AN INTEGRATED KHOROS AND MPI SYSTEM FOR THE DEVELOPMENT OF PORTABLE PARALLEL DSP APPLICATIONS

Nathan Doss and Thom McMahon

Parallel Digital Signal Processing Group
NSF Engineering Research Center
Mississippi State University
Mississippi State, Mississippi 39762
{doss,thom}@erc.msstate.edu

ABSTRACT

This paper reports on design issues involved in combining two public-domain paradigms to create a parallel software environment for DSP programming. MPI (Message-Passing Interface), a message passing system, is an evolving standard for parallel computing. Khoros is an integrated software environment for DSP. The goal of this work is to describe and demonstrate a software design that exploits Khoros and MPI parallel libraries for the deployment of parallel DSP. The resulting system enables parallel DSP using the Khoros system for development and the MPI system for performance portability. The new system provides MPI-based toolboxes containing data parallel modules and utilizes MPI as a means of communication between modules. We also explore extensions to the Khoros polymorphic data model that include the notion of data distribution. This is an important concept for building data distribution independent parallel libraries and of particular interest in this paper, data distribution independent DSP libraries..

1. INTRODUCTION

Khoros [17,25] is a powerful system, providing a visual programming environment and a wide range of development tools for the development of DSP applications. Cantata [33], the visual programming environment, provides a dataflow model for building applications from collections of modules organized into toolboxes. The application engineer builds a block-diagram representation of the application using icons that represent the various modules, visually linking the icons together in order to specify the communication flow of the design. Khoros provides several toolboxes containing hundreds of modules for algorithms pertaining to 2D/3D plotting, data manipulation, scientific visualization, geometry, matrix operations, image processing, as well as others. For example, the matrix toolbox contains modules that perform various matrix operations such as matrix addition and LU decomposition.

MPI (Message-Passing Interface) [13] is a standard for message passing introduced by the MPI Forum (a collection of researchers from industry, academia, and national laboratories) in April 1994. MPI has its roots in many previous message passing systems and therefore provides most of the features found in those message passing systems such as point-to-point and collective operations. Two of the most important and relatively new aspects of MPI are its support for libraries and inter-group communication. These two features are very important for the development of parallel dataflow applications. Using MPI, parallel libraries can be written independently of one another and independent of the architecture, then used together in a single application. Inter-group communication allows groups of processors that have been logically separated during the process of task subdivision to communicate with one another.

The goal of this project was to integrate Khoros and MPICH[6,9], a model implementation of MPI developed jointly by Argonne National Laboratory and Mississippi State University, merging the development features of Khoros with the portability and functionality of MPI in order to provide a platform for the development of portable parallel DSP applications. This provides application engineers with a powerful and familiar environment for developing and prototyping parallel DSP applications with little input required from the application engineer in order to make the application parallel. Parallel libraries and inter-group communication supported by MPI allow us to achieve these forms of parallelism within the Khoros and MPI system:

- **data parallel modules** MPI is used to build a Khoros toolbox of data parallel operations and algorithms.
- **parallel flow between modules** Inter-group communication and other MPI communication techniques provide a model for parallel flow of data between modules. Additionally, this communication can be initiated through an abstract “polymorphic” data layer that hides issues such as data distributions and format.

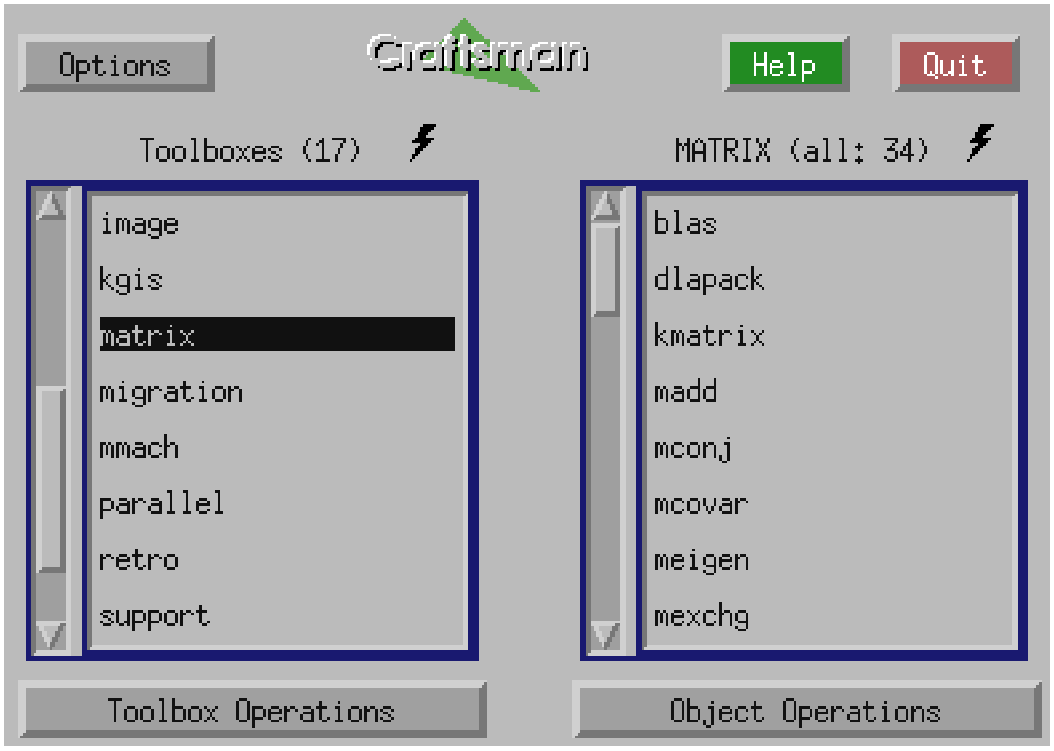


Figure 1. Screen Snapshot of the Craftsman Interface

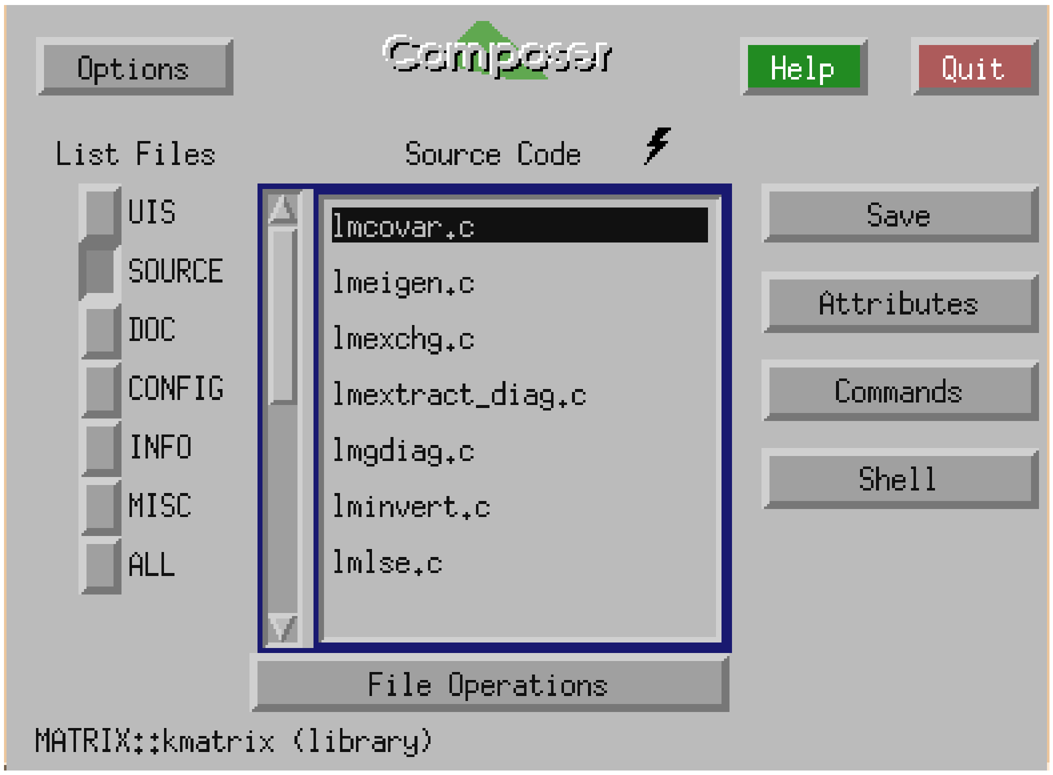


Figure 2. Screen Snapshot of the Composer Interface

- **task parallelism** Through Khoros, a graph of independent operations is built. Using MPI's support for library development, data parallel tasks with no data interdependencies can execute concurrently.

An important aspect to developing this system is determining the correct abstractions to use for data representation. The choice of abstraction greatly affects the efficiency of both the data parallel modules and the inter-module communication. The data abstraction is also a very important factor in determining how data parallel libraries will be structured and designed in general.

This paper first introduces the reader to both the Khoros and MPI systems. A description of the integrated Khoros and MPI system as well as the key design issues and challenges are reviewed followed by a discussion of parallel versions of a two basic DSP operations.

2. BACKGROUND

This section introduces both Khoros and MPI in some detail. We do not attempt to cover all aspects of either system (as that is beyond the scope of this paper) but do attempt to detail aspects of each system relevant to the work we have done.

2.1. Khoros

Many DSP applications are developed by combining and reusing well-understood libraries for basic DSP operations. Khoros provides a large set of DSP operations as well as an environment where new libraries of DSP operations can be built and developed. The development environment includes support for source code development, manual page development, copying existing objects, binary and library compilation and structuring.

Another important aspect of Khoros is its visual environment (called Cantata) for combining DSP operations into "workspaces". Khoros workspaces resemble dataflow graph structures and can be thought of as visual programs that can be "run".

Khoros also provides a general purpose data abstraction that provides a device-independent method for accessing data (these services are called the "data management services" in Khoros) using different models. The most general of the models supported by Khoros is the polymorphic data model which allows applications to access the data in an application specific way. For example, a matrix multiplication operation might access two-dimensional data as a matrix while a scalar multiplication operation might access the data as one-dimensional vectors.

The manual set for Khoros [17] provides detailed informationon Khoros. Additional information on Khoros can be found at the Khoral Research, Inc. site on the World Wide Web [18].

2.2.1. Development Environment

As noted previously, the Khoros development environment provides support for many different aspects of program development. The two main tools that provide this support are "Craftsman" and "Composer". Khoros also provides an interactive GUI builder, "Guise".

Craftsman Khoros operations are organized as a set of toolboxes. Examples include a "matrix" toolbox that provides matrix operations such as addition, multiplication, transposition and a "wavelet" toolbox that provides operations based on wavelet transforms. Craftsman (see Figure 1) provides an interface for constructing new toolboxes and modifying existing toolboxes (e.g., adding or removing operations from the toolbox).

Composer Khoros toolboxes consist of a set of possibly different kind of objects. These objects may be one of the following types:

- **kroutine** An operation without a graphical interface usually created with C.
- **xvroutine** An operation with a built in graphical interface usually created with C.
- **pane** A graphical user interface that can serve as a wrapper for other Khoros objects.
- **script** A shell script.
- **library** A set of functions that implement various operations possibly used by kroutines and xvroutines.

Composer (see Figure 2) provides an interface for modifying an object and it's associated resources (i.e., man pages, C source files, etc.).

2.1.2. Cantata

Figure 3 shows a very simple workspace from Cantata. Each of the boxes represents an operation on the data. Lines entering the left edge of a box represent input to the operation (data is communicated through shared files or shared memory). Lines exiting the right edge of a box represent the output of the operation. If the small window icon in a box is clicked with the mouse, a "pane" is activated that allows the user to modify the

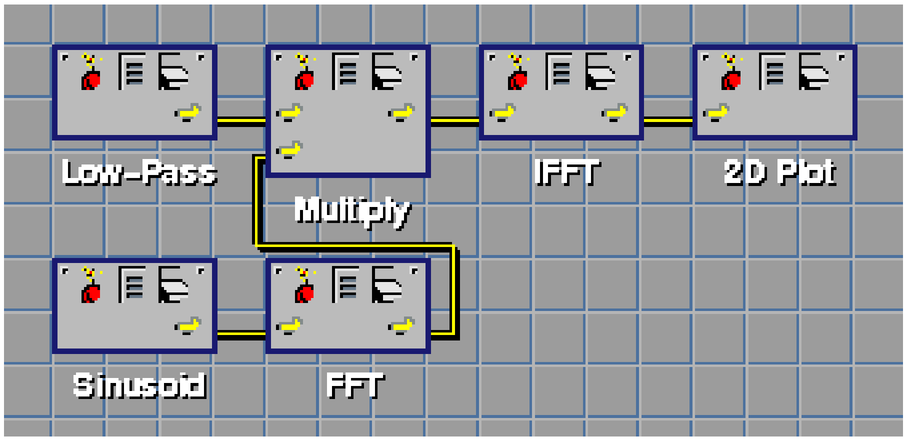
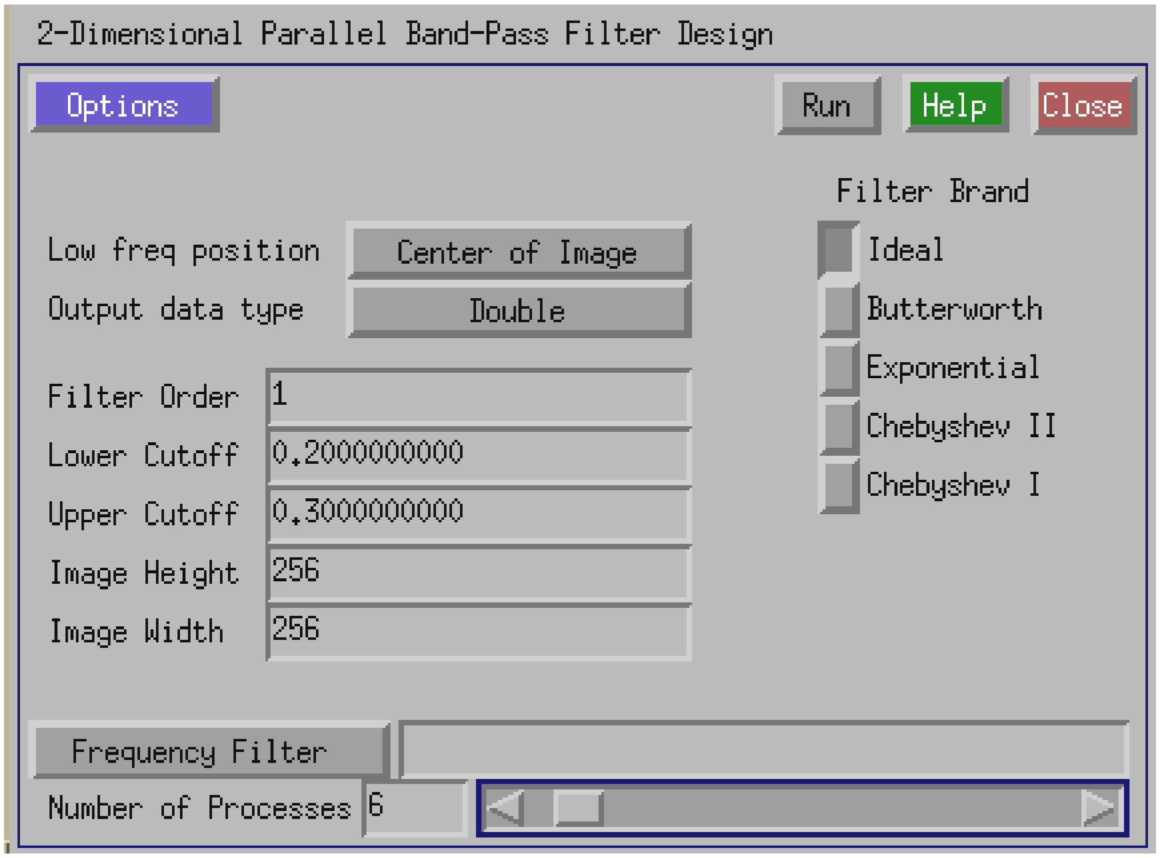


Figure 3. An Example Khoros Workspace



2-Dimensional Parallel Band-Pass Filter Design

Low freq position: Center of Image
Output data type: Double

Filter Order: 1
Lower Cutoff: 0.2000000000
Upper Cutoff: 0.3000000000
Image Height: 256
Image Width: 256

Filter Brand
[✓ Ideal]
Butterworth
Exponential
Chebyshev II
Chebyshev I

Frequency Filter
Number of Processes: 6

Figure 4. A Pane from a Khoros Application

parameters of the operation (see Figure 4).

2.1.3. Khoros Data Model

One of the goals of Khoros is to provide a framework upon which independently developed libraries can work together. Of central importance to this goal is the ability to encapsulate the details of data storage, transfer, and access. As shown in Figure 5, Khoros provides two levels of abstraction. At the lowest level are the “Data Management Services” which manage many of the low level details of data management; including data transport and file format. Above this level are various data services which provide abstract views of the data. The most widely used and most general data service is the polymorphic data service (also called the polymorphic data model). The data services provide the means by which applications can access the data in a format which is natural to the application. Although there are other data services (e.g., geometric data service) we only discuss the polymorphic data service in this paper.

Polymorphic Data Model The polymorphic data model consists of several attributes:

- **mask** provides validity informatio
- **location** location in three dimensional space
- **time** location in time
- **map** used for compression
- **value** the actual values of the data

Of primary concern is the “value” component which contains the actual data values. It can be thought of as a series of volumes in time (width, height, depth, elements, time). For example, data that represents the temperature and salinity of the ocean for the month of January could be represented as:

- **(x,y,z)** three dimensional location in the ocean (corresponds to width, height, depth in the value segment).
- **time** the hour and the day when the data sample was taken.
- **temperature,salinity** these are the “elements” of the data. A color picture might have three elements (red, green, blue).

Data can also have other attributes associated with it (e.g., type - integer, double, float, etc.).

Application Programmer’s Interface Khoros provides a set of primitive functions for accessing and

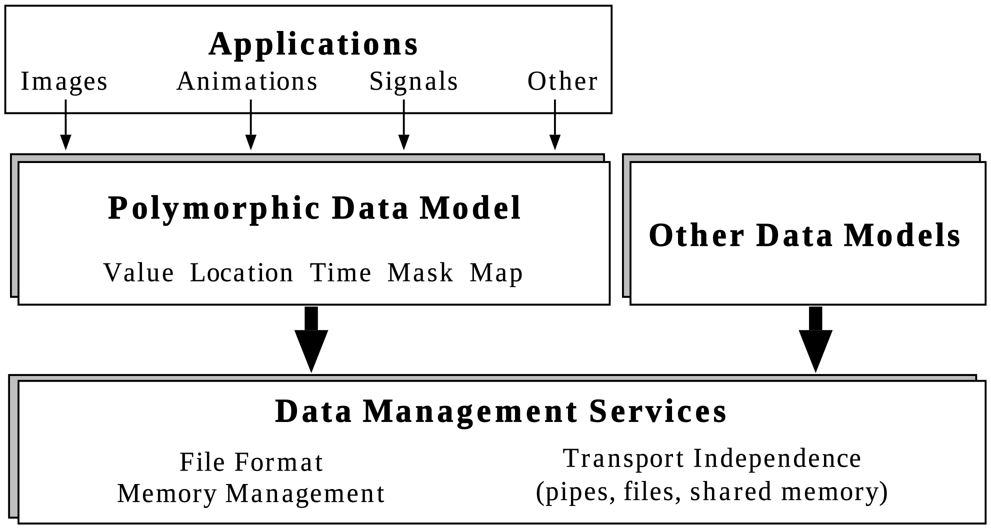


Figure 5. Khoros Design for Supporting Abstract Data Access

manipulating data objects. Some of the more important functions include:

- Functions for creating and destroying data objects:

  kpds_open_input_object() - oens a data object for input

  kpds_open_output_object() - opens a data object for output

  kpds_close_object() - closes an object

- Functions for setting, copying, and retrieving object attributes:

  kpds_get_attributes() - gets attributes associated with a particular object

  kpds_set_attributes() - sets attributes of a particular object

  kpds_copy_object() - copies the data and attributes of an object to another object

- Functions for reading and writing an object:

  kpds_put_data() - writes data to an object

  kpds_get_data() - reads data from an object

The Khoros manual set [17] has documentation on the complete set of API functions.

2.2. MPI

MPI (Message-Passing Interface) is a standard for high-performance portable message passing that supports an explicit MIMD message passing model. One of the key goals of MPI is support for safe message passing libraries, something missing in most of the message passing systems that existed prior to MPI.

The rest of this section introduces the MPI interface and gives a gentle introduction to writing programs with MPI. Emphasis is also placed on writing libraries with MPI. The discussion uses the C bindings for MPI functions, although Fortran bindings for each function are set forth in the MPI standard. For a complete description of the MPI standard, the MPI standard document contains a full description of the MPI interface. Additional information on MPI can be found at the MPI World Wide Web page [10].

2.2.1. Basic MPI

Although MPI is a fairly large specification in terms of the number of functions in the API, it supports a fairly small number of orthogonal concepts:

- Point-to-Point Communication
- Collective Communication
- Process Groups
- Communication Contexts
- Virtual Topologies
- Profiling

Although there are over 120 functions in the MPI interface, we concentrate our discussion on the following nine functions: MPI_Init, MPI_Finalize, MPI_Comm_rank, MPI_Comm_size, MPI_Bcast, MPI_Send, MPI_Recv, MPI_Comm_split, MPI_Comm_dup, and MPI_Intercomm_create.

Environment All MPI programs must initialize and close the MPI environment. This is accomplished with the MPI_Init and MPI __Finalize functions. A minimal MPI program, as shown below, must have these two functions.

```
#include <mpi.h
main (argc, argv)
int argc;
char **argv;
{
    MPI_Init(&argc, &argv);
    MPI_Finalize();
}
```

As multiple instances of the previous code may get instantiated during a parallel execution, functions for determining the number of instances of the program (size or number of processes) and a position (rank in collection of instances) are necessary.

```
#include <mpi.h
main (argc, argv)
int argc;
char **argv;
{
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,
                        &size);
    MPI_Comm_rank(MPI_COMM_WORLD,
                        &rank);
    MPI_Finalize();
}
```

As can be seen from this simple code, the first argument to both the size and rank functions is MPI_COMM_WORLD. MPI_COMM_WORLD is known as a communicator and contains information about all processes in the current MPI program execution. It is initialized by the MPI_Init call.

Communicators Communicators may contain various types of information, but the two most important elements of a communicator are a process group and a communication context. Process groups and communication contexts serve two very important functions in MPI:

- **Scope** Groups scope an MPI operation by limiting the processing elements that participate in an operation. This is important, for example, in collective operations where one may not want the group of all processes to participate in the operation. In this case, a communicator other than MPI_COMM_WORLD would be used.

- **Safety** Communication contexts provide separate message spaces or message contexts to applications. This is especially important when independently written libraries are used together in the same program.

Initially, MPI_COMM_WORLD is the only defined communicator. Two important operations that can be used to construct new communicators are MPI_Comm_dup and MPI_Comm_split.

```
#include <mpi.h>

main (argc, argv)
int argc;
char **argv;
{
    int     rank, size;
    MPI_Comm dup_comm, split_comm;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,
                        &size);
    MPI_Comm_rank(MPI_COMM_WORLD,
                        &rank);

    MPI_Comm_dup(MPI_COMM_WORLD,
                    &dup_comm);
    MPI_Comm_split(MPI_COMM_WORLD,
                    rank%2, 0, &split_comm);
    MPI_Finalize();
}
```

The output communicator from MPI_Comm_dup contains the same group as the input communicator but a different communication context. MPI_Comm_split splits or divides the group of a communicator as well as providing a new context. In the above example, the output communicator from MPI_Comm_split will consist of all odd processes if the process' rank is odd or all even processes if the process' rank is even.

Until now, we have only mentioned one type of communicator. The communicators we have seen are called "intercommunicates"; communication only occurs within one group. MPI also supports communication between two distinct groups through "intercommunicator." In contrast to an intracommunicator, intercommunicators consist of two disjoint groups and are constructed from two intracommunicators with MPI_Intercomm_create.

Point to Point MPI provides two basic functions for sending and receiving messages. MPI_Recv receives messages that are sent by MPI_Send.

```
#include <mpi.h>

main (argc, argv)
int argc;
char **argv;
{
    int rank, size, tag=1, count=1;
    int message;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,
                        &size);
    MPI_Comm_rank(MPI_COMM_WORLD,
                        &rank);

    message = rank;
    if (rank == 0)
            MPI_Send(&message, count, MPI_INT,
                    tag, MPI_COMM_WORLD);
    else if (rank == 1)
            MPI_Recv(&message, count, MPI_INT,
                    tag, MPI_COMM_WORLD,
                    &status);

    MPI_Finalize();
}
```

In this program, process 0 sends a message to process 1. The message consists of one integer (count=1 type=MPI_INT is sent from message). A communicator must be specified for all communication operations (in this case MPI_COMM_WORLD is used) as well as a tag. Tags provide another level of selectivity; the tag of the send must match the tag of the receive.

MPI provides several variations on basic send and receive operations. Send and receive operations may be non-blocking. There are also different send modes that provide different protocols for sending (e.g., a synchronous send that does not complete until the receive operation on the destination process has begun).

MPI provides several variations on basic send and receive operations. Send and receive operations may be non-blocking. There are also different send modes that provide different protocols for sending (e.g., a synchronous send that does not complete until the receive operation on the destination process has begun).

Collective Collective operations work on a group of processes (specified in a communicator). Most MPI collective operations can be categorized according to the data motion of the operation:

- **One to all** One process, the "root", contributes data that is sent to all others. An example of this is the broadcast operation MPI_Bcast.
- **All to one** All processes contribute data that is sent to only one of them. An example is a gather operation where a vector is distributed among a group of processes. After the gather, the "root" contains the whole vector.
- **All to all** All processes contribute data, all processes receive data. An example would be an allgather operation that can be thought of as a all-to-one gather followed by an one-to-all broadcast.

The following program illustrates the broadcast operation.

```
#include <mpi.h>
main (argc, argv)
int argc;
char **argv;
{
    int rank, size;
    int message;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,
                        &size);
    MPI_Comm_rank(MPI_COMM_WORLD,
                        &rank);

    if (rank == 0)
            message = 999;
    MPI_Bcast(&message, count, MPI_INT, 1,
                    MPI_COMM_WORLD);

    MPI_Finalize();
}
```

After the broadcast operation all processes in MPI_COMM_WORLD contain 999 in message. One important aspect of MPI collective operations is that all processes in the communicator group (in this example, the communicator was MPI_COMM_WORLD) must call the collective operation.

2.2.2. Writing Parallel Libraries with MPI

MPI features for library safety provide an environment where application programmers can expect reliable

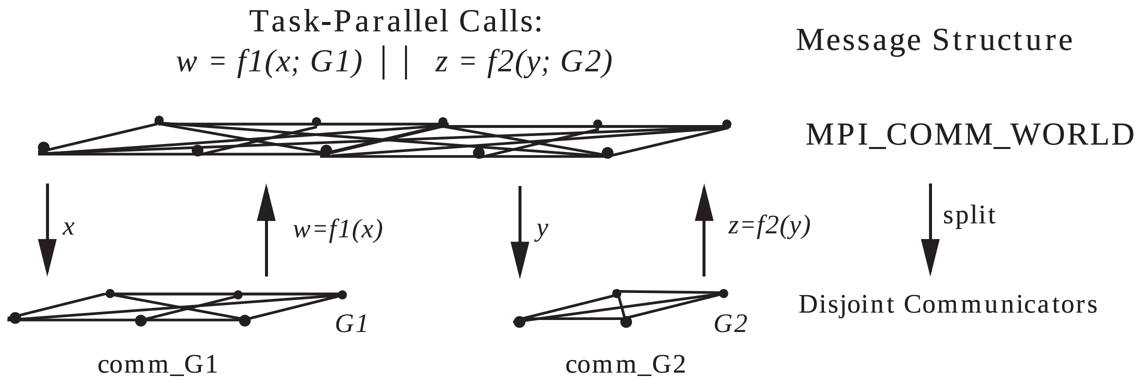


Figure 6. Example communicator construction and structure for task parallel applications.

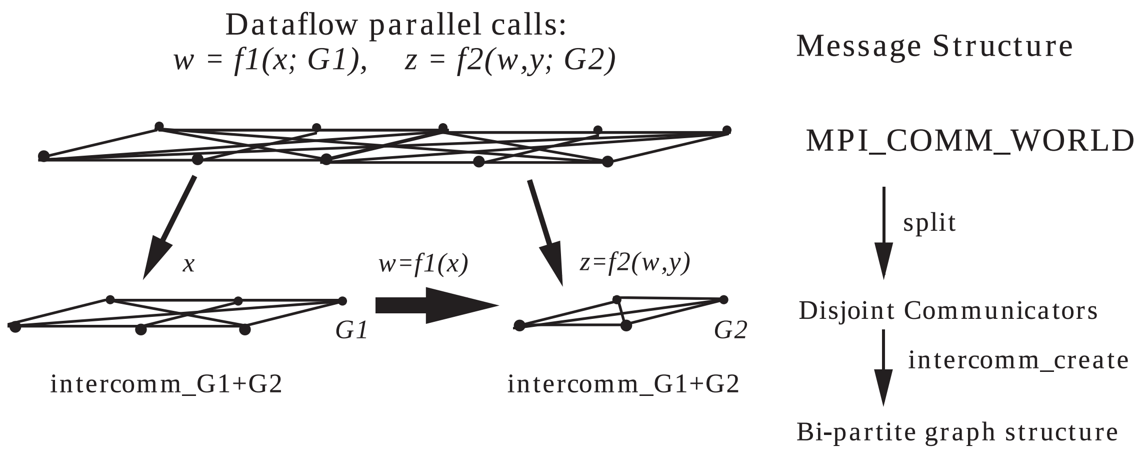


Figure 7. Example communicator construction and structure for dataflow application.

message-passing behavior without sacrificing performance. The use of separate communication contexts (i.e., distinct communicators) by different libraries (or different library invocations) insulates communication internal to the library execution from external communication. This allows the invocation of the library even if there are pending communications, and avoids the need to synchronize each entry into and exit from library code. A treatment of writing libraries with MPI is available in [30].

Figure 6 illustrates a possible way to structure communicators for a task parallel application. MPI_COMM_WORLD is first split (using MPI_Comm_split) into two distinct communicators, comm_G1 and comm_G2. Two separate tasks are invoked upon the two communicators. Figure 7 further shows how dataflow communication might occur between two process groups using intercommunicators. MPI __COMM_WORLD is first split using MPI_Comm_split, then joined in an intercommunicator relationship using MPI_Intercomm_create. The data is first passed through function f1 in the left side of the intercommunicator. The result is then passed to the right group and passed through function f2.

2.2.3. MPICH Implementation of MPI

MPICH [6,9] is a freely available implementation of MPI jointly developed at Argonne National Laboratory and Mississippi State University. It currently supports a wide range of platforms from supercomputers (e.g., Cray YMP, C90), parallel computers (e.g., IBM SP2, Intel Paragon), to networks of workstations. MPICH can be retrieved from: ftp://ftp.erc.msstate.edu/pub/mpi/mpich/.

3. KHOROS AND MPI INTEGRATED ENVIRONMENT

The design of an integrated MPI and Khoros development environment for portable parallel DSP applications consisted of the following:

- An investigation of the Khoros 2 system as it applies to DSP programming in order to recognize the areas that can be effectively and efficiently parallelized through integration with MPI.
- Definition of a “runtime” startup mechanism that Cantata can use with various parallel hardware.
- Extension of the Khoros data representation system to include support for data distribution.
- Design of a communication layer between parallel modules using the inter-group communication support provided my MPI.
- Investigation of approaches to optimizing Khoros execution for high-performance computing by compiling Cantata workspaces.

These issues are covered in detail in the following sections.

3.1. Parallel Process Startup

Process startup is not currently defined by the MPI standard, leaving these decisions to be made by the various implementors of MPI. For the MPICH implementation, process startup for one of the communication devices used in this project, known as the p4 device, requires the existence of a special file as well as special command line arguments to the programs. This file contains information about the programs to run and the machines those programs are to be run on, while the command line arguments identify the location of this file. Each MPI program when it calls MPI_Init, accesses the low-level p4 device code, which coordinates global initialization of all processes, such as assignment of unique process numbers. Since MPI programs that rely on the p4 device will not run correctly if these setup files are not created, it was necessary to bypass the Cantata process startup mechanism and create our own. Also, Cantata runs all modules on the same machine; MPI allows us to use all of the resources available, which is highly desirable.

Internally in the Cantata code, functions were added to take care of part of the MPI setup. These functions, in short, examine all the internal data structures used to house information about the workspace and modules within the workspace, identifying the ones associated with the parallel toolbox. From each of these identified data structures, these functions extract information about how many processes to use for each module as well as what connections to other parallel modules exist; the combined information is then written to a file. This plan seemed easy at first, but navigating the Khoros data structures proved to be a laborious and time-consuming task.

Once the file is created, a script called kmpirun transforms it into the special file used by the p4 device, starts up the parallel module programs, and starts up a master program called kmpihost. The kmpihost program, along with a special KMPI_Init function, was used to create the communicators required for safe communication between and within parallel modules. Each parallel module has a communicator called KMPI_COMM_WORLD, as well as intercommunicators with which the module can communicate with its connecting modules.

Shown in Figure 4 is a pane for a parallel module, with a slider bar the user can manipulate to specify the number of processors on which to run the module. One of the deficiencies of the current p4 device is the inability to pass different command line arguments to different programs, which inhibited progress of the original project plans. For this and other reasons relating to difficulties in mastering Khoros programming, the use of the p4 device was abandoned temporarily in favor of a much simpler device so that development of the parallel toolboxes could begin. Unfortunately, simplicity came at the cost of much of the portability provided by the p4 device.

Currently, the MPI-2 Forum is working on adding additional features to the MPI standard, including some support for dynamic process startup, which should greatly simplify the Khoros process startup problem. Proposed MPI_Spawn and MPI_Spawn_multiple functions allow dynamic creation of MPI processes, while the MPI_Connect and MPI_Accept functions allow two separate sets of MPI programs to engage in communication. These additional features would enable the kmpirun script to be bypassed, and all process startup could take place within Cantata. For example, the kmpihost program could be integrated into Cantata, with the additional task of starting up all of the parallel modules with MPI_Spawn or, more likely, MPI_Spawn_multiple. The separate modules could connect to each other using MPI_Connect and MPI_Accept, which will create intercommunicators between them, relieving Cantata of this task. These MPI-2 additions are still being debated, and it may be a while before implementations exist.

3.2. Extension of the Polymorphic Data Model for Distributed Data

One of the challenging problems of parallel library development is effectively managing data distribution. The polymorphic data model provided by Khoros (Ptolemy[23] has a similar approach) provides a framework for application domain independent interpretation of data. In this section, we introduce extensions to the polymorphic data model that provide a data distribution independent interpretation of data. This is important since data distribution and how it is handled in a parallel program or library can be one of the most important aspects of performance[2,7].

3.2.1. Data Distribution Fundamentals

The extensions we propose later in this section revolve around two fundamental concerns about how data is actually distributed among processors.

Virtual process topologies Processes can be thought of as having a topology or structured layout. Virtual topologies provide a convenient naming mechanism for process groups as well as providing a more natural way for some applications to logically represent the communication pattern of the processes. Although we do not use MPI topologies directly in our proposed extensions, we do note that MPI provides support for two types of virtual topologies: cartesian and graph. In our work, we limit our discussion to cartesian topologies with a maximum dimension of five as this maps to the five dimensional polymorphic data model.

Distribution types Data may be distributed differently along each dimension of the data. How the data is actually distributed is dependent on the size of the data (N) in a dimension and the number of processes in a dimension (P). There are three basic types of distributions that we consider:

- **linear** Data is distributed linearly among processes. Each process receives N=P data items with process p receiving data elements [[p * (N/P); ..., ((p + 1) * (N/P) - 1)].
- **cyclic** Data is scattered cyclicly among the processes. Each process receives N=P data items with process p receiving elements [p, P + p, 2P + p, ...].
- **all** This is not a “true” distribution since data is replicated on each process, but in many cases this is exactly what is desired by an application to increase performance (traditional tradeoff between time and space).

An important generalization of the linear and cyclic distributions is to consider each element to be a “block” of data elements with a particular block size. This is an important generalization since the linear and cyclic distributions can be thought of as specialized cases of the block linear and block cyclic distributions. For example, the linear distribution can be thought of as a block linear distribution with the block size set to 1 and the cyclic distribution can be thought of as a block cyclic with block size set to 1.

We extend the polymorphic data model by specifying the distribution type in each dimension as well as the virtual process topology.

3.2.2. Data Distribution in the Polymorphic Data Model

In order to specify the distribution of data in the value segment of the polymorphic data model, we propose the addition of a distribution segment to the polymorphic data model that parallels the structure of the value segment. The distribution segment has five dimensions:

- width distribution
- height distribution,
- depth distribution,
- time distribution,
- element distribution.

Each of these dimensions is used to specify how data is to be distributed in the corresponding dimension of the value segment. The default distribution of each dimension is the all distribution. This implies that in the default case, data is replicated in each dimension on each process (i.e., all processes get all the data).

Another aspect of a distribution is the block size. Thus, another segment of the polymorphic data model is the is the block segment for specifying the block size of linear and cyclic distributions. For all distributions, the block size is ignored.

- block size of width distribution,
- block size of height distribution,
- block size of depth distribution,
- block size of time distribution,
- block size of element distribution.

3.2.3. Virtual Topology in the Polymorphic Data Model

Corresponding closely to the distribution segment is the topology segment used to specify the process topology. The topology segment contains information about the number of processes in each dimension:

- process topology width,
- process topology height,
- process topology depth,
- number of processes in time dimension,
- number of processes in element dimension.

By default, all dimensions are set to one except for the topology width which is initially set to the number of processes in the data parallel program. Combining the topology and distribution information provides all that is necessary to actually distribute the data.

3.2.4. Access to Global and Local Information

So far, we’ve seen three new segments that are needed to fully specify data distribution over a set of processes. An addition problem is that one may want either a “global” or a “local” view of the data. For example, the process may want to know the total size of a vector as well as the length of the sub-vector that is available locally. A process may also want to know both the number of processes in a virtual topology dimension as well as the process’ rank in that dimension. These problems are solved by having both a local and global handle to each data object.

Initially a global handle is opened using the existing Khoros open functions, providing access to the global view of the data. Since the object is a distributed object, this handle cannot be used to access the actual data. It serves only as a description of the dimensions and properties of the data. The application programmer then sets distribution and topology attributes on this “global” handle. A call is then made to a new Khoros function that actually applies these attributes to the data and creates a new “local” handle. The local handle contains information about how much data is available locally as well as information about the location of the process in the virtual topology. The actual syntax for this process is described in Section 3.4.

An important thing to note related to global and local views of data is that although the proposed extensions to the polymorphic data model resemble a distributed shared memory model or the distribution features of High Performance Fortran, we do not propose that the polymorphic data model maintain memory coherency when data is replicated or shared in multiple processes. We assume that it is the responsibility of the application programmer to maintain memory consistency. We do not believe this to be overly restrictive since it follows current practice for many distributed memory message passing libraries[2]. If memory consistency is not maintained, we consider the program to be erroneous.

3.2.5. Example: Distribution of a 1-D Vector

Assume that we have a 1-D vector of length N = 10 (in the value segment, width = 10) with a linear distribution and a 1-D process topology consisting of P = 5 processes. In this case it is easy to see each that process will receive two elements. Suppose now that we have a 2-D 2x2 (width = 2, height = 2) process topology. In this case, data will be distributed linearly in the width dimension and replicated in the height dimension as shown in Figure 8. Data that is of lower dimensionality than the process topology is replicated in one or more dimensions.

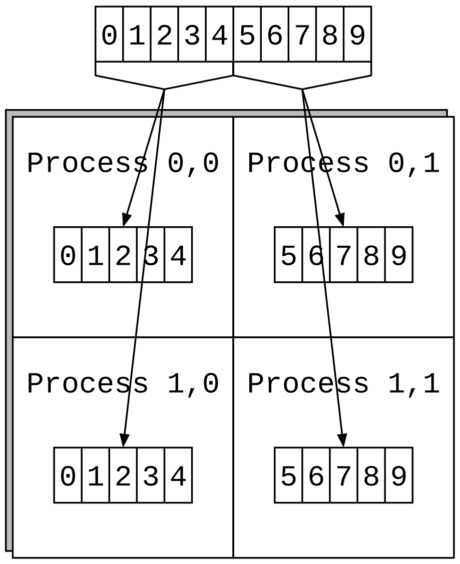


Figure 8. Distribution of a 1-D Vector on a 2-D Process Topology

3.3 Parallel Inter-group Communication

MPI does not contain a model for collective communication between two distinct groups; however we believe these type of operations to be very important for dataflow computing with message passing. Extensions to MPI that support this type of collective operation provide a natural interface for describing data movement from one group to another as well as an interface by which the natural parallelism of group to group communication can be exploited. The additional concern of data distribution further complicates the group to group communication model.

One of the main tradeoffs in data parallel libraries is the decision whether or not to redistribute data before performing an operation on the data. The choice is either to use a non-optimal algorithm on the data as it is or to use an optimal algorithm after redistributing the data. With many dataflow applications, the data is being moved so redistribution can occur during transit with little additional cost. Figure 9 illustrates the use of collective data distribution independent communication between data parallel modules. This section introduces an MPI model for inter-group communication and briefly discusses algorithms for data distribution independent inter-group communication.

3.3.1. A Model for MPI Collective Inter-Group Operations

In [29], we propose a set of intercommunicator collective communication extensions to MPI for consideration in MPI-2. Figure 10 illustrates one of the calls that we have proposed: intercommunicator allgather. An intercommunicator allgather can be thought of as a gather operation on one side of the intercommunicator followed by a broadcast to all processes on the other side. Communication patterns such as this one are very useful for describing inter-group collective operations at an abstract level without concern for how many processes are in each group. We will use this MPI-based terminology when describing the types of inter-group collective operations needed for implementing inter-group distribution independent communication.

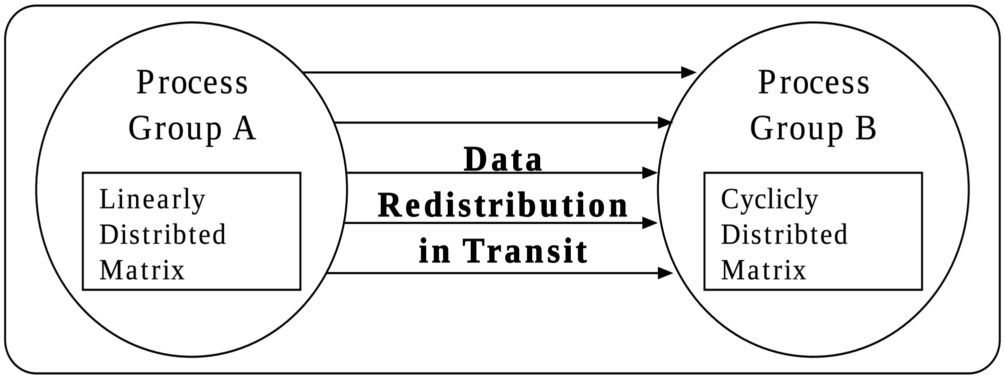

Process Group A
Linearly Distribted Matrix
Data Redistribution in Transit
Process Group B
Cyclicly Distribted Matrix

Figure 9. Data Distribution Independent Inter-Group Communication

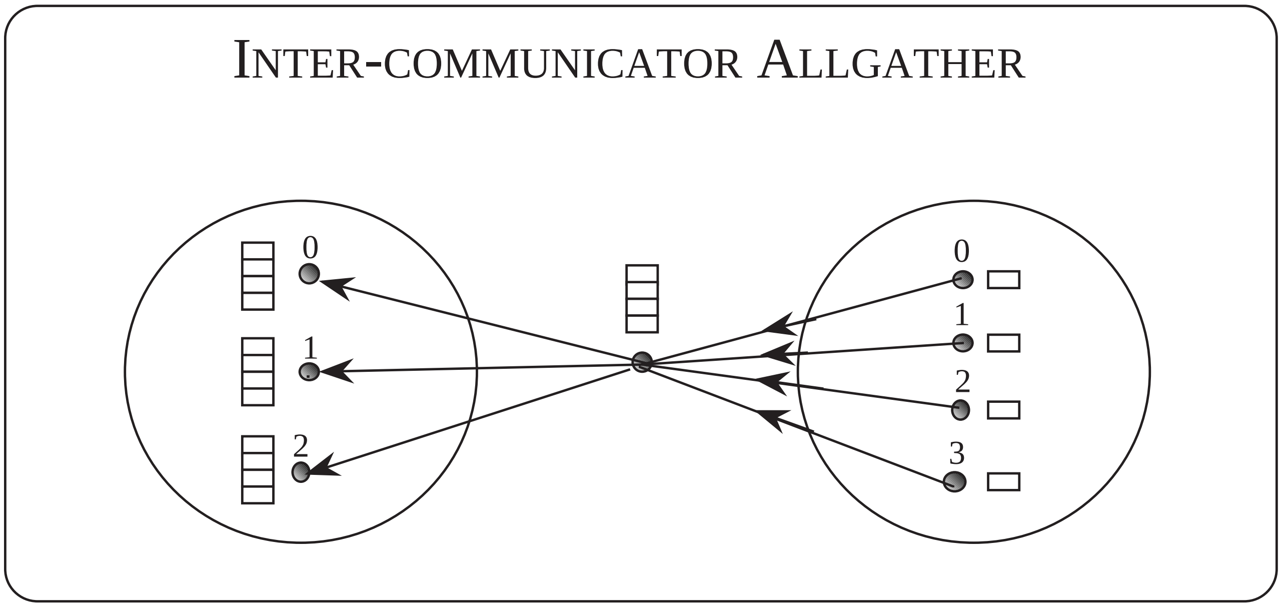


Figure 10. Inter-Group Collective Allgather Operation

3.3.2 Data distribution Independent Inter-Group Communication

Determining the best algorithm for inter-group communication can be quite complex especially when considering hardware issues (e.g., the bandwidth might be much lower between the two groups than within either of the groups - a group of paragon processes in a group communicating over 10Mb/s ether net to a group of SP1 processes). Adding to the complexity of analyzing these types of operations are the virtual topology and data distribution aspects we have introduced.

A naive approach to this problem is to use concurrent gathers in consecutive dimensions until all the data resides in a single process. This data is then transmitted to one process in the remote group. The remote group performs concurrent scatters in each of its dimensions. For linearly distributed data, simple MPI_Gather and MPI_Scatter operations can be used. For other distributions, the MPI_Gatherv and MPI_Scatterv operations can be used to reorder the data into a linear distribution. This approach works and is O(logP) (with a potentially large constant). Each gather (and scatter) is O(logP) with P equal to the number of processes in the current dimension. Distributing data from a 10x10 process topology to a 2x4x4 process topology is 2O(log10) + O(log2) + 2O(log4). Although this does work it does not exploit any parallelism in the connection between the two groups and potentially uses a large amount of temporary storage space for the gather and scatter operations. However, environments where the inter-group connection is relatively slow (compared to the speed of the intra-group connections) and the data set is small, this approach may be among the fastest.

Other algorithms that take advantage of concurrency in the network are possible. For example, it is possible, through a recursive approach, for each process in the sending group to determine the set of remote processes to whom the process should send data as well as the particular data needed by each remote process. Once this list has been made, the intercommunicator collective MPI_Alltoallv can be used to effect the transfer. The efficiency of this operation is dependent upon the complexity of the intercommunicator alltoallv operation. All-to-all operations on a single group have been studied extensively, however little (none that we know of at this time) study has been made of intergroup all-to-all operations.

3.4. API for Integrated Khoros and MPI Environment

The current Khoros polymorphic data model API is fairly simple, allowing the programmer to retrieve, store, and manipulate data objects with just a few functions. These functions have already been described previously and will not be repeated here. Targeting simplicity as well as backwards compatibility with existing applications, the new API for distributed polymorphic data types is kept as close to the original as possible. Only one new function is added, which will be optional within an application, and a number of attributes are combined with the existing set in order to allow the programmer to flexibly specify data distribution.

3.4.1. Opening Input Objects

The actual function for opening input objects, kpds_open_input_object, is kept the same, but the semantics are changed depending on the input argument to the function. Normally, the argument is the name of a file, but we can allow this to be something similar to “mpi=2”, which will accomplish two goals. The “mpi” notifies kpds_open_input_object that the object is not just a normal file, but resides in the memory space of some other group of processes (from here on referred to as the input group) and needs to be received from them using MPI point to point or collective communication calls. The “2” gives the application number of the input group, and specifies which intercommunicator to use for communication with the input group. The kpds_open_input_object routine returns the “global” handle to the distributed object, representing the global view of the data object.

3.4.2. New Attributes

In order to access the data, the kpds_set_attribute function will be used to define the values of additional attributes. These additional attributes, along with their default values and a short description of their function, are listed in Figure 3.4.2. With these attributes it is possible to both define a virtual topology as well as how the data will be distributed over that topology. For example, the following code segment would define a 2x3x4 topology, and distribute a 100x100x100 cube block linear over the width, block cyclic over the height, and “all” over the remaining dimensions.

```
kpds_set_attribute (object,
  KPDS_VIRT_TOPOL,
  2,3,4,0,0);
kpds_set_attribute (object,
  KPDS_DATA_DIST,
  KLINEAR,KCYCLIC,KALL,KALL,KALL);
kpds_set_attribute (object,
  KPDS_BLOCK_SIZE,
  15,5,0,0,0);
kpds_distribute(object);
```

Attribute	Default Value	Description
KPDS_VIRT_TOPOL	n,1,1,1,1	The shape of the virtual topology (n is the number of processes)
KDPS_DATA_DIST	KALL, KALL, KALL,KALL, KALL	The data distribution over the virtual topology. Value values are KLINEAR, KALL, and KCYCLIC.
KDPS_BLOCK_SIZE	1,1,1,1,1	The block sizes for distribution along each dimension. For the "all" distribution, this value is meaningless.

Figure 11. Additional Attributes for Data Distribution

3.4.3. Distribution of Data

Once the topology and distribution attributes have been set, a new polymorphic data function, kpds_distribute, is used to distribute the data in the correct manner, returning a handle to the process' local view of that data. This function behaves differently depending on where the data resides. If the data is in a file on disk, the root processor will read the data from disk and distribute it to all of the processors in the correct manner. However, if the data resides in the memory space of an input group, the data from the input group needs to be sent to the receiving group over the intercommunicator connecting them. This implies some sort of coordination between the root nodes of the two intercommunicators in order to establish the data distribution of the input group and the best way to communicate that data in such a way that it results in the correct distribution within the receiving group. This has already been discussed somewhat in Section 3.3. It is important to note that the local handle returned by kpds_distribute can have different information for each of the processors, since the local data sizes could be different if the distribution is not an even one. The kpds_distribute function is collective and must be called by all processes of a module.

With the local handle, the processes can access their local data in the normal manner using calls to kpds_get_data. Also, the processes can later redefine the topology and distribution attributes of the global data handle. Then, when the kpds_distribute call is made, data is redistributed within the process group according to the new distribution. This is an expensive operation and should not be done often.

Distributed output data objects are fairly similar to input data objects, but with a few differences. Since the data does not previously exist, the kpds_open_output_object only returns a handle to a global view of the output data; there is obviously no need for coordination with other process groups at this point. Also, since output objects have no data associated with them, the kpds_distribute function does nothing more than return a local handle describing the output data space for each process. However, the kpds_close_object is very important, as it is the corresponding half of the kpds_open_input_object call of another process group. When both of these functions have been called, the intercommunicator communication described earlier takes place.

4. COMPILED WORKSPACES APPROACH

Cantata currently operates in what could be called a batch mode, where each box in the workspace represents a single program that has been compiled for the same machine architecture that Cantata runs on. Running a workspace is accomplished internally by simply starting up a number of programs, one for each module. Not only is this gravely inefficient, but it has the potential to "bog" down the machine. Since each module in a Cantata workspace represents a single program, this also leads to a noticeable amount of overhead because Cantata must navigate the internal data structures of the module representations within Khoros in order to determine input and output links as well as user specified arguments for the individual programs. In addition, there is explicit communication defined between connected modules, accomplished through shared memory, files, or some other means. Moreover, many of the modules, such as arithmetic operations and some other simple filter designs, perform such a small amount of work that the overhead of executing a complete program is unjustified. Apropos portability, since each module is pre-compiled, it is infeasible to run the workspace with Cantata on any machine other than one of the same architecture on which the modules were compiled.

Consider the example Cantata workspace shown in Figure 3. Each program represented by a module simply checks correctness of command line arguments, initializes Khoros, sets up some data structures and calls a library routine to do the actual operation. A better approach would be to combine sets of modules into groups that would be run as a single program, doing enough work to justify the overhead of process startup. For example, the Sinusoid and FFT modules could be combined into a single program, as well as the Low-Pass, Multiply, and IFFT modules. Then these programs, along with the 2D Plot module program would be written to disk for later compilation and execution. We will refer to this process as compiling a workspace. One would not wish to combine the Low-Pass with either the Sinusoid or FFT modules, since the user has indicated that these are independent, and thus opportunities for parallelism remain if they are kept separate. The implementation of this involves using graph theory to analyze a Cantata workspace in order to identify separate pieces. Grouping modules together can also be accomplished manually by writing custom programs that call the various Khoros libraries, essentially combining many modules into single programs. However, Khoros has a rather steep learning curve from the programmer's perspective, and it would be much better if this process was automated within Cantata.

Therefore there are two major reasons for using a compiled approach to DSP development:

- efficiency of execution and reduction in overhead
- improved portability and resource utilization

However, it would not be correct to support solely the compiled workspace approach, since the current approach is much better for application development. The user can visually construct applications, debug them by using the pre-compiled modules, and then compile the workspaces after reaching a sufficient satisfaction level. Also, compiled workspaces would not be suitable for "one-shot" applications, where it would be pointless to go through the trouble of compilation.

The integration of MPI with Khoros also becomes more efficient with the use of compiled workspaces because of the removal of unnecessary communication. For example, consider again Figure 3. If the Low-Pass and Multiply modules are grouped together within the same program, and both modules have the same data distribution, then the explicit communication link between them has been transformed into an implicit communication link, and no physical data movement is required. With the previous approach involving intercommunicators to communicate between the separate programs, even if the two modules were running on the same processors, there is no guarantee

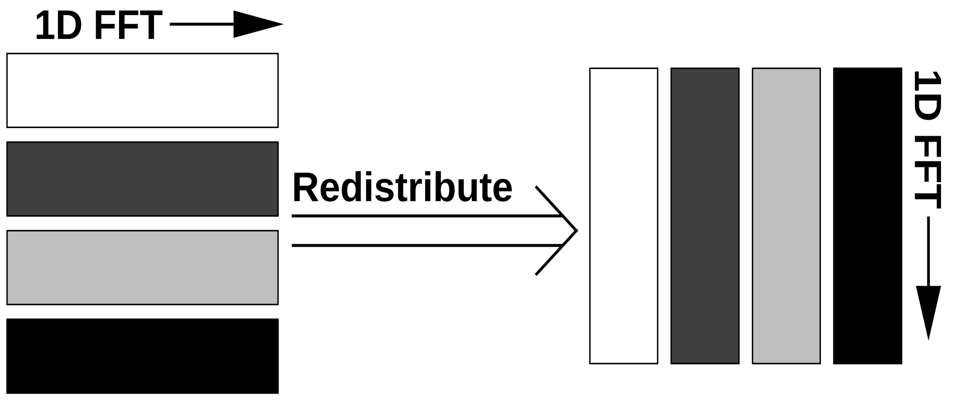


Figure 12. Illustration of redistribution algorithm for computing a parallel FFT.

that each processor has the same piece of data, and at the very least, the data must be copied between process address spaces.

5. DSP ALGORITHMS IN THE KHOROS AND MPI INTEGRATED ENVIRONMENT

In this section we consider how to parallelize two DSP operations (2-D FFT and time-domain convolution) using the integrated Khoros and MPI system described in Section 3.

5.1. Fast Fourier Transform

This section gives an example of how to perform a 2D FFT of an input matrix. The algorithm used is a standard 1D FFT of all rows, followed by a 1D FFT of all columns.

First, we need to open a handle to the input object:

```
    in_matrix = kpds_open_input_object ("mpi=1");
```

For this example, we wish to start out with a two dimensional processor topology, and with our matrix distributed only along the height dimension. Therefore we need to change the default values of KPDS_VIRT_TOPOL and KPDS_DATA_DIST:

```
      kpds_get_attribute (in_matrix,
        KPDS_VIRT_TOPOL,
        &n, &d, &d, &d, &d);

      kpds_set_attribute (in_matrix,
        KPDS_VIRT_TOPOL,
        1, n, 1, 1, 1);

      kpds_set_attribute (in_matrix,
        KPDS_DATA_DIST,
        KALL,KLINEAR,KALL,KALL,KALL
```

The kpds_get_attribute retrieves the number of processes, n. Now, each process must call the kpds_distribute call for in_matrix:

```
      loc_matrix = kpds_distribute (in_matrix);
```

to obtain the handle to the local address space.

First we perform multiple 1D FFTs over the rows of our matrix. Each process uses the kpds_get_data routine with the loc_matrix handle to access one row of local data at a time. We then reset the attributes of the global handle to the matrix in order to specify a width distribution, and redistribute the data:

```
      kpds_set_attribute (in_matrix,
            KPDS_VIRT_TOPOL,
             n, 1, 1, 1, 1);
      kpds_set_attribute (in_matrix,
             KPDS_DATA_DIST,
             KLINEAR,KALL,KALL,KALL,KALL);
      loc_matrix = kpds_distribute (in_matrix);
```

Each process now accesses a column of local data at a time, essentially resulting in a transpose of the matrix. We can then calculate 1D FFTs on the columns.

At this point, the 2D FFT is complete and data needs to be output perhaps to a file, or another group of MPI processes. In order to do this, each process must create an output object, copy its results to this new object, then close the object.

```
   out_obj = kpds_open_output_object ("mpi=2");
   kpds_copy_object (loc_matrix, out_obj);
   kpds_close_object (out_obj);
```

5.1. Convolution

This section demonstrates on approach to parallelizing the convolution of a time domain input signal with an impulse response of a filter. Figure 13 illustrates the method used. Each process receives only part of the time domain signal, and all of the impulse response. A local convolution is performed on each process, followed by a reduction to a root process for the final result.

First, we need to open a handle to the input objects:

```
   signal   = kpds_open_input_object ("mpi=1");
   iresponse = kpds_open_input_object ("mpi=2");
```

We want a one dimensional processor topology with the impulse response replicated on all processes and the time domain signal distributed linearly. Most of the default values are correct for the convolution problem, and we only need to set the distribution for the input signal:

```
      kpds_set_attribute (signal,
             KPDS_DATA_DIST,
             KLINEAR,KALL,KALL,KALL,KALL);
```

Now, each process must call the kpds_distribute call for both signal and iresponse:

```
      loc_signal = kpds_distribute (signal);
      loc_iresponse = kpds_distribute (iresponse);
```

The application can now access the distributed data through loc_signal and loc_iresponse (through the use

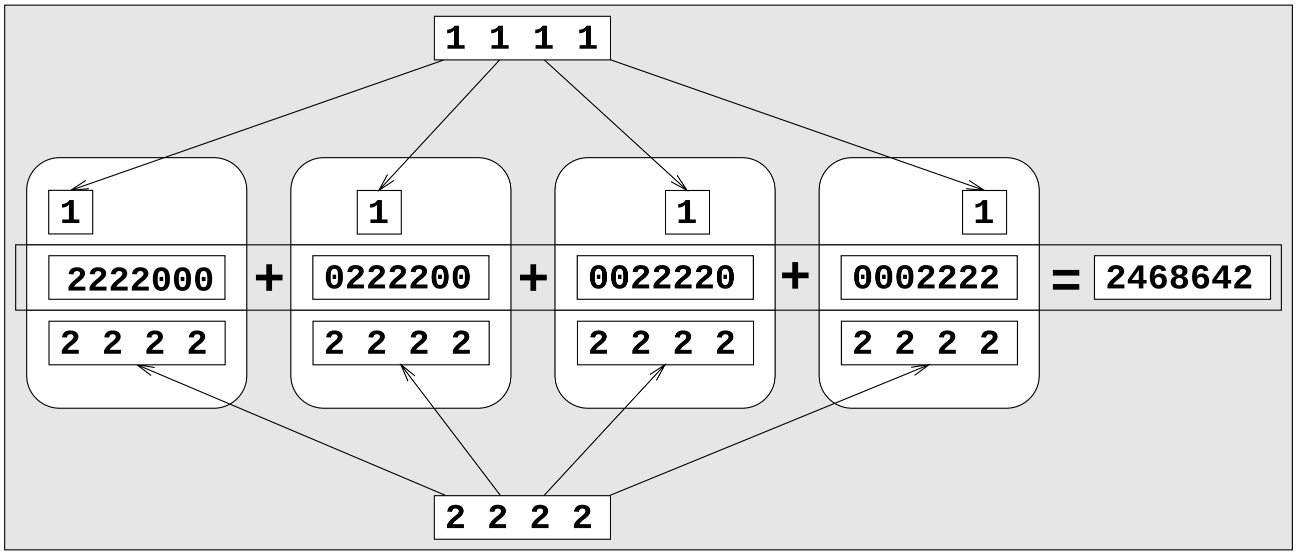


Figure 13. One approach to parallelizing a convolution operation.

of kpds_get_data), and compute the local convolution. This is then reduced to the root node:

```
MPI_Reduce (local_output, result, size,
    mpi_datatype,0,MPI_SUM
    KMPI_COMM_WORLD);
```

The root node creates a polymorphic data object for the result it received, and subsequently outputs and closes the object:

```
out_obj = kpds_open_output_object (“mpi=3”);
kpds_put_data(out_obj, KPDS_VALUE_VECTOR,
     (kaddr) result);
kpds_close_object (out_obj);
```

6. RELATED WORK

Prior to MPI, there have been many popular message passing systems. These include Express[21], PVM[8,5,4], Chimp[11], Vertex (N-Cube)[20], Zipcode[32], NX (Intel)[22], P4, and others. MPI was chosen over any of these as the basis of this work due to its increasing popularity and the fact that it’s a standard. Many of the other message passing systems ceased active development (e.g., Zipcode, P4, NX, Chimp) after the MPI standard was introduced. Another key reason for the choice of MPI was our intimate familiarity with it as it is a large part of our ongoing research.

There are quite a few visual environments for DSP other than Khoros that could have been chosen. Vendors such as Jovian Systems (Pegasus Parallel Processing Design Environment)[16] and Axiom Technology, Inc. (DataFloMP)[1] have developed visual environments for DSP and parallel DSP in particular; however, these systems are proprietary with no general availability to source code. Ptolemy[23] is a freely available system developed at the University of California, Berkeley, focusing on design methodology for digital signal processing and real-time systems. Ptolemy was a viable alternative to Khoros.

One of the main components of this work was designing an interface to Khoros’ polymorphic data model that supported data distributions. Most previous work on data distribution independence has concentrated on data parallel libraries[2,3,27,28,7], is at a lower level of abstraction than the interface we have presented here, and are not as general as the Khoros polymorphic model.

7. SUMMARY

Observations about Khoros:

- Operations are very fine-grained (flexibility is benefit) and thus must be parallelized at a finer grain that optimal for the message passing paradigm.
- Khoros has inordinate amounts of overhead. More work should be done before running workspaces (as is done in other systems such as Ptolemy).

- The current design of Khoros is not a good fit with high performance computing.
- The polymorphic data model provides a strong basis for representing parallel data distribution.

Observations about MPI:

- Intercommunicator collective operations have been proposed as a part of the MPI-2 standard. These type of operations fit naturally with pipeline-structured applications.
- The coarse-grain message passing model is not a good fit with Khoros (as it is currently implemented).
- Dynamic process startup will be a part of MPI-2 and should simplify process management.

Status of Implementation:

- Support for MPI program development and process startup is partially integrated into Khoros.
- Setup for intergroup communicators has been implemented; however this has not been integrated into the Khoros polymorphic data model.
- A few simple MPI-based parallel DSP operations have been implemented; however they do not use the proposed polymorphic data model extensions.

REFERENCES

[1] Axiom Technology, Inc. World Wide Web Page. http://www.axiomtech.com/.

[2] Purushotham V. Bangalore. The Data-Distribution-Independent Approach to Scalable Parallel Libraries. Technical report, Mississippi State University _ Dept. of Computer Science, October 1994. Master's Thesis.

[3] Purushotham V. Bangalore, Anthony Skjellum, Chuck Baldwin, and Steven G. Smith. Data-Distribution-Independent, Concurrent Block LU Factorization. In preparation., January 1994.

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto, and J. Walpole. PVM: Experiences, current Status and Future Direction. In Supercomputing'93 Proceedings, pages 765-6, 1993.

[5] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and Debugging in a Heterogeneous Environment. IEEE Computer, 26(6):88-95, June 1993.

[6] Patrick Bridges, Nathan Doss, William Gropp, Edward Karrels, Ewing Lusk, and Anthony Skjellum. Users guide to mpich, a Portable Implementation of MPI, 1994. MSU/Argonne Joint Documentation.

[7] E. F. Van de Velde. Data Redistribution and Concurrency. Parallel Computing, 16, December 1990.

[8] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM Framework Supports Heterogeneous Network Computing. Computers in Physics, 7(2):166-75, April 1993.

[9] Nathan Doss, William Gropp, Ewing Lusk, and Anthony Skjellum. A Model Implementation of MPI. Technical Report MCS-P393-1193, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, 1994.

[10] Nathan E. Doss. MPI World Wide Web Page. http://www.erc.msstate.edu/mpi/.

[11] Edinburgh Parallel Computing Centre, University of Edinburgh. CHIMP Concepts, June 1991.

[12] Paul M. Embree. C Algorithms for Real-Time DSP. Prentice Hall, Inc., 1995.

[13] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report Computer Science Department Technical Report CS-94-230, University of Tennessee, Knoxville, TN, May 5 1994. To appear in the International Journal of

Supercomputing Applications, Volume 8, Number 3/4, 1994.

[14] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. Solving Problems on Concurrent Processors: Volume I, General Techniques and Regular Problems, chapter The Fast Fourier Transform, pages 187-200. Prentice Hall, 1988.

[15] William Gropp, Ewing Lusk, and Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, 1994. The example programs from the book are available from ftp://info.mcs.anl.gov/pub/mpi/using.

[16] Jovian Systems World Wide Web Page. http://www.jovian.com/jovian/.

[17] Khoral Research, Inc. Khoros Manual, 1994.

[18] Khoral Research, Inc. World Wide Web Page. http://www.khoral.com/.

[19] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. Introduction to Parallel Computing: Design and Analysis of Algorithms, chapter Fast Fourier Transform, pages 377-406. Benjamin/Cummings Publishing Company, Inc., 1994.

[20] nCUBE Corporation. nCUBE 2 Programmers Guide, r2.0, December 1990.

[21] Parasoft Corporation, Pasadena, CA. Express User's Guide, version 3.2.5 edition, 1992.

[22] Paul Pierce. The NX/2 Operating System. In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, pages 384-390. ACM Press, 1988.

[23] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software Synthesis for DSP Using Ptolemy. Jounal on VLSI Signal Processing, 9(1):7-21, January 1995.

[24] John G. Proakis. Digital Signal Processing: Principles, Algorithms, and Applications. Macmillan, 2 edition, 1992.

[25] Rasure and Kubica. The Khoros Application Development Environment. In H.I. Christensen and J.L. Crowley, editors, Experimental Environments for Computer Vision and Image Processing. World Scientific, 1994.

[26] A. Skjellum, S. Ashby, P. Brown, M. Dorr, and A. Hindmarsh. The Multicomputer Toolbox. In G. L. Struble et al., editors, Laboratory Directed Research and Development FY91 - LLNL, pages 24-26. Lawrence Livermore National Laboratory, August 1992. UCRL-53689-91 (Rev 1).

[27] A. Skjellum and C. Baldwin. The Multicomputer Toolbox: Scalable Parallel Libraries for Large-Scale Concurrent Applications. Technical Report UCRL-JC-109251, Lawrence Livermore National Laboratory, December 1991.

[28] Anthony Skjellum. The Multicomputer Toolbox: Current and Future Directions. In Anthony Skjellum and Donna S. Reese, editors, Proceedings of the Scalable Parallel Libraries Conference. IEEE Computer Society Press, October 1993.

[29] Anthony Skjellum, Nathan Doss, and Kishore Viswanathan. Inter-Communicator Extensions to MPI in the MPIX (MPI eXtension) Library, July 1994. To be submitted to Parallel Computing.

[30] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, Proceedings of the Scalable Parallel Libraries Conference, pages 166-173. IEEE Computer Society Press, October 1993.

[31] Anthony Skjellum, Nathan E. Doss, and Purushotham V. Bangalore. Writing Libraries in MPI. In Anthony Skjellum and Donna S. Reese, editors, Proceedings of the Scalable Parallel Libraries Conference, pages 166-173. IEEE Computer Society Press, October 1993.

[32] Anthony Skjellum, Steven G. Smith, Nathan E. Doss, Alvin P. Leung, and Manfred Morari. The Design and Evolution of Zipcode. Parallel Computing, 20(4):565-596, April 1994.

[33] Young, Argiro, and Kubica. Cantata: Visual Programming Environment for the Khoros System. Computer Graphics, 29(2):22-24, May 1995.