# Real-Time Extensions to the Message-Passing Interface (MPI)

*Arkady Kanevsky*
*The MITRE Corp.
202 Burlington Rd.
Bedford, MA 01730-1420
e-mail: arkady@mitre.org

*Anthony Skjellum*
†‡Mississippi State University
Engineering Research Center for
Computational Field Simulation
2 Research Blvd.
Starkville, MS 39759
e-mail: tony@erc.msstate.edu

*Anna Rounbehler*
Sky Computers, Inc.
27 Industrial Ave.
Chelmsford, MA 01824
e-mail: anna@sky.com

January 2, 1997

**Abstract**

This article describes the ongoing work of real-time message passing interface (MPI) standardization. Real-time MPI (MPI/RT) provides a consistent set of extensions and, in some cases, restrictions to the high-performance computing Message Passing Interface Standard, emphasizing changes that enable and support real-time communication and are supportable on embedded and other real-time systems.

*This work was supported in part by the U.S. Air Force Electronic Systems Center and performed under MITRE MOIE Project 03977450 of contract F19628-94-C-0001, managed by Rome Laboratory/C3CB.

†This work was supported in part by the U.S. Air Force Rome Laboratory under DARPA Order D350.

‡Corresponding author: 601-325-8435; FAX: 601-325-8997.

# 1 Introduction

The *MPI* standard [3] defines the user interface and functionality for a wide range of message passing capabilities. MPI strives to provide portability for message-passing code on a variety of platforms ranging from multiple processes on a single workstation (or PC), shared memory platform, distributed shared memory platforms, scalable parallel computers to networks of workstations. The standard specifies a language-independent specification (LIS) as well as C and FORTRAN bindings. While initially targeted for numerical computations, the standard has a broader appeal now and its current extensions target client-server distributed paradigms, embedded applications, and process management.

The approved MPI-1 standard provides point-to-point communication, collective operations, process groups and communication domains, process topologies and environment management and inquiry [6, 11]. MPI-2, which will be standardized in the Spring of 1997, provides additional functionality over MPI-1 in the areas of process creation and management, one-sided communication, collective operations, external interfaces and I/O. It also provides a C++ binding for MPI-1 and MPI-2 functionality.

The main goal of MPI/RT is to provide message-passing functionality with quality of service (QoS). The parameters of QoS include a variety of fault-tolerance and real-time application requirements. Since many of the high-performance real-time applications would like to take advantage of the MPI functionality but require timing guarantees from the message-passing layer the MPI/RT working group was created with the objective of providing the necessary application programming interface (API).

The rest of the paper is organized as follows. Section 2 presents the underlying philosophy of MPI/RT. Section 3 presents the common underlying layer for all real-time paradigms, while section 4 presents real-time paradigms. section 5 provides supporting functionality, whereas certain open issues and further work are described in section 6, and finally, section 7 presents the current status of the MPI/RT standard and future plans.

# 2 Background

Currently, application developers have to become experts on a platform before they can take advantage of its message-passing facilities (even though the notation is now for the most part standard) and achieve the desired performance. The challenges are even greater for developers of real-time applications that are required to satisfy timing constraints and proper interaction with the environment outside of the computing platform. The application design is often so dependent on the current computing platform that it requires complete redesign when ported to a different platform.

This approach hinders the portability of an application to a different platform and upgrades to the currently used one. The current philosophy is that the platform provides the user with an API and it is the application developer's responsibility to satisfy timing and other qualities of service that are required. This philosophy is contradictory to the portability view. MPI/RT has taken the opposite approach. The users provide detailed information about timing constraints of different parts of an application and the interactions between these parts including message-passing data and control message exchanges. The platform, including middleware of which MPI/RT is a part, contains the analysis of user requests and either satisfies them with user required QoS or states that it cannot satisfy the user desired QoS. The denial of the request can be caused by a lack of platform resources.

It is our view that middleware and platform designers understand better than the user how to provide QoS on the platform given enough information about the application. In this way, application programmers can concentrate on improving application code and let middleware providers concentrate on providing the best QoS available on the platform. Application programmers are not required to reveal all the information to MPI/RT and can take it upon themselves to provide some or all QoS. It is quite clear that the exact boundary of the responsibility for providing QoS for the user between the platform with all of its system software and middleware and application is still unknown. But the same trends that lead to the development of higher level languages, operating systems, and middleware, are pushing the development of MPI/RT.

The goal of MPI/RT is to provide the middleware for programmers to create real-time applications with performance portability. MPI/RT provides a consistent set of extensions and, in some cases, restrictions to MPI. MPI/RT adds greater predictability and schedulability to message-passing programming, while modifying and extending the useful concepts embodied in the original standard.

In order to provide the quality of service guarantees for communication, an MPI/RT implementation may need to address a difficult scheduling problem. While there is a lot of work going on in CPU and network real-time scheduling, these results in many cases are not sufficient to provide guarantees for communication. The number of resources that are involved in communication is rather large and is different from one platform to another. For example, for one distributed shared memory platform the scheduling of the following resources have to be addressed for point-to-point data transfer:

1. Sender and receiver nodes application processor CPU,
2. Sender and receiver nodes data bus,
3. Sender and receiver nodes memory,
4. Sender and receiver nodes network interface chip,
5. Underlying physical network that can have various topologies.

All these resources can have their own schedulers that may use completely different techniques. It is quite common to schedule a CPU using priorities, network switches using round robin or no scheduling at all, while network interface chips are scheduled using interrupts and signals.

However, it is hard, if not impossible, for the application programmer to coordinate the use of these resources in order to establish user-required quality of service even with the complete knowledge of the application. Furthermore, even if it was done successfully on one platform, it cannot be ported to a different platform because of the differences between platform architectures. MPI/RT implementors have a better chance of meeting user's quality of service requirements because of their knowledge of their platform, since for most cases they are part of the same organization that designed and built the platform. The MPI/RT standard provides the user specification so the platform designers and MPI/RT implementors will clearly see what quality of service guarantees users want.

In order to improve the chance for satisfying user quality of service requests, MPI/RT recommends early binding. Many of the highly demanding real-time parallel applications come from the radar sensor and signal processing domains. They are characterized by the periodic nature of the environment outside the computing platform, and for them establishing communication channels with QoS (see Section 3) promises the greatest benefits. With this information on the patterns of the communication and the QoS for them, MPI/RT implementations can allocate resources using an algorithm and run-time scheduling criteria that are most suitable for the platform. It also provides a feedback to the application that the QoS can or can not be satisfied prior to actual data transfers. This will allow an implementation to minimize the critical execution path for message passing and potential overhead for implementation of control messages. Hence, the overhead of MPI/RT will decrease, message passing performance using MPI/RT will come close to the platform native message passing performance, and "the price of portability" will be minimized.

This approach has several more benefits. While currently most message passing primitives assume "two-sided" operations (send and receive) of various forms, some platforms prove "one-sided" operations (put and get). For some platforms, one-sided communication provides higher throughput and lower latencies. With the preestablished channels it is possible to exploit "no-sided" communication where application does not issue any data transfer commands and middleware (MPI/RT) does the data transfer operation on behalf of the application at the predetermined times [5] that are part of the channel establishment. The next section provides more details about the channels and two-sided, one-sided and no-sided communications.

# 3 Paradigm-Common Functionality

## 3.1 Channels with Quality of Service

In MPI/RT, persistent channels offer the functionality of a virtual channel [2, 7, 10] within the framework of the MPI standard. Motivations for having virtual channels in MPI/RT include: ability to exploit persistent communications that are common for high performance real-time applications, deadlock and livelock avoidance, virtual channels guarantees for properties critical for timing correctness, and more efficient resource usage by the MPI/RT implementations.

As stated before, MPI/RT as a specification and programming notation encourages early binding in order for MPI/RT implementations to establish user required quality of service, while providing both early and late bindings for data transfer operations. The standard defines a collective channels initiation operation that provides MPI/RT with the big picture of application-desired, point-to-point channels with their respective QoSs. The early knowledge of all the point-to-point channels allows MPI/RT implementation to exploit potential flexibility in satisfying individual channels QoS rather than establishing each channel individually and making arbitrary decisions that may be detrimental to MPI/RT's ability to satisfy all channels QoSs. This functionality is not required to be done prior to any data transfer operations, but is strongly encouraged to maximize MPI/RT's potential performance. The channel establishment operations as well as channel modifications and deletions, can be used at any time, but these operations are expensive and it is harder for the implementation to satisfy later requests and to optimize resource usage, if these requests are relatively frequent.

Following the MPI principle that all communications are done over the communicator (clique or bi-partite group formulation), group MPI/RT channel initialization operations are done over a communicator group. The same application process can participate in more than one communicator group and moreover all processes are by default members of one communication group `MPI_COMM_WORLD`. Hence, a process can participate in channel initialization for more than one communicator. The MPI/RT standard is, however, silent on how the above established channels are mapped on the network channels. This is left to the implementation and is highly dependent on platform architecture, network topologies, routing information etc. The solutions that shared memory platforms would like to use may not be applicable to the distributed memory platforms and vice versa. Implementations should be left free to make such choices.

While not presenting the entire syntax of the collective point-to-point channel initialization operations, we would like to stress several parameters that carry semantic information. First, the operations allow specification of information for all point-to-point channels over a single communicator the process would like to use. This includes several point-to-point channels between the same pair of processes. Using this specification, an application can establish any virtual topology between processes. The operation returns a request handle for each channel or an error for each channel that can not be established.

Each channel is specified by Quality of Service parameters, message buffers and an error handling request. QoS is either one of the real-time paradigms (see Section 4) or a "softer" quality of service that does not provide an absolute guarantee for each data transfer. The optional period can also be specified as part of the quality of service. Since no guarantee can be absolute (because of hardware and software faults) the channel initialization operation specifies an error handler that will be invoked by MPI/RT when the data transfer quality of service was not achieved. This is a part of the generic functionality MPI/RT provides for application fault-tolerance (see Section 3.2).

The buffer specification allows an implementation to minimize message copying and is a required part of the no-sided communication paradigm. In the simplest form, an application requests a single buffer of specified datatype, the number of elements of this type, and the buffer starting address. The address is in the application space but is under MPI/RT implementation control during the data transfer times. A more sophisticated version specifies a buffer pool with the above information for each buffer, as well as the requested implementation buffer management strategy. A separate set of functions is also provided to pass buffer control between the application and the implementation that specifies requested user buffer management strategy.

MPI/RT also provides functionality to establish collective channels with the quality of service. These play the same role for collective operations (like scatter, gather, broadcast, all-to-all scatter-gather) as point-to-point channels for individual send/receive operations. Notice that the specification of the quality of service, buffers and others may differ from one collective operation to another.

### 3.2 Fault-Tolerance

Currently, MPI/RT provides limited functionality for fault-tolerance. However there are two areas where functionality is specified. In both areas the functionality is provided for an application to recover from faults rather then preventing faults. But by satisfying QoS for channels an implementation can and should take into account platform provided fault-tolerance information. First, for all data transfer and some other MPI functionality, the timeout parameter is added. This allows the application to associate an error handler with the timeout error return from the operation. Second, there is an error handler associated with each channel that will be invoked by MPI/RT if the application requested QoS for the data transfer operation can not be provided. Recall, that for both time-driven and event-driven paradigms this implies that the time requested for the operation completion or invocation was not satisfied, and hence, timeout occurred.

## 4 Paradigms

The emerging MPI/RT standard currently defines functionality to support three well-known real-time paradigms: time-driven, event-driven, and priority-driven. While each paradigm is well understood in itself, the mixture of paradigms, their interaction on a single platform, and the meaning of the mixed specification is the area of intense study by the MPI/RT working group. Some applications prefer two or more layers of QoS hierarchy, but the order of the hierarchy differs widely from one application domain to another. Below we present the details of the three real-time paradigms and the QoS specification for each of them.

## 4.1 Time-Driven Paradigm

The primary goal of the time-driven approach to MPI/RT is to allow a real-time application sufficient control of the environment in which it is running so that it can explicitly schedule its message-passing activities and resource usage. Since MPI, the underpinning of MPI/RT, is designed as a message-passing library, it cannot schedule by itself, but must depend upon the operating system and communication and network protocols to enforce specified schedules.

An application using time-driven MPI/RT QoS will be able to specify time intervals to bound the resource usage of communication operations using globally synchronized clock values, and the implementation of time-driven MPI/RT will fulfill these requirements with minimal changes to the MPI standard.

The existing MPI message transfer operations lack two parameters that we consider critical for real-time applications, particularly for the time-driven programming paradigm. These are a starting time of the operation and a timeout for completion of the operation. The starting time of an operation and the timeout should be considered as special cases of an event. While certain applications (especially embedded ones) prefer an even finer granularity of control, we sought to strike a balance between the feasibility of an implementation and what time-driven application designers want to use. For example, there is a hard lower bound for the starting time, but no hard upper bound on the starting time, in the current specification.

One distinctive characteristic of the time-driven approach to real-time message-passing is its lack of need for queues and system buffers. Applications use *ready mode* message-passing implicitly. A ready-mode send may be started *only* if the matching receive has already been posted [6, 11]. On many systems, this allows the removal of a hand-shake operation and results in improved performance. Since a parallel time-driven program must globally schedule all message transmissions, the message receiver always knows to expect an incoming message. Thus, for reasons of efficiency and simplicity, a time-driven MPI/RT implementation should not do any handshaking (as many of the existing non-real-time implementations do). It is rather up to the application to specify times (for start and timeout) to ensure that the sender/receiver (local/remote) pairs are working in synchrony.

Another distinctive feature is a potentially more efficient way of using notifications, which can be more minimal (shorter critical instruction path) than with other approaches. A time-driven MPI/RT application does not need to be notified when a message is transmitted successfully and on time; instead it is notified only when an error occurs (e.g., a timeout expires). A matter of significant discussion in the MPI/RT group concerns precisely what should happen to messages left on the network when timeout expires.

An activity interval, specified by a starting time and a timeout, is an input parameter for a scheduled message send. The purpose of this parameter is to ensure that the system resources required to satisfy this operation will not be used outside of a specified interval. These resources can be narrowly interpreted to refer to the interprocess communications network. A broader interpretation would include memory accesses, node busses, network interface cards, and so on. Again, while we prefer a finer granularity of control, we have tried to strike a balance between the feasibility of an implementation and what time-driven schedule designers want to use.

The starting time and timeout are somewhat symmetric. The starting time ensures that the resources needed for a data transfer operation will be available at the specified start time. The timeout parameter, in contrast, would ideally specify the time when all resources required by the message transfer operation are no longer in use. That is, after the time specified in the timeout, irrespective of whether the operation completed successfully or not, all system resources (physical network, network interface cards, node buses, message buffers, etc.) have been released and can be used for subsequent message-passing operations.

Unfortunately, in practice these guarantees often cannot be met. The MPI/RT timeout therefore specifies that the message transfer should be stopped and the calling application should be notified if the operation has not completed by the time specified by the timeout. Since the message may be progressing through a multi-stage network, a time-driven MPI/RT implementation may need to send a message from the receiver node to the sender to indicate that the timeout has occurred. The resulting error messages may not be received by the timeout deadline, and they may use resources after the timeout. Thus the application may need to reserve resources to handle such events. It should not be the responsibility of the MPI/RT implementation to provide this bound, since any guarantees that can be given from the perspective of a user-level message-passing library would be too naive to be useful. The application itself is in a much better position to know timing and performance details relevant to establishing such a bound, including details of the platform and knowledge of the run-time patterns of communication. Even for the application, it may be extremely difficult to establish such bounds, especially if the real-time performance characteristics of the operating system or the underlying runtime system are poorly known or highly variable.

The starting times and timeouts of the activity interval in time-driven MPI/RT data transfer operation calls are specified by a structure called a *MPIRT_TIME_OBJECT* (one instance of the structure is used for each). A *MPIRT_TIME_OBJECT* has two fields, *MPIRT_TIME_OBJECT_TYPE* and *MPIRT_TIME_OBJECT_TIME*. *MPIRT_TIME_OBJECT_TYPE* must have one of two values, *ABSOLUTE*, or *RELATIVE*. When a starting time

is replaced by *MPIRT_TIME_IGNORE*, then there is no hard constraint on when the operation should start. Implicitly, it should start as soon as possible, just as with the current MPI calls. Similarly, when a timeout is given by a *MPIRT_TIME_IGNORE*, there is to be no hard constraint on when the operation should end. Furthermore, the second field of a *MPIRT_TIME_OBJECT* is insignificant if the first field is set to *MPIRT_TIME_IGNORE*.

For a *MPIRT_TIME_OBJECT* whose first field is *ABSOLUTE* or *RELATIVE*, the second field (*MPIRT_TIME_OBJECT_TIME*) should be a field containing a double precision floating point number. In either case (*ABSOLUTE* or *RELATIVE*) the *MPIRT_TIME_OBJECT_TIME* refers to the global synchronized clock, but in the relative case, an actual constraint is to be derived at run-time by adding *MPIRT_TIME_OBJECT_TIME* to the most recent reading of the global synchronized clock. Note that even in the *ABSOLUTE* form, an actual time requirement cannot necessarily be constructed until the value of the time object at the time of the execution of the call is known.

## 4.2 Event-Driven Paradigm

The event-driven paradigm supports the specification of events that either trigger or stop application or MPI operations. MPI/RT provides an API for both levels of the specification. The low-level, event-driven paradigm provides a mechanism for scheduling (with QoS) an application handler upon the completion of a data transfer operation. The high-level event-driven paradigm provides a mechanism for scheduling any application activity with QoS, including MPI data transfer, application functions triggered from a system event, an application event, or MPI event. Both paradigms allow users to manage MPI, system, and user resources using events.

The QoS for the event-driven paradigm is the bound required on the activation time of an event handler for a delivered event. The low-level event-driven paradigm will consider local event bounds, whereas the high-level event-driven paradigm will consider "global" event bounds. Additional specification is under consideration that allows users to provide the bound on the number of events over some time interval. This is similar to most specifications for aperiodic tasks [2, 9].

### 4.2.1 Lower-Level Event-Driven Paradigm

The lower-level event-driven paradigm provides functionality in order to specify a request handler and a local event that will be used by a MPI/RT implementation as a trigger to schedule the request. Request handlers are an ideal mechanism for implementing the event-driven paradigm. Handlers are introduced in MPI-2. The functionality of this paradigm can be used with either MPI or MPI/RT operations' requests. To help users better manage resources, two events for the data transfer completions are introduced. One event specifies the local completion of the data transfer, that is when the message buffer can be reused, an event which is currently available on most platforms. The other specifies the global completion of the data transfer, meaning the channel resources can be reused.

Once the event handler has been posted, the handler function is to be called within the event-driven QoS after the given request reaches the event condition. When the handler is called, it is passed the *request*, the status of the request, and the input parameters for the event handler. If the condition handler cannot be called within the specified QoS then the failure handler is called. Like the request handler, the failure routine is passed the *request* argument, that request's status, and the input arguments for the error handler.

The request handler is assumed to be "full-weight." That is, it can execute any MPI call or system-specific synchronization call and may run for an indeterminate amount of time (i.e., it is not restricted like a signal handler.) Also, handlers do not implicitly "consume" their request(s). The request passed to a handler can still be waited on or freed by the process before or after the handler is called, unless the handler itself explicitly frees the request.

### 4.2.2 High-Level Event-Driven Paradigm

The main goal of the high-level event-driven approach to MPI/RT is to help the application to control the run-time environment in which it is running with explicit scheduling of MPI and computation activities and resource usage. Coordination is required between MPI, the operating system, as well as communication and network protocols to enforce the schedules. In a nutshell, an application using event-driven MPI will be able to specify intervals *guarded* by the specified events in order to bound the resource usage of communication and computation activities.

Currently many applications "wait" on system events or user control messages to schedule a handler, that in turn schedules several application activities: functions, processes, threads, and data transfers. The model for high-level event-driven paradigm presented in this section establishes the direct coupling between events and application activities without user handlers. Just as MPI provides the interface for data flow, the high level event-driven section provides the interface for control flow.

An event is a discrete, atomic, instantaneous state transition without any liveliness [1]. The events can be both persistent and one-time only. Three types of events are specifiable: system events, communication events, and user events. Each event is identified by name. The name is associated with a persistent event. The type of event indicates the type of the resource that generated the event. System events are generated by the platform environment, for example operating system. Communication events are coupled with persistent channels that are introduced in earlier sections, and are generated or captured by MPI. User events are dedicated to the synchronization of the resource usage among different processes (nodes) on the platform, and are generated by the application.

For the high-level event-driven paradigm, events are not necessarily local to the process or even a node. Each process registers the persistent event names with MPI that it wants MPI to "monitor" and the persistent event names that the process will generate.

In the current MPI/RT draft, all the communication events are associated with the MPI/RT channel use. This proposal contains only two events associated with the channel, which are introduced in the low level event-driven section: local and global communication completion. In order to match these events with the guarded activities properly, MPI associates a persistent global name with a channel. The channel name can be either provided to an implementation by the application or the implementation will assign a name to a channel. Hence, there are two persistent event names associated with the channel. For a channel named $\alpha$ they are: $\alpha$*_local_complete* and $\alpha$*_global_complete*. The user can provide the channel names and MPI will assign them to the channels, or the user can specify *MPIRT_IGNORABLE* and MPI will provide the channel names and return them as an out parameter. The names on both endpoints of the channel must match.

User events have meaning only to the application. MPI is just a mechanism to match user events and responses as well as the mechanism for event delivery and response triggers. An application assigns a persistent name to a user event and notifies MPI about which process generates this event. This is the only event type that is generated by the user. The events of two other event types are generated by MPI and the system. MPI delivers all the events to the processes that are registered for them and then triggers application functions or data transfers according to the events that guard the activity.

For any function or communication operation, an application can specify events that trigger its start and its termination if it is not finished. The main purpose of the events is to *guard* the interval when the activity may use resources. This is analogous to the time-driven paradigm where no resources will be used by an MPI/RT data transfer operation prior to its starting time of the operation time interval and, to the best of the MPI implementation effort, no resources will be used after timeout of the operation time interval.

The time interval of the time-driven real-time MPI contains two events that are specified by time stamps. From this perspective, the time-driven paradigm is just a subset of the event-driven one. There is, however, one critical difference that lies in the ability of the application to schedule its non-MPI activities. For the time-driven paradigm, there are existing facilities to start non-MPI activities using OS timers, spin-locks and others. These facilities and the synchronized clocks allow the application to coordinate all of its activities, MPI and non-MPI, both local and global. There are no analogous mechanisms for event-driven paradigm, and event delivery/monitoring across the entire platform requires application action and sufficient communication support. This is the place where MPI can really help.

Events "guard" a liveliness interval within which the activity can use resources. While many different activities are of interest for real-time and embedded system for this proposal, we concentrated on two activities: application functions and MPI/RT data transfer operations over a channel. The guards use two lists. The first one is the list of events whose conjuncture trigger the activity. The second one is the list of events, such that any event on the list stops the activity if it is not yet finished by itself. If we need more comprehensive arithmetic of actions we can add such functionality later. For completeness we may, for example, add action *IGNORABLE* for the empty action list. MPI is responsible for delivering events and for triggering (start or stop) an activity if it is eligible. As stated before, for now only two activities are considered: application functions and MPI/RT data transfer operations. Each application process registers event names it wants MPI to monitor and event names it will generate. Since an application can only generate user events, only user event names that application will generate need to be registered with MPI. MPI is already aware of where and how system and communication events are generated. The issue of how the events are delivered to the guarded activity is left to the implementation. The MPI/RT standard provides the functionality for an application to notify an MPI/RT implementation about application generated events.

### 4.3 Priority-Driven Paradigm

In MPI/RT, priorities are specified and fixed *per channel* by a field in the QoS argument to the channel creation calls. As with other QoS parameters, the processes at both ends of the channel must provide the same priority or an error occurs.

Because varying platforms may provide different levels of support for message priority at the OS level and below, MPI/RT specifies little about how message priorities are implemented. In addition to passing message priority information to the appropriate OS and hardware layers, a high-quality MPI/RT implementation will order operations internally according to priority information. For example, given the choice between performing two different communication operations (such as receiving one message or another), the higher priority communication should be performed first. If the high priority communication blocks or stalls, lower priority communication may be initiated. Notice that in the general case, this implies that communication may need to be preempted. For example, if the user initiates a low-priority nonblocking send, and then begins a high-priority send, the low-priority send would be stalled in favor of the high-priority send.

MPI/RT makes no attempt to correlate process and message priorities, and indeed, has nothing to do with process priorities whatsoever with exception of request and error handlers. Such functionality should instead be provided by domain-specific middleware. Thus, while an MPI/RT *implementation* may need to concern itself with process priorities, the *interface* itself does not.

# 5 Support

## 5.1 Synchronized Clocks

Most platforms that support real-time applications provide tightly synchronized system clocks that are dependent upon special hardware support. There are several compelling reasons for having highly synchronized clocks [9]:

1. Fine-grained, accurate instrumentation is needed for all approaches to real-time message-passing systems, and even for performance measurements in non real-time systems.

2. Real-time applications require precise timing correctness. These systems also require demonstrations of this correctness, and often require delicate tuning for optimal performance. Well-synchronized clocks are necessary to support these requirements in a parallel environment.

3. Applications should be able to adjust scheduled times for portability, based upon the quality of the synchronized clocks. For example, time padding will be needed to adapt scheduled message-passing operations on heterogeneous processing nodes.

4. The primary goal of time-driven real-time MPI is to support application specification of resource usage. For time-driven MPI/RT, all resources that are used for communication need to be scheduled in order to achieve predictable behavior. These scheduled resources can include:

   (a) Distributed Memory,

   (b) Shared Memory,

   (c) Communication Bandwidth,

   (d) Communication Fabric, and

   (e) Computation,

   as well as others, such as platform specific resources related to inter-process communication.

The processing nodes (which may contain one or more CPUs) that comprise a real-time parallel processing system may have access to several clocks. We designate one of the clocks as a globally synchronized clock. For each process of a program that uses MPI/RT, this clock will be the one accessed by the MPI/RT implementation. There is an underlying assumption that each process is associated with a fixed processing node. The process accesses its globally synchronized clock through its associated node.

For most platforms, each node has a local clock, and this is periodically corrected in order for it to serve as a synchronized clock for all MPI/RT processes active at that node. However, other alternatives are not precluded. For example, there may even be a single clock at one of the nodes or even completely outside of the participating nodes, which all the processing nodes access over a network (which may also be used for regular data transfer operations) to get a synchronized clock value. A fundamental assumption is made that the system clocks will be *monotonically non-decreasing.* We also assume that the underlying operating system will hide any artifacts resulting from the overflow of system clock counters.

We present several parameters that describe the synchronized clocks that can be accessed at run-time, compile-time or both.

**Resolution (Tick)** Resolution represents the time between two successive clock ticks. The resolution of the various synchronized clocks in a heterogeneous system may differ.

**Drift** Drift indicates, for each synchronized clock, a guaranteed upper bound on the error in the rate of the clock. That is, if the drift is $\delta$, then the clock rate (measured in seconds per actually elapsed second) is guaranteed to be between $1 - \delta$ and $1 + \delta$. Drift is, as stated above, dimensionless.

Drift can be effectively used to bound the accuracy of measurement of small time intervals, when they are measured by the difference between two readings of the same synchronized clock.

An implementation of MPI/RT should reset the drift parameter when global system clocks are synchronized.

**Skew** Skew is a maximum bound on the absolute value of the difference between simultaneous values of the synchronized clocks in distinct nodes. Note that this refers to ideal values, not the result of any real reading operations.

**Accuracy** Accuracy is a maximum bound on the absolute value of the difference between simultaneous values of the synchronized clock and an ideal clock started at the synchronized clock hypothetical starting time (some critical instant in time).

If synchronized clocks are periodically corrected in order to deal with drifts and other inaccuracies, the calculation of the interval accuracy based upon drift will be too pessimistic for large intervals.

This value can also be used for cross-platform synchronization.

**Access Time** Access Time is a maximum bound on the time to execute a call of `MPI_WTIME`, This, of course, assumes that the execution of the call is not interrupted by the operating system. This undesirable and ambiguous caveat is necessary with current operating systems (especially those with virtual memory) because an absolute guarantee would be so long as to be practically unusable.

The values of the listed parameters do not depend on what applications are running but rather on the parallel environment. These parameters represent constraints on the changes to the environment. For example, the skew should not be increased when new processes are added while an application is running. Various caveats for the above parameters can be added to allow implementors to provide finite bounds, of some residual value to users.

## 5.2 Instrumentation

Instrumentation is an essential aspect in providing application developers with the metrics needed to monitor quality of service assurances and fine tune specific hardware configurations [8]. These metrics support performance portability, maintainability and fault tolerance. Real-time instrumentation includes, but is not restricted to, monitoring application performance and monitoring MPI/RT performance. Other performance monitoring directly related to the overhead of MPI/RT operations will be implementation dependent. An important benefit from performance monitoring is the ability to capture global and local resource utilization information.

Currently, there is little information available to the user concerning internal MPI events. In some circumstances where timing is critical, an application could benefit from information about times of resources used by internal MPI (and native) communication events (and states). Real-time instrumentation must be sensitive to the impact of monitoring on the timing behavior of an application. To minimize this impact, MPI/RT instrumentation will include interfaces for existing monitoring instruments and/or event loggers. This design provides implementation independence.

Real-time instruments will not duplicate efforts provided in profiling tools, although some of the information collected may be duplicated. The distinction between profiling and instrumentation will be defined by global and local resource requirements and impact on timing requirements. Implementors may choose to use "profiling" hooks when applicable if performance can be achieved.

Run time instruments support performance monitoring, decision analysis and fault tolerance. The MPI instrument monitoring API is designed to monitor, collect and output metrics. For systems that have a generic performance monitoring capability, MPI/RT monitoring may be integrated into the existing capability. Metrics that are obtained from performance monitoring may provide decision analysis criteria for conditional heuristics. These heuristics guide fault tolerance policies and support load balancing schemes.

Performance monitoring for both application specific information, and MPI/RT specific information are accommodated. The standard will not dictate this level of detail, but instead provide recommendations to implementors and users.

# 6 Other Areas for Future Work

The work on MPI/RT thus far has been a breeding-ground for a lot of new discussion about generalizing and improving the MPI specification as well as adding real-time features. Quality of service, noticably absent in the main MPI standard, but present in the real-time standard, offers new opportunities. For instance, the issue of progress (asynchronous completion of pending non-blocking operations independent of user program activity) is a requirement of MPI for point-to-point operations. In MPI-2, debate is on-going about whether to have such a progress promise for collective non-blocking operations, because of perceived difficulty of efficient implementation on certain platforms if progress is mandated. From the MPI/RT perspective, the promise of asynchronous completion of non-blocking operations can be viewed as a type of quality of service, and one can envision a generalization of MPI to support non-blocking point-to-point and collective operations with or without progress guarantees. Coupled with early binding, this could offer applications greater control, and implementations greater optimization. While this is not possible under the base MPI specification at present, it is a potential "orthogonalization" that could be introduced into the real-time standard first, and later be considered for the non-real-time MPI.

Additionally, MPI is successful, in part, because of the useful abstractions that it contains. One such abstraction is the process group, and the more useful communicator that supports data transmission among processes in that group (there is also a two-group variant). In the MPI/RT setting, where resources are more explicitly at issue, though accessed indirectly through quality-of-service requests, further abstractions seem called for, building on those already encompassed by MPI. For instance, channels are built based on a communicator, which provides the processes from which pairs are selected (point-to-point channels), or provides the entire set for collective channels. Sets of channels are themselves a legitimate data structure, and they collectively embody a set of resources allocated to them. Providing ways to transition these resources mutually exclusively between sets of channels over the same communicator is a key abstraction we have yet to add. Furthermore, providing an abstraction that is a set of communicators would likewise add some flexibility in certain circumstances. Both of these ideas would help support the continued use of the well-formed parallel calls contemplated by MPI, for which the communicator provides communication safety. Considering algorithms for dynamic transition of resources between the channel sets constitutes an important area of inquiry, not necessarily ready for standardization.

Sets of channels sharing a common underlying resource, and asserting different priorities is yet another example of the need for sets. It has been noted that priorities associated with channels are too early bound; we may have to solve this problem by reintroducing priority-oriented communication calls that provide late binding, or by providing sets of channels that share a common underlying admission test and concomitant resources. Such choices remain to be made as we finalize the MPI/RT's first standard release.

In general, the collective admission tests that we suppose in MPI/RT, require the solution of a mixed integer non-linear program, for which the mathematical structure must be studied, and appropriate approximate solution techniques realized within implementations. It is clear that for certain, relatively static systems, such optimizations should be done offline, and there is no a priori reason in general to assume that the online approximations need be as good as those computed at great effort offline. In fact, it may be totally inappropriate to compute the solutions online in highly constrained systems, motivating the need for MPI/RT to provide an external interface to such allocation computation results. Making these computations open to the user as well as offline is a separate issue.

Notwithstanding the above, we have developed a number of useful abstractions that will make MPI/RT non-trivially useful in its first release, even if we defer certain of the issues described in this section to the second standard release, in order to gain more practical experience in implementations and application uses of the standard.

# 7 Conclusions

The MPI/RT standard constitutes the first effort to provide portable specification for real-time passing user requirements of which we are aware. It allows the domain of portable message passing high performance parallel computation (MPI domain) to be enlarged to include to embedded and time-critical applications. While still not in its final stage, MPI/RT clearly reveals the functionality missing from MPI and different application design approaches that real-time applications are using, and addresses these omissions.

Currently, there are two groups of which we are aware, one at MSU and one at SKY Computers, Inc., that are working on the first prototype implementation of the standard. The latest draft of the standard can be found in *ftp://aurora.cs.msstate.edu/pub/mpi* [4]. One can join the MPI/RT standard working group by sending a message *subscribe mpi-realtime* to *majordomo@mcs.anl.gov*.

# 8 Acknowledgement

The authors would like to thank all the members of the MPI/RT working group without whom this work would not been achieved.

# References

[1] D. M. Auslander and C. H. Tham. *Real-Time Software for Control: Programming Examples in C.* Prentice Hall, 1990.

[2] D. Ferrari. A new admission control method for real-time communication in an Internetwork. In D. Son, editor, *Advances in Real-Time Systems.* Prentice Hall, 1995. Chapter 5.

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

[4] Message Passing Interface Forum. Real-time message passing interface standard draft. ftp://aurora.cs.msstate.edu/pub/mpi/rt-2-02dec96.ps, 1996. latest draft.

[5] Richard A. Games, Arkady Kanevsky, Peter C. Krupp, and Leonard G. Monk. Real-time communication scheduling for massively parallel processors (position paper). In *Real-Time Technology and Applications Symposium*, pages 76–85. IEEE, 1995.

[6] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

[7] Rainer Händel and Manfred N. Huber. *Integrated Broadband Networks: An Introduction to ATM-Based Networks.* Addison-Wesley, 1991.

[8] F. Jananian. Run-time monitoring of real-time systems. In D. Son, editor, *Advances in Real-Time Systems.* Prentice Hall, 1995. Chapter 18.

[9] Hermann Kopetz and Paulo Verissimo. Realtime and dependability concepts. In Sape Mullender, editor, *Distributed Systems, 2nd Edition*, chapter 16, pages 411–446. Addisson-Wesley, 1993.

[10] A. Mehra, A. Indiresan, and K. G. Shin. Resource management for real-time communication: Making theory meet practice. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications*, pages 130–138, 1996.

[11] Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* MPI, 1996.