The Communication Semantics of the Message Passing Interface

Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby

UUCS-06-012

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

31 October 2006

Abstract

The Message Passing Interface (MPI) standard is a natural language document that describes a software library for interprocess communication. Automatic reasoning about the reactive nature of programs communicating via MPI libraries is not possible without also analizing the library being used. Many distributed programs that use MPI are relatively brief compared to the libraries that implement MPI. A formal specification of the communication semantics of the MPI standard (i) enables modular automatic reasoning of MPI based parallel programs independent of the library implementation, (ii) provides a mathematically precise declaration of the natural language intent of the MPI specification, (iii) enables mathematical reasoning about libraries that implement the standard, and (iv) allows for reasoning about the standard itself. We have created such a specification of the point to point operations and present it in this report. We also discuss some preliminary efforts to accomplish (i) above.

Disclaimer: While the semantics have been proof-read once, the actual semantics document is continually evolving. We are developing a tool–MPIC–that can be used to verify programs against this semantic specification. When the MPIC tool is released there will be a new version. The MPIC tool will also have an accompanying technical report.

Although every effort has been made to correctly model the intent of the MPI 1.1 specification, we make no claim regarding the correctness of the model contained herein. Please notify the authors if a discrepancy is found.

The Communication Semantics of the Message Passing Interface *

Robert Palmer Ganesh Gopalakrishnan Robert M. Kirby

31 October 2006

Abstract

The Message Passing Interface (MPI) standard is a natural language document that describes a software library for interprocess communication. Automatic reasoning about the reactive nature of programs communicating via MPI libraries is not possible without also analyzing the library being used. Many distributed programs that use MPI are relatively brief compared to the libraries that implement MPI. A formal specification of the communication semantics of the MPI standard (i) enables modular automatic reasoning of MPI based parallel programs independent of the library implementation, (ii) provides a mathematically precise declaration of the natural language intent of the MPI specification, (iii) enables mathematical reasoning about libraries that implement the standard, and (iv) allows for reasoning about the standard itself. We have created such a specification of the point to point operations and present it in this report. We also discuss some preliminary efforts to accomplish (i) above.

1 Introduction

Standards documents are one of the powerful tools for developing portable, reusable, and correct implementations of complex systems. In almost all cases, they are initially created as semi-formal documents, often containing gaping holes and potentially ambiguous statements. Over time, thanks to the experience gained from

*Supported in part by NSF award CNS-0509379 and a grant from Microsoft Corporation.

the widespread use of systems built according to the standard, they evolve to become much more rigorous and coherent. Yet, without a *mathematical* (formal) description, they still leave much room for misinterpretation – often unfortunately coinciding with the increased scale of design and deployment of systems built according to the standard.

The IEEE Floating Point standard [6] is a resounding success story in this area. It was initially conceived as a standard that helped minimize the danger of non-portable floating point implementations. As a fortunate side effect of the infamous Intel Pentium division bug, it now has incarnation in various higher order logic specifications (e.g., [5]), and routinely finds applications in *formal proofs* of modern microprocessor floating point hardware circuits. We strongly believe that the MPI communication standard – one of the most widely used in high performance computing – has the vast potential of being solidified in a similar fashion.

MPI is already a success story in the area of software library standardization, in that a collection of primitives that support message passing based communication for high performance computing has been widely adopted. Unfortunately, the MPI standard [10] uses natural language descriptions, as well as examples to communicate definitions, semantics, and other important details. Experience shows that this can lead to errors that result from unstated assumptions, ambiguities, and unclear causal dependencies. Some of the recent additions to MPI, such as *one-sided communication constructs*, are so tricky to understand that even simple algorithms using them have been shown to be incorrect (e.g., [9, 12]). These errors can be progressively eliminated by relying on *formal* (mathematical) descriptions of MPI, and employing modern formal verification techniques such as model checking [3].

1.1 Related Work

Significant inroads have been made in formalizing the MPI standard. The earliest work we are aware of is that of Georgelin et. al. [4] where the authors create a LOTOS description of MPI_BSEND, MPI_SSEND, MPI_RSEND, and MPI_RECV along with the collective operations MPI_BROADCAST, MPI_GATHER, and MPI_SCATTER. The MPI_ANY_SOURCE and MPI_ANY_TAG wild-cards are modeled. The above operations are modeled using the channel primitives of the LOTOS description language. They then apply the model to verification of some MPI programs.

More recent work by Siegel and Avrunin includes:

- In [13, 15] the authors create a mathematically precise model of MPI_SEND, MPI_RECV, MPI_SENDRECV, and MPI_BARRIER. The MPI_ANY_SOURCE and MPI_ANY_TAG wild-cards are expressly disallowed due to the additional non-determinism that are introduced with their use. The process universe is a fully connected graph where edges in the graph are a pair of FIFO channels. Synchronous communications are modeled as an interleaved execution of two processes where a send is followed immediately by the corresponding receive in an execution trace. A number of theorems and their proofs are presented with regards to synchronous communication in the proposed model.

- In [14] the authors model a simple MPI based 2D diffusion simulation and verify the model using both SPIN and INCA. Models of MPI_SEND, MPI_RECV, MPI_BARRIER, and MPI_SENDRECV are used in connection with the diffusion simulation. Some of the results from [13] are also presented.

- In [16] the authors verify the output of a distributed numerical program using a model checker and a sequential version of the same program.

Other work in modeling MPI related programs includes [12] where model checking is applied to a program using the one-sided locking routines of MPI 2.

We have also modeled MPI_SEND, MPI_RECV and MPI_BARRIER for use with both SPIN [2] and Zing [11].

1.2 Motivations

With several existing models of MPI one may naturally ask, why have another? To answer this question, consider the following points.

1. There are 35 operations related to point to point communication described in chapter 3 of the MPI 1.1 standard [10]. The most aggressive modeling effort we are aware of contains four (4) of these operations.

2. There is no tool based reasoning support for any mathematically rigorous model of MPI. The only rigorous model we are aware of is the one described in [13]. Models created in languages such as SPIN and Zing are

dependent upon the model checker implementation for a formal description of the language semantics.[1]

3. Many of the errors that are seen in MPI programs are derived not in the use of the blocking sends, rather in the translation from the use of these simple send primitives to the more aggressive counterparts in the ongoing effort to optimize. None of the existing models have representations of these more aggressive operations.

4. A mathematically precise representation of a larger subset of the MPI operations is necessary to create an industrially useful tool for reasoning about programs that communicate using MPI libraries.

5. To serve as a specification, the MPI standard does not mandate implementation details beyond the function signatures and the existence of some symbols. No mention is made of how messages are to be transmitted from one process to another. To make a sufficiently complete model of any MPI program, these details must be filled in. Existing models make no distinction between what is specified in the standard and what is added to support tool based reasoning.

A mathematically precise specification the MPI standard can serve, not only to reason about programs that employ the MPI libraries, it can be used to reason about the various MPI library implementations and the standard itself.

1.3 A Formal Model

Our formal model of the MPI specification is expressed using the Temporal Logic of Actions [7, 1]. TLA is a formal logic containing standard ZF set theory, an action operator that induces a transition relation, and some limited temporal logic. It's semantics are well understood by mathematicians and computer scientists, independent of any verification tool. Implementation details can be abstracted using set theoretic operations in combination with the action operator.

A program that uses MPI can be specified as a formula in TLA representing a distributed computation. The MPI operations are modeled as operators on variables in the formula. Comments accompany individual logical clauses referencing

[1]While this may not necessarily be the case for LOTOS or INCA, any language developed as input for a verification tool that is undergoing active development could deviate from the previously published semantics.

the page and line number of the MPI standard that requires the given clause where possible.

The action operator of TLA induces a transition relation on logical formulas. Variables that are *primed* (i.e., foo') indicate the value of that variable in the next state of the system. Every transition specifies the values of all variables in the next state. Transitions are total functions from valuations of variables to valuations of variables. New operators can be defined for any finite arity as a combination of the existing operators, user defined variables and constants, and parameters to the operator.

1.3.1 Transition Granularity

Using a TLA operator to represent an MPI operation implies that such operations will require exactly one transition to complete. Our model assumes that only one MPI operation will be applied in a given transition. Although 28 of the operations related to point to point communication are modeled in this way, we could see no way to model some of the point to point operations using only one transition. In particular, MPI_SEND, MPI_SSEND, MPI_BSEND, MPI_RSEND, MPI_RECV, MPI_SENDRECV, and MPI_SENDRECV_REPLACE could not be modeled in a single transition. The reason for this is quite simple: Each of these operations writes some variable *and then* waits for some other variable to be written by another process. The specification of which explicitly requires at least two (2) transitions. We will demonstrate in this paper how to model the remaining operations as a sequential composition of the provided operators.

To model operations requiring more than one transition we adopt the same convention described in [13] noting that the sequential composition of MPI_ISEND and MPI_WAIT on a single process is semantically equivalent to MPI_SEND and that a transition sequence with a MPI_SEND followed immediately by the corresponding MPI_RECV can be considered synchronous. Although it is possible to apply sequential and operator composition in such a way that all 35 operations can be derived using a minimal subset of the MPI operations, to facilitate cross referencing the MPI standard, all 28 single transition operations in our model are modeled separately.

1.4 What is and is not modeled

Before diving into all the details, it is important to note that not everything in MPI 1.1 is present in the model. In particular, the following are either not present, are limited in their current modeling, or are currently only placeholders:

- Data. Data, such as arrays of floating point values, objects, etc., could be modeled using TLA. It is, however, not necessary in most cases to retain the actual data of the distributed simulation to verify reactive properties of nodes participating in the distributed simulation. Therefore we allow a placeholder for data such that it can be included when necessary.

- Data manipulation operations. There are many operations specified by MPI to pack and manipulate data. These are not currently modeled, but could be if there were sufficient interest.

- Operations on communicators and topologies. These are modeled to a limited extent to enable point to point communications on intra-communicators. We currently model the operations shown in Figure 2 in addition to the point to point operations of chapter 3 of MPI 1.1 shown in Figure 1. Such operations on communicators and topologies should be a strait forward extension of this work.

- Implementation details. To the greatest extent possible we have avoided asserting implementation details that might constrain an implementation. One obvious ramification of this omission is that modeling return codes of MPI operations is completely eliminated (see Pg 11 of [10]).

- Transient buffering of messages created by the standard mode send (MPI_SEND, MPI_ISEND, MPI_SEND_INIT). We require the system to either eventually buffer these send requests or to never buffer them. It is not clear at the time of this writing how to model the situation where a buffer may be available for some but not all of the program execution.

Our model includes the point to point operations shown in figure 1 as single transition TLA operators. The argument order and meaning is as specified in the MPI standard for each operator, adding the pid of the process that is applying the operator as the last argument.

2 Conventions

In TLA, the whitespace in the document is significant. Sequences of logical conjuncts can become quite large and are therefore formatted as bulleted lists, the bullet being the logical and ($\wedge$) or the logical or ($\vee$) operator.

MPI_GET_COUNT	MPI_REQUEST_FREE	MPI_TEST_CANCELED
MPI_BUFFER_ATTACH	MPI_WAITANY	MPI_SEND_INIT
MPI_BUFFER_DETACH	MPI_TESTANY	MPI_BSEND_INIT
MPI_ISEND	MPI_WAITALL	MPI_SSEND_INIT
MPI_IBSEND	MPI_TESTALL	MPI_RSEND_INIT
MPI_ISSEND	MPI_WAITSOME	MPI_RECV_INIT
MPI_IRSEND	MPI_TESTSOME	MPI_START
MPI_IRECV	MPI_IPROBE	MPI_STARTALL
MPI_WAIT	MPI_PROBE	
MPI_TEST	MPI_CANCEL	

Figure 1: Point to point operations included in the TLA specification.

MPI_BARRIER	MPI_GROUP_SIZE	MPI_GROUP_RANK
MPI_COMM_SIZE	MPI_COMM_RANK	MPI_COMM_COMPARE
MPI_INIT	MPI_FINALIZE	MPI_INITIALIZED
MPI_ABORT		

Figure 2: Additional MPI operations modeled to enable tool based reasoning on MPI based parallel programs.

Our modeling is influenced by the desire to model SPMD style programs in connection with the TLA MPI specification. As such, all program variables are assumed to be arrays of variables (one for each process in the computation).

When specifying the next state of a variable, it is necessary to completely specify that next state. As an example suppose variable $rank \in [0..(\mathtt{N}-1) \rightarrow 0..(\mathtt{N}-1)]$ is the rank variable declared by a process. When a process calls `MPI_COMM_RANK` the rank variable would be passed into the function. Our model of `MPI_COMM_RANK` requires that the $rank$ be passed to the operator, not $rank[pid]$. As such, we assume that any parameter that might be written by an operator is an array $a : 0..(\mathtt{N}-1) \rightarrow \alpha$ where only the i^th element (i.e., $a[i]$) is ever accessed by the applying process.

When using the action operator, the value of all variables in the next state must be specified. The specification of the MPI operations includes sometimes many UNCHANGED commands which are short hand for $f' = f$. The MPI operators completely specify all of the MPI variables. In addition, those user variables that may be changed by application of the operator are also either updated or marked as UNCHANGED.

Comments of the form $n.m$ indicate the corresponding page (n) and line (m) numbers that require the particular feature. All comments are enclosed in shaded regions.

3 Data Structures

This section presents the elements of the model that are introduced to mathematically specify the constructs of MPI. Appendix A contains the entire model. We will refer to it throughout the remainder of the presentation.

3.1 Constants

Symbols that are defined in the MPI standard are modeled as constant values. We have included the subset of symbols that are necessary for the point to point communications on intra-communicators.

In addition to these symbols, we introduce four (4) additional constants. These constant values are useful to (i) make an instantiated program model finite, and (ii) to provide some information that is implicitly available to the MPI system. The additional constants are:

- N. The number of processes in the distributed computation.

- MAX_COMM. The maximum number of communicators.
- TYPES. The set of strings representing user specified types.
- TAGS. The set of integers representing user specified tags.
- SEND_IS_BUFFERED. A flag to indicate whether a send can be buffered by the MPI system.

3.2 Variables

Variables are functions. Functions need not have homogeneous domains or ranges. The elements of the domains or ranges need not be numbers (they could be other functions, or strings, or values). The variables in the model are group, communicator, requests, initialized, bufsize, message_buffer, and collective.

Functions therefore model data structures such as records, arrays, and sequences. Functions can represent a sequence in that elements can be modified, added, or deleted in the range or domain using the action operator. For example, a sequence $\langle 3, 2, 1 \rangle$ can be modeled by the function

$$s(x) = \begin{cases} 3 & \text{if } x = 1, \\ 2 & \text{if } x = 2, \\ 1 & \text{if } x = 3, \text{ and} \\ \text{undefined} & \text{otherwise} \end{cases}$$

If we wish to append $\langle 4, 5 \rangle$ to this sequence we would let $\langle 3, 2, 1 \rangle \circ \langle 4, 5 \rangle = \langle 3, 2, 1, 4, 5 \rangle$ as

$$s'(x) = \begin{cases} 4 & \text{if } x = 4 \\ 5 & \text{if } x = 5 \\ s(x) & \text{otherwise} \end{cases}$$

As a shorthand we write $x = a..b$ for $x = \{y \in \mathbb{N} : a \leq y \leq b\}$. If x is a set, TLA denotes $SUBSETx$ to be the power-set or the set of all possible subsets of x.

3.2.1 Groups and Communicators:

A group is a set of integers representing process IDs $members \in SUBSET(0..(\mathtt{N}-1))$ and the $size$ of $members$, $size = |members|$. If foo is a Group then

$foo.members$ is the set of pids in the group and $foo.size$ is the number of elements in $foo.members$ (i.e., $foo[members] \in SUBSET(0..\mathtt{N}-1)$ and $foo[size] = |foo(members)|$).

A ranking function and inverse ranking function are maps $ranking : 0..(\mathtt{N} - 1) \rightarrow 0..(\mathtt{N}-1), invranking : 0..(\mathtt{N}-1) \rightarrow 0..(\mathtt{N}-1)$ such that $\forall k \in Dom(ranking) : \exists n \in 0..(\mathtt{N}-1) : ranking[k] = n \wedge invranking[n] = k \wedge \forall m \in Dom(ranking) : ranking[k] = ranking[m] \Rightarrow k = m$. A ranking and inverse ranking function are associated with each group.

A communication universe is record containing a group handle $group$ and a collective context handle $collective$. Groups and Communicators are referenced by handles on processes. Thus the mapping from handles to group or communicator records may be different on any process.

3.2.2 The collective context

Each communicator has a collective context associated with it. The collective context is not directly accessible to the user program, only through the handle in the associated communicator.

Our model currently includes MPI_BARRIER as two transitions: MPI_BARRIER_INIT and MPI_BARRIER_WAIT. The collective context is a record having

$$\begin{array}{ll} participants & \rightarrow x \in SUBSET(0..(N-1)) \\ root & \rightarrow 0..(\mathtt{N}-1) \\ type & \rightarrow \{"barrier"\} \\ state & \rightarrow \{"in", "out", "vacant"\} \end{array}$$

All processes in the communicator's group must participate in the collective communication. Collective operations operate under a simple state machine. When no process is in the communication the state is “vacant” and the participants set is empty. As processes enter the operation their pid is added to the participants set and the first process changes the state from “vacant” to “in” and sets the type of the communication to “barrier”. Processes are only allowed to enter the communication when the state is “in”. MPI_BARRIER_INIT performs the addition of a process to the participant set when the state is “vacant” or “in” and the process is not represented in the set of participants; blocking the process applying this operator otherwise.

When all processes in the group are in the participant set then the state of the operation changes from “in” to “out” and processes are allowed to exit. MPI_BARRIER_WAIT blocks the calling process until the state is "out", removes the process applying the operator from the participant set, and sets the state to "vacant" if the process is the last to leave the communication.

Additional collective operations can be implemented by adding additional collective message types to the range of $collective.type$ and appropriate checks on the parameters that are passed to the operators.

3.2.3 Requests

The set of requests represent the point to point contexts of all communicators. Messages are paired only if they have the same communicator handle (which in our model are unique across space and time).

A message is represented by the envelope that includes all information needed to pair and transmit point to point communication operations. We model messages as a record (i.e., a function having character strings as elements of the domain) as follows:

$$
\begin{array}{ll}
data & \rightarrow \mathtt{Buffers} \\
src & \rightarrow 0..(\mathtt{N}-1) \cup \{\mathtt{MPI\_ANY\_SOURCE}\} \\
dest & \rightarrow 0..(\mathtt{N}-1) \\
msgtag & \rightarrow \mathtt{TAGS} \cup \{\mathtt{MPI\_ANY\_TAG}\} \\
dtype & \rightarrow \mathtt{TYPES} \cup \{\mathtt{MPI\_TYPES}\} \\
num & \rightarrow \mathbb{N} \\
universe & \rightarrow 0..(\mathtt{MAX\_COMM}-1) \\
state & \rightarrow \{\text{“}send\text{''}, \text{“}recv\text{''}\}
\end{array}
$$

Where `Buffers` is a placeholder for future inclusion of data in a model.

A request is the bookkeeping information needed to manage messages within a process. Request objects are required to be opaque to the user process and are therefore represented by a function `requests` : $\mathbb{N} \rightarrow Request$ where the set $Request$ is the set of all possible request objects. The request handle is the element of the domain of the `requests` function which returns the associated request object.

With $Seq(\mathbb{N})$ as the set of all sequences of natural numbers, we model request objects as records as follows:

$$\begin{array}{ll}
error & \rightarrow \mathbb{N} \\
active & \rightarrow \{TRUE, FALSE\} \\
transmitted & \rightarrow \{TRUE, FALSE\} \\
buffered & \rightarrow \{TRUE, FALSE\} \\
started & \rightarrow \{TRUE, FALSE\} \\
canceled & \rightarrow \{TRUE, FALSE\} \\
deallocated & \rightarrow \{TRUE, FALSE\} \\
ctype & \rightarrow \{"send", "bsend", "ssend", "rsend", "recv"\} \\
persist & \rightarrow \{TRUE, FALSE\} \\
match & \rightarrow Seq(\mathbb{N}) \\
message & \rightarrow Messages
\end{array}$$

A new request is appended to the requests function as described above. Each request record is accessible by the user process through its associated handle until that record is marked as deallocated either by successful application of a message completion operator such as MPI_WAIT or MPI_REQUEST_FREE. The handles are set to MPI_REQUEST_NULL at this time and become unaccessible to the user process.

3.2.4 Message buffers and buffer size

Users may wish to provide buffer space to the MPI system and allow the MPI system to manage that buffer space. Calls to MPI_BSEND, MPI_IBSEND, and MPI_BSEND_INIT use this buffer that is specified through MPI_BUFFER_ATTACH.

Not modeling data, the buffers are represented by a counting semaphore to track resource availability. Only one buffer can be attached to the MPI system for a process at a time. We approximate the use of the buffer space as follows. The user specifies how many messages can be stored in the buffer by the call to MPI_BUFFER_ATTACH. When a message is activated one buffer slot is consumed until the message is transmitted or canceled. Accordingly, MPI_BUFFER_DETACH blocks the process applying the operator until all buffered messages have either been transmitted or canceled.

$$\begin{array}{ll}
message_buffer: & 0..(\mathtt{N}-1) \rightarrow \mathbb{N} \\
bufsize: & 0..(\mathtt{N}-1) \rightarrow \mathbb{N}
\end{array}$$

The message_buffer variable is a function that represents the counting semaphore for each process. The bufsize variable is a function that represents the maximum values for each of the associated message_buffer variables.

3.3 Statuses

MPI operations return information to the user program in two ways. The first is the return value of a function. We do not model this. The second way information is returned to the user program is via the status object.

We model a status as a record with members as follows:

$$\begin{array}{ll} state & \rightarrow \{"defined", "undefined", "empty"\} \\ MPI\_SOURCE & \rightarrow 0..(\mathtt{N}-1) \cup \{\mathtt{MPI\_PROC\_NULL}, \mathtt{MPI\_ANY\_SOURCE}\} \\ MPI\_TAG & \rightarrow \mathtt{TAGS} \cup \{\mathtt{MPI\_ANY\_TAG}\} \\ MPI\_ERROR & \rightarrow \mathbb{N} \\ count & \rightarrow \mathbb{N} \\ canceled & \rightarrow \{TRUE, FALSE\} \end{array}$$

4 Collective Communications

5 Closing the model for use with model checking

Many things are left and specified by MPI. Among these are details on how messages are communicated between processes. So far we've introduced the request, status, and communicator records. Using the temporal logic of actions we now have sufficient structure in our model to specify the pairing, buffering, transmitting, and completing of messages.

5.1 Completing messages

5.1.1 Envelope matching

Envelopes match according to the operator Match shown in appendix A.

5.1.2 Pairing messages

Messages are paired together as a send request and a receive request. Program order must be observed on both the send and receive process when matching two requests. To enforce this policy, the operator that performs message pairing specifies the earliest active message in the sequence that has not been canceled, transmitted, or paired previously. Operator Pair contains the logic of this operation. Pairs of messages that have been started, not matched, not canceled, not transmitted, and where one message is a send and the other message is a receive can be paired. In addition we require messages to have matching envelopes.

5.1.3 Transmitting messages

Once messages are paired appropriately they may complete in any order. Thus it is not enough to model communication as the pairing of messages. The Transmit operator contains the logic involved in passing data from one process to another. Only messages that have been started, have not been canceled, have not previously been transmitted, and have been previously paired can be transmitted. The request is updated to reflect that the corresponding message has been transmitted.

5.1.4 Buffering messages

Message buffering can happen under two circumstances. The first is when the user specifically requests MPI to buffer the outgoing messages using commands such as MPI_IBSEND. These messages may be buffered at any time after the message is started and before the message has transmitted. The operator Buffer_bsend contains the logic to mark requests when messages have been buffered appropriately. Thereby allowing the sending process to continue when the corresponding message completion operator is applied.

When using MPI_SEND this system may choose to buffer the outgoing message. We allow this to happen at any time after the message is posted up until the message is transmitted or canceled. However it may also be the case that the MPI system will never buffer such a message. The operator Buffer_send performs this operation.

It is possible, from the user's perspective, for the message to be buffered and transmitted before the user program regains control. For this reason we allow the message to be buffered up to the point where the message is actually transmitted or canceled by the user.

6 Modeling MPI Programs

There are many ways to model programs in TLA. The +CAL tool makes it significantly easier to take this step [8]. We will describe a similar modeling paradigm that suites our needs.

In modeling MPI programs in connection with the TLA MPI specification, we assume for simplicity that all programs are written in the SPMD style. Although this is not required, it is required that all variables be declared as arrays as described in section 2.

It is also convenient to assume that all programs make only MPI function calls, although adding procedure calls is a relatively trivial extension. Closing the environment and making available other standard system procedures is an important area of research but is beyond the scope of this work.

6.1 Sequential execution

Let PC be an array $[0..(\mathtt{N}-1) \rightarrow \mathit{Labels}]$ such that each process $i \in 0..(\mathtt{N}-1)$ in the distributed computation has a program counter represented by $PC[i]$. The transition relation of a sequential program can be specified as a disjunct of conjuncts where each conjunct has (i) a current PC guard, (ii) the specified next PC after executing the conjunct, and (iii) an action associated with the current PC that modifies the state – perhaps only the PC itself.

All control statements can be modeled using the explicit PC and an IF construct provided by TLA.

6.2 Multiple step MPI procedures

As mentioned before, when using a multi-step MPI operator these can be compiled into some sequence of single-step operators. We present possible solutions for the seven contained in MPI that are not present in our TLA model.

The MPI operations can be modeled using a sequence of transitions with $proc$ being the pid of the process, "in"[2] being the starting PC of the call to MPI_SEND, "intermediate" being the middle PC, and "out" being the return PC, and the variable `req` $\in [0..(\mathtt{N}-1) \rightarrow \mathit{Request}]$. We also consider the status variable $stat : [0..(\mathtt{N}-1) \rightarrow \mathit{Status}]$ as defined in the appendix.

[2] Strings are valid PC values in TLA. Recall that the PC is a function whose domain is the set of pids and the range is in this case a string.

6.2.1 MPI_SEND

Applying MPI_SEND in a program having sequential execution can be implemented follows:

$$\begin{array}{ll} \lor & \land\ pc[proc] = \text{“}\texttt{in}\text{''} \\ & \land\ pc = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{intermediate}\text{''}] \\ & \land\ \texttt{MPI\_Isend}(\texttt{buf}, \texttt{count}, \texttt{datatype}, \texttt{dest}, \texttt{tag}, \texttt{com}, \texttt{req}, \texttt{proc}) \\ \lor & \land\ pc[proc] = \text{“}\texttt{intermediate}\text{''} \\ & \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{out}\text{''}] \\ & \land\ \texttt{MPI\_Wait}(\texttt{req}, \texttt{stat}, \texttt{proc}) \end{array}$$

6.2.2 MPI_BSEND

Applying MPI_BSEND is as follows:

$$\begin{array}{ll} \lor & \land\ pc[proc] = \text{“}\texttt{in}\text{''} \\ & \land\ pc = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{intermediate}\text{''}] \\ & \land\ \texttt{MPI\_Ibsend}(\texttt{buf}, \texttt{count}, \texttt{datatype}, \texttt{dest}, \texttt{tag}, \texttt{com}, \texttt{req}, \texttt{proc}) \\ \lor & \land\ pc[proc] = \text{“}\texttt{intermediate}\text{''} \\ & \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{out}\text{''}] \\ & \land\ \texttt{MPI\_Wait}(\texttt{req}, \texttt{stat}, \texttt{proc}) \end{array}$$

The restrictions on attaching buffers and managing the buffer space are identical.

6.2.3 MPI_SSEND

Applying MPI_SSEND is as follows:

$$\begin{array}{ll} \lor & \land\ pc[proc] = \text{“}\texttt{in}\text{''} \\ & \land\ pc = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{intermediate}\text{''}] \\ & \land\ \texttt{MPI\_Issend}(\texttt{buf}, \texttt{count}, \texttt{datatype}, \texttt{dest}, \texttt{tag}, \texttt{com}, \texttt{req}, \texttt{proc}) \\ \lor & \land\ pc[proc] = \text{“}\texttt{intermediate}\text{''} \\ & \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{out}\text{''}] \\ & \land\ \texttt{MPI\_Wait}(\texttt{req}, \texttt{stat}, \texttt{proc}) \end{array}$$

6.2.4 MPI_RSEND

Applying MPI_RSEND is as follows:

$$
\begin{array}{ll}
\lor & \land\ pc[proc] = \text{“}\texttt{in}\text{''} \\
& \land\ pc = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{intermediate}\text{''}] \\
& \land\ \texttt{MPI\_Irsend}(\texttt{buf}, \texttt{count}, \texttt{datatype}, \texttt{dest}, \texttt{tag}, \texttt{com}, \texttt{req}, \texttt{proc}) \\
\lor & \land\ pc[proc] = \text{“}\texttt{intermediate}\text{''} \\
& \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{out}\text{''}] \\
& \land\ \texttt{MPI\_Wait}(\texttt{req}, \texttt{stat}, \texttt{proc})
\end{array}
$$

6.2.5 MPI_RECV

Applying MPI_RECV is as follows:

$$
\begin{array}{ll}
\lor & \land\ pc[proc] = \text{“}\texttt{in}\text{''} \\
& \land\ pc = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{intermediate}\text{''}] \\
& \land\ \texttt{MPI\_Irecv}(\texttt{buf}, \texttt{count}, \texttt{datatype}, \texttt{source}, \texttt{tag}, \texttt{com}, \texttt{req}, \texttt{proc}) \\
\lor & \land\ pc[proc] = \text{“}\texttt{intermediate}\text{''} \\
& \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{out}\text{''}] \\
& \land\ \texttt{MPI\_Wait}(\texttt{req}, \texttt{stat}, \texttt{proc})
\end{array}
$$

6.2.6 MPI_SENDRECV

Overloading req and $stat$ to be arrays of records appropriately, MPI_SENDRECV could be implemented as follows:

$$
\begin{array}{ll}
\lor & \land\ pc[proc] = \text{“}\mathtt{in}\text{''} \\
& \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\mathtt{intermediate_recv}\text{''} \\
& \land\ \mathtt{MPI\_Isend}(\mathtt{sendbuf}, \mathtt{sendcount}, \mathtt{sendtype}, \mathtt{dest}, \mathtt{sendtag}, \mathtt{com}, \mathtt{req1}, \mathtt{proc}) \\
\lor & \land\ pc[proc] = \text{“}\mathtt{intermediate_recv}\text{''} \\
& \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\mathtt{wait}\text{''} \\
& \land\ \mathtt{MPI\_Irecv}(\mathtt{recvbuf}, \mathtt{recvcount}, \mathtt{recvtype}, \mathtt{source}, \mathtt{recvtag}, \mathtt{com}, \mathtt{req2}, \mathtt{proc}) \\
\lor & \land\ pc[proc] = \text{“}\mathtt{in}\text{''} \\
& \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\mathtt{intermediate_send}\text{''} \\
& \land\ \mathtt{MPI\_Irecv}(\mathtt{recvbuf}, \mathtt{recvcount}, \mathtt{recvtype}, \mathtt{source}, \mathtt{recvtag}, \mathtt{com}, \mathtt{req2}, \mathtt{proc}) \\
\lor & \land\ pc[proc] = \text{“}\mathtt{intermediate_send}\text{''} \\
& \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\mathtt{wait}\text{''} \\
& \land\ \mathtt{MPI\_Isend}(\mathtt{sendbuf}, \mathtt{sendcount}, \mathtt{sendtype}, \mathtt{dest}, \mathtt{sendtag}, \mathtt{com}, \mathtt{req1}, \mathtt{proc}) \\
\lor & \land\ pc = \text{“}\mathtt{wait}\text{''} \\
& \land\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\mathtt{out}\text{''} \\
& \land\ \mathtt{MPI\_Waitall}(2, [\mathtt{req}\ \mathtt{EXCEPT}\ ![\mathtt{proc}] = [0 \mapsto \mathtt{req1}[\mathtt{proc}], 1 \mapsto \mathtt{req2}[\mathtt{proc}]]], \mathtt{stat}, \mathtt{proc})
\end{array}
$$

6.2.7 MPI_SENDRECV_REPLACE

In addition to overloading req and $stat$ to be arrays of records appropriately we add a temporary variable for receiving the results. MPI_SENDRECV_REPLACE could be implemented as follows:

$$
\begin{array}{ll}
\vee & \wedge\ pc[proc] = \text{“}\texttt{in}''\\
& \wedge\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{intermediate}_{\texttt{r}}\texttt{ecv}''\\
& \wedge\ \texttt{MPI\_Isend}(\texttt{buf}, \texttt{sendcount}, \texttt{sendtype}, \texttt{dest}, \texttt{sendtag}, \texttt{com}, \texttt{req1}, \texttt{proc})\\
\vee & \wedge\ pc[proc] = \text{“}\texttt{intermediate}_{\texttt{r}}\texttt{ecv}''\\
& \wedge\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{wait}''\\
& \wedge\ \texttt{MPI\_Irecv}(\texttt{tempbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{source}, \texttt{recvtag}, \texttt{com}, \texttt{req2}, \texttt{proc})\\
\vee & \wedge\ pc[proc] = \text{“}\texttt{in}''\\
& \wedge\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{intermediate}_{\texttt{s}}\texttt{end}''\\
& \wedge\ \texttt{MPI\_Irecv}(\texttt{tempbuf}, \texttt{recvcount}, \texttt{recvtype}, \texttt{source}, \texttt{recvtag}, \texttt{com}, \texttt{req2}, \texttt{proc})\\
\vee & \wedge\ pc[proc] = \text{“}\texttt{intermediate}_{\texttt{s}}\texttt{end}''\\
& \wedge\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{wait}''\\
& \wedge\ \texttt{MPI\_Isend}(\texttt{sendbuf}, \texttt{sendcount}, \texttt{sendtype}, \texttt{dest}, \texttt{sendtag}, \texttt{com}, \texttt{req1}, \texttt{proc})\\
\vee & \wedge\ pc = \text{“}\texttt{wait}''\\
& \wedge\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}\texttt{copy}''\\
& \wedge\ \texttt{MPI\_Waitall}(2, [\texttt{req}\ \text{EXCEPT}\ ![\text{proc}] = [0 \mapsto \texttt{req1}[\text{proc}], 1 \mapsto \texttt{req2}[\text{proc}]]], \texttt{stat}, \texttt{proc})\\
\vee & \wedge\ pc = \text{“}\texttt{copy}''\\
& \wedge\ pc' = [pc\ EXCEPT\ ![proc] = \text{“}out'']\\
& \wedge\ sendbuf' = [buf\ EXCEPT\ ![proc] = temp[proc]]
\end{array}
$$

6.3 An example

An example program is included in Appendix A. This program exercises the immediate mode synchrnous send, along with the immediate mode receive. Processes are conceptually placed in a ring. Even ranked processes send to the neighbor with higher rank (mod ring size), synchronize on the barrier, and then receive from the neighbor having lower rank (again mod ring size). Odd ranked processes receive from the neighbor with lower rank, synchronize on the barrier and then send to the neighbor with higher rank.

The program is represented as a disjunct of conjuncts similar in style to Section 6.2. This operator has one parameter which is the process id of the process that is currently executing–therein we model the SPMD style where every process executes the same program image.

The next state relation for the entire system is the initial state of the model $Init$ and henceforth () the $Next$ relation that performs either a $Pair$, $Transmit$, $Buffer$, or $Proc$ move for some pid at any step.

7 Conclusions

The TLA model of MPI in connection with this paper describes the reactive behavior of all 35 point to point communication operations from chapter 3 of the MPI 1.1 standard.

We have closed the model for model checking single threaded programs that communicate via MPI point to point operations. We have provided the additional MPI operations necessary to initialize, determine the rank of a process, the size of a communicator's group, and exit according to the MPI standard.

References

[1] Martín Abadi, Leslie Lamport, and Stephan Merz. A TLA solution to the RPC-Memory specification problem. In M. Broy, S. Merz, and K. Spies, editors, *Formal System Specification: The RPC-Memory Specification Case Study*, volume 1169 of *Lecture Notes in Computer Science*, pages 21–66. Springer-Verlag, Berlin, 1996.

[2] Steven Barrus, Ganesh Gopalakrishnan, Robert M. Kirby, and Robert Palmer. Verification of MPI programs using SPIN. Technical Report UUCS-04-008, The University of Utah, 2004.

[3] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[4] Philippe Georgelin, Laurence Pierre, and Tin Nguyen. A formal specification of the MPI primitives and communication mechanisms. Technical report, LIM, 1999.

[5] John Harrison. Formal verification of square root algorithms. *Formal Methods in System Design*, 22(2):143–154, March 2003. Guest Editors: Ganesh Gopalakrishnan and Warren Hunt, Jr.

[6] IEEE standard for radix-independent floating-point arithmetic, ANSI/IEEE Std 854-1987.

[7] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley Professional, 2002.

[8] Leslie Lamport. A +CAL user's manual. http://research.microsoft.com/users/lamport/tla/p-manual.pdf, April 2006.

[9] Glenn R. Luecke, Silvia Spanoyannis, and Marina Kraeva. The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a cray T3E-600. *Concurrency and Computation: Practice and Experience*, 16(10):1037–1060, 2004.

[10] MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[11] Robert Palmer, Steven Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby. Gauss: A framework for verifying scientific computing software. In *SoftMC: Workshop on Software Model Checking*, number 953 in ENTCS, August 2005.

[12] Salman Pervez. Byte-range-locks using mpi-one-sided communication: a report. Technical report, The University of Utah, 2006. Submitted.

[13] Stephen F. Siegel and George Avrunin. Analysis of mpi programs. Technical Report UM-CS-2003-036, Department of Computer Science, University of Massachusetts Amherst, 2003.

[14] Stephen F. Siegel and George S. Avrunin. Verification of mpi-based software for scientific computation. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software*, volume 2989 of *LNCS*, pages 286–303, Barcelona, April 2004. Springer.

[15] Stephen F. Siegel and George S. Avrunin. Modeling wildcard-free MPI programs for verification. In *ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 95–106, Chicago, June 2005.

[16] Stephen F Siegel, Anastasia Mironova, and George S Avrunin nad Lori A Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In Lori Pollock and Mauro Pezz, editors, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 157–168, Portland, ME, July 2006.

A The Full Specification

MODULE *mpi_base*

The formal *MPI* library specification.

Robert Palmer

The University of *Utah*

School of Computing

Some notes : — Need to split the buffer rule into rules - one for user specified buffering and one for system provided buffering. – don't really know how

- Need to add deallocation of requests to the model as in *mpi_wait*.

- Need to add more sematnics.

- Need to cause buffers to be freed appropriately when a message is sent.

- Need to add a return code to indicate success or error and error handling.

- Need to fix the buffering of standard mode sends such that they might block forever.

EXTENDS *Naturals*, *TLC*, *Sequences*, *FiniteSets*

Constants are given values in the configuration file that accompanies this document: *mpi_base.cfg*

CONSTANTS

N,	The number of processes in the computation.
MAX_COMM,	The highest allowed handle value for a communicator. This is not in the standard but makes our model finite.
MAX_GROUP,	The highest allowed handle value for a group.
TYPES,	The set of user defined types.
TAGS,	The set of user defined tags.
SEND_IS_BUFFERED,	A flag to indicate whether sends are to be buffered.
RANK_ORDERINGS_SIGNIFICANT,	a flag to indicate whether all possible ranking orders should be considered in verification
MPI_COMM_WORLD,	The handle for *MPI_COMM_WORLD*.
MPI_ANY_SOURCE,	The wildcard source rank.
MPI_ANY_TAG,	The wildcard tag value.
MPI_PROC_NULL,	Section 3.11 Null Processes
MPI_REQUEST_NULL,	A special handle value for requests.

	Set this to 0 in the configuration file and make the initial values of the requests occupied to avoid an array out-of-bounds error.
MPI_SUCCESS,	The return value of a successful call to an *MPI* procedure.
MPI_IDENT,	5.4: Two communicator handles refer to the same communicator.
MPI_CONGRUENT,	The communicator handles are different; communicators differ only in context.
MPI_SIMILAR,	The communicator handles are different; communicators have the same group, however both context and ranking differ.
MPI_UNEQUAL,	The communicator handles are different; communicators have different groups, contexts, and rankings.
MPI_UNDEFINED,	A special rank returned to a process that is not a member of the queried communicator.
MPI_INT,	*MPI* defined *datatype* for integers
MPI_FLOAT,	*MPI* defined *datatype* for floating point numbers
UB	The upper bound on the tag range 19.27 − 19.31

MPI_GROUP_EMPTY \ ∗ The empty group

Variables represent the state of the *MPI* system at any given time. None of these state elements are specified by the standard. However they are useful to describe what is specified. In particular mention is made of handles that reference opaque objects. The communicator and requests arrays are such opaque objects that are referenced by integer handles that in our model are unique across both space and time (*i.e.*, the same value is used for *MPI_COMM_WORLD* on all processes for the entire execution etc.).

VARIABLES

communicator,	An array of communication universe objects.
bufsize,	The size of the user attached *message_buffer*.
message_buffer,	The user attached buffer.
requests,	A array of message requests lists, one per process. Although we do model the allocation of request objects by adding a structure to a list of requests, we are not modeling the freeing of requests more than setting the associated handle to *MPI_REQUEST_NULL*.
initialized,	An array of flags that indicate whether *MPI_Init*

	has been called by a given process.
collective,	The collective contexts for all communicators
group,	The array of groups
Memory	A model of memory for individual processes.

Type invariant

Memory is considered a *program_var*

$$mpi\_vars \triangleq \langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$$

$$\begin{aligned} Messages \triangleq [&src : (0 \,..\, (N-1)) \cup \{MPI\_ANY\_SOURCE\}, \quad \text{21.24} - \text{21.25}\\ &dest : (0 \,..\, (N-1)), \quad \text{19.39}\\ &msgtag : 0 \,..\, UB \cup \{MPI\_ANY\_TAG\}, \quad \text{19.28}\\ &dtype : TYPES \cup \{MPI\_FLOAT, MPI\_INT\},\\ &numelements : Nat,\\ &universe : (MPI\_COMM\_WORLD \,..\, (MPI\_COMM\_WORLD + MAX\_COMM)),\\ &state : \{\text{"send"}, \text{"recv"}\},\\ &addr : Nat] \end{aligned}$$

$$Message\_types \triangleq \{\text{"send"}, \text{"bsend"}, \text{"ssend"}, \text{"rsend"}, \text{"recv"}\}$$

$$Collective\_types \triangleq \{\text{"barrier"}\}$$

$$Collective\_states \triangleq \{\text{"in"}, \text{"out"}, \text{"vacant"}\}$$

$$\begin{aligned} Request \triangleq [&error : Nat,\\ &active : \text{BOOLEAN},\\ &transmitted : \text{BOOLEAN},\\ &buffered : \text{BOOLEAN},\\ &started : \text{BOOLEAN},\\ &cancelled : \text{BOOLEAN},\\ &deallocated : \text{BOOLEAN},\\ &ctype : Message\_types,\\ &persist : \text{BOOLEAN},\\ &match : Seq(Nat),\\ &message : Messages] \end{aligned}$$

$$Requests \triangleq [(0 \,..\, (N-1)) \rightarrow Seq(Request)]$$

22.1 – 22.8

$$\begin{aligned} \text{Statuses} \triangleq [&\text{state} : \{\text{"defined"}, \text{"undefined"}, \text{"empty"}\},\\ &MPI\_SOURCE : (0 \,..\, (N-1)) \cup \{MPI\_PROC\_NULL, MPI\_ANY\_SOURCE\},\\ &MPI\_TAG : TAGS \cup \{MPI\_ANY\_TAG\},\\ &MPI\_ERROR : Nat, \end{aligned}$$

count : Nat,
cancelled : BOOLEAN]

Refactored into *Memory* to allow for a uniform treatment of the model of memory and to facilitate modelling using pointer arithmetic for member accesses.

21.45 – 21.48

Status variables are explicitly allocated by the user. Therefore they are present in the *Memory* of individual processes. We will use a simple offset mechanism to return the individual member addresses within *Memory*.

22.1 – 22.8

$Status\_Cancelled(base) \triangleq base$
$Status\_Count(base) \triangleq base + 1$
$Status\_Source(base) \triangleq base + 2$
$Status\_Tag(base) \triangleq base + 3$
$Status\_Err(base) \triangleq base + 4$

$Initialized \triangleq [0 \,..\, (N-1) \rightarrow \{\text{"initialized"}, \text{"uninitialized"}, \text{"finalized"}\}]$

$MessageBuffers \triangleq [0 \,..\, (N-1) \rightarrow Nat]$

$BufferSizes \triangleq [0 \,..\, (N-1) \rightarrow Nat]$

Groups can be different on different processes.

$Group \triangleq [0 \,..\, (N-1) \rightarrow$ 138.37 – 138.38
$[MPI\_COMM\_WORLD \,..\, (MPI\_COMM\_WORLD + MAX\_GROUP) \rightarrow$
$[members : \text{SUBSET } (0 \,..\, (N-1)),$
$size : 0 \,..\, N,$
$ranking : [0 \,..\, (N-1) \rightarrow 0 \,..\, (N-1)],$
$invranking : [0 \,..\, (N-1) \rightarrow 0 \,..\, (N-1)]]]]$

$Communicator \triangleq [0 \,..\, (N-1) \rightarrow$
$[MPI\_COMM\_WORLD \,..\, (MPI\_COMM\_WORLD + MAX\_COMM) \rightarrow$
$[group : MPI\_COMM\_WORLD \,..\, (MPI\_COMM\_WORLD + MAX\_GROUP),$
$collective : MPI\_COMM\_WORLD \,..\, (MPI\_COMM\_WORLD + MAX\_COMM)]]]$

$Collective \triangleq [(MPI\_COMM\_WORLD \,..\, (MPI\_COMM\_WORLD + MAX\_COMM)) \rightarrow$
$[participants : \text{SUBSET } (0 \,..\, (N-1)),$
$root : (0 \,..\, (N-1)),$
$type : Collective\_types,$
$state : Collective\_states]]$

$Comm\_inv \triangleq communicator \in Communicator$

$Buff\_inv \triangleq bufsize \in BufferSizes$

$Msg\_buf\_inv \triangleq message\_buffer \in MessageBuffers$

$Initialized\_inv \triangleq initialized \in Initialized$

$Request\_inv \triangleq requests \in Requests$

$Col\_inv \triangleq collective \in Collective$

$group\_inv \triangleq group \in Group$

$MPI\_Type\_Invariant \triangleq$
$\land\ communicator \in Communicator$
$\land\ bufsize \in BufferSizes$
$\land\ message\_buffer \in MessageBuffers$
$\land\ initialized \in Initialized$
$\land\ requests \in Requests$
$\land\ collective \in Collective$

Make_request is a rule to simplify the expressions that create a new request object. Section 3.7.1

$Make\_request(err, act, com, sta, buf, cty, per, mat, can, mes) \triangleq$
$[error \mapsto err,$ The error code associated with this request
$active \mapsto act,$ The message was initiated
$transmitted \mapsto com,$ Data was transmitted by this message
$started \mapsto sta,$ Start this request
$buffered \mapsto buf,$ The data was copied from the input address
$cancelled \mapsto can,$ Whether the request was *cancelled*
$deallocated \mapsto$ FALSE, A new request is created in an allocated state
$ctype \mapsto cty,$ The type of message (send, *bsend*, *rsend*, or *ssend*)
$persist \mapsto per,$ Whether the request is a persistent communication
$match \mapsto mat,$ The matching < process,handle >
$message \mapsto mes]$ The message envelope associated with this request

The initial values for the *MPI* specification state variables. These are not specified by the standard, however these initial values make the TLA+ representation complete such that it can be verified using *TLC*.

$MPI\_Specification\_Init \triangleq$
$\land\ requests = [i \in (0 \mathrel{..} (N-1)) \mapsto$ Create an instance of *MPI_REQUEST_NULL* for each process.
$\langle Make\_request(0,$ FALSE, FALSE, FALSE, FALSE,
"send", FALSE, $\langle\rangle$, TRUE,
$[src \mapsto 0,$
$dest \mapsto 0,$
$msgtag \mapsto MPI\_ANY\_TAG,$
$dtype \mapsto 0,$
$numelements \mapsto 0,$
$universe \mapsto MPI\_COMM\_WORLD,$
$state \mapsto$ "send",
$addr \mapsto 0])\rangle]$
$\land\ bufsize = [i \in (0 \mathrel{..} (N-1)) \mapsto 0]$ Each process starts with no user attached buffer.
$\land\ message\_buffer = [i \in (0 \mathrel{..} (N-1)) \mapsto 0]$ Each process starts with no messages buffered.
$\land\ initialized = [i \in (0 \mathrel{..} (N-1)) \mapsto$ "uninitialized"$]$ Each process starts uninitialized.

$\land$ *communicator* $= [a \in 0 \,..\, (N-1) \mapsto [i \in MPI\_COMM\_WORLD \,..\, (MPI\_COMM\_WORLD + MAX$

$\land$ *collective* $= [i \in (MPI\_COMM\_WORLD \,..\, (MPI\_COMM\_WORLD + MAX\_COMM)) \mapsto$

$[participants \mapsto \{\},$

$root \mapsto 0,$

$type \mapsto$ "barrier",

$state \mapsto$ "vacant"]]

$\land$ $\lor$ $\land \neg RANK\_ORDERINGS\_SIGNIFICANT$ * In this case, choose an arbitrary ordering

$\land$ CHOOSE $f \in [0 \,..\, (N-1) \rightarrow 0 \,..\, (N-1)]$: * 12.41 − 12.42 order is not specified.

CHOOSE $finv \in [0 \,..\, (N-1) \rightarrow 0 \,..\, (N-1)]$: * The inverse of f

$\forall k \in$ DOMAIN f :

$\exists n \in 0 \,..\, (N-1)$:

$\land f[k] = n$

$\land finv[n] = k$

$\land \forall m \in$ DOMAIN $f : f[k] = f[m] \Rightarrow k = m$

$\land$ $group = [a \in 0 \,..\, (N-1) \mapsto [i \in (MPI\_COMM\_WORLD \,..$

$((MPI\_COMM\_WORLD + MAX\_GROUP))) \mapsto$

IF $i = MPI\_COMM\_WORLD$

THEN

$[members \mapsto \{x \in 0 \,..\, (N-1) :$ TRUE$\},$

$size \mapsto N,$

$ranking \mapsto f,$

$invranking \mapsto finv]$

ELSE

$[members \mapsto \{\},$

$size \mapsto 0,$

$ranking \mapsto [j \in 0 \,..\, (N-1) \mapsto 0],$

$invranking \mapsto [j \in 0 \,..\, (N-1) \mapsto 0]]]]$

$\lor$ $\land RANK\_ORDERINGS\_SIGNIFICANT$ * *in* this case, try all orderings

$\land \exists f \in [0 \,..\, (N-1) \rightarrow 0 \,..\, (N-1)]$: 12.41 − 12.42 order is not specified.

$\exists finv \in [0 \,..\, (N-1) \rightarrow 0 \,..\, (N-1)]$: The inverse of f

$\forall k \in$ DOMAIN f :

$\exists n \in 0 \,..\, (N-1)$:

$\land f[k] = n$

$\land finv[n] = k$

$\land \forall m \in$ DOMAIN f :

$\land f[k] = f[m] \Rightarrow k = m$

$\land group = [a \in 0 \,..\, (N-1) \mapsto$

$[i \in (MPI\_COMM\_WORLD \,..\, ((MPI\_COMM\_WORLD + MAX\_GROUP))) \mapsto$

IF $i = MPI\_COMM\_WORLD$

THEN

$[members \mapsto \{x \in 0 \,..\, (N-1) :$ TRUE$\},$

$size \mapsto N,$

$ranking \mapsto f,$

$invranking \mapsto finv]$

ELSE

$[members \mapsto \{\},$

$size \mapsto 0,$

$$
\begin{aligned}
&\mathit{ranking} &&\mapsto [j \in 0 \,..\, (N-1) \mapsto 0],\\
&\mathit{invranking} &&\mapsto [j \in 0 \,..\, (N-1) \mapsto 0]]]]
\end{aligned}
$$

A correct *MPI* program is one in which all messages that are posted are eventually transmitted or *cancelled*. A message that is posted but never transmitted is in error. It seems that a message that is transmitted but never completed locally may also be in error . . . I should check on this.

$$
\begin{aligned}
&\mathit{Messages\_sent\_are\_received\_and\_completed} \triangleq\\
&\quad \forall\, i \in (0 \,..\, (N-1)) :\\
&\quad\quad \forall\, m \in (1 \,..\, \mathit{Len}(\mathit{requests}[i])) :\\
&\quad\quad\quad \text{LET } r \triangleq \mathit{requests}[i][m] \text{ IN}\\
&\quad\quad\quad r.\mathit{active} \leadsto\\
&\quad\quad\quad \land\ \lor\ r.\mathit{transmitted}\\
&\quad\quad\quad \ \lor\ r.\mathit{cancelled}\\
&\quad\quad\quad \land\ \neg r.\mathit{active}
\end{aligned}
$$

There is some issue with regards to where the UNCHANGED identifiers should be living. I am using the following protocol:

1. Rules that have parameters that might be changed will declare the UNCHANGED value appropriately inside the rule for those parameters.
2. Variables that are passed as parameters to rules must be declared as UNCHANGED appropriately outside the rule unless the parameter might be modified by the rule when the rule is used.
3. Constants (such as a literal number, 0 for example) or CONSTANT values need not be declared as UNCHANGED .
4. *MPI* based rules always indicate the UNCHANGED terms for *MPI* state variables. Program models also indicate UNCHANGED for *MPI* variables only when no *MPI* rule is fired in that transition.

Conventions on parameters.

1. Parameters that are set (*i.e.*, OUT or *INOUT*) are all arrays from $0 \,..\, (N-1)$ with one instance of each object for each process in the model.
2. All other parameters (*i.e.*, IN) are the single instance of the variable value being passed, or are constant.

These rules perform the communication or "matching" of messages that is necessary to complete the *MPI* communication infrastructure. They are in no way specified in the standard, except that messages are spoken of as being transmitted from one process to another and matching.

$\alpha \rightarrow \beta \rightarrow$ BOOLEAN

No change in state

$Match(a, b) \triangleq$

$\land \quad Assert((a.state = \text{"recv"} \land b.state = \text{"send"}) \lor$
$\qquad (a.state = \text{"send"} \land b.state = \text{"recv"}),$
$\qquad \text{"Error: Match attempted with two send or receives."})$

$\land \quad (a.src = b.src \lor a.src = MPI\_ANY\_SOURCE \lor b.src = MPI\_ANY\_SOURCE)$ 21.14 – 21.15

$\land \quad a.dest = b.dest$

$\land \quad a.dtype = b.dtype$ 23.17, 23.24 – 23.27

$\land \quad (a.msgtag = b.msgtag \lor a.msgtag = MPI\_ANY\_TAG \lor b.msgtag = MPI\_ANY\_TAG)$ 21.15 – 21.16

$\land \quad a.universe = b.universe$ 19.34 – 19.37

$\land \quad \lnot a.src = MPI\_PROC\_NULL$ 60.48 – 61.1

$\land \quad \lnot a.dest = MPI\_PROC\_NULL$

$\land \quad \lnot b.src = MPI\_PROC\_NULL$

$\land \quad \lnot b.dest = MPI\_PROC\_NULL$

21.13 – 21.14 count need not be matched in point to point messages.

Messages match in program order pairwise between processes, however they may complete in a nondeterministic order on both the sender and receiver. This tends to imply that Communicate should in fact be two rules. And it also seems to imply that completion of a message can happen on one side and then on the other also in a non-deterministic way. Therefore *Transmit* should complete only one side of the communication.

Pairs messages together such that they result in a communication eventually.

$Pair \triangleq$

$\land \exists i \in 0 \ldots (N-1):$
$\quad \exists j \in 0 \ldots (N-1):$
$\quad\quad \exists m \in 1 \ldots Len(requests[i]):$
$\quad\quad\quad \exists n \in 1 \ldots Len(requests[j]):$
$\quad\quad\quad \text{LET } a \triangleq requests[i][m] \text{ IN}$
$\quad\quad\quad \text{LET } b \triangleq requests[j][n] \text{ IN}$
$\quad\quad\quad \land a.started$
$\quad\quad\quad \land b.started$
$\quad\quad\quad \land \lnot a.cancelled$
$\quad\quad\quad \land \lnot b.cancelled$
$\quad\quad\quad \land \lnot a.transmitted$
$\quad\quad\quad \land \lnot b.transmitted$
$\quad\quad\quad \land \lor \land a.message.state = \text{"send"}$
$\quad\quad\quad\quad\quad \land b.message.state = \text{"recv"}$
$\quad\quad\quad\quad \lor \land a.message.state = \text{"recv"}$
$\quad\quad\quad\quad\quad \land b.message.state = \text{"send"}$
$\quad\quad\quad \land a.match = \langle\rangle$
$\quad\quad\quad \land b.match = \langle\rangle$
$\quad\quad\quad \land Match(a.message, b.message)$
$\quad\quad\quad \land \forall r \in 1 \ldots Len(requests[i]):$ This conjunct enforces the fifo
$\quad\quad\quad\quad \forall s \in 1 \ldots Len(requests[j]):$
$\quad\quad\quad\quad \text{LET } c \triangleq requests[i][r] \text{ IN}$
$\quad\quad\quad\quad \text{LET } d \triangleq requests[j][s] \text{ IN}$

```
∧ ∨ ∧ c.message.state = “send”
      ∧ d.message.state = “recv”
  ∨ ∧ c.message.state = “recv”
      ∧ d.message.state = “send”
∧ Match(c.message, d.message)
∧ a.started
∧ b.started
∧ ¬c.cancelled
∧ ¬c.transmitted
∧ ¬d.cancelled
∧ ¬d.transmitted
∧ c.match = ⟨⟩
∧ d.match = ⟨⟩
⇒ ∧ m ≤ r        Section 3.7.4
   ∧ n ≤ s
```

```
      ∧ requests′  = [requests EXCEPT
                        ![i] =
                          [@ EXCEPT ![m] =
                            [@ EXCEPT !.match = ⟨j, n⟩]],
                        ![j] =
                          [@ EXCEPT ![n] =
                            [@ EXCEPT !.match = ⟨i, m⟩]]]
  ∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, initialized, collective⟩
```

Causes the communication that is already paired to complete.
Need to move arrays of data too.

```
Transmit ≜
  ∧ ∃ i ∈ 0 .. (N − 1) :
      ∃ j ∈ 1 .. Len(requests[i]) :
       LET m ≜ requests[i][j]IN
        ∧ m.started
        ∧ ¬m.cancelled
        ∧ ¬m.transmitted
        ∧ m.match ≠ ⟨⟩
        ∧ requests′ = [requests EXCEPT ![i] =
                          [@ EXCEPT ![j] =
                            [@ EXCEPT !.transmitted = TRUE]]]
        ∧ IF ¬requests[m.match[1]][m.match[2]].transmitted
            THEN
             IF m.message.state = “recv”
              THEN Memory′ = [Memory EXCEPT ![i] = [@ EXCEPT ![m.message.addr] = Memory[m.matc
              ELSE  Memory′ = [Memory EXCEPT ![m.match[1]] = [@ EXCEPT ![requests[m.match[1]][m.ma
            ELSE
             UNCHANGED ⟨Memory⟩
```

$\land$ IF $m.ctype =$ "bsend"
  THEN
   $message\_buffer' = [message\_buffer$ EXCEPT $![i] = @ - 1]$
  ELSE
   UNCHANGED $\langle message\_buffer \rangle$
$\land$ UNCHANGED $\langle group, communicator, bufsize, initialized, collective \rangle$

The specification indicates that messages are buffered in an asyncronous manner. The rule *Buffer* is not part of the standard but necessary to allow buffering to complete asynchronously.

$Buffer \triangleq$
$\lor \land \exists i \in (0 \,..\, (N-1)) :$
  $\exists m \in 1 \,..\, Len(requests[i]) :$
   $\land requests[i][m].started$
   $\land requests[i][m].active$
   $\land \neg requests[i][m].buffered$
   $\land \neg requests[i][m].cancelled$
   $\land \neg requests[i][m].transmitted$
   $\land \lor \land requests[i][m].ctype =$ "bsend" Buffering is provided explicitly by the user.
     $\land requests' =$
      $[requests$ EXCEPT $![i] =$
       $[@$ EXCEPT $![m] =$
        $[@$ EXCEPT $!.buffered =$ TRUE$]]]$
    $\lor \land requests[i][m].ctype =$ "send" Buffering may be provided by the system.
     $\land \lor requests' =$
       $[requests$ EXCEPT $![i] =$
        $[@$ EXCEPT $![m] =$
         $[@$ EXCEPT $!.buffered =$ TRUE$]]]$
      $\lor$ UNCHANGED $requests$
$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, initialized, collective \rangle$

General Comments:

1.19.23 – 19.24 The message source is provided in the envelope implicitly. Operators in our model must be passed this information as a parameter. As such we extend the argument list to include *proc*, being the unique identity of the applying process.

Section 3.2 Blocking Send and Receive Operations

Section 3.2.1 Blocking send

Can these really be done in a single transition? I am thinking that it is not possible under an interleaving semantics. In particular, either the send must be two transitions or the receive must be two transitions, it cannot be the case that they are both only one transition.

$MPI\_Send(buf, count, datatype, dest, tag, comm, proc) \triangleq$
$MPI\_Isend; MPI\_Wait$

Section 3.2.4 Blocking receive If receive is modeled using only one transition, it is just a combination of the *MPI_Irecv* and Communicate rules.

$MPI\_Recv(buf, count, datatype, source, tag, comm, status) \triangleq$
$MPI\_Irecv; MPI\_Wait$

Section 3.2.5 Return status

Returns in count the number of data elements in the message represented by status.

$MPI\_Get\_count(status, datatype, count, return, proc) \triangleq$ 22.24 – 22.37
$\land Assert(Memory[proc][Status\_Cancelled(status)] = \text{FALSE},$ 54.47
"Error: count is undefined on a status from a cancelled message.")
$\land Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Get_count called before process was initialized.")
$\land Memory' = [Memory \text{ EXCEPT } ![proc] = [@ \text{ EXCEPT } ![count] = Memory[proc][Status\_Count(status)]]]$
$\land \text{UNCHANGED } \langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$

Section 3.4 Communication Modes

Notes: These, like the above blocking communications really should be modeled using two transitions. In this way, the interleaving semantics is able to schedule another process to complete the communications.

$MPI\_Bsend(buf, count, datatype, dest, tag, comm, proc) \triangleq$

$MPI\_Ssend(buf, count, datatype, dest, tag, comm, proc) \triangleq$

$MPI\_Rsend(buf, count, datatype, dest, tag, comm, proc) \triangleq$

Section 3.6 *Buffer* allocation and usage

We ignore the buffer argument as data is abstracted away in our model. Buffering is modeled as a counting semaphore, keeping track of the resources available but not exactly which resources are used or what is done with those resources.

Return value is unspecified.

$MPI\_Buffer\_attach(buffer, size, return, proc) \triangleq$ < 34.17 – 34.33 >
$\land Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Buffer_attach called with proc not in initialized state.")
$\land Assert(bufsize[proc] = 0,$ 34.32
"Error: MPI_Buffer_attach called when processes buffer is non-zero.")

$\land\ bufsize' = [bufsize\ \text{EXCEPT}\ ![proc] = size[proc]]$ $< /34.17 - 34.33 >$
$\land\ \text{UNCHANGED}\ \langle group, communicator, message\_buffer, requests, initialized, collective\rangle$
$\land\ \text{UNCHANGED}\ Memory$

Again we ignore the *buffer_addr* argument as we are abstracting data.

The standard does not indicate what the result is when there is no buffer currently attached.

$MPI\_Buffer\_detach(buffer\_addr, size, return, proc) \triangleq$ $< 34.36 - 35.2 >$
$\land\ Assert(initialized[proc] = \text{"initialized"},$ 200.10 − 200.12
"Error: MPI_Buffer_detach called with proc not in initialized state.")
$\land\ Assert(bufsize[proc] \neq 0,$
"Error: MPI_Buffer_detach called when no buffer is currently associated with this process.")
$\land\ bufsize' = [bufsize\ \text{EXCEPT}\ ![proc] = 0]$ 34.46
$\land\ \forall j \in 1 \mathrel{..} Len(requests[proc]) :$ 34.47
$requests[proc][j].ctype = \text{"bsend"} \Rightarrow requests[proc][j].transmitted$
$< /34.36 - 35.2 >$
$\land\ Memory = [Memory\ \text{EXCEPT}\ ![proc] = [@\ \text{EXCEPT}\ ![size] = bufsize[proc]]]$ 34.47
$\land\ \text{UNCHANGED}\ \langle group, communicator, message\_buffer, requests, initialized, collective\rangle$

Section 3.7.2 Communication initiation

Notes: I am not sure how to model this construct. The main problem lies in the nondeterministic buffering scheme that the standard referrs to. For a correct program one must expect no buffering, however is it possible to write a program in such a way as to require synchronous handshakes?

Start a non-blocking standard send. 38.17 − 38.35, 58.13 − 58.18

$MPI\_Isend(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq$
$\land\ Assert(initialized[proc] = \text{"initialized"},$ 200.10 − 200.12
"Error: MPI_Isend called with proc not in initialized state.")
$\land\ Assert(proc \in group[proc][communicator[proc][comm].group].members,$
"Error: MPI_Isend called on a communicator which this process is not a member of.")
$\land\ \text{LET}\ msg \triangleq$
$[addr \mapsto buf,$
$src \mapsto group[proc][communicator[proc][comm].group].ranking[proc],$
$dest \mapsto dest,$
$msgtag \mapsto tag,$
$dtype \mapsto datatype,$
$numelements \mapsto count,$
$universe \mapsto comm,$
$state \mapsto \text{"send"}]$
IN
$requests' = [requests\ \text{EXCEPT}\ ![proc] =$ 40.40, 35.37 − 35.39
$@ \circ \langle Make\_request(0, \text{TRUE}, \text{FALSE}, \text{TRUE}, \text{TRUE}, \text{"send"}, \text{FALSE}, \langle\rangle, \text{FALSE}, msg)\rangle]$
$\land\ Memory' = [Memory\ \text{EXCEPT}\ ![proc] = [@\ \text{EXCEPT}\ ![request] = Len(requests[proc]) + 1]]$ 40.41

$\wedge$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, initialized, collective \rangle$

Set up a non-blocking buffered send. 39.1 – 39.19, 58.13 – 58.18

$MPI\_Ibsend(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq$
$\wedge$ $Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Ibsend called with proc not in initialized state.")
$\wedge$ $Assert(message\_buffer[proc] < bufsize[proc]$, 28.6, 35.34 – 35.35
"Error: MPI_Ibsend called when insufficient buffering was available.")
$\wedge$ $Assert(proc \in group[proc][communicator[proc][comm].group].members$,
"Error: MPI_Ibsend called on a communicator which this process is not a member of.")
$\wedge$ LET $msg \triangleq$
$[addr \mapsto buf$,
$src \mapsto group[proc][communicator[proc][comm].group].ranking[proc]$,
$dest \mapsto dest$,
$msgtag \mapsto tag$,
$dtype \mapsto datatype$,
$numelements \mapsto count$,
$universe \mapsto comm$,
$state \mapsto$ "send"$]$
IN
$requests' = [requests$ EXCEPT $![proc] =$ 40.40
$@ \circ \langle Make\_request(0,$ TRUE, FALSE, TRUE, TRUE, "bsend", FALSE, $\langle\rangle$, FALSE, $msg)\rangle]$
$\wedge$ $Memory' = [Memory$ EXCEPT $![proc] = [@$ EXCEPT $![request] = Len(requests[proc]) + 1]]$ 40.41
$\wedge$ $message\_buffer' = [message\_buffer$ EXCEPT $![proc] = @ + 1]$ 28.6 Consume necessary buffer space
$\wedge$ UNCHANGED $\langle group, communicator, bufsize, initialized, collective \rangle$

Tested

Set up a non-blocking synchronous send. 39.21 – 39.39, 58.13 – 58.18

$MPI\_Issend(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq$
$\wedge$ $Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Issend called with proc not in initialized state.")
$\wedge$ $Assert(proc \in group[proc][communicator[proc][comm].group].members$,
"Error: MPI_Issend called on a communicator which this process is not a member of.")
$\wedge$ LET $msg \triangleq$
$[addr \mapsto buf$,
$src \mapsto group[proc][communicator[proc][comm].group].ranking[proc]$,
$dest \mapsto dest$,
$msgtag \mapsto tag$,
$dtype \mapsto datatype$,
$numelements \mapsto count$,
$universe \mapsto comm$,
$state \mapsto$ "send"$]$
IN
$requests' = [requests$ EXCEPT $![proc] =$ 40.40
$@ \circ \langle Make\_request(0,$ TRUE, FALSE, TRUE, FALSE, "ssend", FALSE, $\langle\rangle$, FALSE, $msg)\rangle]$

$\land$ *Memory'* $=$ [*Memory* EXCEPT ![*proc*] $=$ [@ EXCEPT ![*request*] $=$ *Len*(*requests*[*proc*]) $+ 1$]] 40.41
$\land$ UNCHANGED $\langle$*group*, *communicator*, *message_buffer*, *bufsize*, *initialized*, *collective*$\rangle$

Set up a non-blocking ready send. 40.1 – 40.19, 58.13 – 58.18

MPI_Irsend(*buf*, *count*, *datatype*, *dest*, *tag*, *comm*, *request*, *return*, *proc*) $\triangleq$
$\land$ *Assert*(*initialized*[*proc*] $=$ "initialized", 200.10 – 200.12
"Error: MPI_Irsend called with proc not in initialized state.")
$\land$ *Assert*(*proc* $\in$ *group*[*proc*][*communicator*[*proc*][*comm*].*group*].*members*,
"Error: MPI_Irsend called on a communicator which this process is not a member of.")
$\land$ *Assert*($\exists\, k \in$ (1 . . *Len*(*requests*[*dest*])) : 37.6 – 37.8
$\land$ *requests*[*dest*][*k*].*active*
$\land \neg$*requests*[*dest*][*k*].*transmitted*
$\land \neg$*requests*[*dest*][*k*].*cancelled*
$\land$ *Match*(*requests*[*proc*][*request*].*message*, *requests*[*dest*][*k*].*message*),
"Error: MPI_Start tried to start a rsend request when no matching message exists.")
$\land$ *requests'* $=$ [*requests* EXCEPT ![*proc*] $=$ 40.40
LET *msg* $\triangleq$
[*addr* $\mapsto$ *buf*,
src $\mapsto$ *group*[*proc*][*communicator*[*proc*][*comm*].*group*].*ranking*[*proc*],
dest $\mapsto$ *dest*,
msgtag $\mapsto$ *tag*,
dtype $\mapsto$ *datatype*,
numelements $\mapsto$ *count*,
universe $\mapsto$ *comm*,
state $\mapsto$ "send"]
IN
@ $\circ$ $\langle$*Make_request*(0, TRUE, FALSE, TRUE, FALSE, "rsend", FALSE, $\langle\rangle$, FALSE, *msg*)$\rangle$]
$\land$ *Memory'* $=$ [*Memory* EXCEPT ![*proc*] $=$ [@ EXCEPT ![*request*] $=$ *Len*(*requests*[*proc*]) $+ 1$]] 40.41
$\land$ UNCHANGED $\langle$*group*, *communicator*, *bufsize*, *message_buffer*, *requests*, *initialized*, *collective*$\rangle$

Set up a non-blocking receive. 40.21 – 40.39, 58.13 – 58.18

MPI_Irecv(*buf*, *count*, *datatype*, *source*, *tag*, *comm*, *request*, *return*, *proc*) $\triangleq$
$\land$ *Assert*(*initialized*[*proc*] $=$ "initialized", 200.10 – 200.12
"Error: MPI_Irecv called with proc not in initialized state.")
$\land$ *Assert*(*proc* $\in$ *group*[*proc*][*communicator*[*proc*][*comm*].*group*].*members*,
"Error: MPI_Irecv called on a communicator which this process is not a member of.")
$\land$ LET *msg* $\triangleq$
[*addr* $\mapsto$ *buf*,
src $\mapsto$ *source*,
dest $\mapsto$ *group*[*proc*][*communicator*[*proc*][*comm*].*group*].*ranking*[*proc*],
msgtag $\mapsto$ *tag*,
dtype $\mapsto$ *datatype*,
numelements $\mapsto$ *count*,
universe $\mapsto$ *comm*,

$state \quad \mapsto$ "recv"]

IN

$requests' = [requests$ EXCEPT $![proc] =$ 40.40

$@ \circ \langle Make\_request(0,$ TRUE, FALSE, TRUE, FALSE, "recv", FALSE, $\langle\rangle$, FALSE, $msg)\rangle]$

$\wedge\ Memory' = [Memory$ EXCEPT $![proc] = [@$ EXCEPT $![request] = Len(requests[proc]) + 1]]$ 40.41

$\wedge$ UNCHANGED $\langle group, communicator, message\_buffer, bufsize, initialized, collective\rangle$

Section 3.7.3 Communication Completion

Would if . . . then . . . else be a better, more readable form here? Maybe not because we need to block.

Wait for request to complete. Return information about the message in status. 41.23 – 42.6

No specification on what the status value is when a send is posted with *MPI_PROC_NULL*

Specifies next state for status and request

$MPI\_Wait(request, status, return, proc) \triangleq$

LET $r \triangleq requests[proc][Memory[proc][request]]$ IN

$\wedge\ Assert(initialized[proc] =$ "initialized", 200.10 – 200.12

"Error: MPI_Wait called with proc not in initialized state.")

$\wedge \vee \wedge Memory[proc][request] \neq MPI\_REQUEST\_NULL$ 41.32 – 41.39 The request handle is not the null han

$\wedge\ r.active$ The request is active.

$\wedge \vee \wedge r.message.src \neq MPI\_PROC\_NULL$ The message source is not null

$\wedge\ r.message.dest \neq MPI\_PROC\_NULL$ The message destination is not null

41.32 – Blocks until complete

$\wedge \vee r.transmitted$ The communication actually happened or

$\vee\ r.cancelled$ the communication got *cancelled* by the user program or

$\vee\ r.buffered$ the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).

A status object for a completed communication.

$\wedge\ Memory' =$

$[Memory$ EXCEPT $![proc] =$ 41.36

$[@$ EXCEPT $![Status\_Cancelled(status)] = r.cancelled \wedge \neg r.transmitted,$ 54.46

$![Status\_Count(status)] = r.message.numelements,$

$![Status\_Source(status)] = r.message.src,$

$![Status\_Tag(status)] = r.message.msgtag,$

$![Status\_Err(status)] = r.error,$

$![request] =$ IF $r.persist$ THEN $@$ ELSE $MPI\_REQUEST\_NULL]]$ 41.32 – 41.35,

$\vee \wedge \vee r.message.src = MPI\_PROC\_NULL$ The source or destination was actually

$\vee\ r.message.dest = MPI\_PROC\_NULL$ the null process

$\wedge\ Memory' = [Memory$ EXCEPT $![proc] =$ 41.36

$[@$ EXCEPT $![Status\_Cancelled(status)] = r.cancelled,$

$![Status\_Count(status)] = 0,$

$![Status\_Source(status)] = MPI\_PROC\_NULL,$

$![Status\_Tag(status)] = MPI\_ANY\_TAG,$

```
                    ![Status_Err(status)] = 0,
                    ![request] = IF r.persist THEN @ ELSE MPI_REQUEST_NULL]]   41.32 – 41.35,
   ∧ requests' = [requests EXCEPT ![proc] =   58.34
               [@ EXCEPT ![Memory[proc][request]] =
                 IF r.persist
                  THEN
                   [@ EXCEPT !.active = FALSE]
                  ELSE
                   [@ EXCEPT
                      !.active = FALSE,
                      !.deallocated = TRUE]]]
 ∨ ∧ ∨ ¬r.active                             41.40 – 41.41 The request is not active
     ∨ Memory[proc][request] = MPI_REQUEST_NULL   or the request handle is null
   ∧ Memory' = [Memory EXCEPT ![proc] =   41.36
                [@ EXCEPT ![Status_Cancelled(status)] = FALSE,
                          ![Status_Count(status)] = 0,
                          ![Status_Source(status)] = MPI_ANY_SOURCE,
                          ![Status_Tag(status)] = MPI_ANY_TAG,
                          ![Status_Err(status)] = 0]]
   ∧ UNCHANGED ⟨requests⟩
 ∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, initialized, collective⟩
```

Test whether the request referenced has completed.
Specifies next state for request, flag, and status.

```
MPI_Test(request, flag, status, return, proc) ≜
 LET r ≜ requests[proc][Memory[proc][request]] IN
 ∧ Assert(initialized[proc] = "initialized",   200.10 – 200.12
        "Error: MPI_Test called with proc not in initialized state.")
 ∧ ∨ ∧ Memory[proc][request] ≠ MPI_REQUEST_NULL   The request handle is not the null handle.
     ∧ r.active                            The request is active.
     ∧ ∨ ∧ r.message.src ≠ MPI_PROC_NULL   The message source is not null
         ∧ r.message.dest ≠ MPI_PROC_NULL   The message destination is not null
                                            42.20 – 42.21
         ∧ IF ∨ r.transmitted   The communication actually happened or
              ∨ r.cancelled     the communication got cancelled by the user program or
              ∨ r.buffered      the communication got buffered either into explicit user provided
                                buffer space or into system provided buffer space (if regular send is used).
           THEN
            ∧ Memory' = [Memory EXCEPT ![proc] =
                 [@ EXCEPT ![Status_Cancelled(status)] = r.cancelled ∧ ¬r.transmitted,
                           ![Status_Count(status)] = r.message.numelements,
                           ![Status_Source(status)] = r.message.src,
                           ![Status_Tag(status)] = r.message.msgtag,
                           ![Status_Err(status)] = r.error,
```

$![\mathit{flag}] = \text{TRUE},$
$![\mathit{request}] = \text{IF } r.\mathit{persist} \text{ THEN } @ \text{ ELSE } \mathit{MPI\_REQUEST\_NULL}]]$
$\land \mathit{requests}' =$
$[\mathit{requests} \text{ EXCEPT } ![\mathit{proc}] =$
$[@ \text{ EXCEPT } ![\mathit{Memory}[\mathit{proc}][\mathit{request}]] =$
$[@ \text{ EXCEPT } !.\mathit{active} = \text{FALSE}]]]$ 42.22 – 42.23, 58.34 Not modeling deallocation
ELSE 42.23 – 42.24
$\land \mathit{Memory}' = [\mathit{Memory} \text{ EXCEPT } ![\mathit{proc}] = [@ \text{ EXCEPT } ![\mathit{flag}] = \text{FALSE}]]$ status is undefined 4
$\land \text{UNCHANGED } \langle \mathit{requests} \rangle$
$\lor \land \lor r.\mathit{message}.\mathit{src} = \mathit{MPI\_PROC\_NULL}$ The source or destination were actually
$\lor r.\mathit{message}.\mathit{dest} = \mathit{MPI\_PROC\_NULL}$ the null process 42.29 – 42.31
$\land \mathit{Memory}' = [\mathit{Memory} \text{ EXCEPT } ![\mathit{proc}] =$
$[@ \text{ EXCEPT } ![\mathit{Status\_Cancelled}(\mathit{status})] = \text{FALSE},$
$![\mathit{Status\_Count}(\mathit{status})] = 0,$
$![\mathit{Status\_Source}(\mathit{status})] = \mathit{MPI\_PROC\_NULL},$
$![\mathit{Status\_Tag}(\mathit{status})] = \mathit{MPI\_ANY\_TAG},$
$![\mathit{Status\_Err}(\mathit{status})] = 0,$
$![\mathit{flag}] = \text{TRUE},$
$![\mathit{request}] = \text{IF } r.\mathit{persist} \text{ THEN } @ \text{ ELSE } \mathit{MPI\_REQUEST\_NULL}]$
$\land \mathit{requests}' =$
$[\mathit{requests} \text{ EXCEPT } ![\mathit{proc}] =$
$[@ \text{ EXCEPT } ![\mathit{Memory}[\mathit{proc}][\mathit{request}]] =$
$[@ \text{ EXCEPT } !.\mathit{active} = \text{FALSE}]]]$ 42.22 – 42.23, 58.34 Not modeling deallocation
$\lor \land \lor \neg r.\mathit{active}$ The request is not active or the request
$\lor \mathit{Memory}[\mathit{proc}][\mathit{request}] = \mathit{MPI\_REQUEST\_NULL}$ handle is null 42.29 – 42.31
$\land \mathit{Memory}' = [\mathit{Memory} \text{ EXCEPT } ![\mathit{proc}] =$
$[@ \text{ EXCEPT } ![\mathit{Status\_Cancelled}(\mathit{status})] = \text{FALSE},$
$![\mathit{Status\_Count}(\mathit{status})] = 0,$
$![\mathit{Status\_Source}(\mathit{status})] = \mathit{MPI\_ANY\_SOURCE},$
$![\mathit{Status\_Tag}(\mathit{status})] = \mathit{MPI\_ANY\_TAG},$
$![\mathit{Status\_Err}(\mathit{status})] = 0,$
$![\mathit{flag}] = \text{TRUE}]]$
$\land \text{UNCHANGED } \langle \mathit{requests} \rangle$
$\land \text{UNCHANGED } \langle \mathit{group}, \mathit{communicator}, \mathit{bufsize}, \mathit{message\_buffer}, \mathit{initialized}, \mathit{collective} \rangle$

Frees the request specified.
Modifies request.

$\mathit{MPI\_Request\_free}(\mathit{request}, \mathit{return}, \mathit{proc}) \triangleq$
$\land \mathit{Assert}(\mathit{initialized}[\mathit{proc}] =$ “initialized”, 200.10 – 200.12
“MPI_Request_free called with proc not in initialized state.”)
$\land \mathit{Assert}(\neg \mathit{requests}[\mathit{proc}][\mathit{Memory}[\mathit{proc}][\mathit{request}]].\mathit{active},$ 43.37 – 43.39
“MPI_Request_free called with an inactive request.”)
$\land \mathit{Memory}' = [\mathit{Memory} \text{ EXCEPT } ![\mathit{proc}] = [@ \text{ EXCEPT } ![\mathit{request}] = \mathit{MPI\_REQUEST\_NULL}]]$ 43.20 Not m
$\land \text{UNCHANGED } \langle \mathit{group}, \mathit{communicator}, \mathit{bufsize}, \mathit{message\_buffer}, \mathit{requests}, \mathit{initialized}, \mathit{collective} \rangle$

Section 3.7.5 Multiple Completions

Wait for one of the requests referenced in *array_of_requests* to complete.
Specifies next state for index and status

$MPI\_Waitany(count, array\_of\_requests, index, status, return, proc) \triangleq$

$\quad \text{LET } r(v) \triangleq requests[proc][Memory[proc][array\_of\_requests + v]] \text{ IN}$

$\quad \land Assert(initialized[proc] = \text{"initialized"},$ 200.10 – 200.12
$\qquad\qquad \text{"Error: MPI\_Waitany called with proc not in initialized state."})$

$\quad \land \lor \exists i \in (0 \,..\, (count - 1)) :$ 45.44 – 45.46 Blocks–chooses arbitrarily one that satisfies the following:

$\qquad \land Memory[proc][array\_of\_requests + i] \neq MPI\_REQUEST\_NULL$ The request handle is not the null

$\qquad \land r(i).active$ The request is active.

$\qquad \land \lor \land r(i).message.src \neq MPI\_PROC\_NULL$ The message source is not null

$\qquad\qquad \land r(i).message.dest \neq MPI\_PROC\_NULL$ The message destination is not null

$\qquad\qquad \land \lor r(i).transmitted$ The communication actually happened or

$\qquad\qquad\quad \lor r(i).cancelled$ the communication got *cancelled* by the user program or

$\qquad\qquad\quad \lor r(i).buffered$ the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).

$\qquad\qquad \land Memory' = [Memory \text{ EXCEPT } ![proc] =$ 45.46 – 45.47

$\qquad\qquad\quad [@ \text{ EXCEPT}$

$\qquad\qquad\quad ![Status\_Source(status)] = r(i).message.src,$ 45.47 – 45.48

$\qquad\qquad\quad ![Status\_Tag(status)] = r(i).message.msgtag,$

$\qquad\qquad\quad ![Status\_Err(status)] = r(i).error,$

$\qquad\qquad\quad ![Status\_Count(status)] = r(i).message.numelements,$

$\qquad\qquad\quad ![Status\_Cancelled(status)] = r(i).cancelled \land \neg r(i).transmitted,$ 54.46

$\qquad\qquad\quad ![array\_of\_requests + i] = \text{IF } r(i).persist \text{ THEN } @ \text{ ELSE } MPI\_REQUEST\_NULL,$ 46.1 –

$\qquad\qquad\quad ![index] = i]]$ 45.46

$\qquad\quad \lor \land \lor r(i).message.src = MPI\_PROC\_NULL$ The source or destination was actually

$\qquad\qquad\quad \lor r(i).message.dest = MPI\_PROC\_NULL$ the null process

$\qquad\qquad \land Memory' = [Memory \text{ EXCEPT } ![proc] = [@ \text{ EXCEPT}$

$\qquad\qquad\quad ![Status\_Source(status)] = MPI\_PROC\_NULL,$

$\qquad\qquad\quad ![Status\_Tag(status)] = MPI\_ANY\_TAG,$

$\qquad\qquad\quad ![Status\_Err(status)] = 0,$

$\qquad\qquad\quad ![Status\_Count(status)] = 0,$

$\qquad\qquad\quad ![Status\_Cancelled(status)] = r(i).cancelled,$

$\qquad\qquad\quad ![array\_of\_requests + i] = \text{IF } r(i).persist \text{ THEN } @ \text{ ELSE } MPI\_REQUEST\_NULL,$ 46.2

$\qquad\qquad\quad ![index] = i]]$ 45.46

$\qquad \land requests' = [requests \text{ EXCEPT } ![proc] =$ 46.1, 58.34

$\qquad\qquad\qquad [@ \text{ EXCEPT } ![Memory[proc][array\_of\_requests + i]] =$

$\qquad\qquad\qquad\quad [@ \text{ EXCEPT } !.active = \text{FALSE}]]]$

$\quad\; \lor \forall i \in (0 \,..\, (count - 1)) :$ 46.3 – 46.4

$\qquad \land \lor \neg r(i).active$ The request is not active or the request

$\qquad\quad \lor Memory[proc][array\_of\_requests + i] = MPI\_REQUEST\_NULL$ handle is null

$\qquad \land Memory' = [Memory \text{ EXCEPT } ![proc] =$ 46.5

$[@ \text{ EXCEPT}$
$![Status\_Source(status)] = MPI\_ANY\_SOURCE,$
$![Status\_Tag(status)] = MPI\_ANY\_TAG,$
$![Status\_Err(status)] = 0,$
$![Status\_Count(status)] = 0,$
$![Status\_Cancelled(status)] = \text{FALSE},$
$![index] = MPI\_UNDEFINED]]$ 46.5
$\land \text{ UNCHANGED } \langle requests \rangle$
$\land \text{ UNCHANGED } \langle group, communicator, bufsize, message\_buffer, initialized, collective \rangle$

Test whether one of the requests referenced in *array_of_requests* has completed.

$MPI\_Testany(count, array\_of\_requests, index, flag, status, return, proc) \triangleq$
$\text{LET } r(v) \triangleq requests[proc][Memory[proc][array\_of\_requests + v]] \text{ IN}$
$\land Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Testany called with proc not in initialized state.")
$\land \lor \exists i \in (0 \,..\, (count - 1)):$ 46.28 – 46.29
$\land array\_of\_requests[proc][i] \neq MPI\_REQUEST\_NULL$ The request handle is not the null handle.
$\land r(i).active$ The request is active.
$\land \lor \land r(i).message.src \neq MPI\_PROC\_NULL$ The message source is not null
$\land r(i).message.dest \neq MPI\_PROC\_NULL$ The message destination is not null
$\land \text{ IF } \lor r(i).transmitted$ The communication actually happened or
$\lor r(i).cancelled$ the communication got *cancelled* by the user program or
$\lor r(i).buffered$ the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).
THEN
$\land Memory' = [Memory \text{ EXCEPT } ![proc] = [@ \text{ EXCEPT}$
$![flag] = \text{TRUE},$ 46.29
$![Status\_Source(status)] = r(i).message.src,$ 46.30
$![Status\_Tag(status)] = r(i).message.msgtag,$
$![Status\_Err(status)] = r(i).error,$
$![Status\_Count(status)] = r(i).message.numelements,$
$![Status\_Cancelled(status)] = r(i).cancelled \land \neg r(i).transmitted,$ 54.46
$![index] = i,$ 46.29
$![array\_of\_requests + i] = \text{IF } r(i).persist \text{ THEN } @ \text{ ELSE } MPI\_REQUEST\_NULL]]$ 4(
$\land requests' =$ 46.30 – 46.31, 58.34
$[requests \text{ EXCEPT } ![proc] =$
$[@ \text{ EXCEPT } ![Memory[proc][array\_of\_requests + i]] =$
$[@ \text{ EXCEPT } !.active = \text{FALSE}]]]$
ELSE
$\land Memory' = [Memory \text{ EXCEPT } ![proc] = [@ \text{ EXCEPT}$ status is explicitly undefined.
$![flag] = \text{FALSE},$ 46.33
$![index] = MPI\_UNDEFINED]]$ 46.33 – 46.34
$\land \text{ UNCHANGED } \langle requests \rangle$
$\lor \land \lor r(i).message.src = MPI\_PROC\_NULL$ The source or destination were actually

$\lor\ r(i).message.dest = MPI\_PROC\_NULL$ the null process 61.3 – 61.4
$\land\ Memory' = [Memory$ EXCEPT $![proc] = [@$ EXCEPT
$![flag] =$ TRUE, 46.29
$![Status\_Source(status)] = MPI\_PROC\_NULL,$
$![Status\_Tag(status)] = MPI\_ANY\_TAG,$
$![Status\_Err(status)] = 0,$
$![Status\_Count(status)] = 0,$
$![Status\_Cancelled(status)] = r(i).cancelled,$
$![index] = i,$ 46.29
$![array\_of\_requests + i] =$ IF $r(i).persist$ THEN @ ELSE $MPI\_REQUEST\_NULL]]$ 46.31 –
$\land\ requests' =$ 46.31 – 46.32, 58.34
$[requests$ EXCEPT $![proc] =$
$[@$ EXCEPT $![Memory[proc][array\_of\_requests + i]] =$
$[@$ EXCEPT $!.active =$ FALSE]]]
$\lor\ \forall\, i \in (0\,..\,(count - 1)):$ 46.35 – 46.37
$\land\ \lor\ \neg r(i).active$ The request is not active or the request
$\lor\ array\_of\_requests[proc][i] = MPI\_REQUEST\_NULL$ handle is null
$\land\ Memory' = [Memory$ EXCEPT $![proc] = [@$ EXCEPT
$![flag] =$ TRUE, 46.36
$![Status\_Source(status)] = MPI\_ANY\_SOURCE,$ 46.36
$![Status\_Tag(status)] = MPI\_ANY\_TAG,$
$![Status\_Err(status)] = 0,$
$![Status\_Count(status)] = 0,$
$![Status\_Cancelled(status)] =$ FALSE,
$![index] = MPI\_UNDEFINED]]$ 46.36
$\land$ UNCHANGED $\langle requests \rangle$
$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, initialized, collective \rangle$

A long version of *MPI_Waitall* – includes the line by line reference.
Specifies the next state for *array_of_requests* and *array_of_statuses*.

$MPI\_Waitall(count, array\_of\_requests, array\_of\_statuses, return, proc) \triangleq$
LET $r(v) \triangleq requests[proc][Memory[proc][array\_of\_requests + v]]$ IN
$\land\ Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Waitall called with proc not in initialized state.")
$\land\ \lor\ \forall\, i \in (0\,..\,(count - 1)):$ 47.18
$\land\ Memory[proc][array\_of\_requests + i] \neq MPI\_REQUEST\_NULL$ The request handle is not the null
$\land\ r(i).active$ The request is active.
$\land\ \lor\ \land\ r(i).message.src \neq MPI\_PROC\_NULL$ The message source is not null
$\land\ r(i).message.dest \neq MPI\_PROC\_NULL$ The message destination is not null
$\land\ \lor\ r(i).transmitted$ The communication actually happened or
$\lor\ r(i).cancelled$ the communication got *cancelled* by the user program or
$\lor\ r(i).buffered$ the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).
$\lor\ \land\ \lor\ r(i).message.src = MPI\_PROC\_NULL$ The source or destination was actually

$\lor\ r(i).message.dest = MPI\_PROC\_NULL$ the null process

$\land\ array\_of\_requests' =$
$[array\_of\_requests \text{ EXCEPT } ![proc] =$ 47.22 – 47.23
$[j \in 0 \, .. \, (count - 1) \mapsto$
$\text{IF } r(j).persist$
THEN
$array\_of\_requests[proc][j]$
ELSE
$MPI\_REQUEST\_NULL]]$

$\land\ array\_of\_statuses' = [array\_of\_statuses \text{ EXCEPT } ![proc] =$ 47.18 – 47.21
$[j \in (0 \, .. \, count - 1) \mapsto$
$\text{IF } \lor\ r(j).message.src = MPI\_PROC\_NULL$
$\lor\ r(j).message.dest = MPI\_PROC\_NULL$
THEN 61.3 – 61.4
$[state \mapsto \text{"defined"},$ A status object for a communication
$MPI\_SOURCE \mapsto MPI\_PROC\_NULL,$ with a null process
$MPI\_TAG \mapsto MPI\_ANY\_TAG,$
$MPI\_ERROR \mapsto 0,$
$count \mapsto 0,$
$cancelled \mapsto r(j).cancelled]$
ELSE
$[state \mapsto \text{"defined"},$ A status object for a completed communication.
$MPI\_SOURCE \mapsto r(j).message.src,$
$MPI\_TAG \mapsto r(j).message.msgtag,$
$MPI\_ERROR \mapsto r(j).error,$
$count \mapsto r(j).message.numelements,$
$cancelled \mapsto r(j).cancelled \land \neg r(i).transmitted]]]$ 54.46

$\land\ requests' = [requests \text{ EXCEPT } ![proc] =$ 47.22, 58.34 Not modeling deallocation
$[j \in 1 \, .. \, Len(@) \mapsto$
$\text{IF } \exists\, k \in 0 \, .. \, (count - 1) : j = array\_of\_requests[proc][k]$
THEN
$[requests[proc][j] \text{ EXCEPT } !.active = \text{FALSE}]$
ELSE
$requests[proc][j]]]$

$\lor\ \forall\, i \in 0 \, .. \, (count - 1) :$ 47.23 – 47.24
$\land\ \lor\ Memory[proc][array\_of\_requests + i] = MPI\_REQUEST\_NULL$ The request handle is null or
$\lor\ \neg r(i).active$ not active
$\land\ Memory' = [Memory \text{ EXCEPT } ![proc] =$
$[j \in 1 \, .. \, Len(Memory[proc]) \mapsto$
$\text{IF } j \in array\_of\_statuses \, .. \, (array\_of\_statuses + ((count * 5) - 1))$
THEN
$\text{IF } (j - array\_of\_statuses)\%5 = 0$
THEN FALSE
ELSE
$\text{IF } (j - array\_of\_statuses)\%5 = 1$

```
                  THEN 0
                  ELSE
                   IF (j − array_of_statuses)%5 = 2
                    THEN MPI_ANY_SOURCE
                    ELSE
                     IF (j − array_of_statuses)%5 = 3
                      THEN MPI_ANY_TAG
                      ELSE
                       IF (j − array_of_statuses)%5 = 4
                        THEN 0
                        ELSE Assert(FALSE, "Internal Error: Cannot have any other cases.")
              ELSE Memory[proc][j]]]
          ∧ UNCHANGED ⟨array_of_requests, requests⟩
     ∧ UNCHANGED ⟨group, communicator, bufsize, message_buffer, requests, initialized, collective⟩
```

Test whether all reqeusts referenced in *array_of_requests* have completed.

```
MPI_Testall(count, array_of_requests, flag, array_of_statuses, return, proc) ≜
  LET r(v) ≜ requests[proc][array_of_requests[proc][v]]IN
   ∧ Assert(initialized[proc] = "initialized",    200.10 – 200.12
            "Error: MPI_Testall called with proc not in initialized state.")
   ∧ IF ∨ ∀ i ∈ (0 . . (count − 1)) :    48.15
            ∨ ∧ array_of_requests[proc][i] ≠ MPI_REQUEST_NULL    The request handle is not the null handle.
              ∧ r(i).active                                      The request is active.
              ∧ ∨ ∧ r(i).message.src ≠ MPI_PROC_NULL            The message source is not null
                  ∧ r(i).message.dest ≠ MPI_PROC_NULL           The message destination is not null
                  ∧ ∨ r(i).transmitted   The communication actually happened or
                    ∨ r(i).cancelled     the communication got cancelled by the user program or
                    ∨ r(i).buffered      the communication got buffered either into explicit user provided
                                         buffer space or into system provided buffer space (if regular send is used).
            ∨ ∧ ∨ r(i).message.src = MPI_PROC_NULL      The source or destination were actually
                ∨ r(i).message.dest = MPI_PROC_NULL     the null process
        ∨ ∀ i ∈ (0 . . (count − 1)) :   48.16
            ∨ array_of_requests[proc][i] = MPI_REQUEST_NULL
            ∨ ¬r(i).active
      THEN
       ∧ array_of_statuses′ = [array_of_statuses EXCEPT ![proc] =
           [i ∈ (0 . . (count − 1)) ↦
            IF ∨ r(i).message.src = MPI_PROC_NULL
               ∨ r(i).message.dest = MPI_PROC_NULL
             THEN                                          61.3 – 61.4
              [state        ↦ "defined",                   A status object for a communication
               MPI_SOURCE ↦ MPI_PROC_NULL,               with a null process
               MPI_TAG     ↦ MPI_ANY_TAG,
               MPI_ERROR   ↦ 0,
```

$$
\begin{array}{l}
\quad\quad\quad\quad \mathit{count} \quad\quad\quad \mapsto 0, \\
\quad\quad\quad\quad \mathit{cancelled} \quad\quad \mapsto \text{FALSE}] \\
\quad\quad\quad \text{ELSE} \\
\quad\quad\quad \text{IF } \lor \mathit{array\_of\_requests}[\mathit{proc}][i] = \mathit{MPI\_REQUEST\_NULL} \quad \text{48.21} \\
\quad\quad\quad\quad\quad \lor \neg r(i).\mathit{active} \\
\quad\quad\quad\quad \text{THEN} \\
\quad\quad\quad\quad [\mathit{state} \mapsto \text{“empty”}, \quad \text{The resultant empty status.} \\
\quad\quad\quad\quad\ \mathit{MPI\_SOURCE} \mapsto \mathit{MPI\_ANY\_SOURCE}, \\
\quad\quad\quad\quad\ \mathit{MPI\_TAG} \mapsto \mathit{MPI\_ANY\_TAG}, \\
\quad\quad\quad\quad\ \mathit{MPI\_ERROR} \mapsto 0, \\
\quad\quad\quad\quad\ \mathit{count} \mapsto 0, \\
\quad\quad\quad\quad\ \mathit{cancelled} \mapsto \text{FALSE}] \\
\quad\quad\quad\quad \text{ELSE} \quad \text{48.17 – 48.18} \\
\quad\quad\quad\quad [\mathit{state} \mapsto \text{“defined”}, \quad \text{A status object for a completed communication.} \\
\quad\quad\quad\quad\ \mathit{MPI\_SOURCE} \mapsto r(i).\mathit{message}.\mathit{src}, \\
\quad\quad\quad\quad\ \mathit{MPI\_TAG} \mapsto r(i).\mathit{message}.\mathit{msgtag}, \\
\quad\quad\quad\quad\ \mathit{MPI\_ERROR} \mapsto r(i).\mathit{error}, \\
\quad\quad\quad\quad\ \mathit{count} \mapsto r(i).\mathit{message}.\mathit{numelements}, \\
\quad\quad\quad\quad\ \mathit{cancelled} \mapsto r(i).\mathit{cancelled} \land \neg r(i).\mathit{transmitted}]]] \quad \text{54.46} \\
\quad \land \mathit{requests}' = \quad \text{48.18 – 48.19, 58.34 Not modeling deallocation} \\
\quad\quad [\mathit{requests} \text{ EXCEPT } ![\mathit{proc}] = \\
\quad\quad\quad [i \in 1 \ldots \mathit{Len}(@) \mapsto \\
\quad\quad\quad\quad \text{IF } \exists j \in 0 \ldots (\mathit{count} - 1) : \mathit{array\_of\_requests}[\mathit{proc}][j] = i \\
\quad\quad\quad\quad\ \text{THEN} \\
\quad\quad\quad\quad\quad [\mathit{requests}[\mathit{proc}][i] \text{ EXCEPT } !.\mathit{active} = \text{FALSE}] \\
\quad\quad\quad\quad\ \text{ELSE} \\
\quad\quad\quad\quad\quad \mathit{requests}[\mathit{proc}][i]]] \\
\quad \land \mathit{array\_of\_requests}' = [\mathit{array\_of\_requests} \text{ EXCEPT } ![\mathit{proc}] = \\
\quad\quad [i \in 0 \ldots (\mathit{count} - 1) \mapsto \\
\quad\quad\quad \text{IF } r(i).\mathit{persist} \\
\quad\quad\quad\ \text{THEN} \\
\quad\quad\quad\quad \mathit{array\_of\_requests}[\mathit{proc}][i] \quad \text{58.34 – 58.35} \\
\quad\quad\quad\ \text{ELSE} \\
\quad\quad\quad\quad \mathit{MPI\_REQUEST\_NULL}]] \quad \text{48.19 – 48.21} \\
\quad \land \mathit{flag}' = [\mathit{flag} \text{ EXCEPT } ![\mathit{proc}] = \text{TRUE}] \quad \text{48.15} \\
\text{ELSE} \\
\quad \land \mathit{flag}' = [\mathit{flag} \text{ EXCEPT } ![\mathit{proc}] = \text{FALSE}] \quad \text{48.21 – 48.22} \\
\quad \land \mathit{array\_of\_statuses}' = [\mathit{array\_of\_statuses} \text{ EXCEPT } ![\mathit{proc}] = \\
\quad\quad [i \in 0 \ldots (\mathit{count} - 1) \mapsto \\
\quad\quad\quad [\mathit{array\_of\_statuses}[\mathit{proc}][i] \text{ EXCEPT } !.\mathit{state} = \text{“undefined”}]]] \\
\quad \land \text{UNCHANGED } \langle \mathit{array\_of\_requests}, \mathit{requests} \rangle \\
\land \text{UNCHANGED } \langle \mathit{group}, \mathit{communicator}, \mathit{bufsize}, \mathit{message\_buffer}, \mathit{initialized}, \mathit{collective} \rangle
\end{array}
$$

Wait for some subset of the requests referenced in *array_of_requests* to complete.
The ordering of *array_of_indices* or *array_of_statuses* is not specified.
Not modeling the possibility of arbitrary ordering of the *array_of_indices* or *array_of_statuses*.

$MPI\_Waitsome(incount,\ array\_of\_requests,\ outcount,$
$\quad array\_of\_indices,\ array\_of\_statuses,\ return,\ proc) \triangleq$
$\text{LET } r(v) \triangleq requests[proc][array\_of\_requests[proc][v]] \text{ IN}$
$\text{LET } msgs \triangleq$
$\{x \in (0 \,..\, (incount - 1)):$ The messages that have completed in the *array_of_requests*
$\land\ array\_of\_requests[proc][x] \neq MPI\_REQUEST\_NULL$ The request handle is not the null handle.
$\land\ r(x).active$ The request is active.
$\land\ \lor\ r(x).transmitted$ The communication actually happened or
$\lor\ r(x).cancelled$ the communication got *cancelled* by the user program or
$\lor\ r(x).buffered\}$ the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).
IN
$\land\ Assert(initialized[proc] = \text{"initialized"},$ 200.10 – 200.12
$\quad \text{"Error: MPI\_Waitsome called with proc not in initialized state."})$
$\land\ \lor\ \land\ Cardinality(msgs) > 0$ 48.45
$\land\ outcount' = [outcount \text{ EXCEPT } ![proc] = Cardinality(msgs)]$ 48.46
$\land\ \exists\, seq \in Seq(msgs):$ from *FiniteSets.tla* module!
$\land\ \forall\, s \in msgs:$
$\exists\, n \in 1 \,..\, Len(seq):$
$\land\ seq[n] = s$
$\land\ \forall\, m \in 1 \,..\, Len(seq): seq[n] = seq[m] \Rightarrow m = n$
$\land\ array\_of\_indices' = [array\_of\_indices \text{ EXCEPT } ![proc] =$
$[i \in 0 \,..\, (incount - 1) \mapsto$
$\text{IF } i < Len(seq)$
$\text{THEN } seq[i + 1]$
$\text{ELSE } array\_of\_indices[proc][i]]]$
$\land\ array\_of\_statuses' = [array\_of\_statuses \text{ EXCEPT } ![proc] =$
$[i \in 0 \,..\, (incount - 1) \mapsto$
$\text{IF } i < Len(seq)$
THEN
$[state \mapsto \text{"defined"},$ A status object for a completed communication.
$MPI\_SOURCE \mapsto r(seq[i + 1]).message.src,$
$MPI\_TAG \mapsto r(seq[i + 1]).message.msgtag,$
$MPI\_ERROR \mapsto r(seq[i + 1]).error,$
$count \mapsto r(seq[i + 1]).message.numelements,$
$cancelled \mapsto r(seq[i + 1]).cancelled \land \neg r(seq[i + 1]).transmitted]$ 54.46
ELSE
$array\_of\_statuses[proc][i]]]$
$\land\ requests' = [requests \text{ EXCEPT } ![proc] =$
$[i \in 1 \,..\, Len(requests[proc]) \mapsto$
$\text{IF } \exists\, m \in msgs: i = array\_of\_requests[proc][m]$

$\text{THEN } [r(i) \text{ EXCEPT } !.active = \text{FALSE}]$
$\text{ELSE } \; r(i)]]$
$\land\; array\_of\_requests' = [array\_of\_requests \text{ EXCEPT } ![proc] =$
$[i \in 0 \,..\, (incount - 1) \mapsto$ 49.2 – 49.4
$\text{IF } \land \exists\, m \in msgs : i = array\_of\_requests[proc][m]$
$\land\; r(i).persist$
THEN
$array\_of\_requests[proc][i]$
ELSE
$MPI\_REQUEST\_NULL]]$
$\lor\; \land \forall\, i \in (0 \,..\, (incount - 1)) :$ 49.5
$\lor\; array\_of\_requests[proc][i] = MPI\_REQUEST\_NULL$
$\lor\; \neg requests[proc][array\_of\_requests[proc][i]].active$
$\land\; outcount' = [outcount \text{ EXCEPT } ![proc] = MPI\_UNDEFINED]$ 49.5 – 49.6
$\land\; \text{UNCHANGED } \langle array\_of\_indices, array\_of\_statuses, requests, array\_of\_requests \rangle$
$\land\; \text{UNCHANGED } \langle group, communicator, bufsize, message\_buffer, initialized, collective \rangle$

Test for some subset of the requests referenced in the *array_of_requests* to complete.
Defined in terms of *MPI_Waitsome*.

$MPI\_Testsome(incount, array\_of\_requests, outcount,$
$array\_of\_indices, array\_of\_statuses, return, proc) \triangleq$
$\text{LET } r(v) \triangleq requests[proc][array\_of\_requests[proc][v]] \text{IN}$
$\text{LET } msgs \triangleq$
$\{x \in (0 \,..\, (incount - 1)) :$ The messages that have completed in the *array_of_requests*
$\land\; array\_of\_requests[proc][x] \neq MPI\_REQUEST\_NULL$ The request handle is not the null handle.
$\land\; r(x).active$ The request is active.
$\land\; \lor\; r(x).transmitted$ The communication actually happened or
$\lor\; r(x).cancelled$ the communication got *cancelled* by the user program or
$\lor\; r(x).buffered\}$ the communication got buffered either into explicit user provided buffer space or into system provided buffer space (if regular send is used).
IN
$\land\; Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Testsome called with proc not in initialized state.")
$\land\; \lor\; \exists\, i \in (0 \,..\, (incount - 1)) :$ 49.35 – 49.36, 49.5
$\land\; array\_of\_requests[proc][i] \neq MPI\_REQUEST\_NULL$
$\land\; r(i).active$
$\land\; \text{IF } Cardinality(msgs) > 0$ number of completed messages
THEN
$\land\; outcount' = [outcount \text{ EXCEPT } ![proc] = Cardinality(msgs)]$ 48.46
$\land\; \exists\, seq \in Seq(msgs) :$ from *FiniteSets.tla* module!
$\land\; \forall\, s \in msgs :$
$\exists\, n \in 1 \,..\, Len(seq) :$
$\land\; seq[n] = s$ 48.47 – 49.2
$\land\; \forall\, m \in 1 \,..\, Len(seq) : seq[n] = seq[m] \Rightarrow m = n$

$\land$ *array_of_indices'* = [*array_of_indices* EXCEPT ![*proc*] =
  [$j \in 0 \, .. \, (incount - 1) \mapsto$
  IF $j < Len(seq)$
   THEN $seq[j+1]$
   ELSE *array_of_indices*[*proc*][j]]]
$\land$ *array_of_statuses'* = [*array_of_statuses* EXCEPT ![*proc*] =
  [$j \in 0 \, .. \, (incount - 1) \mapsto$
  IF $j < Len(seq)$
   THEN
    [*state* $\mapsto$ "defined", A status object for a completed communication.
    MPI_SOURCE $\mapsto$ $r(seq[j+1])$.*message.src*,
    MPI_TAG $\mapsto$ $r(seq[j+1])$.*message.msgtag*,
    MPI_ERROR $\mapsto$ $r(seq[j+1])$.*error*,
    count $\mapsto$ $r(seq[j+1])$.*message.numelements*,
    cancelled $\mapsto$ $r(seq[j+1])$.*cancelled* $\land \neg r(seq[j+1])$.*transmitted*] 54.46
   ELSE
    array_of_statuses[*proc*][j]]]
$\land$ *requests'* = [*requests* EXCEPT ![*proc*] =
  [$j \in 1 \, .. \, Len(requests[proc]) \mapsto$
  IF $\exists \, m \in msgs : j = array\_of\_requests[proc][m]$
   THEN [$r(j)$ EXCEPT !.*active* = FALSE] 58.34
   ELSE $r(j)$]]
$\land$ *array_of_requests'* = [*array_of_requests* EXCEPT ![*proc*] =
  [$j \in 0 \, .. \, (incount - 1) \mapsto$ 49.2 – 49.4
  IF $\land \exists \, m \in msgs : j = array\_of\_requests[proc][m]$
    $\land$ $r(j)$.*persist* 49.2 – 49.4
   THEN
    array_of_requests[*proc*][j] 58.34 – 58.35
   ELSE
    MPI_REQUEST_NULL]]
ELSE 49.35
 $\land$ *outcount'* = [*outcount* EXCEPT ![*proc*] = 0]
 $\land$ UNCHANGED $\langle$*array_of_indices*, *array_of_statuses*, *requests*, *array_of_requests*$\rangle$
$\lor$ $\land \forall \, i \in (0 \, .. \, (incount - 1))$: 49.5
  $\lor$ *array_of_requests*[*proc*][i] = *MPI_REQUEST_NULL*
  $\lor$ $\neg$*requests*[*proc*][*array_of_requests*[*proc*][i]].*active*
 $\land$ *outcount'* = [*outcount* EXCEPT ![*proc*] = *MPI_UNDEFINED*] 49.5 – 49.6, 49.36
$\land$ UNCHANGED $\langle$*group*, *communicator*, *bufsize*, *message_buffer*, *requests*, *initialized*, *collective*$\rangle$

Section 3.8 Probe and Cancel

What happens in the following scenerio: 1: send 2: probe 1: cancel 2: *recv*

Probe for a message. Nonblocking; note the leading IF

$MPI\_Iprobe(source, tag, comm, flag, status, return, proc) \triangleq$
$\land Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Testany called with proc not in initialized state.")
$\land$ IF $\exists i \in (0..(N-1))$: 51.39 – 51.41
$\exists j \in (1..Len(requests[i]))$:
LET $m \triangleq requests[i][j].message$ IN
$\land \lor m.src = source$
$\lor source = MPI\_ANY\_SOURCE$
$\land \lor m.msgtag = tag$
$\lor tag = MPI\_ANY\_TAG$
$\land m.universe = comm$ unique across space/time – not required by standard
$\land m.state =$ "send" 51.41 – 51.42 must match
$\land requests[i][j].active$ 51.41 – 51.42
$\land \neg requests[i][j].transmitted$
$\land \neg requests[i][j].cancelled$
THEN
$\exists i \in (0..(N-1))$: 51.39 – 51.41
$\exists j \in (1..Len(requests[i]))$:
LET $m \triangleq requests[i][j].message$ IN
$\land \lor m.src = source$
$\lor source = MPI\_ANY\_SOURCE$
$\land \lor m.msgtag = tag$
$\lor tag = MPI\_ANY\_TAG$
$\land m.universe = comm$ unique across space/time – not required by standard
$\land m.state =$ "send" 51.41 – 51.42 must match
$\land requests[i][j].active$ 51.41 – 51.42
$\land \neg requests[i][j].transmitted$
$\land \neg requests[i][j].cancelled$
$\land \forall k \in (1..Len(requests[i]))$: least match
$\land requests[i][k].active$
$\land \neg requests[i][k].cancelled$
$\land \neg requests[i][k].transmitted$
$\Rightarrow j \leq k$
$\land Memory' = [Memory$ EXCEPT $![proc] =$
$[[loc \in 1..Len(Memory[proc]) \mapsto$ 51.42
IF $loc = Status\_Cancelled(status)$
THEN FALSE
ELSE
IF $loc = Status\_Count(status)$
THEN $m.numelements$
ELSE
IF $loc = Status\_Source(status)$
THEN $m.src$
ELSE

$\text{IF } loc = Status\_Tag(status)$
$\text{THEN } m.msgtag$
ELSE
$\text{IF } loc = Status\_Err(status)$
$\text{THEN } requests[i][j].error$
$\text{ELSE } Memory[proc][loc]]$
$\text{EXCEPT } ![flag] = \text{TRUE}]]$ 51.39

ELSE

$\land Memory' = [Memory \text{ EXCEPT } ![proc] = [@ \text{ EXCEPT } ![flag] = \text{FALSE}]]$ 51.44 Status is undefined

$\land \text{UNCHANGED } \langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$

Wait on a probe for a message. 52.24 – 52.25

$MPI\_Probe(source, tag, comm, status, return, proc) \triangleq$

$\land Assert(initialized[proc] = \text{"initialized"},$ 200.10 – 200.12
"Error: MPI_Testany called with proc not in initialized state.")

$\land \exists i \in (0 \,..\, (N-1)):$
$\exists j \in (1 \,..\, Len(requests[i])):$
$\text{LET } m \triangleq requests[i][j].message\text{IN}$
$\land \lor m.src = source$
$\lor source = MPI\_ANY\_SOURCE$
$\land \lor m.msgtag = tag$
$\lor tag = MPI\_ANY\_TAG$
$\land m.universe = comm$ unique across space/time – not required by standard
$\land m.state = \text{"send"}$
$\land requests[i][j].active$
$\land \neg requests[i][j].transmitted$
$\land \neg requests[i][j].cancelled$
$\land \forall k \in (1 \,..\, Len(requests[i])):$
$\land requests[i][k].active$
$\land \neg requests[i][k].cancelled$
$\land \neg requests[i][k].transmitted$
$\Rightarrow j \leq k$
$\land Memory' = [Memory \text{ EXCEPT } ![proc] =$
$[loc \in 1 \,..\, Len(Memory[proc]) \mapsto$ 51.42
$\text{IF } loc = Status\_Cancelled(status)$
$\text{THEN } requests[i][j].cancelled \land \neg requests[i][j].transmitted$
ELSE
$\text{IF } loc = Status\_Count(status)$
$\text{THEN } m.numelements$
ELSE
$\text{IF } loc = Status\_Source(status)$
$\text{THEN } m.src$
ELSE
$\text{IF } loc = Status\_Tag(status)$

$\quad\quad\quad\quad$ THEN $m.msgtag$
ELSE
IF $loc = Status\_Err(status)$
THEN $requests[i][j].error$
ELSE $Memory[proc][loc]]]$
$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$

Cancel an active request.
What do you do when the request is *MPI_REQUEST_NULL*?

$MPI\_Cancel(request, return, proc) \triangleq$ 54.8 – 54.10
$\land$ $requests' = [requests$ EXCEPT $![proc] =$
$[@$ EXCEPT $![Memory[proc][request]] =$
$[@$ EXCEPT $!.cancelled =$ TRUE$]]]$
$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$
$\land$ UNCHANGED $\langle Memory\rangle$

Test whether a request was *cancelled* successfully.

$MPI\_Test\_cancelled(status, flag, return, proc) \triangleq$ 54.46 – 55.1
$\land$ $Memory' = [Memory$ EXCEPT $![proc] = [@$ EXCEPT $![flag] = Memory[proc][Status\_Cancelled(status)]]]$
$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$

Section 3.9 Persistent communication requests

Create a persistant standard mode send request.

$MPI\_Send\_init(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq$
$\land$ $Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Send_init called with proc not in initialized state.")
$\land$ $requests' = [requests$ EXCEPT $![proc] =$ 56.4 – 56.5
LET $msg \triangleq [addr \mapsto buf,$
$src \mapsto group[proc][communicator[proc][comm].group].ranking[proc],$
$dest \mapsto dest,$
$msgtag \mapsto tag,$
$dtype \mapsto datatype,$
$numelements \mapsto count,$
$universe \mapsto comm,$
$state \mapsto$ "send"$]$
IN
$@ \circ \langle Make\_request(0,$ FALSE, FALSE, FALSE, FALSE, "send", TRUE, $\langle\rangle$, FALSE, $msg)\rangle]$
57.42 – 57.46
$\land$ $Memory' = [Memory$ EXCEPT $![proc] = [@$ EXCEPT $![request] = Len(requests[proc]) + 1]]$
$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, initialized, collective\rangle$

Create a persistant buffered mode send request.

$MPI\_Bsend\_init(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq$

$\land Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Bsend_init called with proc not in initialized state.")

$\land requests' = [requests$ EXCEPT $![proc] =$ 56.26

LET $msg \triangleq [addr \mapsto buf,$
$src \mapsto group[proc][communicator[proc][comm].group].ranking[proc],$
$dest \mapsto dest,$
$msgtag \mapsto tag,$
$dtype \mapsto datatype,$
$numelements \mapsto count,$
$universe \mapsto comm,$
$state \mapsto$ "send"]

IN

$@ \circ \langle Make\_request(0,$ FALSE, FALSE, FALSE, FALSE, "bsend", TRUE, $\langle\rangle$, FALSE, $msg)\rangle]$
57.42 – 57.46

$\land Memory' = [Memory$ EXCEPT $![proc] = [@$ EXCEPT $![request] = Len(requests[proc]) + 1]]$

$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, initialized, collective\rangle$

Create a persistant synchronous mode send request.

$MPI\_Ssend\_init(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq$

$\land Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Ssend_init called with proc not in initialized state.")

$\land requests' = [requests$ EXCEPT $![proc] =$ 56.46

LET $msg \triangleq [addr \mapsto buf,$
$src \mapsto group[proc][communicator[proc][comm].group].ranking[proc],$
$dest \mapsto dest,$
$msgtag \mapsto tag,$
$dtype \mapsto datatype,$
$numelements \mapsto count,$
$universe \mapsto comm,$
$state \mapsto$ "send"]

IN

$@ \circ \langle Make\_request(0,$ FALSE, FALSE, FALSE, FALSE, "ssend", TRUE, $\langle\rangle$, FALSE, $msg)\rangle]$
57.42 – 57.46

$\land Memory' = [Memory$ EXCEPT $![proc] = [@$ EXCEPT $![request] = Len(requests[proc]) + 1]]$

$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, initialized, collective\rangle$

Create a persistant ready mode send request.

$MPI\_Rsend\_init(buf, count, datatype, dest, tag, comm, request, return, proc) \triangleq$

$\land Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Rsend_init called with proc not in initialized state.")

$\land requests' = [requests$ EXCEPT $![proc] =$ 57.18

LET $msg \triangleq [addr \mapsto buf,$
$src \mapsto group[proc][communicator[proc][comm].group].ranking[proc],$

$$
\begin{array}{l}
\qquad\qquad\qquad dest \mapsto dest, \\
\qquad\qquad\qquad msgtag \mapsto tag, \\
\qquad\qquad\qquad dtype \mapsto datatype, \\
\qquad\qquad\qquad numelements \mapsto count, \\
\qquad\qquad\qquad universe \mapsto comm, \\
\qquad\qquad\qquad state \mapsto \text{“send”}] \\
\quad \text{IN} \\
\qquad @ \circ \langle Make\_request(0, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{“rsend”}, \text{TRUE}, \langle\rangle, \text{FALSE}, msg)\rangle] \quad \fbox{57.42 – 57.46} \\
\land Memory' = [Memory \text{ EXCEPT } ![proc] = [@ \text{ EXCEPT } ![request] = Len(requests[proc]) + 1]] \\
\land \text{UNCHANGED } \langle group, communicator, bufsize, message\_buffer, initialized, collective\rangle
\end{array}
$$

Create a persistant receive request.

$$
\begin{array}{l}
MPI\_Recv\_init(buf, count, datatype, source, tag, comm, request, return, proc) \triangleq \\
\land Assert(initialized[proc] = \text{“initialized”}, \quad \fbox{200.10 – 200.12} \\
\qquad\qquad \text{“Error: MPI\_Recv\_init called with proc not in initialized state.”}) \\
\land requests' = [requests \text{ EXCEPT } ![proc] = \quad \fbox{57.39} \\
\quad \text{LET } msg \triangleq [addr \mapsto buf, \\
\qquad\qquad\qquad src \mapsto source, \\
\qquad\qquad\qquad dest \mapsto group[proc][communicator[proc][comm].group].ranking[proc], \\
\qquad\qquad\qquad msgtag \mapsto tag, \\
\qquad\qquad\qquad dtype \mapsto datatype, \\
\qquad\qquad\qquad numelements \mapsto count, \\
\qquad\qquad\qquad universe \mapsto comm, \\
\qquad\qquad\qquad state \mapsto \text{“recv”}] \\
\quad \text{IN} \\
\qquad @ \circ \langle Make\_request(0, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{FALSE}, \text{“recv”}, \text{TRUE}, \langle\rangle, \text{FALSE}, msg)\rangle] \quad \fbox{57.42 – 57.46} \\
\land Memory' = [Memory \text{ EXCEPT } ![proc] = [@ \text{ EXCEPT } ![request] = Len(requests[proc]) + 1]] \\
\land \text{UNCHANGED } \langle group, communicator, bufsize, message\_buffer, initialized, collective\rangle
\end{array}
$$

Start a persistant communication.

What happens when a ready mode send is started and then the receive is *cancelled* before the communication has a chance to transmit?

$$
\begin{array}{l}
MPI\_Start(request, return, proc) \triangleq \\
\land Assert(initialized[proc] = \text{“initialized”}, \quad \fbox{200.10 – 200.12} \\
\qquad\qquad \text{“Error: MPI\_Start called with proc not in initialized state.”}) \\
\land Assert(\neg requests[proc][Memory[proc][request]].active, \quad \fbox{58.9} \\
\qquad\qquad \text{“Error: MPI\_Start tried to start a request that is already active.”}) \\
\land Assert(Memory[proc][request] \neq MPI\_REQUEST\_NULL, \\
\qquad\qquad \text{“Error: MPI\_Start tried to start a request that is null.”}) \\
\land Assert(requests[proc][Memory[proc][request]].ctype = \text{“rsend”} \Rightarrow \quad \fbox{58.10 – 58.11} \\
\qquad\qquad \exists\, j \in (0 \mathrel{..} (N-1)) : \\
\qquad\qquad\quad \exists\, k \in (1 \mathrel{..} Len(requests[j])) :
\end{array}
$$

$\land$ $requests[j][k].active$
$\land$ $\neg requests[j][k].transmitted$
$\land$ $\neg requests[j][k].cancelled$
$\land$ $Match(requests[proc][Memory[proc][request]].message,\ requests[j][k].message)$,
"Error: MPI_Start tried to start a rsend request when no matching message exists.")
$\land$ $Assert(requests[proc][Memory[proc][request]].ctype =$ "bsend" $\Rightarrow$
$message\_buffer[proc] < bufsize[proc]$,
"Error: MPI_Start tried to start a bsend request when insufficient buffering was available.")
$\land$ $Assert(requests[proc][Memory[proc][request]].persist$, 57.44 – 57.45, 58.8
"Error: MPI_Start tried to start a non-persistant request.")
$\land$ $requests' = [requests$ EXCEPT $![proc] =$
[@ EXCEPT $![Memory[proc][request]] =$
[@ EXCEPT
$!.active =$ TRUE, 58.9
$!.started =$ TRUE,
$!.transmitted =$ FALSE,
$!.cancelled =$ FALSE]]]
$\land$ IF $requests[proc][Memory[proc][request]].ctype =$ "bsend"
THEN
$message\_buffer' = [message\_buffer$ EXCEPT $![proc] = @ + 1]$
ELSE
UNCHANGED $\langle message\_buffer \rangle$
$\land$ UNCHANGED $\langle group, communicator, bufsize, initialized, collective \rangle$
$\land$ UNCHANGED $\langle Memory \rangle$

Start a list of persistant communications.
Can you start many rsends with only one matching receive posted? –maybe yes
Can you start many bsends with only enough buffering for a subset of the sends? –maybe no

$MPI\_Startall(count, array\_of\_requests, return, proc) \triangleq$
LET $m \triangleq \{x \in (0 \,..\, (count - 1)) : requests[proc][Memory[proc][array\_of\_requests + x]].ctype =$ "bsend"$\}$I
$\land$ $Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Startall called with proc not in initialized state.")
$\land$ $Assert(\forall\, i \in (0 \,..\, (count - 1)) : \neg requests[proc][array\_of\_requests[i]].active$,
"Error: MPI_Startall called with some request already active.")
$\land$ $Assert(\forall\, i \in (0 \,..\, (count - 1)) : array\_of\_requests[i] \neq MPI\_REQUEST\_NULL$,
"Error: MPI_Startall called with some request null.")
$\land$ $Assert(\forall\, i \in (0 \,..\, (count - 1)) :$
$requests[proc][array\_of\_requests[i]].ctype =$ "rsend" $\Rightarrow$ 58.10 – 58.11
$\exists\, j \in (0 \,..\, (N - 1)) :$
$\exists\, k \in (1 \,..\, Len(requests[j])) :$
$\land$ $requests[j][k].active$
$\land$ $\neg requests[j][k].transmitted$
$\land$ $\neg requests[j][k].cancelled$
$\land$ $Match(requests[proc][array\_of\_requests[i]].message,\ requests[j][k].message)$,

"Error: MPI_Start tried to start a rsend request when no matching message exists.")
$\land$ $Assert(\forall\, i \in (0 \,..\, (count - 1)):$
$requests[proc][array\_of\_requests[i]].ctype =$ "bsend" $\Rightarrow$
$message\_buffer[proc] + Cardinality(m) < bufsize[proc],$
"Error: MPI_Start tried to start a bsend request when insufficient buffering was available.")
$\land$ $Assert(\forall\, i \in (0 \,..\, (count - 1)):$
$requests[proc][array\_of\_requests[i]].persist,$ 57.44 – 57.45, 58.8
"Error: MPI_Start tried to start a non-persistant request.")
$\land\ requests' = [requests$ EXCEPT $![proc] =$
$[i \in (1 \,..\, Len(requests[proc])) \mapsto$
IF $\exists\, j \in (0 \,..\, (count - 1)) : array\_of\_requests[j] = i$
THEN 58.9
$[requests[proc][i]$ EXCEPT
$!.active =$ TRUE,
$!.started =$ TRUE,
$!.transmitted =$ FALSE,
$!.cancelled =$ FALSE]
ELSE
$requests[proc][i]]]$
$\land\ message\_buffer' = [message\_buffer$ EXCEPT $![proc] = @ + Cardinality(m)]$
$\land$ UNCHANGED $\langle group, communicator, bufsize, initialized, collective\rangle$
$\land$ UNCHANGED $\langle Memory\rangle$

Section 3.10 Send-receive

Can this be done with only one transition? I don't think so.

$MPI\_Sendrecv(sendbuf, sendcount, sendtype, dest,$
$sendtag, recvbuf, recvcount, recvtype,$
$source, recvtag, comm, status) \triangleq$

Section 4.3 Barrier

$MPI\_Barrier\_init(comm, return, proc) \triangleq$
$\land\ \lor\ \land\ collective[communicator[proc][comm].collective].state =$ "vacant"
$\land\ collective' = [collective$ EXCEPT $![communicator[proc][comm].collective] =$
$[@$ EXCEPT
$!.participants = @ \cup \{proc\},$
$!.type =$ "barrier",
$!.state =$ "in"]]
$\lor\ \land\ collective[communicator[proc][comm].collective].state =$ "in"
$\land\ proc \notin collective[communicator[proc][comm].collective].participants$
$\land\ collective' = [collective$ EXCEPT $![communicator[proc][comm].collective] =$
$[@$ EXCEPT $!.participants = @ \cup \{proc\}]]$
$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, requests, initialized\rangle$

∧ UNCHANGED ⟨*Memory*⟩

MPI_Barrier_wait(*comm*, *return*, *proc*) ≜
∧ ∨ ∧ *collective*[*communicator*[*proc*][*comm*].*collective*].*participants* = *group*[*proc*][*communicator*[*proc*][*c*
∧ *proc* ∈ *collective*[*communicator*[*proc*][*comm*].*collective*].*participants*
∧ *collective*[*communicator*[*proc*][*comm*].*collective*].*state* = "in"
∧ *collective*′ = [*collective* EXCEPT ![*communicator*[*proc*][*comm*].*collective*] =
[@ EXCEPT
!.*participants* = @ \ {*proc*},
!.*state* = "out"]]
∨ ∧ *proc* ∈ *collective*[*communicator*[*proc*][*comm*].*collective*].*participants*
∧ *collective*[*communicator*[*proc*][*comm*].*collective*].*state* = "out"
∧ IF *collective*[*communicator*[*proc*][*comm*].*collective*].*participants* = {*proc*}
THEN
collective′ = [*collective* EXCEPT ![*communicator*[*proc*][*comm*].*collective*] =
[@ EXCEPT
!.*participants* = {},
!.*state* = "vacant"]]
ELSE
collective′ = [*collective* EXCEPT ![*communicator*[*proc*][*comm*].*collective*] =
[@ EXCEPT !.*participants* = @ \ {*proc*}]]
∧ UNCHANGED ⟨*group*, *communicator*, *bufsize*, *message_buffer*, *requests*, *initialized*⟩
∧ UNCHANGED ⟨*Memory*⟩

Section 5.3.1 *Group* Accessors

No text description.
MPI_Group_size(*gr*, *size*, *return*, *proc*) ≜
∧ *Assert*(*initialized*[*proc*] = "initialized", 200.10 – 200.12
"Error: MPI_Group_size called with proc not in initialized state.")
∧ *Memory*′ = [*Memory* EXCEPT ![*proc*] = [@ EXCEPT ![*size*] = *group*[*proc*][*gr*].*size*]]
∧ UNCHANGED ⟨*group*, *communicator*, *bufsize*, *message_buffer*, *requests*, *initialized*, *collective*⟩

No text description.
MPI_Group_rank(*gr*, *rank*, *return*, *proc*) ≜
∧ *Assert*(*initialized*[*proc*] = "initialized", 200.10 – 200.12
"Error: MPI_Group_rank called with proc not in initialized state.")
∧ *Memory*′ = [*Memory* EXCEPT ![*proc*] =
[@ EXCEPT ![*rank*] =
IF *proc* ∈ *group*[*proc*][*gr*].*members* THEN *group*.*ranking*[*proc*] ELSE *MPI_UNDEFINED*]]
∧ UNCHANGED ⟨*group*, *communicator*, *bufsize*, *message_buffer*, *requests*, *initialized*, *collective*⟩

MPI_Group_translate_ranks(*group*1, *n*, *ranks*1, *group*2, *ranks*2, *return*, *proc*) ≜

$\land$ *Assert*(*initialized*[*proc*] = "initialized", 200.10 – 200.12
"Error: MPI_Group_translate_ranks called before MPI_Init.")
$\land$ *Assert*(*group*1 $\in$ *MPI_COMM_WORLD* . . (*MPI_COMM_WORLD* + *MAX_GROUP*),
"Error: MPI_Group_translate_ranks called with invalid handle for group1.")
$\land$ *Assert*(*group*2 $\in$ *MPI_COMM_WORLD* . . (*MPI_COMM_WORLD* + *MAX_GROUP*),
"Error: MPI_Group_translate_ranks called with invalid handle for group2.")
$\land$ *Assert*(*n* = *Cardinality*(DOMAIN *ranks*1), 138.3
"Error: MPI_Group_translate_ranks called with invalid n.")
$\land$ *Memory*′ = [*Memory* EXCEPT ![*proc*] =
[*i* $\in$ 1 . . *Len*(*Memory*[*proc*]) $\mapsto$
IF *i* $\in$ *ranks*2 . . (*ranks*2 + *n*)
THEN *group*[*proc*][*group*2].*ranking*[*group*[*proc*][*group*1].*invranking*[*ranks*1[*i*]]]
ELSE *Memory*[*proc*][*i*]]] not quite right as there is no possibility of *MPI_UNDEFINED* being assigned.
$\land$ UNCHANGED *mpi_vars*

MPI_Group_compare(*group*1, *group*2, *result*, *return*, *proc*) $\triangleq$
$\land$ *Assert*(*initialized*[*proc*] = "initialized", 200.10 – 200.12
"Error: MPI_Group_compare called before MPI_Init.")
$\land$ *result*′ = [*result* EXCEPT ![*proc*] =
IF $\lor$ *group*1 = *group*2 138.31
$\lor$ $\land$ *group*[*proc*][*group*1].*members* = *group*[*proc*][*group*2].*members*
$\land$ *group*[*proc*][*group*1].*ranking* = *group*[*proc*][*group*2].*ranking*
THEN
MPI_IDENT
ELSE 138.32
IF $\land$ *group*[*proc*][*group*1].*members* = *group*[*proc*][*group*2].*members*
$\land$ *group*[*proc*][*group*1].*ranking* $\neq$ *group*[*proc*][*group*2].*ranking*
THEN
MPI_SIMILAR
ELSE
MPI_UNEQUAL] 138.33
$\land$ UNCHANGED *mpi_vars*

Section 5.3.2 *Group* Constructors

MPI_Comm_group(*comm*, *gr*, *return*, *proc*) $\triangleq$
$\land$ *Assert*(*initialized*[*proc*] = "initialized", 200.10 – 200.12
"Error: MPI_Comm_group called before MPI_Init.")
$\land$ *Memory*′ = [*Memory* EXCEPT ![*proc*] = [@ EXCEPT ![*gr*] = *communicator*[*proc*][*comm*].*group*]] 139.19
$\land$ UNCHANGED *mpi_vars*

MPI_Group_union(*group*1, *group*2, *newgroup*, *return*, *proc*) $\triangleq$
$\land$ *Assert*(*initialized*[*proc*] = "initialized", \ * 200.10 – 200.12
"*Error* : *MPI_Group_union called before MPI_Init*.")
$\land \exists\, i \in 0\,..\,(MAX\_GROUP - 1)$:
LET *newmembers* $\triangleq$ *group*[*proc*][*group*1].*members* $\cup$

$group[proc][group2].members$

IN

$\wedge\ group[proc][i] = MPI\_GROUP\_EMPTY$

$\wedge\ newgroup' = [newgroup \ \text{EXCEPT}\ ![proc] = i]$

$\wedge\ group' =$

$[\text{group}\ \text{EXCEPT}\ ![proc] =$

$[@\ \text{EXCEPT}\ ![i] =$

$[members \mapsto newmembers,$

$size \mapsto Cardinality(newmembers),$

$ranking \mapsto$

$[j \in 0 \,..\, (Cardinality(newmembers) - 1) \mapsto$

$\text{IF}\ \ j < group[proc][group1].size$

$\text{THEN}\ group[proc][group1].ranking[j]$

$\text{ELSE}\ \ group[proc][group1].ranking[j]]]]]]\ \setminus *$ incorrect, need to fix

$\wedge\ \text{UNCHANGED}\ \langle communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$

Section 5.4.1 *Communicator* Accessors

$MPI\_Comm\_size(comm, size, return, proc) \triangleq$

$\wedge\ Assert(initialized[proc] = \text{"initialized"},$ 200.10 – 200.12

"MPI_Comm_size called with proc not in initialized state.")

$\wedge\ Memory' = [Memory\ \text{EXCEPT}\ ![proc] = [@\ \text{EXCEPT}\ ![size] = group[proc][communicator[proc][comm].g$

$\wedge\ \text{UNCHANGED}\ mpi\_vars$

$MPI\_Comm\_rank(comm, rank, return, proc) \triangleq$

$\wedge\ Assert(initialized[proc] = \text{"initialized"},$ 200.10 – 200.12

"MPI_Comm_rank called with proc not in initialized state.")

$\wedge\ Memory' = [Memory\ \text{EXCEPT}\ ![proc] = [@\ \text{EXCEPT}\ ![rank] = group[proc][communicator[proc][comm].$

$\wedge\ \text{UNCHANGED}\ \langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$

$MPI\_Comm\_compare(comm1, comm2, result, return, proc) \triangleq$

$\wedge\ Assert(initialized[proc] = \text{"initialized"},$ 200.10 – 200.12

"MPI_Comm_rank called with proc not in initialized state.")

$\wedge\ \text{IF}\ comm1 = comm2$

THEN

$result' = MPI\_IDENT$

ELSE

$\text{IF}\ \wedge\ communicator[proc][comm1].group = communicator[proc][comm2].group$

$\wedge\ communicator[proc][comm1].group.ranking = communicator[proc][comm2].group.ranking$

THEN

$result' = MPI\_CONGRUENT$

ELSE

$\text{IF}\ communicator[proc][comm1].group = communicator[proc][comm2].group$

THEN

$result' = MPI\_SIMILAR$

ELSE

$result' = MPI\_UNEQUAL$

$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$

Section 7.5 Startup

199.12 – 199.17

Initialize the participation of this process within a distributed computation.

$MPI\_Init(argc, argv, return, proc) \triangleq$

$\land$ $Assert(initialized[proc] =$ "uninitialized", 199.12
"MPI_Init called with proc not in uninitialized state.")

$\land$ $initialized' = [initialized$ EXCEPT $![proc] =$ "initialized"$]$ 199.13

$\land$ UNCHANGED $\langle Memory\rangle$

$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, requests, collective\rangle$

Finalize the participation of this process within a distributed computation.
Do buffered operations complete when the message is transmitted or buffered?

$MPI\_Finalize(return, proc) \triangleq$

$\land$ $Assert(initialized[proc] =$ "initialized", 200.10 – 200.12
"Error: MPI_Finalize called with proc not in initialized state.")

$\land$ $Assert(\forall i \in (1 \,..\, Len(requests[proc])) :$ 199.47
$\neg requests[proc][i].active$,
"Error: MPI_Finalize called when some message was still active.")

$\land$ $Assert(bufsize[proc] = 0$,
"Error: MPI_Finalize called before the buffer is detached.")

$\land$ $initialized' = [initialized$ EXCEPT $![proc] =$ "finalized"$]$ 199.46

$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, requests, collective\rangle$

$\land$ UNCHANGED $\langle Memory\rangle$

Determine whether *MPI_Init* has been called.

$MPI\_Initialized(flag, return, proc) \triangleq$

$\land$ $Memory' = [Memory$ EXCEPT $![proc] = [@$ EXCEPT $![flag] =$
IF $initialized[proc] =$ "initialized" 200.2
THEN TRUE
ELSE FALSE$]]$

$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$

"Best effort to clean up"

$MPI\_Abort(comm, errorcode, return, proc) \triangleq$

$\forall p \in (0 \,..\, (N-1)) :$
$\forall m \in (1 \,..\, Len(requests[p])) :$
$\land$ $requests[p][m].active$
$\land$ $\neg requests[p][m].transmitted$
$\Rightarrow requests[p][m]' = [requests[p][m]$ EXCEPT $!.cancelled =$ TRUE$]$

$\land$ UNCHANGED $\langle group, communicator, bufsize, message\_buffer, requests, initialized, collective\rangle$
