Department of
Computer Science

The VISA User's Guide

Matthew Haines and Wim Bohm

Technical Report CS-93-102

March 10, 1993

Colorado State University

The VISA User's Guide

Matthew Haines* Wim Böhm

Computer Science Department
Colorado State University
Fort Collins, CO 80523

March 8, 1993

Abstract

Programming distributed memory systems that lack a single addressing space remains a difficult task at best. The VISA system provides the user (or compiler) with a single addressing space that can be shared among the participating nodes. Access to the system is provided through a set of VISA primitives that control all aspects of the VISA space, from allocating memory to accessing data to freeing memory. This document explains how the VISA system is organized and how the primitives operate.

1 Introduction

Large-scale distributed memory multiprocessors represent the current state-of-the-art in high-performance architectures, yet software support for these complex machines still lags. It is often necessary for a programmer to explicitly specify the creation and distribution of parallel threads (i.e. code segments) using a sequential language that has been augmented with library calls for managing parallel threads of execution. And for distributed memory multiprocessors that lack hardware support for a single addressing space, the programmer must also divide the data structures among the distributed memories (called *data decomposition*), and provide the message passing necessary for sending and receiving remote values needed for local computation. Moreover,

*Supported in part by a grant from Sandia National Laboratories

to reduce unnecessary remote references, the programmer must match the distribution of data with the distribution of threads to maximize local references. This requires the ability to match the decomposition with the expected access pattern of the program, resulting in the need for general decompositions. Clearly this is a difficult task for even the most basic application, and, in an ideal world, should not be left to the programmer to solve.

Though most parallel programming languages do not provide support for data decompositions [10], there has been some recent effort to remedy this neglect. FortranD [6] and other Fortran extensions [1, 15] augment Fortran with statements that allow the programmer to specify a limited number of decompositions, and the compiler then uses this information to generate a data-parallel program with implicit message passing for sending and receiving remote values. While this approach has proved useful for many scientific applications, it suffers from three flaws that keep it from being widely utilized. First, this strategy is based on extensive data dependence analysis that requires very regular (i.e. predictable) computations and data structure references to be successful. Symbolic subscript terms with unknown values, coupled subscripts, and nonzero or nonunity coefficients of loop indices often make dependence analysis impossible for even the most sophisticated parallelizing compilers [14]. Second, this technique requires that data structure sizes and the number of processors to be used be known at compile time, which restricts the ability to run the application using varying parameters without re-compilation each time. Third, this approach requires a very intelligent compiler that is often not available on a given system, and when provided, forces the programmer to use a specific language. Thus this approach is not language-independent. Other parallel languages either provide similar support for decompositions that are used by the compiler to generate data parallel programs [7, 8, 12], and thus suffer the same problems as the FortranD approach, or lack the ability to control decomposition altogether [3].

Another approach to simplifying the problem of programming distributed memory machines is to utilize underlying support for a single addressing space. This is most commonly provided by either the operating system (often termed *distributed shared memory*) or the hardware. These DSM systems [2, 9, 11] provide a single addressing space to the compiler so that the programmer can code in a shared-memory fashion, leaving the details of data decomposition and message passing to the operating system. Decomposition occurs by dividing the data structures among the DSM *pages* that will be exchanged when attempting to access a page that is not local. Caching techniques are used to increase the availability of these pages, but this enhancement comes at the cost of expensive coherence protocols. The main disadvantage of these systems is the inability to control the decompositions so that the data can be matched to the threads. Another problem with this approach is *false sharing*, in which values from two different data structures are placed on the same page, and thus access to these values must be mutually exclusive although they are really independent of each other.

Our approach, which we call VISA, is similar to the DSM approach in that we provide a single addressing space to the compiler, but differs from the DSM approach in that we provide the

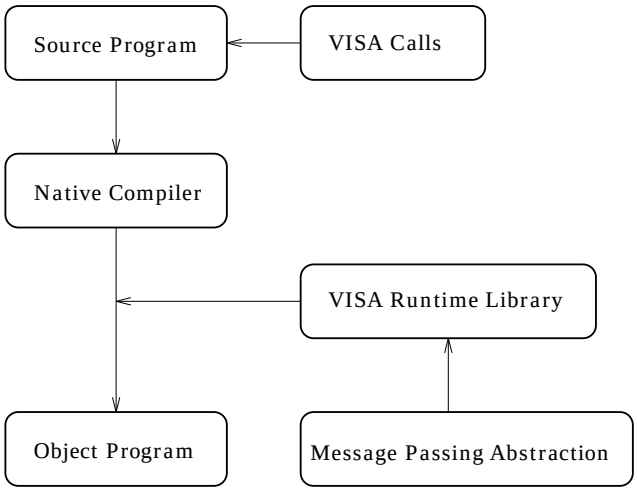

Figure 1: Overview of the VISA system

ability to control decompositions, similar to the language approach, and eliminate false sharing by reducing the granularity of the page to the size of each individual data structure. The result is a language-independent method for providing the programmer with a shared-memory paradigm while exposing the decomposition of data so that it may be altered to reduce remote references, thus improving performance. Currently, VISA is running atop the Vertex 3.0 operating system on the nCUBE/2 multiprocessor.

The remainder of the paper is organized as follows: Section 2 provides an overview of the VISA addressing space and the supporting system. Section 3 describes the data structures used to control and access the VISA space. Section 4 outlines the VISA interface functions, and Section 5 describes the currently-defined data mapping functions. We conclude and outline future enhancements in Section 6.

2 The VISA Single Addressing Space

VISA is a runtime system that provides a single addressing space and general data decomposition functions to the programmer or compiler. The interaction of the VISA system with the other language components is depicted in Figure 1. The programmer (or compiler) augments a parallel program with VISA primitives for allocating and referencing the data structures to be kept in the single addressing space. Local variable references are unaffected by the VISA system. The augmented program is then compiled using the native language compiler of choice, and linked with the VISA library to create the object program, which can then be executed on a distributed memory multiprocessor. All message passing for handling remote references is handled implicitly

by the VISA system through the use of a *message passing abstraction* that is included in the VISA library.

2.1 The Message Passing Abstraction

The VISA system relies on an underlying message passing abstraction to support both synchronous (blocking) and asynchronous (non-blocking) operations. Since these operations are provided by most host operating systems for distributed memory multiprocessors, VISA can be easily ported to another multiprocessors by simply modifying the message passing abstraction to make the proper native calls.

Specifically, the abstraction supports the following operations:

- *WriteMsg*, a **non-blocking send** abstraction, designed to provide point-to-point communication. We can use this primitive to build higher-level abstractions such as *broadcast* and *multicast*.
- *ReadMsg*, a **blocking receive** abstraction, designed to provide explicit synchronization when reading a remote value. Selective message screening can be accomplished using a message *key*, which is composed of a message type and sender/receiver designator.
- *MsgInterruptHandler*, an implicit **non-blocking receive** abstraction, designed to provide asynchronous message reception. Asynchronous message reception requires polling at some level to determine when a message arrives and take appropriate action. Most systems, including the nCUBE/2, provide hardware polling for incoming messages, resulting in a hardware interrupt that is caught by the operating system, and then passed into the user-level in the form of an interrupt, which can be caught by user-level programs. Therefore VISA traps the interrupt caused by an incoming message and invokes a routine called the *message interrupt handler* to deal with the incoming message. After taking appropriate action, the interrupt handler returns to the instruction that was executing when the interrupt occurred. If the interrupt handler is allowed to be invoked at any arbitrary time, it cannot be allowed to modify the global state of the computation. However, since VISA messages do modify the global state, such as in adding an element to a global structure, the interrupt handler must be selectively disabled during the times when these global structures are accessed. Thus, VISA provides a mechanism for enabling and disabling the interrupt handler so that a consistent global state can be maintained. This mechanism is provided through the BEGIN_CS (begin critical section) and END_CS (end critical section) primitives.

Built atop this message passing abstraction is the VISA system, providing a single addressing space and general data decompositions to ease the burden of distributed programming while

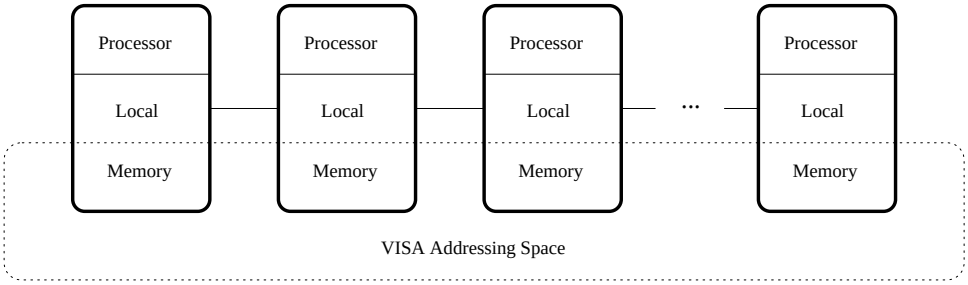

Figure 2: The VISA Addressing Space

at the same time provide explicit control over the distribution of data, which is necessary for achieving good performance over a variety of data access patterns.

2.2 The Single Addressing Space

One of the main goals of the VISA system is to provide a single addressing space over a set of distributed memories, so that compilers that assume a shared addressing space is available, such as the Sisal compiler, can be implemented on distributed memory multiprocessors. To create the shared addressing space, each node reserves a portion of local memory to be contributed to the VISA system, resulting in a distributed addressing space for storing shared data (see Figure 2). The VISA system then assigns each shared data structure a *virtual address* that corresponds to this addressing space. This, in turn, results in the data structure being divided among the memories, such that each memory "owns" a portion of the data structure. Ownership is necessary to implement the VISA update protocol, *owner-writes*, in which the ownership of a data structure element (or portion) is fixed, and only the owner is allowed to update the portion it owns. However, to increase availability of certain data structures, *replication* of any VISA data structure is supported, though the user is responsible for ensuring that all copies are consistent. This eliminates the need for implementing costly coherence protocols especially when implemented in software.

2.3 Data Distribution with Mapping Functions

Informally, the problem of data distribution (or decomposition) is to divide the data structures that a program uses among the memory elements so as to minimize certain desired measures, such as total execution time or number of remote references. For a shared memory multiprocessor, this becomes a trivial task, since the placement of a data structure does not (or should not) effect performance in a shared memory system (by definition). However, for non-shared memory

multiprocessors, such as distributed memory multiprocessors, data distribution becomes a more serious concern. There are two important points to be made about data distribution for a distributed memory multiprocessor:

1. The time required to access local memory is typically an order of magnitude less than the time to access remote memory. Therefore, if no attempt is made to tolerate latency, optimal execution time occurs only when all data references are local. However, this is clearly not possible. For example, it is possible that every processor will need every element of a data structure. If the data structure is to be distributed, then clearly some processors will not have local access to the elements they need. Another problem occurs when the reference pattern is unknown at the time of distributing the data. Also, some interconnection networks (e.g. ring) have faster access times to *neighboring* nodes than to distant nodes. For these non-uniform access machines, non-local data references should be on neighboring nodes as opposed to distant nodes. Again, this is not always possible.
2. The principle of *locality* states that memory references are grouped together in both space (*spatial locality*) and time (*temporal locality*). This implies that if we reference a particular data item, then there is a good probability that we will issue a reference for the same data item very soon (temporal locality), or we will reference another data item that is physically close to the original reference (spatial locality).

If we combine these points, then we have the outline for a data distribution scheme:

- Determine the access pattern.
- Distribute the data so as to maximize the local references.
- Distribute the non-local references so as to maximize the neighboring references (only if there is a discrepancy between neighboring and distant access latencies).
- If a reference is remote, then attempt to either make the reference local in the future, or try to make references to related items local in the future, or both.

This distribution scheme takes advantage of the observations that were made about the behavior of distributed memory multiprocessors and their programs, but does not address the *feasibility* of the approach. Of the assumptions made, the ability to determine access patterns is by far the most idealistic. This is reflected in the current alternative methods being used for distributing data structures:

- The compiler controls the distribution of data structures. This is the approach taken by the parallelizing compiler camp [4, 15]. The basic idea is to distribute the data structures

according to some *distribution function*, and then to analyze the array subscripts to determine whether or not, for a particular thread, a given reference is local or remote. If the reference is remote, the the appropriate communication primitives are generated to retrieve the value at runtime. The distribution functions are formalized so that a compiler can make sense of them, and this formalization is equally useful when considering other approaches.

- The compiler controls the distribution with the help of the programmer. This approach is an extension to the compiler controlled approach in that the programmer helps the compiler in identifying the data access patterns by the use of *pragmas*, which are source level compiler directives. Since the programmer may have a better idea as to how the data will be accessed [6], most compilers that perform the data distribution for the programmer will accept these "hints" so that the proper data distribution function can be selected.

- The compiler controls the distribution with the help of run-time profiles [13]. Again, this approach attempts to help the automated distribution process, but rather than have the programmer tell the compiler how the data will be accessed, the compiler simply "watches" several characteristic runs and notes the distribution patterns used for those runs. The compiler then selects a distribution function that will come closest to this observed reference behavior. The advantage this approach has over the pragmas is that the programmer may be unaware of the reference pattern, and thus be unable to help with the distribution. The disadvantage is that if the profiled runs are not characteristic of the actual reference patterns, or if the reference patterns vary with the input data, then this approach may be misleading.

- The programmer controls the distribution explicitly. Since all of the above techniques require intelligent compilers that are not always (or often) available, a common technique for distributing data is for the programmer to explicitly distribute the data and then insert the appropriate communication primitives into the source code, all "by hand." Though this approach requires very little software support (only the message passing interface is needed), the user is required to determine the access patterns and then distribute the data accordingly using explicit message passing primitives. Clearly this contradicts the efforts of raising programming to a higher level of abstraction.

The VISA approach to this complex problem is to provide a comprehensive set of mapping functions that are representative of common scientific data access patterns, and allow for the user to create new mapping functions as needed. The mapping function is then specified upon requesting memory from the single addressing space using the *visa_malloc* function. This allows a compiler that is generating the VISA primitives to invoke *visa_malloc* with the desired mapping function, either obtained from analysis or through user directives. Likewise, a programmer using the VISA primitives directly can select the desired mapping function for each data structure without having to specify the actual message passing details necessary for implementing such a distribution scheme. A complete list of the currently defined mapping functions is given in Section 5.

Field	Function
low_range	The base (lowest) address for this data structure
high_range	The highest address for this data structure
local_base	The offset to the desired element
nelems	The number of elements in this data structure
size	The size of each element in this data structure
blocksize	The blocksize (elements per block) used for distribution
start_node	The node ID on which to begin distributing the blocks
stride	The stride at which to distribute the blocks
replicate	A boolean to determine if this data structure is replicated
next	A pointer to the next entry in the table

Table 1: Description of a `range_map` entry

3 VISA Data Structures

3.1 The Range Map

In order to support general data decompositions, each VISA data structure must have associated information to enable address translation "on-the-fly", which is needed in the general case when data structure sizes and processor configurations are not known at compile time. Thus, at the heart of the VISA addressing system is the `range_map` data structure, which stores the information necessary to perform an address translation. Table 1 depicts the fields of a `range_map` entry.

We call it a `range_map` because we associate only one entry for each VISA data structure, which we can do since every VISA data structure is allocated a contiguous segment of the VISA space, so all addresses falling within a specified range (`low_range` ... `high_range`) necessarily belong to a specific data structure. This allows us to store information about each data structure rather than having to store information about each VISA address. We now describe how these fields are used in the address translation process.

3.2 Address Translation

In theory, the VISA addressing space is a single addressing space, similar to a shared memory addressing space for a workstation. However, in reality, just as the workstation hardware must translate each address to determine which memory chip and offset the address is referring to, the

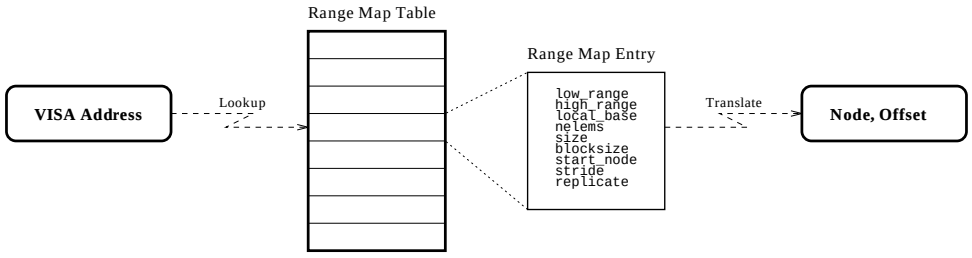

Figure 3: VISA Address Translation Diagram

VISA system must translate each VISA address into a corresponding processor memory and local offset. Central to the address translation process (depicted in Figure 3) is the concept of *blocksize*, which defines the granularity (in terms of data structure elements) of the data decomposition. Each VISA data structure is divided into blocks of size `blocksize`, and distributed among the nodes, starting with `start_node` and continuing with a stride of `stride` until all of the blocks have been distributed. By varying the *control parameters* (`blocksize`, `start_node`, and `stride`), we can implement various mapping functions, as further explained in Section 5.

Once the data structure blocks have been distributed among the nodes, address translation can proceed as follows:

- Starting with *address*, which is the VISA address of the desired datum, we subtract the `low_range`, resulting in a relative element position:
  `element = address - low_range`.
- Next we compute which of the blocks contains the element we are interested in:
  `block = element / blocksize`.
- Next we calculate the offset within a block where the element is located:
  `block_offset = element mod blocksize`.
- Now we can compute the actual node ID that contains the block (and hence the element) that we desire. If the `replicate` flag is set, then the computed *node* is always the local node ID, indicating that each node has a copy of the desired block. Otherwise, we compute:
  `node = (start_node + (block * stride)) mod P`,
  where `P` is the number of nodes.
- If the number of blocks is greater than the number of nodes, then some nodes will have more than one block allocated to them. To determine which of the local blocks contains the element we are interested in, we compute:
  `node_block = block / P`.
- Now we compute the actual offset of the desired datum within the node:
  `offset = node_block * blocksize + block_offset`.

- Finally, if the computed `node` matches the local node ID, then we can adjust the `offset` to point to the actual datum in local memory by adding the `local_base` to the previously-computed `offset`:
  `offset += local_base`.

The translation process yields a `node` and `offset` for the given VISA address. If the computed `node` matches the local node ID, then the computed `offset` points to the actual position in local memory where the desired datum is located. However, if the computed `node` is not local, then the computed `offset` is not valid since it was computed using the wrong `local_base`. Therefore we send a message to the specified node (*send VISA_REQUEST*), requesting that the desired datum be retrieved from its local memory and sent back, where we will be waiting for it (*receive VISA_REPLY*).

The disadvantage of this translation process is that the control parameters need to be looked-up for each address translation, since the control parameters can vary for each VISA data structure. This can be an expensive process when the result is a local reference, and so we have implemented an optimization that avoids translation when the data structure is guaranteed to be local, and is discussed further in Section 4.

The alternative to this variable control parameter scheme is to have a *fixed* blocksize, start_pe, and stride for every data structure, then the address translation calculations can proceed directly from the information provided in the virtual address bits. We implemented this fixed addressing scheme as a VISA runtime option, and found that although translation proceeds at a faster rate, the fixed control parameters often causes mis-alignment with the parallel loops accessing the data structures, resulting in excessive remote references and severely degraded performance [5]. Therefore we opt for the ability to avoid latency by providing flexible data distributions that can easily be matched to the access patterns of the parallel loops. This requires that each data structure have its own set of control parameters, and that these parameters be fetched for address translation.

4 VISA Access Functions

We now introduce the VISA access functions, which can be classified into two main categories: routines for allocating and deallocating VISA space and routines for accessing VISA space. We also introduce a special function designed to help avoid the address translation for variables that are always local.

4.1 Allocating and Deallocating VISA Space

- V_ADDRESS **visa_malloc** (int *nelems*, int *size*, map_function *map*, int *map_arg*)
  This function allocates a block of VISA space (*nelems* * *size* bytes), which will be distributed according to *map*, and returns a pointer to the start of the allocated space.

- void **visa_free** (V_ADDRESS *address*)
  This function returns the given portion of VISA space to the free pool.

4.2 Accessing the VISA Space

- char **visa_get_c** (V_ADDRESS *address*)
  This function returns the desired character value from the given VISA address.

- int **visa_get_i** (V_ADDRESS *address*)
  This function returns the desired integer value from the given VISA address.

- float **visa_get_f** (V_ADDRESS *address*)
  This function returns the desired floating-point value from the given VISA address.

- double **visa_get_d** (V_ADDRESS *address*)
  This function returns the desired double floating-point value the given VISA address.

- void **visa_get_m** (POINTER *data*, V_ADDRESS *address*, int *size*)
  This function copies the block of data starting at the given VISA address and for a length of *size* into the location pointed to by *data*.

- void **visa_put_c** (char *value*, V_ADDRESS *address*)
  This function places *value* into the given VISA address location.

- void **visa_put_i** (int *value*, V_ADDRESS *address*)
  This function places *value* into the given VISA address location.

- void **visa_put_f** (float *value*, V_ADDRESS *address*)
  This function places *value* into the given VISA address location.

- void **visa_put_d** (double *value*, V_ADDRESS *address*)
  This function places *value* into the given VISA address location.

- void **visa_put_m** (POINTER *data*, V_ADDRESS *address*, int *size*)
  This function copies the block of data pointed to by *data* into the given VISA address location.

- void **visa_update_c** (uchar *reduction*, char *value*, V_ADDRESS address)
  This function updates the value stored in the given VISA address with *value*. The *reduction* argument specifies how the update is to be performed, and the current values for *reduction* are *V_SUM* and *V_PRODUCT*.
- void **visa_update_i** (uchar *reduction*, int *value*, V_ADDRESS address)
  This function updates the value stored in the given VISA address with *value*.
- void **visa_update_f** (uchar *reduction*, float *value*, V_ADDRESS address)
  This function updates the value stored in the given VISA address with *value*.
- void **visa_update_d** (uchar *reduction*, double *value*, V_ADDRESS address)
  This function updates the value stored in the given VISA address with *value*.

4.3 Optimizing VISA Addresses Translation

In Section 3.2 we introduced the VISA address translation process, which consists mainly of finding the appropriate range map entry for a given address and performing a few calculations using the control parameters from the range map entry, resulting in a computed `node` and `offset`. For remote references, this translation time is small compared to the time required to fetch the remote value. However, for local references, this translation time is larger than the actual time required to perform a true local reference. Many times these remote references are *always* local, such an for replicated data structures or blocks of data that are entirely owned by a processor. When this is the case, we can replace the base VISA address to this data structure with the actual local offset of the data, so that future references to this VISA structure can occur without translation, and thus at about the same rate as a true local reference.

The local offset that replaces the VISA address must be tagged so that the VISA routines know that this is not a VISA address but the real local offset, and thus bypass the translation process. Currently this tag consists of setting the second high-order bit in the address, which places the actual offset outside of the possible range of VISA addresses.

To perform the optimization, the following conditions must hold:

- The runtime parameter for allowing optimization (**-vo**) has been set.
- The address has not already been optimized.
- The computed `node` for the address matches the local node ID and the given address is the base address for this structure (i.e. address = `low_range`).
- The entire structure is local (i.e. `blocksize == nelems`).

- The address is not going to be passed to another node.

When all of these conditions hold, it is safe to replace the VISA address with the actual computed `offset`. With the exception of the last, these conditions are checked by the **visa_optimize** routine, and if any fail, the original VISA address is returned as the result of the optimization. The last condition, that the address not be sent to another node, is enforced by the placement of the **visa_optimize** routine in the code.

- V_ADDRESS **visa_optimize** (V_ADDRESS *address*)
  This routine attempts to replace the given VISA address with the actual computed `offset` so that future references to this data structure can occur without translation.

5 VISA Data Mapping Functions

A data mapping function describes how a data structure is to be distributed among the participating memories. More formally, data mapping functions can be defined as follows: let D be a data structure with elements $e_1 \ldots e_n$, and M a distributed memory multiprocessor with processors $P_1 \ldots P_m$, each processor with a local address space A available for sharing and with local addresses $a_1 \ldots a_l$. We define a *mapping function* $\mathcal{F} : D \rightarrow M \times A$ for each element $e \in D$, as the set of (P_i, a_j) pairs that will receive a copy of e, such that $\forall e \in D,\ \mathcal{F}(e) \neq \emptyset$. That is, every element must be mapped to some local address of some processor. A distribution function $\mathcal{F}$ is *non-replicating* if $\forall e \in D,\ |\mathcal{F}(e)| = 1$. A distribution function $\mathcal{F}$ is *replicating* if $\exists e \in D,\ s.t.\ |\mathcal{F}(e)| > 1$.

The principle of *locality* states that memory references are grouped together in both space (*spatial locality*) and time (*temporal locality*). We attempt to exploit locality and minimize remote references by providing general mapping functions that can be used to align data references with the parallel loop slices that access the data.

Each data structure is allowed to have a different mapping function, since it is unlikely that all data structures in a program would benefit from the same mapping function. For example, control structures might be allocated using a *replicate_map* so that each processor has local access to the structures, an array might use a *block_map* mapping function so that the array is divided into blocks and distributed among the processors for efficient parallel access, and a shared counter would be implemented using a *scalar_map* to ensure consistency. Each of these maps is implemented by varying the *control parameters* discussed in Section 3, namely the `blocksize`, `stride`, `start_node`, and `replicate` values. Thus the job of every mapping function, which is called from the **visa_malloc** routine to set up the `range_map` entry, is to define how these

Mapping Function	Blocksize	Start PE	Stride	Replicate
scalar_map	n	map_arg	1	No
replicate_map	n	P_{id}	1	Yes
block_map	n/p	map_arg	1	No
interleave_map	1	map_arg	1	No

Table 2: Control Parameter Settings for Various 1D Mapping Functions

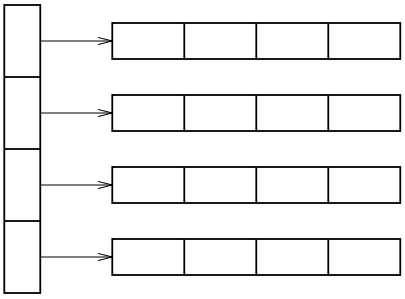

Figure 4: Two-Dimensional Array in Sisal

control parameters are to be established. Table 2 details these parameter settings for several one-dimensional mapping functions.

Two dimensional data structures in VISA (and Sisal) are provided as one dimensional pointer arrays, where each element of the array points to another one dimensional structure. For example, a two dimensional array is represented as an array of pointers to rows of data, as depicted in Figure 4. Mapping functions for two dimensional arrays must therefore consider both the array of pointers as well as the arrays of data. Since the pointer arrays are only written once (when initialized), they can be replicated to increase the availability of all rows to all processors. This guarantees that accessing any element of the matrix will generate at most one remote reference. So assuming that all pointer arrays are allocated using the *replicate_map*, Table 3 details how the control parameters are established for each of the data rows. The *map_arg* for these mapping functions is typically i, corresponding to the i^{th} row of the matrix. Thus for the *matrix_row_map* mapping function, the *map_arg* varies from 0 to p, so that each of the p processors gets a row of the matrix. The *rbs* variable for the *matrix_block_map* function represents the blocksize (or number of elements) in each of the blocks. Figure 5 depicts the *matrix_row_map* and *matrix_block_map* for an 8x8 matrix on 4 processors, where $rbs = 4$, the number of elements in each row of the blocks.

Though we have only presented a few possible mapping functions for one dimensional and two dimensional structures, it is possible to create many different mapping functions, given the ability to modify the data control parameters. This *general* approach to data distribution is necessary to

Mapping Function	Blocksize	Start PE	Stride	Replicate
matrix_row_map	n	map_arg	1	No
matrix_col_map	1	map_arg	1	Yes
matrix_block_map	$rbs * n/p$	map_arg/rbs	rbs	No

Table 3: Control Parameter Settings for Various two dimensional Mapping Functions

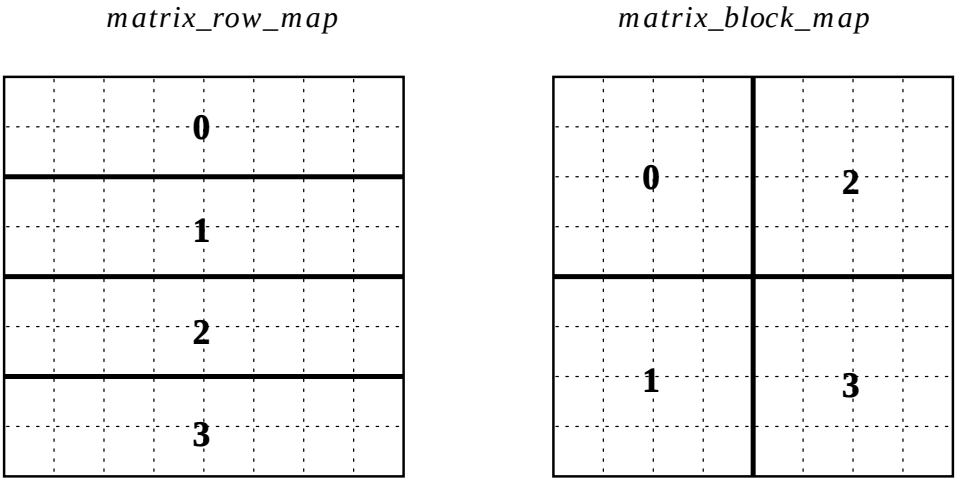

Figure 5: Row and Block Matrix Mapping Functions

accommodate the various reference patterns that applications exhibit, and VISA allows the user to add to the set of available mapping functions so that customized decompositions are possible.

6 Conclusion and Enhancements

We have introduced the VISA runtime system as a method for creating a single addressing space to raise the programmer (or compiler) from the details of a message passing architecture. The addressing space supports general data decomposition functions so that the distribution of data can be matched to the access patterns of a program so that remote references may be minimized. This paper outlines the organization and implementation of the VISA system, and provides information about the VISA primitives necessary to create and access the VISA space.

VISA started as a method for creating a language-independent and machine-independent approach to providing a single addressing space that so many compilers and programmers take for granted. Specifically, we created the VISA system to provide for a distributed memory implementation of the Sisal programming language, whose compiler assumes a shared memory abstraction exists for storing user data structures. Therefore VISA currently supports Sisal, as well as C, on the nCUBE/2 distributed memory multiprocessor. We are also working, or considering work, to enhance the VISA system as follows:

- We are currently working with people at the Lawrence Livermore National Laboratory to create a NUMA-based version of VISA, where NUMA stands for Non-Uniform Memory Access machines. These are machines that provide a single addressing space, but whose memories are physically distributed and so the access time to various portions of the memory can vary. Examples of NUMA machines include the BBN TC2000 and the KSR-1. Since data decomposition is still important in such as architecture, we wish to use the VISA system to control the data distribution rather than provide a single addressing space. Thus mapping functions for such an implementation would specify in which type of memory the data structure is to be stored. A NUMA implementation also should take advantage of the ability to pass around actual memory addresses rather than data in the message passing substrate.

- Compiler support for the VISA system is necessary for increased performance and further abstracting the details of an architecture from a programmer. The compiler can help the **visa_optimize** primitive by relaxing the constraint that the entire structure must be local. That constraint exists because the base is optimized and the entire structure is not local, then accessing part of the non-local area would result in a memory violation. Compiler analysis could determine that this would actually not happen even though the structure is not held locally, and the optimization could proceed. Compiler support for helping

to analyze the access patterns using sophisticated dependence analysis techniques would alleviate the need for programmers having to specify the mapping functions, while allowing the user the ability to override the compiler decision.

- We are also considering different language interfaces, such as object-oriented and different machine bases, such as other DMMP and NUMA machines for the future of VISA.

References

[1] V. Balasundaram, G. Fox, K. Kennedy, and U. Kremer. An interactive environment for data partitioning and distribution. In *Distributed Memory Computing Conference*, Charleston, SC, April 1990.

[2] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors. Technical Report Rice COMP TR89-91, Rice University, April 1989.

[3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[4] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990.

[5] Matthew Haines and Wim Böhm. On the design of distributed memory sisal. Technical Report CS-92-144, Colorado State University, Fort Collins, CO, January 1992.

[6] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[7] C. Koelbel and P. Mehrotra. Compiling global name-Space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Computing*, 2(4):440–451, October 1991.

[8] J. Li and M. Chen. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.

[9] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[10] Cherri M. Pancake and Donna Bergmark. Do parallel languages respond to the needs of scientific programmers. *IEEE Computer*, 23(12):13–24, December 1990.

[11] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Unifying synchronization and data transfer in maintaining coherence of distributed shared memory. Technical Report GIT-CS-88/23, Georgia Institute of Technology, June 1988.

[12] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *ACM SIGPLAN*, Portland, OR, June 1989.

[13] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* The MIT Press, 1989. Research Monographs in Parallel and Distributed Computing.

[14] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An emperical study of Fortran programs for paralleliz-ing compilers. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):356–364, July 1990.

[15] H. Zima, H. Bast, and M. Gerndt. Superb: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1986.