SPMD Execution in the Presence of Dynamic Data Structures

Rajiv Gupta

Department of Computer Science, University of Arizona, Tucson, AZ 85721

1. Introduction

A wide range of applications make use of regular dynamic data structures. Dynamic data structures are typically required because either the size of the data structure cannot be determined at compile-time or the form of the data structure depends upon the data that it contains which is not known until runtime. Some commonly used regular data structures include link lists, trees, and two dimensional meshes. Lists and trees are used by applications such as the N-body problem [10] and sparse Cholesky factorization [8, 11], dynamic meshes are used for solving partial differential equations and quad-trees are used by applications such as solid modeling, geographic information systems, and robotics [9].

Recently researchers have focussed on the translation of programs written using the shared-memory paradigm for parallel SPMD (single-program, multiple-data) execution on distributed-memory machines [1, 6]. In this approach the distribution of shared data arrays among the processors is specified as a mapping between array indices and processor ids. This mapping is either specified by the user or generated automatically by the compiler. The compiler translates the program into an SPMD program using the owner-computes rule in which parallelism is exploited by having all processors simultaneously operate on portions of shared data structures that reside in their respective local memories. The mapping between array indices and processor ids is also used by the compiler to generate interprocessor communication necessary for SPMD execution.

While the above approach is applicable to programs that use shared data arrays, it cannot be directly applied to programs with shared pointer-based dynamic data structures. This is due to the manner in which dynamic data structures are constructed and accessed. Unlike shared data arrays that are created and distributed among the processors at compile-time, dynamic data structures must be created and distributed among the processors at runtime. A mechanism for specifying the distribution of nodes in a dynamic data structure is also needed. Unlike arrays whose elements have indices, the nodes of a dynamic data structure have no unique names that could be used to specify a mapping of nodes in a dynamic data structure to processors.

On a shared-memory machine pointers through which shared data structures are accessed are implemented as memory addresses. Clearly this implementation of a pointer is not applicable to a distributed-memory machine. Even if a global pointer representation was available, an additional problem exists. Once a dynamic data structure is distributed among the processors, the pointers that link the nodes would also be scattered across the processors. Each processor must traverse these links to access the nodes in the data structure. Thus, the links must be broadcast to all processors creating excessive interprocessor communication. Thus an efficient mechanism for traversing and accessing a dynamic data structure is required.

In this chapter an approach for distributing and accessing dynamic data structures on distributed-memory machines is presented that addresses the problems described above. Language and compiler support for SPMD execution of programs with dynamic data structures is developed. Extensions to a C-like language are developed which allow the user to specify dynamic name generation and data distribution strategies. The dynamic name generation strategy assigns a name to each node added to a dynamic data structure at runtime. A dynamic distribution strategy is a mapping between names and processor ids that is used to distribute a dynamic data structure as it is created at runtime. The compilation strategy used in this approach allows the traversal of a data structure without generating interprocessor communication. The name assigned to a node is a function of the position of the node in the data structure. Each processor can generate the names of the nodes independently and traverse the data structure through name generation without requiring interprocessor communication.

The subsequent sections first present the language constructs for declaring regular distributed data structures and specifying dynamic name generation and distribution strategies. The semantics of pointer operations used to construct and manipulate local and distributed dynamic data structures is described. Compilation techniques for translating programs with dynamic data structures into SPMD programs are presented next. Extensions for supporting irregular dynamic data structures are developed. Some compile-time optimizations are also briefly discussed. Discussion of related work concludes this chapter.

2. Language Support for Regular Data Structures

In this section language constructs required for expressing processor structures, declaring local and distributed dynamic data structures, and specifying dynamic name generation and distribution strategies are presented. The semantics of pointer related operations used for constructing and manipulating dynamic data structures are also discussed.

2.1 Processor Structures

To express the processing requirements the user is provided with processor declaration statements shown below. In each of these declarations the user specifies the number of processors required. The selection of the declaration (TREE, ARRAY or MESH) determines the topology in which the processors are to be organized. Neighboring processors in the given topologies are considered to interact frequently. Thus, it is preferable that these processors are directly linked. The advantage of specifying topologies is that the subset of processors that are allocated to the program can be selected to match the communication patterns expected to occur frequently in the program. Good embeddings of the available topologies into the physical topology of the system, for example a hypercube, can be predetermined. Routing techniques for different topologies can be implemented once and used at runtime by all applications. In the declarations below, in case of a TREE the degree of each node and the total number of processors is specified by the user and in the cases of ARRAY and MESH structures the number of processors along each dimension are specified.

`processor` TREE (*degree*; *num*);
`processor` ARRAY (*num*);
`processor` MESH (num_1; num_2);

Each processor in a processor structure is also assigned a logical id denoted as Π which contains one or more fields (π_1, π_2 ... π_m). The number of fields chosen should be appropriate for uniquely identifying each processor in a given processor topology. The fields in the processor id have positive non-zero values. The processor id Π_0 (=(0,0,..)) will never be assigned to any processor and therefore it is used to indicate a null processor id. The mapping between the names (Ψ) of data items and processor ids (Π) will determine the manner in which the data structure is distributed among the processors. We assume that processor ids of the ARRAY and MESH topologies are represented by a single integer and a pair of integers respectively. The processor id of the TREE topology consists of two integers, the first identifies a level in the tree topology and the second identifies the specific processor at that level.

2.2 Dynamic Data Structures

Dynamic data structures are declared using pointer variables as in commonly used languages such as C. The user can declare a *base* type which describes the nodes used in building a data structure. From this base type the user can derive new types which represent simple *local* and *distributed* data structures. In addition, complex *hierarchical* data structures may also be specified. Each level in the hierarchy of such a data structure is a simple local or distributed data structure. By declaring variables of the above types the user can then write programs that construct and manipulate dynamic data structures.

The nodes of a distributed data structure are spread among the processors using a distribution scheme specified by the user. All nodes belonging to a local data structure either reside on a single processor or are replicated on all processors. If a local data structure is not a part of any distributed hierarchical data structure, then it is replicated on all processors. However, if the local data structure is a substructure contained within a distributed level of a hierarchical data structure, then all nodes in the local substructure reside on the single processor at which the parent node containing the local substructure resides.

In this work SPMD execution is based upon the owner-computes rule. Thus, a statement that operates on a non-replicated data structure is executed by a single processor. On the other hand statements that operate on replicated local data structures are executed by all processors. As a result any modification made to a replicated local data structure is made by each processor to its local copy. Therefore at any given point in the program the copies of a local data structure as seen by different processors are identical.

The type of a pointer variable is used by the compiler to determine how the data structure associated with the pointer variable is to be implemented. In addition to declaring pointer variables representing local and distributed dynamic data structures, the user may also declare temporary pointer variables of the base type to assist in the traversal of dynamic data structures.

There is one major restriction that is imposed on distributed data structures. A node may belong to a single distributed data structure at a given time. This restriction allows the processor at which the node resides to be uniquely determined. If a node belonging to a local or distributed data structure is to be added to another distributed data structure, the user must explicitly copy the node from the former to the latter. In the remainder of this section the type and variable declarations and the semantics of operations involving pointer variables is described in greater detail.

The type declaration of a pointer type is augmented with an attribute which specifies whether the data structure built using this type is to be distributed across the processors or it is simply a local data structure.

`type` NodeType `is` { `distributed` | `local` } BaseType;

Doubly linked data structures are viewed as being derived from singly linked structures created by associating with a unidirectional link, a link in the reverse direction. The creation of a link from one node to the next automatically creates the reverse link. In a language like C a doubly-linked data structure is implemented by introducing two distinct links that must be explicitly manipulated by the user. This is because C does not provide a language construct for expressing reverse links. Given a link ptr, the presence of a reverse link $revptr$ is expressed using the **reverse** construct shown below. The reverse pointers, such as $revptr$, can be only used for traversing a data structure. The creation of new nodes must be carried out using the original pointer ptr.

```
*pointertype ptr;
reverse (ptr,revptr);
```

A pointer variable in a program may be of a base type, a local data structure type, and a distributed data structure type. These variables are used to create new nodes, traverse existing data structures, access data stored in a data structure, and manipulate data structures through pointer assignments. The pointer variables are classified into three general categories: *handles*, *constructors*, and *temporaries*. The semantics of pointer related operations is based upon the category to which a pointer variable belongs.

A *handle* is a pointer variable which is of the type representing a local or distributed data structure. If a node is created using a handle it is assumed that this is the first node in the data structure. If the handle points to a local data structure, then a node is created at each processor so that a replicated structure can be built. However, if the local data structure is a substructure of a distributed data structure, then the node is created at the processor where the containing node of the distributed structure resides. If the handle points to a distributed structure, the distribution scheme will enable the identification of the processor at which the node is to be located. On the other hand if the current data structure is a substructure of another distributed data structure, then the processor at which the node resides will be inherited from the parent node. Since a handle is essentially a global variable that provides access to a distributed data structure, it is replicated on all processors.

The pointer variables that are fields of a node belonging to a local or distributed data structure are called *constructors*. These pointer variables are used to connect together the nodes belonging to the data structure. When an additional node is added to a data structure using a constructor, it is known that the data structure is being expanded. In the case of distributed data structures this information enables us to ascertain the processor at which the new node is to be located.

Temporary pointer variables assist in the traversal and construction of a pointer-based data structure. A temporary is declared to be of a base type. If a temporary points to a node in a local or distributed data structure it can be used to *traverse* the data structure. The same temporary can be used to traverse a local data structure in one part of the program and a distributed data structure in another part of the program. It can be used to create and initialize single nodes which can be added to a local or distributed data structure. After the node has been added to a data structure the temporary pointer variable can be used to traverse the data structure. The type of a temporary pointer varies from simply being a base type to a local or distributed data structure type derived from the base type. Since a temporary may be used by all processors to traverse a distributed data structure, it is replicated on all processors. A node created using a temporary can be added to the data structure.

The above discussion was primarily limited to a single data structure. Constructors and handles also play an important role in programs involving multiple data structures. In particular, they can be used to split or merge distributed data structures. For example, given a distributed link list, let us consider a constructor pointing to an element of the link list. If this constructor is assigned to another handle of a distributed link list type, then the original list is split at the position of the constructor and now forms a list corresponding to the handle to which the constructor is assigned. Thus, the above process results in splitting of a distributed link list into two distributed lists. The merging of distributed lists can be achieved by assigning the value of the handle corresponding to one list to the constructor within the last list element of the other list.

2.3 Name Generation and Distribution Strategies

Each element of a pointer-based data structure is assigned a name. A name denoted as Ψ will contain one or more fields (ψ_1, ψ_2 ... ψ_n). The manner in which these names are to be derived is specified by the user; hence the number of fields appropriate for describing the data structure is also chosen by the user. The name generation strategy must specify the name of the first node in the data structure. In addition, given a node n with name Ψ the computation of names of nodes directly connected to n are also specified. In other words the computation of the name of each node $n \rightarrow ptr_i$ denoted as $\Psi.ptr_i$ is also specified. If the link represented by ptr_i is a bidirectional one, then the computation of $\Psi.revptr_i$ from Ψ is also specified. The construct for specifying the naming strategy for a distributed data structure *DistType* is shown below.

```
name DistType {
      first: ( Ψ0 );
      Ψ.ptr1: f1(Ψ);
      Ψ.revptr1: f1^-1(Ψ);
      ......
      Ψ.ptrn: fn(Ψ);
      Ψ.revptrn: fn^-1(Ψ);
}
```

This mapping of a distributed data structure type on a given processor topology is specified by the user. The construct for specifying the distribution strategy of a distributed type *DistType* on a processor topology *ProcStruc* is shown below.

```
distribute DistType on ProcStruc {
            Π = ( π1 = g1(Ψ), π2 = g2(Ψ), ....... πm = gm(Ψ) )
}
```

2.4 Examples

The constructs described in the preceding sections are illustrated through three examples: a tree, a two dimensional mesh, and a list of link lists (an example of a hierarchical data structure). These examples demonstrate that distribution strategies similar to cyclic and block-cyclic distributions used for static arrays can be expressed for dynamic data structures using the name generation and distribution constructs described in the preceding section.

First let us consider the specification of a distributed tree shown below. A general name generation and distribution strategy applicable to trees of an arbitrary degree n are presented.

```
type Node is distributed
         struct {
               int data;
               *Node ptr1, ptr2, ... ptrn
         };
var *Node atree;
```

The naming strategy that is presented next assigns a single integer as a name to each node in the tree data structure. The root of the tree is assigned the name one, the successive nodes at a given level are assigned consecutive numbers as their names, and lower numbers are assigned to lower levels in the tree. Fig. 2.1 shows the names of the nodes.

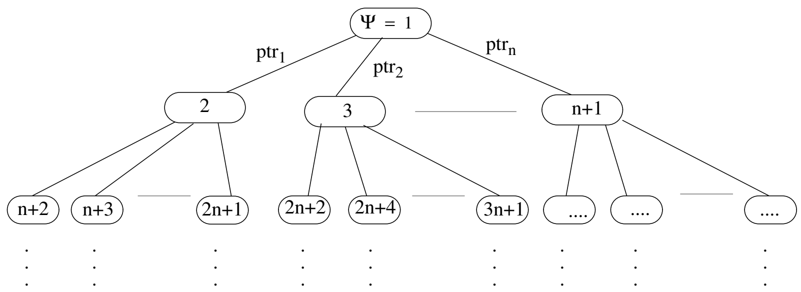

Fig. 2.1. Naming strategy for the distributed tree data structure.

The name (Ψ) assigned to a node is simply the position of the node in the distributed tree data structure. The name of any other node $P \rightarrow ptr_i$ is computed from the name of node P. Implementation of a doubly-linked tree structure requires an additional function for computing the position of node $P \rightarrow revptr_i$ from the position of node P. The function which computes the name for node $P \rightarrow revptr_i$ from the name of node P is the inverse of the function that computes the name of node $P \rightarrow ptr_i$ from the name of node P.

```
name Node {
      first: 0
      .....
      Ψ.ptr_i: n (Ψ − 1) + i + 1
      Ψ.revptr_i: (Ψ−i−1)/(n+1)
      .....
}
```

In order to describe the distribution of a tree on a processor structure, both the processor structure and the dynamic data structure are viewed as consisting of series of levels. Successive levels of the data structure are mapped to successive levels of the processor structure. The root nodes of the data structure are created on the first level of the processor structure. If the number of levels in the data structure is greater than the number of levels in the processor structure, multiple levels of the data structure will be mapped to a single level of the processor structure. The children of a node assigned to a processor at the last level in the processor structure are assigned to processors at the first level of the processor structure. The nodes in the data structure at a given level are distributed equally among the processors in the corresponding level in the processor structure. In this scheme a processor id is represented as $\Pi = (p, l)$ and it refers to the p^{th} processor at level l. The function *ProcId* given below returns the processor at which a node with position Π should be located.

```
distribute Node on ProcStruc { Π = ProcId(Ψ) }

function ProcId( Ψ ) {
/* function returns processor id Π to which node Ψ is mapped */
/* NumLevels(ProcStruc) - number of levels in ProcStruc */
/* NumProcs(l) - number of processors at level l of ProcStruc */
   currentΨ = currentlevel = 0;
   while (true) {
         currentlevel++;
         l = (currentlevel - 1) mod NumLevels(ProcStruc) + 1;
         for j = 1..n^(currentlevel−1) {
               p = (j-1) mod NumProcs(l) + 1;
               currentΨ++;
               if (Ψ == currentΨ) { return ( p,l ) }
}}}
```

A desirable property of the above strategy is that it tends to divide a data structure into pieces which are identical to the processor structure. For example, if a binary tree data structure is mapped to a fixed size binary tree of processors, this strategy divides up the data structure into binary trees of the size of the processor structure. This is illustrated in Fig. 2.2 by the distribution of a binary tree data structure on a binary tree processor structure with seven processors.

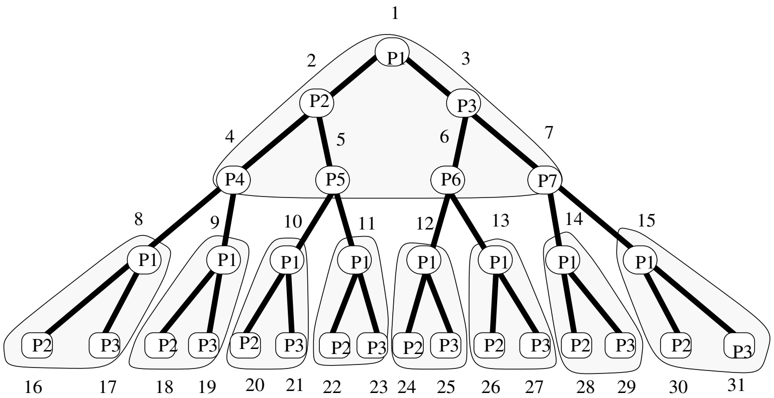

Fig. 2.2. Distributing a binary tree structure on processor TREE (2, 7).

Consider the specification of the mapping of a data structure representing a two dimensional mesh to a $n \times n$ processor mesh. Simple mapping functions which assign a unique name to each node and distribute the data structure among the processors are given below. Both the names and processor ids are specified as integer pairs. The name of a node is the same irrespective of the parent node through which it is computed and hence it is unique.

constant n = 2;
processor MESH (1:n, 1:n);
type
    Node **is struct** { **int** data; **Node* left,right };
    DistMesh **is distributed** *Node*;
    name *DistMesh* {
        first: $(\psi_1 = 0,\ \psi_2 = 0)$;
        $\Psi.left$: $(\psi_1 + 1,\ \psi_2)$;
        $\Psi.right$: $(\psi_1,\ \psi_2 + 1)$;
    }
    distribute *DistMesh* **on** MESH {
        $\Pi = (\ \pi_1 = \psi_1 \bmod n,\ \pi_2 = \psi_2 \bmod n\ )$
    }
var **DistMesh* amesh;

Fig. 2.3 illustrates the division of a mesh among the processors in MESH (1:2, 1:2). As one can see, like the distribution strategy for the tree, this distribution strategy also divides the mesh data structure into smaller pieces corresponding to the mesh processor structure.

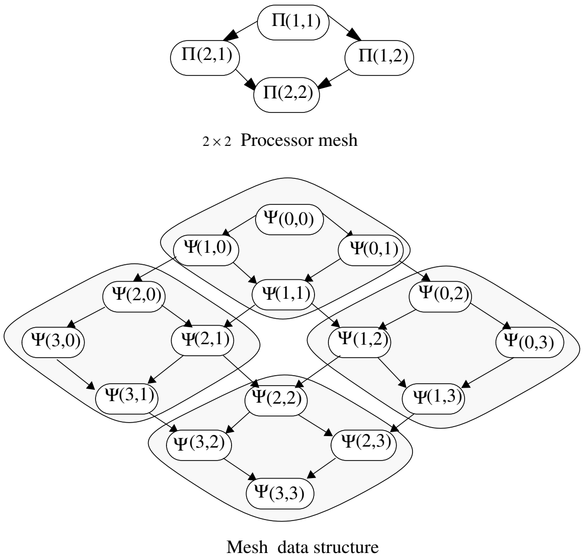

Fig. 2.3. Distributed mesh data structure.

The final example is that of a hierarchical data structure. The data structure represents a list of link lists. Lists at two levels in the hierarchy are distributed along the two dimensions of the MESH processor structure. The lists at the second level of the data structure are doubly linked. The naming strategy used by the two types of lists is the same and consists of consecutive numbers. The distribution of the top level list assigns consecutive pairs of list elements to consecutive processors on the first column of processors. The distribution of a list contained within an element of the top level list is carried out in pairs along the processors belonging to a row in the MESH. The processor row number to which such a list is assigned is inherited from the containing element of the top level list. This is indicated by the assignment of an asterisk to π_1 in the distribution strategy for List. In Fig. 2.4 the names of the nodes are written inside the nodes and the integer pairs are processor ids at which the nodes reside.

```
constant n = 2;
processor MESH (1:n, 1:n);
type
    List is distributed
            struct { int data; *List next; reverse (next,prev) };
    Node is distributed
            struct { *List alist; *Node down };
var *Node ListOfLists;
name List { first: 1; ψ.next: ψ + 1; ψ.prev: ψ - 1 }
name Node {first: 1; ψ.down: ψ + 1 }
```

distribute List **on** MESH { Π=(π_1=*, π_2= ($\lceil\psi/2\rceil$-1) mod n + 1) }
distribute Node **on** MESH { Π=(π_1= ($\lceil\psi/2\rceil$-1) mod n + 1, π_2=1) }

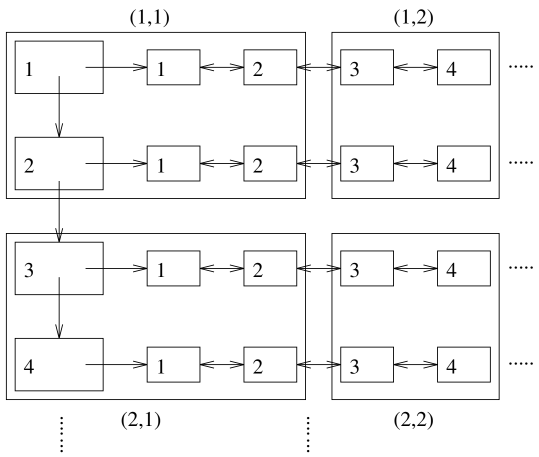

Fig. 2.4. A list of link lists.

The distribution of the tree data structure and the mesh data structures presented in this section are analogous to the cyclic distribution schemes used for static arrays. In contrast the distribution specified for a list of link lists is similar to a block-cyclic distribution used for distributing static arrays.

3. Compiler Support for Regular Data Structures

In this section first a representation of pointers that is suitable for distributed memory machines is developed. This representation is based upon names that are global to all processors rather than simple memory addresses that are local to individual processors. The translation rules for operations involving dynamic data structures to enable owner-computes rule based SPMD execution are described in detail.

3.1 Representing Pointers and Data Structures

On a shared memory machine a pointer variable is simply the memory address of the data item that it points to. To accommodate distributed data structures the notion of pointers must be generalized so that they can be used for the distribution of a data structure across the processors in the system. A pointer variable **P** which points to a node in a *distributed* data structure is represented as $ds < \Psi : [\Pi, \alpha] >$, whose components are interpreted as follows:

P.ds: uniquely identifies the data structure;
P.Ψ: is the *name* of the node in the data structure **ds** to which **P** points;
P.Π: is the *processor* at which the node resides; and
P.α: is the *address* in processor **P.**Π's memory at which the node resides.

The pointer $ds < 0 : [\Pi_0, 0] >$ represents a *nil* pointer and **P.**α**.data** denotes the address of the data field *data* of the node **P** points to. When a new node is added to a distributed data structure its name Ψ is computed from the names of its neighbors using the user specified name generation strategy. From the name of a node Ψ, the processor Π at which it resides is computed.

Each processor represents a distributed data structure as a *hash table* which is identified by **ds**. The hash table is initially empty and a nil assignment to the handle of a distributed data structure clears the hash table. The processor Π is computed from Ψ only when the node is initially created. Each processor stores the information that the node Ψ exists and that it resides at processor Π in the hash table. In addition, the processor Π also stores the address of the node α. The node name Ψ is used to access and store the information. The runtime mechanism for identifying the processor that will execute a given statement uses the information stored in the hash table when the statement computes a value in a node of the dynamic data structure. The same information is also used for generating interprocessor communication required for the execution of the statement under owner-computes rule based SPMD execution.

The hash table is manipulated using the following functions: *PutNode(ds,*Ψ*)* stores information, *GetNode(ds,*Ψ*)* retrieves information, *DeleteNode(ds,*Ψ*)* deletes nodes in the hash table, and *ClearTable(ds)* reinitializes the hash table. It should be noted that although the hash table is replicated at each processor the nodes in the data structure are not replicated. The hash table representation enables us to avoid interprocessor communication during run-time resolution at the expense of storage at each node.

Temporary pointer variables and pointer variables of a local data structure type are also represented as $ds < \Psi : [\Pi, \alpha] >$. In case of local data structures ds is not a hash table but rather the address of the first node in the data structure and the field Ψ is always zero. The field Π is Π_0 if the structure is replicated at all processors; otherwise it is the id of the processor at which the local structure resides. Finally, α is the local address of the node in the processor's local memory. In the case of temporaries initially fields ds, Ψ, and Π are zero and if the temporary points to a single node created through it, then α is the address of that node. However, if the temporary points to a local or distributed data structure its fields take the form associated with local and distributed pointer types. The Ψ and Π values of a temporary can be examined to discern whether it currently points to a single node which does not belong to any data structure, a local data structure, or a distributed data structure. This distinction is important since it determines the semantics of the operations performed using a temporary pointer variable.

3.2 Translation of Pointer Operations

The translation of all types of operations that are performed on pointer variables is discussed in this section. The semantics of each operation depends upon the categories of the pointer variables involved in the operation. Initially all pointers are nil. When a new node is created using a pointer variable, the node's representation is computed and assigned to the pointer variable. As mentioned earlier, the nodes in a distributed data structure are referred to by their names and a distributed data structure is traversed through the generation of these names. The order in which the node names are generated corresponds to the order in which the nodes can be accessed using the links in the data structure. The compiler uses the following communication primitives during code generation:

- the operation SEND(pid,addr) sends the value stored at the address *addr* on the executing processor to processor *pid*;
- the operation global send GSEND(addr) sends the value stored at address *addr* on the executing processor to all other processors; and
- RECV(addr,pid) receives a value from *pid* and stores it at the address *addr* on the executing processor.

In this section f, f^{-1} and g are used to denote a name generation function of a regular pointer, a name generation function of a reverse pointer and a distribution function.

Translation of Node Creation Operations

Nodes can be created using the *new* operation on pointer variables. First the semantics and translation of the *new* operation for handles and temporaries is discussed and then the constructors are considered.

$\boxed{\text{new(P)}}$

P is a distributed handle:

If a node is created using a handle to a distributed data structure it is assumed to be the first node in the data structure whose name can be found from the name generation scheme provided by the user. From the name of the node the processor at which the node will reside is determined from the distribution strategy provided by the user. In case the data structure is a substructure of a hierarchical data structure and the containing structure at the preceding level in the hierarchy is also a distributed structure, then a processor offset is computed based upon the containing data structure. This offset is used in conjunction with the user specified mapping strategy to determine the processor at which the node is to reside.

In order to implement the above strategy with each data structure an attribute $ds.\Pi_{inherit}$ is associated which is set at runtime when the enclosing node in an hierarchical data structure is created. If there is no enclosing node then, it is set to Π_0. The processor offset associated with a data structure ds, denoted as $ds.\Pi_{offset}$, is computed from $ds.\Pi_{inherit}$ and the user specified distribution strategy. For data structures that do not require any offset (eg., simple data structures that are not contained in any hierarchical structure), $ds.\Pi_{offset}$ is simply Π_0. Since a handle is replicated at each processor all processors carry out the assignment. However, only the processor at which the node is to reside allocates storage for the node. The hash table is updated to indicate the state of the data structure.

```
ClearTable(P.ds); P.Ψ = first;
if handle P is contained in a distributed node
    { P.ds.Π_offset = Π_inherit − g(P.Ψ) }
else { P.ds.Π_offset = Π_0 }
P.Π = g(P.Ψ) + P.ds.Π_offset;
if (mypid == P.Π) { P.α = malloc ( SizeOfNode ) } else { P.α = nil }
PutNode(ds,P.Ψ)=(exists=true,processor=P.Π,address=P.α);
Initialize Π_inherit for all enclosed data structures;
```

P is a local handle:

If a node is created using a handle to a local data structure the value of Ψ is zero. If the local data structure is not a substructure of a distributed data structure, then all processors allocate storage since a local data structure is replicated at each processor and each processor assigns its own id to Π. On the other hand if the local structure is a substructure of a distributed structure, then only the processor at which the containing node resides allocates storage for the structure.

```
P.Ψ = 0; P.Π = mypid;
if ((P.Π == mypid) or (P.Π == Π_0))
    { P.α = malloc ( SizeOfNode ) }
```

P is a temporary:

Finally in case of a temporary, all processors allocate storage for a node in their respective local memories. The translation rules described above are summarized in code below. In this code *mypid* represents the id of the processor executing the code. A temporary is replicated at each processor. A temporary is assigned $0 < 0 : [\Pi_0, \alpha] >$ where α is the address of the node. The storage is allocated by each processor in its local memory.

```
P.ds = 0; P.Ψ = 0; P.Π = Π_0; P.α = malloc ( SizeOfNode );
```

$\boxed{\text{new}(P \rightarrow ptr_i)}$

P points to a node in a local data structure:

If the nodes are created using constructors belonging to local data structures a new node is created and added to the data structure as shown below. In addition, if the link involved in the operation is a bidirectional link then code is generated to create a bidirectional link. The value of $P.\Pi$ determines whether the local data structure is a replicated or an unreplicated data structure.

if ((P.Π == mypid) or (P.Π == Π_0)) { addr = malloc (SizeOfNode) }
else { addr = nil }
store pointer $0 < 0 : [P.\Pi, addr] >$ at P.α.ptr_i;
if ptr_i is bidirectional { store $0 < 0 : [P.\Pi, P.\alpha] >$ at addr.$revptr_i$ }

P points to a node in a distributed data structure:

The creation of a node using a constructor belonging to a distributed data structure requires the computation of the name of the new node and assignment of this node to a processor. $CreatePtr_i(ds, \Psi)$ computes and returns the pointer representation of the newly created node. It makes use of the name generation and distribution schemes for assigning the node to a processor. Recall that a node cannot be created using a reverse pointer $P \rightarrow revptr$.

$P \rightarrow ptr_i = CreatePtr_i(P.ds, P.\Psi)$
function CreatePtr_i (ds,Ψ) {
  $\Psi = f(\Psi)$; $\Pi = g(\Psi) + ds.\Pi_{offset}$;
  if (mypid == Π) { α = malloc (SizeOfNode) } **else** { $\alpha = 0$ }
  PutNode(ds,Ψ) = (exists = true, processor = Π, address = α)
  return($ds < \Psi : [\Pi, \alpha] >$)
}

Translation of Traversal Operations

References must be made to the pointer fields of the nodes in the data structure in order to traverse the data structure. In the case of distributed data structures these references are transformed so that the modified code performs the traversal through node *names* using $TraversePtr_i$(ds,Ψ) and $TraverseRevptr_i$(ds,Ψ) functions. On the other hand the traversal of local data structures does not require the computation of *names*.

$\boxed{P \rightarrow ptr_i \ (P \rightarrow revptr_i)}$

P points to a node in a local data structure

if ((P.Π == mypid) or (P.Π == Π_0))
  { P.α.ptr_i (P.α.$revptr_i$) is the address at which the node is stored }

P points to a node in a distributed data structure

TraversePtr_i(P.ds,P.Ψ) (Traverse$Revptr_i$(P.ds,P.Ψ))
returns the representation of $P \rightarrow ptr_i$ ($P \rightarrow revptr_i$)

```
function TraversePtr_i ( ds,Ψ ) {
  Ψ = f(Ψ)
  (exists, Π, α) = GetNode(ds,Ψ)
  if exists { return ( ds < Ψ : [Π, α] > ) }
  else { return ( ds < 0 : [Π_0, 0] > ) }
}

function TraverseRevptr_i ( ds,Ψ ) {
  Ψ = f^{-1}(Ψ)
  (exists, Π, α) = GetNode(ds,Ψ)
  if exists { return ( ds < Ψ : [Π, α] > ) }
  else { return ( ds < 0 : [Π_0, 0] > ) }
}
```

Translation of Pointer Assignments

The building and modification of a dynamic data structure is carried out using pointer assignments. First the affects of a nil assignment are considered and then assignment of one pointer variable to another is considered.

$\boxed{\text{P = nil}}$

A nil assignment to a handle (local or distributed) essentially eliminates the nodes in the entire data structure. On the other hand, a nil assignment to a temporary, or a constructor accessed through a temporary, that points to a node in a data structure eliminates the node. A nil assignment to a temporary variable which does not point to any other data structure simply sets the pointer to nil and does not have an affect on any other data structure. In summary a major function of nil assignments is to delete one or all nodes from a local or distributed data structure.

P is a temporary that does not point to a data structure

P = $0 < 0 : [\Pi_0, 0] >$

P is a distributed constructor

if (P.Π == mypid)
{ delete node corresponding to $P.\Psi$ from hash table for $P.ds$ }

P = $ds < 0 : [\Pi_0, 0] >$

P is a local constructor

if ((P.Π == mypid) or (P.Π == Π_0))
{ delete node associated with P.Ψ }

P = $ds < 0 : [\Pi_0, 0] >$;

P is a distributed handle
ClearTable(P.ds); P = $ds < 0 : [\Pi_0, 0] >$
P is a local handle
`if` ((P.Π == mypid) or (P.Π == Π_0))
{ delete entire local structure }
P = $ds < 0 : [\Pi_0, 0] >$;

$\boxed{\mathrm{P = Q}}$

An assignment of one pointer variable to another pointer variable can have the following effects depending on the types of the pointer variables involved. A pointer assignment can result in simple *copying* of a pointer, *addition* of a node to a data structure, *splitting* of a distributed data structure into two distinct data structures, and *merging* of two data structures into one. These actions are described next in greater detail below.

Copying a pointer: The copying of pointers occurs when dealing with local data structures or when the assignment is made to a temporary variable. If P is a temporary pointer variable, which currently does not point to any data structure, then after it is assigned Q, both P and Q point to the same place. If both P and Q are pointers associated with local data structures (i.e., a local handles or constructors), then after the assignment P also points to the same address as Q. Assignments to handles of local data structures may result in addition and deletion of nodes from the data structure that it represents.

P.ds = Q.ds; P.Ψ = Q.Ψ; P.Π = Q.Π; P.α = Q.α

Adding a single node: Consider the situation in which Q is a temporary pointer variable through which a single node has been created. This node can be added to a local or distributed data structure by assigning Q to P, where P is a handle or constructor of a local or distributed data structure. After the assignment, Q behaves as a pointer to the data structure to which the node has been added. Notice that since the node associated with the temporary pointer Q is replicated at each site, no communication is required during this operation.

`if` ((P.Π == mypid) or (P.Π == Π_0))
{ P.α = Q.α }
Q.Ψ = P.Ψ; Q.Π = P.Π;
PutNode(P.ds,P.Ψ)

Restructuring a distributed data structure: If both P and Q point to the same distributed data structure and P is not a constructor, then assignment of Q to P simply results in pointer copying. However if P is a constructor, then the data structure is restructured. The node pointed to by Q now takes the position that P points to, and all nodes connected directly or indirectly through constructors originating at Q are also shifted to new positions. After restructuring, Q also points to the position to which the constructor P points.

The restructuring is performed by function *Reposition* which also generates interprocessor communication to transfer nodes to their new sites.

```
Reposition(Q.ds,P.ds,Q.Ψ,P.Ψ);
Q.Ψ = P.Ψ; Q.Π = P.Π; Q.α = P.α;

function Reposition(oldds,newds,oldΨ,newΨ) {
  nodelist = {(oldΨ,newΨ)} ∪ {(old,new): old is derived from oldΨ}
  for each pair (old,new) ∈ nodelist st
      (mypid=g(old)) and (mypid≠g(new)) {
         (exists,Π,α) = GetNode(oldds,old); DeleteNode(oldds,old);
         SEND(g(new),α);
         PutNode(newds,new)=(exists=true,processor=g(new),address=0)
  }
  for each pair (old,new) ∈ nodelist st
      (mypid≠g(old)) and (mypid=g(new)) {
         α = malloc( SizeOfNode ); DeleteNode(oldds,old);
         RECV(α,g(old));
         PutNode(newds,new)=(exists=true,processor=mypid,address=α)
  }
  for each pair (old,new) ∈ nodelist st
      (mypid=g(old)) and (mypid=g(new)) {
         (exists,Π,α) = GetNode(oldds,old); DeleteNode(oldds,old);
         PutNode(newds,new)=(exists=true,processor=mypid,address=α)
  }
  Update temporaries pointing to repositioned nodes
}
```

Merging and splitting of distributed data structures: Assignments involving handles and constructors belonging to different data structures, at least one of which is distributed, results in transfer of nodes between the data structures. If P is the handle of a data structure and it is assigned Q which is a handle or a constructor of another data structure, then all nodes starting from Q's position are transferred to the data structure corresponding to P. Interprocessor communication must be generated to reposition the nodes at their new sites. If nodes are transferred from a distributed data structure to a replicated local data structure, then the above operation will essentially result in broadcast of the data structure to all processors. The restructuring is performed by function *Reposition*.

Translation of Node Data Accesses

Accessing data stored in local data structure is straightforward if it is replicated at each processor. However, in case of data in non-replicated data structures or distributed data structures, first the processor at which the data resides is located. Next either to store or retrieve data, interprocessor communication may be required. In order to perform these accesses instructions LOAD and STORE are defined. The LOAD operation copies the contents of data field *data* of the node pointed to by the pointer P into a scalar t located at processor *pid*. The STORE operation copies the contents of a scalar t residing at processor *pid* into the data field *data* of the node pointed to by the pointer P. In the LOAD/STORE operations defined below the scalar variable t is a temporary that resides at processor *pid*.

```
LOAD t, pid, P, data == t_pid = P→data
   if ( P.Ψ == P.α == 0 and P.Π == Π_0 ) { error – nil pointer }
   elseif ((P.Ψ == 0) and (P.Π == Π_0)) {
     – P points to a replicated local data structure
     if (mypid == pid ) { t = P.α.data }
   } else { – P points to a non-replicated local or distributed structure
     if (P.Π == mypid == pid) { t = P.α.data }
     elseif (P.Π == mypid ≠ pid) { SEND(pid,P.α.data) }
     elseif (P.Π ≠ mypid == pid) { RECV(t,P.Π) }
     else { – the processor is not involved in the execution because
              P.Π ≠ mypid ≠ pid }
   }
STORE t, pid, P, data == P→data = t_pid
   if ( P.Ψ == P.α == 0 and P.Π == Π_0 ) { error – nil pointer }
   elseif ((P.Ψ == 0) and (P.Π == Π_0)) {
     – P points to a replicated local data structure
     if ( mypid == pid ) { GSEND(t) }
     else RECV(P.α.data,pid)
   } else { – P points to a non-replicated local or distributed structure
     if (P.Π == mypid == pid) { P.α.data = t }
     elseif (P.Π == mypid ≠ pid) { RECV(P.α.data,pid) }
     elseif (P.Π ≠ mypid == pid) { SEND(P.Π,t) }
     else { – the processor is not involved in the execution because
              P.Π ≠ mypid ≠ pid }
   }
```

An Example

The translation of a small code fragment using the rules presented in this section is demonstrated next. The code fragment shown below traverses a distributed link list of integers. The integer value in each of the elements of the link list is incremented.

```
constant n = 4;
processor ARRAY (n);
```

```
type Node struct { int data; *Node next };
     List is distributed Node;
name List { first: 1; ψ.next: ψ+1 };
distribute List  ARRAY { π = (ψ − 1)mod n + 1 };
var *List alist; *Node tlist;

tlist = alist;
while ( tlist ≠ Nil ) {
      tlist→data = tlist→data + 1;
      tlist = tlist→next
}
```

The loop for the traversal of the linked list is transformed into the loop shown below which will be executed in parallel by all processors in ARRAY. Under SPMD execution each processor will be responsible for incrementing list elements that reside in its local memory. As we can see the traversal is achieved through name generation and hence it does not require interprocessor communication. When the entire list has been traversed, the function $TraverseNext(ds, \psi)$ returns nil. The temporary pointer *tlist* used to traverse the link list is replicated on all processors in ARRAY.

```
tlist = alist;
while (tlist != nil) {
      /* tlist→data = tlist→data + 1 */
      LOAD t, tlist.Π, tlist, data
      p,tlist.Π = p,tlist.Π + 1
      STORE p, tlist.Π, tlist, data
      /* tlist = tlist→next */
      tlist = TraverseNext(tlist.ds,tlist.ψ)
}
function TraverseNext ( ds,ψ ) {
      ψ = ψ + 1
      (exists, ψ, α) = GetNode(ds,ψ)
      if exists { return ( ds < ψ : [π, α] >) }
      else { return ( ds < 0 : [π0, 0] > ) }
}
function CreateNext ( ds,ψ ) {
      ψ = ψ + 1
      π = ProcId( ψ )
      if (mypid == π)
          { α = malloc ( SizeOfNode ) }
      else { α = 0 }
      PutNode(ds,ψ) =
          (exists = true, processor = π, address = α)
      return( ds < ψ : [π, α] > )
}
```

```
function ProcId( ψ )
        { return ( (ψ - 1) mod 4 + 1 ) }
```

4. Supporting Irregular Data Structures

In this section we present extensions of the approach described for regular data structures so that it can also be used to support irregular data structures. In the examples considered so far the data structures considered were regular in nature and the name generated for each data item in the data structure was unique. In case of irregular data structures it is not possible to precisely specify the topology of the data structure and hence the name generation strategy. One possible approach that can be employed approximates an irregular data structure by a regular data structure for the purpose of computing the positions and names of the nodes. A consequence of such an approximation is that multiple names may be created for nodes with multiple parents. Thus from the perspective of our approach, regular data structures may be viewed as unaliased data structures and irregular data structures may be viewed as aliased data structures. The following language construct allows the user to communicate to the compiler as to whether a distributed data structure is aliased or unaliased.

```
type NodeType is { aliased | unaliased } distributed BaseType;
```

As an example, consider a distributed directed acyclic graph (DAG) shown in Fig. 4.1. In the mapping process the DAG has been approximated by a binary tree and mapped to a binary tree of processors using the strategy described previously. This process causes the nodes in the DAG with multiple parents to have multiple names. A different name is used to access the node depending on the parent through which traversal is being performed. The shared node can be accessed using node name 5 through node 2 or using node name 6 through node 3. When a node is initially created it is assigned to the processor indicated by the node name. When additional links are created to point to the same node (as by the assignment statement in Fig. 4.1), an alias is created with each additional link while the location of the node remains the same. For the DAG shown in Fig. 4.1, the shared node will be located on processor P_5, if it is created by executing the statement "$new(P \rightarrow leftchild \rightarrow rightchild)$". If it is created using statement "$new(P \rightarrow rightchild \rightarrow leftchild)$" then it will be located at processor P_6. From the above example it is clear that an irregular data structure must be declared as an aliased data structure. Although in this example we considered an acyclic data structure, in general this approach handles cyclic data structures as well.

In order to implement aliasing, the node information stored at each processor should be augmented to indicate whether the node is an alias of another node. If the node is an alias, the field $orig\Psi$ will contain the position of the node of which the current node is an alias, otherwise *orig*Ψ is zero. A pointer assignment which assigns to a handle/constructor another pointer, that points to a node in the data structure, causes the creation of an alias. A nil assignment causes the destruction of a name. If the destroyed name has aliases one of these aliases must be chosen as the new name for the node and the node information of all aliases is updated. Thus, for each node we must also maintain the list of its aliases. The field *aliases* contains this information. In addition we must also traverse the graph and change the names of all nodes that were derived from the destroyed name. This, processor is quite similar to the restructuring process described for regular data structures. However, there is one important difference. Although the node *names* are modified the nodes are not migrated. This is because for an aliased data structure it is not clear whether the migration will result in a superior distribution of nodes.

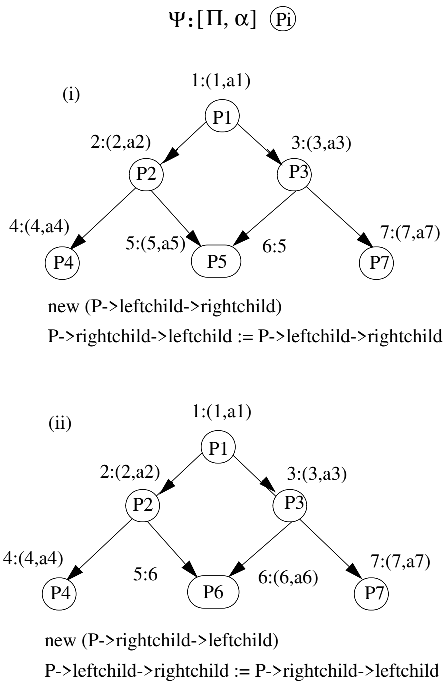

Fig. 4.1. A directed acyclic graph.

$\boxed{\text{Creation of an alias}}$

P = Q, where P is a distributed handle/constructor:

PutNode(P.ds,P.Ψ) =

(exists=true, processor=Q.Π, address=Q.α, origΨ=Q.Ψ, aliases = null)

add to the list of aliases of origΨ another alias P.Ψ

Destruction of a name

```
P = nil, where P is a distributed handle/constructor:
  let P.aliases = {a_1,a_2,...a_n}
  select a_1 as the new name for the node
  PutNode(P.ds,a_1) = (exists=true, processor=P.Π,
      address=P.α, origΨ=0, aliases={a_2,...a_n})
  for a_2....a_n {
      origΨ = a_1; aliases = aliases - {a_1};
      traverse data structure using P.Ψ and
      find all node names that were derived from P.Ψ
      recompute the position field of these nodes from a_1
  }
```

The $Traverseptr_i(ds, \Psi)$ and $TraverseRevptr_i(ds, \Psi)$ must be modified so that when nodes are traversed using their aliases, the aliases are replaced by the original *names*. This, is important because if this is not done then creation of a node's alias will require the creation of aliases for all nodes reachable through pointer traversals from the newly aliased node. For a structure with loops, such as a circular link list, infinite number of aliases will be created. The function $CreatePtr_i(ds, \Psi)$ remains the same because no aliases are ever created during the creation of new nodes.

```
function TraversePtr_i ( ds,Ψ ) {
   Ψ = f(Ψ)
   (exists, Π, α, origΨ) = GetNode(ds,Ψ)
   if not exists { return ( ds < 0 : [Π_0, 0] > ) }
   elseif (origΨ ≠ 0) { return ( ds < origΨ : [Π, α] > ) }
   else return ( ds < Ψ : [Π, α] > )
}
function TraverseRevptr_i ( ds,Ψ ) {
   Ψ = f^{-1}(Ψ)
   (exists, Π, α, origΨ) = GetNode(ds,Ψ)
   if not exists { return ( ds < 0 : [Π_0, 0] > ) }
   elseif (origΨ ≠ 0) { return ( ds < origΨ : [Π, α] > ) }
   else return ( ds < Ψ : [Π, α] > )
}
```

It should be noted that the implementation of an aliased data structure is more general than an unaliased data structure. Therefore an unaliased data structure can also be correctly handled by the implementation for an aliased data structure. In such a situation no aliases will be created and hence the list of aliases will always be empty for all nodes.

5. Compile-time Optimizations

The focus of this chapter has been on translation or code generation rules SPMD execution. In reality to obtain good performance, optimizations must also be performed. In context of programs based upon shared data arrays, communication optimizations have been found to be useful [2]. Similar optimizations can also be applied to programs with dynamic data structures. Some additional optimizations should be performed in context of dynamic data structures.

Faster traversal functions: Dynamic data structures are accessed through link traversals which can be slow in contrast to indexed random accesses to array elements. Thus, techniques to allow faster traversal of dynamic data structures would be of potential benefit. In the linked list example of the preceding section, each processor iterates through all of the node positions in the linked list, although each processor only operates on those elements of the list that are local to itself. Through static analysis the compiler can determine that the iterations of the loop can be executed in parallel. Thus, the overhead of traversal can be reduced by using a function which only iterates through the nodes that reside on the same processor.

Parallel creation of dynamic data structures: While it is clear that parallelism is exploited during SPMD execution of code that operates on a distributed data structure, the same is not true during the creation of a distributed dynamic data structure. This is because each processor must update its hash table when a node is added to the data structure. Through compile-time analysis the situations in which a data structure can be created in parallel by a distinct creation phase can be identified. The updating of the hash table can be delayed to the end of the creation phase. Each processor can broadcast the names of the nodes that it created at the end of the creation phase to update the hash tables of other processors. In this approach parallelism is exploited during the creation of the data structure and the updating of the hash tables is also carried out in parallel.

Storage optimization: Another source of significant overhead in programs with dynamic data structures arises due to dynamic allocation and deallocation of storage. During the restructuring of a data structure we can reduce the storage allocation overhead by reusing already allocated storage. For example, if elements of a distributed data structure are redistributed among the processors, storage freed by elements transferred from a processor can be reused by elements transferred to the processor.

6. Related Work

This chapter addressed the problem of supporting dynamic data structures on distributed-memory machines using owner-computes rule based SPMD execution. The implementation of globally shared dynamic data structures is made possible by assignment of global names to the elements in a data structure. The compiler is responsible for translating a sequential program into parallel code that executes in SPMD mode. In spirit this approach is similar to the approach originally proposed by Callahan and Kennedy for programs with shared data arrays [1]. The techniques described in this chapter differ from the preliminary version of this work [3] in the following way. In earlier work aggregate operations, such as merging and splitting, of distributed data structures were not allowed and hierarchical data structures were not considered.

An alternative strategy for supporting dynamic data structures has been proposed in [7]. While the approach presented in this chapter relies on the transfer of distributed data among the processors to enable the execution of program statements, the approach proposed by Rogers et al. relies of migration of computation across the processors to enable the execution of statements that require operands distributed among various processors.

To obtain good performance, programs with dynamic data structures must be transformed to expose greater degrees of parallelism. Furthermore techniques for identifying data dependences among program statements are required to perform these transformations. Language extensions can also be developed to facilitate the analysis. In [4, 5] Hendren et al. address the above issues.

Acknowledgement. This work was supported in part by the National Science Foundation through a Presidential Young Investigator Award CCR-9157371.

References

1. Callahan, D., Kennedy, K. (1988): Compiling programs for distributed-memory multiprocessors. Journal of Supercomputing **2**, 151–169
2. Gong, C., Gupta, R., Melhem, R. (1992): Compilation techniques for communication optimizations on distributed-memory machines. Proceedings of International Conference on Parallel Processing **II**, St. Charles, Illinois, 39–46
3. Gupta, R. (1992): SPMD execution of programs with pointer-based data structures on distributed-memory machines. Journal of Parallel and Distributed Computing **16(2)**, 92–107
4. Hendren, L.J., Hummel, J., Nicolau, A. (1992): Abstractions for recursive pointer data structures: improving the analysis and transformation of imperative programs. Proceedings of ACM Conference on Programming Language Design and Implementation, 249–260
5. Hendren, L.J., Gao, G.R. (1992): Designing programming languages for analyzability: a fresh look at pointer data structures. Proceedings of the International Conference on Computer Languages, Oakland, California, 242–251
6. Hiranandani, S., Kennedy, K., Koelbel, C., Kremer, U., Tseng, C.-W. (1991): An overview of the Fortran D programming system. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, California

7. Rogers, A., Carlisle, M.C., Reppy, J.H., Hendren, L.J. (1995): Supporting dynamic data structures on distributed-memory machines. ACM Transactions on Programming Languages and Systems **17(2)**, 233–263
8. Rothberg, E. (1992): Exploiting memory hierarchy in sequential and parallel sparse Cholesky factorization. Ph.D. Thesis, Technical Report STAN-CS-92-1459, Stanford University
9. Samet, H. (1989): The Design and Analysis of Spatial Data Structures. Addison-Wesley
10. Warren, M.S., J.K. Salmon (1993): A parallel hashed oct-tree n-body algorithm. Proceedings of Supercomputing
11. Zhang, G., Elman, H.C. (1992): Parallel sparse Cholesky factorization on a shared memory multiprocessor. Parallel Computing **18**, 1009–1022