Common Compiler and Data Movement Interface Specification

Parallel Compiler/Runtime Consortium: Task 6

November 18, 1996

1 Overview

The Message Passing Interface standard [3] is a powerful, flexible, portable generic solution to the requirements of general-purpose message-passing codes. MPI's signal achievement was the compilation of most common communications functionality under one roof, for the convenience of the parallel programmer.

Performance vs Flexibility Nonetheless, the design of software standards inevitably trades away the high performance of a tailored solution in favor of the portability and flexibility of a generic solution. The combined performance and portability requirements of a specific family of applications may be better met by considering only a **subset** of a more generic standard.

One such family of applications are the parallel runtime systems targeted by major parallel compiler environments for languages such as HPF ([1, 5, 2] and pC++ ([4]). The node codes created by these compilers are, by and large, variants of the single-program, multiple-node (SPMD) programming model. Their runtime requirements for communication tend to be regular, to follow idiomatic patterns such as the owner-computes rule, nearest-neighbor shift, and reduction.

These systems each offer significantly different programming models to the compiler and enduser: regular arrays and array sections from Adlib and F90D, irregular arrays from Parti/CHAOS, and parallel element collections from pC++. Nonetheless, they implement enough common functionality internally to share some runtime communication requirements. As a result, we speculated that a compiler data movement interface for these systems could omit large parts of the MPI standard, and focus instead on the MPI subset required to implement the standard idioms of distributed regular- and irregular-array runtime behavior.

Organization In the following report, we describe some empirical results comparing the specific common runtime requirements of four PCRC software systems.

We then summarize these results in terms of a "laundry list" of common communications functionality, and a summary of superfluous MPI features which can safely be omitted from a common communications interface. Finally, we summarize these results in terms of PCCMPI: a Parallel Compiler Common MPI standard.

2 Survey: PCRC Runtime Applications

We collected all the MPI constant and function call data from four PCRC applications: Kivanc Dincer's MPI CHAOS port, Xiaoming Li's F90D MPI runtime port, Bryan Carpenter's MPI Fortran support, and Dennis Gannon's MPI pC++ runtime libraries. We then extracted raw coverage data (reproduced in Appendix A) for the MPI functions and constants, and analyzed them to uncover common requirements. In addition to the core MPI functionality shared by all these systems, two "annexes" emerged. Each of these annexes represents an additional group of MPI functionality which was central to the success of one survey application, but excluded by the other three.

2.1 F90D MPI Runtime System

The Fortran 90D runtime system [5], ported to MPI by Xiaoming Li of NPAC, provides runtime support for the Fortran 90D compiler [6]. The D compiler generates Fortran 77 SPMD node codes, which call the D runtime system to allocate local arrays and carry out collective communication over the global arrays they represent.

2.2 MPI CHAOS Runtime System

The MPI port of Parti/CHAOS [2], by Kivanc Dincer of NPAC, augments the basic F90D runtime system. CHAOS supports runtime scheduling of irregular computations by implementing the inspector/executor model.

2.3 MPI Adlib Runtime System

The Adlib runtime system ([1]), developed by Bryan Carpenter of NPAC, provides a set of abstract data types representing parallel arrays. Written in C++, Adlib provides the runtime support for the Subset HPF compiler system. As in HPF, an Adlib array can be distributed over a logical *process array,* which the library maps transparently to available physical processors. Adlib also provides control abstractions for data-parallel array traversal, as well as common collective communication patterns such as remap, gather-scatter, and arithmetic reduction.

2.4 MPI pC++ Runtime System

The pC++ system [4], developed by Dennis Gannon et al of Indiana University, extends C++ to support data-parallel operations over collections of objects. These collections can then be aligned and distributed over the memory hierarchy of the parallel machine. pC++ also includes a mechanism for encapsulating SPMD-style computation in a thread-based computing model. The MPI-based implementation of the pC++ runtime system allows processor threads to access remote elements from any processor via the pC++ program's shared name space. On behalf of the pC++ user, the MPI runtime system handles allocation of collection classes, the management of element accesses, and termination of parallel collection operations.

3 Analysis of Coverage Data

For each of the four parallel runtime systems, we derived lists of calls to MPI functions. We then compared the usage patterns of each runtime system. Some parts of the MPI specification (the "core") were exercised by the majority of codes; other parts were used by only one client. These we divided into "spontaneous" uses of MPI functionality (those which could potentially be avoided by using a more standard call) and "annexes" of functionality (clearly systematic usage patterns for MPI functions unused by others). Finally, large regions of MPI went unused by any runtime system.

3.1 Core MPI Functionality

A handful of MPI functions were used by at least three of the four surveyed systems. As might be expected, they cover the basics of MPI operation: initialization, extracting communicator information, and (of course) basic message passing.

- **MPI_Init, MPI_Finalize.** These functions initialize and terminate the MPI execution environment, respectively; they were used by all four runtime systems. F90D also used the **MPI_INITIALIZED** test to make sure that `MPI_INIT` has been properly called.
- **MPI_Get_count, MPI_Comm_size, MPI_Comm_rank.** The first function returns the number of "top level" elements, the second finds the size of a group associated with a communicator, and the last determines the rank of the caller in the communicator. Collectively, these three functions serve to define the bounds of the computation space and support global-to-local and logical-to-physical mappings.
- **MPI_Send, MPI_Recv.** These basic, blocking message passing functions were used nearly universally, although Adlib uses the asynchronous versions instead throughout.
- **MPI_Irecv.** Three of four systems also called the nonblocking (asynchronous) version of the receive function, allowing the overlap of communication with computation. Perhaps due to the prevalence of the owner computes model, or the idioms of distributed data-parallel computation, the asynchronous send counterpart (**MPI_Isend**) was used only rarely, by Adlib. The nonblocking test for message arrival, **MPI_Iprobe,** was used by both Adlib and pC++.

3.2 Non-core MPI Functionality

In addition, the following MPI functions were used by at least two of the four surveyed systems.

- **MPI_Abort.** This function brings the MPI execution environment to an abrupt halt. Error and exception handling has not been a primary focus of most dataparallel codes. Adlib and pC++ both use `MPI_Abort` to escape from unexpected situations.
- **MPI_Barrier.** This routine blocks all processes in a barrier synchronization. The pC++ and CHAOS runtime systems used explicit barrier synchronization, while Adlib and F90D/RT relied instead on the completion of regular, all-to-all synchronous communication patterns to create natural collective barriers.

- **MPI_COMM_RANK, MPI_COMM_SIZE, MPI_COMM_CREATE, MPI_COMM_GROUP.** The first two functions determine the rank of the calling process in the communicator, and the size of the group associated with the communicator, respectively. The `_CREATE` and `_GROUP` functions create a communicator and access the associated group. Both the F90D and pC++ systems used these functions, creating new communicators and groups to help implement array structures; by contrast, Adlib relied on its own internal data structures to track these values.

- **MPI_Wait.** Adlib and pC++ used this routine, which waits for a send or receive to complete. Adlib also used **MPI_Waitany**, which waits for any specified send or receive to complete.

- **MPI_Test.** CHAOS and pC++ used this routine, which tests for the completion of a send or receive without waiting.[1]

[1] The pC++ runtime system also uses **MPI_Testany,** which could instead be implemented directly by the compiler as a loop over outstanding requests, performing a nonblocking `MPI_Test` operation on each, and escaping from the loop on first success.

3.3 Systematic, unique usage patterns.

These sets of MPI functions were called by only one of the four surveyed systems, but clearly represent a style of usage which cannot be "patched around" to recast the code in terms of more standard functions. We think of these as alternative "annexes" of the PCCMPI standard which are clearly useful, but not universally applicable.

- **MPI_CART_CREATE, MPI_CART_COORDS, MPI_CART_GET, MPI_CART_RANK, MPI_CART_SHIFT.** These functions all support use of relatively high-level Cartesian topology information that can be associated with a communicator. Only the F90D runtime system uses these functions. Adlib, by comparison, uses its own internal data structures to keep track of more-or-less equivalent information.
- **MPI_Type_commit, MPI_Type_contiguous, MPI_Type_free, MPI_Type_hindexed, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_struct.** These MPI routines allow the creation and management of indexed datatypes (structures, vectors) with offsets in bytes. These are used extensively by Adlib, and only by Adlib.
- **MPI_Buffer_attach, MPI_Bsend.** User-defined buffer space was only an issue in pC++, the runtime system whose interface is closest to the enduser (programmer). The first routine allows the programmer to attach a user-defined buffer for sending, and then execute the basic send with this user-specified buffering.

3.4 Possibly redundant MPI functionality.

Still other MPI functions were called by only one of the four surveyed systems, and could potentially be replaced by more standard strategies based on the more common functions listed in previous sections.

- **MPI_Probe, MPI_Ssend.** Blocking test for a message and basic synchronous send; used only by pC++.
- **MPI_Address.** Gets the location of an address in memory into an `MPI_Aint`. Like the `MPI_Aint` type itself, used only by Adlib.
- **MPI_SENDRECV, MPI_ALLGATHER, MPI_GROUP_INCL.** The first function simultaneously sends and receives a message; the second gathers data from all tasks and delivers it to all tasks. The last one produced a new group by reordering an existing group and taking only the listed members. All three routines are used only by F90D, for specific optimizations. The compiler could potentially replace these calls with lower-level primitives to achieve the same functionality with roughly the same performance.
- **MPI_Testany.** Only pC++ used this routine, which tests for completion of any previously initialized communication in a supplied array of requests. This function could be alternatively implemented directly by the compiler as a loop over outstanding requests, performing an `MPI_Test` operation on each, and escaping from the loop on first success.

3.5 Unused MPI functionality.

The remaining MPI functions never seem to be used by any of the four surveyed runtime systems. This is not to say that they are not useful portions of the MPI specification; just that they were not found to be immediately useful to four parallel runtime system designers working independently.

- **Key-attribute database functions.** That is, `MPI_Attr_*` and `MPI_Keyval_*`.
- **Error handlers.** MPI supports the association of an error handler with each communicator (`MPI_Errhandler_*`), but no runtime system found this useful.
- **Graph topology.** While F90D used Cartesian topologies extensively, the more general Graph topology (`MPI_Graph_*`) went unused by all systems. This is a clear example of a mismatch between an overly general tool (MPI) and the more specific application it supports (distributed array management).
- **General-purpose groups.** While F90D used some basic Group support, the more esoteric `MPI_Group_*` operations (intersection and union, for example) were unused.
- **Intracommunicators.** No system found any use for intracommunicators (`MPI_Intercomm_*`).
- **MPI_Pack, MPI_Unpack.** No system wasted time packing or unpacking datatypes to and from contiguous memory.
- **Ready sends.** Ready sends (`MPI_rsend_*`) went unused.
- **Persistent request handles.** Nobody used `MPI_Start` or `MPI_Startall`.

4 Summary

The results that emerge from our survey of the four runtime systems are not as clear-cut, perhaps, as we had hoped at the start of the project. While all four systems do use a nontrivial core of MPI routines to carry out common functionality, there are still significant differences in the style and level at which they manage their internal affairs.

Most significantly, the F90D runtime system tended to use the high-level, all-to-all communications facilities built into MPI, and then took advantage of the opportunity to associate Cartesian topology information with MPI communicators. By contrast, Adlib used lower-level functions to implement all-to-all communications directly, and used MPI's support for defining new indexed datatypes to represent its local arrays and array sections.

Parallel Compiler Common MPI. We define "core PCCMPI" as the subset of MPI functionality that excludes data descriptor concerns (the debate between Adlib's MPI defined datatypes and F90D's Cartesian communicator topologies). That is, PCCMPI includes **only** those 19 functions listed in sections 3.1 and 3.2, plus perhaps pC++'s 2 user-defined buffering functions from section 3.3. To these 21 functions, then, we optionally add the two data description annexes from section 3.3: Adlib's use of MPI's defined datatypes (7 functions), and F90D's use of MPI's Cartesian topologies (5 functions).

In total, then, PCCMPI consists of 33 routines, out of the 129 defined in the MPI standards. Whether this economy of expression would translate into improved performance depends on the quality of the reimplementation of the PCCMPI communication layer. We can assert that the PCCMPI implementor would benefit from the ability to focus on optimizing a few selected alternative forms of each communication routine, rather than the broad space of routines demanded by symmetry and universality,

Appendix A. MPI Constant and Function Coverage Information

Fn or Constant	Adlib	F90D	CHAOS	pC++
MPI_Finalize	X	X	X	X
MPI_Init	X	X	X	X
MPI_Get_count	X	X	-	X
MPI_Comm_size	X	-	X	X
MPI_Comm_rank	X	X	X	X
MPI_Irecv	X	-	X	X
MPI_Send	-	X	X	X
MPI_Recv	-	X	X	X

Table 1: MPI Functions and Constants used by at least 3 of 4 runtime systems.

Fn or Constant	Adlib	F90D	CHAOS	pC++
MPI_COMM_WORLD	X	X	-	-
MPI_ALLREDUCE	-	X	X	-
MPI_ANY_SOURCE	X	X	-	-
MPI_DOUBLE_PRECISION	-	X	X	-
MPI_INTEGER	-	X	X	-
MPI_MAX	-	X	X	-
MPI_MIN	-	X	X	-
MPI_REAL	-	X	X	-
MPI_SUM	-	X	X	-
MPI_WTIME	-	X	X	-
MPI_Abort	X	-	-	X
MPI_Barrier	-	-	X	X
MPI_Iprobe	-	-	X	X
MPI_Test	-	-	X	X
MPI_Wait	X	-	-	X

Table 2: MPI Functions and Constants used by 2 of the 4 runtime systems.

Fn or Constant	Adlib	F90D	CHAOS	pC++
MPI_2DOUBLE_PRECISION	-	X	-	-
MPI_2INTEGER	-	X	-	-
MPI_2REAL	-	X	-	-
MPI_Address	X	-	-	-
MPI_Aint	X	-	-	-
MPI_all	-	X	-	-
MPI_ALLGATHER	-	X	-	-
MPI_any	-	X	-	-
MPI_ANY_TAG	-	X	-	-
MPI_BAND	-	X	-	-
MPI_BCAST	-	X	-	-
MPI_BOR	-	X	-	-
MPI_BOTTOM	X	-	-	-
MPI_Bsend	-	-	-	X
MPI_Buffer_attach	-	-	-	X
MPI_BXOR	-	X	-	-
MPI_BYTE	-	X	-	-
MPI_CART_COORDS	-	X	-	-
MPI_CART_CREATE	-	X	-	-
MPI_CART_GET	-	X	-	-
MPI_CART_RANK	-	X	-	-
MPI_CART_SHIFT	-	X	-	-
MPI_CHAR	X	-	-	-
MPI_COMM_CREATE	-	X	-	-
MPI_COMM_GROUP	-	X	-	-
MPI_COMM_RANK	-	X	-	X
MPI_COMM_SIZE	-	X	-	X
MPI_COMPLEX	-	X	-	-
MPI_count	-	X	-	-
MPI_Datatype	X	-	-	-
MPI_dotproduct	-	X	-	-
MPI_END	-	X	-	-
MPI_GROUP_INCL	-	X	-	-
MPI_Initialize	-	-	X	-
MPI_INITIALIZED	-	X	-	-

Table 3: MPI Functions and Constants used by only 1 of 4 runtime systems.

Fn or Constant	Adlib	F90D	CHAOS	pC++
MPI_INT	X	-	-	-
MPI_Isend	X	-	-	-
MPI_LAND	-	X	-	-
MPI_LOGICAL	-	X	-	-
MPI_LOR	-	X	-	-
MPI_LXOR	-	X	-	-
MPI_MINLOC	-	X	-	-
MPI_msg	-	-	-	X
MPI_msgEmpty	-	-	-	X
MPI_Probe	-	-	-	X
MPI_PROC_NULL	-	X	-	-
MPI_PROD	-	X	-	-
MPI_Request	X	-	-	X
MPI_REQUEST_NULL	X	-	-	-
MPI_SENDRECV	-	X	-	-
MPI_SOURCE	X	-	-	-
MPI_Ssend	-	-	-	X
MPI_Status	X	-	-	-
MPI_STATUS_SIZE	-	X	-	-
MPI_Testany	-	-	-	X
MPI_Type_commit	X	-	-	-
MPI_Type_contiguous	X	-	-	-
MPI_Type_free	X	-	-	-
MPI_Type_hindexed	X	-	-	-
MPI_Type_hvector	X	-	-	-
MPI_Type_indexed	X	-	-	-
MPI_Type_struct	X	-	-	-
MPI_Waitany	X	-	-	-
MPI_wrapper	-	X	-	-

Table 4: MPI Functions and Constants used by only 1 of 4 runtime systems (cont'd).

References

[1] Bryan Carpenter. Adlib. 1995.

[2] Kivanc Dincer. Parti/CHAOS for MPI. 1995.

[3] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[4] D. Gannon and P. Beckman et al. pC++. 1995.

[5] Xiaoming Li. F90D Runtime System. 1995.

[6] NPAC. The F90D Compiler. 1995.