|epcc|

THE UNIVERSITY OF EDINBURGH

EPCC-SS97-06

MPI Datatypes Toolset: Library Support

Berent Eskikaya

Abstract

Creating MPI [1] derived datatypes that correspond to sections of data sets can be a laborious task, and requires a detailed knowledge of memory layout. This project seeks to provide library support for creating MPI derived datatypes that match the commonly-seen data substructures in parallel programs. C and Fortran are both supported. The library provides a high-level interface that shields the user from memory layout concerns.

This library can be obtained from ????.

Contents

1 Introduction

The Message Passing Interface (MPI) [1] provides parallel programmers with a library that implements an inter-process communication capability. One area of difficulty in using MPI involves the transfer of compound data structures and substructures [3]. Since MPI is a library, programmers wishing to transfer compound data have to explicitly describe the memory layout of their data structures to MPI. This places the extra burden of having to know the low-level details of how the compound data structures are arranged in memory on the user. A typical example is that the user needs to be concerned with the stride between the memory addresses of the basic types that comprise the compound datatypes. This is an error-prone task. It is desirable to isolate the user from these low-level details and provide an interface that allows her/him to construct MPI derived datatypes that specify the needed compound structures or substructures.

2 Background

The 1996 Summer Scholarship project *MPI Datatypes Toolset* [2] involved the development of a parser-level tool that recognises marked data structures in the user's program and generates the necessary code for the construction of the matching MPI derived datatypes. This relieves programmers of the burden of having to describe to MPI the memory layout of their C `structs` and Fortran `COMMON` blocks. Also addressed in this project was library support for transferring pointer-based data structures such as trees or lists. Such structures require special consideration, since pointer values on one processor are not meaningful on another, and extra information has to be transferred so that the original structure of the data can be reconstructed at the receiving end.

There has also been some work in providing library support for the construction of compound data substructures, such as subarrays within arrays. The MPI-2 standard [4] defines a new interface that creates an MPI datatype specifying an n-dimensional subarray of an n-dimensional array. It is also possible to implement interfaces that allow the user to construct template subarrays from collapsing dimensions, or define the halo areas – with required depth – of a given array. Such functionalities are commonly needed in parallel programming; however, they require an intimate knowledge of the memory layout of datatypes, and are therefore burdensome for most programmers. This project aims to address the need to hide these complexities behind a library interface.

3 Motivation

This section demonstrates the complexities associated with creating substructures of compound datatypes. Figure 1 illustrates the creation of a column of a two-dimensional array. It is possible to define a single column using a datatype constructor such as `MPI_Type_vector`. However, attempting to use multiple array columns will result in incorrect memory access, as the next item of the "column" datatype begins at point B, thus exceeding the array bounds. The user has to explicitly re-adjust the upper bound of the "column" datatype, and construct a new datatype, in order to enable the next "column" to be found at point A. The code needed for this task is given below:

```
int array[M][N];
int type[2], size[2] = { 1, 1 };
MPI_Aint displacement[2];
MPI_Datatype newtype;

MPI_Type_vector(M, 1, N, MPI_INT, &type[0]);

type[1] = MPI_UB;
displacement[0] = 0;
MPI_Type_extent(MPI_INT, &displacement[1]);

MPI_Type_struct(2, size, displacement, type, &newtype);
MPI_Type_commit(&newtype);
```

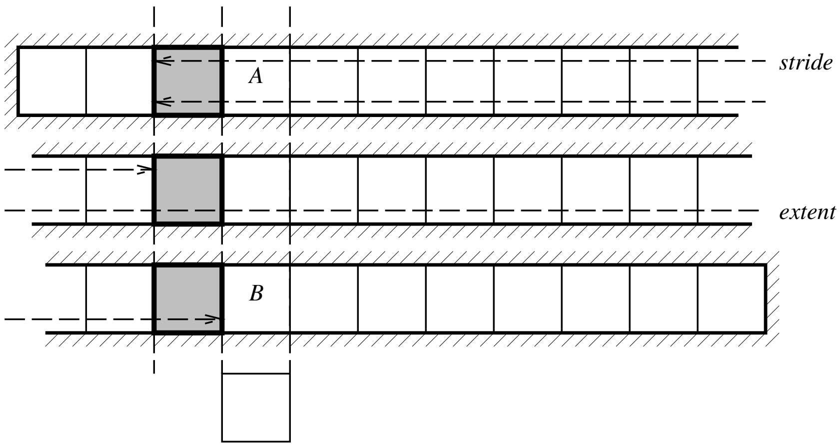

Figure 1: *Array Column Constructed Using MPI Vector Datatype*

One can see that even a simple and common case such as distinguishing a column of a two-dimensional array requires that the programmer has a distinct understanding of memory layout. The provision of a library that assists with the creation of commonly-seen substructures can be useful, especially coupled with the parser-based creation of derived datatypes addressed in [2]. As illustrated in Figure 2, the parser and library support could work together to allow the user to construct derived datatypes and many of the commonly-seen data substructures without having to be concerned with the details of memory storage.

The following section describes the library support implemented in this project.

4 Library Support

4.1 MPI-2 Subarray Constructor

The recently released MPI-2 interface [4] includes a subarray constructor that creates a datatype matching an n-dimensional subarray of an n-dimensional array. The first effort in this project

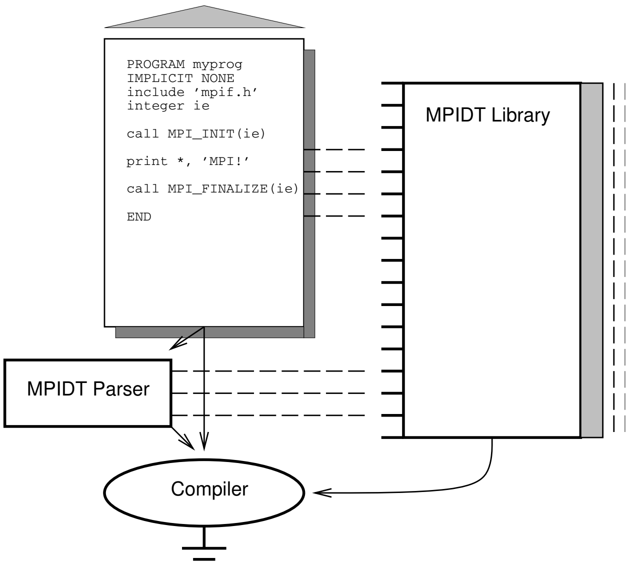

Figure 2: *The parser and library support isolate the user from most of the low-level memory concerns needed for the MPI datatype creation.*

involved the development of this routine. This provides an early implementation of the subarray constructor, thus enabling programmers to start utilising it and easily port their code to the MPI-2 interface when vendor implementations become available. C and Fortran index orders are supported.

The prototype for the routine is as follows:

MPIDT_Create_subarray(`int` **ndims,** `int` **array_of_sizes[],** `int` **array_of_subsizes[],** `int` **array_of_starts[],** `int` **order,** `MPI_Datatype` **oldtype,** `MPI_Datatype` ***newtype)**

IN	**ndims**	number of array dimensions
IN	**array_of_sizes**	number of elements in each dimension of full array
IN	**array_of_subsizes**	number of elements in each dimension of subarray
IN	**array_of_starts**	starting coordinates of subarray
IN	**order**	C-order or Fortran-order flag
IN	**oldtype**	array element datatype
OUT	**newtype**	new datatype

The routine conforms to the specifications defined in the MPI-2 standard, e.g. erroneous conditions are as defined therein.

4.1.1 Subarray Constructor with Default Ordering

The MPI-2 subarray constructor described above is a general purpose routine; however, it is possible to provide routines that directly address more common cases. One example is that the index ordering can be assumed to be implicit in the calling language. Since C and Fortran programs typically use the index ordering convention of their associated language, some programmers may not even be aware of the differences between the two conventions. A routine is provided in this library that removes the requirement of indicating the index ordering, with the purpose of making the common case simpler.

The prototype for the routine is similar to that of `MPIDT_Create_subarray`, with the argument that specifies the index ordering flag removed.

4.1.2 Subarray Constructor for Nonfixed Position

Another routine has been implemented that allows the user to specify a "floating" subarray block that is not fixed to any position within the array. This is useful for the tiling of subarray blocks if the programmer wishes to use multiple subarray blocks. The interface is similar to the routine described in §4.1.1, except that the argument `array_of_starts` has been removed.

4.1.3 Template Subarray Constructor

The concept of subarray construction and the tiling of subarray blocks leads to the idea of other library routines that can provide convenient features that are not included in the MPI-2 subarray constructor. One example is a routine that constructs a subarray template, thereby enabling the user to specify multiple instances of a subarray block. The user needs to merely indicate in which dimension the next item of the "subarray" datatype is to be found. Figure 3 illustrates two subarray blocks being fetched when multiple subarray blocks are to be found along dimension two.

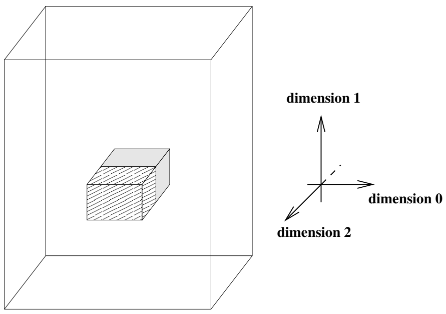

Figure 3: *Finding Multiple Subarray Blocks Using the Subarray Template Constructor.*

The prototype for the routine is as follows:

MPIDT_Create_templ_subarray(`int` **ndims,** `int` **array_of_sizes[],** `int` **array_of_subsizes[],** `int` **array_of_starts[],** `int` **order,** `MPI_Datatype` **oldtype,** `MPI_Datatype` ***newtype)**

IN	**ndims**	number of array dimensions
IN	**array_of_sizes**	number of elements in each dimension of full array
IN	**array_of_subsizes**	number of elements in each dimension of subarray
IN	**array_of_directions**	number of elements to move in each dimension for the next block
IN	**oldtype**	array element datatype
OUT	**newtype**	new datatype

It should be noted here that when specifying the `array_of_directions`, C and Fortran programmers should list the dimensions in the order corresponding to their respective languages.

4.2 Construction of Sub-Space within Array

Another function implemented in this project enables the user to define a sub-space partition within an array. This functionality is analogous to the `MPI_Cart_sub` routine for isolating a sub-space of a process topology. The user specifies the dimensions that should be covered in the extracted sub-space, as well as the direction in which to find the next instance of the sub-space. Where full elements of some dimensions are needed, this function would be conceptually easier to use than `MPIDT_Create_subarray`, since there is no need to specify the sizes of the subarrays in each dimension; the full length of the dimension is included by default.

MPIDT_Create_array_subdims(`int` **ndims,** `int` **array_of_sizes[],** `int` **array_of_remains[],** `int` **direction,** `MPI_Datatype` **oldtype,** `MPI_Datatype` ***newtype)**

IN	**ndims**	number of array dimensions
IN	**array_of_sizes**	size in each dimension of full array
IN	**array_of_remains**	logical array specifying dimensions covered
IN	**direction**	direction in which to move to find “next” subarray
IN	**oldtype**	array element datatype
OUT	**newtype**	new datatype

The following code fragment illustrates the high-level interface provided for creating a column of a two-dimensional array:

```
int array[M][N];
int sizes[2] = { M, N }, remains[2] = { 1, 0 };
int num_dims = 2, direction = 0;
MPI_Datatype newtype;
MPIDT_Create_array_subdims(num_dims, sizes, remains, direction,
                           MPI_INT, &newtype);
```

4.3 Array Halo Construction

Regular domain decomposition is a commonly-seen model used in parallel computing, and enjoys wide use in a variety of application areas such as image-processing and the modelling of many physical phenomena. This decomposition method often requires that processes swap the boundaries (halos) of their multi-dimensional data sets. It is therefore useful to be able to construct datatypes matching these boundaries. An interface that creates datatypes corresponding to the boundaries of arrays is provided.

MPIDT_Create_array_halo(`int` **ndims,** `int` **array_of_sizes[],** `int` **depth,** `MPI_Datatype` **oldtype,** `MPI_Datatype` **newtypes_front[],** `MPI_Datatype` **newtypes_back[])**

IN	**ndims**	number of array dimensions
IN	**array_of_sizes**	size in each dimension of full array
IN	**depth**	depth of array boundary
IN	**oldtype**	array element datatype
OUT	**newtypes_front**	“front” boundary in each dimension
OUT	**newtypes_back**	“back” boundary in each dimension

Figure 4 illustrates an example defining the boundaries of a three-dimensional array. The user code is shown and the resulting subblocks returned by the routine are depicted in the diagram. The routine hides from the user the complexities of creating the boundaries. A simple call to the function, indicating the basic structure of the original array, the required depth of the boundaries, creates the datatypes defining the “front” and “back” boundaries of the array in each dimension. The default depth is 1; the user may specify any depth as long as the array bounds are not exceeded.

```
int array[I][J][K];
int sizes[3] = { I, J, K };
int num_dims = 3;
int depth = 1;
MPI_Datatype newtypes_front[3];
MPI_Datatype newtypes_back[3];

MPIDT_Create_array_halo(num_dims, sizes, depth, MPI_INT,
                        newtypes_front, newtypes_back);
```

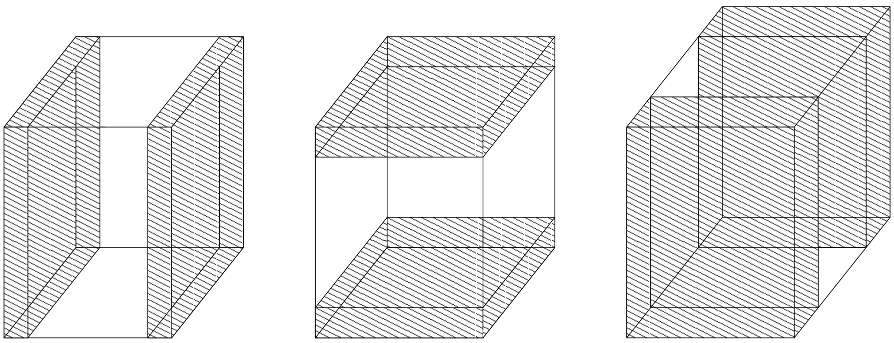

Figure 4: *Specifying the Boundaries of a Three-Dimensional Array*

It should be noted that for both C and Fortran programmers, the order of the dimensions in the arrays returned (`newtypes_front` and `newtypes_back`) conform to the order in which the array sizes are declared by the user. The reason for this convention is to enable programmers to easily match the relevant dimension in the returned datatype with that in their original array.

4.3.1 Halo Construction without Overlapping Boundaries

Boundary swapping may require the exclusion of any overlapping sections of boundaries. In particular, this may be necessary since processes may update, at the same iteration, all of their boundaries by copying the relevant data from the other processes. This may imply some loss of data for the overlapping sections of boundaries: the results of updating one boundary may get overwritten when another boundary gets updated. Therefore, overlap sections usually require special consideration on the programmer’s part when communicating with other processes. In order to address this issue, a function is provided that enables the user to create array boundaries not containing the overlap areas. The prototype similar to the `MPIDT_Create_array_halo` function, with a logical parameter that specifies whether overlap sections are to be included.

Figure 5 illustrates this concept using a two-dimensional array. The boundaries in dimensions one and two are selected as shown; the overlapping section is excluded.

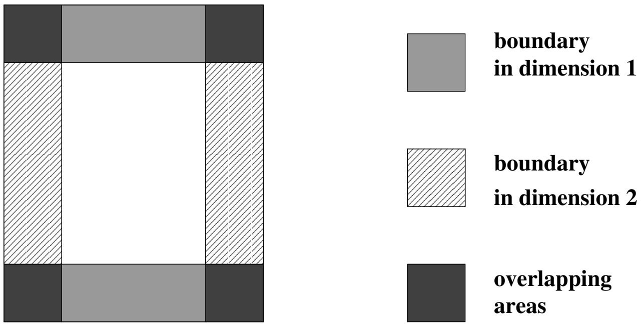

Figure 5: *Overlapping boundary sections can be excluded.*

4.3.2 Halo Construction with Varying Depths

The required boundary areas of a data block are not necessarily the same thickness in all dimensions. Therefore, it is desirable to enable the user to specify different boundary depths for the dimensions of the data set. A routine is implemented that accepts an array argument `array_of_depths` in place of the single integer `depth` argument mentioned above.

The order in the `array_of_depths` is assumed to be row-major or column-major according to the calling programming language.

4.4 Fortran 90 Triplet Notation Support

Fortran 90 allows programmers to specify (possibly non-contiguous) array sections by using subscript triplet notation. Figure 6 illustrates the selection of non-contiguous elements of an array by using this syntax.

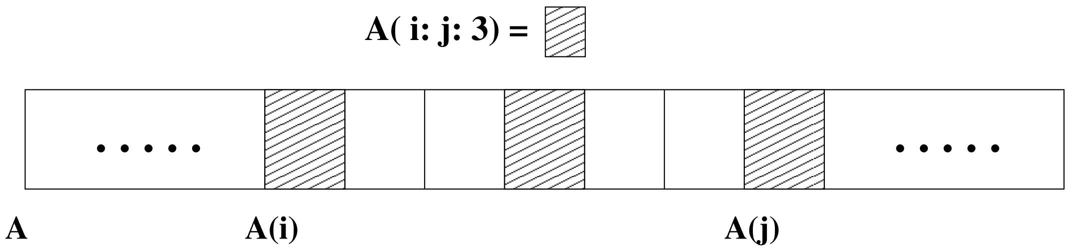

Figure 6: *A non-contiguous array section specified using Fortran 90 triplet notation.*

One critical problem is encountered when passing non-contiguous array sections to non-blocking MPI communication calls. The compiler may copy the non-contiguous data into a contiguous temporary buffer and pass this buffer to the MPI communications routine. The buffer may be released when the MPI call returns; however, MPI still needs to access the data in the background in order to complete the communication. This may cause incorrect memory accesses.

The difficulty described here stems from the incompatibility between Fortran 90 and the specification of non-contiguous data sets in MPI. One approach to bridging the gap between these two specifications is to provide a routine that enables the construction of MPI derived datatypes that match array sections created by Fortran 90 triplet notation. The routine implemented here allows programmers to define an MPI derived datatype by indicating the relevant subscript triplets.

MPIDT_Create_array_section(`int` **ndims,** `int` **array_of_sizes[],** `int` **array_of_triplets[][3],** `MPI_Datatype` **oldtype,** `MPI_Datatype` ***newtype)**

IN	**ndims**	number of array dimensions
IN	**array_of_sizes**	size in each dimension of full array
IN	**array_of_triplets**	array specifying subscript triplet in each dimension
IN	**oldtype**	array element datatype
OUT	**newtype**	new datatype

The following code segment illustrates the user code that constructs an array section by utilising this routine. The MPI derived datatype can be used in non-blocking communication without the risk of illegal memory accesses.

```
integer array(I,J,K)
integer sizes(3)
integer triplet(3,3)
integer sectype, sendreq, ierr

sizes(1) = I; sizes(2) = J; sizes(3) = K;

!-- Send array(:, J-2:3:-2, 1)
triplet(1,1) = MPIDT_INDEX_ALL
triplet(2,1) = J-2;  triplet(2,2) = 3;  triplet(2,3) = -2
triplet(3,1) = 1;  triplet(3,2) = MPIDT_INDEX_NONE

call MPIDT_Create_array_section(3, sizes, triplet, MPI_INTEGER,  &
                                sectype, ierr)
call MPI_Isend(array, 1, sectype, dest, tag, comm, sendreq, ierr)
```

When specifying subscript triplets, the distance between the start index and the end index does not have to be divisible by the stride; the routine will select the required elements occurring before the specified end index.

As can be seen from the example code fragment, the user may also specify negative strides. A diagnostic warning message is given if the stride in any dimension will not cover the intended elements, e.g. when the start index is greater than the end index with a positive stride.

Since it is aimed at assisting Fortran programmers, this routine assumes column-major ordering.

4.4.1 Fortran 90 Support for Negative Indices

The routine described in the previous section assumes that subscripts start at 1, the default start index for Fortran. However, Fortran 77 and Fortran 90 allow programmers to impose a desired range of values on the subscripts by indicating the lower and upper bounds explicitly. In Fortran 90, non-positive subscript values are allowed, as well as positive start indices other than 1. A routine is implemented here that supports this feature. The user needs to indicate the starting indices of the array in each dimension. This is done through an extra argument, `array_of_starts`, that is supplied in addition to the arguments that existed in the previous routine.

4.5 General Implementation Issues

The routines are implemented in C, and Fortran wrappers are provided. Where appropriate, all arrays and dimensions are assumed to be in the order corresponding to the calling language (with the obvious exception of the MPI-2 subarray constructor, which has an argument representing the index order flag). The Fortran bindings achieve the correct index ordering by reversing the array indices and converting to C subscripts where necessary.

In each routine, error-checks are made and warning messages given wherever possible, the most typical case being that the specified subblocks may exceed array bounds.

It is hoped that the library routines are efficient and cause little memory overhead. Creating intermediate datatypes are avoided where possible, and intermediate datatypes created in the routines, as well as dynamic allocations are freed up before exiting the routine.

5 Discussion and Further Work

The library support provided in this project is useful for the creation of data substructures required by many parallel applications. It hides the complexity of creating MPI datatypes for data substructures, and provides a high-level interface that is simple and straightforward to use. All library routines use only standard MPI datatype constructors, making the library portable to MPI implementations on any platform. The library can be accessed from ????.

Further work in the area of library support could include the implementation of routines for creating MPI derived datatypes that match the local part of global data distributed by HPF directives. The MPI-2 standard includes support for such datatype creation. In addition, the current library support can be enhanced according to user feedback.

The transfer of compound datatypes faces new issues with the increasingly wide use of object-oriented programming languages. These languages introduce the notion of a new compound datatype: a class. Can objects be transferred in a similar way to other complex datatypes? This would imply that not only do the data fields associated with the object get shipped, but the object's methods should somehow be made accessible at the receiving end as well. Particularly with the advent of intrinsically object-oriented languages like Java, the issue of being able to transfer the data state of objects becomes more important.

For non-object-oriented languages, the mechanisms for providing access to routines residing on remote locations has typically used the *RPC* (Remote Procedure Calls) paradigm, which provides an interface that enables applications to communicate with each other across networks. Client applications make requests on the server applications, which perform the service requested and send back the reply. This model has also been prevalent in most standards adopted for the remote accessing of objects' methods in object-oriented languages. The industry's standard for heterogeneous systems of distributed objects, *CORBA* (Common Object Request Broker Architecture) [9], is one example. It enables the interface for the methods residing on the remote machine to be made available on the local machine, thereby allowing method calls on remote objects. Java's standard, *RMI* (Remote Method Invocation) works according to a similar model.

RMI involves efforts concerning the transfer of the internal states of objects. Objects, like any other data, can be arguments to remote methods invoked via RMI, which raises the need for being able to marshal these objects and ship them over to the remote server, which, in turn, should unmarshal them. The Java Development Kit (*JDK*) 1.1 [8] and later versions include a Java interface (`Serialize`) that ensures that the data fields of objects (which are not necessarily in contiguous storage) get packed into a contiguous bytestream which can be sent across a network medium.

The *Serialize++* [7] library serves to parallel the functionality of Java's `Serialize` interface for C++, thus enabling the user to pack the current data state of a Java object and restore the data state in an equivalent C++ object at the other end (or vice versa). For example, one can "serialize" an object such as a Java `Vector` (`java.util.Vector`) or `Stack` (`java.util.Stack`), and have its data state restored in a C++ object of equivalent implementation. The porting from one object to another is achieved by a functionality that creates a mapping relationship between the objects at the sender and receiver ends. An important point here is that users can specify their own mapping rather than using the default mapping. This is comparable to MPI datatypes that are used in communications calls. In much the same way that MPI programmers specify two datatypes for the sender and receiver ends of the communication, here the programmer needs to specify two objects for both ends of the communication, and ensure that the desired data fields of one object get restored into the correct fields of another. The ability to "serialize" the data states of objects also carries an important implication: if it is possible to serialize C++ objects, then it is also possible to exchange C++ objects with other C++ objects. In effect, objects *can* be transferred as long as the receiver end has a reasonably compatible object that contains the functionality needed to make use of the data.

When a local object is serialized and passed as a parameter in an RMI call, the byte code for the local object's methods are not passed in the stream – only the fields that represent the current state of the object's data members. However, with the portability of Java's byte code, it is possible to transfer an *entire* object, including methods, between machines running the Java Virtual Machine. One potential problem with this is that along with the byte code for the desired object, that of the classes with which the object has dependency relationships (such as inheritance or aggregation) will also have to be loaded, causing network traffic.

`Voyager` [10] is a system that implements mobile Java objects. When an object residing on a processor is asked to move to another processor, the object state is serialized, meaning that the member data is packed (without the byte code). Next, a central server is contacted, which checks whether the byte code for the object's class resides on the destination machine of the object; if not, it is sent over. Also transferred is the byte code for any other classes on which the current object depends. Once the byte code reaches the target location, the serialized object can finally also be sent over and deserialized at its new location where it can start executing any invoked methods. Thus, effectively, an entire Java object has been exchanged between processors.

The issue of transferring datatypes can be generalised to considering the possibility of transferring objects. With increasingly wide use of parallel computing and object-oriented programming, both areas are likely to see the implementation of libraries and other tools that provide higher levels of abstraction for data transfer and insulate the user from architectural details.

References

[1] MPI: A Message-Passing Interface Standard. June 1995.

[2] MPI Datatypes Toolset. EPCC-SS96-10. Werner Augustin. EPCC Summer Scholarship Programme 1996.

[3] MPI Datatypes Toolset: Concepts II. A. Gordon Smith. EPCC Summer Scholarship Programme internal document. 31 July 1997.

[4] MPI-2: Extensions to the Message-Passing Interface. University of Tenessee, July 1997.

[5] Parallel Programming with MPI. Peter S. Pacheco. Morgan Kaufmann Publishers, 1997.

[6] Using MPI. William Gropp, Ewing Lusk and Anthony Skjellum. The MIT Press, 1994.

[7] Serialize.h++ Technical Report 97-05. 1997.
`http://www.roguewave.com/products/serialize/serialfax.html`

[8] JDK 1.1.4 Documentation. 1997.
`http://www.javasoft.com/products/jdk/1.1/docs/guide/index.html`

[9] The Object Management Group's Common Object Request Broker Architecture (OMG/CORBA)
`http://www.acl.lanl.gov/CORBA/`

[10] Voyager Homepage. 1997. Object Space.
`http://www.objectspace.com/Voyager/`

Berent Eskikaya was born in Istanbul, Turkey in 1976. She graduated in May 1997 with a Bachelor of Science degree in Computer Science from Allegheny College in Pennsylvania.

This project was proposed and supervised by A. Gordon Smith of the EPCC. He graduated with a BSc (hons) in Computer Science from the University of Edinburgh in 1992.