal, since only data in ghost region are actually fetched.

An alternative to fetching the ghost region through the procedure just described is to create a linear ordering through the $(1+2 \cdot \lceil \frac{d}{S1} \rceil) \cdot (1+2 \cdot \lceil \frac{d}{S2} \rceil) \cdot (1+2 \cdot \lceil \frac{d}{S3} \rceil) - 1$ processors covered by the 26 subregions using the notion of a space filling curve. Using this linear ordering, $(1+2 \cdot \lceil \frac{d}{S1} \rceil) \cdot (1+2 \cdot \lceil \frac{d}{S2} \rceil) \cdot (1+2 \cdot \lceil \frac{d}{S3} \rceil)$ CSHIFTs of unit distance along one of the three processor axes would suffice. For $d < min(S1, S2, S3)$, this number is 26. In practice, if $2d < min(S1, S2, S3)$, as is often the case in the neighbor interactions in large–scale N–body simulations, the number of CSHIFTs can be further reduced to $3 + \lceil \frac{S1+2d}{S1} \rceil \cdot \lceil \frac{S2+2d}{S2} \rceil \cdot \lceil \frac{S3+2d}{S3} \rceil$ axis–parallel CSHIFTs with no increase in the amount of data being moved. Three of the CSHIFTs are used to align the upper–left corner of the ghost region with that of the $S1 \times S2 \times S3$ subgrid (of local subdomains) of a processor. These CSHIFTs are performed on the unaliased array with the offsets being d, since the offset d may not be a multiple of $S1, S2$, or $S3$. After this alignment, the entire ghost region resides in $\lceil \frac{S1+2d}{S1} \rceil \cdot \lceil \frac{S2+2d}{S2} \rceil \cdot \lceil \frac{S3+2d}{S3} \rceil$ processors, with one of the processors being the target processor. Therefore, the total number of additional CSHIFTs equals the number of processors over which the ghost region is distributed after the alignment less one. For $2d < min(S1, S2, S3)$, the number of CSHIFTs after alignment is seven for a total of 10. This implementation is illustrated below in which all CSHIFTs are performed on the orignal arrays, but all assignments make use of array sectioning on aliased arrays to extract and assign data from the ghost region. The array NBR_POT is used to store the local subdomains and its ghost region.

```
      REAL*8 POT(K,L,M,N)
CMF$LAYOUT POT(:SERIAL,:,:,:)
      REAL*8 POT_ALIAS(K,S1,S2,S3,P1,P2,P3)
CMF$LAYOUT POT_ALIAS(:SERIAL,:SERIAL,:SERIAL,:SERIAL,:,:,:)
      REAL*8 NBR_POT(K,S1+2d,S2+2d,S3+2d,P1,P2,P3)
CMF$LAYOUT NBR_POT(:SERIAL,:SERIAL,:SERIAL,:SERIAL,:,:,:)
      INTEGER I,J,K,II,JJ,KK
      INTEGER OFF1,OFF2,OFF3, INDEX,C1,C2,C3,DISP,D1,D2,D3

C     Alignment of ghost region with the subgrid
      POT = CSHIFT(POT,2,-D)
      POT = CSHIFT(POT,3,-D)
      POT = CSHIFT(POT,4,-D)

C     calculate how many processors the aligned ghost region cover

      C1 = (S1+2*D-1)/S1+1
      C2 = (S2+2*D-1)/S2+1
      C3 = (S3+2*D-1)/S3+1
      D1 = MOD(S1+2*D-1,S1)+1
      D2 = MOD(S2+2*D-1,S2)+1
      D3 = MOD(S3+2*D-1,S3)+1
      OFF1 = S1
      OFF2 = S2
      OFF3 = S3

      DO DISP = 0,C1*C2*C3-1

C     a axis-parallel CSHIFT along a simple space filling curve --
C     the snake ordering
         I = DISP/(C2*C3)
         INDEX = MOD(DISP,C2*C3)
         J = INDEX/C3
         IF (MOD(I,2) .EQ. 1) J = C2-J-1
         K = MOD(INDEX,C3)
         IF (MOD((J+I),2) .EQ. 1) K = C3-1-K

         IF (DISP .GT. 0) THEN
            IF (MOD(DISP,C2*C3) .EQ. 0) THEN
               IF (I .EQ. C1-1) THEN
                  POT = CSHIFT(POT,2,D1)
               ELSE
                  POT = CSHIFT(POT,2,OFF1)
               END IF
               OFF2 = -OFF2
               OFF3 = -OFF3
            ELSE IF (MOD(DISP,C3) .EQ. 0) THEN
               IF (J .EQ. C2-1) THEN
                  POT = CSHIFT(POT,3,D2)
               ELSE IF ((J .EQ. C2-2) .AND. (OFF2 .LT. 0)) THEN
                  POT = CSHIFT(POT,3,-D2)
               ELSE
                  POT = CSHIFT(POT,3,OFF2)
               END IF
               OFF3 = -OFF3
            ELSE
               IF (K .EQ. C3-1) THEN
                  POT = CSHIFT(POT,4,D3)
               ELSE IF ((K .EQ. C3-2) .AND. (OFF3 .LT. 0)) THEN
                  POT = CSHIFT(POT,4,-D3)
               ELSE
                  POT = CSHIFT(POT,4,OFF3)
               END IF
            END IF
         END IF

C     array sectioning on local axes

         IF ((I .EQ. C1-1) .AND. (J .EQ. C2-1) .AND.
     $        (K .EQ. C3-1)) THEN
           NBR_POT(:,I*S1+1:S1+2*D,J*S2+1:S2+2*D,K*S3+1:S3+2*D,:,:,:)=
     $           POT_ALIAS(:,S1+1-D1:S1,S2+1-D2:S2,S3+1-D3:S3,:,:,:)
         ELSE IF ((I .EQ. C1-1) .AND. (J .EQ. C2-1)) THEN
           NBR_POT(:,I*S1+1:S1+2*D,J*S2+1:S2+2*D,K*S3+1:(K+1)*S3,:,:,:)
     $           = POT_ALIAS(:,S1+1-D1:S1,S2+1-D2:S2,:,:,:,:)
         ELSE IF ((I .EQ. C1-1) .AND. (K .EQ. C3-1)) THEN
           NBR_POT(:,I*S1+1:S1+2*D,J*S2+1:(J+1)*S2,K*S3+1:S3+2*D,:,:,:)
     $           = POT_ALIAS(:,S1+1-D1:S1,:,S3+1-D3:S3,:,:,:)
         ELSE IF ((J .EQ. C2-1) .AND. (K .EQ. C3-1)) THEN
           NBR_POT(:,I*S1+1:(I+1)*S1,J*S2+1:S2+2*D,K*S3+1:S3+2*D,:,:,:)
     $           = POT_ALIAS(:,:,S2+1-D2:S2,S3+1-D3:S3,:,:,:)
         ELSE IF (I .EQ. C1-1) THEN
           NBR_POT(:,I*S1+1:S1+2*D,J*S2+1:(J+1)*S2,K*S3+1:(K+1)*S3,
     $        :,:,:) = POT_ALIAS(:,S1+1-D1:S1,:,:,:,:,:)
         ELSE IF (J .EQ. C2-1) THEN
           NBR_POT(:,I*S1+1:(I+1)*S1,J*S2+1:S2+2*D,K*S3+1:(K+1)*S3,
     $        :,:,:) = POT_ALIAS(:,:,S2+1-D2:S2,:,:,:,:)
         ELSE IF (K .EQ. C3-1) THEN
           NBR_POT(:,I*S1+1:(I+1)*S1,J*S2+1:(J+1)*S2,K*S3+1:S3+2*D,
     $        :,:,:) = POT_ALIAS(:,:,:,S3+1-D3:S3,:,:,:)
         ELSE
           NBR_POT(:,I*S1+1:(I+1)*S1,J*S2+1:(J+1)*S2,K*S3+1:(K+1)*S3,
     $           :,:,:) = POT_ALIAS
         END IF
      ENDDO
```

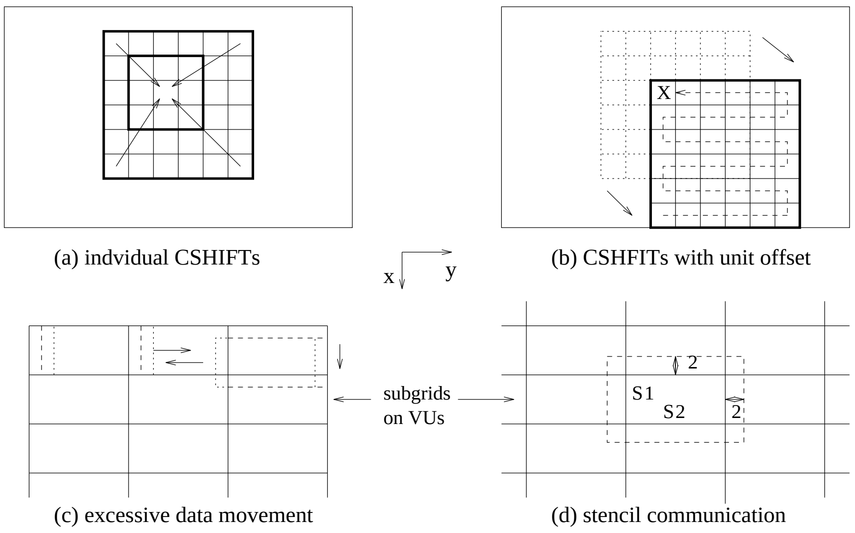

X
(a) indvidual CSHIFTs
(b) CSHFITs with unit offset
x
y
subgrids on VUs
S1
S2
2
2
(c) excessive data movement
(d) stencil communication

FIG. 5.1. *Optimizing communication in fetching ghost regions through the use of space filling curves. Examples are in 2–D for clarity of presentation.*

Note that, excluding the first three CSHIFTs used for the alignment, the interprocessor data motion in the above approach is also minimal.

We implemented both the above approaches. We also applied the two approaches naively to individual subdomains without aliasing and blocking with respect to the depth of the ghost region, as shown in Figure 5.1 (a) and (b). Both naive approaches involve excess data motion, since the approach in (a) requires CSHIFTs with the largest offsets while the approach in (b) moves arrays back and forth as shown in Figure 5.1 (c). In both cases, a processor in fact receives some of the data more than once.

For the hierarchical N–body computations in 3–D, all subdomains marked i in Figure 5.2 interact with the subdomain marked b for the so–called interactive–field to local–field conversion. Accounting for all 10 relevant planes with respect to subdomain b yields a total of 875 subdomains marked i. For a subdomain b on the boundary of the $S1 \times S2 \times S3$ subdomains local to a processor, the i–marked subdomain furthest away is at distance four. Thus, for our example $d = 4$. Table 5.1 summarizes the data motion requirements for the four methods for $S1 = S2 = S3 = 8$. In this Table, K refers to the number of elements fetched for each subdomain.

From Table 5.1, it is clear that our most effective alternative is more than two orders of magnitude faster than the most straightforward implemention of fetching subdomains in the ghost region. Using array alising, minimizing the number of shift operations using a space filling curve in the ghost region improved the performance on the Connection Machine CM–5 by about 50% compared to direct fetches. We expect the relative perfor-

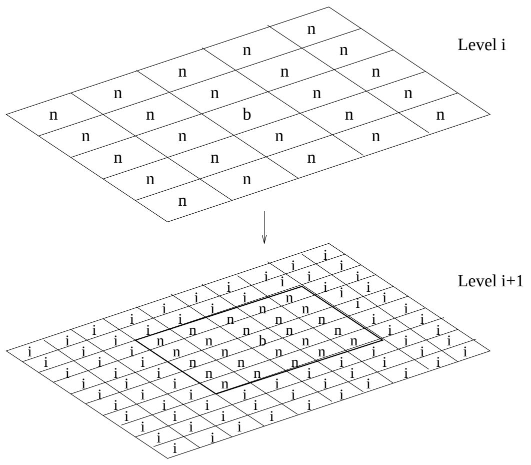


FIG. 5.2. *A plane through the subdomains that must be fetched in interactive–field to local–field conversion in hierarchical N–body methods in three dimensions.*

TABLE 5.1
Comparison of data motion needs for the fetching of a ghost region four subdomains deep on all sides of an $8 \times 8 \times 8$ *subgrid assigned to each processor on a 32 processor CM–5E. The local region with its ghost region is stored in a local* $16 \times 16 \times 16$ *aliased array.*

Method	Number of non–local boxes fetched	Number of local box moves	Number of CSHIFTS	Relative time	
				$K = 12$	$K = 72$
Direct on unaliased arrays	696,960	1,161,600	1,206	169	178
Linearized unaliased arrays	85,888	596,608	1,333	19.4	18.2
Direct on aliased arrays	3,584	7,680	54	1.63	1.48
Linearized aliased arrays	4,352	6,400	10	1	1

mance of the four alternatives for fetching variables for the ghost region to be similar on other architectures.

Increased granularity of the local subdomains, i.e., increased values of $S1$, $S2$ and $S3$, does not affect the number of subregions in the ghost region, only their sizes, except for the corner regions that remain fixed for a fixed d. Hence, we expect the relative merits of the four alternatives to be preserved for increased granularity of local subdomains. In fact, we expect the relative effectiveness of the two alternatives using aliasing to increase with increased size of local subdomains. The scalability with respect to machine size is somewhat harder to assess in that even though ghost regions are adjacent in the physical domain they are not necessarily adjacent or even "local" with respect to processors. Hence, global aspects of the communication system may be important. However, for a fixed size of the subdomains mapped to processors, i.e., increasing the total number of subdomains along with the

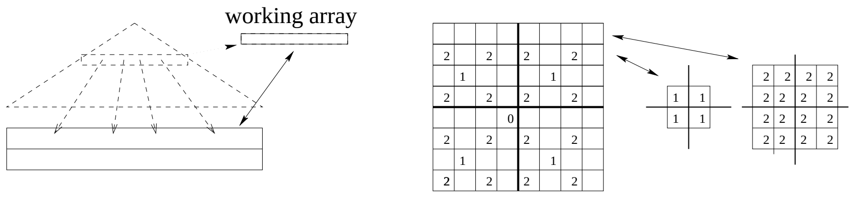


FIG. 5.3. `Multigrid-embed/multigrid-extract` *of subdomain variables between congruent small and large arrays.*

number of processors, the amount of communication per processor remains fixed. Hence, if the average communication distance for communication between adjacent subdomains from the mapping of the physical address space to the processor address space employed by the compiler/run–time system does not increase, then bandwidth requirement per processor does not increase either. Thus, for such communication networks and address space mappings, scalability with respect to machine size should be good, and the relative order between the alternatives preserved.

5.2. Multigrid–embed and Multigrid–extract. In hierarchical methods, interactions between congruent arrays of different size occur frequently in proceeding to coarser or finer domain decompositions. In our implementation of the hierarchical N–body methods, we use working arrays of sizes that correspond to the current levels of the hierarchy, while a permanent 5–D array of a shape determined by the leaf–level domain decomposition is used for storage of persistent data, as illustrated in Figure 5.3 and described in Section 4. The computations require frequent interactions between the smaller working arrays, with sizes dependent upon hierarchical level, and the permanent "full resolution" array.

If the elements of the working array and the corresponding elements of the permanent array are assigned to the same processor, then in fact all that is required for exchange of data is a local memory copy. Unfortunately, it is not possible to know at compile time whether or not two corresponding elements in the two arrays are assigned to the same processor, since the number of processors used is a run–time variable, and so are in most cases also the number of leaf–level subdomains, and thus the extents of the parallel axes of the permanent 5–D array. At compile time, it is necessary to assume that assignments between the working and permanent arrays be performed using a general communications mechanism. At run–time, some of these general communications may be converted into local copy operations. The automatic detection and resolution of what type of communication action is needed, requires fairly sophisticated run–time system optimizations, and is beyond the capabilities of any current run–time system for distributed memory architectures. However, if the programmer explicitly identifies that the assignment is between array elements assigned to the same processor, then run–time systems with a much lower degree of optimization may still be able to carry out the assignment using local memory copy instead of a general communications mechanism. This is the case for the CM–5 run–time system. The array aliasing mechanism can be used by the programmer to identify that an assignment is in fact between array elements allocated to the same processor.

We now illustrate how to use array aliasing to identify that the assignment of variables from a working array into a larger permament array in fact is a local memory copy in all processors. A straightforward CMF expression for a `Multigrid-embed` operation at hierarchical level $(h - i)$ is

```
      FAR_POT(2,:,2**(I-1):L:2**I,2**(I-1):M:2**I,2**(I-1):N:2**I)
     &        = TMP.
```

This code fragment would cause the compiler to invoke general communication as explained above. However, if the array `TMP`, which stores the variables for the subdomains at level i of the hierarchy, has at least one subdomain per processor, then the general communication is avoided by ideintifying the assignment as local copying as illustrated in the code fragment below:

```
      REAL*8 FAR_POT_ALIAS(2,K,S1,S2,S3,P1,P2,P3)
CMF$LAYOUT FAR_POT_ALIAS(:SERIAL,:SERIAL,:SERIAL,:SERIAL,:SERIAL,:,:,:)
      REAL*8 TMP_ALIAS(K,R1,R2,R3,P1,P2,P3)
CMF$LAYOUT TMP_ALIAS(:SERIAL,:SERIAL,:SERIAL,:SERIAL,:,:,:)

      IBGN = 2**(I-1)
      ISTD = 2**I
      FAR_POT_ALIAS(2,:,IBGN:S1:ISTD,IBGN:S2:ISTD,IBGN:S3:ISTD,:,:,:)
     &     = TMP_ALIAS
```

In the above code, an alias is created for the two arrays to separate their local addresses from their processor addresses. Array sectioning is then performed on the local axes and general communication is avoided.

If the array `TMP` corresponds to a level of the hierarchy that has fewer subdomains than the number of processors, and therefore the elements of the array `TMP` cannot be guaranteed to be assigned to the same processors as the corresponding elements of the permanent array, then `Multigrid-embed` requires interprocessor communication. This lack of alignment is typically the case, since arrays of a size smaller than the number of processors typically are allocated to processor addresses contiguosly. This form of allocation is used on the CM–5. Whenever `TMP` has fewer subdomains than the number of processors, introducing a second temporary array, `TMP2`, congruent to `TMP` and the permanent array, and with one element per processor, may improve the performance. Assignments between `TMP2` and the permanent array can be carried out as described above, while the assignments between `TMP` and `TMP2` require interprocessor communication for at least some elements. However, this communication is much more efficient than

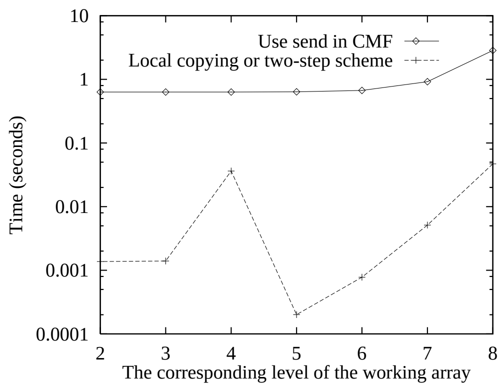


FIG. 5.4. *Performance comparison of* `Multigrid-embed` *implementations with and without array aliasing for a depth–eight 3–D hierarchy on a 256 processor CM-5E. The two–step scheme was used for the first two cases and the remaining cases used only local copying.*

the assignment between `TMP` and the permanent array, since in general, `TMP2` is much smaller than the permanent array. Array sectioning can be used to specify the assignment between `TMP` and `TMP2`, but any syntactically correct form of assignment can be used as well.

On the CM–5E, the performance improvement achieved through the techniques above for `Multigrid-embed` was as much as a factor of a hundred or better, as shown in Figure 5.4.

5.3. Alignment of nonconforming arrays through reshaping. Many operations in the data parallel programming model are only valid on conforming arrays. Thus, reshaping interacting arrays such that they are conforming is sometimes necessary, and often very desirable for good performance. If the reshaping can be accomplished with no communication, then clearly there may be a substantial benefit. The example we use from our N–body code is the alignment of 1–D particle arrays with the 5–D arrays for hierarchical data, without any growth in memory requirements for uniform particle dsitributions. Communication is required in organizing the particle data for locality of reference with respect to the leaf–level subdomains to which they belong, but no communication is required for the actual reshaping.

The particle data for N–body codes are frequently given as a collection of 1–D arrays; one array for each particle attribute, such as charge, mass, velocity and coordinates. In the hierarchical N–body methods, the particle data are used in establishing leaf–level subdomain representations prior to the upward pass through the hierarchy (see Figure 3.2). Interaction between the hierarchical domain decomposition and the particle data occurs again after the downward traversal of the hierarchy. In order to maximize the locality in particle and leaf–level subdomain interactions, it is desir-

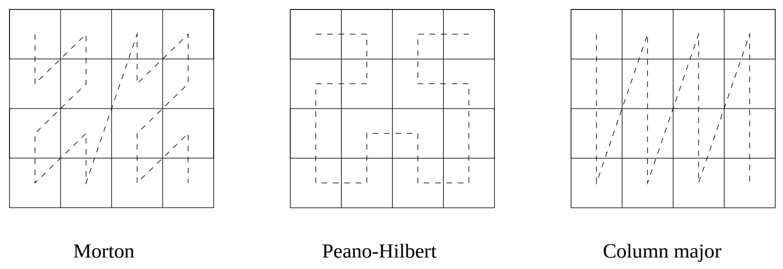


FIG. 5.5. *Construction of sorting keys for particle alignment with some useful orderings of subdomains.*

able to allocate particles to the same processor as the leaf–level subdomain to which they belong. To accomplish this task, the particles are first reordered such that they are ordered in the same way across processors as the leaf–level subdomains are ordered across processors. Then, reshaping the 1–D particle arrays to 4–D arrays with three parallel axis of the same extent as the three parallel axis of the permament 5–D array yields the desired result. Using array aliasing the reshaping requires no communication. Below we described a particle reordering and array reshaping that result in collocation of leaf–level subdomains and their particles for uniform distributions.

The leaf–level subdomain to which a particle belongs is determined by the coordinates of the leaf–level subdomain and the particle. Hierarchical subdomain orderings, such as Morton [8] or Peano–Hilbert [5], shown in Figure 5.5, allow one sorted order to be used for any depth of the hierarchical domain decomposition with the ordering properties preserved from level to level. The Morton ordering is achieved by constructing keys for sorting the particles by interleaving the bits of the particle coordinates. The Peano–Hilbert ordering is achieved by constructing keys recursively. Assume the addresses of the particle coordinates are $x_n ... x_1 x_0$, $y_n ... y_1 y_0$, and $z_n ... z_1 z_0$, respectively. First, the Gray code [7] of $x_n y_n z_n$ is generated as the three leading bits of the key $k_{3n+2} k_{3n+1} k_{3n}$. Second, the ith three bits of key $k_{3i+2} k_{3i+1} k_{3i}$ determine how the $(i-1)$th bits of the three axes x_{i-1}, y_{i-1} and z_{i-1} should be encoded to form the $(i-1)$th three bits of the key $k_{3i-1} k_{3i-2} k_{3i-3}$. If the ordering of the subdomains is row or column major, as in the typical linearization of array addresses used in common programming languages, then resorting is required for each depth of the hierarchy.

For the row or column major ordering of subdomains, the keys for sorting particles are constructed by partitioning each axis coordinate for a particle into two parts, one corresponding to the axis coordinate of the subdomain to which it belongs, and the other corresponding to its relative location within the subdomain. A row or column major ordering is then created for the particle axes coordinates corresponding to the subdomain

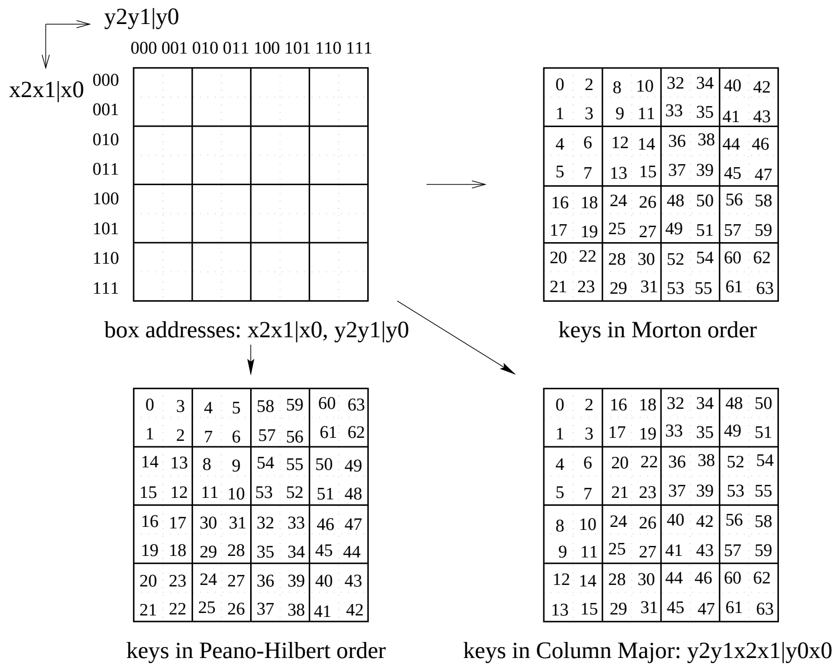

y2y1|y0
000 001 010 011 100 101 110 111
x2x1|x0
000
001
010
011
100
101
110
111
box addresses: x2x1|x0, y2y1|y0
keys in Morton order
keys in Peano-Hilbert order
keys in Column Major: y2y1x2x1|y0x0

FIG. 5.6. *Sorting particles for maximum locality in reshaping particle arrays.*

coordinates. The part representing the relative location within a subdomain may be in arbitrary order, but could also use a row or column major ordering, as is illustrated in Figure 5.5.

Figure 5.6 illustrates three different orderings of 16 2–D subdomains each with four particles.

For the Connection Machine implementation for which performance data are reported in Section 7, the default run–time system ordering of the subdomains is column major order, with an optional row major ordering. We used the default ordering in our benchmarks. We illustrate the steps of our column major sort below:

Algorithm: (Column major sort)

1. Find the layout of the parallel axes of the 5–D arrays storing hierarchical subdomain data using the intrinsic mapping functions,
2. For each particle, generate the coordinates of the subdomain to which it belongs, denoted by $xx..x$, $yy..y$, and $zz...z$;
3. Split the subdomain coordinates into a processor address and a local memory address, written as $x..x|x..x$, $y..y|y..y$, $z..z|z..z$, according to the layout of the 5–D arrays;
4. Form keys for sorting by concatenating the processor addresses with the local memory addresses, written as $z..zy..yx..x|z..zy..yx..x$;
5. Sort.

If there is at least one subdomain per processor, then for a uniform particle distribution, each particle in the sorted 1–D particle array will be allocated to the same processor as the data of the leaf–level subdomain to which it belongs. The leaf–level subdomain data is contained in the 5–D array representation of hierarchical subdomain data.

The reshaping of the 1–D array to a 4–D array with three parallel axes such that the shape of its parallel axes conform with the shape of the three parallel axes of the 5–D array can be straightforwardly achieved using array aliasing, and no communication will be required. For a near–uniform particle distribution, it is expected that the coordinate sort will leave most particles in the same processor memory as the leaf–level boxes to which they belong. But the reshaping has to be expressed using gather operations, as shown in the following code fragment:

```
C     START_PTR(i,j,k) stored the location of the first particle
C     belonging to subdomain (i,j,k) in the 1-D particle array.

      INTEGER NUM_PTCLPERBOX(L,M,N), START_PTR(L,M,N)
      LOGICAL MASK(L,M,N)
CMF$LAYOUT NUM_PTCLPERBOX(:,:,:)
CMF$LAYOUT START_PTR(:,:,:)
CMF$LAYOUT MASK(:,:,:)

      DO II = MIN_PTCLPERBOX, MAX_PTCLPERBOX-1
         MASK =  (II < NUM_PTCLPERBOX)

         FORALL(I=1:L, J=1:M, K=1:N, MASK(I,J,K))
     $         X_4D(II+1,I,J,K) = X(START_PTR(I,J,K)+II)
      ENDDO
```

5.4. Exploiting symmetry in all–to–all communication. All–to–all communication [6] is a critical operation in several applications, such as the naive (direct) N–body algorithm, but also in molecular dynamics computations where a cut–off radius is used. In such computations, all–to–all interactions take place between all molecules within the cut–off radius of each molecule. All–to–all communication is also required within local regions in the hierarchical N–body methods, as defined by the so–called *near–field*. This field contains all particles too close to apply the approximations on which the hierarchical field evaluation is based.

Using the reshaped 4–D array particle representation described above, the all–to–all communication can be carried out based on the use of space–filling curves, analogously to the use of such curves in fetching ghost regions. Exploiting symmetry in the all–to–all interaction is fairly straightforward based on the linear ordering offered by the space–filling curve. The observation has been made by many others, see for instance [2]. The idea of reducing communication by exploiting symmetry is shown in a 2–D example in Figure 5.7. As subdomain 0 traverses subdomains 1–4, the interactions

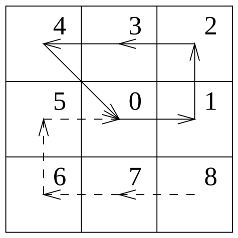


FIG. 5.7. *Exploiting symmetry in all–to–all comunication.*

between subdomain 0 and each of the four subdomains will be computed. The contributions from the four subdomains to subdomain 0 are accumulated and communicated along with subdomain 0. Using data parallel programming, subdomains 5–8 will traverse subdomain 0 while subdomain 0 traverses subdomains 1–4. The contributions from subdomains 5 – 8 to subdomain 0 are accumulated and stored in subdomain 0. Finally, the two contributions to subdomain 0 are combined.

6. Optimizing computation.

The optimizations presented in this section focus on two issues:

- carrying out operations on nonconforming arrays without growth in memory requirements and without invoking general communication,
- improving the performance of large numbers of BLAS type of operations applied to small operands.

The first situation occurs frequently when a given function is applied many times, and at least one of the operands is shared between many applications of the function. An example is a multivalued "constant coefficient" convolution kernel. The second situation is also very common in many scientific and engineering codes. For instance, the simulation of fluid flow using a finite difference or finite volume approach involves inner products or matrix–vector multiplications in each grid point, or for each finite volume.

For our reference application, the hierarchical N–body method due to Anderson, the first type of situation occurs frequently in the hierarchical computations. In fact only eight distinct matrices are required for parent–child domain interactions for all such interactions in 3–D, while a total of 1206 matrices are required for interactive–field computations regardless of the number of subdomains, and independent of the level to which the comnputations are applied[2]. With respect to the second case, level–1 BLAS can be aggregated into level–3 BLAS, that can be further aggregated into *multiple–instance* level–3 BLAS. Multiple–instance BLAS is supported in the Connection Machine Scientific Software Library, CMSSL [10]. The ag-

[2] The 1206 matrices correspond to the 1206 possible relative interactive–field box locations with offsets in $[-5,5]\times[-5,5]\times[-5,5]\backslash[-2,2]\times[-2,2]\times[-2,2]$.

gregation from level–1 to level–2 BLAS is made in the problem formulation, while the aggregation into multiple–instance level–3 BLAS is carried out either through aliasing alone, or aliasing in combination with copying of some local arrays. The copy operation allow for operands of increased size for the level–3 BLAS.

The benefits of higher level BLAS compared to lower level ones is that performance may be improved through a higher degree of locality of reference. For instance, an inner product allows two floating–point operations to be carried out for each pair of elements fetched from memory, with an additional memory reference required to store the result. A matrix–vector multiplication allows for two floating–point operations for each matrix elements fetched. Additional memory references are required for the input vector, and for storing the result vector. A matrix–matrix multiplication performed on $b \times b$ blocks allow $2b^3$ floating–point operations to be performed for each $2b^2$ elements fetched with additional memory reference required to store a $b \times b$ result block for every $2N/b$ blocks read and $2b^2N$ floating–point operations performed. Thus, for these operations, an inner product (level–1 BLAS) approaches one floating–point operation per memory reference as operand size grows, a matrix–vector multiplication (level–2 BLAS) approaches two floating–point operations per memory reference, while a matrix–matrix multiplication (level–3 BLAS) approaches b floating–point operations per memory reference. Since memory bandwidth on most processor architectures is a limiting factor it is highly desirable to use higher level BLAS whenever possible.

Another factor often at least as important for performance is the memory access pattern. Access patterns that lead to frequent or consistent cache misses, TLB misses, or DRAM page faults may have severe performance impact, even when all data is in primary memory. If the array layout in memory was known at compile time, and if all operations were expressed in loop nests or straight–line code or some combination thereof in the source code, then an optimizing compiler may successfully handle the relevant loop ordering and blocking required for peak performance. However, if the array layout is not known at compile time, and if subroutine calls are used for some of the functions, then in addition to the unknown memory layout, the optimization of memory accesses is further complicated by the need for interprocedural analysis. In practice, the situation may be even more difficult through the use of commercial libraries often available only as binaries. In addition to fewer memory references per floating–point operation, the higher–level BLAS have the benefits to offer more choices for memory accesses to the implementer of the function. For instance, an inner product is defined through a single loop on two one–dimensional arrays, while matrix–matrix multiplication requires three loops and involves three two–dimensional arrays.

The multiple–instance feature provides at least one loop variable in addition to the loops required for a single–instance BLAS, and thus, offers additional freedom in organizing memory accesses. The multiple–instance feature does not affect the operations count per memory reference. However, looping overhead may be reduced, for instance, if the multiple–instance feature allows for loop fusion. On the CM–5E, aggregating inner products of length 12 into matrix–vector multiplication resulted in a processor efficiency of about 36%, while the aggregation of inner products of length 72 resulted in an efficiency of 59%.

Next we present two examples of aggregation, one without copying, the other with copying. Both examples also demonstrates how nonconforming arrays can be used effectively without memory growth or inefficient memory management. Array aliasing is used to accomplish both tasks.

6.1. Aggregation of Matrix–Vector multiplication into multiple–instance Matrix–Matrix multiplication. In the example below, we assume that each processor has a collection of subdomains forming a local array of shape $S1 \times S2 \times S3$, with $S1, S2, S3 \geq 2$, as shown in Figure 6.1. Each subdomain stores a vector of length K and must perform a matrix–vector multiplication with a $K \times K$ matrix. Moreover, each subdomain uses the *same* matrix. In order to conserve storage, the $K \times K$ matrix is *not* replicated among the subdomains, but stored in a $K \times K \times P$ array with P being the number of physical processors. In addition to the challenge of aggregation, the problem as presented also hav the challenge that the two arrays of vectors (one input and one output for each subdomain), and the array of matrices are not conforming. This challenge as well as the aggregation can be handled through the aliasing feature of CMF. The loop structure is shown by the pseudo–code fragment

```
      REAL*8 MTX(K,K,P)
CMF$LAYOUT MTX(:serial,:serial,:)
      REAL*8 INPV(K,L,M,N), RESV(K,L,M,N)
CMF$LAYOUT INPV(:serial,:,:,:)
CMF$LAYOUT RESV(:serial,:,:,:)

C     Creat array alias for INPV and RESV
      REAL*8 INPV_ALIAS(K,S1,S2,S3,P), RESV_ALIAS(K,S1,S2,S3,P)
CMF$LAYOUT TMP_ALIAS(:serial,:serial,:serial,:serial,:)
CMF$LAYOUT VAL_ALIAS(:serial,:serial,:serial,:serial,:)

      DO I=1,S1,2
      DO J=1,S2, 2
      DO K=1,S3, 2
          CALL MATRIX_VECTOR_MULTIPLY(
                      RESV_ALIAS(:,I,J,K,:),
                      MTX(:,:,:),
                      INPV_ALIAS(:,I,J,K,:))
      ...
      ENDDO
```

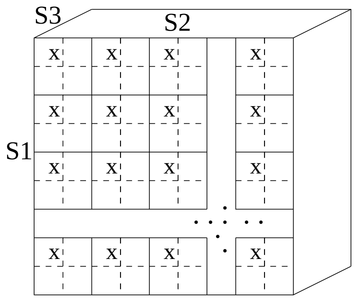


FIG. 6.1. *The subdomains assigned to a processor.*

The three loops iterate through every other subdomain along the three axes, which in fact is the requirement in the N–body application from which the example is taken. The above loop nest is embedded in another triple loop nest that implements the use of eight different matrices, one for each child of a parent subdomain. The loop body contains a call to the matrix–vector multiplication subroutine with the matrix having shape $K \times K$. Since the same translation matrix is used in the inner three loops, these loops could in principle be combined into a single matrix–matrix multiplication with one matrix of shape $K \times K$ and the other of shape $K \times \frac{S1}{2} \cdot \frac{S2}{2} \cdot \frac{S3}{2}$. However, such combining is possible through loop fusion only if the stride for the axis of length $\frac{S1}{2} \cdot \frac{S2}{2} \cdot \frac{S3}{2}$ is constant. This condition does not hold, as shown in Figure 6.2. The largest number of columns that can be treated with a fixed stride is $max(S1/2, S2/2, S3/2)$. The aggregation of the matrix–vector multiplications into multiple–instance matrix–matrix multiplication may be implemented as

```
DO I=1,S1,2
    CALL MATRIX_MULTIPLY_MI(
                RSEV_ALIAS(:,I,1:S2:2,1:S3:2,:),
                MTX(:,:,:),
                INPV_ALIAS(:,I,1:S2:2,1:S3:2,:)
...
ENDDO
```

where we assume that one of the two axes of extent $S2$ and $S3$ is used for aggregation of vectors into matrices, and the other for the multiple–instance axis. Explicit looping is still required for the axis of extent $S1$.

In aggregating matrix–vector operations into matrix–matrix operations, not only is the number of vectors being aggregated of interest, but also the stride between successive vectors, since it affects the number of cache misses, TLB misses and DRAM page faults in the multiple–instance matrix–matrix multiplication. With cubic, or close to cubic subgrids for minimum communication, either the extents of the subgrid axes are the same or they differ by a factor of two. For relatively small subgrids, the

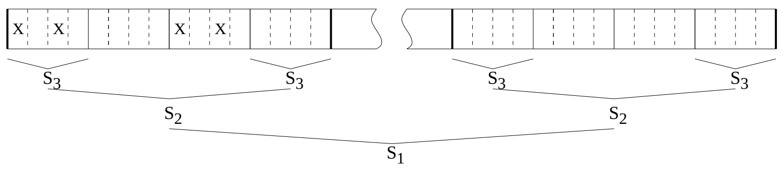


FIG. 6.2. *The layout of the subdomains in local processor memory.*

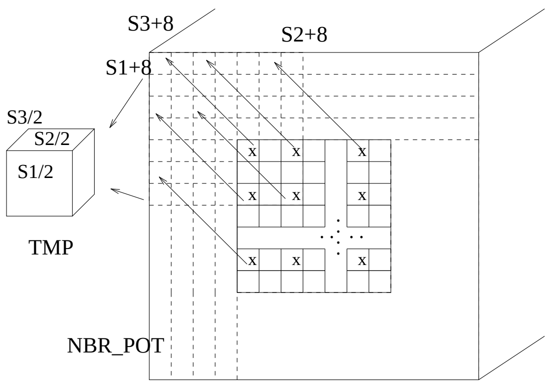


FIG. 6.3. *Aggregation of BLAS operations through copying.*

difference in size of the multiplicand due to the difference in subgrid axes extents has a larger impact on performance than DRAM page faults and TLB thrashing on CM–5. Hence, for the CM–5 we aggregate vectors into a matrix along the axis with the largest local extent. If two axes, or all three axes are of the same length, the vectors are aggregated along the axis with the largest local extent and with the smallest stride. For relatively large subgrids, vectors are aggregated along the axis with the smallest stride.

On the CM–5E, the efficiency in the above operation improved from 36% to 55% for $K = 12$ and subgrid of extents $32 \times 32 \times 16$. The matrices are of shape 12×12 and 12×8 with sixteen such instances handled in a single call. For $K = 72$ and a subgrid of extents $16 \times 16 \times 8$, the efficiency improved only a fraction of a percent and remained 60%.

6.2. Aggregating Matrix–Vector Multiplication into Matrix–Matrix Multiplication Through Copying. We also investigated the performance benefits of aggregation of matrix–vector multiplication into matrix–matrix multiplication through copying. In the context of the N–body code we used this technique for the interactive–field to local–field conversions, i.e., in handling the subdomains marked i in Figure 5.2. Recall that for these computations the local subdomains with their ghost regions are stored as one local subgrid of shape $(S1+8) \times (S2+8) \times (S3+8)$. The copying is illustrated in Figure 6.3.

The loop nests before and after copying are as follows, respectively.

Again, the loop nests are inside another triple loop nest with loop indices `II, JJ`, and `KK` which effectively loops through the eight siblings of a parent.

```
      REAL*8 MTX(K,K,P), RESV(K,S1/2,S2/2,S3/2,P)
CMF$LAYOUT MTX(:serial,:serial,:)
CMF$LAYOUT RESV(:serial,:serial,:serial,:serial,:)

C     Creat array alias for INPV
      REAL*8 INPV_ALIAS(K,S1+8,S2+8,S3+8,P)
CMF$LAYOUT INPV_ALIAS(:serial,:serial,:serial,:serial,:)

      DO I=1,S1,2
      DO J=1,S2,2
      DO K=1,S3,2
          CALL MATRIX_VECTOR_MULTIPLY(
                                        RESV(:,I/2,J/2,K/2,:),
                                        MTX(:,:,:),
                                        INPV_ALIAS(:,I+II,J+JJ,K+KK,:)
      ...
      ENDDO


      REAL*8 TMP(K,S1/2,S2/2,S3/2,P)
CMF$LAYOUT TMP(:serial,:serial,:serial,:seiral,,:)

C     Creat array alias for TMP and RESV
      REAL*8 TMP_ALIAS(K,S1*S2*S3/8,P), RESV_ALIAS(K,S1*S2*S3/8,P)
CMF$LAYOUT TMP_ALIAS(:serial,:serial,:)
CMF$LAYOUT RESV_ALIAS(:serial,:serial,:)

      TMP = INPV(:,1:S1:2,1:S2:2,1:S3:2,:,:,:)
      CALL MATRIX_MULTIPLY( RESV_ALIAS(:,:,:),
                            MTX(:,:,:),
                            TMP_ALIAS(:,:,:))
```

For $S1 = 32, S2 = 32, S3 = 16$ and $K = 12$, the efficiency of the 12×12 by 12×2048 matrix multiplication is 75%. If there are no DRAM page faults, the copying requires $2K$ cycles for each vector for which the matrix multiplication ideally takes K^2 cycles. Thus, the relative time for copying is about $2/K$. This amounts to about 17% for $K = 12$, and less than 4% for $K = 72$. With the cost of copying included, the measured efficiency decreased to 53%. For $S1 = 16, S2 = 16, S3 = 8$ and $K = 72$, the efficiency of the 72×72 by 72×256 matrix multiplication is 85%. Including the cost of copying, the measured efficiency is 78%. When the copy cost is included, the cost of copying back the result vector is insignificant since it is amortized over all 875 matrix multiplications that generate the same result vector for each subdomain in the N–body code.

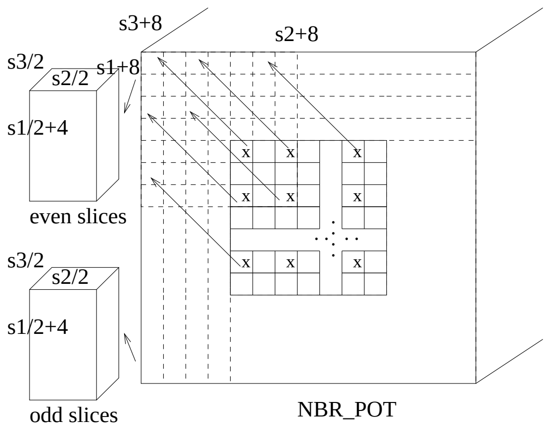


FIG. 6.4. *Reducing the cost of copying in aggregation of BLAS operations.*

If the copy back cost is amortized over fewer matrix–vector multiplications, then copy may actually decrease performance. For instance, if it is amortized over only eight multiplications, as in other parts of the N–body code, then for $K = 12$ the efficiency was reduced from the 55% mentioned in the previous subsection to 52%, while for $K = 72$ copying amortized over eight applications in fact increased the efficiency from 60% to 77%.

The copying cost can be reduced by copying a whole column block of $(S1 + 8) \times S2/2 \times S3/2$ subdomains into two linear memory blocks; one for even slices, and one for odd slices, as shown in Figure 6.4. In our application, the cost of copying is reduced to $\frac{4 \cdot 100}{875} \cdot \frac{(S1+8)}{(S1 \cdot K)}$ of that of matrix multiplication, assuming no page faults. Including the cost of copying, the efficiency reaches 60% and 80% for $K = 12$ and $K = 72$, respectively.

7. The Effectiveness of the Performance optimizations in an hierarchical N–body method.

The use of the data parallel performance techniques described here, in an implementation of the hierarchical N–body method due to Anderson resulted in an execution time of 180 seconds on a 256 processor CM–5E for the potential evaluation of 100 million interacting particles. Twelve integration points per sphere ($K = 12$) were used in this case. The overall efficiency is about 27%, and is fairly independent of machine size. With $K = 72$ integration points on the sphere, the efficiency improves to 35%.

The timings breakdown for the potential field calculation of 100 million particles on a 256 processor CM–5E is shown in Table 7.1 for $K = 12$ and $K = 72$. The hierarchy depths optimal with respect to floating–point operations, are 8 and 7, respectively. The communication time for $K = 12$ is 22.3% of the total running time and 10% for $K = 72$, demonstrating that our techniques for reducing and managing data motion are very effective. The communication time includes the time for sorting the input particles,

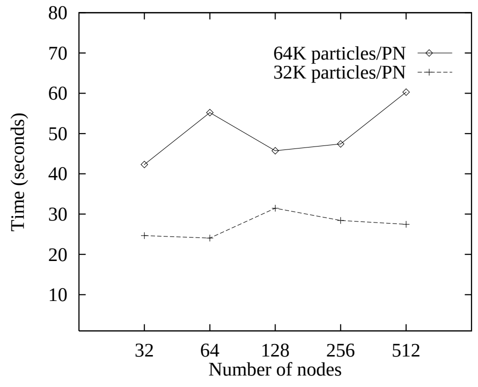


FIG. 7.1. *Scalability of our hierarchical N–body code on the CM–5.*

reshaping 1–D particle arrays to 4–D particle arrays, the multigrid functions, the fetching of ghost regions at all levels, replicating matrices at every level, and the CSHIFTs used in the direct evaluation. In the computation time we include the time for forming the far–field potential for leaf–level subdomains, all BLAS operations, the copying in the aggregation of BLAS operations for better arithmetic efficiency, the masking in distinguishing boundary subdomains from interior subdomains, the evaluation of the potential due to particles in the far–field, and finally the direct evaluation in the near–field.

Figure 7.1 shows that the speed of our code scales linearly with the number of processors and the number of particles. These timings were collected on CM–5s due to the unavailability of a variety of configurations of CM–5E systems. All cases use uniform particle distributions in a 3–D cubic domain, 12 integration points per sphere, and optimal hierarchy depths. It is clear from Figure 7.1 that for a fixed number of particles per processor, the efficiency remains independent of the number of processors. The slight fluctuation is mainly due to the fluctuation in the number of floating–point operations per particle for the optimal hierarchy depth, as shown in Figure 7.2.

8. Conclusions. In this paper, we presented a set of techniques for optimizing both communication and computation performance using the array aliasing mechanism in CMF. The array aliasing feature provides a way of separating the local memory address space from the processor address space while staying in the data parallel programming paradigm, and therefore allows the programmer to explicitly manage local and remote memory references at the data parallel language level.

The effectiveness of our techniques is demonstrated on an implementation in Connection Machine Fortran of Anderson's hierarchical $O(N)$ N–body method. CMF is the only data parallel language that has ever

Table 7.1

The breakdown of the communication and computation time for potential evaluation for 100 million particles on a 256 processor CM–5E.

Breakdown	$K = 12$		$K = 72$	
	Time (sec.)	% of total	Time (sec.)	% of total
Communication	39.75	22.3	89.01	9.99
Sort	19.60	11.0	16.04	1.80
Reshape	2.618	1.47	2.482	0.28
Upward pass – multigrid in T_{F2F}	0.107	0.06	0.092	0.01
Downward pass	8.410	4.71	56.39	6.33
– Multigrid in T_{L2L}	0.215	0.12	0.162	0.02
– Multigrid in T_{F2L}	0.484	0.27	0.385	0.04
– Fetching ghost boxes in T_{F2L}	5.160	2.89	8.610	0.97
– Replicate (T_{F2L})	2.550	1.43	47.23	5.30
Near–field – CSHIFTs	9.013	5.05	14.01	1.57
Computation	138.6	77.7	802.2	90.01
Precompute T_{F2F} matrices	0.006	0.00	0.575	0.06
Precompute T_{L2L} matrices	0.005	0.00	0.572	0.06
Precompute T_{F2L} matrices	0.003	0.00	0.235	0.03
Init–potential	2.506	1.40	14.01	1.57
Upward pass – BLAS for T_{F2F}	0.783	0.44	3.459	0.39
Downward pass	63.62	35.7	166.5	18.7
– BLAS for T_{L2L}	0.601	0.34	4.320	0.48
– BLAS for T_{F2L}	34.98	19.6	141.6	15.9
– Copy in T_{F2L}	12.90	7.23	9.990	1.12
– Masking in T_{F2L}	15.14	8.49	10.53	1.18
Far–field	4.678	2.62	90.74	10.2
Near–field – direct evaluation	65.63	36.8	525.2	58.9
Near–field – masking	1.371	0.77	0.952	0.11
Total	178.4	100	891.2	100

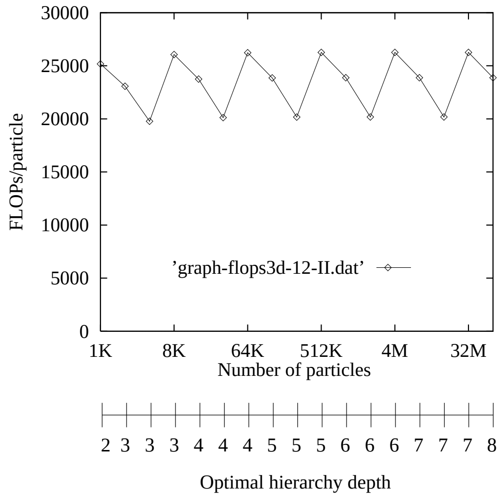


FIG. 7.2. *Floating–point operations per particle for optimal hierarchy depth, K=12.*

supported array aliasing. The overall efficiency achieved for the evaluation of the potential field of 100 million uniformly distributed particles and $K = 12$ integration points on the sphere was 27% on a 256 processor CM–5E. For $K = 72$ integration points the efficiency is about 35%.

Acknowledgment. We would like to thank Christopher Anderson for providing us with the sequential program and for many helpful discussions. We also thank the National Center for Supercomputer Applications at the University of Illinois at Urbana/Champaign, the Navy Research Laboratory Washington D.C. and the Massachusetts Institute of Technology for providing Connection Machine system CM–5/5E access.

REFERENCES

[1] C. R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comp.*, 13(4):923–947, July 1992.
[2] J. H. Applegate, M. R. Douglas, Y. Gursel, P. Hunter, C. L. Seitz, and G. J. Sussman. A digital orrery. *IEEE Trans. on Computers*, C-34(9):822–831, September 1985.
[3] H. Forum. High Performance Fortran; language specification, version 1.0. *Scientific Programming*, 2(1 - 2):1–170, 1993.
[4] L. Greengard and W. D. Gropp. A parallel version of the fast multipole method. In *Parallel Processing for Scientific Computing*, pages 213–222. SIAM, 1989.
[5] D. Hilbert. Über die stetige Abbildung einer Linie auf Flächenstück. *Math. Annln*, (38):459–460, 1891.
[6] S. L. Johnsson and C.-T. Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.
[7] E. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs. NJ, 1977.

[8] H. Samet. *Design and Analysis of Spatial Data Structures.* Addison–Wesley, 1990.
[9] Thinking Machines Corp. *CM Fortran Reference Manual, Version 2.1*, 1993.
[10] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.1*, 1993.
[11] Thinking Machines Corp. *CM Fortran Libraries Reference Manual, version 2.2*, 1994.
[12] F. Zhao. An O(N) algorithm for three–dimensional N–body simulations. Technical Report AI Memo 995, MIT, Artificial Intelligence Laboratory, October 1987.