A Formal Specification of MPI 2.0

Guodong Li, Robert Palmer, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby

Abstract—Message passing using libraries implementing the Message Passing Interface (MPI) standard is the dominant communication mechanism in high performance computing (HPC) applications. Yet, the lack of an implementation independent formal semantics for MPI is a huge void that must be filled, especially given the fact that MPI will be implemented on novel hardware platforms in the near future. To help reason about programs that use MPI for communication, we have developed a formal TLA+ semantic definition of a majority of MPI 2.0 functions to augment the existing standard. We also present a framework to extract models from simple SPMD-style C programs, so that designers may understand the semantics of MPI by exercising short, yet pithy, communication scenarios written in C/MPI. In this paper, we describe (i) the TLA+ specification of MPI 2.0, (ii) the model extraction and verification framework that helps facilitate explicit-state model checking of formal semantic definitions, (iii) an error trail replay facility in the Visual Studio environment. These benefits suggest that a formal semantic definition and exploration approach as described here must accompany every future effort in creating parallel and distributed programming libraries.

Index Terms—MPI, Formal Specification, TLA+, Model Checking

1 Introduction

The Message Passing Interface (MPI, [27]) library has become a *de facto* standard in HPC, and is being actively developed and supported through several implementations [7], [26], [5]. However, it is well known that even experienced programmers misunderstand MPI APIs because they are described in natural languages. The behavior of APIs observed through ad hoc experiments on actual platforms is not a conclusive or comprehensive description of the standard. A formalization of the MPI standard will help users avoid misunderstanding the semantics of MPI functions. However, formal specifications, as currently written and distributed, are inaccessible to most practitioners.

In our previous work [18], we presented the formal specification of around 30% of the 128 MPI-1.0 primitives (mainly for point-to-point communication) in TLA+ [28]. TLA+ enjoys wide usage in industry by engineers (e.g. in Microsoft [29] and Intel). The TLA+ language is easy to learn. A new user can understand our specification and start practicing it after a half-an-hour tutorial. Additionally, in order to help practitioners access our specification, we built a C front-end in the Microsoft Visual Studio (VS) parallel debugger environment, through which users can submit and run short (perhaps tricky) MPI programs with embedded assertions, called litmus tests. A short litmus test may exhibit high degree of interleaving and its running will reveal the nuance of the semantics of involved MPI primitives. Such tests are turned into TLA+ code and run through the TLC model checker [28], which searches all the reachable states to check properties such as deadlocks and user defined invariants. This permits practitioners to play with (and find holes in) the semantics in a formal setting.

While we have demonstrated the merits of our previous work ([18]), this paper, the journal version of our poster paper [13], handles far more details including those pertaining to data transfers. In this work, we have covered much of MPI-2.0 (has over 300 API functions, as opposed to 128 for MPI-1.0). In addition, our

Supported in part by NSF award CNS-00509379 and Microsoft HPC Institutes Program.

- *G. Li, M. Delisi, G. Gopalakrishnan and R. M. Kirby are with School of Computing, University of Utah, Salt Lake City, Utah. E-mail: {ligd,delisi,ganesh,kirby}@cs.utah.edu.*
- *R. Palmer is with Intel Validation Research Labs, Oregon. E-mail: robert.l.palmer@intel.com.*

new work provides a rich collection of tests that help validate our specifications. It also modularizes the specification, permitting reuse.

Model Validation. In order to make our specification be faithful to the English description, we (i) organize the specification for *easy traceability*: many clauses in our specification are cross-linked with [27] to particular page/line numbers; (ii) provide comprehensive unit tests for MPI functions and a rich set of litmus tests for tricky scenarios; (iii) relate aspects of MPI to each other and verify the self-consistency of the specification (see section 4.10); (iv) provide a programming and debugging environment based on TLC, Phoenix, and Visual Studio to help engage expert MPI users (who may not be formal methods experts) into experimenting with our semantic definitions.

The structure of this paper is as follows. We first discuss the related work on formal specifications of large standards; other work on applying formal methods to verify MPI programs is also discussed. Then we give a motivating example and introduce the specification language TLA+. This example illustrates that vendor MPI implementations do not capture the nuance of the semantics of an MPI call. The formal specification is presented in section 4, where the operational semantics of representative MPI calls are presented in a language abstracted from TLA+. In section 5 we describe a C MPI front-end that translates MPI programs written in C into TLA+ code, plus the verification framework that helps users execute the semantics. Finally we give the concluding remarks.

2 Related Work

The idea of writing formal specifications of standards and building executable environments is a vast area. The IEEE Floating Point standard [10] was initially conceived as a standard that helped minimize the danger of non-portable floating point implementations, and now has incarnations in various higher order logic specifications (e.g., [8]), finding routine applications in *formal proofs* of modern microprocessor floating point hardware circuits. Formal specifications using TLA+ include Lamport's Win32 Threads API specification [29] and the RPC Memory Problem specified in TLA+ and formally verified in the Isabelle theorem prover by Lamport, Abadi, and Merz [1]. In [11], Jackson presents a lightweight object modeling notation called Alloy, which has tool support [12] in terms of formal analysis and testing based on Boolean satisfiability methods.

Each formal specification framework solves modeling and analysis issues specific to the object being described. In our case, we were initially not sure how to handle the daunting complexity of MPI nor how to handle its modeling, given that there has only been very limited effort in terms of formal characterization of MPI.

Georgelin and Pierre [6] specify some of the MPI functions in LOTOS [4]. Siegel and Avrunin [24] describe a finite state model of a limited number of MPI point-to-point operations. This finite state model is embedded in the SPIN model checker [9]. They [25] also support a limited partial-order reduction method – one that handles wild-card communications in a restricted manner, as detailed in [20]. Siegel [23] models additional 'non-blocking' MPI primitives in Promela. Our own past efforts in this area are described in [2], [17], [21], [19]. None of these efforts: (i) approach the number of MPI functions we handle, (ii) have the same style of high level specifications (TLA+ is much closer to mathematical logic than finite-state Promela or LOTOS models), (iii) have a model extraction framework starting from C/MPI programs, and (iv) have a practical way of displaying error traces in the user's C code.

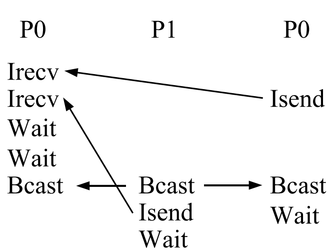
3 Motivation

MPI is a standardlized and portable message-passing system defining a core of library routines useful to a wide range of users writing portable message-passing programs in Fortran, C or C++. Version 1 and 2.0 were released in 1994 and 1997 respectively. Currently more than a dozen implementations exist, on a wide variety of platforms. All segments of the parallel computing communicty including vendors, library writers and application scientists will benefit from a formal specification of this standard.

3.1 Motivating Example

MPI is a portable standard and has a variety of implementations [7], [26], [5]. MPI programs are often manually or automatically (e.g., [3]) re-tuned when ported to another hardware platform, for example by changing its basic primitives (e.g., `MPI_Send`) to specialized versions (e.g., `MPI_Isend`). In this context, it is crucial that the designers performing code tuning are aware of the very fine details of the MPI semantics. Unfortunately, such details are far from obvious. For illustration, consider the following MPI pseudo-code involving three processes:

```
P0: {MPI_Irecv(rcvbuf1, *, req1);
     MPI_Irecv(rcvbuf2, from 1, req2);
     MPI_Wait(req1);
     MPI_Wait(req2);
     MPI_Bcast(revbuf3, root=1);
     ...}
P1: {sendbuf1 = 10;
     MPI_Bcast(sendbuf1, root=1);
     MPI_Isend(sendbuf2, to 0, req);
     MPI_Wait(req);
     ...}
P2: {sendbuf2 = 20;
     MPI_Isend(sendbuf2, to 0, req);
     MPI_Bcast(recvbuf2, root=1);
     MPI_Wait(req);
     ...}
```



Process 1 and 2 are designed to issue *immediate mode* sends to process 0, while Process 0 is designed to post two immediate-mode receives (*i.e.* it may match the send from P1 or P2), the first of which is a wildcard receive. These processes also participate in a broadcast communication with P1 as the root.

Consider some simple questions pertaining to the execution of this program:

1) *Is there a case where a deadlock is incurred?* If the broadcast is synchronizing such that the call at each process is blocking, then the answer is 'yes', since P0 cannot complete the broadcast before it receives the messages from P1 and P2, while P1 will not isend the message until the broadcast is complete. On the other hand, this deadlock will not occur if the broadcast is non-synchronizing. As in an actual MPI implementation `MPI_Bcast` may be implemented as synchronizing or non-synchronizing, this deadlock may not be observed through ad hoc experiments on a vendor MPI library. Our specification takes both bases into consideration and always gives reliable answers.
2) *Suppose the broadcast is non-synchronizing, is it possible that a deadlock occurs?* The answer is 'yes', since P0 may first receive a message from P1, then get stuck in waiting for another message from P1. Unfortunately, if we run this program in a vendor MPI implementation, P1 may receive messages first from P2 and then from P1. In this case no deadlock occurs. Thus it is likely that we do not encounter this deadlock even we run the program for 1,000 times. In contrast, the TLC model checker enumerates all execution possibilities and is guaranteed to detect this deadlock.
3) *Suppose there is no deadlock, is it guaranteed that rcvbuf1 in P0 will eventually contain the message sent from P2?* The answer is 'no', since P1's incoming messages may arrive out of order. However, running experiments on a vendor implementation may indicate that the answer is yes, esp. when the message delivery delay from P1 to P0 is greater than that from P2 to P0. In our framework, we can add in P0 an assentation $\texttt{rcvbuf1} == 20 \wedge \texttt{rcvbuf2} == 10$ right before the broadcast call. If it is possible under the semantics for other values to be assigned to these two variables, then the model checker will find the violation.
4) *Suppose there is no deadlock, when can the buffers be accessed?* Since all sends and receives use the immediate mode,

the handles that these calls return have to be tested for completion using an explicit `MPI_Test` or `MPI_Wait` before the associated buffers are allowed to be accessed. Vendor implementations may not give reliable answer for this question. In contrast, we can move the assertions mentioned in the response to the previous question to any other point before the corresponding `MPI_wait`s. The model checker then finds violations—meaning that the data cannot be accessed on the receiver until after the `wait`.

5) *Will the first receive always complete before the second?* No such guarantee exists, as these are *immediate mode* receives which are guaranteed only to be *initiated* in program order. Again, the result obtained by observing the running of this program in a vendor implementation may not be accurate. In order to answer this question, we can reverse the order of the `MPI_Wait` commands. If the model checker does not find a deadlock then it is possible for the operations to complete in either order.

The MPI reference standard [27] is a non machine-readable document that offers English descriptions of the individual behaviors of MPI primitives. It does not support any executable facility that helps answer the above kinds of simple questions in any tractable and reliable way. Running test programs, using actual MPI libraries, to reveal answers to the above kinds of questions is also futile, given that (i) various MPI implementations exploit the liberties of the standard by specializing the semantics in various ways, and (ii) it is possible that some executions of a test program are not explored in these actual implementations.

Thus we are motivated to write a formal, high-level, and executable standard specification for MPI 2.0. The availability of a formal specification allows formal analysis of MPI programs. For example, we have based on this formalization to create an efficient dynamic partial order reduction algorithm [22]. Moreover, the model checker incorporated in our framework — MPI-TLC — enables users to execute the formal semantic definitions and verify MPI programs.

3.2 TLA+ and TLC

The specification is written in TLA+ [28], a formal specification notation widely used in industry. It is a formal specification language based on (untyped) ZF set theory. Basically it combines the expressiveness of first order logic with temporal logic operators. TLA+ is particularly suitable for specifying and reasoning about concurrent and reactive systems.

TLC, a model checker for TLA+, explores all reachable states in the model defined by the system. TLC looks for a state (i.e. an assignment of values to variables) where (a) an invariant is not satisfied, (b) there are no exits (deadlocks), (c) the type invariant is violated, or (d) a user-defined TLA+ assertion is violated. When TLC detects an error, a minimal-length trace that leads to the bad state is reported (in our framework this trace turns into a Visual Studio debugger replay of the C source).

Why we opt for TLA+ against other specification languages/tools? ...

How easy to port the spec to other languages? ...

4 Specification

TLA+ provides basic modules for set, function, record, string and sequece. We first extend the TLA+ library by adding the definitions of advanced data structures including array, map, and ordered set (`oset`), which are used to model a variety of MPI objects. For instance, MPI groups and I/O files are represented as ordered sets.

The approximate sizes (without including comments and blank lines) of the major parts in the current TLA+ specification are shown in Table 1, where #funcs and #lines give the number of MPI functions and code lines respectively. We do not model primitives whose behavior depends on the underlying operating system. For deprecated items (e.g., MPI_KEYVAL_CREATE), we only model their replacement (MPI_COMM_CREATE_KEYVAL).

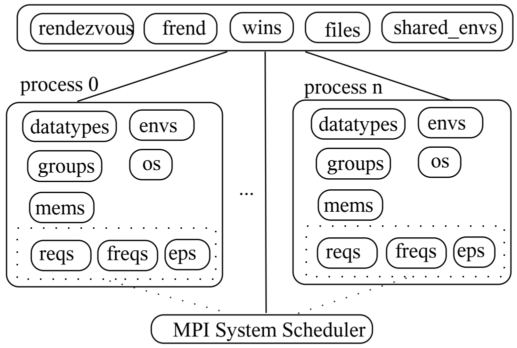


Fig. 1. MPI objects and their interaction

Main Module	#funcs(#lines)
Point to Point Communication	35(800)
Userdefined Datatype	27(500)
Group & Communicator Management	34(650)
Intra Collective Communication	16(500)
Topology	18(250)
Environment Management in MPI 1.1	10(200)
Process Management	10(250)
One sided Communication	15(550)
Inter Collective Communication	14(350)
I/O	50(1100)
Interface & Environment in MPI 2.0	35(800)

Table 1. Size of the Specification

4.1 Data Structures

The data structures modeling explicit and opaque MPI objects are shown in figure 1. Each process contains a set of local objects such as the local memory object `mems`. Multiple processes coordinate with each other through shared objects `rendezvous`, `wins`, and so on. The message passing procedure is simulated by the *MPI system scheduler*, whose task includes matching requests at origins and destinations and performing message passing.

Request object `reqs` is used in point-to-point communications to initiate and complete messages. A message contains the source, destination, tag, data type, count and communicator handle. It carries the data from the origin to the target. Note that noncontiguous data is represented as (user-defined) datatypes. Similarly, file request object `freqs` is for parallel I/O communications.

A group is used within a communicator to describe the participants in a communication "universe". Communicators `comms` are divided into two kinds: intra-communicators each of which has a single group of processes, and inter-communicators each of which has two groups of processes. A communicator also includes virtual topology and other attributes.

A rendezvous is a place shared by the processes participating in a collective communication. A process stores its data to the rendezvous on the entry of the communication and fetches the data from the rendezvous on the exit. Similarly `frend` is the rendezvous object for file operations.

For one-sided communications, epoches `epos` are used to control remote memory accesses; each epoch is associated with a "window", modeled by `wins`, which is made accessible to accesses by remote accesses. Similarly, a "file" supporting I/O accesses is shared by a group of processes.

Other MPI objects are represented as components in a shared environment `shared_envs` and local environments `envs`. The underlying operating system is abstracted as `os` in a limited sense, which includes those objects (such as physical files on the disk) visible to the MPI system. Since the physical memory at each process is an important object, we extract it from `os` and define a separate object `mems` for it.

4.2 Notations

We present our specification using notations extended from TLA+.

4.2.1 TLA+

The basic concept in TLA+ is functions. A set of functions is expressed by $[domain \rightarrow range]$. Notation $f[e]$ represents the application of function f on e; and $[x \in S \mapsto e]$ defines the function f such that $f[x] = e$ for $x \in S$. For example, the function f_{double} that doubles the input natural number is given by $[x \in \mathbb{N} \mapsto 2 \times x]$; and $f_{double}[4] = 8$.

For a n-tuple $\langle e_1, \cdots, e_n \rangle$, $e[i]$ returns its i^{th} component. It is actually a function mapping i to $e[i]$ for $1 \leq i \leq n$. Thus function f_{double} is equivalent to the tuple $\langle 2, 4, 6, 8, \cdots \rangle$. An order set is actually an array.

Notation $[f \texttt{ EXCEPT } ![e_1] = e_2]$ defines a function f' such that $f' = f$ except $f'[e_1] = e_2$. A @ appeared in e_2 represents the old value of $f[e_1]$. For example, $[f_{double} \texttt{ EXCEPT } ![3] = @{+}10]$ is the same as f_{double} except that it returns 16 when the input is 3. Similarly, $[r \texttt{ EXCEPT } !.h = e]$ represents a record r' such that $r' = r$ except $r'.h = e$, where $r.h$ returns the h-field of record r.

The basic temporal logic operator used to define transition relations is the next state operator, denoted using $'$ or *prime*. For example, $s' = [s \texttt{ EXCEPT } ![x] = e]$ indicates that the next state s' is equal to the original state s except that x's value is changed to e.

For illustration, consider a stop watch that displays hour and minute. A typical behavior of the clock is the sequence $(hr = 0, mnt = 0) \rightarrow (hr = 0, mnt = 1), \rightarrow, \cdots, \rightarrow, (hr = 0, mnt = 59), (hr = 1, mnt = 0), \rightarrow, \cdots$, where $(hr = 0, mnt = 1)$ is a state in which the hour and minute have the value 0 and 1 respectively.

The next-state relation is a formula expressing the relation between the values of hr and mnt in the old (first) state $time$ and new (second) state $time'$ of a step. It assert that mnt equals mnt + 1 except if mnt equals 59, in which case mnt is reset to 0 and hr is increased by 1.

$$
\begin{array}{l}
time' = \texttt{let } c = time[mnt] \neq 59 \texttt{ in} \\
\quad [time \texttt{ EXCEPT } ![mnt] = \text{if } c \text{ then } @ + 1 \text{ else } 0, \\
\qquad\qquad\qquad ![hr] = \neg c \ ? \ (@ + 1) \ : \ @ \\
\quad]
\end{array}
$$

Additionally, we introduce some commonly used notations when defining the semantics of MPI primitives.

$\Gamma_1 \diamond \Gamma_2$	the concatenation of queue Γ_1 and Γ_2
$\Gamma_1 \diamond x_k \diamond \Gamma_2$	the queue with x being the k^{th} element
ϵ	null value
α	an arbitrary value
$\top$ and $\bot$	boolean value $ture$ and $false$
$\Gamma_1 \sqsubseteq \Gamma_2$	Γ_1 is a sub-array (sub-queue) of Γ_2
$\overrightarrow{v}$	v is an array
$f \uplus \langle x, v \rangle$	a new function (map) f_1 such that $f_1[x] = v$ and $\forall y \neq x.\ f_1[y] = f[y]$
$f\vert_x$	the index of element x in function f, *i.e.* $f[f\vert_x] = x$
$c \ ? \ e_1 \ : \ e_2$	**if** c **then** x **else** y
$\texttt{size}(f)$	the number of elements in function f
$\texttt{remove}(f, k)$	remove from f the item at index k
$\texttt{unused\_index}(f)$	return an i such that $i \notin \texttt{DOM}(f)$

TLA+ allows us to specify operations in a declarative style. For illustration we show below a helper function used to implement the `MPI_COMM_SPLIT` primitive, where `DOM`, `RNG`, `CARD` return the domain, range and cardinality of a set respectively. This code directly formalizes the English description (see page 147 in [27]): "This function partitions the group into disjoint subgroups, one for each value of `color`. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by `key`, with ties broken according to their rank in the old group. When the process supply the color value `MPI_UNDEFINED`, a null communicator is returned." In contrast, such declarative specification cannot be done in the C language.

$$
\begin{array}{ll}
 & \texttt{Comm\_split}(group, \overrightarrow{colors}, \overrightarrow{keys}, proc) \doteq \\
1: & \texttt{let } rank = group|_{proc} \texttt{ in} \\
2: & \texttt{if } \overrightarrow{colors}[rank] = \texttt{MPI\_UNDEFINED} \texttt{ then } \texttt{MPI\_GROUP\_NULL} \\
3: & \texttt{else} \\
4: & \quad \texttt{let } s = \{k \in \texttt{DOM}(group) : \overrightarrow{colors}[k] = \overrightarrow{colors}[rank]\} \texttt{ in} \\
5: & \quad \texttt{let } s_1 = \\
6: & \quad\; \texttt{choose } g \in [0 \mathrel{..} \texttt{CARD}(s) - 1 \rightarrow \texttt{DOM}(group)] : \\
7: & \quad\; \wedge\ \texttt{RNG}(g) = s \\
8: & \quad\; \wedge\ \forall i, j \in s : \\
9: & \qquad\quad g|_i < g|_j \Rightarrow \\
10: & \qquad\qquad \vee\ \overrightarrow{keys}[i] < \overrightarrow{keys}[j] \\
11: & \qquad\qquad \vee\ \overrightarrow{keys}[i] = \overrightarrow{keys}[j] \wedge i < j \\
12: & \quad \texttt{in } [i \in \texttt{DOM}(s_1) \mapsto group[s_1[i]]]
\end{array}
$$

After collecting the color and key information from all other processes, a process $proc$ calls this function to create the group of a new function. Line 1 calculates the rank of this process in the group; line 4 obtains a set of processes of the same color as $proc$'s; lines 5-11 sort this set in the ascending order of keys, with ties broken according to the ranks. For example, suppose $group = \langle 2, 5, 1 \rangle$, $\overrightarrow{colors} = 1, 0, 0$ and $\overrightarrow{keys} = \langle 0, 2, 1 \rangle$. Then the call of this function at process 5 creates a new group $\langle 1, 5 \rangle$.

4.2.2 Operational Semantics

The formal semantics of a MPI function is modeled by a state transition. A system state consists of explicit and opaque objects mentioned above. We write $\texttt{obj}_p$ for the object `obj` at process p. For example, $\texttt{reqs}_p$ refers to the request object (for point-to-point communications) at process p.

We use notation $\stackrel{\circ}{=}$ to define the semantics of an MPI primitive, and $\doteq$ to introduce a helper function. The pre-condition $cond$ of a primitive, if exists, is specifies by "$\texttt{requires}\ \{cond\}$". An error occurs if this pre-condition is violated. In general a transition is expressed as $\frac{guard}{action}$, where $guard$ specifies the requirement for the transistion to be triggered, and $action$ defines how the MPI objects are updated after the transition. When the guard is satisfied, the action is performed and the system state is modified. A null guard will be omitted, meaning that the transition will always be made.

For instance, the semantics of `MPI_Buffer_detach` is shown below. The pre-condition says that buffer at process p must exist; the guard indicates that the call will block until all messages in the buffer have been transmitted (*i.e.* the buffer is empty); the action is to write the buffer address and the buffer size into the p's local memory, and deallocate the space taken by the buffer. The `buffer` locates in the `envs` object. A variable such as $buff$ is actually a reference to a location in the memory; in many cases we simply write $buff$ for $\mathtt{mems}_p[buff]$.

$$\mathtt{MPI\_Buffer\_detach}(buff, size, p) \doteq$$
$$\mathbf{requires}\ \{\mathtt{buffer}_p \neq \epsilon\}$$
$$\frac{\mathtt{buffer}_p.capacity = \mathtt{buffer}.max\_capacity}{\mathtt{mems}'_p[buff] = \mathtt{buffer}_p.buff \ \wedge \ \mathtt{mems}'_p[size] = \mathtt{buffer}_p.size \ \wedge \ \mathtt{buffer}'_p = \epsilon}$$

In the following we describe briefly the specification of representative MPI primitives. The semantics presented here are abstracted from the actual TLA+ code for succinctness and readability, which has been tested thoroughly using the TLC model checker. The entire specification including tests and examples and the verification framework are available online [15].

4.3 Point-to-point Communication

In our formalization, a blocking primitive is implemented as an asynchronous operation followed immediately by a wait operation, *e.g.* $\mathtt{MPI\_Ssend} = \mathtt{MPI\_Issend} + \mathtt{MPI\_Wait}$ and $\mathtt{MPI\_Sendrecv} = \mathtt{MPI\_Isend} + \mathtt{MPI\_Wait} + \mathtt{MPI\_Irecv} + \mathtt{MPI\_Wait}$. The semantics of core point to point communication functions are shown in figures 3 and 4; and an example illstruating how a MPI program is "executed" according to these semantics is in figure 2. The reader is supposed to refer to these semantics when reading through this section.

A process p appends its send or receive request containing the message to its request queue $\mathtt{reqs}_p$. A send request contains information about the destination process, the communication tag to be matched, the data value to be send, and the status (omitted here) of the message. This request also includes boolean flags indicating whether the request is persistent, active, live, canceled and deallocated or not. For brevity we do not show the last three flags when presenting the content of a request in the queue. In addition, in order to model a ready send, we include in a send request a field $prematch$ of format $\langle destination\ process, request\ index\rangle$ which refers to the receive request that matches this send request. A receive request has the similar format, except that it includes the buffer address and a field to store the incoming data. Initially the data is missing (represented by the "_" in the data field). Later on an incoming message from a sender will replace the "_" with the data it carries. Notation $v\_$ indicates that the data may be missing or contain a value. For example, $\langle buf, 0, *, \_, \top, \top, \langle 0, 5\rangle\rangle_2^{recv}$ is a receive request such that: (i) the source process is process 0, and the tag is `MPI_ANY_TAG`; (ii) the incoming data is still missing; (iii) it is a persistent request that is still active; (iv) it has been prematched with the send request with index 5 at process 0; (v) the index of this request in the request queue is 2.

A standard send may or may not buffer the outgoing message. If buffer space is available, then it behaves the same as a send in the buffered mode; otherwise it is equal to a send in the synchronous mode. A buffered mode send will buffer the outgoing message and may complete before a matching receive is posted; while a synchronous send will complete successfully only if a matching receive is posted. A ready mode send may be started only if the matching receive is already posted. This is represented by the guard in its rule, which requires there must be a matching receive posted in some process q.

Relation $\rightleftharpoons$ defines the meaning of "matching". When a send request matches a receive request before the actual transferring occurs, it stores the $\langle destination\ process,\ destination\ request\ index\rangle$ in its $prematch$ field (abbreviated as ω), and

p_0	p_1	p_2
$\texttt{Issend}(v_1, dst = 1, tag = 0, req = 0)$	$\texttt{Irecv}(b, src = 0, tag = *, req = 0)$	$\texttt{Irecv}(b, src = *, tag = *, req = 0)$
$\texttt{Irsend}(v_2, dst = 2, tag = 0, req = 1)$	$\texttt{Wait}(req = 0)$	$\texttt{Wait}(req = 0)$
$\texttt{Wait}(req = 0)$		
$\texttt{Wait}(req = 1)$		

step	*event*	$\texttt{reqs}_0$	$\texttt{reqs}_1$	$\texttt{reqs}_2$
1	$\texttt{issend}(v, 1, 0, 0)$	$\langle 1, 0, v, \bot, \top, \epsilon\rangle_0^{ssend}$		
2	$\texttt{irecv}(b, 0, *, 1)$	$\langle 1, 0, v, \bot, \top, \epsilon\rangle_0^{ssend}$	$\langle b, 0, *, \_, \bot, \top, \epsilon\rangle$	
3	$\texttt{irecv}(b, *, *, 2)$	$\langle 1, 0, v, \bot, \top, \epsilon\rangle_0^{ssend}$	$\langle b, 0, *, \_, \bot, \top, \epsilon\rangle_0^{recv}$	$\langle b, *, *, \_, \bot, \top, \epsilon\rangle_0^{recv}$
4	$\texttt{irsend}(v, 2, 0, 0)$	$\langle 1, 0, v_1, \bot, \top, \epsilon\rangle_0^{ssend} \diamond \langle 1, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rsend}$	$\langle b, 0, *, \_, \bot, \top, \epsilon\rangle_0^{recv}$	$\langle b, *, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{recv}$
5	$\texttt{transfer}(0, 1)$	$\langle 1, 0, \_, \bot, \top, \epsilon\rangle_0^{ssend} \diamond \langle 1, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rsend}$	$\langle b, 0, *, v_1, \bot, \top, \epsilon\rangle_0^{recv}$	$\langle b, *, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{recv}$
6	$\texttt{wait}(0, 0)$	$\langle 1, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rsend}$	$\langle b, 0, *, v_1, \bot, \top, \epsilon\rangle_0^{recv}$	$\langle b, *, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{recv}$
7	$\texttt{wait}(1, 0)$	$\langle 1, 0, v_2, \bot, \top, \langle 2, 0\rangle\rangle_1^{rsend}$	$\langle b, 0, *, v_1, \bot, \top, \epsilon\rangle_0^{recv}$	$\langle b, *, *, \_, \bot, \top, \langle 0, 1\rangle\rangle_0^{recv}$
8	$\texttt{transfer}(0, 2)$		$\langle b, 0, *, v_1, \bot, \top, \epsilon\rangle_0^{recv}$	$\langle b, *, *, v_2, \bot, \top, \langle 0, 1\rangle\rangle_0^{recv}$
9	$\texttt{wait}(0, 2)$		$\langle b, 0, *, v_1, \bot, \top, \epsilon\rangle_0^{recv}$	
10	$\texttt{wait}(0, 1)$			

Fig. 2. A point-to-point communication program and one of its possible executions. Process p_0 sends messages to p_1 and p_2 in synchronous send mode and ready send mode respectively. The scheduler first forwards the message to p_1, then to p_2. A request is deallocated after the wait call on it. The execution follows from the semantics shown in figure 3.

the corresponding receive request stores the $\langle source\ process,\ source\ request\ index\rangle$. In this case they are "pre-matched", enforcing that they must match in later transferring. A process sending data in the ready mode will establish the connection with its pre-matched receiver. If two requests are not pre-matched (*i.e.* their *prematch* fields are empty), the system will decide whether they match at run time by looking into their source, destination and tag information.

When a persistent communication request is created, we set its *presistent* flag. A communication using a persistent request is initiated by the `start` function. When this function is called, the request should be inactive. The request becomes active after the call. A pending, nonblocking communication can be canceled by a `cancel` call, which marks the request for cancellation. A `free_request` call marks the request object for deallocation and set the request handle to `MPI_REQUEST_NULL`. An ongoing communication will be allowed to complete and the request will be deallocated only after its completion.

The `wait` call returns when the operation identified by the request *req* is complete. Let us look closer at the definition of *send_wait* (see figure 3). When the call is made with a null or inactive request request, or the target process is null, the operation returns immediately (lines 1-3). Line 5 identifies the current request in the request queue. If the request has sent the data, and it is not persistent or has been marked for deallocation, then the request handle is set to `MPI_REQUEST_NULL` after the call (line 10). Otherwise (lines 7-8), if the data have not been sent (*i.e.* $v\_ \neq \_$), then the request is intact. Note that depending on the send mode the wait call may or may not complete in this case. If the data have been, then the request becomes inactive. If the request has marked to be canceled, then the call returns immediately (lines 11-12), allowing an ongoing communication to complete. A send in a synchronous mode will complete only if a matching receive is posted (lines 15-16). If no buffer is used, then a send will be blocked until the data is transferred (line 14,22); otherwise it returns intermediately. A send in buffer mode will also deallocate the data from the system buffer (line 20). After this call is over, the request is no longer "live", indicating that the corresponding wait operation has been called.

For a receive request, the call will complete only after the value field is filled (*i.e.* the data is not equal to _) and all previous incoming messages in the same request queue have been fetched. That is, matched messages must be fetched in a FIFO manner. This is guaranteed

`Data Structures`

$send\ request$: important fields + less important fields

$\langle destination(dst) : \texttt{int},\ tag : \texttt{int},\ value,\ persistent(pr) : \texttt{bool},\ active : \texttt{bool},\ prematch(\omega)\rangle^{mode} +$
$\langle cancelled : \texttt{bool},\ dealloc : \texttt{bool},\ live : \texttt{bool}\rangle$

$recv\ request$: important fields + less important fields

$\langle buffer(buf) : \texttt{int},\ source(src) : \texttt{int},\ tag : \texttt{int},\ value,\ persistent(pr) : \texttt{bool},\ active : \texttt{bool},\ prematch(\omega)\rangle^{recv} +$
$\langle cancelled : \texttt{bool},\ dealloc : \texttt{bool},\ live : \texttt{bool}\rangle$

$\texttt{ibsend}(v, dst, tag, p) \triangleq$ buffer send
$\texttt{requires}\ \{\texttt{size}(v) \leq \texttt{buffer}_p.vacancy\}$
$\texttt{reqs}'_p = reqs_p \diamond \langle dst, tag, v, \perp, \top, \epsilon\rangle^{bsend}\ \wedge$
$\texttt{buffer}'_p.vacancy = \texttt{buffer}_p.vacancy - \texttt{size}(v)$

$\texttt{issend}(v, dst, tag, p) \triangleq$ synchronous send
$\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle dst, tag, v, \perp, \top, \epsilon\rangle^{ssend}$

match a send request and a receive request
$(\langle p, dst, tag_p, \omega_p, k_p\rangle \doteqdot \langle src, q, tag_q, \omega_q, k_q\rangle) \doteq$
$\texttt{if}\ \omega_q = \epsilon\ \wedge\ \omega_q = \epsilon\ \texttt{then}$
$\quad \wedge \quad tag_p \in \{tag_q, \mathrm{ANY\_TAG}\}$
$\quad \wedge \quad q = dst$
$\quad \wedge \quad src \in \{p, \mathrm{ANY\_SOURCE}\}$
$\texttt{else}$
$\quad \omega_q = \langle q, k_q\rangle\ \wedge\ \omega_q = \langle p, k_p\rangle$

$\texttt{irsend}(v, dst, tag, p) \triangleq$ ready send

$$\frac{\exists q : \exists\langle src, tag_1, \_, pr_1, \top, \epsilon\rangle_k^{recv} \in \texttt{reqs}_q : \langle p, dst, tag, \epsilon, \texttt{len}(\texttt{reqs}_p)\rangle \doteqdot \langle src, q, tag_1, \epsilon, k\rangle}{\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle dst, tag, v, \perp, \top, \langle q, k\rangle\rangle^{rsend}\ \wedge\ \texttt{reqs}'_q.\omega = \langle p, \texttt{len}(\texttt{reqs}_p)\rangle}$$

$\texttt{isend} \triangleq \texttt{if}\ use\_buffer\ \texttt{then ibsend else issend}$

$\texttt{irecv}(buf, src, tag, p) \triangleq$
$\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle buf, src, tag, \_, \perp, \top, \epsilon\rangle^{recv}$

persistent (inactive) request for synchronous send
$\texttt{issend\_init}(\texttt{v}, \texttt{dst}, \texttt{tag}, \texttt{p}) \triangleq$
$\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle dst, tag, v, \top, \perp, \epsilon\rangle^{ssend}$

persistent (inactive) receive request
$\texttt{irecv\_init}(buf, src, tag, p) \triangleq$
$\texttt{reqs}'_p = \texttt{reqs}_p \diamond \langle buf, src, tag, \_, \top, \perp, \epsilon\rangle^{recv}$

$\texttt{start}(req, p) \triangleq$ start a persistent request
$\texttt{requires}\ \{\texttt{reqs}_p[req].pr\ \wedge\ \neg\texttt{reqs}_p[req].active\}$
$\texttt{reqs}'_p[req] = [\texttt{reqs}_p[req]\ \mathrm{EXCEPT}\ !.active = \top]$

$\texttt{cancel}(\texttt{req}, \texttt{p}) \triangleq$ cancel a request
$\texttt{reqs}'_p[req] = [\texttt{reqs}_p[req]\ \mathrm{EXCEPT}\ !.cancelled = \top]$

$\texttt{free\_request}(req, p) \triangleq$ free a request
$\texttt{let}\ \Gamma_1 \diamond \langle dst, tag, v\_, pr, act, \epsilon\rangle_{req}^{mode} \diamond \Gamma_2 = \texttt{reqs}_q\ \texttt{in}$
$\texttt{if}\ act\ \texttt{then}\ \texttt{reqs}'_p[req].dealloc = \top$
$\texttt{else}\ \texttt{reqs}'_p = \Gamma_1 \diamond \Gamma_2\ \wedge\ req' = \mathrm{REQUEST\_NULL}$

$\texttt{MPI\_Isend}(buf, count, dtype, dest, tag, comm, req, p) \triangleq$
$\wedge\ \texttt{isend}(read\_data(\texttt{mems}_p, buf, count, dtype),$
$\qquad \texttt{comms}_p[comm].groups[dest], tag, p)$
$\wedge\ req' = \texttt{len}(reqs_p)$

$\texttt{MPI\_Irecv}(buf, count, dtype, source, tag, comm, req, p) \triangleq$
$\wedge\ \texttt{irecv}(buf, \texttt{comms}_p[comm].groups[source], tag, p)$
$\wedge\ req' = \texttt{len}(reqs_p)$

$\texttt{recv\_wait}(req, p) \triangleq$ wait for a receive request to complete
$\vee\ req = \mathrm{REQUEST\_NULL}$
$\vee\ \texttt{reqs}_p[req].src = \mathrm{PROC\_NULL}$
$\vee\ \neg\texttt{reqs}_q[req].active$
$\vee$
$\quad \texttt{let}\ \Gamma_1 \diamond \langle dst, tag, v\_, pr, \top, \omega\rangle_{req}^{recv} \diamond \Gamma_2 = \texttt{reqs}_q\ \texttt{in}$
$\quad \texttt{let}\ \langle new\_reqs, new\_req\rangle =$
$\quad\quad \texttt{if}\ pr\ \wedge\ \neg\texttt{reqs}_p[req].dealloc\ \texttt{then}$
$\quad\quad\quad \langle \Gamma_1 \diamond \langle dst, tag, v\_, pr, \perp, \omega\rangle^{recv} \diamond \Gamma_2, req\rangle$
$\quad\quad \texttt{else}\ \langle \Gamma_1 \diamond \Gamma_2, \mathrm{REQUEST\_NULL}\rangle\ \texttt{in}$
$\quad \texttt{if}\ \texttt{reqs}_q[req].cancelled\ \texttt{then}$
$\quad\quad \texttt{reqs}'_p = new\_reqs\ \wedge\ req' = new\_req$
$\quad \texttt{else}$

$$\frac{v\_ \neq \_\ \wedge\ \nexists\langle buf_1, src_1, tag_1, v_1, pr_1, \top, \omega_1\rangle_k^{recv} \in \Gamma_1 : \langle dst, p, tag, \omega, req\rangle \doteqdot \langle src_1, q, tag_1, \omega_1, k\rangle}{\texttt{mems}'_p[buf] = v\_\ \wedge\ req' = new\_req\ \wedge\ \texttt{reqs}'_p = new\_reqs}$$

$\texttt{send\_wait}(req, p) \triangleq$ wait for a send request to complete
$\vee\ req = \mathrm{REQUEST\_NULL}$
$\vee\ \texttt{reqs}_p[req].dst = \mathrm{PROC\_NULL}$
$\vee\ \neg\texttt{reqs}_q[req].active$
$\vee$
$\quad \texttt{let}\ \Gamma_1 \diamond \langle dst, tag, v\_, pr, \top, \omega\rangle_{req}^{mode} \diamond \Gamma_2 = reqs_q\ \texttt{in}$
$\quad \texttt{let}\ \langle new\_reqs, new\_req\rangle =$
$\quad\quad \texttt{if}\ pr\ \wedge\ \neg\texttt{reqs}_p[req].dealloc\ \vee\ v\_ \neq \_\ \texttt{then}$
$\quad\quad\quad \langle \Gamma_1 \diamond \langle dst, tag, v\_, pr, (v\_ \neq \_), \omega\rangle^{mode} \diamond \Gamma_2, req\rangle$
$\quad\quad \texttt{else}\ \langle \Gamma_1 \diamond \Gamma_2, \mathrm{REQUEST\_NULL}\rangle\ \texttt{in}$
$\quad \texttt{if}\ \texttt{reqs}_q[req].cancelled\ \texttt{then}$
$\quad\quad \texttt{reqs}'_p = new\_reqs\ \wedge\ req' = new\_req$
$\quad \texttt{else if}\ mode = ssend\ \texttt{then}$

$$\frac{(\neg use\_buffer \Rightarrow (v\_ = \_))\ \wedge\ \exists q : \exists\langle src_1, tag_1, \_, pr_1, \top, \omega_1\rangle_k^{recv} \in \Gamma_1 : \langle dst, p, tag, \omega, req\rangle \doteqdot \langle src_1, q, tag_1, \omega_1, k\rangle}{\texttt{reqs}'_p = new\_reqs\ \wedge\ req' = new\_req}$$

$\quad \texttt{else if}\ mode = bsend\ \texttt{then}$
$\quad\quad \texttt{reqs}'_p = new\_reqs\ \wedge\ req' = new\_req\ \wedge$
$\quad\quad \texttt{buffer}'.capaticy = \texttt{buffer}.capaticy - \texttt{size}(v)$
$\quad \texttt{else}$

$$\frac{\neg use\_buffer \Rightarrow (v\_ = \_)}{reqs'_p = new\_reqs\ \wedge\ req' = new\_req}$$

Fig. 3. Modeling point-to-point communications (I)

message transferring from process p to process q

$$
\begin{array}{l}
\texttt{transfer}(p,q) \doteq \\
\quad \texttt{reqs}_p = \Gamma_1^p \diamond \langle dst, tag_p, v, pr_p, \top, \omega_p\rangle_i^{send} \diamond \Gamma_2^p \wedge \\
\quad \texttt{reqs}_q = \Gamma_1^q \diamond \langle buf, src, tag_q, \_, pr_q, \top, \omega_q\rangle_j^{recv} \diamond \Gamma_2^q \wedge \\
\quad \langle p, dst, tag_p, \omega_p, i\rangle \asymp \langle src, q, tag_q, \omega_q, j\rangle \wedge \\
\quad \nexists \langle dst, tag_1, v, pr_1, \top, \omega_1\rangle_m^{send} \in \Gamma_1^p : \\
\qquad \nexists \langle buf, src, tag_2, \_, pr_2, \top, \omega_2\rangle_n^{recv} \in \Gamma_1^q : \\
\qquad\quad \langle p, dst, tag_1, \omega_1, m\rangle \asymp \langle src, q, tag_2, \omega_2, n\rangle \\
\hline
\quad \texttt{reqs}_p' = \\
\qquad \texttt{let } b = \texttt{reqs}_p[i].live \texttt{ in} \\
\qquad\quad \texttt{if } \neg b \wedge \neg\texttt{reqs}_p[i].pr \texttt{ then } \Gamma_1^p \diamond \Gamma_2^p \\
\qquad\quad \texttt{else } \Gamma_1^p \diamond \langle dst, tag_p, \_, pr_p, b, \omega_p\rangle^{send} \diamond \Gamma_2^p \wedge \\
\quad \texttt{reqs}_q' = \\
\qquad \texttt{let } b = \texttt{reqs}_q[j].live \texttt{ in} \\
\qquad\quad \texttt{if } \neg b \wedge \neg\texttt{reqs}_q[j].pr \texttt{ then } \Gamma_1^q \diamond \Gamma_2^q \\
\qquad\quad \texttt{else } \Gamma_1^q \diamond \langle buf, p, tag_q, v, pr_q, b, \omega_q\rangle^{recv} \diamond \Gamma_2^q
\end{array}
$$

$$
\begin{array}{l}
has\_completed(req, p) \doteq \ \text{whether a request has completed} \\
\vee\ \exists \langle buf, src, tag, v, pr, \top, \omega\rangle_{req}^{recv} = \texttt{reqs}_q[req] \\
\vee\ \exists \langle dst, tag, v\_, pr, \top, \omega\rangle_{req}^{mode} = \texttt{reqs}_q[req] \wedge \\
\quad \vee\ mode = bsend \\
\quad \vee\ mode = rsend \wedge (use\_buffer \vee (v\_ = \_)) \\
\quad \vee\ mode = ssend \wedge (use\_buffer \vee (v\_ = \_)) \wedge \\
\qquad \exists q : \exists \langle buf_1, src_1, tag_1, \_, pr_1, \top, \omega_1\rangle_k^{recv} \in \texttt{reqs}_q : \\
\qquad \langle dst, p, tag, \omega, req\rangle \asymp \langle src_1, q, tag_1, \omega_1, k\rangle
\end{array}
$$

$$
\begin{array}{l}
wait(count, req, p) \doteq \ \text{wait for one request to complete} \\
\texttt{if } \texttt{reqs}_p[req].mode = recv \\
\quad \texttt{then } \texttt{recv\_wait}(req) \\
\quad \texttt{else } \texttt{send\_wait}(req)
\end{array}
$$

wait for any request in $\overrightarrow{req_{array}}$ to complete

$$
\begin{array}{l}
\texttt{wait\_any}(count, \overrightarrow{req_{array}}, index, p) \doteq \\
\texttt{if } \forall i \in 0..count-1 : \overrightarrow{req_{array}}[i] = \texttt{REQUEST\_NULL} \vee \\
\quad \neg\texttt{reqs}_p[\overrightarrow{req_{array}}[i]].active \\
\texttt{then } \texttt{UNDEFINED} \\
\texttt{else } \dfrac{\texttt{choose } i : has\_completed(\overrightarrow{req_{array}}[i], q)}{index' = i}
\end{array}
$$

wait for all requests in $\overrightarrow{req_{array}}$ to complete

$$
\begin{array}{l}
\texttt{wait\_all}(count, \overrightarrow{req\_array}, p) \doteq \\
\forall i \in 0 \mathinner{..} count-1 : wait(\overrightarrow{req_{array}}[i], q)
\end{array}
$$

wait for all enabled requests in $\overrightarrow{req_{array}}$ to complete

$$
\begin{array}{l}
\texttt{wait\_some}(incount, \overrightarrow{req_{array}}, outcount, \overrightarrow{indice_{array}}, p) \doteq \\
\texttt{if } \forall i \in 0 \mathinner{..} count-1 : \overrightarrow{req_{array}}[i] = \texttt{REQUEST\_NULL} \vee \\
\quad \neg\texttt{reqs}_p[\overrightarrow{req_{array}}[i]].active \\
\texttt{then } \texttt{UNDEFINED} \\
\texttt{else} \\
\quad \texttt{let } (\overrightarrow{index}, count) = \\
\qquad \texttt{choose } \overrightarrow{A} \sqsubseteq \overrightarrow{req_{array}},\ \texttt{max } k \in 1 \mathinner{..} incount-1 : \\
\qquad\quad \forall l \in 0 \mathinner{..} k-1 : has\_completed(\overrightarrow{A}[l], p) \\
\quad \texttt{in} \\
\quad\quad \dfrac{\texttt{wait\_all}(count, \overrightarrow{index}, p)}{\begin{array}{c} outcount' = count \wedge \\ \overrightarrow{indice_{array}}' = \overrightarrow{index} \end{array}}
\end{array}
$$

Fig. 4. Modeling point-to-point communications (II)

by the requirement that there exists no message in Γ_1 with the same source and tag as the current message. During the transition, the call will write the value into the memory and update the request queue. A non persistent request will be removed from the queue when it is finished.

It is the rule `transfer` that models the message passing mechanism: if a send message in process p's queue matches a receive request in q's queue, then the data is transferred. By matching we require: (1) the source and destination of the message and the receive request should match; so do their communication contexts and tags. (2) messages from the same source to the same destination should be matched in a FIFO order. When these requests are non persistent and not live (indicating that the corresponding wait operations have been called), they will be removed from the queues.

In our implementation, the requirement for a request to be complete is modeled by the $has\_completed$ function. A receive request is complete when the data have been received. A send request in the buffer mode is complete when the data have been buffered or transferred. This function is used to implement communication operations of multiple completions. For example, `MPI_Waitany` blocks until one of the communication associated with requests in the array has completed. It returns in $index$ the array location of the completed request. `MPI_Waitall` blocks until all communications complete, and returns the statuses of all requests. `MPI_Waitsome` waits until at least one of the communications completes and returns the completed requests.

4.4 Datatype

A general datatype is an opague object that specifies a sequence of basic datatypes and integer displacements. The `extend` of a datatype is the span from the first byte to the last byte in this datatype. A datatype can be derived from simpler datatypes through datatype constructors. The simplest datatype constructor, modeled by `contiguous_copy`, allows replication of a datatype into contiguous locations. For example, $contiguous\_copy(2, \langle\langle \texttt{double}, 0\rangle, \langle \texttt{char}, 8\rangle\rangle)$ results in $\langle\langle \texttt{double}, 0\rangle, \langle \texttt{char}, 8\rangle, \langle \texttt{double}, 16\rangle, \langle \texttt{char}, 24\rangle\rangle$.

Constructor `type_vector` constructs a type consisting of the replication of a datatype into locations that consist of equally spaced blocks; each block is obtained by concatenating the same number of copies of the old datatype. `type_indexed` allows one to specify a noncontiguous data layout where displacements between blocks need not be equal. `type_struct` is the most general type constructor; it allows each block to consist of replications of different datatypes. These constructors are defined with the `contiguous_copy` constructor and the `set_offset` function (which increases the displacements of the items in the type by a certain offset). Other constructors are defined similarly. For instance,

$$
\begin{aligned}
&\texttt{type\_vector}(2,2,3,\langle\langle\texttt{double},0\rangle,\langle\texttt{char},8\rangle\rangle) = \\
&\quad \langle\langle\texttt{double},0\rangle,\langle\texttt{char},8\rangle,\langle\texttt{double},16\rangle,\langle\texttt{char},24\rangle, \\
&\quad\ \langle\texttt{double},48\rangle,\langle\texttt{char},56\rangle,\langle\texttt{double},64\rangle,\langle\texttt{char},72\rangle\rangle \\
&\texttt{type\_indexed}(2,\langle 3,1\rangle,\langle 4,0\rangle,\langle\langle\texttt{double},0\rangle,\langle\texttt{char},8\rangle\rangle) = \\
&\quad \langle\langle\texttt{double},64\rangle,\langle\texttt{char},72\rangle,\langle\texttt{double},80\rangle,\langle\texttt{char},88\rangle, \\
&\quad\ \langle\texttt{double},96\rangle,\langle\texttt{char},104\rangle,\langle\texttt{double},0\rangle,\langle\texttt{char},8\rangle\rangle \\
&\texttt{type\_struct}(3,\langle 2,1,3\rangle,\langle 0,16,26\rangle,\langle\texttt{float}, \\
&\quad \langle\langle\texttt{double},0\rangle,\langle\texttt{char},8\rangle\rangle,\texttt{char}\rangle) = \\
&\quad \langle\langle\texttt{float},0\rangle,\langle\texttt{float},4\rangle,\langle\texttt{double},16\rangle,\langle\texttt{char},24\rangle, \\
&\quad\ \langle\texttt{char},26\rangle,\langle\texttt{char},27\rangle,\langle\texttt{char},28\rangle\rangle
\end{aligned}
$$

When creating a new type at process p, we store the type in an unused place in the $\texttt{datatypes}_p$ object, and have the output reference *datatype* point to this place. When deleting a datatype at process p, we remove it from the $\texttt{datatypes}_p$ object and set the reference to `MPI_DATATYPE_NULL`. Derived datatypes support the specification of noncontiguous communication buffers. We show here how to read data from such buffers: noncontiguous data are "packed" into contiguous data which may be "upacked" later in accordance to other datatypes.

Datatype operations are local function — no interprocess communication is needed when such an operation is executed. In the transition relations, only the `datatypes` object at the calling process is modified. For example, the transition implementing `MPI_Type_index` is as follows. Note that arguement *blocklengths* is actually the start address of the block length array in the memory; auguments *oldtype* and *newtype* store the references to datatypes in the

Data Structures

$$typemap : \langle type, displacement(disp) : \texttt{int}\rangle\ \texttt{array}$$

replicate a datatype into contiguous locations

$$
\begin{aligned}
& contiguous\_copy(count, dtype) \doteq \\
& \texttt{let}\ F(i) = \\
& \quad \texttt{if}\ i = 1\ \texttt{then}\ dtype \\
& \quad \texttt{else}\ F(i-1) \diamond \\
& \qquad [k \in \mathrm{DOM}(dtype) \mapsto \\
& \qquad \langle dtype[k].type, dtype[k].disp + (i-1) * \texttt{extend}(dtype)\rangle \\
& \qquad] \\
& \texttt{in}\ F(count)
\end{aligned}
$$

$$set\_offset(dtype, offset) \doteq$$ adjust displacements

$$\quad [k \in \mathrm{DOM}(dtype) \mapsto \langle dtype[k].type, dtype[k] + offset\rangle]$$

$$
\begin{aligned}
& \texttt{type\_vector}(count, blocklength, stride, dtype) \stackrel{\circ}{=} \\
& \texttt{let}\ F(i) = \\
& \quad \texttt{if}\ i = count\ \texttt{then}\ \langle\rangle \\
& \quad \texttt{else}\ contiguous\_copy(blocklength, \\
& \qquad\quad set\_offset(dtype, \texttt{extend}(dtype) * stride * i)) \\
& \qquad \diamond F(i+1) \\
& \texttt{in}\ F(0)
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{type\_indexed}(count, \overrightarrow{blocklengths}, \overrightarrow{displacements}, dtype) \stackrel{\circ}{=} \\
& \texttt{let}\ F(i) = \\
& \quad \texttt{if}\ i = 0\ \texttt{then}\ \langle\rangle \\
& \quad \texttt{else}\ F(i-1) \diamond contiguous\_copy(\overrightarrow{blocklengths}[i-1], \\
& \qquad set\_offset(dtype, \overrightarrow{displacements}[i-1] * \texttt{extend}(dtype)) \\
& \texttt{in}\ F(count)
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{type\_struct}(count, \overrightarrow{blocklengths}, \overrightarrow{displacements}, \overrightarrow{dtypes}) \stackrel{\circ}{=} \\
& \texttt{let}\ F(i) = \\
& \quad \texttt{if}\ i = 0\ \texttt{then}\ \langle\rangle \\
& \quad \texttt{else}\ F(i-1) \diamond contiguous\_copy(\overrightarrow{blocklengths}[i-1], \\
& \qquad\quad set\_offset(\overrightarrow{dtypes}[i-1], \overrightarrow{displacements}[i-1]) \\
& \texttt{in}\ F(count)
\end{aligned}
$$

$$
\begin{aligned}
& create\_datatype(datatype, dtype, p) \doteq \\
& \texttt{let}\ index = \texttt{unused\_index}(\texttt{datatypes}_p)\ \texttt{in} \\
& \quad \texttt{datatypes}'_p[index] = dtype\ \wedge \\
& \quad datatype' = index
\end{aligned}
$$

$$
\begin{aligned}
& \texttt{type\_free}(datatype, p) \\
& \texttt{datatypes}'_p = datatype_p \setminus \{datatypes_p[datatype]\}\ \wedge \\
& datatype' = \texttt{DATATYPE\_NULL}
\end{aligned}
$$

read (non-contiguous) data from the memory

$$
\begin{aligned}
& read\_data(mem, buf, count, dtype) \doteq \\
& \texttt{let}\ read\_one(buf) = \\
& \quad \texttt{let}\ F_1(i) = \\
& \qquad \texttt{if}\ i = 0\ \texttt{then}\ \langle\rangle \\
& \qquad \texttt{else}\ F_1(i-1) \diamond mem[buf + dtype[i-1].disp] \\
& \quad \texttt{in}\ F_1(\texttt{size}(tp)) \\
& \texttt{in let}\ F_2(i \in 0\,..\,count) = \\
& \quad \texttt{if}\ i = 0\ \texttt{then}\ \langle\rangle \\
& \quad \texttt{else}\ F_2(i-1) \diamond \\
& \qquad\quad read\_one(buf + (i-1) * \texttt{extend}(dtype)) \\
& \texttt{in}\ F_2(count)
\end{aligned}
$$

Fig. 5. Modeling datatype operations

`datatypes` objects.

$$\begin{array}{l}
\texttt{MPI_Type_index}(count, blocklengths, displacements, oldtype,\\
\quad newtype, p) \mathrel{\hat=}\\
\texttt{let } \overrightarrow{lengths} = [i \in 0 \mathinner{..} count \mapsto \mathrm{mems}_p[blocklengths + i]] \texttt{ in}\\
\texttt{let } \overrightarrow{displacements} =\\
\quad [i \in 0 \mathinner{..} count \mapsto \mathrm{mems}_p[displacements + i]] \texttt{ in}\\
\texttt{let } type_index = \texttt{unused_item}(datatypes_p) \texttt{ in}\\
\texttt{let } dtype = \mathrm{datatypes}_p[oldtype] \texttt{ in}\\
\wedge\ datatypes_p[type_index] =\\
\quad \texttt{type_indexed}(count, \overrightarrow{blocklengths}, \overrightarrow{displacements}, dtype)\\
\wedge\ \mathrm{mems}_p[newtype]' = type_index
\end{array}$$

4.5 Collective Communication

All processes participating in the communication coordinate with each other through the shared `rendezvous` (or `rend`) object. There is a `rend` object corresponding to each communicator; and `rend`$[cid]$ refers to the rendezvous used by the communicator with context id cid. A `rend` object consists of a sequence of communication slots, each of which stores the information of a collective communication. In each slot, the $status$ field (abbreviated as τ) records the status of each process: 'e' ('entered'), 'l' ('left') or 'v' ('vacant', which is the initial value); the $shared\_data$ field stores the data shared among all processes; and $data$ stores the data sent by each process to the rendezvous. We use the notation Ψ to represent the content in a `rend` object.

Most collective communications are synchronizing, while the rest (like `MPI_Bcast`) can either be synchronizing or non-synchronizing. A collective primitive is implemented by a loose synchronization protocol: in the first "init" phase, process p checks whether there exists a slot such that p has not participant in. A negative answer means that p is initializing a new collective communication, thus p creates a new slot, sets its status to be 'entered' and stores its value v in this slot. If there are slots indicating that p has not joined the associated communications (*i.e.* p's status is 'v'), then p registers itself in the first of such slots by updating its status and value in the slot. Rule syn_{init} and syn_{write} are the simplified cases of syn_{put}. This phase is the same for both synchronizing and non-synchronizing communications.

After the "init" phase, process p proceeds to its next "wait" phase. Among all the slots p locates the first one indicating that it has entered but not left the associated communication. If the communication is synchronizing, then it has to wait until all other processes in the same communication have finished their "init" phases; otherwise it does not have to wait. If p is the last process that leaves, then the entire collective communication is over and the communication slot can be removed from the queue; otherwise p just updates its status to be 'left'.

$$\begin{array}{l}
\texttt{Data Structures}\\
\quad rendezvous\ for\ a\ communication:\\
\qquad \langle status(\tau) : [p : \texttt{int} \rightarrow \{\text{'}l\text{'}, \text{'}e\text{'}, \text{'}v\text{'}\}],\\
\qquad\ shared_data(sdata),\ data : [p : \texttt{int} \rightarrow value]\rangle\texttt{array}
\end{array}$$

process p joins the communication and stores the shared data v_s and its own data v in the rendevous

$$\begin{array}{l}
\texttt{syn}_{\texttt{put}}(cid, v_s, v, p) \doteq\\
\texttt{if } cid \notin \mathrm{DOM}(\mathrm{rend}) \texttt{ then}\\
\quad \mathrm{rend}'[cid] = \langle\langle\langle p, \text{'}e\text{'}\rangle\rangle, v_s, \langle\langle p, v\rangle\rangle\rangle\\
\texttt{else if } \forall slot \in \mathrm{rend}[cid] : slot.\tau[p] \in \{\text{'}e\text{'}, \text{'}l\text{'}\} \texttt{ then}\\
\quad \mathrm{rend}'[cid] = \mathrm{rend}[cid] \diamond \langle\langle\langle p, \text{'}e\text{'}\rangle\rangle, v_s, \langle\langle p, v\rangle\rangle\rangle\\
\texttt{else}\\
\quad \dfrac{\begin{array}{l}\mathrm{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus \langle p, \text{'}v\text{'}\rangle, v_s, S_v\rangle \diamond \Gamma_2 \wedge\\ \forall slot \in \Gamma_1 : slot.\tau[p] \neq \text{'}v\text{'}\end{array}}{\begin{array}{l}\mathrm{rend}'[cid] =\\ \Gamma_1 \diamond \langle \Psi \uplus \langle p, \text{'}e\text{'}\rangle,\ v_s,\ S_v \uplus \langle p, v\rangle\rangle \diamond \Gamma_2\end{array}}
\end{array}$$

$$\texttt{syn}_{\texttt{init}}(cid, p) \doteq \texttt{syn\_write}(cid, \epsilon, \epsilon, p)$$

$$\texttt{syn}_{\texttt{write}}(cid, v, p) \doteq \texttt{syn\_write}(cid, \epsilon, v, p)$$

process p leaves the synchronizaing communication

$$\begin{array}{l}
\texttt{syn}_{\texttt{wait}}(cid, p) \doteq\\
\dfrac{\begin{array}{l}\mathrm{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus \langle p, \text{'}e\text{'}\rangle, v_s, S_v\rangle \diamond \Gamma_2 \wedge\\ \forall k \in \mathrm{comms}_p[cid].group : \Psi[k] \in \{\text{'}e\text{'}, \text{'}l\text{'}\} \wedge\\ \forall slot \in \Gamma_1 : slot.\tau[p] = \text{'}e\text{'}\end{array}}{\begin{array}{l}\mathrm{rend}'[cid] =\\ \quad \texttt{if } \forall k \in \mathrm{comms}_p[cid].group : \Psi[k] = \text{'}l\text{'} \texttt{ then } \Gamma_1 \diamond \Gamma_2\\ \quad \texttt{else } \Gamma_1 \diamond \langle \Psi \uplus \langle p, \text{'}l\text{'}\rangle,\ v_s,\ S_v\rangle \diamond \Gamma_2\end{array}}
\end{array}$$

process p leaves the non-synchronizaing communication

$$\begin{array}{l}
\texttt{asyn}_{\texttt{wait}}(cid, p) \doteq\\
\dfrac{\begin{array}{l}\mathrm{rend}[cid] = \Gamma_1 \diamond \langle \Psi \uplus \langle p, \text{'}e\text{'}\rangle, v_s, S_v\rangle \diamond \Gamma_2 \wedge\\ \forall slot \in \Gamma_1 : slot.\tau[p] = \text{'}e\text{'}\end{array}}{\begin{array}{l}\mathrm{rend}'[cid] =\\ \quad \texttt{if } \forall k \in \mathrm{comms}_p[cid].group : \Psi[k] = \text{'}l\text{'} \texttt{ then } \Gamma_1 \diamond \Gamma_2\\ \quad \texttt{else } \Gamma_1 \diamond \langle \Psi \uplus \langle p, \text{'}l\text{'}\rangle,\ v_s,\ S_v\rangle \diamond \Gamma_2\end{array}}
\end{array}$$

Fig. 6. The basic protocol for collective communications

These protocols are used to specify collective communication primitives. For example,

p_0	p_1	p_2
$\texttt{syn}_{\texttt{put}}(cid = 0, sdata = v_s, data = v_0)$	$\texttt{syn}_{\texttt{init}}(cid = 0)$	$\texttt{syn}_{\texttt{write}}(cid = 0, data = v_2)$
$\texttt{asyn}_{\texttt{wait}}(cid = 0)$	$\texttt{syn}_{\texttt{wait}}(cid = 0)$	$\texttt{syn}_{\texttt{wait}}(cid = 0)$
$\texttt{syn}_{\texttt{init}}(cid = 0)$		

step	*event*	rend[0]
1	$\texttt{syn}_{\texttt{put}}(0, v_s, v_0, 0)$	$\langle\langle\langle 0, \text{`}e\text{'}\rangle\rangle, v_s, \langle\langle 0, v_0\rangle\rangle\rangle$
2	$\texttt{syn}_{\texttt{init}}(0, 1)$	$\langle\langle\langle 0, \text{`}e\text{'}\rangle, \langle 1, \text{`}e\text{'}\rangle\rangle, v_s, \langle\langle 0, v_0\rangle\rangle\rangle$
3	$\texttt{syn}_{\texttt{wait}}(0, 0)$	$\langle\langle\langle 0, \text{`}l\text{'}\rangle, \langle 1, \text{`}e\text{'}\rangle\rangle, v_s, \langle\langle 0, v_0\rangle\rangle\rangle$
4	$\texttt{syn}_{\texttt{init}}(0, 0)$	$\langle\langle\langle 0, \text{`}l\text{'}\rangle, \langle 1, \text{`}e\text{'}\rangle\rangle, v_s, \langle\langle 0, v_0\rangle\rangle\rangle \diamond \langle\langle\langle 0, \text{`}e\text{'}\rangle\rangle, \epsilon, \langle\rangle\rangle$
5	$\texttt{syn}_{\texttt{write}}(0, v_2, 2)$	$\langle\langle\langle 0, \text{`}l\text{'}\rangle, \langle 1, \text{`}e\text{'}\rangle, \langle 2, \text{`}e\text{'}\rangle\rangle, v_s, \langle\langle 0, v_0\rangle, \langle 2, v_2\rangle\rangle\rangle \diamond \langle\langle\langle 0, \text{`}e\text{'}\rangle\rangle, \epsilon, \langle\rangle\rangle$
6	$\texttt{syn}_{\texttt{wait}}(0, 2)$	$\langle\langle\langle 0, \text{`}l\text{'}\rangle, \langle 1, \text{`}e\text{'}\rangle, \langle 2, \text{`}l\text{'}\rangle\rangle, v_s, \langle\langle 0, v_0\rangle, \langle 2, v_2\rangle\rangle\rangle \diamond \langle\langle\langle 0, \text{`}e\text{'}\rangle\rangle, \epsilon, \langle\rangle\rangle$
7	$\texttt{syn}_{\texttt{wait}}(0, 1)$	$\langle\langle\langle 0, \text{`}e\text{'}\rangle\rangle, \epsilon, \langle\rangle\rangle$

Fig. 7. An example using the collective protocol. Three processes participate in collective communications via a communicator with context ID = 0. Process p_0's asynchronous wait returns even before p_2 joins the synchronization; it also initializes a new synchronization after it returns. Process p_2, the last one joining the synchronization, deallocates the slot. The execution follows from the semantics shown in figure 6.

MPI_Bcast is implemented as two transitions: MPI_Bcast$_{init}$ and MPI_Bcast$_{wait}$. The root first sends its data to the rendezvous in MPI_Bcast$_{init}$, then by using the asyn$_{wait}$ rule it can return immediately without waiting for the completion of other processes. On the other hand, if the call is synchronizing then it will use the syn$_{wait}$ rule. In contrast, a non-root process p needs to call the syn$_{wait}$ because it must wait for the data from the root to "reach" the rendezvous.

In the MPI_Gather call, each process including the root sends data to the root; and the root stores all data in rank order. Expression $[i \in \text{DOM}(gr) \rightarrow \texttt{rend}_p[comm.cid].data[gr[i]]]$ returns the concatenation of the data of all processes in rank order. Function $write\_data$ writes an array of data into the memory. MPI_Scatter is the inverse operation to MPI_Gather. In MPI_Alltoall, each process sends distinct data to each of the receivers. The j^{th} block sent from process i is received by process j and is placed in the i^{th} block of the receive buffer. Additionally, data from all processes in a group can be combined using a reduction operation op. The call of MPI_Scan at a process with rank i returns in the receive buffer the reduction of the values from processes with ranks $0, \cdots, i$ (inclusive).

MPI-2 introduces extensions of many of MPI-1 collective routines to intercommunicators, each of which contain a local group and a remote group. In this case, we just need to replace $\texttt{comms}_p[cid].group$ with $\texttt{comms}_p[cid].group \cup \texttt{comms}_p[cid].remote\_group$ in the rules shown in figure 6. In our TLA+ specification we take both cases into account when designing the collective protocol.

For example, if the $comm$ in MPI_Bcast is an intercommunicator, then the call involves all processes in the intercommunicator, broadcasting from the root in one group (group A) to all processes in the other group (group B). All processes in group B pass the same value in argument $root$, which is the rank of the root in group A. The root passes the value MPI_ROOT in $root$, and other processes in group A pass the value MPI_PROC_NULL in $root$.

4.6 Communicator

Message passing in MPI is via communicators, each of which specifies a set (group) of processes that participate in the communication. Communicators can be created and destroyed dynamically by coordinating processes. Information about topology and other attributes of a communicator can be updated too. An intercommunicator is used for communication between two disjoint groups of processes. No topology is associated with an intercommunicator.

$$
\begin{array}{l}
\texttt{bcast}_{\texttt{init}}(\mathit{buf}, v, \mathit{root}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad (\mathit{comm.group}[\mathit{root}] = p)\ ? \\
\qquad \texttt{syn}_{\texttt{put}}(\mathit{comm.cid}, v, \epsilon, p)\ :\ \texttt{syn}_{\texttt{init}}(\mathit{comm.cid}, p) \\
\texttt{bcast}_{\texttt{wait}}(\mathit{buf}, v, \mathit{root}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad \texttt{if}\ \mathit{comm.group}[\mathit{root}] = p\ \texttt{then}\ \mathit{need\_syn}\ ? \\
\qquad \texttt{syn}_{\texttt{wait}}(\mathit{comm.cid}, p)\ :\ \texttt{asyn}_{\texttt{wait}}(\mathit{comm.cid}, p) \\
\quad \texttt{else}\ \texttt{syn}_{\texttt{wait}}(\mathit{comm.cid}, p) \wedge \\
\quad \mathtt{mems}'_p[\mathit{buf}] = \mathtt{rend}_p[\mathit{comm.cid}].\mathit{sdata}
\end{array}
$$

$$
\begin{array}{l}
\texttt{gather}_{\texttt{init}}(\mathit{buf}, v, \mathit{root}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad \texttt{syn}_{\texttt{write}}(\mathit{comm.cid}, v, p) \\
\texttt{gather}_{\texttt{wait}}(\mathit{buf}, v, \mathit{root}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad \texttt{if}\ \mathit{comm.group}[\mathit{root}] \neq p\ \texttt{then}\ \mathit{need\_syn}\ ? \\
\qquad \texttt{syn}_{\texttt{wait}}(\mathit{comm.cid}, p)\ :\ \texttt{asyn}_{\texttt{wait}}(\mathit{comm.cid}, p) \\
\quad \texttt{else}\ \texttt{syn}_{\texttt{wait}}(\mathit{comm.cid}, p) \wedge \\
\quad \texttt{let}\ \mathit{gr} = \mathit{comm.group}\ \texttt{in} \\
\quad \texttt{let}\ \mathit{data} = [i \in \mathtt{DOM}(\mathit{gr}) \rightarrow \\
\qquad\qquad\qquad \mathtt{rend}_p[\mathit{comm.cid}].\mathit{data}[\mathit{gr}[i]]]\ \texttt{in} \\
\quad \mathtt{mems}'_p = \mathit{write\_data}(\mathtt{mems}_p, \mathit{buf}, \mathit{data})
\end{array}
$$

$$
\begin{array}{l}
\texttt{scatter}_{\texttt{init}}(\mathit{buf}, \overrightarrow{v}, \mathit{root}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad (\mathit{comm.group}[\mathit{root}] = p)\ ? \\
\qquad \texttt{syn}_{\texttt{put}}(\mathit{comm.cid}, \overrightarrow{v}, \epsilon, p)\ :\ \texttt{syn}_{\texttt{init}}(\mathit{comm.cid}, p) \\
\texttt{scatter}_{\texttt{wait}}(\mathit{buf}, \overrightarrow{v}, \mathit{root}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad \texttt{if}\ \mathit{comm.group}[\mathit{root}] = p\ \wedge\ \neg \mathit{need\_syn}\ \texttt{then} \\
\qquad \texttt{asyn}_{\texttt{wait}}(\mathit{comm.cid}, p) \\
\quad \texttt{else}\ \texttt{syn}_{\texttt{wait}}(\mathit{comm.cid}, p) \wedge \\
\quad \mathtt{mems}'_p[\mathit{buf}] = \mathtt{rend}_p.\mathit{sdata}[\mathit{comm.group}|_p]
\end{array}
$$

$$
\begin{array}{l}
\texttt{alltoall}_{\texttt{init}}(\mathit{buf}, \overrightarrow{v}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad \texttt{syn\_write}(\mathit{comm.cid}, \overrightarrow{v}, p) \\
\texttt{alltoall}_{\texttt{wait}}(\mathit{buf}, \overrightarrow{v}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad \texttt{syn}_{\texttt{wait}}(\mathit{comm.cid}, p) \wedge \\
\quad \texttt{let}\ \mathit{gr} = \mathit{comm.group}\ \texttt{in} \\
\quad \texttt{let}\ \mathit{data} = [i \in \mathtt{DOM}\ \mathit{gr} \rightarrow \\
\qquad\qquad\qquad \mathtt{rend}[\mathit{comm.cid}].\mathit{data}[\mathit{gr}[i]][\mathit{gr}|_p]]\ \texttt{in} \\
\quad \mathtt{mems}'_p = \mathit{write\_data}(\mathtt{mems}_p, \mathit{buf}, \mathit{data})
\end{array}
$$

$$
\begin{array}{l}
\mathit{reduce\_range}(\texttt{op}, \overrightarrow{\mathit{data}}, \mathit{start}, \mathit{end}) \doteq \\
\quad \texttt{let}\ F(i) = \texttt{if}\ i = \mathit{start}\ \texttt{then}\ \overrightarrow{\mathit{data}}[i] \\
\qquad\qquad\qquad \texttt{else}\ \texttt{op}(F(i-1), \overrightarrow{\mathit{data}}[i]) \\
\quad \texttt{in}\ F(\mathit{end})
\end{array}
$$

$$
\mathit{reduce}(\texttt{op}, \overrightarrow{\mathit{data}}) \doteq \mathit{reduce\_range}(\texttt{op}, \overrightarrow{\mathit{data}}, 0, \texttt{size}(\overrightarrow{\mathit{data}}))
$$

$$
\begin{array}{l}
\texttt{scan}_{\texttt{init}}(\mathit{buf}, v, \mathit{op}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad \texttt{syn}_{\texttt{write}}(\mathit{comm.cid}, v, p) \\
\texttt{scan}_{\texttt{wait}}(\mathit{buf}, v, \mathit{op}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad \texttt{syn}_{\texttt{wait}}(\mathit{comm.cid}, p) \wedge \\
\quad \texttt{let}\ \mathit{gr} = \mathit{comm.group}\ \texttt{in} \\
\quad \texttt{let}\ \mathit{data} = [i \in 0 \mathinner{..} \mathit{gr}|_p \mapsto \mathtt{rend}_p[\mathit{comm.cid}].\mathit{data}[\mathit{gr}[i]]] \\
\quad \texttt{in}\ \mathtt{mems}'_p[\mathit{buf}] = \mathit{reduce\_range}(\texttt{op}, \mathit{data}, 0, \mathit{gr}|_p)
\end{array}
$$

$$
\begin{array}{l}
\texttt{inter\_bcast}_{\texttt{init}}(\mathit{buf}, v, \mathit{root}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad (\mathit{comm.group}[\mathit{root}] = \mathtt{ROOT})\ ? \\
\qquad \texttt{syn\_put}(\mathit{comm.cid}, v, \epsilon, p)\ :\ \texttt{syn}_{\texttt{init}}(\mathit{comm.cid}, p) \\
\texttt{inter\_bcast}_{\texttt{wait}}(\mathit{buf}, v, \mathit{root}, \mathit{comm}, p) \stackrel{\circ}{=} \\
\quad \texttt{if}\ \mathit{root} \in \{\mathtt{PROC\_NULL}, \mathtt{ROOT}\}\ \wedge\ \neg \mathit{need\_syn}\ \texttt{then} \\
\qquad \texttt{asyn}_{\texttt{wait}}(\mathit{comm.cid}, p) \\
\quad \texttt{else}\ \texttt{syn}_{\texttt{wait}}(\mathit{comm.cid}, p) \wedge \\
\quad \mathtt{mems}'_p[\mathit{buf}] = \mathtt{rend}_p[\mathit{comm.cid}].\mathit{sdata}
\end{array}
$$

Fig. 8. Modeling collective communications

4.6.1 Group

A group defines the participants in the communication of a communicator. It is actually an ordered collection of processes, each with a rank. An ordered set containing n elements ranging from 0 to N can be modeled as a function:

$$[i \in 0 \mathinner{..} n-1 \rightarrow 0 \mathinner{..} N]$$

Given a group gr modeled as an ordered set, the rank of a process p in this group is given by $gr|_p$, and the process with rank i is by $gr[i]$.

The distinct concatenation of two ordered sets s_1 and s_2 is obtained by appending the elements in $s_2 \setminus s_1$ to s_1:

$$
\begin{array}{rl}
s_1 \boxplus s_2 \doteq & [i \in 0 \mathinner{..} (|s_1| + |s_2| - 1) \mapsto \\
& \quad i < |s_1|\ ?\ s_1[i]\ :\ s_2[i - |s_1|]].
\end{array}
$$

The difference and the union of two ordered sets are given by

$$
\begin{array}{l}
s_1 \ominus s_2 \doteq \\
\quad \texttt{let}\ F(i \in 0..|s_1|) = \\
\qquad (i = 0)\ ?\ \langle\rangle\ :\ (s_1[i-1] \notin s_2)\quad ?\ F(i-1) \boxplus \langle s_1[i-1]\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad :\ F(i-1) \\
\quad \mathtt{IN}\ F[|s_1|] \\
s_1 \odot s_2 \doteq \\
\quad \texttt{let}\ F(i \in 0..|s_1|) = \\
\qquad (i = 0)\ ?\ \langle\rangle\ :\ (s_1[i-1] \in s_2)\quad ?\ F(i-1) \boxplus \langle s_1[i-1]\rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad :\ F(i-1) \\
\quad \mathtt{IN}\ F[|s_1|] \\
s_1 \oplus s_2 \doteq s_1\ \boxplus\ (s_2 \ominus s_1)
\end{array}
$$

Function $\mathtt{incl}(s, n, \mathit{ranks})$ creates an ordered set that consists of the n elements in s with ranks $\mathit{ranks}[0], \ldots, \mathit{ranks}[n-1]$; $\mathtt{excl}$ creates an ordered set that is obtained by deleting from s those elements with ranks $\mathit{ranks}[0], \ldots, \mathit{ranks}[n-1]$; $\mathtt{range\_incl}$ ($\mathtt{range\_excl}$) accepts a ranges argument of form $(\mathit{first\ rank}, \mathit{last\ rank}, \mathit{stride})$ indicating ranks in s to be included (excluded) in the new ordered set.

$$\mathit{incl}(s, n, \mathit{ranks}) \doteq [i \in 0 \mathinner{..} n-1 \mapsto s[\mathit{ranks}[i]]]$$

$$\mathit{excl}(s, n, \mathit{ranks}) \doteq s \ominus (\mathit{incl}(s, n, \mathit{ranks}))$$

$$
\begin{array}{l}
\mathit{range\_incl}(s, n, \mathit{ranges}) \doteq \\
\texttt{let}\ \mathit{flatten}(\mathit{first}, \mathit{last}, \mathit{stride}) = \\
\quad \texttt{if}\ \mathit{last} < \mathit{first}\ \texttt{then}\ \langle\rangle \\
\quad \texttt{else}\ \mathit{first} \diamond \mathit{flatten}(\mathit{first} + \mathit{stride}, \mathit{last}, \mathit{stride}) \\
\texttt{in} \\
\texttt{let}\ F(i) = \\
\quad \texttt{if}\ i = 0\ \texttt{then}\ \langle\rangle \\
\quad \texttt{else}\ \texttt{let}\ \mathit{ranks} = \mathit{flatten}(\mathit{ranges}[i-1]) \\
\qquad\quad \texttt{in}\ F(i-1) \diamond \mathit{incl}(s, \texttt{size}(\mathit{ranks}), \mathit{ranks}) \\
\texttt{in}\ F(n)
\end{array}
$$

$$\mathit{range\_excl}(s, n, \mathit{ranges}) \doteq s \ominus (\mathit{range\_incl}(s, n, \mathit{ranges}))$$

For example, suppose $s_1 = \langle a, b, c, d\rangle$ and $s_2 = \langle d, a, e\rangle$, then $s_1 \oplus s_2 = \langle a, b, c, d, e\rangle$,

$s_1 \odot s_2 = \langle a, d\rangle$, and $s_1 \ominus s_2 = \langle b, c\rangle$. Suppose $s = \langle a, b, c, d, e, f, g, h, i, j\rangle$ and $ranges = \langle\langle 6, 7, 1\rangle, \langle 1, 6, 2\rangle, \langle 0, 9, 4\rangle\rangle$, then $range\_incl(s, 3, ranges) = \langle g, h, b, d, f, a, e, i\rangle$ and $range\_excl(s, 3, ranges) = \langle c, j\rangle$.

Since most group operations are local and their execution do not require interprocess communication, in the transition relations corresponding to such operations, only the `groups` object at the calling process is modified. For example, the transition implementing the union of two groups is as follows.

$$\begin{array}{l}\texttt{MPI\_Group\_union}(group_1, group_2, group_{new}, p) \doteq \\ \textbf{let } gid = \texttt{unused\_item}(\texttt{groups}_p) \textbf{ in} \\ \texttt{groups}'_p = \texttt{groups}_p \uplus \langle gid, \\ \quad \texttt{groups}_p[group_1] \oplus \texttt{groups}_p[group_2]\rangle \wedge \\ group'_{new} = gid\end{array}$$

4.6.2 Communicator Operations

Communicator constructors and destructors are collective functions that are invoked by all processes in the involved group. When a new communicator is created, each participanting process first invokes the "synchronization initialization" primitive (mentioned in the section 4.5) to express its willing to join the creation; then it calls the "synchronization wait" primitive to wait for the joining of all other processes; finally it creates the local version of the new communicator and store it in its `comms` object.

Communicators may be attached with arbitrary pieces of information (called attributes). When a attribute key is allocated (*e.g.* by calling the `MPI_Comm_create_keyval`) and stored in the `keyvals` object, it is attached with a copy callback function, a delete callback function and an extra state for callback functions. When a communicator is created using functions like `MPI_Comm_dup`, all callback copy functions for attributes are invoked (in arbitrary order). When the copy function returns $flag = \perp$, then the attribute is deleted in the created communicator; otherwise the new attribute value is set to the value returned in $attribute\_val\_out$.

The `MPI_Comm_dup` code shown in figure 9 creates a new intracommunicator with the same group and topology as the input intracommunicator. The association of cached attributes is controlled by the copy callback functions. As the new communicator must have a unique context id, the the process with rank 0 picks an unused context id, write it to the shared area of the rendezvous, and registers it in the system. In the "synchronization wait" phase each process fetches the unique context id, finds a place for the new communicator in its `comms` object, and updates the reference to this place.

Intercommunicator operations are a little more complicated. For example, `Intercomm_merge` creates an intracommunicator from the union of the two groups of a intercommunicator. All processes should provide the same $high$ value within each of the two groups. The group providing the value $high = \top$ should be ordered before the one providing $high = \perp$; and the order is arbitrary if all processes provide the same $high$ argument.

The TLA+ specification of communicator operations is more detailed, where we need to: (i) check whether all processes propose the same `group` and the group is a subset of the group associated with the old communicator; (ii) have the function returns `MPI_COMM_NULL` to processes that are not in the `group`; (iii) call the error callback functions when errors occur.

4.6.3 Topology

A topology can provide a convenient naming mechanism for the processes within a communicator, and additionally, may assist the runtime system in mapping the processes onto hardware. A topology can be represented by a graph, with nodes and edges standing for processes and communication links respectively. In some cases it is desirable to use Cartesian topologies (of arbitrary dimensions).

Primitive `Cart_create` builds a new communicator with Cartesian topology information. Arguments $ndims$ and $dims$ give the number of dimensions and an integer array specifying the number of processes in each dimension respectively. $periods$ specifies whether the grid is periodic or not in each dimension; and $reorder$ specifies whether ranks may be reordered or not. If the total size of the grid is smaller than the size of the group of $comm$, then those processes not fitting into the grid are returned `MPI_COMM_NULL`. Here the helper function $range\_product(ndims, dims, i, j)$ computes the value of $dims[i] \times \cdots \times dims[j]$.

Function $coord\_2\_rank$ translates the logical process coordinates to process ranks; function $rank\_2\_coord$ is the rank-to-coordinates translator. They are used to implemented the `MPI_Cart_rank` and `MPI_Cart_coords` primitives.

For further illustration we give the code of `MPI_Cart_shift`. When a `MPI_Sendrecv` operation is called along a coordinate direction to perform a shift of data, the rank of a source process for the receive and the rank of a destination process for the send can be calculated by this `MPI_Cart_shift` function. The dir argument indicates the dimension of the shift. In the case of an end-off shift, out-of-range processes will be returned the value `MPI_PROC_NULL`. Clearly `MPI_Cart_shift` is not a collective function.

4.7 Process Management

The MPI-2 process model allows for the creation and cooperative termination of processes after an MPI application has started. Since the runtime environment involving process creation and termination is not modeled, we do not specify `MPI_Comm_spawn`, which starts multiple copies of an MPI program specification, `MPI_Comm_spawn_multiple`, which starts multiple executable specifications, and `MPI_Comm_get_parent`, which is related to the "spawn" primitives.

Some functions are provided to establish communication between two groups of MPI processes that do not share a communicator. One group of processes (the *server*) indicates its willingness to accept connections from other groups of processes; the other group (the *client*) connects to the server. In order to the client to locate the server, the server provides a *port_name* that encodes a low-level network address. In our specification it consists of a process id and a port number. A server can publish a port_name with `MPI_Publish_name` and clients can retrieve the port name from the service name.

`Data Structures`
$\quad communicator:$
$\qquad cid: \texttt{int},\ group: \texttt{oset},\ remote\_group: \texttt{oset},$
$\qquad topology,\ attributes: \texttt{map}$

$create\_comm(comm, keyvals) \doteq$
$\texttt{let}\ copy\_attr(comm, attr, keyvals) =$
$\quad \texttt{let}\ keyval = keyvals[attr.key]\ \texttt{in}$
$\quad \texttt{let}\ y = keyval.copy\_attr\_fn(comm,$
$\qquad\qquad attr.key, keyval.extra\_state, attr.value)\ \texttt{in}$
$\quad [comm\ \mathtt{EXCEPT}\ !.attributes =$
$\qquad \texttt{if}\ y.flag = \perp\ \texttt{then}\ \texttt{remove}(@, attr.key)$
$\qquad \texttt{else}\ @ \uplus \langle attr.key, y.attribute\_val\_out\rangle$
$\quad]\ \texttt{in}$
$\texttt{let}\ traverse(T) =$
$\quad \texttt{if}\ T = \{\}\ \texttt{then}\ comm$
$\quad \texttt{else choose}\ attr \in T:$
$\qquad\quad copy\_attr(traverse(T \setminus \{attr\}), attr, keyvals)\ \texttt{in}$
$\texttt{if}\ attributes \notin \mathrm{DOM}(comm)\ \texttt{then}\ comm$
$\texttt{else}\ traverse(comm.attributes)$

$\texttt{comm\_dup}_{\texttt{init}}(comm, newcomm, p) \mathrel{\hat{=}}$
$\texttt{let}\ cid = \texttt{next\_comm\_cid in}$
$\texttt{if}\ comm.gr|_p = 0\ \texttt{then}$
$\quad \texttt{syn\_put}(comm, cid, \epsilon, p)\ \wedge\ \texttt{register\_cid}(cid)$
$\texttt{else syn\_init(comm, p)}$

$\texttt{comm\_dup}_{\texttt{wait}}(comm, newcomm, p) \mathrel{\hat{=}}$
$\texttt{syn\_wait}(comm, p) \wedge$
$\texttt{let}\ slot \diamond \Gamma = \mathtt{rend}[comm.cid]\ \texttt{in}$
$\texttt{let}\ cid = slot.sdata\ \texttt{in}$
$\texttt{let}\ new\_index = \texttt{unused\_item}(\mathtt{comms}_p)\ \texttt{in}$
$\mathtt{comms}'_p = \mathtt{comms}_p \uplus \langle new\_index,$
$\qquad [create\_comm(comm, keyvals_p)\ \mathtt{EXCEPT}\ !.cid = cid]\rangle \wedge$
$newcomm' = new\_index$

$\texttt{intercomm\_merge}_{\texttt{init}}(intercomm, high, intracomm_{new}, p) \mathrel{\hat{=}}$
$\texttt{let}\ cid = \texttt{next\_comm\_cid in}$
$\texttt{if}\ comm.gr|_p = 0\ \texttt{then}$
$\quad \texttt{syn\_put}(intercomm, cid, high, p)\ \wedge\ \texttt{register\_cid}(cid)$
$\texttt{else syn\_write(intercomm, high, p)}$

$\texttt{intercomm\_merge}_{\texttt{wait}}(intercomm, high, intracomm_{new}, p) \mathrel{\hat{=}}$
$\texttt{syn\_wait}(intercomm, p) \wedge$
$\texttt{let}\ slot \diamond \Gamma = \mathtt{rend}[intercomm.cid]\ \texttt{in}$
$\texttt{let}\ cid = slot.sdata\ \texttt{in}$
$\texttt{let}\ new\_index = \texttt{unused\_id}(\mathtt{comms}_p)\ \texttt{in}$
$\texttt{let}\ lr = intercomm.group \oplus intercomm.remote\_group\ \texttt{in}$
$\texttt{let}\ rl = intercomm.remote\_group \oplus intercomm.group\ \texttt{in}$
$\texttt{let}\ group =$
$\quad \texttt{if}\ \forall i, j \in intercomm.group \cup intercomm.remote\_group:$
$\qquad \mathtt{rend}[intercomm.cid].data[i] =$
$\qquad \mathtt{rend}[intercomm.cid].data[j]\ \texttt{then choose}\ gr \in \{lr, rl\}$
$\quad \texttt{else}\ high\ ?\ lr\ :\ rl$
$\mathtt{comms}'_p = \mathtt{comms}_p \uplus (new\_index,$
$\qquad [create\_comm(@, keyvals_p)\ \mathtt{EXCEPT}$
$\qquad\quad !.cid = cid,$
$\qquad\quad !.group = group$
$\qquad\quad !.remote\_group = \epsilon$
$\qquad]) \wedge$
$intercomm'_{new} = new\_index$

Fig. 9. Modeling communicator operations

```
Data Structures
  Cartesian topology :
    ndims : int, dims : int array, periods : bool array
  coordinate : int array

range_product(ndims, dims, i, j) ≐
let F(k) = k > j ? 1 : dims[k] * F(k + 1) in F(i)

cart_create_init(comm, ndims, dims, periods, reorder,
    comm_cart, p) ≗
let cid = next_comm_cid in
if comm.gr|_p = 0 then
  syn_put(comm, cid, ε, p) ∧ register_cid(cid)
else syn_init(comm, p)

cart_create_wait(comm, ndims, dims, periods, reorder,
    comm_cart, p) ≗
syn_wait(comm, p) ∧
let slot ◇ Γ = rend[comm.cid] in
let cid = slot.sdata in
let new_index = unused_item(comms_p) in
let comm_new =
  if proc ≤ range_product(ndims, dims, 0, ndims − 1)
  then COMM_NULL
  else
  [create_comm(comm_old, keyvals_p) EXCEPT
    !.cid = cid,
    !.group = reorder ? permute(@) : @
  ] ⊎
  ⟨topology,
    [ndims ↦ ndims, dims ↦ dims, periods ↦ periods]⟩
in comms'_p = comms_p ⊎ ⟨new_index, comm_new⟩ ∧
   comm_cart' = new_index

coord_2_rank(coord, ndims, dims) ≐
let F(n) = if n = size(coord) then 0
  else range_product(ndims, dims, n + 1, ndims − 1)×
         coord[n] + F(n + 1)
in F(0)

rank_2_coord(rank, ndims, dims) ≐
let F(x, n) = if n = 0 then ⟨x⟩
  else F(x ÷ dims[n], n − 1) ◇ (x % dims[n])
in F(rank, ndims − 1)

cart_shift(comm, dir, disp, p) ≐
let tp = comm.topology in
let ⟨dims, ndims⟩ = ⟨tp.dims, tp.ndims⟩ in
let rank = comm.group|_p in
let coord = rank_2_coord(rank, ndims, dims) in
let f(i) =
  if ¬tp.periods[rank] ∧ (i ≤ dims[dir] ∨ i < 0)
  then PROC_NULL
  else coord_2_rank([coord EXCEPT ![dir] = i],
                    ndims, dims)
in [rank_source ↦ f((@ − disp) % dims[dir]),
    rank_dest ↦ f((@ + disp) % dims[dir])
    ]
```

Fig. 10. Modeling topology operations

A server first calls `MPI_Open_port` to establish a port at which it may be contacted; then it calls `MPI_Comm_accept` to accept connections from clients. This port name may be reused after it is freed with `MPI_Close_port`. All published names must be unpublished before the corresponding port is closed.

Call `MPI_Comm_accept` is collective over the calling communicator. It returns an intercommunicator that allows communication with the client. In the "init" phase, the root process sets the port's client group to be its group. In the "wait" phase, each process creates a new intercommunicator with the local (remote) group being the server (client) group of the port. Furthermore, the root process sets the port's status to be 'waiting' so that new connection requests from clients can be accepted.

Call `MPI_Comm_connect` establishes communication with a server specified by a port name. It is collective over the calling communicator and returns an intercommunicator in which the remote group participated in an `MPI_Comm_accept`. We do not model the time-out mechanism; instead, we assume the time out period is infinitely long (thus will lead to deadlock if there is no matching `MPI_Comm_accept`). As shown in the code, the root process picks a new context id in its "init" phase. In the "wait" phase, each process creates a new intercommunicator; and the root process updates the port so that the server can proceed to create intercommunicators.

4.8 One-sided Communication

Remote Memory Access (RMA) allows one process to specify all communication parameters, both for the sending side and for the receiving side. This mechanism separates the communication of data from the synchronizations.

A process exposes a "window" of its memory accessible by remote processes. The *wins* object represents the group of processes that own and access the set of windows they expose. The management of this object, *e.g.* the creation and destroying of a window, is similar to that of the communicator object *comms* except that window operations are synchronizing.

Data Structures

$port$:
$\langle name : \langle proc : \mathtt{int}, port : \mathtt{int}\rangle,\ cid : \mathtt{int},$
$status : \{\text{'}connected\text{'}, \text{'}waiting\text{'}\},$
$server\_group : \mathtt{oset},\ client\_group : \mathtt{oset}\rangle$

$\mathtt{open\_port}(port\_name, p) \doteq$
$\mathtt{let}\ new\_port\_id = \mathtt{unused\_item}(\mathtt{ports}_p)\ \mathtt{in}$
$\mathtt{let}\ new\_port = [name \mapsto \langle p, new\_port\_id\rangle, status \mapsto \text{'}waiting\text{'}]\ \mathtt{in}$
$\mathtt{ports}'_p = \mathtt{ports}_p \uplus [new\_port\_id \mapsto new\_port] \wedge$
$port\_name' = new\_port.name$

$\mathtt{close\_port}(port\_name, p) \doteq$
$\mathtt{requires}\{port\_name \notin \mathtt{service\_names}\}$
$\mathtt{ports}'_p = \mathtt{remove}(\mathtt{ports}_p, port\_name.port)$

$\mathtt{comm\_accept}_{\mathtt{init}}(port\_name, root, comm, newcomm, p) \stackrel{\circ}{=}$
$\mathtt{let}\ port\_no = port\_name.port\ \mathtt{in}$
$\mathtt{if}\ comm.gr|_p = root\ \mathtt{then}$
$$\frac{\mathtt{ports}_p[port\_no].status = \text{'}waiting\text{'} \wedge \mathtt{syn}_{\mathtt{put}}(comm, port\_no, \epsilon, p)}{\mathtt{ports}'_p[port\_no] = [\mathtt{ports}_p[port\_no]\ \mathtt{EXCEPT}\ !.server\_group = comm.group]}$$
$\mathtt{else}\ \mathtt{syn}_{\mathtt{init}}(\mathtt{comm}, \mathtt{p})$

$\mathtt{comm\_accept}_{\mathtt{wait}}(port\_name, root, comm, newcomm, p) \stackrel{\circ}{=}$
$$\frac{\mathtt{syn}_{wait}(comm, p)}{\begin{array}{l}\mathtt{let}\ port\_no = \mathtt{rend}_p[cid].sdata\ \mathtt{in}\\ \mathtt{let}\ port = \mathtt{ports}_{comm.group[root]}[port\_no]\ \mathtt{in}\\ \dfrac{port.status = \text{'}connected\text{'}}{\begin{array}{l}\mathtt{comms}'_p[newcomm] = \\ \quad [cid \mapsto port.cid,\\ \quad group \mapsto port.server\_group,\\ \quad remote\_group \mapsto port.client\_group] \wedge\\ (p = comm.group[root]) \Rightarrow\\ \quad \mathtt{ports}'_p[port\_no].status = \text{'}waiting\text{'}\end{array}}\end{array}}$$

$\mathtt{comm\_connect}_{\mathtt{init}}(port\_name, root, comm, newcomm, p) \stackrel{\circ}{=}$
$\mathtt{let}\ port = \mathtt{ports}_{port\_name.proc}[port\_name.port]\ \mathtt{in}$
$\mathtt{let}\ cid = \mathtt{next\_comm\_cid}\mathtt{in}$
$\mathtt{if}\ comm.gr|_p = root\ \mathtt{then}$
$$\frac{port.status = \text{'}waiting\text{'} \ \wedge\ \mathtt{syn}_{\mathtt{put}}(comm, cid, \epsilon, p)}{\mathtt{register\_cid}(cid)}$$
$\mathtt{else}\ \mathtt{syn}_{\mathtt{init}}(\mathtt{comm}, \mathtt{p})$

$\mathtt{comm\_connect}_{\mathtt{wait}}(port\_name, root, comm, newcomm, p) \stackrel{\circ}{=}$
$$\frac{\mathtt{syn}_{wait}(comm, p)}{\begin{array}{l}\mathtt{let}\ cid = \mathtt{rend}_p[comm.cid].sdata\ \mathtt{in}\\ \mathtt{let}\ port = \mathtt{ports}_{comm.group[root]}[port\_no]\ \mathtt{in}\\ \mathtt{let}\ \langle host, port\_no\rangle = \langle port\_name.proc, port\_name.port\rangle\ \mathtt{in}\\ \mathtt{comms}'_p[newcomm] = \\ \quad [cid \mapsto cid,\\ \quad group \mapsto comm.group,\\ \quad remote\_group \mapsto \mathtt{ports}_{host}[port\_no].server\_group] \wedge\\ (p = comm.group[root]) \Rightarrow\\ \quad \mathtt{ports}'_p[port\_no].status = \text{'}connected\text{'} \wedge\\ \quad \mathtt{ports}'_p[port\_no].client\_group = comm.group \wedge\\ \quad \mathtt{ports}'_p[port\_no].cid = cid\end{array}}$$

Fig. 11. Modeling client-server communications

RMA communication calls associated with a window occur at a process only within an epoch for this window. Such an epoch starts with a RMA synchronization call, proceeds with some RMA communication calls (`MPI_Put`, `MPI_Get` and `MPI_Accumulate`), and completes with another synchronization call. RMA communications fall in two categories: *active target* communication, where both the origin and target processes involve in the communication, and *passive target* communication, where only the origin process involves in the communication. We model active (passive) target communication with the `eps` (`locks`) object.

`MPI_Win_start` and `MPI_Win_complete` start and complete an access epoch (with $mode = ac$) respectively; while `MPI_Win_post` and `MPI_Win_wait` start and complete an exposure epoch (with $mode = ex$) respectively. There is one-to-one matching between access epoches at origin processes and exposure epoches on target processes. Distinct access epoches for a window at the same process must be disjoint; so must distinct exposure epoches. In a typical communication, the target process first calls `MPI_Win_post` to start an exposure epoch, then the origin process calls `MPI_Win_start` to starts an access epoch, and then after some RMA communications it calls `MPI_Win_complete` to complete this access epoch, finally the target process calls `MPI_Win_wait` to complete the exposure epoch. This `MPI_Win_post` call will block until all matching class to `MPI_Win_complete` have occured. Both `MPI_Win_complete` and `MPI_Win_wait` enforce completion of all preceding RMA calls. If `MPI_Win_start` is blocking, then the corresponding `MPI_Win_post` must have executed. However, these calls may be nonblocking and complete ahead of the completion of others.

A process p maintains in $\mathtt{eps}_p$ a queue of epoches. Each epoch contains a sequence of RMA communications yet to be completed. Its $match$ field contains a set of $\langle matching\ process,\ matching\ epoch\rangle$ tuples, each of which points to a matching epoch at another process. An epoch becomes inactive when it is completed. When a new epoch ep is created and appended to the end of the epoch queue, this matching information is updated by calling the helper function $find\_match$, which locates at a process the first active epoch that has not be matched with ep. Additionally, since `MPI_Win_start` can be non-blocking such that it may complete before `MPI_Win_post` is issued, `MPI_Win_post` needs to update the matching information each time it is called. We do not remove completed epoches because their status may be needed by other processes to perform synchronization.

Designed for passive target communication, `MPI_Win_lock` and `MPI_Win_unlock` start and complete an access epoch repsectively. They are similar to those for active target communication, except that no corresponding exposure epoches are needed. Accesses that are protected by an exclusive lock will not be concurrent with other accesses to the same window. We maintain these epoches in a different object `locks`, which resides in the `envs` object in our specification.

RMA communication call `MPI_Put` transfers data from the caller memory to the target memory; `MPI_Put` transfers data from the target memory to the caller memory; and `MPI_Accumulate` updates locations in the target memory. When each of these calls is issued, it is appended to the current active access epoch which may be in the `eps` or `locks` object. Note that there is at most one active access epoch for a window at each process. The calls in an epoch is performed in a FIFO manner. When a call completes, it is removed from the queue.

The `active_transfer` rule performs data transferring: when the corresponding exposure epoch exists, the first RMA communication call in the current active epoch is carried out and the value v will be written (or reduced) to the memory of the destination. The rule for passive target communication is analogous.

4.9 I/O

MPI provides routines for transferring data to or from files on an external storage device. An MPI file is an ordered collection of typed data items. It is opened collectively by a group of processes. All subsequent collective I/O operations on the file are collective over this group.

MPI supports blocking and nonblocking I/O routines. As usual, we model a blocking call by a nonblocking one followed by a wait call such as `MPI_Wait`. In addition to normal collective routines (*e.g.* `MPI_File_read_all`), MPI provides *split collective* data access routines each of which is split into a begin routine and an end routine. Thus two rounds of synchronizations are needed for a collective I/O communication to complete. This is analogous to our splitting the collective communications into an "init" phase and a "wait" phase.

Since at each process each file handle may have at most one active split collective operation, the `frend` object, which represents the place where processes rendezvous, stores the information of one operation rather than a queue of operations for each file.

With respect to this fact, we design a protocol shown below to implement collective I/O communications: in the first "begin" phase, process p will proceed to its "end" phase provided that it has not participated in the current synchronization (say `syn`) and `syn`'s status is '$entering$'. Note that if all expected processes have participated then `syn`'s status will advance to '$leaving$'. In the "end" phase, p is blocked if `syn` is not in leaving status or p has left. The last leaving process will delete the `syn`. Here notation τ and Ψ return the status and the participants of a synchronization respectively.

p_0	p_1	p_2
$\texttt{win\_start}(group = \langle 1,2\rangle, win_0)$	$\texttt{win\_post}(group = \langle 0\rangle, win_0)$	$\texttt{win\_post}(group = \langle 0\rangle, win_0)$
$\texttt{put}(origin = 0, target = 1, win_0)$	$\texttt{win\_wait}(win_0)$	$\texttt{win\_wait}(win_0)$
$\texttt{get}(origin = 0, target = 2, win_0)$		
$\texttt{win\_complete}(win_0)$		

step	*event*	$\texttt{eps}_0$	$\texttt{eps}_1$	$\texttt{eps}_2$
1	$\texttt{win\_post}(\langle 0\rangle, win_0, 1)$		$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\}\rangle_0^{ex}$	
2	$\texttt{win\_post}(\langle 0\rangle, win_0, 2)$		$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\}\rangle_0^{ex}$
3	$\texttt{win\_start}(\langle 1,2\rangle, win_0, 0)$	$\langle 0, \langle 1,2\rangle, \langle\rangle, \top, \{\langle 1,0\rangle, \langle 2,0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$
4	$\texttt{put}(0, 1, win_0, 0)$	$\langle 0, \langle 1,2\rangle, \langle\langle 0,1\rangle^{put}\rangle, \top, \{\langle 1,0\rangle, \langle 2,0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$
5	$\texttt{get}(0, 2, win_0, 0)$	$\langle 0, \langle 1,2\rangle, \langle\langle 0,1\rangle^{put} \diamond \langle 0,1\rangle^{get}\rangle,$ $\top, \{\langle 1,0\rangle, \langle 2,0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$
6	$\texttt{active\_transfer}(0)$	$\langle 0, \langle 1,2\rangle, \langle\langle 0,2\rangle^{get}\rangle, \top, \{\langle 1,0\rangle, \langle 2,0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$
7	$\texttt{win\_complete}(win_0, 0)$	$\langle 0, \langle 1,2\rangle, \langle\rangle, \bot, \{\langle 1,0\rangle, \langle 2,0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$
8	$\texttt{win\_wait}(win_0, 2)$	$\langle 0, \langle 1,2\rangle, \langle\rangle, \bot, \{\langle 1,0\rangle, \langle 2,0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \bot, \{\langle 0,0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \top, \{\langle 0,0\rangle\}\rangle_0^{ex}$
9	$\texttt{win\_wait}(win_0, 1)$	$\langle 0, \langle 1,2\rangle, \langle\rangle, \bot, \{\langle 1,0\rangle, \langle 2,0\rangle\}\rangle_0^{ac}$	$\langle 0, \langle 0\rangle, \langle\rangle, \bot, \{\langle 0,0\rangle\}\rangle_0^{ex}$	$\langle 0, \langle 0\rangle, \langle\rangle, \bot, \{\langle 0,0\rangle\}\rangle_0^{ex}$

Fig. 12. An active target communication example. The execution shows a case of strong synchronization in the window win_0's with wid 0. Process p_0 creates an access epoch, p_1 and p_2 creates an exposure epoch respectively. An epoch becomes inactive after it completes. For brevity we omit the value in a RMA operation. The execution follows from the semantics shown in figure 13.

Data Structures

$\texttt{frend}$ for each file :
$\langle status(\tau) : \{`entering'(`e'),\ `leaving'(`l')\},$
$participants(\psi) : \texttt{int set},$
$[shared\_data], [data : \langle proc : \texttt{int}, data\rangle\ \texttt{set}]\rangle$

process p joins the synchronization

$\texttt{file}_{\texttt{put}}(\texttt{fid}, \texttt{v}_\texttt{s}, \texttt{v}, \texttt{p}) \doteq$
$\texttt{if}\ fid \notin \texttt{DOM}\ \texttt{frend}\ \texttt{then}$
$\quad \texttt{frend}'[fid] = \langle `e`, \{p\}, v_s, \{\langle p, v\rangle\}\rangle$
$\texttt{else}$

$$\frac{\texttt{frend}[fid] = \langle `e', \Psi, v_{s_1}, S_v\rangle\ \wedge\ p \notin \Psi}{\texttt{frend}'[fid] = \langle (\Psi \cup \{p\} = \texttt{files}_p[fid].group)\ ?\ `l`\ :\ `e`,\ \Psi \cup\ \{p\},\ v_s,\ S_v \cup \{\langle p, v\rangle\}\rangle}$$

$\texttt{file}_{\texttt{begin}}(\texttt{fid}, \texttt{p}) \doteq \texttt{file}_{\texttt{put}}(\texttt{fid}, \epsilon, \epsilon, \texttt{p})$
$\texttt{file}_{\texttt{write}}(\texttt{fid}, \texttt{v}, \texttt{p}) \doteq \texttt{file}_{\texttt{put}}(\texttt{fid}, \epsilon, \texttt{v}, \texttt{p})$

process p leaves the synchronization

$\texttt{file}_{\texttt{end}}(\texttt{fid}, \texttt{p}) \doteq$

$$\frac{\texttt{frend}[fid] = \langle `l', \Psi \cup \{p\}, v_s, S_v\rangle}{\texttt{frend}'[fid] = \texttt{if}\ \Psi = \{\}\ \texttt{then}\ \epsilon\ \ \texttt{else}\ \langle `l', \Psi,\ v_s,\ S_v\rangle}$$

We use the `files` object to store the file information, which includes an *individual* file pointer, which is local to a process, and a *shared* file pointer, which is shared by the group of processes that opened the file. These pointers are used to locate the positions in the file relative to the current view. A file is opened by the `MPI_File_open` call, which is collective over all participanting processes.

When a process p wants to access the file in the operating system `os.file`, it appends a read or write request to its request queue $\texttt{freqs}_p$. A request contains information about the offset in the file, the buffer address in the memory, the number of items to be read, and a flag indicating whether this request is active or not. The MPI system schedules the requests in the queue asynchronously, allowing the first active access to take effect at any time. After the access is finished, the request becomes inactive, and a subsequent wait call will return without being blocked. Note that we need to move the file pointers after the access to the file.

Analogous to usual collective communications, a split collective data access call is split into a begin phase and a end phase. For example, in the begin phase a collective read access reads the data from the file and stores the data in the `frend` object; then in the end phase it fetches the data and updates its own memory.

4.10 Evaluation

How to ensure that our formalization is faithful with the English description? To attack this problem we rely heavily on testing in our formal framework. We provide comprehensive

`Data Structures`

$epoch$:

$\langle wid : \texttt{int},\ group : \texttt{oset},\ rma : (RMA\ communication)\ \texttt{array}, active : \texttt{bool},\ match : \langle \texttt{int}, \texttt{int}\rangle\ \texttt{set}\rangle^{mode:\{ac,ex,fe\}}$

$lock : \langle wid : \texttt{int},\ RMA : (RMA\ communication)\ \texttt{array},\ active : \texttt{bool}\rangle^{type:\{\text{EXCLUSIVE},\text{SHARED}\}}$

$RMA\ communication : \langle source(src) : \texttt{int},\ destination(dst) : \texttt{int},\ value\rangle^{op:\{put,get,accumulate\}}$

match access epoches and exposure epoches

$find\_match(mode, group, p) \doteq$
$\{\langle q, \texttt{first}\ k\rangle \mid q \in group\ \wedge$
$\quad \texttt{eps}_q[k].mode = mode\ \wedge$
$\quad p \in \texttt{eps}_q[k].group\ \wedge\ \nexists\langle p, \alpha\rangle \in \texttt{eps}_q[k].match\}$

$\texttt{win\_post}(group, win, p) \mathrel{\hat=}$ start an exposrue epoch

$\texttt{requires}\ \{\nexists\langle win.wid, \alpha, \alpha, \top, \alpha\rangle^{ex} \in \texttt{eps}_p\}$
$\texttt{let}\ mt = find\_match(ac, group, p)\ \texttt{in}$
$\texttt{eps}'_p = eps_p \diamond \langle win.wid, group, \langle\rangle, \top, mt\rangle^{ex}\ \wedge$
$\forall q \in group : \exists\langle q, k\rangle \in mt \Rightarrow$
$\quad \texttt{eps}'_q[k].mt = \texttt{eps}_q[k].mt \cup \langle p, \texttt{len}(\texttt{eps}'_p)\rangle$

$\texttt{win\_start}(group, win, p) \mathrel{\hat=}$ start an access epoch

$\texttt{requires}\ \{\nexists\langle win.wid, \alpha, \alpha, \top\rangle^{ac} \in \texttt{eps}_p\}$
$\texttt{let}\ mt = find\_match(ex, group, p)\ \texttt{in}$
$\texttt{let}\ action =$
$\quad \texttt{eps}'_p = eps_p \diamond \langle win.wid, group, \langle\rangle, \top, mt\rangle^{ac}\ \wedge$
$\quad \forall q \in group : \exists\langle q, k\rangle \in mt \Rightarrow$
$\quad\quad \texttt{eps}'_q[k].mt = \texttt{eps}_q[k].mt \cup \langle p, \texttt{len}(\texttt{eps}'_p)\rangle$
$\texttt{in if}\ \neg is\_block\ \texttt{then}$
$\quad\quad action$
$\quad \texttt{else}$
$$\frac{\forall q \in group : \exists ep^{ex} \in \texttt{eps}_q : p \in ep.group}{action}$$

$\texttt{win\_complete}(win, p) \mathrel{\hat=}$ complete an access epoch

$\texttt{let}\ k = \texttt{first}\ i :$
$\quad \texttt{eps}_p[i].wid = win.wid\ \wedge$
$\quad \texttt{eps}_p[i].mode = ac\ \wedge\ \texttt{eps}_p[i].active$
$\texttt{in}$
$$\frac{\forall \texttt{eps}_p[k].rma = \langle\rangle}{\begin{array}{l}\texttt{if}\ \neg is\_block\ \texttt{then}\\ \quad \texttt{eps}'_p[k].active = \bot\\ \texttt{else}\\ \quad \dfrac{\texttt{size}(\texttt{eps}_p[k].match) = \texttt{size}(\texttt{eps}_p[k].group)}{\texttt{eps}'_p[k].active = \bot}\end{array}}$$

$\texttt{win\_wait}(win, p) \mathrel{\hat=}$ complete an exposure epoch

$\texttt{let}\ k = \texttt{first}\ i :$
$\quad \texttt{eps}_p[i].wid = win.wid\ \wedge$
$\quad \texttt{eps}_p[i].mode = ex\ \wedge\ \texttt{eps}_p[i].active$
$\texttt{in}$
$$\frac{\forall call \in \texttt{eps}_p[k] : \neg call.active\ \wedge\ \forall\langle q, i\rangle \in \texttt{eps}_p[k].match : \neg\texttt{eps}_q[i].active}{\texttt{eps}'_p[k].active = \bot}$$

$\texttt{RMA\_op}(type, origin, target, disp, v, op, win, p) \mathrel{\hat=}$
$\texttt{if}\ \exists k : \texttt{locks}_p[i].wid = win.wid\ \wedge$
$\quad\quad \texttt{locks}_p[i].active\ \texttt{then}$
$\quad \texttt{let}\ k = \texttt{first}\ i :$
$\quad\quad \texttt{locks}_p[i].wid = win.wid\ \wedge\ \texttt{locks}_p[i].active$
$\quad \texttt{in}\ \texttt{locks}'_p[k].rma =$
$\quad\quad \texttt{locks}_p[k].rma \diamond \langle origin, target, disp, v, op\rangle^{type}$
$\texttt{else}$
$\quad \texttt{let}\ k = \texttt{first}\ i :$
$\quad\quad \texttt{eps}_p[i].wid = win.wid\ \wedge$
$\quad\quad \texttt{eps}_p[i].mode = ac\ \wedge\ \texttt{eps}_p[i].active$
$\quad \texttt{in}\ \texttt{eps}'_p[k].rma =$
$\quad\quad \texttt{eps}_p[k].rma \diamond \langle origin, target, disp, v, op\rangle^{type}$

$\texttt{put}(origin, target, addr_{origin}, disp_{target}, win, p) \mathrel{\hat=}$
$\texttt{RMA\_op}(put, origin, target, disp_{target},$
$\quad read\_data(\texttt{mems}_p, addr_{addr}), win, p)$

perform active message passing origining at process p

$\texttt{active\_transfer}(p) \mathrel{\hat=}$
$\texttt{let}\ k = \texttt{first}\ i :$
$\quad \texttt{eps}_p[i].mode = ac\ \wedge\ \texttt{eps}_p[i].rma \neq \langle\rangle$
$\texttt{let}\ \langle src, dst, disp, v, op\rangle^{type} \diamond \Gamma = \texttt{eps}_p[k]\ \texttt{in}$
$\texttt{eps}'_p[k].rma = \Gamma\ \wedge$
$\texttt{if}\ type = get\ \texttt{then}$
$\quad \texttt{mems}'_p = write\_data(\texttt{mems}_p, win.base + disp, v)$
$\texttt{else if}\ type = put\ \texttt{then}$
$\quad \texttt{let}\ \langle q, \alpha\rangle = \texttt{eps}_p[k].match\ \texttt{in}$
$\quad \texttt{mems}'_q = write\_data(\texttt{mems}_q, win.base + disp, v)$
$\texttt{else}$
$\quad \texttt{let}\ \langle q, \alpha\rangle = \texttt{eps}_p[k].match\ \texttt{in}$
$\quad \texttt{mems}'_p = reduce\_data(\texttt{mems}_p, win.base + disp, v, op)$

start an access epoch for passive target communication

$\texttt{win\_lock}(lock\_type, dst, win, p) \mathrel{\hat=}$
$\texttt{requires}\ \{\nexists\langle win.wid, \alpha, \alpha, \top\rangle^{ex} \in \texttt{eps}_p\}$
$\texttt{if}\ lock\_type = \texttt{SHARED}\ \texttt{then}$
$\quad \texttt{locks}'_p = locks_p \diamond \langle win.wid, dst, \langle\rangle, \top\rangle^{lock\_type}$
$\texttt{else}$
$\quad \forall q \in win.group : \nexists k :$
$$\frac{\texttt{locks}_q[k].wid = win.wid\ \wedge\ \texttt{locks}_q[k].active}{\texttt{locks}'_p = locks_p \diamond \langle win.wid, dst, \langle\rangle, \top\rangle^{lock\_type}}$$

complete an access epoch for passive target communication

$\texttt{win\_unlock}(dst, win, p) \mathrel{\hat=}$
$\texttt{let}\ k = \texttt{first}\ i :$
$\quad \texttt{locks}_p[i].wid = win.wid\ \wedge\ \texttt{eps}_p[i].active$
$\texttt{in}$
$$\frac{\texttt{locks}_p[k].rma = \langle\rangle}{\texttt{locks}'_p[k].active = \bot}$$

Fig. 13. Modeling one-sided communications

Data Structures

file information at a process :

$fid : \texttt{int},\ group : \texttt{oset},\ fname : \texttt{string},$
$amode :\ mode\ \texttt{set},\ size : \texttt{int},\ view,$
$pointers(pts) : \langle p_{shared}, \texttt{int},\ p_{ind} : \texttt{int}\rangle$

file access request :

$\langle file(fh),\ offset : \texttt{int},\ buf : \texttt{int},$
$count : \texttt{int},\ active : \texttt{bool}\rangle$

nonblocking file access

$\texttt{iread}(fh, offset, buf, count, req, p) \doteq$
$\quad \texttt{freqs}'_p = \texttt{freqs}_p \diamond \langle fh, offset, buf, count, \top\rangle^{read}$
$\texttt{iwrite}(fh, offset, buf, count, req, p) \doteq$
$\quad \texttt{freqs}'_p = \texttt{freqs}_p \diamond \langle fh, offset, buf, count, \top\rangle^{write}$

$\texttt{file\_access}(p) \doteq$ perform file access asynchronously

$\texttt{let}\ \langle fh, offset, buf, count, \top\rangle^{mode} \diamond \Gamma = \texttt{freqs}_p\ \texttt{in}$
$\texttt{freqs}'_p = \langle fh, offset, buf, count, \bot\rangle^{mode} \diamond \Gamma\ \wedge$
$\texttt{if}\ mode = write\ \texttt{then}$
$\quad \texttt{let}\ v = read\_mem(\texttt{mems}_p, buf, count)\ \texttt{in}$
$\quad\quad \texttt{files}'_p[fh.fid].pts = move\_pointers(fh, v)\ \wedge$
$\quad\quad \texttt{os.file}'_p = write\_file(fh, \texttt{os.file}, v)$
$\texttt{else}$
$\quad \texttt{let}\ v = read\_file(fh, \texttt{os.file}_p, offset, count)\ \texttt{in}$
$\quad\quad \texttt{files}'_p[fh.fid].pts = move\_pointers(fh, v)\ \wedge$
$\quad\quad \texttt{mems}'_p = write\_mem(\texttt{mems}_p, buf, v)$

$\texttt{file\_wait}(req, p) \doteq$
$\texttt{let}\ \Gamma_1 \diamond \langle fh, offset, buf, count, \bot\rangle^{mode}_{req} \diamond \Gamma_2 = \texttt{freqs}_p$
$\texttt{in}\ \texttt{freqs}'_p = \Gamma_1 \diamond \Gamma_2$

the begin call of split collective file read operation

$\texttt{file\_read\_all}_{\texttt{begin}}(fh, offset, buf, count, p) \doteq$
$\quad \texttt{file}_{\texttt{write}}(fh.fid,$
$\quad\quad read\_file(fh, \texttt{os.file}_p, offset, count), p)$

the end call of split collective file read operation

$\texttt{file\_read\_all}_{\texttt{end}}(fh, buf, p) \doteq$
$\quad\quad \texttt{file}_{\texttt{end}}(fh.fid, p)$
$\quad \texttt{let}\ v = \texttt{frend}[fh.fid].data[p]\ \texttt{in}$
$\quad \texttt{files}'_p[fh.fid].pts = move\_pointers(fh, v)\ \wedge$
$\quad \texttt{mems}'_p = write\_mem(\texttt{mems}_p, buf, v)$

$\texttt{file\_write\_all}_{\texttt{begin}}(fh, buf, count, p) \doteq$
$\quad \texttt{file}_{\texttt{write}}(fh.fid, read\_mem(\texttt{mems}_p, buf, count), p)$

$\texttt{file\_write\_all}_{\texttt{end}}(fh, buf, p) \doteq$
$\quad\quad \texttt{file}_{\texttt{end}}(fh.fid, p)$
$\quad \texttt{let}\ v = \texttt{frend}[fh.fid].data[p]\ \texttt{in}$
$\quad \texttt{files}'_p[fh.fid].pts = move\_pointers(fh, v)\ \wedge$
$\quad \texttt{os.file}'_p = write\_file(fh, \texttt{os.file}_p, v)$

Fig. 14. Modeling I/O operations

unit tests and a rich set of short litmus tests of the specification. Generally it suffices to test Local, collective, and asynchronous MPI primitives on one, two and three processes respectively. These test cases, which include many simple examples in the MPI reference, are hand-written directly in TLA+ and modeled checked using TLC. As we have mentioned in section 3, thanks to the power of the TLC model checker our framework supports thorough testing of MPI programs, thus giving more precise answers than vendor MPI implementations can.

Another set of test cases are built to verify the *self-consistency* of the specification. For a communication (pattern), there may be many ways to express it. Thus it is possible to relate aspects of MPI to each other. Actually, in the MPI definition certain MPI functions are explained in terms of other MPI functions.

We introduce the notation $\texttt{MPI\_}A \simeq \texttt{MPI\_}B$ to indicate that A and B have the same functionality with respect to their semantics.

Our specification defines a blocking point-to-point operation by a corresponding non-blocking operation followed immediately by a `MPI_Wait` operation. Thus we have

$$\begin{aligned}
\texttt{MPI_Send(n)} &\simeq \texttt{MPI_Isend(n)} + \texttt{MPI_Wait}\\
\texttt{MPI_Recv(n)} &\simeq \texttt{MPI_Irecv(n)} + \texttt{MPI_Wait}\\
\texttt{MPI_Sendrecv}(\texttt{n}_1,\texttt{n}_2) &\simeq \texttt{MPI_Isend}(\texttt{n}_1) + \texttt{MPI_Irecv}(\texttt{n}_2)\\
&\quad + \texttt{MPI_Wait} + \texttt{MPI_Wait}
\end{aligned}$$

Typical relationships between the MPI communication routines, together with some examples, include:

- A message can be divided into multiple sub-messages sent separately.

$$\begin{aligned}
\texttt{MPI_A}(\texttt{k}\times\texttt{n}) &\simeq \texttt{MPI_A(n)}_1 + \cdots + \texttt{MPI_A(n)}_k\\
\texttt{MPI_A}(\texttt{k}\times\texttt{n}) &\simeq \texttt{MPI_A(k)}_1 + \cdots + \texttt{MPI_A(k)}_n
\end{aligned}$$

- A collective routine can be replaced by several point-to-point or one-sided routines.

$$\begin{aligned}
\texttt{MPI_Bcast(n)} &\simeq \texttt{MPI_Send(n)} + \cdots + \texttt{MPI_Send(n)}\\
\texttt{MPI_Gather(n)} &\simeq \texttt{MPI_Recv(n/p)}_1 + \cdots\\
&\quad + \texttt{MPI_Recv(n/p)}_p
\end{aligned}$$

- Communications using `MPI_Send`, `MPI_Recv` can be implemented by one-sided communications.

$\texttt{MPI\_Win\_fence} + \texttt{MPI\_Get(n)} + \texttt{MPI\_Win\_fence} \simeq$
$\texttt{MPI\_Barrier} + \texttt{MPI\_Recv(d)} + \texttt{MPI\_Recv(n)} +$
$\texttt{MPI\_Barrier},$
where d is the address and datatype information

- Process topologies do not affect the results of message passing. Communications using a communicator that implements a random topology should has the same semantics as the communication with a process topology (like a Cartesian topology).

Our specification is shown to meet the correctness requirements by model checking test cases.

4.11 Discussion

It is important to point out that we have not modeled all the details of the MPI standard. We list below the details that are skipped and the reasons why we do not model them:

- *Implementation details.* To the greatest extent possible we have avoided asserting implementation-specific details in our formal semantics. One obvious example is that the **info** object, which is one arguments of some MPI 2.0 functions, is ignored.
- *Physical Hardware.* The underlying, physical hardware is invisible in our model. Thus we do not model low-level topology functions such as `MPI_Cart_map` and `MPI_Graph_map`.
- *Profiling Interface.* The MPI profiling interface is to permit the implementation of profiling tools. It is irrelevant to the semantics of MPI functions.
- *Runtime Environment.* Since we do not model the operation system to allow for the dynamic process management (*e.g.* process creation and cooperative process termination), MPI routines accessing the runtime environment such as `MPI_Comm_spawn` are not modeled. Functions associated with the thread environment are not specified either.

Often our formal specifications end up resembling programs written using detailed data structures, *i.e.* they are not as "declarative" as possible. We believe that this is in some sense inevitable when attempting to obtain executable semantics of real world APIs. Even so, TLA+ based "programs" can be considered superior to executable models created in C: (i) the notation has a precise semantics, as opposed to C, (ii) another specification in a programming language can provide complementary details, (iii) in our experience, there are still plenty of short but tricky MPI programs that can be executed fast in our framework.

5 VERIFICATION FRAMEWORK

Our modeling framework uses the Microsoft Phoenix [14] Compiler as a front-end for C programs. Of course other front-end tools such as GCC can also be used. The phoenix framework allows developers to insert a compilation phase between existing compiler phases in the process of lowering a program from language independent MSIL (Microsoft Intermediate Language) to device specific assembly. We place our phase at the point where the input program has (i) been simplified into a single static assignment (SSA) form, with (ii) a homogenized pointer referencing style that is (iii) still device independent.

From Phoenix intermediate representation (IR) we build a state-transition system by converting the control flow graph into TLA+ relations and mapping MPI primitives to their names in TLA+. Specifically, control locations in the program are represented by states, and program statements are represented using transitions. Assignments are modeled by their effect on the memory. Jumps have standard transition rules modifying the values of the program counters. This transition system will completely capture the control skeleton of the input MPI program.

The architecture of the verification framework is shown in Figure 15. The user may input a program in any language that can be compiled using the Phoenix back-end — we have experimented only with C. The program is compiled into an intermediate representation, the Phoenix IR. We read the Phoenix IR to create a separate intermediate representation, which is used to produce TLA+ code. The TLC model checker integrated in our framework enables us to perform verification on the input C programs. If an error is found, the error trail is then made available to the verification environment, and can be used by our tool to drive the Visual Studio debugger to replay the

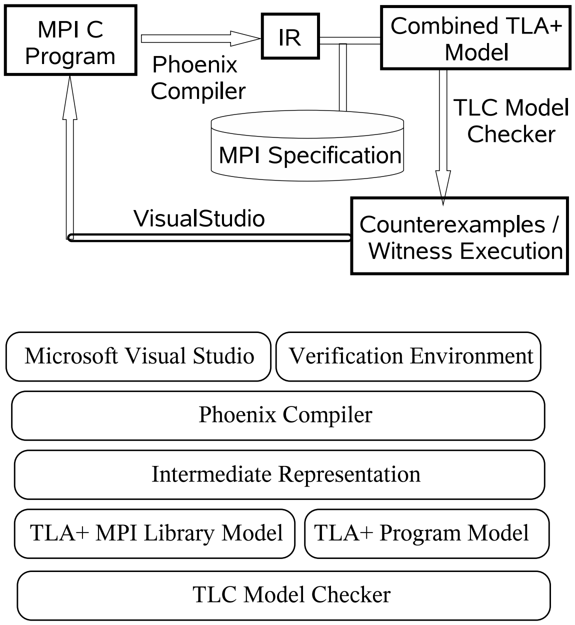


Fig. 15. Architecture of the verification framework. The upper (bottom) one indicates the flow (hierarchical) relation of the components.

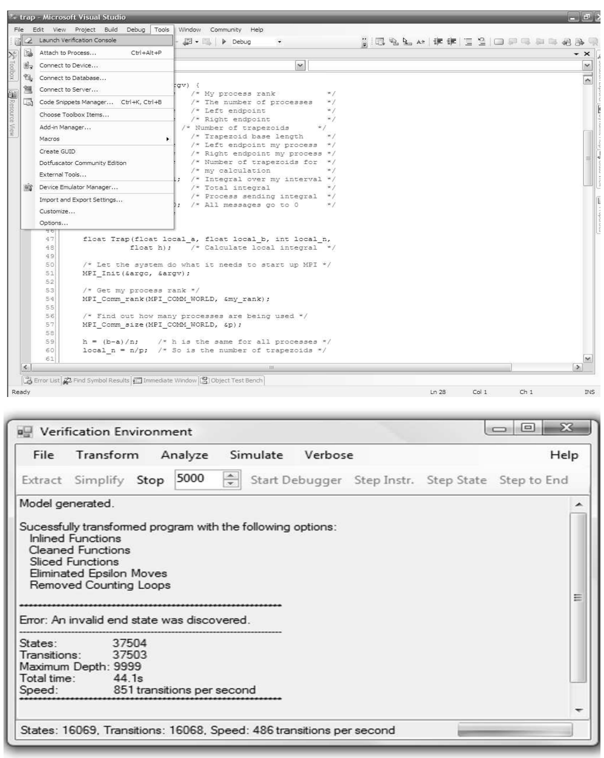


Fig. 16. Two screenshots of the verification framework.

trace to the error. In the following we describe the simplification, code generation and replay capabilities of our framework.

Simplification. In order to reduce the complexity of model checking, we perform a sequence of transformations: (i) inline all user defined functions (currently function pointers and recursion are not supported); (ii) remove operations foreign to the model checking framework, e.g. `printf`. (iii) slice the model with respect to communications and user assertions: the cone of influence of variables is computed using a chaotic iteration over the program graph, similar to what is described in [16]; (iv) eliminate redundant counting loops.

Code Generation. During the translation from Phoenix IR to TLA+, we build a record map to store all the variables in the intermediate language. The address of a variable x is given by the TLA+ expression $map.x$; and its value at the memory is returned by $mems[map.x]$. Before running the TLC, the initial values of all constants and variables are specified (*e.g.* in a configuration file). The format of the main transition relation is shown below, where N is the number of processes, and $predefined\_nxt$ is the "system" transition which performs message passing for point-to-point communications, one-sided communications, and so on. In addition, "program" transitions $transition_1, transition_2, \cdots$ are produced by translating MPI function calls and IR statements. In the examples shown later we only show the program transition part.

$$
\begin{array}{ll}
\vee & \wedge\, predefined\_nxt \\
 & \wedge\, \mathtt{UNCHANGED}\ \langle\langle map \rangle\rangle \\
\vee & \exists pid \in 0..(N-1) : \\
 & \vee\, transition_1 \\
 & \vee\, transition_2 \\
 & \vee \cdots \\
\vee & \forall pid \in 0..(N-1) : \\
 & \wedge\, pc[pid] = last\_label \\
 & \wedge\, \mathtt{UNCHANGED}\ all\_varaibles
\end{array}
$$

Error Trail Generation. In the event that the model contains an error, an error trail is produced by the model checker and returned to the verification environment. To map the error trail back onto the actual program we observe MPI function calls and the changes in the error trail to variable values that appear in the program text. For each change on a variable, we step the Visual Studio debugger until the corresponding value of the variable in the debugger matches. We also observe which process moves at every step in the error trail and context switch between processes in the debugger at corresponding points. When the error trail ends, the debugger is within a few steps of the error with the process that causes the error scheduled. The screenshots in figure 16 show the debugger interface and the report of an error trace.

Examples. A simple C program containing only one statement "if (rank == 0) MPI_Bcast (&b, 1, MPI_INT, 0, comm1) " is translated to:

$$
\begin{array}{ll}
\lor & \land\ pc[pid] = L_1\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_2] \\
 & \land\ mems' = [mems\ \mathtt{EXCEPT}\ ![pid] = \\
 & \qquad [@\ \mathtt{EXCEPT}\ ![map.t_1] = (mems[pid][map.\_rank] = 0)]] \\
\lor & \land\ pc[pid] = L_2\ \land\ mems[pid][map.t_1] \\
 & \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_3] \\
\lor & \land\ pc[pid] = L_2\ \land\ \neg(mems[pid][map.t_1]) \\
 & \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_5] \\
\lor & \land\ pc[pid] = L_3\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_4] \\
 & \land\ \text{MPI\_Bcast\_init}(map.\_b, 1, \text{MPI\_INT}, 0, map.\_comm1, pid) \\
\lor & \land\ pc[pid] = L_4\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_5] \\
 & \land\ \text{MPI\_Bcast\_wait}(map.\_b, 1, \text{MPI\_INT}, 0, map.\_comm1, pid)
\end{array}
$$

At label L_1, the value of $rank\ ==\ 0$ is assigned to a temporary variable t_1, and the pc advances to L_2. In the next step, if the value of t_1 is true, then the pc advances to L_3; otherwise to the exit label L_5. The broadcast is divided into an "init" phase (where pc advances from L_3 to $L-4$) and a "wait" phase (where pc advances from L_4 to $L-5$). Now consider a more complicated C program shown below

```
int main(int argc, char* argv[])
{
  int rank;
  int data;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    data = 10;
    MPI_Send(&data,1,MPI_INT,1,0,MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(&data,1,MPI_INT,0,0,MPI_COMM_WORLD,
             &status);
  }
  MPI_Finalize();
  return 0;
}
```

is translated to the following TLA+ code. When we run the TLC to demonstrate the absence of deadlocks for 2 processes, 51 distinct states are visited, and the depth of the complete state graph search is 17. The verification time is less than 0.1 second on a 3GHz processor with 1GB of memory. However, although it suffices in general to perform the test on a small number of processes, increasing the number of processes will increase the verification time exponentially. Thus we are implementing efficient methods such as partial order reduction algorithms [22][30] to reduce the state space.

$$
\begin{array}{ll}
\lor & \land\ pc[pid] = \_main\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_1] \\
 & \land\ \text{MPI\_Init}(map.\_argc, map.\_argv, pid) \\
\lor & \land\ pc[pid] = L_7\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_9] \\
 & \land\ mems' = [mems\ \mathtt{EXCEPT}\ ![pid] = Update(@, map.\_data, 10)] \\
 & \land\ changed(mems) \\
\lor & \land\ pc[pid] = L_6\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_{14}] \\
 & \land\ \text{MPI\_Irecv}(map.\_data, 1, \text{MPI\_INT}, 0, 0, \\
 & \qquad\qquad \text{MPI\_COMM\_WORLD}, map.tmprequest_1, pid) \\
\lor & \land\ pc[pid] = L_1\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_2] \\
 & \land\ \text{MPI\_Comm\_rank}(\text{MPI\_COMM\_WORLD}, map.\_rank, pid) \\
\lor & \land\ pc[pid] = L_2\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_5] \\
 & \land\ mems' = [mems\ \mathtt{EXCEPT}\ ![pid] = \\
 & \qquad Update(@, map.t_{277}, mems[pid][map.\_rank] = 0)] \\
 & \land\ changed(mems) \\
\lor & \land\ pc[pid] = L_5\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_7] \\
 & \land\ mems[pid][map.t_{277}] \\
\lor & \land\ pc[pid] = L_5\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_6] \\
 & \land\ \neg(mems[pid][map.t_{277}]) \\
\lor & \land\ pc[pid] = L_9\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_{13}] \\
 & \land\ \text{MPI\_Isend}(map._{d}ata, 1, \text{MPI\_INT}, 1, 0, \\
 & \qquad\qquad \text{MPI\_COMM\_WORLD}, map.tmprequest_0, pid) \\
\lor & \land\ pc[pid] = L_{11}\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_{12}] \\
 & \land\ \text{MPI\_Finalize}(pid) \\
\lor & \land\ pc[pid] = L_{13}\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_{11}] \\
 & \land\ \text{MPI\_Wait}(map.tmprequest_0, map.tmpstatus_0, pid) \\
\lor & \land\ pc[pid] = L_{14}\ \land\ pc' = [pc\ \mathtt{EXCEPT}\ ![pid] = L_{11}] \\
 & \land\ \text{MPI\_Wait}(map.tmprequest_1, map.\_status, pid)
\end{array}
$$

6 Conclusion

To help reason about programs that use MPI for communication, we have developed a formal TLA+ semantic definition of MPI 2.0 operations to augment the existing standard. We described this formal specification, as well as our framework to extract models from SPMD-style C programs. We discuss how the framework incorporates high level formal specifications, and yet allows designers to experiment with these specifications, using model checking, in a familiar debugging environment. Our effort has helped identify a few omissions and ambiguities in the original MPI reference standard document. The experience gained so far suggests that a formal semantic definition and exploration approach as described here must accompany every future effort in creating parallel and distributed programming libraries.

In future, we hope to write general theorems (inspired by our litmus tests), and establish them using the Isabelle theorem prover that has a tight TLA+ integration.

Appendix A

Give real TLA+ code of two MPI calls, one from point-to-point communcations (such as `MPI_Wait`) and one from collective communications.

...

References

[1] Martín Abadi, Leslie Lamport, and Stephan Merz, *A tla solution to the rpc-memory specification problem.*, Formal Systems Specification, 1994, pp. 21–66.

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

[3] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany, *Transformations to parallel codes for communication-computation overlap*, SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing (Washington, DC, USA), IEEE Computer Society, 2005, p. 58.

[4] P. Van Eijk and Michel Diaz (eds.), *Formal description technique lotos: Results of the esprit sedos project*, Elsevier Science Inc., New York, NY, USA, 1989.

[5] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall, *Open MPI: Goals, concept, and design of a next generation MPI implementation*, Proceedings, 11th European PVM/MPI Users' Group Meeting (Budapest, Hungary), September 2004, pp. 97–104.

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

[7] William Gropp, Ewing L. Lusk, Nathan E. Doss, and Anthony Skjellum, *A high-performance, portable implementation of the mpi message passing interface standard*, Parallel Computing **22** (1996), no. 6, 789–828.

[8] John Harrison, *Formal verification of square root algorithms*, Formal Methods in System Design **22** (2003), no. 2, 143–154, Guest Editors: Ganesh Gopalakrishnan and Warren Hunt, Jr.

[9] G. Holzmann, *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), no. 5, 279–295.

[10] IEEE, *IEEE Standard for Radix-independent Floating-point Arithmetic, ANSI/IEEE Std 854-1987*, 1987.

[11] Daniel Jackson, *Alloy: A new technology for software modeling*, TACAS '02, LNCS, vol. 2280, Springer, 2002, p. 20.

[12] Daniel Jackson, Ian Schechter, and Hya Shlyahter, *Alcoa: the alloy constraint analyzer*, ICSE '00: Proceedings of the 22nd international conference on Software engineering (New York, NY, USA), ACM Press, 2000, pp. 730–733.

[13] Guodong Li, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby, *Formal specification of the MPI-2.0 standard in TLA+*, Principles and Practices of Parallel Programming (PPoPP), 2008.

[14] Microsoft, *Phoenix academic program*, http://research.microsoft.com/phoenix, 2007.

[15] http://www.cs.utah.edu/formal_verification/mpitla/.

[16] Flemming Nielson, Hanne R. Nielson, and Chris Hankin, *Principles of program analysis*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[17] Robert Palmer, Steven Barrus, Yu Yang, Ganesh Gopalakrishnan, and Robert M. Kirby, *Gauss: A framework for verifying scientific computing software.*, SoftMC: Workshop on Software Model Checking, ENTCS, no. 953, August 2005.

[18] Robert Palmer, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby, *An approach to formalization and analysis of message passing libraries*, Formal Methods for Industry Critical Systems (FMICS) (Berlin), 2007, Best Paper Award.

[19] Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby, *The communication semantics of the message passing interface*, Tech. Report UUCS-06-012, The University of Utah, October 2006.

[20] ———, *Semantics Driven Dynamic Partial-order Reduction of MPI-based Parallel Programs*, PADTAD '07, 2007.

[21] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp, *Formal verification of programs that use MPI one-sided communication*, Recent Advances in Parallel Virtual Machine and Message Passing Interface (Berlin/Heidelberg), LNCS, vol. 4192/2006, Springer, 2006, pp. 30–39.

[22] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp, *Formal verification of programs that use MPI one-sided communication*, Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI), LNCS 4192, 2006, Outstanding Paper, pp. 30–39.

[23] Stephen F. Siegel, *Model Checking Nonblocking MPI Programs*, VMCAI 07 (B. Cook and A. Podelski, eds.), LNCS, no. 4349, Springer-Verlag, 2007, pp. 44–58.

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

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

[26] Jeffrey M. Squyres and Andrew Lumsdaine, *A Component Architecture for LAM/MPI*, Proceedings, 10th European PVM/MPI Users' Group Meeting (Venice, Italy), Lecture Notes in Computer Science, no. 2840, Springer-Verlag, September / October 2003, pp. 379–387.

[27] The Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.*, 1995.

[28] Leslie Lamport, research.microsoft.com/users/lamport/tla/tla.html.

[29] Leslie Lamport, The Win32 Threads API Specification. research.microsoft.com/users/lamport/tla/threads/threads.html.

[30] Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby, *Isp: A tool for model checking mpi programs*, Principles and Practices of Parallel Programming (PPoPP), 2008.