High Performance Massively Parallel Abstract Data Type Components

I-Ling Yen, Farokh Bastani, Taghreed Al-Marzoq, and Ernst Leiss

Department of Computer Science
University of Houston
Houston, TX 77204-3475

ABSTRACT

It is difficult to program massively parallel machines in a modular way without suffering from excessive loss of performance. We develop a new approach that merges information hiding clients and servers to achieve high communication bandwidth for transmitting requests and receiving responses. It uses multi-entry data structures, massive-state-transition interface operations, and a four-level decomposition approach to achieve both structured programming and information hiding within the ADT implementation.

I. Introduction

Massively parallel computer architectures which have become commercially available in the past few years have introduced a new twist in the problem of software development. The potential for dramatic performance speed-up as well as simplified programming makes these systems extremely attractive. However, the difficulty of programming these systems in a modular way has also become apparent. The concepts of modular programming, such as information hiding components [Par75] and levels of abstraction [Dij68], where programs are constructed using four basic information hiding components, namely, abstract data type, functional, interface, and control components, have a well-known side effect -- the deterioration in system performance [Par75]. Techniques which combine structured programming and performance improvement for conventional systems do not scale up to fine-grain parallel systems. Thus, it is imperative to develop methods for resolving the structure and performance clash for implementing information hiding components in massively parallel systems. Among the information hiding components, abstract data type (ADT) components play a most significant role in the development of large software systems, especially for data intensive applications. In this paper, we will address the problems and outline possible solutions for implementing ADT components on massively parallel computers.

In conventional systems, special techniques are used to achieve structured programming for large systems without sacrificing performance. One approach is to optimize sequences of requests. For example, context dependent transformations for sequential programs have been developed which merge component encapsulations in the execution image to mitigate the loss of performance; this frequently reduces \( O(n^k) \) program segments to \( O(n^{k-1}) \) segments, for \( k > 1 \) [Bas84]. Another approach is used for implementing ADTs when components are encapsulated within different processes, namely, multilevel data structures [Bas88a] which are structured at different levels and data structure integration is performed during idle time to decrease the response time without affecting the functionality of the component.

For parallel systems, Gehani [Geh84] has proposed the broadcasting sequential processors (BSP) concept. Here, an ADT can be modeled as a server and clients access the abstraction via an interface processor. In addition to the BSP approach, VLSI technology can be used to implement ADTs directly in integrated circuits [Gui82]. Both approaches model the system as a hierarchy of clients and servers. They provide fast response time since a separate processor is dedicated to each element of the data structure. When a request arrives, all the elements of the abstraction are operated upon simultaneously, providing a parallel implementation of each individual operation to yield a constant response time for most requests. However, for multiple requests, these implementations face a potential bottleneck due to their one-request-at-a-time nature [Bas88b].

In a massively parallel environment, the client functions are also implemented using parallel algorithms; hence a massive number of requests will be issued in parallel for an ADT. Thus, conventional methods which are geared toward efficient processing of individual requests instead of a group of parallel requests can not be used. Fundamental changes in the ADT implementation techniques are required. In [Bas88b], a method of merging clients and servers is proposed, where parallel requests are handled simultaneously to avert the potential bottleneck under massively parallel environments. To facilitate such parallel handling of multiple requests, different kinds of data structures and interface operations should be considered.

Another problem with the implementation of massively parallel ADTs is the potentially high complexity within the abstraction. To facilitate the efficient implementation of various client functions, several choices of different mapping functions and various data structure representations have to be provided. Also, current parallel architectures are based on different computation models, memory models, and interconnection networks, unlike the sequential ones which are all fundamentally von Neumann architectures with relatively minor differences. Thus, further encapsulation of architecture dependent details, such as mapping, are essential for implementing high quality parallel software components. These problems invite the idea of splitting...
an ADT implementation into several layers, each encapsulating some specific type of information. A beneficial side effect of doing so is to provide a systematic decomposition paradigm for implementing ADT in a parallel environment.

In Section II, we will introduce a four-level ADT implementation semantics which provides a paradigm for designing potentially complex massively parallel ADTs in a modular way. In Section III, we will present new design concepts which facilitate the efficient handling of parallel client requests for massively parallel ADT implementation, including appropriate interface operations and data structures. Also, three classes of data types are identified to provide a systematic handle for developing various data types. Section IV discusses the ADT design issues and examples for each class of data types. Finally, Section V concludes the paper.

II. Information Hiding Layers in Massively Parallel ADTs

In conventional systems, the state of an ADT can be represented by a tuple \( S = (D, I, R) \), where \( D \) denotes a collection of data elements, \( I \) denotes the information associated with the data elements, and \( R \) denotes the relation among these data elements. In a massively parallel system, the state of an ADT has to be represented by a new tuple \( S = (D, I, R, D_L, R_L, PE, Conn) \), where \( D, I, R \) represent the same sets as for the conventional ADT, \( D_L \) denotes the collection of data element entries in the logical level, \( R_L \) denotes the relations among logical data elements, \( PE \) represents the set of PEs (processing elements) participating in the implementation of the ADT, and \( Conn \) denotes the relations (i.e., connections) among the PEs. Figure 1 shows the structure of a layered implementation of massively parallel ADTs. Figures 2 and 3 take the set ADT as an example and compare the conventional and massively parallel implementations. A hash table is used as the logical data structure in this example and the physical layer contains a hypercube interconnection network.

\[ S = (D, I, R, D_L, R_L, PE, Conn) \]

Figure 1. Four information hiding layers for ADT implementation.

The four information hiding layers are defined in the following.

**Aggregate Level.** This is the highest and most stable level; it deals with operations which involve more than one ADT. The state representation will also involve \( n > 1 \) abstractions, \((D, I, R), 1 \leq i \leq n\). A set of O-functions, which cause transitions in the state of the ADT, will be provided by the aggregate level that will cause state transitions on these abstractions and then spread to their lower levels.

**Abstract Level.** The representation of the abstract level is similar to the conventional one where the state is represented by \((D, I, R)\). A set of O-functions will be provided that will cause state transitions on \((D, I, R)\), which will then result in state transitions on \((D_L, R_L)\) and \((PE, Conn)\). References to the state of \((D, I, R)\) will be handled by a set of V-functions.

**Logical Level.** For each ADT, we can identify one or more suitable logical data structure representations for implementing it. The state of the logical data structure is expressed by \((D_L, R_L)\). The logical level will provide a set of O-functions which cause state transitions on \((D_L, R_L)\), which then result in state transitions on \((PE, Conn)\).

**Physical Level.** In order to achieve parallelism, data elements have to be distributed over processors. A data distribution representation can be defined by a mapping function \( M : (D_L, R_L) \rightarrow (PE, Conn) \) or \( M : (e, r_i) \rightarrow p_{fj}, 1 \leq i \leq n, 1 \leq j \leq n_p \), where \( n \) is the number of data elements in \( D_L \) and \( n_p \) is the number of PEs in \( PE \). One or more mapping functions may be provided for an ADT. A set of O-functions is provided which cause state transitions on \((PE, Conn)\).

In this four-level semantics, each level of the ADT encapsulates manageable information where the stability of the ADT toward performance enhancement or architecture changes increases while the potential complexity reduces.

III. Massively Parallel ADT Design Issues

To allow efficient handling of multiple requests, the logical data structure selected for implementing the ADT has to allow multiple accesses in parallel. Also, the interface operations provided by the ADT have to contain a massive number of client requests. Here we discuss the desired property of the logical data structure and interface operations for an ADT in massively parallel systems. Also, a classification of different data types is given to facilitate the development of systematic methods for the actual design of various ADTs.
3.1. Multiple Entry Data Structure

Let \( LD = (D_I, R_I) \) represent the logical data structure which implements some ADT, where \( D_I \) denotes the set of data elements in \( LD \) and \( R_I \) denotes the relation of each data element in \( D_I \) to other elements. Let 
\[
LD_{entry} = \{ d_I \mid d_I \in D_I \land 1 \leq k \leq |D_I| \}
\]
denote the set of entry points of \( LD \), where \( LD_{entry} = D_I \). Also, for each \( d_I \) in \( LD_{entry} \), let \( LD_{access} \) denote the set of data elements which can be accessed through entry point \( d_I \). A logical data structure \( LD \) is a multi-entry data structure if \( |D_I|/|LD_{entry}| \) is a small value which is close to one and \( \forall k, k' \mid k \neq k' \), \( |LD_{access}, d_I| - |LD_{access}, d_I'| \) is a small value which is close to zero. In a multi-entry data structure, all the entry points can be accessed at the same time, and each of them has to be accessed exclusively. A sorted array is an example of a multi-entry data structure while a tree is a counter example since the root of a tree is the only entry point for any access.

The most desirable multi-entry data structure is a complete multi-entry data structure in which every element is directly accessible. For example, a hash table with chaining is a multi-entry data structure since all the nodes directly in the hash table are entry points. However, it is not a complete multi-entry data structure since the nodes chained in the linked-lists are not directly accessible. On the other hand, the hash table with linear probing is a complete multi-entry data structure.

3.2. Massive-State-Transition Interface Operations

In conventional ADTs, the set of operations generally includes the operations of creating or disposing an ADT instance and those of adding or removing an element from an ADT instance. This type of operations fosters a view of an ADT instance as a structured collection of individual elements resulting in programs that manipulate bits instead of blocks. In parallel computations, a client function which accesses an ADT will generally issue a parallel access instruction, which corresponds to a massive number of conventional requests. A model based on incremental state changes cannot be used to process multiple requests efficiently. Also, if we have massive state changes, there is the possibility of conflicting requests. Thus, the interface operations should cause a massive number of parallel state transitions, i.e., operations which operate on two entire instances of the ADT and generate a new object. This type of operations is usually very costly in sequential or coarse-grain parallel systems, but is elegant and efficient for massively parallel environments.

Consider the set as an example. Elements in an instance of a conventional set ADT are added or deleted one by one through interface operations such as add_element and delete_element. However, in massively parallel systems, binary operations such as union and difference, which cause a massive number of state transitions, are more suitable.

3.3. Classes of ADTs

To get a handle on designing different types of ADTs, we can classify them into different categories. In [Bas88b], three major classes of ADT are identified. Since there are many similarities among data structures in the same category, we can provide systematic methods in formalizing strategies for selecting interface operations, selecting data structure representations and mapping functions, for specification, and so on.

As we defined in Section II, the three abstract level entities \( (D_I, R_I) \) span the external view of an ADT. We can map this tuple \((D_I, R_I)\) to a corresponding data structure graph \( G = (V, E) \). The vertices \( V \) are derived from \( D_I \), while the edges \( E \) are derived from the structural relation \( R_I \). According to the pattern of the structural graph, we can partition the class of all data structures into three subclasses, namely, unrelated, amorphous, and crystalline structures.

Unrelated collection. If for all possible states \((D_I, R_I)\) of an ADT, the corresponding structure graph \((V, E)\) is a separate component, i.e. \( E = \emptyset \), then the data type is an unrelated collection. The set of structural relations \( R \) of an unrelated collection is also an empty set. Examples of ADTs in this class are set, bag, search table, etc.

Crystalline structure. A data type is a crystalline structure if the corresponding data structure graph, \( G \), is crystalline-like for all of its possible states, \((D_I, R_I)\). Mesh, torus and hypercube are examples of crystalline-like graphs. Examples of crystalline ADTs are list, vector, matrix, etc.

Amorphous structure. If a data type is not an unrelated or a crystalline structure, then it is an amorphous structure. Examples of amorphous ADTs are tree, graph, semantic network, etc.

IV. Design Considerations for each Class of ADTs

In this section, we discuss each class of ADTs layer by layer, proceeding from general design considerations to the implementation details of a typical member of that class.

4.1. Design of ADTs with Unrelated Structure

4.1.1. Interface Operations for the Abstract Level. As discussed in Section 3.2, the O-functions and V-functions for an ADT in massively parallel systems should be operations which operate on entire instances of the ADT. For all classes of data types, an operation which creates an entire instance of a data type at one step and another which disposes an entire instance are required. Besides these operations, binary operations which operate on two entire instances should also be provided.

For the ADT set, we need O-functions such as create, dispose, and binary operations. For V-functions, operations which access the state of the entire set are appropriate. These interface operations are specified in the following. They can be compared with single transition operations such as create an empty set, add an element to a set, remove an element from a set, etc., that are usually provided for a conventional ADT set.

O-functions:

- create: list → set;
- dispose: set → ;
- union: set × set → set;
- empty: set → boolean;

V-functions:

- subset: set × set → boolean;
- get: set → list;
- difference: set × set → set;

Note that both create and get involve a massive number of data elements passed as parameter. The list data structure is used for this purpose. This will be discussed in detail next.

4.1.2. Aggregate Level Structures and Interface Operations. Client functions that access an ADT must manipulate a massive number of data elements when they invoke these operations. This requires some data structure for the client function to manipulate these data elements. One way is to provide a client-visible representation, i.e., the aggregate structure, for the data type. A
reasonable choice of such client-visible data representation is the list data structure where all data elements to be passed to the interface operations are simply stored in a list.

A set of O-functions can be provided on the selected aggregate data structure by the aggregate level to achieve some desired functions. Some useful interface operations for list are apply-to-all, reduction, and type transformation operations.

The operations for the set are specified in the following.

apply_to_all: set × operation → set;
reduction: set × operation → element;
set/list: set → list; :: transform set to list
list/set: list → set; :: transform list to set

4.1.3. Logical Level Data Structures and Primitives. The logical level encapsulates the logical data structures and provides primitives which allow the higher levels, such as abstract and aggregate levels, to access the data structures. For each ADT, we can identify one or more suitable logical data structure representation (D1, R1), (D2, R2), . . . , to implement it. Different logical data structures can be selected for efficient implementation of different client functions. We have to provide a unified set of logical level primitives for different choices of data representation to achieve better information hiding.

For our set ADT example, we choose the hash table representation to implement it. The hash table is used globally to contain all elements of all sets involved in the client functions. Each instance (i.e., each set) is represented by a Boolean vector corresponding to the global hash table.

The operations provided at the logical level must be augmented by primitives that facilitate efficient search and reduction. Also, for ADTs with unrelated structure, there should be alignment operations to position instances in order to reduce the communication overhead for those binary functions which involve two instances. In addition, due to the empty structural relation for this class of data types, a normalization operation is required to normalize the random input into the selected logical level data structure. The operations for the set ADT example are specified in the following.

retrieve: bit vector → list;
normalize: list → bit vector;

Since the normalized Boolean vector corresponds to a global hash table, any two instances are always aligned. Thus, there is no extra requirement for alignment operations.

4.1.4. Physical Level Primitives. The lowest level of the structured implementation of massively parallel ADTs is the physical level which encapsulates the mapping functions, communication procedures, and other system dependent details. Not only for various architectures, but also for different client functions or ADT operations, different mapping functions, communication strategies, etc., may be required to achieve high performance implementation. Thus, several mapping functions, communication procedures, and so on, may be encapsulated in the physical layer and a unified set of primitives should be provided to achieve information hiding. Some physical level operations which are common for all classes of data types have been identified in (Bas88b).

For data types with unrelated structure, we need nonuniform communication procedures to handle the normalization and alignment operations on the randomly organized data elements. Interface operations, such as send and receive, for point to point communication are needed. For the set ADT example, the set of primitives which should be provided by the physical level are,

send: element × queue → queue;
receive: queue → element ∪ null;
copy: bit_vector × bit_vector → bit_vector;

The operation send causes a processor to send a message to the message queue of the destination processor, and receive returns the first message removed from the queue if it is nonempty, null otherwise. The operation copy copies one set of data elements from their original PEs to the PEs which hold the other set of data elements. These three physical level primitives are standard and can be provided by the underlying system.

4.1.5. Sample Code for the Set ADT Implementation.

function normalize (ele: list) : bit_vector;
begin
parallel for i := 0 to n-1 do
bitvec[i] := 0;
if (i < length(ele)) then loc := h (ele[i]); send (ele[i], loc); endif;
while (x := receive ≠ null) do
if hashbl.occupied[i] then
if (ele) then bitvec[i] := 1; else send (x, (i+1) mod n); endif;
else hashbl.occupied[i] := true; hashbl.occupied[i] := ele[i]; bitvec[i] := 1;
endif;
endwhile;
endparallel;
return (bitvec);
end.

There are n (0 through n-1) processors involved in this computation where each processor represents an entry in the hash table (hashbl). The first function (f=length(ele)) processors hold the input elements and they will compute the hash value for those elements. Each element is sent to its corresponding location in the hash table where it is processed further, either rehashed or put into the hash table or identified as an existing element. When the correct position of an element in the hash table is determined, the boolean vector bitvec is updated by setting the corresponding bit. This vector represents the set and is returned after the computation.

function union (set1, set2: set) : set;
begin
set1 := copy (set1, set2);
parallel for i := 0 to n-1 do set3[i] := set1[i] or set2[i]; endparallel;
return (set3);
end.

4.1.6. Some Examples of Client Functions. The primitives provided at the logical and physical levels have been used in the code of the functions at the higher levels. Here, we will give some typical client functions to illustrate the use of the interface operations provided at the abstract and aggregate levels.

function intersection (set1, set2: set) : set;
begin return (difference (set1, difference (set1, set2)));
end;

function check_spelling (wlist: word_list) : word_list;
begin return (set3 (difference (listset(wlist), dictionary)));
end.

All the mathematical operations for set such as intersection, complement, etc., can be implemented using union and difference operations. The check_spelling function transforms the words in a list into a word-set and computes the difference between the word-set and the dictionary to identify misspelled words. It illus-
trate that in massively parallel systems, using type transformation can yield code that is efficient and simple.

4.2. Design of ADTs with Crystalline Structure

4.2.1. Interface Operation for Abstract Level. Consider the ADT matrix. Operations for creating and disposing entire instances are required. Moreover, we need operations which create a matrix out of existing matrices or by accepting all its elements as input. Also, interface functions which allow binary operations among corresponding elements of two matrices should be provided. The set of O-functions and V-functions for matrix abstraction is given in the following.

O-functions:
- create: range x 2d_list -> matrix;
- create: range x int_function -> matrix;
- dispose: matrix -> ;
- append: matrix x matrix x [row, column] -> matrix;
- extract: matrix x range -> matrix;
- matrix_op: matrix x matrix x operation -> matrix;
- permute: matrix x function -> matrix;

V-functions:
- get_matrix: matrix -> 2d_list;
- get_range: matrix x [row, column, both] -> range;

4.2.2. Aggregate Level Structures and Interface Operations. Since 2d_list (2-dimensional list or matrix itself) and list are chosen as the standard aggregate level representation, there is no need for matrix or list abstractions to have a specific aggregate level. However, if some specific type transformation operations are required, they can be placed in this level. Also, standard operations, such as apply_to_all and reduction, can be provided. For other data types in this class, the general design for aggregate level will be same as that for unrelated structures.

4.2.3. Logical Level Data Structure and Primitives.

For the matrix abstraction, we use a two dimensional list (2d_list) as the logical level data structure. The logical level primitives provided are as follows.

regular_shift: 2d_list x integer x direction -> 2d_list;
irregular_shift: 2d_list x function -> 2d_list;

Operation regular_shift shifts elements with a fixed displacement and direction. Any other shifts can be performed by irregular_shift. Operations for normalization or alignment are not necessary since they can be efficiently implemented by these shift functions.

4.2.4. Physical Level Primitives. Since data elements are highly organized in the crystalline structure, uniform communication primitives are required in addition to point-to-point communication primitives. Besides operations send and receive, primitives which encapsulate the implementation of efficient data movement according to the underlying interconnection networks should be provided.

For the ADT matrix, the following are the physical level primitives besides the standard send, receive, and copy primitives.

shift_left: matrix x distance -> matrix;
shift_right: matrix x distance -> matrix;
shift_up: matrix x distance -> matrix;
shift_down: matrix x distance -> matrix;

4.2.5. Sample Code for the Matrix ADT Implementation.

```c
function permute (M: matrix; mvfunc: function) : matrix;
function determine_shift (mvfunc: function; shift_type: shift_type; disp: integer; direction: direction); // Check if the move function is regular or irregular.
end;

begin
determine_shift (mvfunc, shift_type, disp, direction);
end;
```

The logical level function regular_shift will invoke the physical level shift functions according to the desired direction to perform uniform data movement. The primitive irregular_shift will compute the destination positions according to the source positions and the move function (mvfunc), and use send and receive primitives to perform the movement.

4.2.6. Example of a Client Function.

```c
function matrix_multiplication (A, B: matrix; l.m.n: positive) : matrix;
begin
T := create (1..l, 1..m, 1..n); M := append (M, T);
end;
```

4.3. Design of ADTs with Amorphous Structure

4.3.1. Abstract Level Interface Operations. As in the case of other classes, operations for creating and disposing ADT instances are required. Most client functions for this class of data types require operations to propagate along the relation between pairs of data elements. Thus, efficient and expressive primitives for data movement along the structural links are necessary.

For the graph ADT, we can provide parallel operations which create and dispose a graph. Operations for creating a graph from existing graphs or by specifying vertices and edges are provided. Also, as stated above, operations for data movement along the relations are required. The following are the specification of O-functions and V-functions for the graph abstraction.

This set of O-functions can be compared with the conventional one which includes creating empty graph, disposing of a graph, adding or removing a vertex or an edge of a graph.

```c
O-functions:
create: bit_matrix -> graph;
dispose: graph -> ;
append: graph x graph -> graph;
extract: graph x mask -> graph;
union: graph x graph -> graph;
```

200
difference: graph × graph → graph;
V-functions:
propagate: graph × mask × info_list → ;
accept: graph × operation → info_list;
get_vertices: graph → vertices;
um_vertices: graph → natural;

4.3.2. Aggregate Level Structures and Interface Operations.
For amorphous structures, a client function may have to pass all vertices, or all edges, or all information of an entire instance to an abstract level operation. A list can be the aggregate structure when all vertices or edges are required. When information for an entire instance is required, i.e., all vertices and edges have to be provided, we can have either a 2-dimensional list representation, i.e. the adjacency matrix, or a list representation where each data element and its structural relation to other elements are organized into one entry of the list.

For our graph abstraction, we choose to have only the adjacency matrix as the aggregate structure. Also, we use list as the aggregate structure for the client view of the set of vertices in the graph. Besides the standard operations apply to all and reduction, we provide the following transformation operations to let clients conveniently transform their representation to the matrix representation required by the abstraction.

adjacency_list/bit_matrix: adjacency_list → bit_matrix;
bit_matrix/adjacency_list: bit_matrix → adjacency_list;

4.3.3. Logical Level Data Structures and Primitives. Typical logical data structures chosen to implement data types of amorphous structure are the adjacency list and the adjacency matrix. A normalization primitive is needed to transform the client representation into the logical data structure. Since operations generally propagate through the structural relations, primitives which access the neighboring nodes through the data structure can be provided. Communication primitives which allow communication between neighboring nodes are necessary.

For our graph example, we choose the pointer-based representation as the logical level structure. The pointer-based representation is one way of implementing the adjacency list in massively parallel systems, where each data element is assigned to one processor; the structural relations are represented by pointers, associated with each element, which point to the related elements. Normalization primitives as well as communication primitives for all neighboring nodes are provided.

normalization: bit_matrix → pointer_based_structure;
send_to_neighbors: graph × vertex × info → ;

4.3.4. Physical Level Primitives. For amorphous structures, random communication is required to implement parallel functions. Similar to the ADT set, the point to point communication primitives send and receive as well as a copy operation are provided for the ADT graph. Since the mapping function for different choices of logical data structures can be different, we will provide interface primitives such that the higher level functions can choose the desired mapping functions. This allocation primitive is specified in the following.

allocate: pointer_based_structure → PE;

Due to space limitations, the sample code and example client functions for ADT graph are omitted here.

V. Conclusion
We have presented an approach for designing high performance ADT components for massively parallel systems without sacrificing information hiding. Our method begins with an organization of the data type into four layers, namely, the aggregate, abstract, logical, and physical levels. The aggregate level is at the highest layer and provides a schematic representation of the data type along with access and type transformation operations. The abstract level provides a collection of type-specific O- and V-functions that allow efficient manipulation of the data type. These are reminiscent of conventional operations, except that they are "massive transition" operations rather than conventional "single transition" operations. The logical level encapsulates one or more data structures used to implement the data type. These are multiple entry data structures that allow requests to originate at several points within the data structure to effectively eliminate congestion due to communication bottlenecks. The physical level is at the lowest level and encapsulates the mapping functions and interprocess communication details. The operations at the logical and physical levels are visible only within the ADT and are designed to implement efficient operations at the higher levels. The operations at the aggregate and the abstract levels are designed to facilitate efficient client level programs.

To facilitate the systematic design of various ADTs, we have classified them into three classes, namely, unrelated, crystalline, and amorphous collections. We presented general design decisions for each layer of each class of ADT and illustrated the theory with a detailed example from each class.

Future work in this area will include the formal specification of massive transition operations, the analysis of the trade-off between storing a data element locally on one processor versus a distributed representation, and the development of rules for transforming a user level code into an equivalent but more efficient program that directly accesses lower level operations.

BIBLIOGRAPHY