M-Tree: A Parallel Abstract Data Type for Block-Irregular Adaptive Applications

Q. Wu[1], A.J. Field and P.H.J. Kelly

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ.
Email: ajf, phjk@doc.ic.ac.uk

Abstract. This paper describes an abstract data type called M-Tree — a generalization of a quadtree which captures both the data structure and computational structure common to many adaptive problems in science and engineering. It is equipped with a rich set of access functions including *higher-order* operators describing commonly used computational patterns in parallel adaptive computations. This provides a uniform high level abstraction of a wide range of applications including adaptive mesh refinement and adaptive particle simulation and thus enables such applications to be constructed systematically and efficiently. We present examples in which an M-tree is used to solve both an adaptive heat-flow problem and N-body particle simulation. The structured abstraction of commonly-occurring computation patterns in the application provides us with the opportunity to investigate various approaches to load balancing and communication minimization using caching and other techniques. These optimizations are applicable to other problems with a similar structure.

1 Introduction

In this paper we present an abstract data type called "M-Tree", a hierarchical data structure which is used for organizing block-irregular computations generated by recursive domain decomposition. The M-Tree captures both the data structures and computational structures common to many adaptive problems in science and engineering. It is equipped with a rich set of access functions including *higher-order* operators describing commonly-used computational patterns in parallel adaptive computations. This provides a uniform high level of abstraction for a wide range of parallel applications including adaptive mesh refinement and adaptive particle simulation. Thus, with an efficient parallel implementation of the M-tree data structure and its operators, a wide variety of such applications can be constructed systematically.

Without a suitable layer of abstraction, users have to deal with both the application's problems, and maintenance of the tree data structure itself. This

[1] Qian Wu is now with CHAM, 40 High Street, Wimbledon Village, London, UK.

problem becomes more serious for efficient parallel solutions since performance considerations such as load balancing and communications minimization have to be taken into account during construction and manipulation of the dynamic tree structure.

The relationship between M-tree applications and our current implementations is illustrated in Fig. 1.

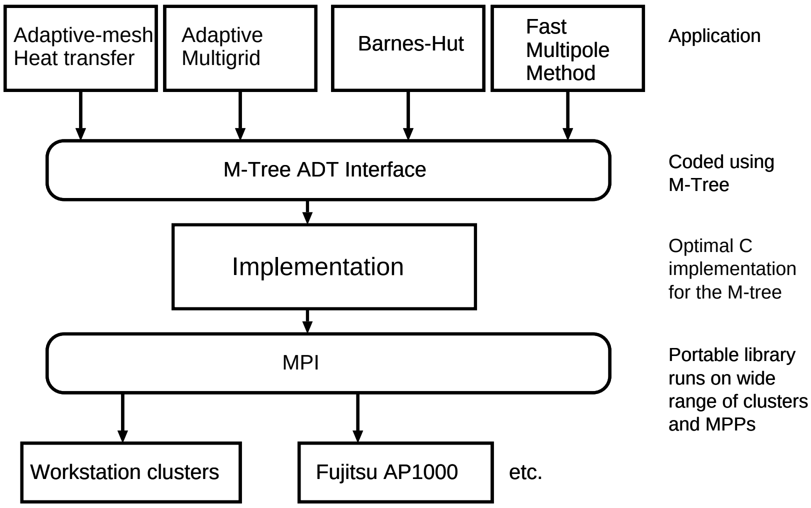

Fig. 1. Relationship between M-tree applications and our current implementations.

In this paper we aim to demonstrate the power and versatility of the approach, and to demonstrate that M-Tree leads to clear and succinct solutions without compromising performance. We have studied a number of applications of the M-Tree and present here two examples of its use: a simple heat conduction problem, and an implementation of the Barnes-Hut N-body particle simulation algorithm. We show that the structured abstraction of commonly-occurring computation patterns in the application provides the opportunity to investigate various approaches to load balancing and communication minimization using caching and other techniques. These optimizations are applicable to other problems with similar structure.

Overview of this paper

In Section 2 we present the basic concepts of the M-tree and its programming style. In Section 2.2 we present a simple example of its use. We then present an N-body simulation as a case study in Section 3. The implementation of the M-tree is presented in Section 4. Section 4.3 shows the performance benefits of some of our optimisations. Section 5 places our work in the context of related efforts, and Section 6 concludes with directions for developing this research.

2 The M-Tree abstract data type

The M-Tree is designed to support a wide range of applications. The common basis is a regional mesh tree in which each node represents a region of the domain and its subtrees are subregions overlaying the region of the parent node. The common computational patterns we need to support include the following:

- parallel computation upon a group of nodes on the tree
- flexible dynamic recursive decomposition
- control for dynamic expansion and contraction over the tree
- access to neighbouring regions, whatever their level of decomposition and
- efficient support for hierarchical treatment of long-range interactions

An M-Tree is a regional mesh tree in which the degree of each nonleaf node is R_x in the x-dimension, R_y in the y-dimension and R_z in the z-dimension. Each node represents a region of the domain and its subtrees are subregions overlaying the region of the parent node. Thus a tree is called a quadtree when $R_x = R_y = 2, and\ R_z = 1$. It can be regarded as a generalization of *quadtree* representing a class of hierarchical data structures whose common property is that they are based on the principle of recursive decompositions of space [17].

The simplified definition of a mesh tree can be described using the following C code:

```
struct MeshTree {
        int status;
        struct Region domain;
        struct NeighbourTree** NbTrees;
        struct MeshTree* parent;
        MeshTree newlevel[rx][ry][rz];
        NodeType* vdata;
}
```

where

`status` indicates if the current node is childless,
`domain` includes the upper and lower bounds of geographic range,
`Nbtrees` is the internal link to a list of the adjacent nodes,
`parent` is the internal link to the parent node,
`newlevel` are the internal links to the subtrees
`NodeType` is the user-defined node-based variable type.

This representation is internal to the ADT and so need not be understood by the applications programmer.

2.1 The M-tree's Operators

The operations over an M-Tree can be divided into *first-order* and *higher-order* operators. The first-order operators perform basic query and update on each node of the M-Tree. The higher-order operators abstract commonly-used patterns of parallel adaptive computations. They are higher-order in the sense that they are parameterized by user-defined functions for local and global computation. We present a brief overview of these operators as follows:

1. **Tree construction:**
   `MT_Init`:
   uses a user-defined partition operation to distribute global data to initialize the M-Tree.
2. **Computational operators - leaf-based:**
   `MT_Map_Leaf`:
   applies a user-defined region-based operation to each leaf node in parallel. The local operation is accompanied by a communication stencil, so that the function can access neighbouring elements.
   `MT_Reduce_Leaf`:
   performs reduction for all leaf nodes in parallel.
   `Ex_comm_Leaf`:
   provides communication among all leaf nodes, according to a specified stencil.
   `MT_Bcast_Leaf`:
   broadcasts to all leaf nodes in parallel.
   `MT_Map_Leaf_Env`:
   applies a user-defined region-based operation to each leaf node in parallel with respect to a global environment parameter. The local operation is accompanied by a filter function that restricts the range of the global environment (the environment is often the M-Tree as a whole, and the filter function specifies which subnodes are needed for each leaf computation).
3. **Computational operators - level-based:**
   `MT_Map_Level`:
   applies a user-defined region-based operation to each node on a given level in parallel, using a stencil as above.
   `MT_Reduce_Level`:
   performs reduction for all nodes on a given level in parallel.
   `Ex_comm_Level`:
   provides communication among all nodes on a given level.
   `MT_Bcast_Level`:
   broadcasts to all nodes on a given level in parallel.
4. **"All-at-once" tree operators:**
   `MT_Up_Pass`:
   traverses the M-tree level-wise from the bottom up and applies user-defined operations to leaf and internal nodes.

MT_Down_Pass:
: traverses the M-tree level-wise top-down and applies user-defined operations to leaf and internal nodes.

MT_Adaptive:
: updates the grid hierarchy, as required, for example after application-dependent error analysis.

MT_Gather:
: collects the elements of the M-Tree into a user-specified data structure.

2.2 Example: adaptive-mesh Jacobi solver for heat flow problem

We consider a simple heat conduction example to illustrate how the M-tree is used in programming a continuum system. Consider a material being heated by boundary temperature difference or an internal hot spot. The two-dimensional steady-state thermal conduction with no internal heat generation is governed by Laplace's equation, $\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$. and boundary condition equations. Such a system of partial differential equations can be solved by numerical discretization. The difference equations can be obtained as:

$$T_{i,j}^{t+1} = \frac{1}{4}(T_{i+1,j}^t + T_{i-1,j}^t + T_{i,j-1}^t + T_{i,j+1}^t)$$

With a suitable grid and a set of finite difference equations, the approximation at each grid point can be achieved by an iterative linear equation solver. The solver computes an initial state of the grid and then applies the finite difference equation iteratively until a certain convergence condition is met. For this simple example, we use the Jacobi method [16]. The global temperature distribution is then approximated by computing the difference equations of the subregions. Regions where accuracy is not satisfactory may be refined further. Such an approximation involves the interaction between subgrids and the global grid in a way similar to the multigrid method [16]. That is, the coarse grid is interpolated to form the initial state of the refined subgrids.

The implementation using an M-Tree is shown in Figure 2. The tree is created from the initial domain using `MT_Init` parameterized by a user-defined C function, `decompfun`, which specifies how the initial domain is decomposed and partitioned among the processors.

Jacobi relaxation is applied repeatedly until convergence is achieved or the number of iterations exceeds `maxconv`. Each relaxation iteration uses `Ex_Comm_Leaf` to exchange halo boundary between the leaf nodes. The user defined function `node_to_temp` extracts the boundary temperature values to be communicated, and the interpolation/un-interpolation functions `intp` and `unintp` are used for boundaries on different refinement levels. `MT_Map_Leaf` is then applied, using the user-supplied Jacobi function `relax_temp`, which operates the regular grids at the tree's leaves.

If convergence is not achieved, `MT_Map_Leaf` is used to interpolate and initialize a finer mesh on each leaf. A similar Jacobi relaxation procedure is then applied to each leaf node's refined mesh.

```
MT_Init(TAG_MESH, buffer, domain, decompfun);
for (outer_iter=0; outer_iter<maxiters; outer_iter++) {
  /* first relaxation on all leaf nodes */
  for (iter=0; iter<maxconv; iter++) {
   Ex_Comm_Leaf(node_to_temp, intp, unintp);
   MT_Map_Leaf(relax_temp);
   /* until converges or exceeds maxconv */
  }
  if(not convergent) {
   /* interpolate each leaf to a finer mesh */
   MT_Map_Leaf(intpt_temp);
   for (iter=0; iter<maxconv; iter++) {
     Ex_Comm_Leaf(node_to_finertemp, intp, unintp);
     MT_Map_Leaf(relax_finertemp);
     /* until converges or exceeds maxconv */
   }
   /* get global maximum truncation error */
   MT_Reduce_Leaf(MAX, calc_trunc_error, &m);
   /*broadcast to all leaf nodes to set refinement tag */
   MT_Bcast_Leaf(set_tag, m);
   MT_Adaptive(refinefun, intpt, unintpt);
  }
}
```

Fig. 2. Heat Transfer Example

`MT_Reduce_Leaf` uses the user-defined function `calc_trunc_error` to estimate the truncation error from the difference between the original mesh and the finer mesh, and find where this is maximum. `MT_Bcast_Leaf` is used to tag the nodes where refinement is needed. Finally `MT_Adaptive` refines the tagged regions using the user-defined interpolation functions.

3 Application to particle simulation

The application we consider here is the Barnes-Hut algorithm [1] for modelling the behaviour of interacting particles in space; there are similar applications in, for example, molecular biology, plasma physics and fluid mechanics. The algorithm is based on the observation that, while forces from nearby particles must be considered separately, forces from a group of particles far enough away can be approximated as one equivalent particle. In three dimensions it uses an oct-tree to store particle information or the collective particle information in the subcubes. The same problem can be cast in two dimensions—the structure is then a quad-tree. Each particle calculates the forces acting on it by querying the hierarchical quadtree/oct-tree. Figure 3 shows how the oct-tree Barnes-Hut

```
... set up initial global_particles, global_domain ...

for (iter=0; iter<maxiter; iter++) {
   /* create oct-tree from global_particles by recursive subdivision of the domain
    * the tree is flattened in the order indicated by partifun and then
    * partitioned/distributed to all processors
    */
   MT_Init(TAG_PARTICLE, global_particles, global_domain, partifun);
   /* compute center of mass of each cube in the oct-tree, starting from the leaves
    */
   MT_Up_Pass(centermass);
   /* calculate accelerations for each leaf's particle list. The list of particles or cells
    * considered is computed by searching the entire oct-tree (the "environment"), but
    * using critical_radius_test to prune out cells sufficiently distant to be
    * treated as a single large mass
    */
   MT_Map_Leaf_Env(critical_radius_test, calculate_accelerations, rootkey);
   /* update position & velocity of each particle by leapfrog integration based
    * on its calculated acceleration
    */
   MT_Map_Leaf(update_body);
   /* Collect the particles from the tree back into global_particles
    */
   MT_Gather(global_particles, global_domain);
}
```

Fig. 3. Implementing the Barnes-Hut algorithm for the N-body problem (simplified)

algorithm is implemented using M-Tree operators. The code shown is slightly simplified and we have omitted initialization details.

The algorithm has three main stages:

Partition: recursively decompose the particle region into eight subregions.
In Figure 3 this is implemented by `MT_Init`;

Mass: each internal node calculates its center of mass using information propagated upwards from its subtrees.
This is handled by the `MT_UP_Pass` operator. Its parameter, `centermass`, is the name of a function which computes the center of mass for each node of the tree by combining information about the mass distribution of the subtrees;

Potential/Force Calculation: For each particle, traverse the tree searching for particles which must be considered. Prune the traversal by approximating a subtree as a single mass if it lies sufficiently far away. This is handled by `MT_Map_Leaf_Env`. The "environment" is the root of the octree itself. The "filter" function `critical_radius_test` returns the list of particles and approximated subtrees which need to be considered for a given particle `MT_Map_Leaf_Env` maps `calculate_accelerations` to each particle with a filtered portion of global tree, to compute the resultant forces.

Update: `MT_Map_Leaf` is used to apply `update_body` to every particle, to update its position according to the forces acting upon it.

Finally, the results are collected and, in the next iteration, a new M-Tree is constructed reflecting the updated positions.

4 Implementation of the M-Tree for particle applications

For portability, the M-tree is implemented as a C library based on MPI[9]. In the current prototype implementation there are separate implementations for mesh-based problems and particle-based problems, although it should be stressed that a common implementation is quite possible by enriching the higher-order operators of the ADT with additional parameters. For simplicity we describe the implementation as it currently stands.

In the mesh-based problems, the most commonly-used computation patterns are mainly among blocks on the same level or on neighbouring levels. The particle-based problems tend to involve computation and communication among treenodes located on all levels. For example in Barnes Hut algorithm, each leaf node needs to compute with a subset of whole tree as its interaction list. In order to achieve efficient access to randomized treenodes, a hashing scheme is used which is discussed in the following section. In what follows we focus on the details of the particle-based M-tree implementation and its performance.

4.1 Implementation

Particle-based solvers are not mesh-based in the sense that a number of discrete particles form a domain according to their positions and they cannot be modelled by variables on gridded points in the domain as in a continuum system. Most of the solvers use the hierarchical structure of the domains to improve the computational complexity. The M-Tree offers the control over such hierarchically structured particle space. In applications such as the N-body problem, each body is influenced by all the other bodies. Thus to calculate the influence on each body, the interactions between the body and all the other bodies must be considered. The computational pattern in this kind of application tends to access the global data structure from each particle, pruning where possible. This leads to an unpredictable access pattern and dynamic load imbalance which are the main issues for parallelization.

The particle simulation libraries are implemented using a hashing scheme similar to the technique proposed by Warren and Salmon [14]. In this scheme each treenode has a unique key defined according to its hierarchical coordinates. Each key corresponds to some physical data inside the domain of each treenode. A hash table is used to map the key to the memory location holding this data. This key space is convenient for tree traversals where a node is accessed directly, without going via its parent; this is needed in some fast N-body algorithms such as the Fast Multipole Method.

The M-tree is flattened and partitioned by a user-defined ordering function. In our test case a Peano-Hilbert ordering [17] is used to achieve maximum locality and load balancing. In this partitioning scheme, each process initially stores those particles for which it is responsible which tend to have maximum intersection of their interaction sets. These local particle sets are a subset of the flattened M-tree and thus stored spatially in Peano-Hilbert order. A software caching scheme is used to minimize the extra message passing for the shared overlapping interaction sets. Whenever a non-local tree node is read, it is also inserted into into the local hash table as subsequent references can be satisfied locally. The variable cacheline size of each remote access can be used to further optimize the overall performance as described below.

4.2 Optimizations

Various optimization approaches have been used in the implementation of the M-tree operators for particle simulation. For example, the **MT_Map_Leaf_Env** operator is implemented by traversing the global M-tree in depth-first order since the partition and load balancing scheme maintains maximum locality. Message passing occurs whenever a non-local tree node is encountered. When replying to request messages from other processors, instead of sending just the requested treenode, a contiguous package of treenodes (a cache line) can be sent. This scheme is based on a cache coherence protocol which exploits the controlled update discipline of the data structure to make efficient use of high-bandwidth high-latency message passing platforms [2]. The optimization is based on the statistical likelihood that further treenodes stored adjacently will be accessed later either by the current particle or by other particles on the same processor. The scheme reduces the number of separate messages at the risk of wasting bandwidth on data which is not eventually used. In Figure 4, a 8192-body test shows that using a cache line size of 50 treenodes almost doubles the speedup obtained using a cache line size of 5.

4.3 Performance

Our implementation is currently only in prototype form, but some initial performance results are available for the Barnes-Hut algorithm. Our experimental work has used the 128-processor Fujitsu AP1000 at Imperial College. Figure 5 shows overall time and speedup for different problem sizes up to 16384 particles. The optimal performance for smaller problems are tested using smaller cacheline size because of the small number of local treenodes when scaling up to 128 processors. It shows that performance continues to improve with large numbers of

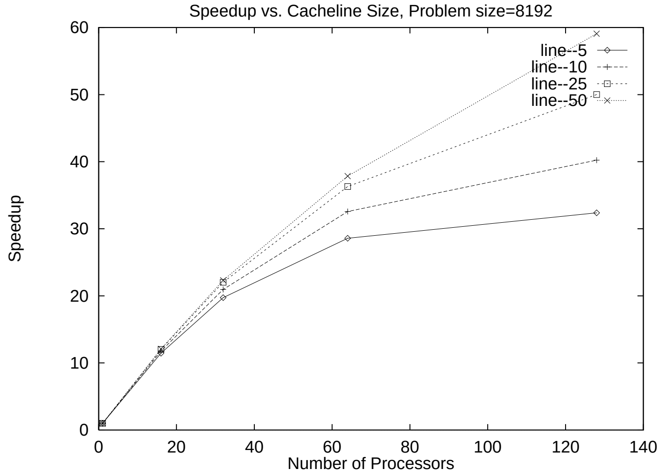

Fig. 4. Speedup with Different Cacheline Sizes

processors, but that there is potential for improvement. Figure 4 shows speedup obtained at different cache line sizes. The performance is substantially improved by using large cache lines, although comparison with a hand-tuned low-level implementation [3] shows that there is considerable room for improvement.

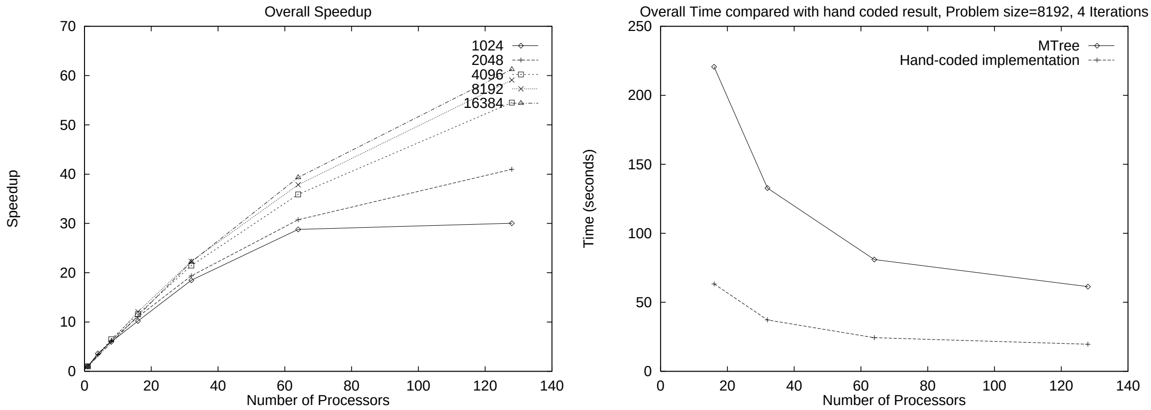

Fig. 5. Overall Speedup (left graph) and Overall Time (right graph)

5 Related work

This project is a further development of our earlier work on skeletons and algebraic program transformation [12, 8]. The skeletons approach aims at building up a general framework for structured parallel programming based on the idea of abstracting commonly-used computation patterns as higher-order functions. This work concerns applications where there is rich domain knowledge and hence plenty of scope for abstraction and optimization. Here we have focussed on the underlying distributed data structure, whose dynamic nature involves important and complex factors which have not properly been addressed in theoretical work up to now. Recent developments like Südholt's Data Distribution Algebras [18] indicate some of the potential. There are several comparable attempts to build general-purpose support for adaptive computations [5, 10, 4].

Using data abstraction to capture commonly-occurring computation forms with a class of application have been well studied, in particular for irregular and dynamic applications. Such systems includes LPARX [13] and DAGH [15]. LPARX provides parallel abstraction for dynamic arrays. The mechanism hides low-level implementation details and provides tools for data distribution, partitioning and mapping, parallel execution and interprocessor communication. The dynamic array supported by LPARX forms one level of data partition and the user has to maintain explicit control over multi-level hierarchy. The idea of collective communication has been widely adopted to enhance the compositionality of concurrent processes. Co-ordination systems such as Archetype [7] and PCN [11] and parallel languages such as CC++ [6] all provide abstractions of collective communication.

6 Conclusions and further work

We have presented the design for a generic package which captures a class of irregular and adaptive algorithms. The user of the package can avoid much of the difficulty of programming such applications, and rely on a well-tested and carefully optimized implementation of the key data structure. The library provides reusability and eases programming real applications. Since the library is generic, there are potential overheads compared with a hand coded particle simulation code [3], which we are currently quantifying.

Directions for further work include:

- Further optimization, including the use of a more efficient hash function, and finding the optimal cache line size.
- Production of a generic M-Tree implementation capturing the domain-dependent behaviour of different adaptive applications by enriching the higher-order ADT operators with additional parameters
- Further application studies, in particular multigrid solvers and time-varying adaptive-mesh problems such as fluid flows.
- Developing a theory for algebraic transformation of M-Tree programs, for example to capture fusion of tree traversals.

Acknowledgements: This work was supported by the UK Engineering and Physical Sciences Research Council under grant number GR/J 87015. We would also like to extend our thanks to the Imperial/Fujitsu Parallel Computing Research Centre for their continued support.

References

1. J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(4), December 1986.
2. Andrew J. Bennett and Paul H. J. Kelly. Efficient shared-memory support for parallel graph reduction. *Future Generation Computer Systems*, 1997. To appear.
3. Simon Boothroyd. Galaxy simulation on the AP1000, 1996. MEng Dissertation, Department of Computing, Imperial College.
4. G. H. Botorog and H. Kuchen. Algorithmic skeletons for adaptive multigrid algorithms. In *Proceedings of IRREGULAR'95, LNCS 980*. Springer-Verlag, 1995.
5. G.F. Carey, M.Sharma, and K.C.Wang. A class of data structures for 2-d and 3-d adaptive mesh refinement. *International Journal for numerical methods in Engineering*, 26, 1988.
6. K.M. Chandy and C. Kesselman. Compositional C++: Compositional parallel programming. Technical report, California Institute of Technology, 1992. Technical Report Caltech CS-TR-92-13.
7. K.M. Chandy, R. Manohar, B.L. Massingill, and D.I. Meiron. Integrating task and data parallelism with the collective communication archetype. Technical report, California Institute of Technology, 1994. Technical Report Caltech CS-TR-94-08.
8. J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel programming using skeleton functions. In *Parallel Architectures And Languages, Europe: PARLE 93*. Springer-Verlag, 1993.
9. Department of Computer Science, Computer Sciences Laboratory, The Australian National University. *MPI: User's Guide*, 1994.
10. D.J. Edelsohn. Hierarchical tree-structures as adaptive meshes. Technical Report SCCS-193, Syracuse Center for Computational Science, NY, 1991.
11. Ian Foster, Robert Olson, and Steven Tuecke. Productive parallel programming: The PCN approach. *Scientific Programming*, 1(1), 1992.
12. Paul H.J. Kelly. *Functional Programming for Loosely-coupled Multiprocessors*. Pitman/MIT Press, 1989.
13. S. R. Kohn. *A Parallel Software Infrastructure for Dynamic Block-Irregular Scientific Calculations*. PhD thesis, Dept. of Computer Science and Engineering, Univ. of California, San Diego, 1995.
14. J. K. Salmon M. S. Warren. A parallel hashed oct-tree n-body algorithm. In *Proceedings of SuperComputing 93*, 1993.
15. M. Parashar and J. C. Browne. Dagh: A data management infrastructure for parallel adaptive mesh refinement techniques. Technical report, Dept. of Computer Science, Univ. of Texas at Austin, 1995. Premiminary Users Guide.
16. W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, second edition, 1992.
17. H. Samet. *The design and analysis of spatial data structures*. MIT Press, 1990.
18. Mario Südholt. Data distribution algebras — a formal basis for programming using skeletons. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, pages 19–38. North-Holland, June 1994.