Language Support for Loosely Synchronous Data Parallelism

Pankaj Kumar[*] Nancy McCracken

pkumar@nova.npac.syr.edu
Dept. of Computer Science and Northeast Parallel Architectures Center,
Syracuse University, Syracuse, NY 13210.

(Extended abstract submitted to ACM–SIGPLAN PPOPP 93)

Abstract

Exploiting data parallelism in programs that construct and manipulate pointer based data structures of arbitrary shape and varying size require support for expressing multiple data parallel operations on scattered data. This paper extends the 'C' language with a new data type called ***Troupe*** and associated ***for_each*** statement to express data parallelism in pointer based dynamic data structures and describes the implementation strategy for efficiently executing such programs on message passing parallel computers. A variety of memory allocators are also introduced that automatically distribute the data in a load balanced manner.

1. Introduction

A variety of scientific and engineering applications require complex data structures whose size and shape vary over time as the computation proceeds. In [1] Fox has classified these application problems as loosely synchronous. Examples of loosely synchronous applications are molecular dynamics [2], particle–in–cell methods [4], adaptive finite element methods [3], many–body simulation [8], multi target missile tracking [5] etc. Apart from irregularly connected data structures, characteristics of these applications are that the data elements may not be identical and while the computation is not synchronized at the microscopic level, it is synchronized at the macroscopic level.

A common technique for implementing loosely connected dynamic data structures of arbitrary shape is by the use of pointers and dynamic memory allocation as discussed in [6]. We assume that it is more appropriate to implement loosely synchronous applications using advanced programming languages like 'C' or C++ which provide support for constructing complex pointer based dynamic data structures.

In this paper, our main concern is writing parallel programs for loosely synchronous class of applications or in general those that manipulate pointer based data structures like trees, list, graphs etc., such that the programmer does not have to worry about synchronization and communication issues and that the programs can be efficiently compiled and executed on message passing MIMD computers.

Due to ease of program development and opportunity for exploiting maximum parallelism, use of data parallel approach of Hillis and Steele [9] is emphasized for implementing loosely synchronous applications. We have been further motivated to adopt data parallel approach, as it has been shown in [15] and [16] that data parallel programs can be efficiently compiled and executed on message passing MIMD computers. However, success has been largely limited to programs with regular data storage and access pattern.

* This work was supported by Center for Advanced Studies and Language Technology, IBM, Toronto, Canada.

Existing data parallel languages like Fortran90D [10], C* [17], Dataparallel C [13], PC++ [14] are not appropriate for implementing programs that manipulate pointer based dynamic data structures as they support data parallelism only on objects with fixed size and shape such as arrays. With regards to exploiting parallelism in pointer based dynamic data structures there are several issues. The first is how to express multiple data parallel operations on data elements which, unlike arrays, are scattered in memory. The second is how to traverse the data structure in parallel. Third, is how to perform several traversal in parallel. It is important to note that as each traversal proceeds it may create/destroy/move data objects. Capability to create data structures in parallel is greatly desired for many applications such as adaptive multigrid finite element methods where generating the mesh itself is time consuming.

In this paper we propose language extensions to 'C' for managing and exploiting data parallelism in pointer based dynamic data structures without having the programmer worry about issues such as data distribution, synchronization and communication.

There have been other efforts to provide programming language support for loosely synchronous applications most of which are based on the object oriented approach. PC++ [14] based on the *Distributed Collection* model defines a new class called **collection** for supporting fine grained data parallelism. Because elements in a collection are treated as linear array it lacks support for performing data parallel operations on objects interconnected in an arbitrary fashion. Another, approach as suggested by Chen [4] in their implementation of N–Body problem is to hide the implementation details by defining library classes for commonly used data structures such as tree, lists etc. The approach presented in this paper differs significantly from the others as it is based on the data parallel approach of CM–Fortran or Fortran90D.

2. Nature of Dynamic Data Parallelism

With regards to exploiting parallelism in pointer based dynamic data structures there are three the main issues. The first issue is how to express data parallel operations on data elements which, unlike arrays, are scattered in memory. The second is how to traverse the data structure in parallel. Third is how to allow many objects to traverse the data structure simultaneously without worrying about synchronization. Pointer based dynamic data structures typically requires support for the following types of data parallel operations.

– ***for_each***(node of the data structure)
– ***for_each(*** child of a node)
– ***for_each***(parent of a node)
– ***for_each***(leaf of a data structure)

and many others depending upon how the data is organized and the nature of the application.

To get greater insight into the nature of data parallelism in programs that manipulate dynamic data structures, we consider two toy programs which are representative of the computational structure of the hierarchical N–Body [7] methods described in [25]. Because the principal data structure for N–Body methods is a tree hence we consider programs that construct a binary tree and then perform certain computation on each node of the tree.

2.1 Example 1.

The purpose of this example is to show how dynamic data structures can be constructed using the data parallel approach. The following data parallel program recursively builds a binary tree for a set of elements.

```
Main
1.          root = Empty
```

```
2.              for_each (element in list)
3.                  Insert_In_Tree (element, root)
                endfor
endmain

Function        Insert_In_Tree (element, node)
1.                  if  (node is Empty)
2.                      Insert element in this node
3.                      return;
                    }
4.                  if (node is leaf)  && ( node–>value <= element–>value)
5.                      Create the left child.
6.                      Insert element in left child
7.                      return
                    endif
8.                  if (node is leaf)  && ( node–>value > element–>value)
9.                      Create the right child.
10.                     Insert element in right child
11.                     return
                    endif
12.                 if (node is not a leaf )
13.                     for_each (child of this node)
14.                         Insert_In_Tree (element, node–>child)
                        endfor
                    endif
endfunc
```

The above program is data parallel is we assume that **for_each** statement creates a logical thread of execution (process) for every data object on which to the operation is to be performed. Due to **for_each** statement in **main** program, insertion of each element in the tree is an independent process, each of which proceeds in parallel. Due to **for_each** statement at line 13 in procedure **Insert_In_Tree** allows each traversal itself to proceed in parallel. The program is free of synchronization issues if we assume that only one thread at a time is allowed to operate on a node of the tree. At any instant it does not matter which thread operates on a node as operations on nodes are independent of each other.

2.2. Example 2.

This example is representative of the algorithm to compute multipole expansion of each node for the N–body simulations based on Barnes–Hut tree. To compute the multipole expansion for a node, first compute the multipoles for each of its children. Once the multipoles for each of its children have been computed the multipole for the node is either a function of the multipole value of the child or is a function of the number of the number of leaves contained in the subtree rooted at the child. The multipoles for a leaf is a function of the element value stored in that leaf. The mathematical details for computing these interactions are given in [25].

A non–recursive data parallel algorithm for computing the multipoles proceeds by traversing the tree bottom up by first computing the multipole forces at the leaf nodes and then at the parent, grandparent and so on. Assuming that the height of the tree is N with root being at height 0 the computation proceeds as shown below.

```
Compute_multipoles (root)
1.  for (h=N;  h >= 0;  h = h – 1)
2.      for_each ( node in tree) at height (h)
3.          if  (node is leaf) then node–>multipole = func(node–>value)
4.          if  (node is not leaf) then
5.                  for_each (child of this node)
6.                          if (node–>child–>multipole < Ncutoff) then
7.                              for_each (leaf of this child)    /* in subtree rooted at node */
8.                                  node–>multipole  = node–>multipole + func(leaf–>value)
9.                              endfor
10.                         else  ParAxis (node, child)
11.                 endfor
12.         endif
13.     endfor
14. endfor
```

In the above procedure, **ParAxis** is a function which implements the mathematical equations involved in computing the multipole expansion. Procedures for computing the forces on the particles and reorganizing the tree can be similarly described using the for_each notation. Two aspects of the above program are worth mentioning which brings out the need for dynamically controlling the data parallelism from within the program. Firstly, at line 2, it is required by the **for_each** statement to know what nodes are there at a particular level in the tree. Secondly, at line 7, it is required by the **for_each** statement to know what are leaves in the sub tree rooted at a particular node.

3. Language Support for Dynamic Data Parallelism

To provide support for the above mentioned data parallel operations it is required that address of each object such as **each node**, **each child,** e**ach parent**, **each leaf** be computable or known at run time by the **for_each** statement. Because in the case of dynamic data structures individual data elements are created as and when needed they are usually scattered in memory as a result their address is not computable based on some known storage scheme. To overcome this problem, our approach is to explicitly pass addresses of data objects as arguments to the **for_each** statement. This is achieved by introducing a data type called **troupe** members of which are pointers to data objects on which the operations are to be performed. As shown in example programs of previous section, each member may traverse the data structure and may perform several data parallel operations during its course of traversal.

3.1 *Troupe*: A Data Type For Dynamic Data Parallelism

A **troupe** is a data type for dynamically maintaining data parallelism associated with different parts of the data structure. It is a collection of pointers to objects of a particular type.When passed as an argument to **for_each** statement (explained in the next section) operations are performed in parallel by each member of the troupe. Members can be added or removed from the **troupe** during program execution by the use of operators **add** or **delete**, and two or more troupes can be merged using the operator **merge**.

Troupe variables are declared by preceding an identifier with **!** symbol. For example, the declaration

```
    struct  treenode  * ! nodes;
```

will result in the creation of a **troupe** variable ***nodes*** capable of storing pointers of type ***treenode***. When used as part of records in Pascal or struct in 'C', troupe can be used to dynamically control data parallelism associated with each data object. For example in the following definition of *treenode*,

```
        struct treenode {
            .......
            struct treenode * ! children;
        };
```

each object of type *treenode* consists of a troupe called ***children*** capable of storing pointers of type *tree node*. Hence, troupe ***children*** can be used to perform operations in parallel on each of the children of a particular node. Recursively, a node may perform operations in parallel on the children of its children.

3.2 *for_each* Statement

A convenient mechanism for expressing data parallelism is the use of *forall* statement as introduced in CM–Fortran. However, unlike the forall construct in Fortran which operates on arrays, we define **for_each** operating on each member of troupe. The **for_each** statement takes as arguments a variable and a troupe of the same type and creates a logical thread corresponding to each member of the troupe. Each member of the troupe is owned and known to one and only one thread. In each thread, the identifier is initialized to the member of the troupe responsible for creating the thread.

```
        for_each (variable::troupe)
        (condition)
        {
            statements;
        }
```

The threads are created only for those members where the condition specified in the for_each statement is true. Variables declared within the **for_each** statement are local to each thread. However, variables declared outside of the **for_each** statement and visible to it are also visible and accessible by each thread. Although, a thread may modify the variables declared outside of **for_each** statement, the modifications are visible only locally. As a general rule, any variable not declared within the **for_each** statement is inherited from the parent.

3.4 Synchronization

The problem of synchronization is overcome by not allowing multiple threads to execute simultaneously on the same data element (notion of distinctness as discussed by Steele in [...]). This is achieved in part by not allowing **troupes** to contain duplicate members. This ensures that each operation invoked by the **for_each** statement will be on a distinct data element. However, due to nested **for_each** statements it is possible that a child thread may be created on a same data object for which a parent thread may already exist. This would lead either to deadlock or inconsistent result. Hence, a child thread on a data object is not permitted if there exists a parent thread on the same data object.

Secondly, by the use of pointers it is possible for more than one thread to operate on the same data element. For example,

```
for_each (p::troupe)
        p–>next–>field = ......
```

In the above program segment it is possible for more than one thread to have the same value in the *next* field. Hence, all of them operate on the same data *p–>next–>field*. A solution to this problem is to first lock the data item in a read or write mode and then apply the operation.

3.3 Parallel Object Creation and Distribution

In our case data distribution is implicitly performed by the dynamic memory allocators. Thus every time a new object is created the corresponding memory allocator decides on which processor to create the object. By having the notion of long pointers which unlike 32 bit numbers are 48 bit long allows us to treat pointers as being unique across all of the processors. These implementation issues will be discussed in detail in the final paper.

3.3.1 *p_malloc*: Parallel Memory Allocator

P_malloc is used for creating a set of objects in parallel. It takes as argument a set of pointers *p1, p2, ..., pn* as indicators for locality of reference and an integer *size* indicating the size of the object to be created.

```
p_malloc (p1, p2, ... pn, size)
char  *p1, *p2, ....,  *pn;
int     size;
```

Any or all of p's can be NULL. If all of the pointers are NULL then the system will create the object without any regards for locality of reference. As an example consider the following code.

```
for_each (q::troupe)
        q–>child = p_malloc (q, size);
```

In the above program segment, in each thread **p_malloc** will return a pointer to new object of size bytes, if possible the new object is created on the same processor and page as that of address **q**.

3.3.2 *s_malloc:* Single Memory Allocator.

Unlike p_malloc which creates a new object in each of the thread, s_malloc creates a single object whose address is returned to each thread. S_malloc accepts the same arguments as p_malloc. For example, in the following code

```
for_each (q::troupe)
        q–>child = s_malloc (q, size);
```

only one instance of object is created whose address is assigned to *q–>child* in each thread. In the case of **s_malloc** the object is created such that it provides good locality of reference to each of the object pointed by q.

Several other types of memory allocators can be defined for different types of applications.

4. Data Parallel Program Using Troupe.

As an example to see how **troupe** and **for_each** can be used, we consider problems discussed earlier in section 2. Structure of each node of the tree as defined in 'C' is as follows.

```
struct BHnode {
        float    value;                /* some force value etc. */
        float  multipole;             /* multipole expansion for this node */
        int      alldone;            /*  used for keeping count of how many children are done */
        BHnode    *parentn;      /* pointer to parent */
        BHnode    *!child;         /* troupe for containing child pointers */
        BHnode    *!leafs;         /* troupe for storing pointers to leafs in this subtree */
        BODY      *body;           /* pointer to the body */
};
```

In the definition of the **BHnode**, we have defined two troupes. The troupe called **child** is used for maintaining pointers to children. Using a ***troupe*** instead of an array of pointers would allow a new thread to be created for traversing the tree along each child. The troupe **leafs** is used for maintaining pointers to the leaf nodes in the sub tree rooted at the particular node. Access to leaf nodes in the sub tree rooted at a particular node is required for computing the multipoles expansions. As will be shown later in the program, as the traversal of the tree proceeds pointers to leafs are passed to the nodes requiring them.

4.1 Data Parallel Program for Example 1.

In the main program described below it is assumed that the set of elements to be inserted in the tree are already contained in the troupe **bodies.** Since, the procedure for putting the bodies in the troupe involves file handling and is rather uninteresting, it has not been described here. Statements 1 and 2 construct the initial tree in parallel. Parallelism is achieved by viewing insertion of each body in the tree as an independent process.

The procedure Insert_In_Tree is responsible for inserting the body pointed by argument **newbody** into the tree pointed by argument **root**. The procedure is also responsible for mainting the troupe **allleafs** which at any point in the computation contains pointers to all the leaf nodes of the tree. Since, the troupe **allleafs** is passed as an argument (i.e. declared outside the procedure) hence, it is shared by all threads (corresponding to each invocation of **Insert_In_Tree**). Since, operations on troupes are associative in nature hence, different threads can **add** or **delete** items from the troupe without requiring any synchronization.

```
        BHnode *!allleafs;
        BHnode  *root;

1.      for_each (bdy::bodies)  {
2.              Insert_In_Tree (bdy, allleafs, root);
```

```
/* Non–Recursive procedure to create Barnes–Hut tree.*/
Insert_In_Tree (newbdy, allleafs, root)
BODY    *body;
BHnode  *!allleafs;      /* troupes */
BHnode  *root;
{
    BHnode  *q, *p, *nodes;
     int      i_am_leaf,  done;

     add (nodes, root);
     done = FALSE;
      while (NOT done )    {
         for_each (p::nodes) {
                 if  (p == NULL) {
                         Insert element in this node;
                         return;
                 }
                 if  (i_am_a_leaf(p) == TRUE) && (p–>value <= newbdy–>value) {
                      q = p_malloc (p, sizeof(BHnode));
                       add (p–>child, q);    /* insert q in troupe  contained in the parent node pointed by p */
                      Insert element in node pointed by q;
                       done = TRUE;
                 }
                 if  (i_am_a_leaf(p) == TRUE) && (p–>value > newbdy–>value) {
                      q = p_malloc (p, sizeof(BHnode));
                      add (p–>child, q);    /* insert q in troupe  contained in the parent node pointed by p */
                      Insert element in node pointed by q;
                       done = TRUE;
                 }
                 if  (i_am_a_leaf(p) == FALSE)  {
                         /* continue search in 4 sub trees */
                         merge (nodes, p–>child);
                 }
         }
     }
```

The procedure starts by adding **root** to an empty troupe **nodes**, and then keeps creating threads (statement 4) until the body pointed by **newbody** has been inserted in the tree. An important aspect of this procedure is the use of the troupes as a field in the struct **BHnode** to store pointers to the child nodes. By using troupe instead of an array of pointers (one for each child) one can create a new thread for traversing the subtree rooted at each of the child. This is done at statement 21, where **merge** takes the contents of troupe **p–>child** and concatenates to the troupe nodes, hence resulting in new threads being created at the statement 4.

4.2 Data Parallel Program for Example 2.

In this procedure, we pass troupe allleafs containing pointers to all the leaf nodes in the tree. This was created by the previous procedure while constructing the tree.

```
        Generate_multipoles (allleafs)
        BHnode  *!allleafs;
        {
                BHnode  *!newnodes, *p, *q, *r;

1.              merge (newnodes, allleafs);
2.              done = FALSE;
3.              while (NOT done) {
4.                for_each (p:newnodes) {
5.                      if (p is a leaf)  {
6.                              p–>multipole = func(p–>value);
7.                              add (p–>parent–>leafs, p);
8.                      }
9.                      if ((p is not a leaf) && (p–>alldone == 2)  {
                                /* i.e. multipoles for each of the child has been computed */
10.                             for_each (q::p–>child)  {
11.                               if ( q–>multipole < Ncutoff)
12.                                 for_each (r::q–>leafs) {
                                          p–>multipole =  p–>multipole + func(r–>value);
13.                               else
14.                                   ParAxis (p,q);
15.                               /* pass the troupe containing leaf nodes to the parent */
16.                               merge (p–>parent–>leafs, q–>leafs);
17.                               add (newnodes, p–>parent);
18.                                 p–>alldone += 1;
                                }
                        }
                }
        }
```

5. Basic Program Execution Strategy.

Our execution methodology is similar to that adopted for Fortran 90D as discussed in [26]. That is, the same program executes on each processor with a single logical thread of control. Parallelism is achieved by performing computations on processor which owns the data element. In our case, this implies that when creating threads for each element in a troupe, the pointer in the troupe is first dereferenced to determine the processor on which the object to which it points resides. Data elements required by the thread but not available on the local processor are fetched by means of communication. Detailed discussion about the program translation and execution will be discussed in the final version of the paper.

6. Current Status

An implementation of the proposed constructs and the corresponding dynamic data distribution and load balancing schemes is currently in progress for Ncube2 and CM5. A loosely synchronous application such as molecular dynamics or N–Body simulations and multiple target missile tracking program shall also be implemented using the proposed constructs and their performance measured on CM–5 or Ncube2 distributed memory parallel computers.

7. Acknowledgements.

We thank Professor Geoffrey Fox for the many helpful suggestions**.**

8. References.

[1] G.C. Fox. The Architecture of Problems and Portable Parallel Software Systems. SCCS–134, NPAC, Syracuse University.

[2] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan. CHARMM: A Program for Macromolecular Energy Minization and Dynamics Calculations. Journal of Computational Chemistry, Vol. 4, No. 2, 187–217, 1983.

[3] Maria–Cecilia Rivara. Design and Data Structure of Fully Adaptive Multigrid, Finite–Element Software, ACM Trans. on Math. Software, Vol. 10, No.3, Sept. 1984, pp 242–264.

[4] D.W. Walker. Characterizing the Parallel Performance of a Large–Scale Particle–In–Cell Plasma Simulation Code. Concurrency: Practice & Experience, Vol. 2(4), 257–288, Dec. 1990.

[5] T.D. Gottschalk, ”A New Multi–Target Tracking Model”, Caltech Report C^{3}P–480, 1987.

[6] N. Wirth. Algorithms + Data Structures = Programs. Prentice Hall.

[7] J. Barnes and P. Hut. A Hierarchical O(NlogN) Force–Calculation Algorithm. Nature, 324, 446–449, 1986.

[8] A.W. Appel. An efficient program for many–body simulation. SIAM J. Sci. Stat. Comput., Vol 6, 1985.

[9] W. Hillis and G. Steele Jr. Data Parallel Algorithms. CACM 29(12), 1170–1183, 1986.

[10] G.Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Dept. of Computer Science, Rice COMP TR90–141, Rice University, Dec. 1990.

[11] M.S. Lam, M.C. Rinard. Coarse–Grain Parallel Programming in Jade. Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP). 1991.

[12] A.S. Grimshaw, J.W.S. Liu. MENTAT: An Object–Oriented Macro Data Flow System. Proceedingd OOPSLA, 1987.

[13] P.J, Hatcher, M.J. Quinn, A.J. Lapadula, R.J. Anderson, and R.R. Jones. Dataparallel C: A SIMD Programming Language for Multicomputers. The 6th Distributed Memory Computing Conference Proceedings, 1990.

[14] J.K. Lee and D. Gannon, Object Oriented Parallel Programming Experiments and Results. In Supercomputing

91, 273–282, 1991.

[15] M.J. Quinn, P.J. Hatcher, and K.C. Jourdenais. Compiling C* programs for a hypercube multicomputer. In Proc. of ACM/SIGPLAN PPEALS 1988, pp57–65, 1988.

[16] P. Mehrotra and J. Van Rosendale. Programming Distributed Memory Architectures Using Kali. ICASE Tech. Report No. 90–69. 1990.

[17] Thinking Machines. C* Reference Manual.

[18] Thinking Machines. Fortran Reference Manual.

[19] S. Bhatt, M. Chen, C.Y. Lin, P. Liu. Abstractions for Parallel N–body Simulations. Proc. Scalable High Performance Computing Conf., 1992.

[20] A. Chien, W. Dally. Experience with Concurrent Aggregates (CA): Implementation and Programming. The 5th Distributed Memory Computing Conference Proceedings, 1990.

[21] T.W. Clark, R.v. Hanxleden, K. Kennedy, C. Koelbel, L.R. Scott. Evaluating Parallel Languages for Molecular Dynamics Computations. Proc. Scalable High Performance Computing Conf., 1992.

[22] G.L. Steele Jr., E. Albert, J.D. Lukas. Data Parallel Computers and the FORALL Statement.

[23] K.P. Belkhale, P. Banerjee. Recursive Partitions on Multiprocessors. The 5th Distributed Memory Computing Conference Proceedings, 1990.

[24] R.P. Weaver, R.B. Schnabel. Automatic Mapping and Load Balancing of Pointer–Based Dynamic Data Structures on Distributed Memory Machines. Proc. Scalable High Performance Computing Conf., 1992.

[25] J.K. Salmon. Parallel Hierarchical N–Body Methods. C3P Physics 206–49, Caltech.

[26] Alok Choudhary, G.C. Fox, S.Ranka, S. Hiranandani, K. Kennedy, C. Koelbel, Chau–Wen Tseng. Compiling Fortran 77D and 90D for MIMD Distributed Memory Machines. To Appear in Frontiers, 92.

[27] Guy L. Steele Jr., Making Asynchronous Parallelism Safe for the World, 18th ACM Proceedings on Principles of Programming Languages, 1990, 218–226.