C.2 Distributed-memory Dense BLAS

This document summarizes the discussions that took place during the meetings of the BLAS Technical Forum concerning the distributed-memory BLAS. The committee did not reach an agreement on how to specify an interface for such a set of routines, however it was felt that this document should keep a record of those discussions.

There has been much interest in the past few years in developing versions of the BLAS for distributed-memory computers [50, 28, 3, 29, 13, 17, 15, 18, 8, 51]. Some of this research proposed parallelizing the BLAS [45, 50, 17, 15, 11, 18, 49, 51], and some implemented a few important BLAS routines [50, 43, 17, 15, 11, 18, 49, 51], such as matrix-matrix multiplication [31, 5, 36, 44, 4, 16, 52] or triangular system solve [34, 40, 41, 10].

Based on this research work, it was agreed that an interface for the distributed-memory BLAS should have the following features:

- The calling sequence definitions should be simple and similar in all targeted programming languages.
- The interface should be effective for the developement of large and high-quality dense linear algebra software for distributed-memory computers.
- The interface should permit broad functionality to enable, facilitate and encourage the development of current and related research projects.

The main advantages of establishing a distributed-memory dense BLAS standard are portability and ease-of-use. In a distributed-memory environment in which the higher level routines and/or abstractions are built upon lower level message-passing and computational routines the benefits of standardization are particularly apparent. Furthermore, the definition of distributed-memory dense basic linear algebra subprograms provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

The goal of the distributed-memory dense BLAS interface simply stated should be to develop a widely used standard for writing message-passing programs performing dense basic linear algebra operations. As such the interface should establish a practical, portable, efficient and flexible standard for distributed-memory dense basic linear algebra operations.

A complete list of goals follows.

- Design an Application Programming Interface (API) (not necessarily for compilers or a system implementation library) well suited for distributed-memory dense basic linear algebra computations.
- Allow efficient communication and computation: minimizing communication startup overhead and volume, while maximizing load balance and local computational performance.
- Allow for re-use of existing message-passing interface standard [30] as well as local basic linear algebra computational kernels such as the *de facto* standard BLAS.
- Allow for implementations that can be used in a heterogeneous environment.
- Define an interface that is not too different from current practice and provide extensions that allow greater flexibility.

- Define an interface that can be implemented on many vendors' platforms, with no significant changes in the underlying system software.
- Semantics of the interface should be language-and data-distribution independent.
- The interface should be designed to allow for thread-safety.

Such a distribute-memory BLAS standard should be intended for use by all those who want to write portable programs performing dense linear algebra operations in Fortran 77, Fortran 90, High Performance Fortran (HPF), C or C++. This includes individual application programmers, developers of dense linear algebra software designed to run on parallel machines, and creators of computational environment and tools. In order to be attrac- tive to this wide audience, the standard must provide a simple, easy-to-use interface for the basic user while not semantically precluding the high-performance computation and communication operations available on advanced computers.

The attractiveness of the distributed-memory dense BLAS at least partially stems from its wide portability as well as the common occurence of dense linear algebra operations in numerical simulations. These programs may run on distributed-memory multiprocessors, networks or clusters of workstations, and combinations of all of these. In addition, shared-memory implementations are possible. The message passing paradigm will not be made obsolete by architectures combining the shared- and distributed-memory views, or by increases in network speeds. It thus should be both possible and useful to implement such a standard on a great variety of machines, including those "machines", parallel or not, connected by a communication network.

The distributed-memory dense BLAS interface should provide many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. Thus, we expect that native, high-performance implementations of this interface could be provided on such machines. At the same time, implementations of such a standard on top of MPI or PVM will provide portability to workstation clusters and heterogeneous networks of workstations.

During the discussions it was agreed that the distributed-memory BLAS should include

- A set of basic dense linear algebra computational operations
- Data-redistribution operations
- Environmental management and inquiry
- Bindings for various widely used programming languages, including high level languages such as High Performance Fortran (HPF)

The distributed-memory dense BLAS interface should specify routines that operate on in-core dense matrices. On entry, these routines assume that the data has been distributed on the processors according to a specific data decomposition scheme that dictates the local storage of the data when it resides in the processors' memory. The data layout information as well as the local storage scheme for these different matrix operands is conveyed to the routines via a descriptor that could be a simple array of integers. The standard could mandate that the first entry of this array identifies the type of the descriptor, i.e., the data distribution scheme it describes. This allows to specify the distributed-memory dense BLAS interface indepen- dently from the data distribution.

The distributed-memory dense BLAS are executed by *processes*, rather than physical processors. In general there may be several processes running on a processor, in which case it is assumed that the runtime system handles the scheduling of processes. In the absence of such a runtime system, the distributed-memory dense BLAS assume one process per processor. A **process** is defined as a basic unit or thread of execution that minimally includes a stack, registers, and memory. Multiple processes may share a physical processor. The term processor refers to the actual hardware. Each process is treated as if it were a processor: a process executing a program or subprogram calling the distributed-memory dense BLAS must exist for the lifetime of the program's or subprogram's run. Its execution should affect other processes' execution only through the use of message-passing calls. With this in mind, the term process is used in all sections of this chapter unless otherwise specified.

The distributed-memory dense BLAS are thus executed by a collection of processes, that are enclosed in a **communication context**. Similarly, a communication context or simply context is associated with every global matrix. The use of a context provides the ability to have separate "universes" of message passing. This means that a collection of processes can safely communicate even if other (possibly overlapping) sets of processes are also communicating. Thus, a context is a powerful mechanism for avoiding unintentional nondeterminism in message passing and provides support for the design of safe, modular software libraries. In MPI, this concept is referred to as a *communicator*.

A context partitions the communication space. A message sent from one context cannot be received in another context. The use of separate communication contexts by distinct libraries (or distinct library routine invocations) insulates communication internal to a specific library routine from external communication that may be going on within the user's program.

In most respects, the terms *process collection* and *context* can be used interchangeably. For example, one may say that an operation is performed "in context X" or "in process collection X". The slight difference here is that the user may define two identical sets of processes (say, two 1×3 process grids, both of which use processes 0, 1, and 2), but each will be enclosed in its own context, so that they are distinct in operation, even though they are indistinguishable from a process collection standpoint.

Another example of the use of context might be to define a normal two-dimensional process grid within which most computation takes place. However, in certain portions of the code it may be more convenient to access the processes as a one-dimensional process grid, whereas at other times one may wish, for instance, to share information among nearest neighbors. In such cases, one will want each process to have access to three contexts: the two-dimensional process grid, the one-dimensional process grid, and a small process grid that contains the process and its nearest neighbors. Therefore, communication contexts allow one to

- create arbitrary groups of processes,
- create an indeterminate number of overlapping and/or disjoint collections of processes, and
- isolate a set of processes so that communication interference does not occur.

A distributed-memory dense BLAS function should create a grid of processes and its enclosing context. This routine returns a context handle, which is a simple integer, assigned by the message-passing library used by a given implementation of the distributed-memory dense BLAS to identify the commu- nication context. Subsequent distributed-memory dense BLAS will be passed these handles, which allow to determine from which context/process collec- tion a routine is being called. The user *should never alter or change these handles;* they are opaque data objects that are only meaningful for the distributed-memory dense BLAS routines.

A defined context consumes resources. It is therefore advisable to release contexts when they are no longer needed. When the entire distributed-memory dense BLAS system is shut down, all outstanding contexts are automatically freed.

Some systems, such as MPI, supply their own version of context. For portability reasons, one thus cannot assume that the communication contexts used by the user's program are usable by the distributed-memory dense BLAS. Therefore, the following interface allows to form a distributed-memory dense BLAS context in reference to a user's context.

The standard could mandate that the second entry of each descriptor is set to the value of the communication context identifying the collection of processes onto which the data is distributed.

The routines of the distributed-memory dense BLAS could require that all global data (vectors or matrices) be distributed across the processes prior to invoking the routines. The data layout and local storage scheme are specified by a particular implementation, but are strictly speaking not part of the standard. Global data is mapped to the local memories of processes assuming an implementation-dependent data distribution scheme.

A **descriptor** is associated with each global array. This descriptor stores the information required to establish the mapping between each global array entry and its corresponding process and memory location. Array descriptors corresponding to distinct layouts are differentiated by their first entry called the type of the descriptor. The data residing in the processes' memory is specified by a pointer. Splitting the local data from the layout's description allows to specify language-dependent interfaces for programming languages providing only simple data structures such as Fortran 77 without affecting the functionality or ease-of-use of the interface.

Most of the distributed-memory dense BLAS should operate within the same communication context, i.e., all distributed operands should be distributed on the same process grid. These operations are said to be **intra-context** operations. Only, very few distributed-memory dense BLAS perform inter-context operations, in which case this feature should clearly be mentioned in the procedure functionality.

Furthermore, all distributed operands involved in the operation should be distributed accordingly to the same decomposition scheme. In other words, the type entry of each descriptor must be equal.

All distributed-memory dense BLAS operations could be **collective**, that is, all processes in the context need to invoke the procedure even if certain processes are not involved in the operation. This situation may happen for example when some processes don't own any data to be operated on.

The distributed-memory dense BLAS manages **system memory** that is used for buffering messages and for storing internal representations of various distributed-memory dense BLAS objects such as communication contexts, local arrays of data, etc. This memory is not directly accessible to the user, and objects stored there are either private or opaque: their size and shape is not visible to the user. Opaque objects are accessed via **handles**, which exist in user space. Private objects cannot be accessed by the user, and are allocated and released within the same routine. distributed-memory dense BLAS that operate on opaque objects are passed handle arguments to access these objects. In addition to their use by distributed-memory dense BLAS calls for object access, handles can participate in assignment and comparisons.

In Fortran and C, all handles have type integer and correspond to the same objects. This means that the user's program can pass a C handle to a Fortran subprogram and conversely, such that in both languages the handle refers to the same object.

Opaque objects are allocated and deallocated by calls that are specific to each object type. These are listed in the sections where the object are described. The calls accept a handle argument of matching type. In an allocate call this is an ouput argument that returns a valid reference to the object. In a call to deallocate this is an input/output argument which returns with an "invalid handle" constant for each object type. Comparisons to this constant are used to test for validity of the handle.

A call to deallocate invalidates the handle and marks the object for deallocation. The object is not accesible to the user after the call. However, distributed-memory dense BLAS need not deallocate the object immediately. Any operation pending (at the time of the deallocate) that involves this object will complete normally; the object will be deallocated afterwards.

An opaque object and its handle are significant only at the process where the object was created, and cannot be transferred to another process.

This design hides the internal representation used for distributed-memory dense BLAS internal data structures, thus allowing similar calls in C and Fortran. It also avoids conflicts with the typing rules in these languages, and easily allows for future extensions of functionality. Note that the objects handles defined in the distributed-memory dense BLAS are exclusively used by the underlying message passing library. The user data itself remains directly accessible in the user's program.

The explicit separating of handles in user space, objects in system space, allows space-reclaiming, deallocation calls to be made at appropriate points in the user program. If the opaque objects were in user space, one would have to be very careful not to go out of scope before any pending operation requiring that object completed. The specified design allows an object to be marked for deallocation, the user program can then go out of scope, and the object itself still persists until any pending operations are complete. Again such a design cannot be applied in general to the user's data.

The requirement that handles support assignment/comparison is made since such operations are common. This restricts the domain of possible implementations. Moreover, such a design has been adopted by most message passing library interfaces such as the MPI.

The intended semantics of opaque objects is that each opaque object is separate from each other; each call to allocate such an object copies copies all the information required for that object. Implementations may avoid excessive copying by substituting referencing for copying. For example, a derived datatype may contain references to its components, rather then copies of its components. In such cases, the implementation must maintain reference counts, and allocate and deallocate objects such that the visible is as if the objects were copied. Such a design is particularly suitable for communication contexts, because the amount of data one has to keep track of is small. However, applying the same concept to the user's data forces the introduction of routines to manage logical templates, adding complexity, and was therefore ruled out.

There are several important language bindings issues not addressed by this document. This section does not discuss the interoperability of message passing between languages. It is fully expected that many implementations should have such features.

A descriptor is associated with each distributed matrix. The entries of the descriptor uniquely determine the mapping of the matrix entries onto the local processes' memories. Since vectors may be seen as a special case of distributed matrices or proper submatrices, the larger scheme just defined encompasses their description as well.

The local storage convention of the distributed matrix operands in every process's memory does not need to be specified by the standard. It is however recommended that convenient data structure are chosen by a given implementation allowing to rely on the sequential BLAS to perform the local computations within a process.

The distributed-memory dense BLAS should not provide mechanisms for dealing with failures in the communication and computation systems. If the distributed-memory dense BLAS is built on an unreliable underlying mechanism, then it is the job of the implementor(s) of the distributed-memory dense BLAS subsystem to insulate the user from this unreliability, or to reflect unrecoverable errors as failures. Whenever possible, such failures will be reflected as errors in the relevant communication or computation call.

Of course, distributed-memory dense BLAS programs can still be erroneous. A program error can occur when a distributed-memory dense BLAS routine is called with an invalid value for any of its arguments. The routine must report this fact and terminate the execution of the program. Each routine, on detecting an error, should call a common error-handling routine, passing to it the current communication context, the name of the routine and the number of the first argument that is in error. For efficiency purposes, the distributed-memory dense BLAS only perform a local validity check of their argument list. If an error is detected in at least one process of the current context, the program execution is stopped.

A global validity check of the input arguments passed to a distributed-memory dense BLAS routine must be performed in the user-level calling procedure. To demonstrate the need and cost of global checking, as well as the reason why this type of checking should not be performed in the distributed-memory dense BLAS, consider the following example: the value of a global input argument is legal but differs from one process to another. The results are unpredictable. In order to detect this kind of error situation, a synchronization point would be necessary, which may result in a significant performance degradation. Since every process must call the same routine to perform the desired operation successfully, it is natural and safe to restrict somewhat the amount of checking operations performed in the distributed-memory dense BLAS routines.

Specialized implementations may call system-specific exception-handling facilities, either via an auxiliary routine or directly from the routine. In addition, the testing programs can take advantage of this exception-handling mechanism by simulating specific erroneous input argument lists and then verifying that particular errors are correctly detected.

Resource errors can also occur when a program exceeds the amount of available system resources. The occurence of this type of error depends on the amount of available resources in the system and the resource allocation mechanism used; this may differ from system to system. A high-quality implementation will provide generous limits on the important resources so as to alleviate the portability problem this represents.