An Evaluation of Computing Paradigms for N-body Simulations on Distributed Memory Architectures*

Collin McCurdy
Department of Computer Science
University of Wisconsin, Madison
cmccurdy@cs.wisc.edu

John Mellor-Crummey
Department of Computer Science
Rice University
johnmc@cs.rice.edu

Abstract

The efficiency of HPF with respect to irregular applications is still largely unproven. While recent work has shown that a highly irregular hierarchical n-body force calculation method *can* be implemented in HPF, we have found that the implmentation contains inefficiencies which cause it to run up to a factor of three times slower than our hand-coded, explicitly parallel implementation. Our work examines these inefficiencies, determines that most of the extra overhead is due to a single aspect of the communication strategy, and demonstrates that fixing the communication strategy can bring the overheads of the HPF application to within 25% of those of the hand-coded version.

1 Introduction

High Performance Fortran (HPF)[7, 10] was developed to simplify parallel programming by having compilers manage most of the details of application parallelization. However, it is still open to debate whether HPF and its supporting compilers will enable users to realize sophisticated parallelizations for complex problems and achieve performance comparable to hand-coded explicitly parallel programs.

In particular, there is some doubt that *irregular applications*[1] can be implemented efficiently in HPF. On message-passing parallel systems, these applications rely on runtime libraries to identify accesses to off-processor data and to coordinate data movement. To keep the overhead of this approach manageable, a good data distribution and communication aggregation are very important. While HPF generalized array assignments using *forall* and *scatter* statements support irregular data movement, the appropriateness and efficiency of such constructs is still largely unproven for full-scale irregular applications.

This paper describes a comparative study of several implementations of an irregular application for n-body simulation. All implementations use an adaptive version of Anderson's method for hierarchical approximation of far-field interactions [2]. (Section 2 provides an overview of this method.) Hierarchical methods for n-body simulation have been of interest to the computational science community not only because of their speed and accuracy, but also because their irregular structure makes efficient parallelization difficult. Hu and Johnsson developed an HPF implementation of an adaptive hierarchical solver using Anderson's method that served as the basis for much of our work [8, 9]. Their landmark implementation demonstrated that sophisticated algorithms for highly irregular problems *can* in fact be implemented in HPF. However, a performance comparison of their HPF implementation with a hand-coded, MPI-based parallel implementation that we developed exposed some costly inefficiencies in the HPF implementation that slow it's running time by as much as a factor of three.

This paper makes several contributions:

- We describe a sophisticated MPI implementation of an adaptive version of Anderson's method which integrates proven techniques to achieve good performance and scalability.
- We present a careful measurement and characterization of overhead in Hu and Johnsson's HPF implementation relative to our MPI reference implementation.

*This work has been supported by DARPA Contract DABT63-92-C-0038 and sponsored by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0159. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA and Rome Laboratory or the U.S. Government.

[1] Irregular applications contain references for which a closed form representation of the data accessed cannot be computed statically.

- We describe a modification to Hu and Johnsson's communication strategy that, when integrated into their implementation, eliminates more than 75% of the performance difference relative to the MPI implementation.

The rest of the paper is organized as follows. Section 2 describes hierarchical n-body methods to provide a context for understanding comparisons of HPF and MPI implementations of Anderson's method. Section 3 describes our MPI implementation in some detail and contrasts it with Hu and Johnsson's HPF implementation both qualitatively and quantitatively. Section 4 describes the key source of inefficiency in Hu and Johnsson's HPF implementation. Section 5 describes a modification to the HPF approach which dramatically improves performance. Section 6 summarizes our results and conclusions.

2 Hierarchical Methods

To compute far-field forces rapidly, hierarchical methods aggregate the effects from bodies a sufficient distance away, computing their influence as part of a group, rather than individually. The principal data structure used to construct groupings for these methods in 3D is an oct-tree. One constructs an oct-tree starting with a root box that contains all of the bodies and then recursively subdividing boxes into 8 boxes of equal size until a stopping condition is met.

Once the tree is formed, an upward pass over the boxes in the tree establishes the far-field approximations for each box. At the leaves, the approximation for a box is computed from the bodies within; at higher levels, the approximation for a box is computed from approximations for boxes it contains. The form of the approximation is application specific.

Here we describe a progression of three $O(n)$ hierarchical methods. First we introduce Greengard and Rokhlin's fast multipole method (FMM) [5]. Next, we describe the adaptive variant of this algorithm. Finally, we describe Anderson's method, which has the same algorithmic structure as the FMM methods, but but a different numerical technique for approximating far-field forces.

Fast Multipole Method. Rather than computing far-field for each body individually, as in the $O(n \log n)$ Barnes-Hut algorithm [3], Greengard and Rokhin's Fast Multipole Method [5] makes use of the observation that when a box A and a box B are "well-separated", the far-field effect of the bodies in box B on those in box A, and vice-versa, can be approximated as a single interaction between the *boxes*. Such interactions between well-separated boxes occur at all levels of the tree, and the savings in computation enable FMM to compute far-field forces in $O(n)$ time. Interactions are computed in a downward pass over the tree. At each level, interactions are computed between boxes at that level that are well-separated, and the results, collected in the form of a "local-field potential," are passed down to the next level. At the lowest level, the local-field potential for a box is passed down to each body inside and interactions between bodies not sufficiently separated are computed.

Adaptive FMM. The FMM algorithm just described assumes a tree of uniform depth. An adaptive variant avoids unnecessary refinement by not subdividing any box that contains fewer bodies than a specified threshhold. The key difference with respect to the non-adaptive algorithm is that the set of boxes with which a given box will interact is not statically known and must be computed from the shape of the adaptive tree by a somewhat complicated algorithm. To simplify implementation and maximize cache locality, several types of interaction lists for each box are computed before the upward pass and a separate computation phase is added between the upward and downward passes. There are three types of interaction computations and therefore three lists: boxes in *list1* are adjacent leaf boxes (and therefore are not sufficiently distant from each other to allow approximation); boxes in *list2* are the same size and well-separated (that is, sufficiently distant from each other to allow approximation); finally, boxes in *list34* are different sized and well-separated from the perspective of one of the boxes but not the other.

Anderson's Method. The algorithmic structure of Anderson's method [2] is the same as that of FMM. Its key difference from FMM is in the way it propagates potentials. For three-dimensional problems, the computational element of FMM is a multipole expansion located at the center of an abstract sphere containing the cluster of bodies; in contrast, Anderson's approximation computes potentials at locations on the circumference of a sphere. Compared to multipole methods, Anderson's method achieves the same level of accuracy with fewer levels in the tree.

3 Implementation Comparison

In this section, we first describe our hand-coded implementation of Anderson's method in some detail. Next, we describe highlights of Hu and Johnsson's HPF implementation. Finally, we compare the performance of these two implementations for several problem sizes and processor counts on a Cray T3E. This comparison shows that the HPF implementation has some significant inefficiencies relative to the hand-coded one.

3.1 The Hand-coded Implementation

The FMM program in the SPLASH-2 suite from Stanford [16] was the starting point for development of our hand-coded MPI implementation of Anderson's method, though our implementation now bears little resemblance to the original. Among the structural changes we have made to the code:

- We use MPI-based explicit communication rather than shared memory.
- We replaced the multipole expansions with Anderson's method for computing potentials.
- We use a 3D octree as the basis for the hierarchical solver rather than a quadtree.

The principal remaining similarity between the implementations is in the record structures used by the hierarchical solver. Below we describe key features of our MPI implementation.

Body Distribution. To distribute bodies among processors, we first compute the position of each body along a Hilbert curve[2], and then sort the bodies according to their position along the curve. Since both the Hilbert curve and the octtree recursively divide space in half along each dimension, all bodies in the same leaf of the octtree are contiguous after the sort. Next, we partition the sorted sequence of bodies among the processors by assigning each processor a contiguous range. We select the partition points to ensure that each processor is assigned all bodies in a subtree of the octtree. With this partitioning, we are able to construct octtrees locally, except for a brief communication phase in which processors exchange information about *shared boxes* (boxes at upper levels of the tree whose subboxes lie on more than one processor) to ensure that the representation of these boxes is globally consistent.

Construction of Neighbor Lists. A key step in adaptive hierarchical methods is building the interaction lists for each box, as described in Section 2. The fine-grained nature of the computation in this phase, combined with its large communication requirements, causes it to be a major bottleneck in the parallelized application if special care is not taken. By transforming the uniprocessor list construction algorithm into a form that enables us to gather non-local data using an efficient inspector-executor strategy, we are able to dramatically reduce the impact of list construction on the parallel runtime. (Details of our list construction algorithm are described in [11].)

Propagation of Potential Information. As noted before, we replicate information about shared nodes at the uppermost levels of the tree to all processors and ensure that all nodes in a subtree below any non-shared node are located on the same processor. This partitioning strategy avoids communication in the downward pass and requires only a single communication step in the upward pass when moving from private nodes to the shared parents.

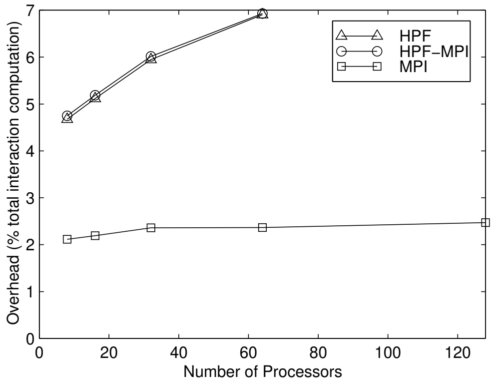
Interaction Computation. As in list construction, we communicate non-local data required in the interaction computation using a variation on the inspector-executor technique. Computation is divided into 3 parts: list1 interactions, list2 interactions and list34 interactions. To ensure load-balance, we move data for boxes involved in each of the three computation phases into a "weighted-block" distribution immediately prior to that phase. This involves looking in the work-list for each box involved in the computation to determine the amount of work it will do, and then minimally redistributing the boxes such that each processor will have approximately the same total amount of work.

3.2 The HPF Implementation

Details of Hu and Johnsson's implementation of Anderson's method can be found in [9]. Here we provide only a brief overview of some similarities and differences between their implementation and our hand-coded MPI implementation.

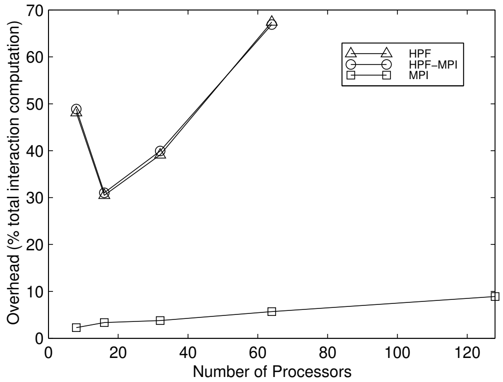
- They represent objects using multiple attribute arrays rather than a single record structure.
- They express communication of non-local data for irregular references using generalized array assignments to "gather" the data before computation. We discuss this in detail in section 4.
- They distribute bodies using a space-filling curve, though they do not exploit the relationship with octtrees to minimize communication during the tree-building phase. Instead they construct the tree level-by-level, block distributing the data for each level. As a result, parent data is *not* necessarily on the same processor as child data.
- They have parallelized the uniprocessor list creation algorithm in a fashion that requires more communication rounds than our approach.
- As a result of their level-by-level block distribution of boxes, they must communicate between each level during the upward and downward passes for propagating potentials.
- They use a clever scheme that allows enables them to approximate a weighted block distribution to load balance the interaction computations.
- They have an extra repartitioning phase after list construction. This step moves box data into the weighted block distribution described above.

[2] Hilbert curves [13] are one of a class of continuous, non-smooth, "space-filling curves" that map a 1-dimensional interval to an N-dimensional volume. Such curves can be constructed to pass arbitrarily close to every point in the volume.

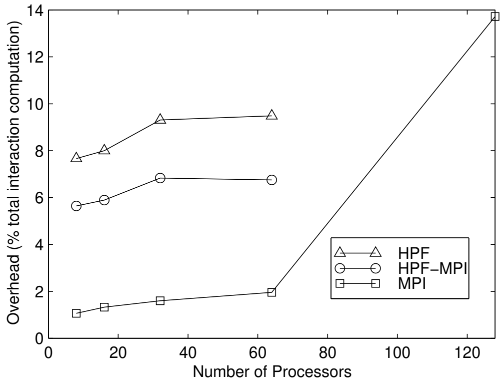


(b) Downwardpass

Figure 1: Upward and downward pass overhead with respect to total interaction computation time.



(a) Make_lists



(b) Tree build

Figure 2: List and tree construction overhead with respect to total interaction computation time.

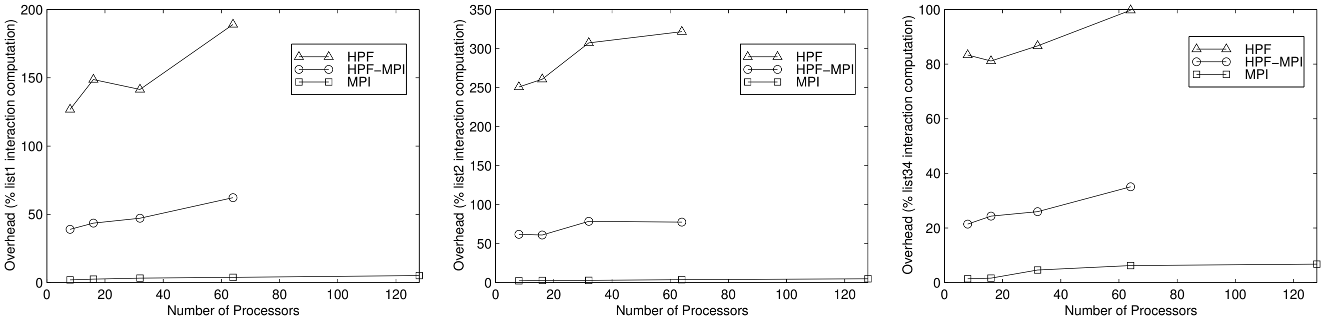
3.3 Performance Comparison

Here we compare the performance of three application variants: Hu and Johnsson's HPF implementation (labeled "HPF" in plots); our hand-coded MPI implementation (labeled "MPI"); and a variation of the HPF implementation in which, for some phases of the algorithm, Hu and Johnsson have replaced HPF constructs which cause communication (*i.e.*, foralls, copy_scatters, etc.) with calls to specialized MPI routines that they developed (labeled "HPF-MPI"). Not all phases are replaced; in some cases (such as list construction) use of the MPI routines would *increase* running time, perhaps because the routines were optimized for course-grained communication. We consider the "HPF-MPI" numbers, where used, to represent lower bounds on the communication time for the HPF implementation.

Experimental setup. All experiments were performed on the Cray T3E-600 at the San Diego Supercomputer Center, which consists of 272 300MHz DEC Alpha 21164 processors (128 of which are available at a time to user jobs). Processors each have 128 megabytes of physical memory (no virtual memory), and are connected by a 3D bi-directional torus interconnect. HPF programs were compiled using PGI's PGHPF compiler, release 2.3-1 for the T3E; all codes were compiled at the -O2 optimization level.

Methodology. In our experiments, we scaled problem size with the number of processors, so a doubling of processors implies a doubling of bodies simulated. Bodies were initially distributed according to a Plummer distribution [1]. Each application variant was run for a single timestep on 8, 16, 32, and 64 processors. Those that didn't run out of memory were run on 128 processors. Table 1 presents the absolute time for each application phase for the 64 processor configuration of the problem. "other" in the table refers to phases in the HPF application that don't exist in the MPI version.

Figures 1–2 show the overhead of the upward pass, downward pass, and list creation, and tree construction. In these figures, overhead is computed as the time spent in these phases as a percentage of the time spent computing interactions within the whole program execution. Figure 3 shows the list construction overhead for list1, list2, and list34 interactions. In this figure, overhead is computed as a percentage of the time to compute interactions associated with that list. All plots



(a) List1 (b) List2 (c) List34

Figure 3: List processing overhead as percentage of list interaction computation time.

phase	*MPI*	*HPF*
build tree	1.19	5.75
make lists	3.45	40.9
upward pass	1.39	3.99
list1 interactions	19.8	59.8
list2 interactions	24.5	92.7
list34 interactions	23.0	39.3
downward pass	1.43	4.18
other	0	5.36
TOTAL_TIME	77.5	255.

Table 1: Absolute times (in secs.) for 64 procs.

are relative to the fastest interaction computation time of the three implementation variants. In all of these figures, a horizontal line across the plot represents perfect scalability since we have scaled problem size with processor count.

Discussion. The most significant differences in partitioning strategy between the HPF and MPI implementations are for the tree construction, upward pass, and downward pass. In these phases, the hand-coded implementation uses a more efficient strategy based on replication of shared nodes in the upper levels of the tree. However, despite these differences, these phases show the smallest differences in performance when comparing the HPF and MPI implementations. The hand-coded implementation always has less overhead; however, the difference in overhead is fairly consistent as the problem size gets larger and the number of processors increases. For example, this overhead is 3-5% for the upward pass.

The remaining phases, which differ the least in terms of partitioning strategy, have the most significant differences in performance when comparing the HPF and MPI implementations. The largest differences are for the list2 interactions (between well-separated boxes). For 64 processors the overhead of the HPF implementation is a *factor of 84* greater than that of the hand-coded implementation. A glance at Table 1 confirms that this difference translates to a substantial difference in running time: the HPF implementation takes longer to compute list2 interactions than the hand-coded version takes to complete an entire timestep cycle.

```
      program tiny
      real*8 A(NUM_ELTS), B(NUM_ELTS), C(NUM_ELTS)
      integer INTER(NUM_INTER, NUM_ELTS)
CHPF$ distribute (block) :: A, B, C
CHPF$ distribute (*,block) :: INTER

CHPF$ ON HOME(A(i))
      do i = 1, NUM_ELTS
         do j = 1, NUM_INTER
            A(i) = B(i) * C(INTER(j,i))
         enddo
      enddo
      end
```

Figure 4: Irregular example program.

```
      program tiny
      real*8 A(NUM_ELTS), B(NUM_ELTS), C(NUM_ELTS)
      real*8, allocatable :: C_LOC(:)
      integer INTER(NUM_INTER, NUM_ELTS)
CHPF$ distribute (block) :: A, B, C, C_LOC
CHPF$ distribute (*,block) :: INTER

      <INSPECT INTER, DETERMINE NON-LOCAL C>
      <ALLOCATE C_LOC, COMMUNICATE C INTO C_LOC>
      <UPDATE INTER>

      do i = 1, NUM_ELTS
         do j = 1, NUM_INTER
            A(i) = B(i) * C_LOC(INTER(j,i))
         enddo
      enddo
      end
```

Figure 5: Inspector-executor version.

In the next section we investigate the source of these performance differences.

4 Performance Rationale

The measurements in Section 3.3 show significant differences in performance between the MPI and HPF implementations of Anderson's method. Here, we analyze the sources of these performance differences.

Two important points of comparison for implementations of irregular applications are 1) how they distribute data (and computation load) across processors, and 2) how they satisfy (recognize and communicate for) references to off-processor data. In the previous section we noted that performance differences in performance between the MPI and HPF implementations do not seem to correspond to differences in data distribution methods. Therefore, here we focus our attention on differences in the methods used to satisfy non-local irregular references.

Both implementations use variations on the *inspector executor* paradigm to acquire non-local data needed to satisfy references. We describe this paradigm to provide a context for understanding the variations used in the HPF and MPI implementations of Anderson's method. In the course of this section, it will become clear that differences in how inspector-executors are used are responsible for the principal differences in performance of the implementations.

4.1 Inspector-Executors

Since no closed form representation of the data accessed by an irregular reference can be computed statically, runtime processing is needed (a) to determine which (if any) accesses through an irregular reference will access off-processor data, and (b) to coordinate necessary data movement. The runtime processing for a reference can be costly, especially the communication. For this reason, it is advantageous to first determine the locality for all dynamic instances of an irregular reference in a loop and then communicate for all non-local values in a single step. This strategy is known as the "inspector-executor" paradigm [12, 14]. The *inspector* determines what non-local data will be accessed, and the *executor* performs the computation on local data and localized non-local data.

Figure 4 abstracts a typical situation in which irregular references arise. Arrays `A`, `B` and `C` are attributes of a single class of object, and array `INTER` describe for each object the other objects in its class that it will interact with. The code fragment corresponds to a phase in which each object interacts with all the other objects on its interaction list. The references to `C` are irregular since they use an indirection array as a subscript.

An inspector for a loop nest should ideally be placed at the outermost loop level allowed by data dependences. In Figure 4 the inspection phase could be placed before the outermost loop. The inspector would execute only once, examining the contents of `INTER(j,i)` and gathering *all* non-local elements of `C` needed by each processor. Resource constraints may make such an inspector placement impossible and require that communication be strip mined to avoid excessive space for storing off-processor values.

To ensure that non-local data accessed multiple times by a single processor in an inspected loop nest is communicated and stored *only once*, a hash table is used

```
foreach Node n accessed in Loop l.
  if n is NOT in localtree then
    nonlocal.INSERT(n)
othersNonlocalIds = ALLTOALL(nonlocal.ids)
found = localtree.SEARCH(othersNonlocalIds)
myNonlocals = ALLTOALL(found)
localtree.INSERT(myNonlocals)
PROCEED WITH LOOP NEST COMPUTATION...
```

Figure 6: Algorithm for inspecting octtrees.

to avoid duplicates. In Figure 4, duplicate values of `INTER(j,i)` for many subscript positions can cause repeated accesses to same non-local element of `C` by a processor. These non-local values need only be gathered and stored once on that processor. Figure 5 shows what the sample code might look like after transformation to inspector/executor style. The statements within "<>" would be translated to one or more calls to routines from a library such as *CHAOS* [4].

To use values collected by an inspector-based communication, the indirection array (`INTER` in Figure 4) must be updated to reflect the locations of the non-local elements of the indexed array (`C`) before use in the executor loop. In their standard usage, inspector-executors transform potentially non-local irregular references into definitely local, *indirect* references. We show that while use of the inspector-executor strategy by our hand-coded MPI implementation works this way, the strategy used in the HPF implementation does not.

4.2 Hand-coded Inspection

Our implementation uses a form of inspector-executors adapted to our choice of Warren and Salmon "hashed-octtrees" [15] for the octtree data structure of the application. Looking up a node in such trees uses a unique identifier (representing the nodes location in the tree) as a key for accessing nodes in a hashtable representing the tree. This approach simplifies management of distributed trees in two ways. First, the identifier for a node is the same on *all* processors. Second, integration of non-local nodes into a local tree is simple: data for non-local nodes is simply added to a processor's hashtable.

The basic algorithm we use to inspect loops over nodes in the octtree is shown in Figure 6. First, each processor inspects its local portion of the computation for accesses to non-local nodes and collects identifiers for these nodes into a hashtable. Next, all processors exchange the IDs in their off-processor hashtables. Third, each processor searches its tree for data requested and then replies with the necessary data if found. Finally, each processor inserts non-local data received into its tree hashtable and the loop computation continues with-

```
      program tiny
      real*8 A(NUM_ELTS), B(NUM_ELTS), C(NUM_ELTS)
      real*8 C_INTER(NUM_INTER, NUM_ELTS)
      integer INTER(NUM_INTER, NUM_ELTS)
CHPF$ distribute (block) :: A, B, C
CHPF$ distribute (*,block) :: INTER, C_INTER

      forall (i=1:NUM_INTER,j=1:NUM_ELTS)
         C_INTER(j,i) = C(INTER(j,i))
      end forall

      do i = 1, NUM_ELTS
         do j = 1, NUM_INTER
            A(i) = B(i) * C_INTER(j,i)
         enddo
      enddo
      end
```

Figure 7: Regularization example code.

phase	redund	nodes accessed
build tree	1	children
make lists	216	chldrn of collgs of prnt
up/downward pass	1	children
list1 interactions	26	colleagues
list2 interactions	189	well separated boxes
list34 interactions	?	(none in full tree)

Table 2: Redundancy factors; worst case, full tree.

out further interruption.

Note that, though we don't have to update any indirection array (since the array being indexed is actually a hash table), we still effectively transform potentially non-local irregular references into definitely local, *indirect*[3] references, as in our earlier description of the standard inspector-executor paradigm.

4.3 Inspection in HPF

As noted earlier, Hu and Johnsson use generalized array assignments to gather non-local data in what amounts to a variation of the inspector-executor technique. However there is an important difference between their technique and the standard inspector-executor strategy described above: their technique transforms potentially non-local irregular references into definitely local, *regular* references.

Figure 7 demonstrates the "regularization" of the irregular reference in our example program. A new array `C_INTER` of the same size and dimension of `INTER` is introduced to represent localized values of `C`. Next, `C_INTER` is assigned the *values* of `C` that are referenced through `INTER`. Because of the way the arrays have been distributed, any non-local elements of `C` referenced by `INTER` are implicitly communicated in this step and will subsequently be local. Finally, the indexed reference in the computation loop is replaced with a reference to the new array.

On the surface, regularization appears to be a simple and elegant solution for handling irregular references in HPF. However, if the index array is much larger than the array(s) it is indexing, this approach can severely hurt performance.

For an illustration of the potential impact, consider the list2 interactions of Anderson's method (between well-separated boxes). Suppose box `B` is an interior node in a full tree; it is then well-separated from 189 other boxes in a 3D oct-tree. Each of those boxes is also well-separated from `B`, so `B` appears on 189 interaction lists. Using the regularization strategy to localize references to `B` in those lists thus causes `B`'s data to be copied 189 times, potentially across processor boundaries.

There are two ways in which such copying can hurt performance. First, multiple copies of `B` are potentially communicated to a single processor (if `B` appears on multiple lists on that processor), thereby increasing communication volume and latency. Second, whether `B`'s data is local or nonlocal, new space must be allocated to store it, resulting in as much as a factor of 189 difference in `B`'s storage requirements (if `B` is only on local lists).

We claim that the additional storage required for regularization is ultimately responsible for the performance difference between the HPF implementation and its hand-coded counterpart. Although the runtime processing required to implement inspection should ideally be placed outside of as many loops as dependences will allow, resource constraints might force their placement within loops. In several phases of the Hu and Johnsson's HPF implementation, their regularization strategy increases storage requirements to such an extent that they needed to place their gather/inspection code within some loops with many iterations. In contrast, we are able to place these inspectors at the outermost level in our hand-coded MPI implementation because we use a hashtable to eliminate storage redundancy.

As supporting evidence for our claim that this is the main source of overhead in the HPF implementation, we point out the correlation between data in Table 2, which shows redundancy factors for various phases of the algorithm, and the results in Figures 1 and 2. The largest difference in overhead between the implementations is in the make-list phase which has the largest redundancy factor. The build-tree, upward and downward passes have little redundancy and have small differences in overhead, despite our hand-coded strategy for reducing communication in these phases described in section 3.

To prove our claim, in the next section we show that removing the redundancy in the HPF implementation eliminates much of difference in performance between the implementations.

[3]The indirection here is through the hashtable.

```
      program tiny
      real*8 A(NUM_ELTS), B(NUM_ELTS), C(NUM_ELTS)
      integer INTER(NUM_INTER, NUM_ELTS)
CHPF$ distribute (block) :: A, B, C
CHPF$ distribute (*,block) :: INTER
      real*8 C_LOC(MAX_LOC*number_of_processors())
      integer map(MAX_LOC*number_of_processors())
CHPF$ distribute (block) :: C_LOC, map

      map = 0
      call hash(map, INTER)

      forall (i=1:MAX_LOC*number_of_processors(), map(i).ne.0)
         C_LOC(i) = C(map(i))
      end forall

      call compute(A, B, C_LOC, INTER)
      end
```

Figure 8: Transformed main program.

5 Synthesis Approach

To prove that communication and storage redundancy due to regularization is largely responsible for the performance gap between Hu and Johnsson's HPF code and our hand-coded MPI implementation, we must show that avoiding this redundancy would improve performance. Unfortunately, regularization is the standard idiom for handling irregular references within the HPF language and there are few alternatives. In the HPF language proper, we have found that *programmer*-driven inspection of data for locality and redundancy is difficult or impossible because:

1. there is no straightforward means of accessing data owned by a particular processor, and
2. there is no notion of arrays local to a processor.

Going back to our example from Section 3, we are unable to efficiently express the hashing of the indirection array `INTER` in HPF global code.

So, barring sophisticated compiler support for creating efficient inspectors automatically such as that described by von Hanxleden [6], we know of no method for a programmer to circumvent the redundancy inherent in the regularization strategy entirely within the HPF language. We *have* found, however, that a modest use of HPF "extrinsic procedures" [7, 10] enables us to avoid the redundancy while retaining the benefits of the high-level HPF model for the rest of the application. Here, we introduce our strategy by demonstrating its use on the simple program from Section 4, show how we have applied it to the Hu and Johnsson's HPF implementation, and present measurements of the revised application.

5.1 General Methodology

HPF extrinsic procedures enable HPF programs to call code not written in strict data-parallel style and allow

```
extrinsic (hpf_local)
subroutine hash(map, INTER)
integer map(:), INTER(:,:)
integer table(NUM_ELTS), sum, i

table = 0
do i = 1,size(INTER,2)
   do j = 1, NUM_INTER
      table(INTER(j,i)) = 1
   enddo
enddo
sum = 0
do i = 1, NUM_ELTS
   if (table(i) .eq. 1) then
      sum = sum + 1
      table(i) = sum
      map(sum) = i
   endif
enddo
do i = 1,size(INTER,2)
   do j = 1, NUM_INTER
      INTER(j,i) = table(INTER(j,i))
   enddo
enddo
end subroutine
```

Figure 9: Simple hashing procedure.

the called code to operate on distributed arrays defined in the HPF program. HPF_LOCAL extrinsic procedures, in particular, have at their disposal several intrinsic functions that are unavailable at the global level. For example, the intrinsic function `size` can be used within an HPF_LOCAL extrinsic to determine the *local* extent of a dimension of a distributed array passed as an argument. Because extrinsic procedures enable processors to perform different operations in parallel (as opposed to identical operations on different parts of the same array) their use is frowned upon by data-parallel purists.

We are interested solely in the ability extrinsic procedures give us to define local arrays and manipulate local sections of distributed arrays. We use this capability first to locally collect a vector of unique indices that will be used to indirectly access a distributed array and then later to perform a communication-free computation with localized data. Figures 8–9 demonstrate how we use these procedures to efficiently localize the irregularly indexed array in the example program shown in Figure 4.

Figure 9 implements `hash`, an HPF_LOCAL procedure to perform the hashing that we are unable to express in the the HPF language proper. We declare an array `table` to be the *global* size of the indexed array `C` and then loop over the local portion of the index array `INTER` to determine the elements of `C` that will be needed locally. The second loop collects the index positions of `C` that will be needed locally into local slots in the array `map`. (Figure 8 shows how the array `map` is later used to gather values of the indirectly indexed array `C`.) Finally, the third loop nest in Figure 9 updates `INTER` with the indices that localized values will have *after* they are gathered.

The transformed main program in Figure 8 declares two new arrays: the mapping array `map` described above and the array `C_LOC` into which elements of `C` will be gathered.[4] After calling the `hash` procedure to compute `map`, a forall loop actually gathers the non-local values needed by each processor. Finally, the computation is performed using the localized data.

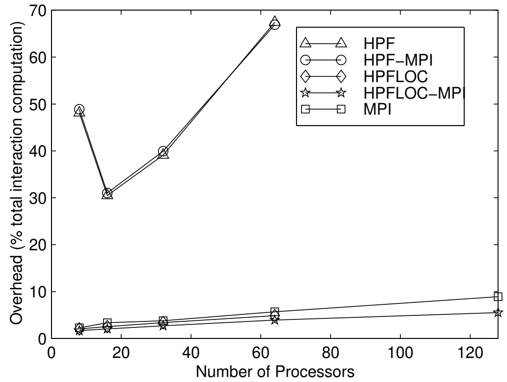
We are forced to package the computation loop into another extrinsic procedure due to what we perceive to be a limitation of HPF: although we have ensured that the values of `C` that will be indirectly referenced by `INTER` will be local in `C_LOC`, there is no way for us to indicate this to the compiler. (The HPF "independent" directive asserts only that there are no dependences in a loop, *not* that no communication is necessary.) The `compute` procedure performs the computation loop using the localized data now available in `C_LOC`. In `compute`, `C_LOC` is indirectly indexed with the `INTER` array, which was rewritten by the `hash` procedure. We do not show the code for the `compute` extrinsic procedure as it is nearly identical to the original program.

5.2 Application

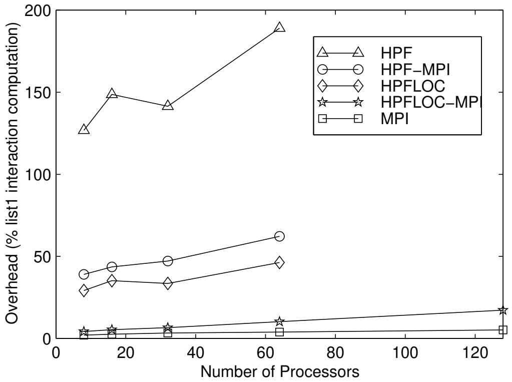
We applied our inspection strategy using HPF_LOCAL to the phases of Hu and Johnsson's HPF implementation where the performance measurements of Section 3.3 indicated that they were needed most: list construction, and interaction computation. Here we provide results from experiments on the transformed code. The overhead measurements in the plots follow the presentation strategy outlined in section 3.3. We show results for HPF, HPF-MPI, and MPI as before. In addition, we introduce HPFLOC, and HPFLOC-MPI variants based on our use of HPF_LOCAL to eliminate communication and storage redundancy.

Figures 10–11 demonstrate the success of our technique applied to the HPF implementation, and substantiate our claim that the performance gap between the HPF and MPI implementations was due to the storage and communication redundancy arising through regularization. When comparing the original HPF version versus HPFLOC on each of the graphs, the reductions are dramatic. For example, Figure 11(a) shows that overhead in list creation is reduced by roughly a factor of 10 by using an HPF_LOCAL inspector. When using this strategy with Hu and Johnsson's MPI-based collective communication rather than the PGHPF default communication, Figures 10–11 show that overhead drops by an additional factor of two or more.

[4]For brevity, in the main program we have omitted "interface" blocks for the `hash` and `compute` extrinsic procedures. These statements are necessary so the compiler can correctly pass global data to local procedures.



(a) List creation



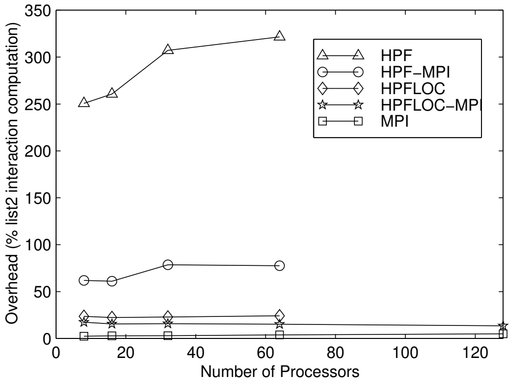
(b) List1

Figure 10: List creation and list1 interaction times as percentage of interactions.

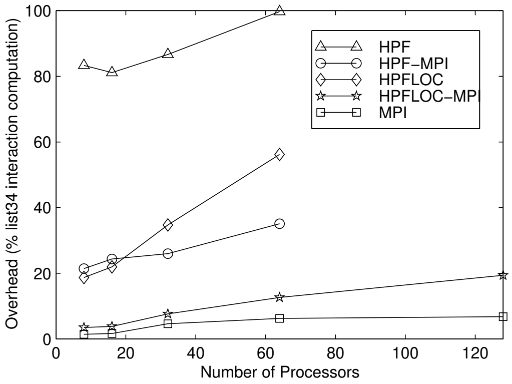
Only the list34 interactions seem to give the HPFLOC version some trouble. We attribute this to the fact that we are forced to use a rather inefficient communication method to move the boxes into the weighted-block distribution. Specifically, because of the way data is laid out and the formulation of the function which determines the new distribution, we are forced to use a `scatter_copy` rather than a `forall` to move the data; unfortunately `scatter_copy` does not provide a good way to move multi-dimensional data. One is forced to use a very general technique by which it is possible to describe the new position of every single element in the array; in this case, we simply want all elements of a row to follow the first element.

6 Summary and Conclusions

We have described and evaluated two sophisticated parallel implementations of an adaptive, hierarchical solver which uses Anderson's method for calculating interactions in n-body systems: our explicitly-parallel MPI implementation and Hu and Johnsson's data-parallel HPF implementation. Our measurements of these implemen-



(a) List2



(b) List34

Figure 11: List2 and list34 times as percentage of interactions.

tations demonstrated that there were significant performance differences between the hand-coded MPI and the HPF implementations. After considerable analysis of the performance and the implementation strategies, we hypothesized that the primary source of inefficiency in the HPF implementation was redundant communication that was necessary to initialize redundant storage that is used for regularizing indirect references. We demonstrated that when this redundancy is eliminated in just two phases of the HPF implementation, with the aid of HPF_LOCAL semantics, the performance of the otherwise unchanged HPF implementation closely approaches that of the hand-coded version.

Figure 12 demonstrates just how close: from the plot we can see that on 64 processors, while the original HPF implementation incurs nearly 300% more overhead than the hand-coded version, our implementation reduces the gap to just 50%. If we accept the assertion that collective communication support in the PGI HPF compiler could be improved to approach the efficiency of the specialized communication routines written by Hu and Johnsson (used by the HPF-MPI and HPFLOC-MPI versions) then there is a fairly consistent gap of about

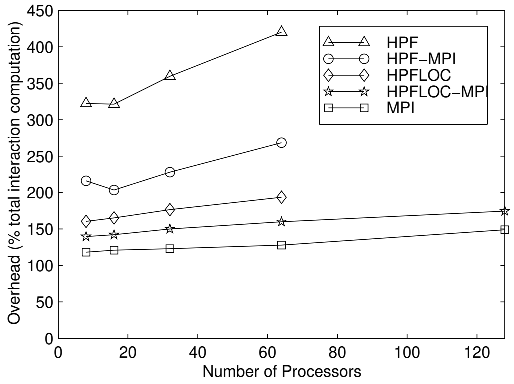


Figure 12: Total time as % of interaction computation.

phase	*MPI*	*HPFLOC-MPI*
BUILD_TREE	1.19	4.08
MAKE_LISTS	3.45	2.37
UPWRD_PASS	1.39	3.33
LIST1_INTR	19.8	25.2
LIST2_INTR	24.5	26.0
LST34_INTR	23.0	23.6
DNWRD_PASS	1.43	4.22
OTHER	0	5.48
TOTAL_TIME	77.5	96.9

Table 3: Absolute times (in secs.) for 64 procs.

25% left between the performance of the hand-coded MPI implementation and the revised HPF implementation.

Two questions remain. First, what accounts for the remaining performance gap and can it be bridged? Second, what are the implications of this work for irregular computation and HPF?

Remaining Gap. From Table 3 it's clear that almost all of the remaining difference in running times between the two implementations can be attributed to the phases of the algorithm that we did not change. The one exception would appear to be the List1 interactions, for which there is still be a substantial gap of five seconds. However, four seconds of this gap are due to a small difference in the *computational* algorithm: all the HPF versions have a test in an innermost loop which we avoided in the MPI implementation. (We accounted for such differences in our definition of overhead, so they don't affect the overhead measurements shown in our plots.) If we adjust for the List1 algorithmic inefficiency, we find that the phases of the HPF implementation that we didn't change account for 86% of the remaining performance gap while the phases we changed account for only 14% of the remaining gap.

There are several factors which contribute to the remaining 14% performance gap in the application phases we modified to use the HPF_LOCAL-based inspection strategy. First, we have taken great pains in the MPI implementation to ensure that no communication of data "holes" takes place, whereas the HPF implementation cannot. For example, when communicating the particles associated with a leaf box, we communicate and store exactly the number of particles associated with that box, while the HPF implementation communicates and stores the maximum number of particles per box for every box. Second, our use of structures to group data in the MPI implementation results in a single communication per object whereas the HPF implementation's use of attribute arrays results in multiple communications per object, which increases communication overhead.

These same factors, of course, also contribute to the performance gap for the phases that we did not modify. There are two components to the remaining difference in the unmodified phases: the extra partitioning phase required in the HPF application, and the algorithmic phases (upward and downward passes, tree build). We first note that the extra partitioning phase was not improved by the use of Hu and Johnsson's MPI routines, primarily because it uses more complicated communication constructs such as scan reductions that they chose not to implement in MPI. The time for this phase could potentially be lower if the PGI runtime library communication routines can be improved.

After accounting for the factors discussed above, we attribute the rest of the performance differences to differences in partitioning strategies used by the HPF and MPI implementations. As described in Section 3 the main feature that distinguishes the partitioning strategy we used in MPI from that used in HPF is the notion of processors sharing boxes in the upper levels of the tree. Given the restrictions on user knowledge of processor/data relationships imposed by the HPF language, it is not clear how one could implement this strategy in HPF. Perhaps instead of a single array representing boxes, one could split them into two sets: those owned by a single processor in a distributed array and those owned by all processors in a separate replicated array. Whether a compiler could efficiently coordinate data motion between the two sets as we have in our implementation, is not obvious.

Implications. We have shown that if arbitrary irregular applications are to be implemented with high efficiency in data-parallel languages such as HPF, then special care must be taken to avoid redundancy in the communication and storage of non-local data. We believe that this work therefore has implications for two groups:

1. *Vendors* of HPF compilers need to provide better support for automatically generating appropriate inspector-executor code for irregular references. Techniques described by von Hanxleden [6] in his dissertation would suffice for the cases we encountered, though extensions to these techniques may be needed for more complex cases, such as those involving multiple levels of indirection. Without compiler-synthesized inspectors and executors, we know of no way to eliminate the communication redundancy that results from regularization without dropping into HPF_LOCAL extrinsics. Attempts to achieve the desired effect by not using a forall loop for regularization of irregular reference patterns caused the PGI HPF compiler to employ run-time resolution with which no speedup is possible.
2. Until better inspector-executor support becomes widely available in HPF compilers, HPF *application developers* would do well to follow our example and use HPF_LOCAL extrinsic procedures to implement inspector-executor style handling for irregular references in cases where storage and communication redundancy prove significant.

7 Acknowledgements

We thank Yu "Charlie" Hu and Lennart Johnsson for giving us unrestricted access to their HPF n-body application. We thank the National Partnership for Advanced Computational Infrastructure at the University of California, San Diego for access to their Cray T3E, and TCCIS at the University of Houston for access to their IBM SP2.

References

[1] S. Aarseth, M. Henon, and R. Wielen. *Astronomy and Astrophysics*, 37, 1994. Reference for Plummer distributions for N-body problems.

[2] C. R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comput*, 13(4):923–947, July 1992.

[3] J. Barnes and P. Hut. A hierarchical $o(n \log n)$ force calculation algorithm. *Nature*, 324:446–449, 1986.

[4] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.

[5] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Physics*, 73:325–348, 1987.

[6] R. v. Hanxleden. *Compiler Support for Machine-Independent Parallelization of Irregular Problems.* PhD thesis, Dept. of Computer Science, Rice University, Dec. 1994. Available as CRPC-TR94494-S

from the Center for Research on Parallel Computation, Rice University.

[7] High Performance Fortran Forum. High Performance Fortran language specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[8] Y. C. Hu and S. L. Johnsson. Implementing $o(n)$ n-body algorithms efficiently in data-parallel languages. *Scientific Programming*, 5(4):337–364, 1996.

[9] Y. C. Hu, S. L. Johnsson, and S.-H. Teng. High Performance Fortran for highly irregular problems. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 13–24, Las Vegas, NV, June 1997.

[10] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[11] C. McCurdy. Efficient techniques for n-body simulation on distributed memory architectures. Master's thesis, Dept. of Computer Science, Rice University, 1999. Forthcoming.

[12] P. Mehrotra and J. Van Rosendale. Compiling high level constructs to distributed memory architectures. In *Proceedings of the 4th Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, Mar. 1989.

[13] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, NY, 1994.

[14] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, Apr. 1990.

[15] M. Warren and J. Salmon. A parallel hashed-octtree n-body algorithm. In *Proceedings of Supercomputing '93*, Portland, OR, Nov. 1993.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.