High Performance Fortran for Highly Irregular Problems*

Y. Charlie Hu
Aiken Computation Lab
Harvard University
Cambridge, MA 02138

S. Lennart Johnsson
Dept. of Computer Science
University of Houston
Houston, TX 77204

Shang–Hua Teng
Dept. of Computer Science
University of Minnesota
Minneapolis, MN 55455

Abstract

We present a general data parallel formulation for highly irregular problems in High Performance Fortran (HPF). Our formulation consists of (1) a method for linearizing irregular data structures (2) a data parallel implementation (in HPF) of graph partitioning algorithms applied to the linearized data structure, (3) techniques for expressing irregular communication and nonuniform computations associated with the elements of linearized data structures.

We demonstrate and evaluate our formulation on a parallel, hierarchical N–body method for the evaluation of potentials and forces of nonuniform particle distributions. Our experimental results demonstrate that efficient data parallel (HPF) implementations of highly nonuniform problems are feasible with the proper language/compiler/runtime support. Our data parallel N–body code provides a much needed "benchmark" code for evaluating and improving HPF compilers.

1 Introduction

Data parallel programming provides an effective way to write maintainable, portable, and scalable parallel codes. It has enjoyed great success in solving many structured problems, such as dense matrix computations, finite difference calculations, nonadaptive multi–grid [8] and nonadaptive hierarchical methods for the potential and force field evaluation of particle interactions [13, 14]. The data parallel programming paradigm has also been used successfully for unstructured finite–element problems [20, 16, 17]. All the data parallel unstructured mesh finite–element problems used elements that uniformly were of the same order and carried out the computations using unassembled stiffness matrices. Thus, though the communication was unstructured and nonuniform, the element computations were uniform. In the case of adaptive hierarchical particle methods, the computations associated with particles and elements of the hierarchy are nonuniform in addition to the unstructured and nonuniform communication requirements. Hence, the adaptive hierarchical particle methods have the characteristics seen in many real–world science and engineering problems, where for instance unstructured grids and elements of varying orders are being used. In Fortran 90, irregular problems typically are described using index arrays to introduce a level of indirection for "gather" and "scatter" array accesses.

Most irregular problems possess as much potential parallelism as their regular counterpart [21, 28], but their nonuniformity is often perceived as making them unfit for the data parallel programming paradigm. Moreover, their nonuniformity poses a serious challenge for load–balancing the computations while communication is kept at as low a level as possible. Furthermore, predicting performance is difficult, though regularization and minimizing communication usually helps in making performance predictions more reliable.

In this paper we present and evaluate a solution to the disparity between the simple array data structures in data parallel languages and the complicated data structures often found in irregular problems, such as trees and other hierarchical structures. Our formulation consists of a set of techniques for (1) linearizing irregular data structures into arrays, (2) partitioning of arrays for load–balance and data–locality, and (3) expressing nonuniform computations and irregular communications associated with the array elements.

We apply our formulation to an adaptive, hierarchical N–body method using nonuniform particle distributions, where the data structure and communication pattern is highly irregular. Our experimental results show that efficient data parallel (HPF) implementations of highly irregular problems are feasible if

- the nodal compiler generate efficient code,

*The first two authors were supported by the U.S. Air Force under grants AFOSR F49620-93-1-0480 and F49620-96-1-0289. The third author was supported by an NSF CAREER award (CCR-9502540), an Alfred P. Sloan Research Fellowship, and an Intel research grant.

To appear in the *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, Nevada, June 1997.

- the runtime system provides efficient gather/scatter operations,
- the language, compiler, and runtime system in some combination support uneven distribution of arrays for load–balancing, and
- ragged arrays are supported (for efficient memory usage).

The third feature is included in the most recent definition of HPF [11], while the forth is not. The nodal code generated by the current generation commercial HPF compilers is not well optimized for many expressions, and the supporting runtime systems are often quite inefficient. The third feature is not supported in current commercial HPF compilers. With respect to ragged arrays, it is interesting to note that the nested data parallel language NESL [3] supports ragged arrays. The technique used to flatten ragged arrays in the compilation of nested data parallel programs relies on highly optimized segmented prefix runtime support to achieve good performance.

Our data parallel N–body code provides a much needed "benchmark" code for evaluating and improving data parallel languages and HPF compilers for highly irregular and nonuniform problems.

2 A Data Parallel Formulation and Its Language/Compiler Requirements

To solve a problem Q on a parallel machine with p processors, explicitly or implicitly, we decompose Q into p subproblems $Q_1, \ldots, Q_p$ with processor j responsible for subproblem Q_j. The process of dividing Q into subproblems is called *problem partitioning*.

For all but so called "embarrassingly parallel" problems, the subproblems $Q_1, \ldots, Q_p$ are not independent; in order to solve subproblem Q_j, processor j needs to gather some data from other processors. Conversely, the solution of other subproblems require that processor j sends data to other processors. The (data, source, destination) tuples form the *communication set*. To achieve good speedup in solving a problem Q, we need to, efficiently, (1) find a problem partitioning that balances the amount of computations and communication among processors; (2) distribute the initial data set among the processors; (3) generate the communication set; (4) gather/scatter data as defined by the communication set. Retaining *data locality* is crucial in reducing communication overhead and in achieving good speedup.

In a data parallel formulation arrays are used to represent the data. In our formulation, the hierarchical data structures are mostly mapped into one–dimensional arrays, with attributes possibly mapped to a second local axis making the arrays two–dimensional. We *linearize* the nonuniform hierarchical structures through the use of space–filling curves, a technique that has been used by others with good results, see for instance [22, 25]. We express our program/algorithm in terms of primitives such as reduction, prefix sum, segmented prefix sum, gather/scatter, and broadcast. These operations can be efficiently supported by data parallel compilers and runtime systems, as demonstrated by the Connection Machine Fortran compiler and its supporting Connection Machine Run–Time System.

A Data-Parallel Formulation Scheme

1. **Linearization**: Space–filling curves has been used for the linearization of both hierarchical N–body problems and unstructured mesh problems with good results. Ideally, the space–filling curve should have the property that for a hierarchical method, locality of data references are preserved as required by the algorithm in traversing the hierarchy. To be treated properly, locality shall be measured in terms of the distributed memory architecture and must take cache effects into account locally as well as between processors, and, unless the network bandwidth is high in comparison with the processing capacity, also the nonuniform access times and bandwidths for interprocessor accesses. In addition, it is desirable that the space–filling curve is easy to generate and results in simple indexing.
2. **Problem Partitioning**: Parallel partitioning is required for dynamic large–scale problems and often desirable for large–scale nonuniform static problems for which partitioning potentially could be performed sequentially off–line. The partition information is used to rearrange the linear data structure of the problem so that, implicitly, the computational load is balanced among processors while the communication among processors is close to minimum, or in fact optimal.
3. **Data Distribution**: With the partitioning information, the initial data is redistributed among the processors such that the computational load is balanced and interprocessor communication in accessing the data structure is close to minimal for the algorithm operating on the data structure.
4. **Communication Set Generation**: This is one of the key steps in a parallel implementation. The partitioner generates the necessary information for the runtime system to block communication between processors, as well as provide caching of data to avoid multiple fetches of the same data between the same pair of processors.
5. **Computation/Communication**: Based on the information generated in the linearization and partitioning, we express the main body of computations in terms of data parallel primitives such as reduction, broadcast, (segmented) prefix sum, gather/scatter, sorting,

and median–selection that have efficient data parallel implementations.

6. **Re–partition and Re–Distribution**: For dynamic problems, we need to repartition and redistribute the data so that the next step of computation can be supported efficiently. A data parallel partitioner is particularly useful in such applications.

The performance of an HPF code for our formulation of nonuniform and unstructured problems depends heavily on the quality of the nodal compiler, runtime gather/scatter operations, and prefix sums.

In the remainder of this paper, we demonstrate how to apply our data parallel formulation for nonuniform problems by applying it to an adaptive hierarchical N–body algorithm for nonuniform particle distributions.

3 Hierarchical N–body Methods

Hierarchical methods for N–body simulations have enabled the simulation of particle systems with up to 100 million particles on Massively Parallel Processors (MPP) installed a few years ago [13], and should allow for the simulation of 1 – 10 billion particle systems in main memory on the MPP systems currently under installation. Large–scale N–body simulations have applications in areas such as celestial mechanics, plasma physics, materials science and molecular design. Hierarchical N–body methods include the nonadaptive $O(N)$ methods by Greengard and Rokhlin [9], Zhao [30], Anderson [1], and the $O(N \log N)$ methods by Appel (proven to be of $O(N)$ in [7]), and by Barnes and Hut [2] and by Callahan and Kosaraju [5].

The methods of Appel, and Barnes and Hut, and Callahan and Kosaraju are readily extended to nonuniformly distributed particles. Carrier, Greengard, and Rokhlin [6] presented an adaptive version of the Greengard–Rokhlin method. Similar extensions can be made to Anderson's and Zhao's methods. [1]

Hierarchical N–body methods pose great challenges to parallel implementations. First, the complex computational structures and mathematics involved demand significant programming efforts; second, the highly irregular and extensive communication patterns make partitioning for locality and load–balancing on parallel platforms harder than partitioning unstructured meshes.

Previously (see Table 1), Salmon and Warren [29], Liu and Bhatt [19], and many others have shown that the Barnes–Hut method can be implemented efficiently on parallel scalable architectures, using the message–passing programming paradigm. Nonadaptive versions of the $O(N)$ methods have also been implemented on parallel machines in both the message–passing and the shared–memory programming paradigms [4]. In contrast, due to the more complicated computational structure, little progress has been made on parallel implementations of adaptive versions of the $O(N)$ methods. To our knowledge, the only parallel implementation of this type of method prior to ours is a shared–memory implementation of a 2–D version of the Greengard–Rokhlin method [25]. In previous work [13, 14], we have shown that efficient scalable code can be produced in data parallel languages for nonadaptive versions of Anderson's method.

Table 1: Summary of previous parallel implementations of hierarchical N–body methods. MP denotes message–passing, and SM shared–memory.

Author	Method	Prog. Model	System	Effi.
Barnes–Hut method				
Warren–Salmon [29]	BH	MP	Intel Delta	28%
Liu–Bhatt [19]	BH	MP	CM–5	30%
Singh et al. [26]	BH	SM	Stanford DASH	
Nonadaptive $O(N)$ methods				
Board et al. [4]	GR	SM	KSR-1	20%
Zhao–Johnsson [31]	Zhao	SIMD	CM–2	12%
Adaptive $O(N)$ methods				
Singh et al. [25]	GR	SM	Stanford DASH	

3.1 Adaptive $O(N)$ Hierarchical Methods

There are two key ideas in $O(N)$ hierarchical methods which together achieve the reduced arithmetic complexity:

1. Approximate the force or potential due to a cluster of particles with a single computational element;

2. Hierarchically form and use the computational elements.

All $O(N)$ methods [1, 9, 30] share the same computational structure and differ only in the approximate computational elements they use. There are two kinds of computational elements used in $O(N)$ methods: *far–field potential representation* and *local–field potential representation*. Both are used to represent the potential due to a cluster of charged particles at evaluation points far away from the cluster. The elements differ in whether the mathematical representation is with respect to the geometric center of the domain containing the cluster of particles whose action is approximated (for far–field potential) or whether it is with respect to the geometric center of the domain of the set of particles upon which the action shall be evaluated (for local–field potential).

A key concept in hierarchical methods is the notion of *well–separateness*. It formally specifies how far away two clusters of particles need to be in order to use the approximate

[1] The proof in [6] that the adaptive method retains $O(N)$ complexity uses the fact that the depth of the hierarchy in computer simulations is limited by the machine precision, i.e. there is a large constant $-log_2\epsilon$ in the big–O. Callahan and Kosaraju's formulation requires $O(N \log N)$ operations independent of the particle distribution.

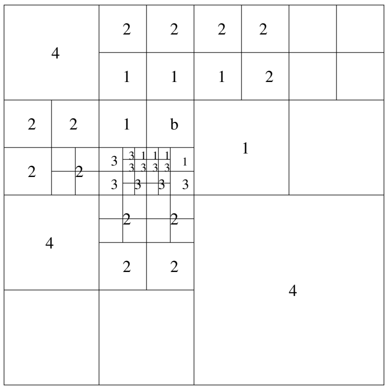


Figure 1: Adaptive domain decomposition, with one–separation near–field.

computational elements. Formally, box i and j are well–separated if box j is a distance $s \cdot sizeof(i)$ away from box i or box i is a distance $s \cdot sizeof(j)$ away from box j, where s is the degree of separation. For example, with *one–separation*, all boxes beyond adjacent boxes are well–separated. The *near–field* of a box is defined to contain all non–well–separated boxes of the same size.

There are three translation operators used in $O(N)$ hierarchical methods to facilitate the hierarchical formation and evaluation of computational elements, namely, shifting the far–field potential representation of a child box to be with respect to the center of its parent box instead of with respect to the center of itself, converting a far–field to a local–field potential, and shifting the local–field potential representation of a parent box to be with respect to the centers of its children boxes instead of the representation being with respect to its own center. In all translations, the well–separateness when evaluating a computational element needs to be maintained.

An adaptive $O(N)$ method starts by constructing an adaptive hierarchy of recursively decomposed subdomains (boxes) of the original computational domain. Every subdivision results in four or eight equally sized boxes for two– and three–dimensions, respectively. Each decomposition is performed adaptively; a box is subdivided if it contains more than a predetermined number of particles. For nonuniform distributions, this rule results in a potentially unbalanced hierarchy, as shown in Figure 1 for two dimensions.

The adaptive methods group neighbor boxes associated with each box in the hierarchy into four lists of boxes, referred to as List–1, –2, –3, and –4. The grouping is according to the different relative locations of those boxes to the box "b" under consideration. Consequently, the four lists of boxes perform different interactions with the box "b", as defined in Table 2. With one–separation, the four lists of boxes relative to box "b" are shown for two dimensions in Figure 1.

Table 2: Interaction lists in adaptive $O(N)$ hierarchical methods.

List	Valid for	Definition
List–1	leaf boxes	non-well-separated boxes
List–2	all boxes	well–separated boxes that are children of the near–field of box b's parent
List–3	leaf boxes	well-separated offsprings of near–field boxes
List–4	all boxes	inverse of List–3

With the above definitions, an adaptive $O(N)$ hierarchical method [6] proceeds in the following steps:

Algorithm. *A generic adaptive $O(N)$ hierarchical method*

1. **Build Hierarchy.** See Section 4.1.

2. **Construct Interaction Lists.** The List–1, –2, –3 are constructed explicitly, via depth–first traversal of the hierarchy. List–4 is not explicitly constructed since it is the dual of List–3.

3. **Form leaf–level far–field potential.** Form far–field potential for leaf boxes directly from the particles inside.

4. **Upward Traversal.** The far–field potential of all internal boxes are formed by shifting and combining those of their child boxes.

5. **Compute List Interactions.** List–2 interactions include every box in the hierarchy, while List–1, –3, and –4 interactions only involve leaf boxes. In List–2 interactions, for every box, the far–field potential of all List–2 interactive boxes are converted into the local–field of that box. For every leaf box, the direct evaluation (the $O(N^2)$ method) is performed between particles inside and those in its List–1 interactive boxes; the far–field potential of its List–3 interactive boxes are evaluated at particles inside. For List–4 interactions, supposedly we need to convert the field of every particle in List–4 interactive boxes into the local–field of every leaf box. Since List–4 is the dual of List–3 and is not constructed explicitly, we simply convert the field of every particle in every leaf box into the local field of interactive boxes of that box that belong to List–3.

6. **Downward Traversal.** The local–field potential of parent boxes is recursively shifted to the center of child boxes and combined with the local—field potential contributed from List–2 and –4 interactions.

7. **Evaluate leaf–level local–field potential.** For every leaf box, the local–field potential is evaluated at the particles inside.

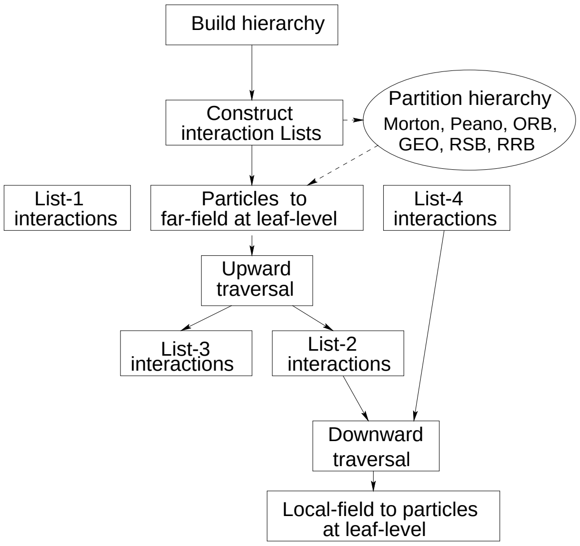


Figure 2: Task and data parallelism in adaptive $O(N)$ hierarchical methods. The oval box shows the partitioning stage parallel implementations.

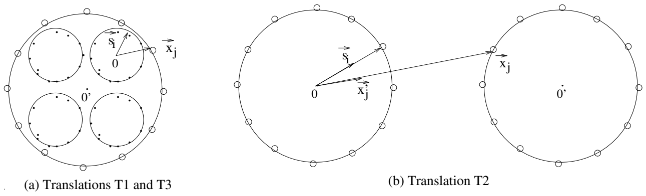


Figure 3: Translations as evaluations of the approximations in Anderson's method.

Table 3 summarizes the computations and active boxes in the above algorithm. The ten steps in Table 3 need not be executed in the order they appear, as long as the execution order satisfies the dependencies between the different steps, as shown in Figure 2.

Note that in the above generic method upward and downward traversals of the hierarchy involving List–2 boxes are performed one level at a time, which simplifies preservation of the dependencies. At all other computation steps, only leaf–level boxes are active.

Computational Elements in Anderson's Method Instead of using multipole expansions, Anderson's method [1] uses Poisson's formula for representing solutions of the underlying Laplace equation. Poisson's formula is expressed as an integration on a sphere. Given a numerical integration formula, the discretized integration becomes a summation of expansions with Legendre functions as the base functions and only relies on the potential values at the integration points. The far–field and local field potentials are called

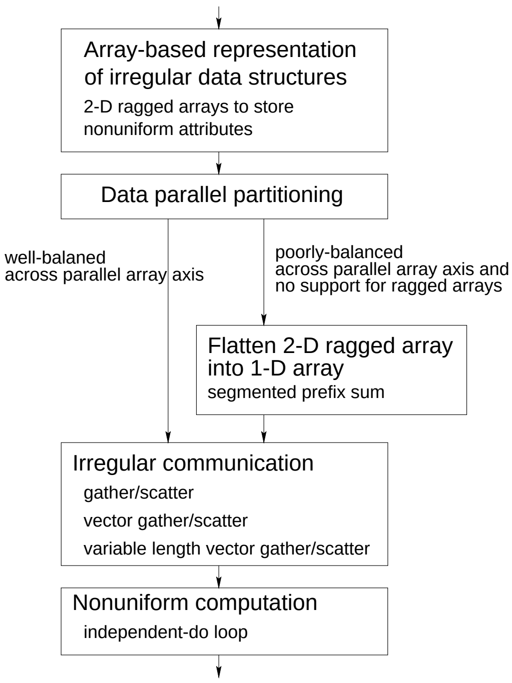


Figure 4: A data parallel formulation for irregular problems.

outer–sphere approximation and *inner–sphere approximation*, respectively. The three translation operations used in $O(N)$ methods, namely, shifting far–field potential, converting far–field to local–field potential, and shifting local–field potential become simple evaluations of the outer–sphere and inner–sphere approximations of the source spheres at the destination spheres of the translation, as shown in Figure 3. As we have shown in [13], the three translation operations can be performed as matrix–vector operations, and furthermore, the translation matrices only depend on the relative locations of source and destination spheres. Since there is only a finite number of such relative locations, the matrices can be precomputed and stored.

4 A Data Parallel Formulation

We now present an efficient data parallel formulation of an adaptive hierarchical N–body algorithm. See Figure 4 for its main components.

4.1 Linearizing Hierarchical Structures

The first step in our formulation embeds the irregular data structure used in the problem into one–dimensional (1–D) array structures. We use Morton or Peano–Hilbert orderings, shown in Figure 5. These are space–filling curves that linearly order a set of points in higher dimensions. To a certain degree, they preserve the physical adjacency of these

Table 3: Computational structure of adaptive $O(N)$ methods.

Step	Computation	Active boxes
1. Build adaptive hierarchy		
2. Construct interaction list–1,2,3		level–by–level
3. Form leaf–level far–field potential	particle to far–field	leaf–level
4. Upward traversal	far–field to far–field	level–by–level
5. List–1 interactions	particle to particle	leaf–level
6. List–2 interactions	far–field to local–field	all
7. List–3 interactions	far–field to particle	leaf–level
8. List–4 interactions	particle to local–field	leaf–level
9. Downward traversal	local–field to local–field	level–by–level
10. Evaluate leaf–level local–field potential	local–field to particle	leaf–level

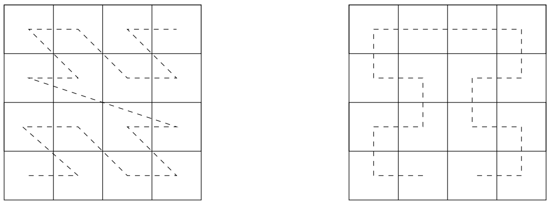
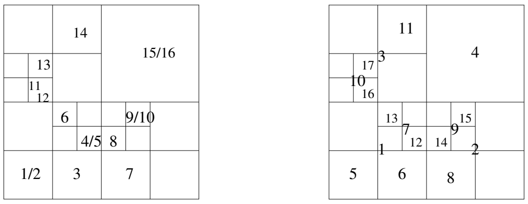


Figure 5: Morton and Peano–Hilbert ordering.



Particles sorted in Morton ordering

Level-by-level (Morton at each level) ordering of adaptive hierarchy boxs

Figure 6: An example of array representation of an adaptive hierarchy. Threshold is 2 particles/leaf–level box.

points. We use these linear orderings as a basis for the partitioning of the nonuniform hierarchy in our N–body problem. Both space–filling orderings we consider have been used in SPMD style programming for the N–body problem, in partitioning unstructured meshes [22], and in message–passing and shared–memory implementations of hierarchical N–body methods [29, 25]. In our data parallel formulation, we use them for linearizing the nonuniform hierarchical structures. The 1–D arrays so generated will be distributed across processors.

The nonuniformities in irregular problems often come from the different amount of computations and communications (together called attributes) associated with different elements in the data structures. The attributes are stored as elements along a local axis. Currently, there is no direct support for ragged arrays in HPF. Therefore, we use 2–D arrays with the local axis having an extent equal to the maximum number of attributes across the parallel axis. 2–D ragged arrays would allow for a more effective memory allocation.

For adaptive N–body methods, we build the hierarchy from the input particle distribution and linearize the hierarchy in the level–by–level order with Morton ordering at each level, as shown in the following data parallel algorithm.

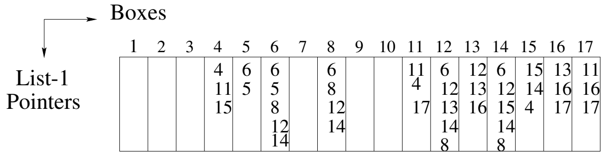
Algorithm. *Data parallel hierarchy building.*
Input. Particle coordinates stored in 1–D arrays, threshold s – maximum number of particles per leaf–level box

1. Sort particles so that for arbitrary threshold s, particles belonging to the same leaf–level box in the adaptive hierarchy to be built are stored contiguously. Morton ordering or Peano–Hilbert ordering with a degree of refinement sufficient to separate all particles (into different boxes) can be used to achieve this. In practice, the degree is bounded by $-\log_2^{\epsilon}$, where ϵ is the machine precision.

2. Construct the hierarchy via scanning on the sorted particle arrays, counting the number of particles per box at the current level and use a mask array to record whether further subdivision of a box (a segment of the particle arrays) is necessary. For every box (corresponding to an array element in the array representation), the construction builds and stores the parent and first child pointers, a pointer to the first particle in the particle arrays and the number of particles in that box. Also stored are the coordinates of the box centers and a logical array for recording whether the box is a leaf box.

Figure 6 shows an example of 16 nonuniformly distributed particles sorted in Morton ordering and the generated linear ordering of the boxes in the corresponding adaptive hierarchy.

The second step uses scan operations h times, where h is the final hierarchy depth. This may appear as inefficient. However, in our simulations the total amount of communication is a low order term compared to the communication



integer list1(max_lst1,num_boxs)
!hpf$ distribute list1(*, block)

Figure 7: Array representations for pointers for List–1 interactive boxes.

in the list interactions. An analytic justification for this observation is given in Section 4.2.

The computations in the adaptive hierarchical N–body methods involve only leaf–level boxes for most steps. Therefore, we store store leaf–level boxes in a separate array, denoted as the *leaf–level box array* in the following. In addition, we reshape the sorted 1–D particle array into a 2–D particle array with a parallel axis for the leaf–level boxes and a local axis for storing particles in each leaf–level box. This third array structure will be aligned with the leaf–level box array, and the particle and leaf–level box interactions requires no communication.

Construction of the three interaction lists in step 2 is performed via several downward traversals of the hierarchy, scanning the entire array of the hierarchy. Details can be found in [12]. The lists constructed are stored in 2–D pointer arrays. A parallel axis is used for the representation of all the boxes in the hierarchy and a local axis is used to store pointers for the same box. Figure 7 shows the representation of List–1 for the example hierarchy in Figure 6. Since the number of each kind of interactive boxes may be different for different boxes, the memory usage of the above 2–D array representation may be inefficient. However, if the particle distribution have some reasonable measure of continuity in the density distribution, then the average and the largest numbers of interactive boxes can be expected to differ by a modest number. For example, for one million particles with the 3–D Plummer distribution, the average and largest number of various types of boxes per box "b" are 25.5 and 47 for List–1, and 149 and 189 for List–2.

4.2 Data Parallel Partitioning and Load–Balancing

The second step of our data parallel formulation is to partition the irregular data structures for load–balancing and data locality. Mathematically, it has been proved [21, 28] that both unstructured meshes and adaptive N–body graphs have provably good partitions, i.e. the ratio of interpartition data movement and intrapartition computation is of the same order as the ratio for regular grids of the same size.

For adaptive N–body methods, once the hierarchy is built and the interaction lists are constructed, the communication and computation cost associated with each box can be easily calculated according to the box counts of the interaction lists and the number of particles in each leaf–level box. Since our partitioning schemes assume alignment of 2–D particle arrays with the leaf–level box array, the cost of particle–box interactions at the leaf–level is counted as part of the leaf–level boxes; all the costs are associated with the boxes.

4.2.1 Mathematical Bases

The interaction lists of the boxes define the *computation graph* for an N–body problem, the *N–body graph*. N-body graphs have a higher node degree than typical finite element meshes, especially when the distribution of particles is nonuniform. The following result of Teng [28] gives an upper bound on the amount of interaction between partitions as a function of the height of the N–body graph.

Theorem 4.1 *Let G be an N-body graph of a set of particles located at $P = \{p_1, ..., p_n\}$ in R^d ($d = 2$ or 3). If the height of the hierarchical tree for P is h, then G can be partitioned into k equal computational weighted subgraphs by removing at most $O(k^{1/d}h^{1/d}n^{1-1/d})$ boxes.*

The proof of the above theorem given in [28] further shows that a partitioning of this quality can be found by the geometric partitioning algorithm (GEO) of Miller, Teng, Thurston, and Vavasis [21] and a variant of spectral partitioning given by Spielman and Teng [27].

4.2.2 Load Balancing and Partitioning Heuristics

Orthogonal recursive bisection (ORB) and Morton and Peano–Hilbert ordering have been used to partition particles in the Barnes–Hut method [24, 29, 25] and to partition boxes in an adaptive fast multipole method [25].

We have developed a library of data parallel partitioning schemes in HPF, as summarized in Table 4. As a reference, we include the level–by–level/Morton (LBL) ordering generated by our data parallel hierarchy building algorithm. In addition to the above heuristics, we also developed an extension of ORB called rotational recursive bisection (RRB). Instead of always alternating the partitioning in x–, y–, and z– directions as in ORB, RRB tries independently for each subpartition, a sequence of line or plane partitioning with random angles, examines the quality of the edge cuts, and chooses the best partition. All above partitioning heuristics make no guarantee on the quality of partitions, but they are very cheap to compute. Except RRB, these heuristics do not make use of the edge connectivity.

The advantage of programming in high–level languages such as HPF is manifested in the data parallel implementations of these heuristics: the partitioning algorithms takes

Table 4: A library of data parallel partitioning and load–balancing schemes. Nodal weights represent amount of computations

Method	Input	load balancing quality	
		nodal weights	edge weights
ORB	nodal weight + coord.	good	unknown
Morton	nodal weight + coord.	good	unknown
Peano–Hilbert	nodal weight + coord.	good	unknown
Level–by–level	nodal weight + coord.	good	unknown
RRB	nodal weight + coord. + edge	good	unknown
GEO	nodal weight + coord. + edge	provably good	provably good
RSB	nodal weight + edge	provably good	provably good

only a few lines of arithmetic array operations plus a subroutine call to the HPF sort intrinsic.

4.2.3 Provably Good Partitioning Algorithms

Geometric partitioning (GEO) [21] and a variant (by Spielman and Teng [27]) of recursive spectral bisection (RSB) [23] both offer guarantees on the quality of the partitions. GEO lends itself to efficient data parallel implementations, as shown in [15]. Extensive use of *sampling* can be used to reduce the computational complexity without a significant degradation in the quality of the partitions. Using this sampling technique GEO is computationally less demanding than a straight–forward implementation of RSB.

Recently, the idea of multi–level contraction to reduce the size of the graphs being partitioning has been introduced to RSB with promising results [18, 10]. Currently, we are incorporating a similar idea in our data parallel GEO implementation.

4.2.4 Uneven Array Distributions for Load–balancing

A fundamental issue with partitioning adaptive hierarchies is that because of the potentially uneven weights associated with the boxes, the partitions generated may contain an uneven number of array elements, and therefore an uneven distribution of arrays is required. Although this distribution feature is included as an approved extension in the recently announced HPF 2.0 [11], no commercial compilers available today supports it yet.

We have developed a simple padding scheme for emulating uneven array distributions in HPF 1.0, as shown in Figure 8. The emulation is achieved by padding with dummy elements array segments of an extent less than the longest, so that these segments become of the same length as the longest segment. A default block distribution of the padded array will result in segments of different extents in the original array being distributed among the processors. This emulation requires no change to the HPF code: computation is only applied to real elements through the use of `independent-do` loops, and communication is performed under mask.

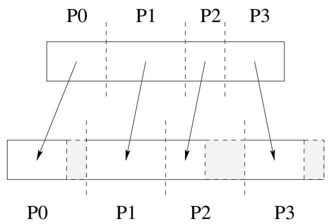


Figure 8: Emulating uneven array distribution of s in HPF.

The extra memory usage required for the above emulation is proportional to the padding factor, i.e. the size of the padded array over that of the original array. In the test cases in Section 5, the padding factor is quite small, 1.13 and 1.15, respectively.

4.3 Expressing Nonuniform Computation and Irregular Communication: Low Nonuniformity

As described in Section 3.1, due to the lack of efficient support for ragged arrays in HPF, we use 2–D arrays to store the uneven amount of attributes. We distinguish two ways of expressing the nonuniform computations and irregular communications according to how uniform the amount of attribute data is. In this section, we use List–1 interactions in adaptive N–body methods as an example for relatively high uniformity. In the next section, we use List–3 interactions as an example for low uniformity.

List–1 interactions involve direct evaluation between particles in each leaf–level box and particles in its List–1 boxes. Using the owner–computes rule, each box needs to first fetch the particles from its Lists–1 boxes, and then perform the direct evaluation. Therefore, associated with each box is a weight for the amount of communication in fetching nonlocal List–1 boxes [2] and a weight for the amount of computation. Our partitioning algorithms described in Section 4.2 try to

[2] Local copying of local List–1 boxes is necessary in the data parallel formulation. Depending on the relative communication and computation speed of the parallel platforms, its cost is often insignificant compared with the nonlocal fetch.

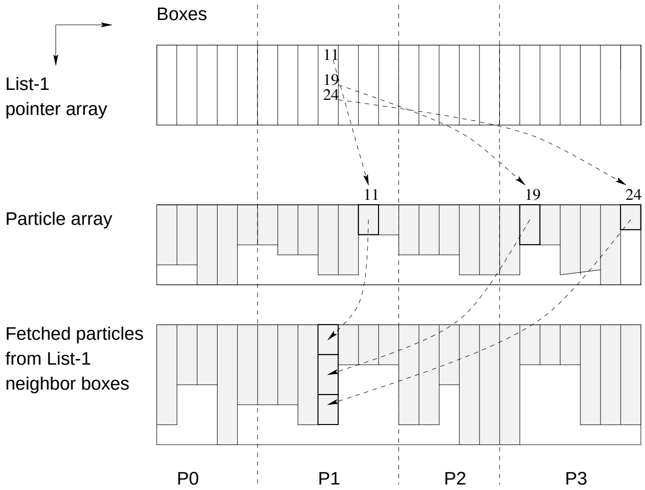


Figure 9: Gather communication in List–1 interactions.

balance the aggregate computation weights of all the boxes on each processor while minimizing the aggregate nonlocal fetching.

Irregular Communication. First, columns of the List–1 pointer array corresponding to leaf–level boxes are extracted into a compressed List–1 array, upon which all the List–1 interactions will be performed.

Figure 9 shows the gather communication in List–1 interactions for fetching the particles in every box's List–1 interactive boxes. In HPF, the gather communications are expressed using array indirections. Since our partitioning algorithms balance the total actual communication and computation for all List–1 interactions, i.e. it calculates the communication load according to the actual number of particles per leaf–level box, a perfectly balanced implementation of the communication requires (1) prefetching and storing particles in all interactive boxes, and (2) variable–length vector gather operations to fetch only useful particle data. Currently, variable–length vector gather operations are not supported in HPF. In addition, the memory required to store the prefetched particles will be high. The alternative of scanning the List–1 pointer array one slice at a time defeats the intended communication balance achieved by the partitioning.

Our implementation takes an input parameter specifying the total memory per node that can be used for the prefetch. The code automatically decides how many slices it can fetch each time, and interleave the direct evaluation with the gather communications.

Nonuniform Computation. The particles belonging to the List–1 interactive boxes fetched in each batch are stored and compressed into a 2–D array with a parallel axis for the boxes and a local axis for the particles in each box. Since each fetch may receive different numbers of particles for different boxes, the resulting 2–D array may be ragged. The direct evaluation now becomes all–to–all interactions between the corresponding columns of the 2–D particle array (ragged) and the prefetched List–1 interactive particle array. This all–to–all interaction is expressed using an independent–do loop for the parallel axis for the boxes and two sequential do loops for looping through the two corresponding columns.

4.4 Expressing Nonuniform Computation and Irregular Communication: High Nonuniformity

If the average and maximal amount of attribute data differ significantly, such as the number of List–3 pointers in adaptive hierarchical methods, the extra memory usage can be unacceptable if the 2–D array representation discussed above is used. Therefore, for highly nonuniform use of 2–D arrays, we explicitly flatten the arrays into 1–D arrays, and rely on segmented prefix sum and spread operations for the data movement.

Figure 10 illustrates the flattening technique in List–3 interactions, in which only leaf–level boxes are active. A compressed List–3 pointer (ragged) array of the same extent as the number of leaf–level boxes contains pointers to the List–3 interactive boxes. The 2–D particle arrays (for x–,y–,z–coordinates and charge) are also ragged. A 2–D potential vector array contains the potential at the integration points of outer–sphere approximations for all boxes. The goal is to, for every box, fetch the potential vectors pointed to by its List–3 pointers, and evaluate the corresponding sphere approximation at the particles inside.

As shown in Figure 10, the flattening technique flattens the 2–D ragged pointer arrays into a 1–D pointer array using a scatter operation. It then gathers the potential vectors using the flattened pointer array. It also needs to expand the particle arrays so that each particle vector is duplicated to correspond to the flattened pointer array. The actual computation, i.e. evaluations of outer–sphere approximations can be performed using `independent-do` loops, inside which a serial do loop is used to loop through the variable–length particle vectors. Finally, a segmented prefix sum is used to accumulate computed force/potential on the particles within each segment, and a scatter is used to put the result back into the force/potential particle array. Again variable–length scatter is desired to move only useful data.

4.5 Other Steps of Adaptive N–body Methods

Leaf–level particle–box interactions are expressed as interactions between the leaf–level box array and the 2–D particle array, and no communication is required since the two arrays are aligned in the parallel memory.

The hierarchy traversal is performed level–by–level. At every level, gather and scatter operations are used to ex-

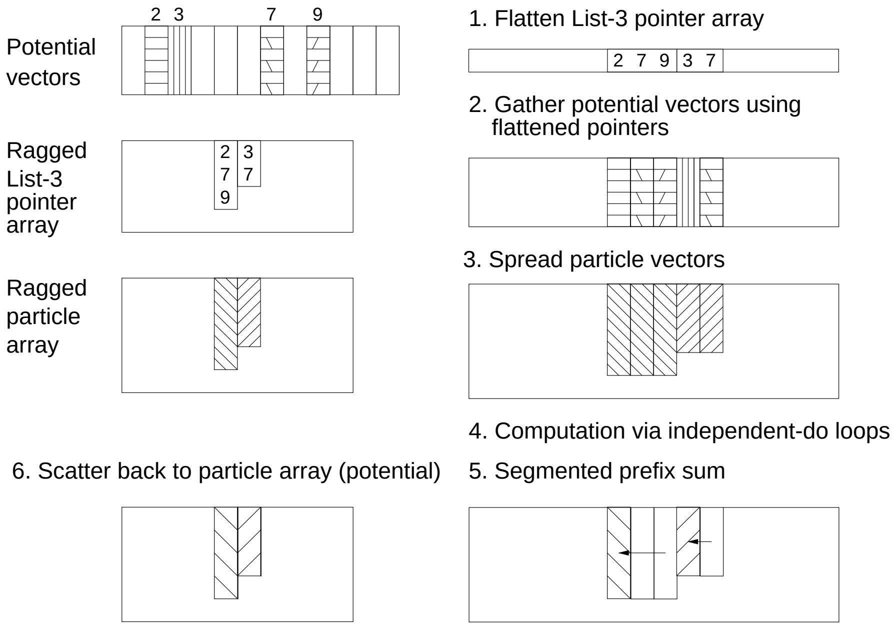


Figure 10: The flattening technique for List–3 interactions.

tract/embed the small working arrays corresponding to the current level boxes and that of the next level, i.e. between parent boxes and children boxes. The efficiency of the communication relies on a partitioning scheme that preserves good parent–child locality. As show in the next Section, different partitioning schemes often perform well for some steps and not so well for other steps of of the adaptive method. Therefore, there is no obvious choice of method.

5 Experimental Results

We have developed HPF implementations of the partitioning algorithms mentioned above, as well as of a 3–D adaptive version of Anderson's N–body method. The experimental results below are for a 3–D Plummer (PLUMMER) and a 250 Megaparsec diameter Cold Dark matter model (DARK), as shown in Table 5. All results are from runs on a 16 wide–node IBM SP2. The HPF compiler by the Portland Group Inc. was used for all simulations.

5.1 Partitioning Results

Table 6 shows the measurements of the partitioning quality for arrays representing active boxes in List–1 and List–2 interactions, including the average and maximal numbers of remote references and floating–point operations among the partitions. The measurements are collected using masked arrays and segmented scan operations.

For List–1 interactions, the partitioning is made with respect to leaf–level boxes. GEO generates the best quality

Table 5: Two test cases. The threshold is 128 particles/leaf–level box.

Characteristics	PLUMMER	DARK
Number of particles	1,024,000	955,728
Hierarchy depth	12	11
Number of boxes	25046	43442
Number of leaf boxes	27696	37668
% of boxes at		
level 1	0.0	0.0
level 2	0.2	0.1
level 3	0.2	0.9
level 4	0.3	5.8
level 5	0.3	25.2
level 6	1.0	22.6
level 7	1.8	18.8
level 8	4.2	15.1
level 9	8.3	8.8
level 10	16.3	2.5
level 11	30.8	0.1
level 12	36.7	
Max/avg number of		
List-1 boxes per box	47/25.5	82/19.5
List-2 boxes per box	189/149.2	189/50.5
List-3 boxes per box	95/4.55	224/10.8
Max/avg child boxes per box	8/7.99	8/7.52
Padding factor with Morton ordering		
for hierarchy array	1.11	1.11
for leaf–level box array	1.13	1.15

Table 6: Partitioning results. 16 partitions are generated. Arrays are initially in the LBL ordering. All partitioning algorithms generate perfect computation balance. For PLUMMER , List–1 interactions have 4600 Mflops/node which runs in 97.7 seconds. List–2 interactions have 2950 Mflops/node which runs 34.9 seconds. The corresponding numbers for DARK are 1420 Mflops/node, 43.2 seconds, 2470 Mflops/node, and 53.4 seconds.

Method	PLUMMER		DARK	
	max. Mbytes/node	Time (sec.)	max. Mbytes/node	Time (sec.)
List–1	(31136 nodes, 671712 edges)		(37668 nodes, 734988 edges)	
LBL	7.10	24.2	5.94	21.1
Morton	6.72	21.0	5.54	19.9
ORB	6.80	22.1	5.30	17.2
RRB (10 trials)	4.80	20.2	5.21	17.9
GEO (10 trials)	4.89	19.8	4.60	17.5
GEO (2 trials)	5.20	20.9	5.54	18.5
List–2	(33040 nodes, 4553496 edges)		(43442 nodes, 2192756 edges)	
LBL	7.6	9.45	2.52	6.49
Morton	13.1	9.85	4.03	6.65
ORB	14.2	10.2	3.37	6.76
RRB (10 trials)	15.0	10.8	3.79	6.70
GEO (10 trials)	15.2	10.9	2.91	6.78
GEO (2 trials)	15.4	11.2	3.87	7.00

partitions of the algorithms considered. For List–2, the partitioning is made with respect to the entire hierarchy of boxes. LBL generates partitions with the best locality, while Morton and Peano orderings generate partitions of a quality similar to that of GEO. The reason for this behavior is that List–2 contains neighbor boxes at the same level, and the level–by–level ordering with Morton ordering for each level maintains the best locality. For both List–1 and List–2, RRB generates significantly better partitions than ORB.

5.2 Overall Code Performance

Table 7 shows the total running time and its breakdown with respect to the ten steps, and the partitioning step of our current adaptive Anderson's method in HPF. As shown in Table 7, the communication currently accounts for 41% and 49% of the total running time for the two investigated distributions. This high communication times are mostly due to the poor performance of the unoptimized gather/scatter runtime system subroutines generated by the pghpf compiler version 2.1. It is expected that an order of magnitude improvement be achieved from highly optimized gather/scatter subroutines. On the computation side, The particle–particle direct evaluation in List–1 interactions in PLUMMER only achieves about 18% efficiency. With a good optimizing nodal compiler and highly optimized runtime system for gather/scatter operations, our code for these simulations is expected to achieve a 20% efficiency.

6 Conclusions

We have presented a general data parallel formulation for nonuniform problems, and demonstrated its use in implementing an adaptive hierarchical N–body method. To our knowledge, this is the first data parallel implementation of such a method. Our experiments provide the first promising evidence that a data parallel approach can be used to efficiently solve highly irregular problems. Key features in HPF compilers for high efficiency in nonuniform hierarchical problems are (1) efficient nodal code generation (2) efficient runtime support, particularly for gather/scatter and parallel prefix operations, (3) language/compiler/runtime support for uneven distributions of arrays for load–balancing, and (4) language/compiler support for ragged arrays for efficient memory usage. Our data parallel N–body code provides a much needed "benchmark" code for evaluating and improving HPF and its compilers.

Acknowledgment

We would like to thank Doug Miles and Paul Kinney for their help with performance tuning of the adaptive code using the pghpf compiler, and Michael Warren for providing the Cold Dark matter model test case. We also thank the Cornell Theory Center for providing the IBM SP2 access.

References

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

Table 7: Running time for one million particles with 3–D Plummer distribution on a 16 wide–node IBM SP2, using Morton ordering partitioning. The RMS potential error compared with the direct method is 1.24E-05.

Step	PLUMMER				DARK			
	Running time (sec.)	% of total FLOPS	Comm. portion	Effi. (%)	Running time (sec.)	% of total FLOPS	Comm. portion	Effi. (%)
1. Build hierarchy	9..35	0.00%	100.0%	0.00	12.7	0.00%	100.0%	0.00
2. Construct interaction lists	73.7	0.00%	100.0%	0.00	176.	0.00%	100.0%	0.00
3. Form leaf–level far–field	6.33	1.83%	21.3%	14.5	6.27	1.69%	25.6%	13.7
4. Upward traversal	2.66	0.16%	72.1%	3.03	3.78	0.23%	73.3%	3.11
5. List–1 interactions	124.	34.3%	21.0%	13.9	77.5	18.3%	31.0%	12.0
6. List–2 interactions	58.2	22.1%	40.0%	19.1	66.7	10.5%	35.2%	8.01
7. List–3 interactions	37.4	21.6%	18.6%	29.0	139.	36.8%	9.65%	30.5
8. List–4 interactions	48.5	17.6%	15.2%	18.3	84.1	30.1%	4.13%	18.2
9. Downward traversal	3.71	0.16%	81.6%	2.17	7.44	7.44%	86.6%	1.58
10. Evaluate leaf–level local–field	5.59	2.23%	38.8%	20.1	12.5	2.07%	7.00%	8.4
11. Partitioning and reordering	1.93	0.00%	100%	0.0	3.07	0.00%	100%	0.0
Total	372	100%	41.0%	13.5	512	100%	49%	9.90

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

[3] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, Feb. 1990.

[4] J. A. Board Jr., Z. S. Hakura, W. D. Elliott, D. C. Gray, W. J. Blanke, and J. Leathrum Jr. Scalable implementations of multipole–accelerated algorithms for molecular dynamics. In *Proc. Scalable High Performance Computing Conference SHPCC94*. IEEE Computer Society Press, May 1994.

[5] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *JACM*, 42:67–90, 1995.

[6] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM J. Sci. Statist. Comput.*, July 1988.

[7] K. Esselink. The order of Appel's algorithm. *Information Processing Letter*, 41:141–147, 1992.

[8] P. O. Fredrickson and O. A. McBryan. Normalized convergence rates for the PSMG method. *SIAM J. Sci. Stat. Comp.*, 12(1):221–229, 1991.

[9] L. F. Greengard. *The rapid evaluation of potential fields in particle systems*. MIT Press, 1988.

[10] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '96*, Philadelphia, PA, November 1996.

[11] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0.δ*, October 19, 1996.

[12] Y. Hu. *Efficient Data Parallel Implementations of Hierarchical N–body Algorithms*. PhD thesis, Harvard University, 1997.

[13] Y. Hu and S. L. Johnsson. A data parallel implementation of hierarchical N–body methods. *International Journal of Supercomputing Applications and High Performance Computing*, 10(1):3 – 40, 1996.

[14] Y. Hu and S. L. Johnsson. Implementing $O(N)$ N–body algorithms efficiently in data–parallel languages. *Journal of Scientific Programming*, 5(4):337 – 364, 1996.

[15] Y. C. Hu, S.-H. Teng, and S. L. Johnsson. A data–parallel implementation of the geometric partitioning algorithm. In *Proc. of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, MN, March 1997.

[16] Z. Johan, K. K. Mathur, S. L. Johnsson, and T. J. Hughes. An efficient communication strategy for Finite Element Methods on the Connection Machine CM–5 system. *Computer Methods in Applied Mechanics and Engineering*, 113:363–387, 1994.

[17] Z. Johan, K. K. Mathur, S. L. Johnsson, and T. J. Hughes. Scalability of finite element applications on distributed–memory parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 119(1 – 2):61 – 72, November 1994.

[18] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report Tech. Rep. CORR 95-035, University of Minnesota, Dept. Computer Science, June 1995.

[19] P. Liu. *The parallel implementation of N–body algorithms*. PhD thesis, Yale University, 1994.

[20] K. K. Mathur, A. Needleman, and V. Tvergaard. Dynamic 3-d analysis of the charpy v–notch test. *Modelling and Simulation Materials Science and Engineering*, 1(4):467–484, July 1993.

[21] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Finite element meshes and geometric separators. *SIAM J. Scientific Computing*, to appear, 1997.

[22] C.-W. Ou, S. Ranka, and G. Fox. Fast and parallel mapping algorithms for irregular problems. *Journal of Supercomputing*, 1(23), 1997.

[23] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, 1990.

[24] J. K. Salmon. *Parallel Hierarchical N–Body Methods*. PhD thesis, California Institute of Technology, 1990.

[25] J. Singh, C. Holt, J. Hennessey, and A. Gupta. A parallel adaptive fast multipole method. In *Supercomputing'93*, pages 54 – 65. IEEE Computer Society, Los Alamitos, 1993.

[26] J. Singh, C. Holt, T. Ttsuka, A. Gupta, and J. Hennessey. Load balancing and data locality in hierarchical N–body methods. Technical Report CSL–TR–92–505, Stanford University, 1992.

[27] D. A. Spielman and S.-H. Teng. Spectral partitioning works: planar graphs and finite element meshes. In *Proceedings of the 37th Annual Symposium on Foundation of Computer Science*, pages 96–107, 1996.

[28] S.-H. Teng. Provably good partitioning and load balancing algorithms for parallel adaptive n-body simulation. *SIAM J. Sci. Comput.*, to appear 1996.

[29] M. Warren and J. Salmon. A parallel hashed oct–tree N–body algorithm. In *Supercomputing'93*, pages 12 – 21. IEEE Computer Society, Los Alamitos, 1993.

[30] F. Zhao. An $O(N)$ algorithm for 3–dimensional N–body simulations. Technical Report AI–TR–995, AI Lab, MIT, 1987.

[31] F. Zhao and S. L. Johnsson. The parallel multipole method on the Connection Machine. *SIAM J. Sci. Statist. Comput.*, 12(6):1420–1437, Nov. 1991.