ABSTRACT

While massively parallel processing promises a high performance implementation for various applications, it also exhibits the difficulty of programming in a modular way. Programs for conventional programming models can be structured as a hierarchy of clients and servers that can be individually implemented using fine grained parallel processors; however, the clients and servers in this approach are separate entities and, hence, create a von Neumann bottleneck that is unacceptable for fine grained parallel systems. In this paper, we present methods of merging distributed clients and servers to allow efficient parallel operations. In particular, we propose parallel primitives as well as normalized representation schemas for efficiently implementing various abstract data types. The mapping strategies to reduce communication cost for various abstract data types on the hypercube structure are also discussed.

Keywords: Abstract Data Types, Distributed Representation, Mapping Strategies, Multiple Entry Data Structures, Performance Analysis, SIMD Hypercube Machines.

INTRODUCTION

Fine grained parallel supercomputers hold great promise for achieving potentially dramatic speed ups in computing power. To realize the full capabilities of these machines for various applications (robotics, expert systems, simulation, databases), ingenious and fundamentally new kinds of data parallel algorithms must be devised. These include memory based models for databases, parallel pattern matching and constraint resolution for expert systems, and direct modeling of physical systems for simulation purposes.

A major problem which impedes the widespread use of fine grained parallel architectures is the difficulty of programming these machines in a modular way. Conventional distributed and coarse grained parallel programming models do not scale up to fine grained parallel systems. Programs for the former systems can be structured as a hierarchy of clients and servers that can be individually implemented using fine grained parallel processors (Ref. 1), such as broadcasting sequential processors (BSP) for high performance abstract data type components, systolic arrays for functional components, and processor networks for interface components. However, the clients and servers in this approach are separate entities, thus creating a von Neumann bottleneck that is unacceptable for fine grained parallel systems.

In this paper we investigate efficient implementation of abstract data types on SIMD hypercube machines. To achieve high performance in parallel programs, the requests of various clients of an abstraction should be processed in parallel. One way to overcome the von Neumann bottleneck is to merge the clients with the servers of an abstraction. We first explore the essentials of this implementation theoretically, and then realize it for various abstract data types. In particular, we propose parallel primitives as well as normalized representation schemas for efficiently implementing different abstract data types. Since the performance of an abstract data type component may vary considerably depending on the placement of its data elements on the physical processors, the mapping strategies to reduce communication cost are also discussed. These abstract components can serve as building blocks for implementing other parallel components, such as parsers, constraint resolvers, and pattern recognizers.

A THEORETICAL EXPLORATION

In this section we investigate the essential elements in implementing a high performance abstract data type component on SIMD hypercube machines.

An abstract data type component is a collection of functions that implement some mathematical objects, such as lists, queues, and sets. Internally, an abstract data type, A, is a tuple

$$A = (D, F)$$

where $D$ is a data structure that can be expressed as a state machine, and $F$ is a collection of functions that operate upon $D$. The data structure, $D$, is also a tuple

$$D = (O, R)$$

in which $O$ denotes the collection of data elements for storing states of $D$ and $R$ denotes the relations among these elements. Thus, $O$ can be expressed by
where \( n \) is the total number of data elements in \( O \), and \( R \) is a function defined on \( a_i, 1 \leq i \leq n \). For data intensive applications on SIMD machines, \( R \) can be defined as a collection of relations \( \{r_i | 1 \leq i \leq n \} \) such that, for element \( a_i \), the associated relation \( r_i \) denotes the absolute position of \( a_i \) in \( D \) or the relative relations between \( a_i \) and other elements \( a_j, j \neq i \). In this case, \( D \) can be expressed as a collection of \( a_i \) and \( r_i \); i.e.,

\[
D = \{(a_i, r_i) | 1 \leq i \leq n\}
\]

where \( n \) may be very large.

A data structure, \( D \), may be represented in several ways on SIMD machines. To search for the best representation of a data structure, we first notice that an SIMD machine such as the Connection Machine (Refs. 2-3) may contain up to tens of thousands of processors, and at a given time can operate only on data elements of identical type. In order to fully exploit the underlying architecture of SIMD machines, a data structure must be represented in a distributed way by having one processor per data element. This representation is called a distributed representation which can also be defined by a mapping function \( f \) from \( D \) to the network of processing elements, say \( \{PE_j\} \), i.e.,

\[
h : (a_i, r_i) \rightarrow PE_j \quad \text{for} \quad 1 \leq i \leq n \quad \text{and} \quad 1 \leq j \leq N
\]

where \( N \) is the total number of \( PE \)'s. Distributed representations can facilitate concurrent operation upon a large set of data and, hence, achieve the maximum degree of parallelism for the implementation of an abstract data type.

However, a distributed representation for \( D \) by itself does not guarantee that requests from multiple clients can be processed simultaneously. To achieve this, the clients and servers of an abstraction must be merged so that a \( PE \) which acts as a server for an abstraction can also act as a client of that abstraction. This is implemented via a kind of distributed representation called a normalized representation which is defined by the above mapping function with

\[
\text{address}(PE_j) = f(a_i)
\]

where \( f \) is a one-to-one function. That is, the address of \( PE \) which is allocated to \( a_i \), by using \( h \), can be computed simply from \( a_i \) or even \( i \). The ability to normalize the data structure is the key to the parallel processing of multiple client requests.

In addition to multiple entry data structures based on normalized representations, we have to provide parallel primitives for an abstraction so that multiple requests can be processed simultaneously. First, consider the computation pattern of a basic parallel primitive \( f \). The computation pattern of \( f \) is used to denote the effect of \( f \) by showing the operation points and data flow patterns in the network of \( PE \)'s. The data flow patterns in a computation pattern are also called a communication pattern. Let \( g \) denote the computation pattern of \( f \). Then \( g \) is defined as follows.

\[
g = \bigcup_{1 \leq i \leq N} g_i
\]

where \( N \) is the total number of \( PE \)'s, and \( g_i \) is the computation pattern by tracing the effect of applying \( f \) to \( PE_i \).

A computation pattern \( g \) is said to be regular if all of the \( g_i \)'s have the same pattern; otherwise, it is irregular. In terms of the number of phases in \( g \) and the change in communication pattern across different phases, the useful parallel primitives on SIMD machines can be classified into five types (see Figure 1). They are

(A) Single phase, single \( PE \) (local processing). This is the simplest type of primitives. A set of \( PE \)'s perform the same operation (i.e., \( f_i \)) upon their own local data elements. No communication is required for this type of primitives.

(B) Single phase, two \( PE \)'s. A set of data elements are moved or copied from original locations (a set of \( PE \)'s) to new locations (a new set of \( PE \)'s). The computation pattern here may be regular or irregular. The messages transferred can be regarded as requests from a set of clients to a set of servers. If required, the servers may return some results to the clients.

(C) Multiple phases, identical pattern (multiple \( PE \)'s). The computation pattern for this type of primitives consists of \( k \) phases, where \( k \) is determined statically or dynamically, and \( k > 1 \). The computation pattern is identical in each of the \( k \) phases.

(D) Multiple phases, deterministic pattern (multiple \( PE \)'s). This type of primitives performs a series of operations in a deterministic fashion (usually, tree-like fashion) on a hypercube structure. Thus, the computation pattern for this type of primitives is regular, but changes from phase to phase. The set of active \( PE \)'s is not defined and thus we may have many different primitives for this type. Two simple examples are aggregate and broadcast. Aggregate is used to collect some summary information from a set of \( PE \)'s, while broadcast is used to duplicate a data element to a set of \( PE \)'s.

![Figure 1. Classifications of basic parallel primitives.](image-url)
Multiple phases, random pattern (multiple PE's). The computation patterns here are irregular. The addresses and the number of intermediate PE's are determined dynamically. Since more than one message may pass through some PE's, this type of primitives implements the concept of pipelining in transferring messages to destinations.

It should be noted that primitives of types B and E may make heavy use of random communication feature of the underlying architectures, and primitives of type D take advantage of the hypercube structure of PE's. The above primitives form a basis for constructing higher levels of primitives to efficiently implement an abstract data type. A useful primitive may be a combination of several of the above operations.

PRACTICAL IMPLEMENTATION

In this section we discuss practical methods to implement a high performance abstract data type component on SIMD hypercube machines. In terms of \( r_i \), we first classify abstract data types on SIMD machines into the following three categories:

1. Unrelated collections: For an unrelated collection, we have
   \[ r_i = 0, \quad 1 \leq i \leq n. \]
   This means that there is no relation among different data elements. Thus,
   \[ D = \{ a_i | 1 \leq i \leq n \}. \]
   Examples of unrelated collections are sets, bags, search tables, and symbol tables.

2. Crystalline collections: Here, for component \( A = (D,F) \), each element, \( a_i \), has an absolute position in \( D \). In this case, \( r_i \) is defined by storing the logical indices of \( a_i \) in \( r_i \); i.e.,
   \[ r_i = f(i), \quad 1 \leq i \leq n. \]
   Examples of crystalline collections are lists (with indices), vectors, matrices, etc.

3. Amorphous collections: Here, for component \( A = (D,F) \), the position of \( a_i \) in \( D \) is determined by the position of \( a_j \) relative to other elements \( a_j, j \neq i \). In this case, \( r_i \) is defined by storing in \( r_i \) the addresses of neighboring elements of \( a_i \) in \( D \); i.e.,
   \[ r_i = f(\&o_{j1}, \ldots, \&o_{jk}), \quad 1 \leq i \leq n \]
   where \( 1 \leq j_1, \ldots, j_k \leq n; \&o_{jm}, 1 \leq m \leq k \), denotes the address of element \( o_{jm} \); and \( o_{j1}, \ldots, o_{jk} \) are neighbors of \( o_i \) in \( D \). Examples of amorphous collections are lists, trees, graphs, and semantic networks that use pointer-based representation.

For an abstract data type \( A = (D,F) \) with \( D = \{ (o_i, r_i) \} \), the simplest one of its distributed representations is to allocate a distinct PE for each \((o_i, r_i)\) in any convenient way.

This representation can be created efficiently and allows clients to access any point of its data structure directly; however, it may need a traversal of the whole data structure, which is time-consuming, to locate a specified element. A suitably normalized form of an abstraction allows its elements to be locatable efficiently. Two approaches are possible for the normalized representation of abstract data types in SIMD machines, and are described in the following.

1. Based on vectors with indexed access. If the data elements of an abstraction are identified by unique sets of indices, then the locations of the data elements (i.e., the addresses of the PE's allocated for those data elements) can be uniquely determined from these indices.

2. Based on vectors with hashed access. If the data elements of an abstraction are identified by unique keys, then we can apply a parallel hash function on these keys to determine the distinct addresses of all data elements. This approach requires a predetermined hash function along with a collision resolution scheme.

Several type-independent primitives are useful for many abstract data types and are described below.

1. \( \alpha(op) \) (Ref. 4): The symbol \( \alpha \) denotes \( \text{ApplyToAll} \); it applies the same \( op \) to all of the data elements at a time; it is a "single phase, single PE" operation.

2. Copy: This primitive is used to copy data elements from one set of PE's to another set of PE's; it is a "single phase, two PE's" operation.

3. Multiple copies: This primitive will copy data elements from one location \( PE \) to multiple locations \( PE \) \( (e.g., \text{broadcast}) \); it is a "multiple phases, deterministic pattern" operation.

4. \( \text{copy}(op) \) (Ref. 4): The symbol \( / \) denotes \( \text{insert} \); we also denote this by \( \text{reduce}(op) \); it reduces a set of elements using \( op \) and returns a scalar value; it is a "multiple phases, deterministic pattern" operation.

In the following subsections, we discuss the data representation and primitive operations for unrelated, crystalline, and amorphous collections, respectively.

Unrelated Collections

An unrelated collection may be in either a random or a normalized representation. The random representation is created through the parallel evaluation of "predicates" upon a collection of data elements having a distributed representation. An unrelated collection will be in a normalized representation if it is created through the use of parallel functional operations such as the union, intersection, and difference operations for sets. The characteristics of unrelated collections are that their data elements are identified by distinct keys, and thus their normalized representation are based on vectors with hashed access.
Two useful primitives for unrelated collections are described in the following.

**Normalize:** It is used to transform an unrelated collection in a random form to the normalized form, and is a “multiple phases, random pattern” operation.

**Search:** It is used to search for a set of keys in parallel. It is also a “multiple phases, random pattern” operation.

Before mentioning any algorithm, we introduce the following notations:

- \( p \) denotes the address of the current PE,
- \( \text{var} \) is a variable in the host,
- \( \text{var}(p) \) is a local variable in \( PE(p) \),
- \( \text{vector}[0..S-1](p) \) is a local vector of size \( S \) in \( PE(p) \),
- “\( = \)” denotes a local assignment within a PE,
- “\( \leftarrow \)” indicates data movement from one PE to another, and
- “\( (op) \)” indicates an inter-PE data movement which has specified the operation \( op \) to be performed by the destination PE.

The last two notations are implemented via four network primitives, namely, send, receive, EmptyQueue, and ResetQueue. The functionalities of these four primitives are explained below.

1. **send:** It is used to assemble packets for each active PE, and then pump them into the interconnection network.
2. **receive:** It enables a PE to fetch the packet at the head of its input queue if that queue is not empty.
3. **EmptyQueue:** It is used to check the existence of any received packet in the input queue for each PE.
4. **ResetQueue:** It is used to discard all packets remaining in queues or in the network.

As an example, consider the parallel retrieval of attributes from a search table. In BSP, this must be done in several cycles. In each cycle an identifier is broadcast to the processors one of which returns the desired attribute (Figure 2a). To retrieve these in parallel we must first obtain a normalized representation of the search table, which is based on vectors with hashed access. The code for parallel fetch is as follows (see Figure 2b):

```plaintext
Procedure ParallelFetch (client : Boolean; id, ide : KeyType; n : Natural; status : StatusType; var attr : AttributeType);
var j, source : 0..n - 1;
  idp : KeyType;
  PacketType : (retrieve, attributes);
begin
  if client then
    j(p) := h(idc(p));
    send ("retrieve", idc(p), p) to PE(j(p));
  while NOT EmptyQueue(p) do
    if NOT EmptyQueue(p) then
      receive (PacketType(p), idp(p), source(p));
      if PacketType(p) = "attributes" then
        store the received attributes in attr
    end;
  end;
end;
```

Here, \( \text{client}(p) \) specifies whether \( PE(p) \) has a key ide(p) the attributes of which need be fetched; \( n \) is the total number of processors. The search table is represented by two variables, id denoting keys and attributes, at each PE. If \( PE(p) \) has a key stored in it then its status is set to \( OCCUPIED \) in which case idp contains the key and attributes contains the associated attributes; otherwise, its status is set to \( FREE \), indicating that \( PE(p) \) does not yet have a key.

Each client uses a hash function \( h \) to determine the address of \( PE(j) \) to which it will transmit its request for ide using the send primitive. When a \( PE \) receives a packet, it first checks the type of that packet, PacketType. If PacketType is "attributes", then this \( PE \) should store the fetched attributes in attr; otherwise, PacketType is "retrieve" and this packet is a request for some key. When a \( PE \) receives a request for key idp, then (a) if its status is \( FREE \) then it sends the message "id does not exist" to the client; (b) if its status is \( OCCUPIED \) and idp = id then it sends the associated attributes, attributes, to the client; otherwise, it forwards the request for idp to the adjacent \( PE \) which has address \( ((p + 1) \mod n) \). Each \( PE \) repeats this process till its input queue is empty; this is indicated by a \( TRUE \) value for EmptyQueue. It should be noted that the con-
ditions expressed in a while statement are used only for determining the termination time of that while loop; they are not used for selecting the set of active PE's to execute the instructions within the loop.

Crystalline Collections

A crystalline collection may have one of two possible representations, namely, index-based and normalized representations. In the index-based representation of a crystalline collection, each data element has an index to indicate its relative position in the data structure, irrespective of where it may actually be allocated. The normalized representation of crystalline collections are based on vectors with indexed abstraction for efficient binary operations, and (2) the stream processing of data.

The following primitives are useful for crystalline collections.

Shift(data/operation): This is applicable for crystalline collections in a normalized form. It moves data or operation forward, backward, or along a ring. The notation shift(operation) means that the processor adjacent to the current one is activated. This primitive is a "single phase, two PE's" operation.

Align: It will move a set of data elements from one set of PE's to a new set of PE's. One example is to transpose a matrix. This primitive is also a "single phase, two PE's" operation.

Prefiz/postfix(op): This primitive will compute in parallel all prefixes or postfixes of vectors using the specified op. It is a "multiple phases, deterministic pattern" operation.

Normalize: This primitive is used to transform a crystalline collection from the index-based representation to the normalized representation, and is a "single phase, two PE's" operation.

Enumerate: A crystalline collection may be created in a temporary representation which is neither an index-based nor a normalized form. In this temporary representation, the data elements are allocated sequentially, but discontinuously. Enumerate is used to transform a crystalline collection from this temporary representation to the index-based representation. This primitive is a "multiple phases, deterministic pattern" operation.

Compact: This primitive is used to transform a crystalline collection from the temporary representation (see enumerate) to the normalized representation in which all data elements are allocated sequentially and contiguously. It is a "multiple phases, identical pattern" operation.

As an example, consider the multiplication of two matrices, \( C = A \times B \), where the dimensions of \( A \), \( B \), and \( C \) are \( L \times M, M \times N \), and \( L \times N \), respectively. Let \( n = \lceil \log_2 (\max(L,N)) \rceil + \lceil \log_2 M \rceil \), \( m = \lceil \log_2 M \rceil \), and \( M' = 2^m \). Thus the total number of processors required is \( 2^n \). Initially, the elements of matrix \( A \) are stored in processors in row-major order, and matrix \( B \) in column-major order. That is, the element \( a_{ij} \) of \( A \) is stored in the processor which has address \( p = i \times M' + j \); and for the element \( b_{jk} \) of \( B \), \( p = k \times M' + j \). After multiplication, element \( c_{ik} \) of the resulting matrix will be stored in \( C[k](i), \) the \( k \)th entry of vector \( C \) in processor \( i \) which has address \( p = i \times M' \).

The idea for solving this problem is to shift \( b_{jk} \) to appropriate processors so that the multiplication of \( a_{ij} \) and \( b_{jk} \) can be performed in parallel. Then the element \( c_{ik} \) can be obtained by using the reduce(+) primitive. The binary representation of \( p \) is denoted by \( p_{q-1:0} \), which can be split into \( i = p_{q-1:m} \) and \( j = p_{m-1:0} \). The program segment for the matrix multiplication is as follows

```
begin
  L' := \max(L,N);
  L'' := \min(L,N);
  k(p) := p_{q-1:m};
  if 0 \leq k(p) < L' then
    for t := 1 to L' do {
      if 0 \leq k'(p) < L'' then {
        temp(p) := A(p) \times B(p);
        reduce(+) (0, m - 1, temp);
        if p_{m-1:0} = 0 then
          C[k(p)](p) := temp(p);
        k(p) := (k(p) + 1) mod L';
        if L > N then
          k'(p) := k(p);
        end;
      } else {
        reduce(+) (t, L', temp);
      }
    }
  end;
end;
```

The shift primitive is defined by

```
Procedure Shift (L', M': Natural;
  var A : MatrixType;
var newp : 0..n - 1;
begin
  newp(p) := [p + (L' - 1)M'] mod (L'M');
  A(newp(p)) := A(p);
end;
```

and reduce(+) is defined by

```
Procedure Reduce(+)(s, e : Natural;
  var A : MatrixType)
var i : Natural;
  temp1 : MatrixType;
begin
  for i := s to e do {
    temp1(p(i)) := A(p), (p(i) = 1);
    A(p) := A(p) + temp1(p), (p, = 0);
  }
end;
```

613
The complexity of this algorithm is \( O(\max(L, N) \times \log(M)) \). If the number of processors is sufficiently large, i.e., \( q \geq [\log L] + [\log M] + [\log N] \), a much more efficient algorithm \( O(\log \max(L, M, N)) \) for matrix multiplication can be devised.

**Amorphous Collections**

An amorphous collection may be in either a pointer-based or a normalized representation. In the pointer-based representation, each node of a graph associated with the pointers to other nodes is allocated a PE. The normalized representation for amorphous collections are based on indexed vectors or hash tables depending upon how their nodes are identified (unique ordinal numbers or unique names). Two normalized forms for graphs are possible. The first one is a list-based representation to the normalized form using adjacency matrix of a graph; that is, we assign a distinct PE to each element of an adjacency matrix according to the logical indices of elements. A graph in the second normalized form can be regarded as a crystalline collection, and no explicit pointers are required.

Amorphous collections are characterized by their communication-oriented algorithms. For graph components [including trees], communication is usually restricted to those pairs of nodes having a pointer between them. A node in a list component, however, may communicate with different nodes in the same list; thus, parallel primitives for list components are different from those for other amorphous collections. Some useful primitives for amorphous collections are described below.

*ReversePtr*: It is used to reverse the direction of pointers in a graph, and is a “single phase, two PE’s” operation.

*AdvPtr*: It is used to advance the temporary pointers in a list to point from one node to another node in the same list. It is a “single phase, two PE’s” operation.

*Send*: This primitive will move data from one node to another via zero or more intermediate nodes for a set of nodes in a graph. It is a “multiple phases, random pattern” operation.

*Convert*: This primitive is used to transform a graph in pointer-based representation to the normalized form using adjacency matrix. It is a “single phase, two PE’s” operation.

For list components in pointer-based representation, we need some higher level primitives which use more than one type of basic primitives discussed before. They are:

*Reduce*: This primitive reduces a list in a binary tree fashion and returns a scalar value.

*Prefix/postfix*: This primitive computes in parallel all initial prefixes/postfixes of a list using the specified op.

**Normalize**: This will convert a list in pointer-based representation to a normalized representation. Naturally, a list must be enumerated first before being normalized.

As an example, consider the problem of comparing two lists, \( l_1 \) and \( l_2 \), to determine whether \( l_1 < l_2 \), \( l_1 = l_2 \), or \( l_1 > l_2 \). (The comparison is done in alphanumeric order.) These two lists are kept in pointer-based representation throughout the computation. This example is used to demonstrate the simultaneous processing of more than one list component. Let \( list \) denote the variable storing the elements of lists, and \( next \) denote the pointer used within lists. Suppose \( l_1 \) and \( l_2 \) are the id’s of the PE’s at the head of two lists, respectively. Initially, all the elements of these two lists are stored in distinct PE’s. The first step here is to connect the corresponding elements in these two lists by pointers called rival. The alphanumeric comparison (denoted by \( \circ \) ) is then performed by each PE after obtaining the data from its rival. The final comparison of the two lists is then computed and the result is stored at the head PE of each list. The program segment for this comparison is described in the following.

```
begin
  MatchLists (l1,l2,next,rival);
  if rival(p) \neq NIL then
    tmplist(rival(p)) \leftarrow list(p);
    ReduceList(\emptyset) (next, ResultList);
end;
```

Here, \( \text{MatchLists} \) matches up the corresponding elements of two lists, \( \text{ResultList} \) stores the result of the comparison of the corresponding elements, and \( \text{ReduceList} \) computes the final result of the comparison of these two lists. Before defining \( \text{MatchLists} \) and \( \text{ReduceList} \), we first introduce three basic primitives, namely, \( \text{ReversePtr}, \text{AdvPtr}, \) and \( \text{send} \). Let \( A \) denote the pointers in a list, and \( B \) be a variable of the same type. Primitive \( \text{ReversePtr} \) is defined as follows.

```
Procedure ReversePtr (A : PtrType; var B : PtrType);
begin
  if A(p) \neq NIL then
    B(A(p)) \leftarrow p;
end;
```

Thus, \( \text{ReversePtr} \) is used to establish a set of pointers in \( B \), which are the reverse of those in \( A \). For a list in pointer-based representation, it is frequently useful to have a temporary pointer in each node to point to another node (not the next one) in the same list. Let \( A \) be this temporary pointer. Initially, \( A \) in each node points to the next node. Then, at each step, \( A \) is updated by the value of \( A \) in the node to which the current node points using \( A \). \( \text{AdvPtr} \) primitive is used to advance \( A \) for this purpose, and is defined by
**MAPPING STRATEGIES**

Since abstract data types are represented in a distributed way on SIMD hypercube machines, the interactions among data elements play an important role in determining their performance, that is, the performance of an abstract data type may vary considerably depending on the mapping of data elements on to the physical processors. In this section we discuss the mapping strategies to reduce communication cost for various abstract data types. The rich communication structure of a hypercube interconnection network makes this mapping feasible.

The major factor in selecting a mapping for an abstraction is the communication pattern embedded in that abstraction; it illustrates the characteristics of both its normalized representation and the associated parallel primitives. The possible communication patterns can be classified into four categories, namely, neighbor communication, group communication, dynamic communication, and random communication. By neighbor communication, we mean that communication occurs only in the neighboring processors of a logical structure such as a graph. That is, only the pairs of processors which represent an edge may allow communication to occur between them. The communication pattern for graphs in pointer-based representation usually belongs to this type. For some abstract data types such as list and matrix, communication may occur within a group of data elements in a fixed binary tree pattern. Examples are the broadcast and reduce(op) operations for a matrix component. The above two categories of communication patterns can be considered as fixed communication. By dynamic communication, we mean the communication pattern which is regular, but subject to change after each step. One example is the communication pattern for the comparison of two lists in pointer-based representation. Random communication pattern means that no regular communication...
we investigate the parallel implementation of abstract data types on SIMD hypercube machines. In this paper we fine-grained parallel systems. Conventional distributed systems such as indexed vectors and hash tables. Parallel primitivess, which are constructed to facilitate parallel operations on an abstraction, are based on the characteristics of underlying architectures. Parallel primitives associated with the normalized representation usually make extensive use of communication for exchanging information among PE's, and thus a mapping strategy must be selected to reduce the communication overhead. Two major factors in selecting a mapping are the communication pattern and locality of data elements. A suitable mapping of the normalized representation onto the underlying network of PE's can greatly improve the performance of an abstract data type.

In summary, an abstract data type on SIMD machines must be in a normalized representation so that its distributed clients and servers can be merged to achieve high performance parallel programs. Parallel primitives must also be provided for facilitating the parallel processing of client requests. To improve the performance, a suitable mapping for the normalized representation of an abstraction should be used to reduce the communication cost.

REFERENCES