EPOS-MPI: a highly configurable Run-time System for Parallel Applications

André Luís Gobbi Sanches and Antônio Augusto Fröhlich
UFSC/CTC/LISHA
PO Box 476
88049-900 Florianópolis - SC, Brazil
{gobbi,guto}@lisha.ufsc.br
http://www.lisha.ufsc.br/∼{gobbi,guto}

Abstract

This paper presents an implementation of MPI over a highly configurable operating system, EPOS. By using the advanced features of EPOS Communication system, MPI was implemented as a thin layer instead of a *middleware* library. Modern software engineering techniques were applied in the development process, such as static metaprogramming and aspect orientation.

Keywords: application-oriented operating systems, generative programming, MPI, collective communication.

1 Introduction

The *Message Passing Interface* (MPI) [3] standard played a major role for the dissemination of parallel programming. Being a *de facto* standard in the filed, with implementations for virtually any distributed memory parallel machine, MPI was one of the main factors that stimulated the cluster computing phenomenon of recent years. As ordinary desktop operating systems begun to figure MPI implementations, it became possible for numerous parallel applications, originally developed for supercomputers, to be executed in low-cost clusters of commodity workstations. Moreover, the availability of a common API for message passing programming encouraged the development of several new parallel applications.

Other nuisances of the cluster computing phenomenon are not so straightforward, however. For instance, most people have "commodity hardware" as an axiom of cluster computing and "commodity software" as a corollary. By analyzing the question in depth, one will easily conclude that what we now call "commodity hardware" could very well be called "supercomputing hardware" ten years ago: commodity processors now feature multiple functional units that operate in parallel; commodity memory hierarchies now deliver amazing storage capacity and bandwidth; and commodity high-speed networks now operate at speeds close to internal busses. In deed, what happened was a convergence of desktop computer hardware toward high-performance.

Notwithstanding, the reasoning presented above fails when applied to "commodity software". As of today, commodity desktop system software can be summarized in two families: UNIX and WINDOWS. Both families have been developed to support interactive, graphical, web-aware applications. In contrast, looking behind the front-end of a supercomputer one is likely to find a highly specialized operating system, designed and implemented to satisfy the heavy demands of parallel applications [17, 13, **?**]. There is no convergence foreseeable in this area, since parallel run-time support systems are way too specialized to support an all-purpose desktop. Indeed, several MPI implementations for clusters of workstations rely on communication subsystems that bypass the host operating system to directly access the interconnection hardware [9, 16, 12, 10]. This is the most evident recognition that commodity system software is not adequate to support parallel computing on clusters of workstations.

In fact, MPI is a standard INTERFACE that was idealized to be implemented on top of a native communication system. The inability of ordinary operating systems to deliver an adequate communication system made room for complex *middleware* implementations that override much of the native communication system [7, 2, 14]. These middleware implementations are far from ideal, for they aggregate unnecessary overhead and usually fail to explore particularities of both run-time system and interconnection hardware. For instance, some high-speed networks feature their own processor and could operate as an asymmetric multiprocessor with the main proces-

sor(s), implementing most of what is needed to deliver an MPI-compliant communication system. This would be impracticable with a middleware implementation.

Nevertheless, if MPI is nothing but an interface, exporting the functionality of a parallel run-time environment —designed and implemented according to modern software engineering techniques— in a way that complies with the standard should be straightforward. Furthermore, it could reveal novel alternatives to handle important aspects such as group communication, collective operations, and data encapsulation. In this scope, this article presents an implementation of MPI based on the *Application-Oriented System Design* method proposed by Fröhlich [4]. The main idea is to produce a set of software components that are able to efficiently bridge a preexisting communication system to the MPI standard, taking in consideration the specific requirements of particular applications. The substratum for this work was the communication system of the *Parallel Embedded Operating System* (EPOS) [5], which is based on the MYRINET high-speed network [1].

The remainder of this article is organized as follows: section section 2 describes the structure of a traditional MPI implementation; sections 3 4 describe the presented implementation and compares its performance to another MPI implementation; section 5 lists related work; and section 6 is this paper conclusion.

2 Traditional MPI Implementations

In order to discuss typical MPI implementations, a descrition of the MPICH implementation follows. MPICH was selected as an example because:

- it's a high-quality implementation, and maybe the most used;
- it's the target of many research projects, specially performance improvement projects;
- its design is well documented and its source code is freely available;

The MPICH implementation is divided in two layers. The upper layer is *architecture independent* and the lower layer, known as ADI (Abstract Device Layer) is dependent on the operating system and the communication device. The *ADI* handles point-to-point communication. The ADI must provide blocking and immediate functions for each of the 4 MPI communication modes. The upper layer provides collective operations, datatype handling and session and error handling.

The MPICH team provides the upper layer, while the lower layer is often maintained by the manufacturer of the communication system. MPICH also provides an ADI template with most functions based on a small subset named *The Channel Interface*. Developing an entire ADI involves a significant amount of work, so that the channel interface is provided to leverage the development process, and the manufacturers can modify the template ADI later for better performance. Figure 1 from [7] illustrates the role of each layer in a MPI_Reduce call.

The manufactures often have to modify the architecture independent layer. Collective operations, for instance, can be adapted to the network topology. Archicteture independent functions are implemented as weak symbols (#pragma) and may be redefined. The distiction between the two layers cannot be always kept. Besides, customizing MPICH requires significant work. MPI collective communication code has almost 9000 lines, the ADI over GM (a low-level API for Myrinet Networks) has more than 28000 and the archicteture independent code has 55000 lines of code. Since no modern software engineering is used (not even object orientation) in the development process, the maintainance of a code this size may be complex.

Besides, one shall recall that MPI is just an standard API for low level communications systems. If huge middleware libraries are needed, that's an evidence that typical operating systems are poor.

3 Design Rational of an Application-oriented MPI

3.1 Collective Communication

Every network topology require a specific routing algorithm. On Ring topology, for instance, the sender must decide if it should send the message to his predecessor or to his successor, seeking the least number of *hops*. Every node that receives a message that is not addressed to itself forwards the message on the same direction.

For instance, on figure 2 node 7 sends a messsage to node 2, in an homogeneous network with 8 nodes. If the message is sent to the left, $7 - 2 = 5$ *hops* are necessary, against $2 - 7 + 8 = 3$ if the message is sent to the right. Therefore, the node 7 will send the message to node 8. Since the message isn't addressed to node 8, it

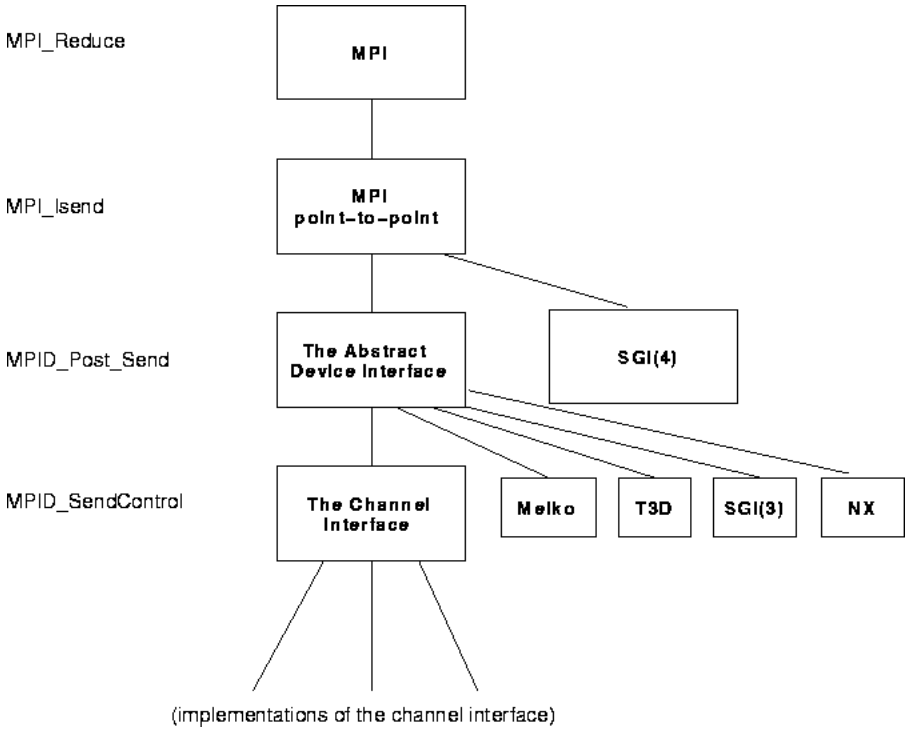

Figure 1: MPICH Architecture

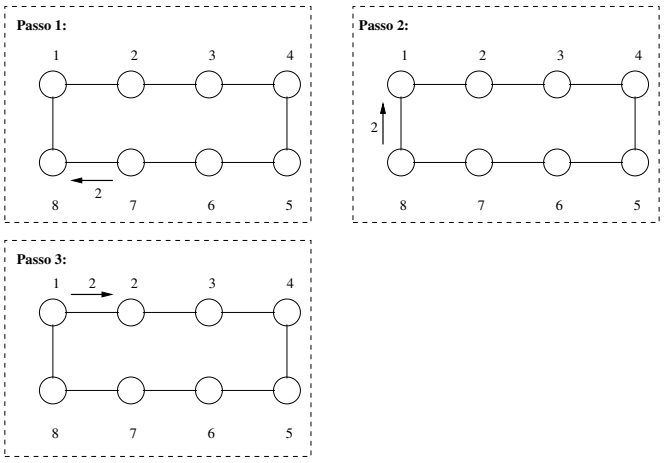

Figure 2: Message Send on Ring Topology

will forward the message to node 1, placed on its right. As long as no other communication happens between the involved nodes, latency will probably be smaller than sending to the left.

However, as long as other communications involving those nodes happen, the other route could provide better results. But there's a higher *probability* that the route involving less nodes could be better, unless it's possible to predict the behaviour of the other nodes based on features of the application. On MPI collective operations it's possible.

On collective operations, the *root* of the operation always sends, receives or sends *and* receives data to all other nodes. Assuming that execution is synchronized (what can be achieved with `MPI_Barrier`), any node can predict the communications involving every other node. Therefore, the routing algorithm shouldn't take only the number of hops in account, but also the behaviour of the other nodes.

The figure 3 shows an algorithm for the `MPI_Scatter` collective operation where the node 1 is the root. On the `MPI_Scatter` operations, the messages always originate from the root node, which becomes the bottleneck of the operation. It's not possible to use a *pipeline*, because the root node must transmit $N - 1$ messages.

The algorithm separates the network in two subnetworks, excluding the root node: nodes 2-5 and nodes 6-8. Then, the root sends a message to each subnetwork, starting from the farther node and ending with the closest. The first message is sent to the subnetwork with more nodes.

The messages that are sent to the farther nodes will have more hops than those sent to the closest nodes, and so they should be sent before.

The `MPI_Scatter` algorithm shown on figure 3 is composed by the sequence the message should be sent: 5, 6, 4, 7, 3, 8, 2. Collective operations algorithms should not care about the routing of the messages, because that is dealt at the point-to-point communication. Only the sequence of communications, which take the topology and the number of hops into account, is important.

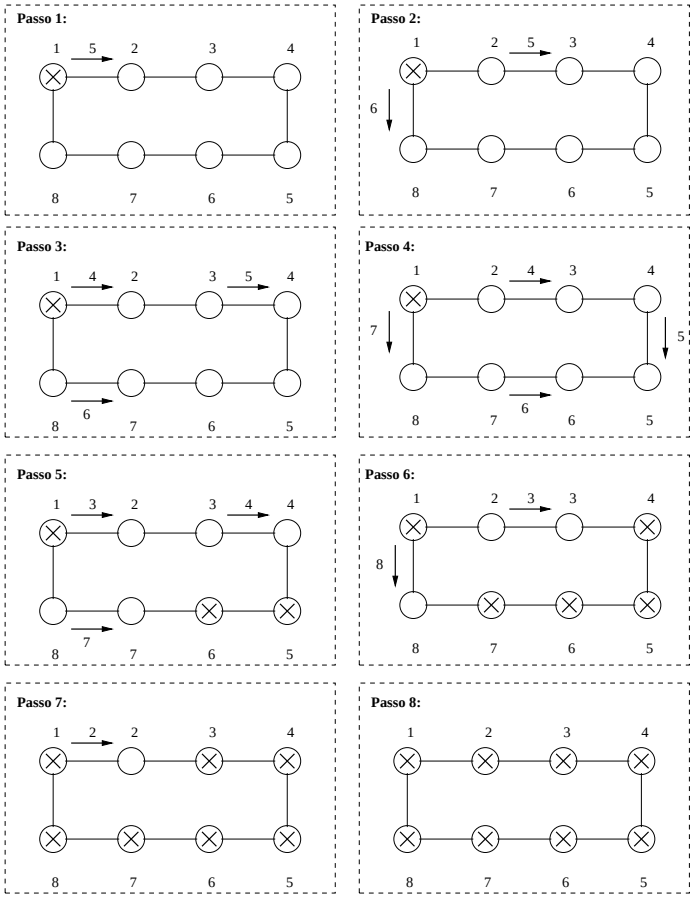

Figure 3: MPI_Scatter on Ring Topology

If the network topology is different, only the order that the messages are sent has to be modified. If the topology is linear, the messages would be sent in the order: 8, 7, 6, 5, 4, 3, 2, 1. No other modification is necessary to implement the operation.

MPI_Barrier	MPI_Gather + MPI_Bcast
MPI_Bcast	tree
MPI_Gather	linear
MPI_Scatter	linear
MPI_Allgather	ring
MPI_Alltoall	N MPI_Bcast
MPI_Reduce	tree
MPI_Allreduce	MPI_Reduce + MPI_Bcast
MPI_Reducescatter	MPI_Reduce + MPI_Scatter

Table 1: Collective Operations Topologies

Even if the topology is complete, no other change is necessary. The messages could be sent in any order, but the latency would be the same: it's not possible to use a *pipeline*, because the root node would still be a bottleneck. Therefore, the *natural topology* of `MPI_Scatter` is the *linear topology*.

The topology of each collective operation depend on the existence or not of a bottleneck. The bottleneck, when there is one, is the root node. Every collective operation is composed by a sequence of point-to-point operations between nodes. `MPI_Scatter`, for example, is composed by a sequence of operations between the root node and the others, and thus can't be optimized by a pipeline.

The `MPI_Bcast` operation, however, sends the same data to all nodes, and thus a receiving node may foward the message on successive steps. In the step i, $2^{(i-1)}$ concurrent operation may occur, decreasing the complexity of the algorithm from `O(N)` to `O(log N)`, what may be called a *tree topology*.

Every MPI collective operation that is not composed by others has linear or tree topology. `MPI_Allgather` has *ring* topology. The only difference between the linear and ring topologies is the sequence of communications, therefore the ring topology may be seen as an extension of the linear one. The topologies of the collective operations, as implemented by MPICH, are listed in the table 1.

3.2 Application Topology

Some applications, classified as *structured* [6], have a natural topology. All collective operations, as shown before, have a natural topology, that is tree or linear. However, the network topology may differ from the application topology, and it's necessary to emulate the application topology respecting the features of the network topology.

As discussed before, the only information concerning the topology is the order which the messages are sent: the numbers of messages will be the same. The application topology is a sequence of N operations if linear, and $logN$ if tree. The application impose no restrictions on the sequence of the communications. The network topology, otherwise, do impose, as can be seen in figure 3. So, the network topology modify only the sequence of the comunications, not the collective operations algorithm itself.

However, the algorithm of some collective operations can modify the sequence of the communications. `MPI_Alltoall`, for instance, can be implemented as a sequence of `MPI_Bcast` operations with different root nodes. If all nodes send their messages in the same sequence, there will always be a bottleneck. Shifting the sequence by the rank of the root may be a possible solution to this problem. Thus, each node will follow a different sequence, avoiding colisions.

The sequence of communication may be seen as a vector whose initial value is defined by the network topology. Its value is modified by the enabled *aspects* [8], which are defined by the application topology. If the application topology is ring and the network topology is hypercube, for example, the linear topology with the *ring* aspect enabled will be selected. The initial sequence will be suitable for linear operations over hypercubes modified by the ring extension. If a specific sequence for the ring topology over hypercube is defined, it will be used instead.

This separation makes possible to implement MPI collective operations in a topology-independent way, as a collection of *aspects* over the sequence and a point-to-point operation between the node and each element of the sequence. One can define new topologies by defining the initial sequence, all collective operations will be available on the new network topology. Thus, most of the collective operation code may be reused.

Complex topologies may be described as collections of simpler topologies. In the first step, the sequence is initialized with the top-level network topology. In the next steps, each element is replaced by the topology one level below, and so successivelly, as figure 4 shows. In that example, the top-level topology is linear, where each node represents a topology with ring topology. In the second step, the node 1 of the top-level topology is replaced by the nodes that compose the subnetwork, in the sequence of a linear topology.

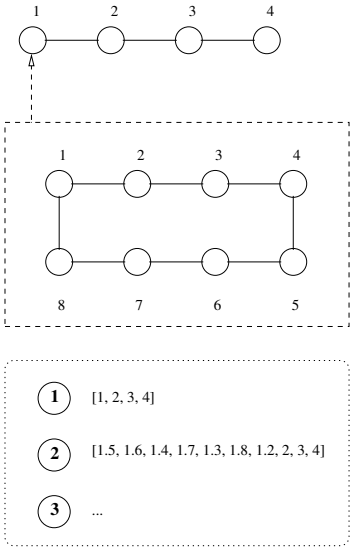

Figure 4: Topology Composition Example

3.3 The Topology Iterator

In C++ standard library [15], a vector can be efficiently represented by a container, and its contents can be accessed by an *iterator*, an objects that references its contents. In the previous section, topologies have been represented as *sequences* of communications. Therefore, topologies may be represented by containers. The collective operations algorithms iterate through the topology container, and executes the respective communications. This leads to the concept of a *Topology Iterator*.

The topology iterator may be the only way the application access the network. It hides the network topology, despite its complexity, e offers a simple interface: the operators `++`, `--` and `*`. Through `++ --` the iterator navigates through the container. The operator `*` references the element in the current position.

The containers which represent the topologies offer similar interface to those offered by the ISO C++ standard library. Each *container* offers an iterator for standard access to its contents. The interface of containers and iterators conforms to the ISO C++ standard.

The algorithms of the collective operation can be implemented using only the iterators, in a topology- independent way. By defining the network topology and the necessary extensions (*aspects*), the operations are instantiated. *Static metaprogramming* [15] is used, replacing the iterator by a single pointer. This approach avoids unnecessary function calls. Thus, the use of topology iterators has no impact in performance. In order to verify it, a program has been implemented with iterators and with pointers, with an aspect enabled. The *assembly* generated by both implementations were equal.

The network topology must supply two containers in order to support the collective operations. Those containers represent the linear and tree application topologies. The rest of the implementation of the collective operations has no dependency on network features.

3.4 Direction of Collective Operations

The `MPI_Gather` and `MPI_Scatter` operations behave in quite a similar way. Both operations involve a sequence of communications with other nodes. They only differ in the point that while `MPI_Scatter` root sends, in `MPI_Gather` the root receives. It's a matter of direction of the messages.

The direction of the messages is an important trait of collective operations. It defines which point-to-point operation will be called by the root node and its complement, which will be called by the other nodes. In the operation `MPI_Gather`, as an example, the root node call a receive operation and the others call a send operation. In the `MPI_Scatter` operation, the opposite happens.

The separation of topology and algorithm allows the reuse of the same implementation for opposite collective operations. The three directions supported are listed on the table 2. The direction from the root node to the others is considered *straight* and the opposite is considered *reverse*. *Bidirectional* is used in operations such as `MPI_Allgather` on which each operation is a send and a receive.

The direction of each collective operation is listed on the table 3.

Name	Root Node	Other nodes
`Straight`	Send	Receive
`Reverse`	Receive	Send
`Bidirectional`	Send and Receive	Send and Receive

Table 2: Possible Directions

Operation	Topology	Direction
`MPI_Bcast`	`tree`	`straight`
`MPI_Gather`	`linear`	`reverse`
`MPI_Scatter`	`linear`	`straight`
`MPI_AllGather`	`linear`	`bidirectional`
`MPI_Alltoall`	`linear`	`bidirectional`
`MPI_Reduce`	`tree`	`reverse`

Table 3: Direction of each Collective Operation

3.5 API Independence

As discussed in the previous section, the direction of a collective operation define the function which will be called on which node. Those functions can be send, receive or send-and-receive functions.

However, those functions doesn't need to be MPI functions. Instead, they can be the point-to-point functions offered by any message passing system, such as PVM. Therefore, the collective operations can be ported to other systems without modification.

The *Direction* class, that represents the direction of the operations, will be parametrized. Its parameters are the point-to-point functions it will call. A set of operations is defined with the blocking operations and another can be defined with the immediate operations, or operations from other API. The operations that compose the set are encapsulated by function objects.

3.6 Global Reduction Operations

The behaviour of global reduction operations (such as `MPI_Reduce`) is API defined. Therefore, they are aspects over the set of function objects. The API must offer support for reduce operations. The aspect will perform the reduce operations each time an iteraction occurs.

4 EPOS-MPI

MPI functions can be divided in three categories according to their functionality:

1. Point-to-point communication
2. Datatype Definition and Handling
3. Collective Communication

Collective Communication has been described in the previous sections in a platform independent way. Point-to-point communication and Datatype Handling, however, depend on the communication hardware and the operating system. In this section, their implementation on the EPOS system is described. In the end of the section, the performance of a prototype is evaluated.

A detailed description of *EPOS* and its communication system can be found in [4].

4.1 Point-to-Point Communication

The MPI standard specifies four communication modes for point-to-point communication, with corresponding functions. The communication mode for each operation is defined by the send function specified. The four modes are:

- *Ready:* this mode should only be used when the receive function is called before the send function. Otherwise, the behaviour is undefined;
- *Buffered:* if the message can't be sent at once, it should not wait. The message is stored in a system buffer and the send function returns;
- *Synchronous:* the send call won't return before the receiving procedure has started;
- *Standard:* the send function may return immediately or wait until the receive call is posted. The behaviour is defined by the implementation;

The communication mode defines if the message is sent as soon as possible or if the communication system wait for a request from the sender. Sending the message as soon as possible implies in smaller latency, but unexpected messages requires an extra copy on the receiver. Before the receive call is posted, the user buffer is not ready yet, and thus the arriving messages must be temporarily stored in a system buffer. The process of waiting for a request to send a message is called *Rendezvous*, and is well described by O'Carroll et al [11].

If the *Ready* send is called, it's assumed that the receive call has already been posted. The message is always sent at once.

If the *Synchronous* send is called, the function cannot return until the receive call is posted. Therefore, the *Rendezvous* protocol is suitable.

If the *Buffered* send is called, the message is sent immediately. If the network is saturated and the message can't be sent right now, it's stored in a system buffer and sent later.

If the *Standard* send is called, *Rendezvous* will be used only in long messages. Short messages should be sent right away.

EPOS communication system offers the *Buffering* configuration option. This option is enabled in order to support the *Buffered* send mode. The *Rendezvous* protocol is supported by the communication system also, through the *scenario aspect* with the same name.

4.2 Message Identification

The MPI standard states that four fields identify a message:

- source
- destination
- tag
- context (or communicator)

This information is called *message envelope*. Send and receive calls may only match if they have identical message envelopes. Two wildcards are possible on the receive call: MPI_ANY_SOURCE and MPI_ANY_TAG.

The order by which messages arrive can be different from the order by which they are requested by the user. Therefore, a queue system is necessary. The queue system is ordered by message envelopes, so it's MPI specific. However, EPOS is supposed to handle all communication functionality, so that MPI can be implemented only as an API. In order to delegate the handling of the queues to EPOS communication system, EPOS envelope abstraction will be enhanced to a *parametrized envelope*.

MPI allows immediate communication, so that it's possible to wait several messages at the same time. Therefore, a *expected* queue is needed. If immediate communication is not used in that application, it's possible to wait only one message each time and the *expected queue* becomes a pointer. This queue system was proposed by O'Carrol et al [11]. However, O'Carrol manages the queues in the MPI implemention, while in EPOS-MPI they are managed in the communication system, as described below.

4.3 Parametrized Envelope

In EPOS, messages involved in communications are represented by an *Envelope* object. EPOS envelope is protocol independent, and the header is just treated as data to be sent. That isn't suitable for handling the queues, because it's necessary to identify and compare messages by their headers, in order to manage the queues in the communication system.

The envelope shouldn't be dependent on any given protocol, but must be able to deal with its header. Generic programming can be used in order to achieve this. An envelope is based on the *concept* of a header, but not in any specific header. Thus, header is a *template parameter* [15] of the envelope class. Envelope has no knowledge on the details of the headers, but can compare them through a standard interface: the operators `==` and `<`. Thus, supported headers must be represented by classes that:

- are continuous data;
- offer the operator `==` for comparison;
- offer the operator `<` for ordering;
- offer the operator `=` for copying.

With the parametrized envelope, EPOS communicator can handle the message queues in a protocol-independent way. The receive call may specify to the communicator the header of the message it expects. The communicator only returns messages that match the specified header.

The use of parametrized envelopes relieves the MPI implementation from queue handling, and thus the queues may be used on the implementation of other message passing systems. Besides, the envelope guarantees *zero copy* in all communications.

The parametrized envelope is an specialization of EPOS envelope abstraction. If the network is heterogeneous, the evenlope abstraction is mapped to the *Typed* class, otherwise to the *Untyped* class. Both classes share the same interface, and both can be specialized by the parametrized envelope.

4.4 The Rendezvous Protocol

The *Rendezvous* protocol is supported by EPOS through the scenario aspect *Synchronous*. If this aspect is enabled, a queue system is also necessary for sending messages. The queue system is similar to the receive queue system. Two queues are used: *requested* and *unrequested*. *Requested* stores the *requests* of messages whose receive call has already been posted and should be sent at once. If the request hasn't been received when the send call is posted, the message is stored in the *unrequested* queue. O'Carrol [11] also used those queues, but just as the receive queues, they were managed by the MPI implementation and not by the operating system.

If the aspect *Rendezvous* is enabled, the class *Envelope* will have an additional attribute that defines if the message should use *Rendezvous*.

4.5 Sending Messages

When a message is sent, an envelope is filled with its header and passed to an EPOS *Communicator* (using the `<<` operator). *Communicators* are objects which represent a communication link in EPOS. They provide basic communication functionality, and may be extended by *aspects* [8]. An example of an *aspect* is the capacity to use the *Rendezvous* protocol, when the *Synchronous* aspect is enabled. According to the attributes of the message, the *Rendezvous* protocol is used or not, as described in subsection 4.1.

If *Rendezvous* is used, the *Communicator* searches the *requested* queue for a request. If the request is found, the message is sent immediatelly. Otherwise, the message is stored in the *unrequested* queue. If the called function is immediate, it returns. If it's blocking, it checks the network events until the send operation is completed.

When a request arrives, the *Communicator* searches the *unrequested* queue. If a matching message is found, it's sent. Otherwise, the request is kept in the *requested* queue.

If the *Rendezvous* protocol isn't used, the message is sent at once or stored in a system buffer and is sent later. When the message is sent it's completed.

The network events are checked every time a blocking function or immediate operation testing function (eg. `MPI_Wait`) is called.

EPOS *Communicator* handles entirely the sending of messages, and the implementation of `MPI_Send` is essentially the initialization of the header:

4.6 Receiving Messages

The messages reception is analogous to the process of sending messages using the *Rendezvous* protocol. When a receive function is called, an envelope with the expected message header and the user buffer's address and size are passed to the *Communicator* (using the `>>` operator). The *Communicator* searches the *unexpected* queue for

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
{
    message_t message;
    message.header.node = MPI_rank;

    // MPI_rank is the global rank of this process
    message.header.context = comm.get_context();
    message.header.tag = tag;
    message.buf = buf;
    message.len = get_extent(datatype)*count;
    message.node = rank2node_id(comm.get_global_rank(dest));

    // send operation
    // epos_comm is the EPOS Communicator
    return (*epos_comm << message);
}
```

Figure 5: caption

a matching message. If a message is found, the operation is completed. Otherwise, the header is stored in the *expected* queue. If the receive call is blocking, the network is checked for events until the operation is completed. If the receive call is immediate, it returns.

When a message arrives, the *Communicator* searches the *expected* queue for a matching header. If one is found, the message is completed. Otherwise, the message is stored in the *unexpected* queue.

The arriving messages' headers overwrite the headers specified by the receive functions, in order to replace any wildcards such as `MPI_ANY_TAG` or `MPI_ANY_SOURCE`.

The network events are verified every time a blocking or an immediate

operation verifying function is called.

The implementation of the `MPI_Recv` function is listed below, in order to illustrate its simplicity. An envelope is instantiated and initialized, sent to a *Communicator* and then analyzed.

```
int MPI_Recv(void * buf, int const count,
             MPI_Datatype const datatype, int const source,
             int const tag, MPI_Comm const comm,
             MPI_Status * const status)
{
    message_t message;

    message.header.node = source;
    message.header.tag = tag;
    message.header.context = comm;

    message.buf = buf;
    message.len = get_extent(datatype)*count;
    message.node = rank2node_id(source);

    // receive operation
    *epos_comm >> message;

    status->count = message.len;
    // the division by get_extent(datatype) is done at MPI_Get_count
    status->MPI_ERROR = 0;
    status->MPI_SOURCE = comm.get_local_rank(message.header.node);
    status->MPI_TAG = message.header.tag;

    return 0;
}
```

Figure 6: caption

4.7 Datatype Handling

Datatype handling may be seen as support functions to point-to-point communication. MPI offers functions to define datatypes as arrangements of the pre-defined types. The functions used in the application define the complexity of the datatype support. Four levels of complexity are taken into account:

1. *None:* `MPI_Byte` is the only datatype used. No support is necessary;
2. *Basic:* only the pre-defined datatypes are used;
3. *Contiguous:* the user defined new datatypes, but they're all contiguous. Only the `MPI_Type_contiguous` function is used;
4. *Total:* the user defined non-contiguous datatypes.

In EPOS, the datatypes are handled by the *envelope* abstraction. The datatype definition functions that are used define the complexity of the envelope that will be used. If non-contiguous datatypes are used, a message can be composed by a collection of buffers. The envelope must have a vector to store the collection of buffers. If only contiguous datatypes are used, a single pointer replaces the vector. EPOS envelope has support for both cases.

If the network is heterogeneous and any datatype except `MPI_Byte` is used the data must be converted in network notation before transmission. In this case, the *Typed* member of the envelope family is selected. Otherwise, the *Untyped* envelope is used.

The datatype handling system is configured through a *script* which analyse the application. For each level of complexity listed above, there's a header file with the definitions necessary to support it. The *script* selects the adequate header based on the MPI symbols found in the application. This selection doesn't alter other unities of the MPI implementation.

The use of MPI Datatypes may influence the configuration of EPOS, but no feature has to be added to the system. EPOS communication system has complete support for MPI datatype handling.

4.8 Evaluation

In order to evaluate the performance of EPOS-MPI, a prototype has been implemented and compared to MPICH over GM. GM is the low-level message passing system for Myrinet Networks. EPOS communication system was emulated over GM for the comparison. Figure 7 shows the latency of both implementations executing a MPI ping-pong application. The application binary was significantly smaller on EPOS-MPI: the application linked with the library has 21k, against 400k using MPICH-GM.

The latency of the prototype was 5% higher, but further improvements on the communication system emulation may reduce it.

5 Related Work

O'Carroll et al [11] has designed and implemented a MPI implementation with zero copy. He used a similar queue system, but handled the queues at the MPI level, and not below it. The use of a parametrized envelope in EPOS communication system also guarantees zero copy but relieves the application from communication management.

Vadhiyar et al obtained performance improvement of 30%-650% by automatically tuning collective communication. When the application starts, a sequence of tests is done in order to find the best algorithm for each collective operation.

The 9 algorithms used by Vadhiyar can be decomposed as presented in this paper, and described as topologies and aspects. Heuristics can be defined in order to calculate the best aspect to apply. Vadhiyar work could be much simplified by using the presented decomposition.

6 Conclusions

In this paper a new MPI implementation was presented. The implementation is based on advanced communication system, so that MPI was implemented as a thin layer and no *middleware* library is required.

In section 3.1 MPI collective operations were decomposed in two abstractions: Topology and Direction. Topology represents the interactions between the communication nodes, and Direction defines the role of each node in

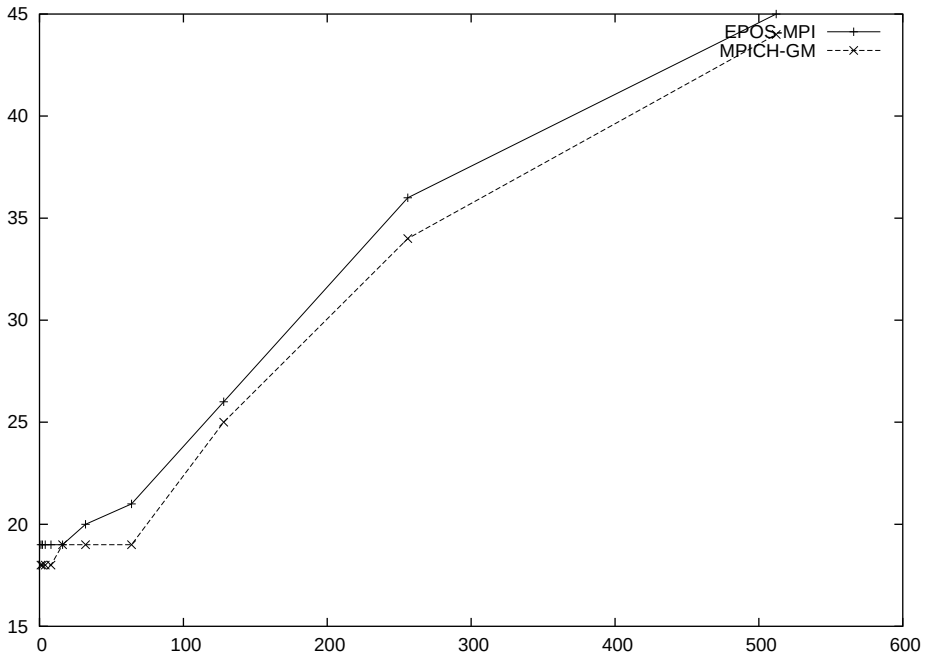

Figure 7: Performance of a MPI ping-pong application

the communication process. Direction is generic on the point-to-point functions it calls, so that the collective operations can be ported to other message-passing systems.

As future work we look for a more detailed investigation of the effects from applying this technique to parallel applications.

References

[1] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[2] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[3] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, 1995. version 1.1.

[4] Antônio Augusto Medeiros Fröhlich. *Application-Oriented Operating Systems*. PhD thesis, GMD-FIRST, June 2001.

[5] Antônio Augusto Medeiros Fröhlich, Gilles Pokam Tientcheu, and Wolfgang Schröder-Preikschat. Epos and myrinet: Effective communication support for parallel applications running on clusters of commodity workstations. *Proceedings of the 8th International Conference on High Performance Computing and Networking*, pages 417–426, May 2000.

[6] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[9] Steven S. Lumetta, Alan M. Mainwaring, and David E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of Supercomputing'97*, Sao Jose, USA, November 1997.

[10] Inc. Myricom. The gm message passing system, 1999.

[11] Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, and Yutaka Ishikawa. The design and implementation of zero copy mpi using commodity hardware with a high performance network. In *Proceedings of the 12th international conference on Supercomputing*, pages 243–250. ACM Press, 1998.

[12] Loic Prylli and Bernard Tourancheau. BIP: a New Protocol Designed for High Performance Networking on Myrinet. In *Proceedings of the International Workshop on Personal Computer based Networks of Workstations*, Orlando, USA, April 1998.

[13] Wolfgang Schröder-Preikschat. PEACE - A Software Backplane for Parallel Computing. *Parallel Computing*, 20(10):1471–1485, 1994.

[14] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.

[15] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3 edition, June 1997.

[16] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Sato. PM: An Operating System Coordinated High Performance Communication Library. In *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 708–717. Springer, April 1997.

[17] S. Wheat et al. PUMA: An Operating System for Massively Parallel Systems. In *Proceedings of the Twenty-Seventh Anual Hawaii International Conference on System Sciences*, Maui, U.S.A., January 1994.