Università di Pisa
Dipartimento di Informatica

Technical Report : TR-97-04

Using MPI for P3L Implementation

Carlo Manconi, Barbara Cantalupo

February 06, 1997

ADDR: Corso Italia 40,56125 Pisa,Italy. TEL: +39-50-887111. FAX: +39-50-887226

UNIVERSITÀ DEGLI STUDI DI PISA
DIPARTIMENTO DI INFORMATICA

Technical Report: TR-97-04

Using MPI for p3l implementation

Barbara Cantalupo & Carlo Manconi

Abstract. This document concerns with the implementation of the P3L parallel programming language on the MPI abstract Machine. First we focus our attention on the characteristics of the MPI Abstract Machine, then we present a preliminary implementaton for each P3L construct based on that architecture.

Keywords. $\mathtt{P^3L}$ Parallel Programming, Skeletons, Implementation Template, MPI.

January 1997

ADDR:Corso Italia 40,56125 Pisa,Italy. TEL:+39-50-887000 - FAX:+39-50-887226
E_MAIL: manconi@di.unipi.it

Contents

1 Introduction

This document concerns with a preliminary study for porting the P^{3}L [DDMO+92] prototype compiling tools on MPI [Mes94].

P^{3}L is a *high-level* structured parallel programming language by which programmers can abstract from low-level implementation details, such as process mapping, interprocessor communications, process scheduling and can concentrate on the form of parallelism that must be exploited to parallelize a given application.

The P^{3}L methodology [BDO+95b, Pel93] is based on:

- the use of a *set of parallel constructs*;
- a *set of compiling tools*, which exploit the structured characteristics of the language and guarantee the efficient portability of applications;
- an *Abstract Machine*, which represents the fundamental mechanisms of the underlying architecture.

As concerns the forms of parallelism included in the parallel language, P^{3}L supplies distinct language constructs for each form of parallelism and allows to express parallel programs only by these constructs. In this way it is possible to devise efficient strategies to solve typical issues like mapping, scheduling, message routing, degree/granularity of parallelism. These strategies are recorded as parametric networks of processes, called *implementation templates*, which implement the basic structure of the construct. An implementation template associated with a parallel construct is composed by a network of processes, which code is defined by implementation processes, a network topology, which is defined by an implementation mapping, and a performance model, which allows to predict the template performance on the basis of the costs exported from the abstract machine and the costs of the user defined code sections.

P^{3}L uses an abstract machine to represent the main characteristics of the target physical architectures. In particular the P^{3}L abstract machine is defined by:

- the virtual processors topology;
- a set of process activation and process communication mechanisms;
- a set of cost measures, one for each mechanism considered.

Such an abstract machine can be mapped directly on the native primitives of the target physical architecture as well as an higher level abstractions. In both cases, it is necessary that a significant value can be measured to closely estimate the real execution time of each exported mechanism on the physical target architecture.

Message Passing Interface (MPI) [Mes95, DOSW95, Pac95] is a portable message-passing standard developed to make easier the development of parallel applications and libraries. MPI defines the syntax and semantics of a core of library routines useful for writing message-passing programs in Fortran 77 or C. The basic idea in MPI is that the message-passing paradigm is widely portable and scalable, and it is easily compatible with distributed-memory multicomputers, shared memory multiprocessors and networks of workstations. Furthermore MPI is sustained by a general believe that message-passing will not be made obsolete by increases in network speeds or by architectures combining shared and distributed-memory components.

This document is organized in two different parts. In the first part, we show how an abstract machine suitable for $\mathtt{P}^3\mathtt{L}$ can be defined referring to the MPI standard [Mes94] and explain the reasons for this choice. In the second part we discuss a design of the $\mathtt{P}^3\mathtt{L}$ implementation templates on the abstract machine, and we give a description of a prototype implementation of that templates. In particular, we first describe the subset of $\mathtt{P}^3\mathtt{L}$ constructs implemented and then we describe the design and the implementation of the corresponding templates.

2 Using MPI to define a $\mathtt{P}^3\mathtt{L}$ abstract machine

A $\mathtt{P}^3\mathtt{L}$ abstract machine is a general purpose DM-MIMD architecture having

- an unlimited number of processors,
- a complete interconnection network,
- distributed memory,
- the following primitives for parallelism management:
  - activation of a set of parallel processes,
  - point to point SEND/RECEIVE communication (both *blocking* and *non blocking*),

– collective communication on a subset of the parallel processes (BROADCAST, SCATTER, GATHER).

This abstract machine extends the usual $\mathtt{P}^3\mathtt{L}$ abstract machine [BDO+95b] with the features of MPI which are the most suitable to be exploited within a portable implementation of $\mathtt{P}^3\mathtt{L}$ implementation templates. In the following section we deeply analyze the MPI mechanisms which are included in the abstract machine.

We decided to use MPI to define the $\mathtt{P}^3\mathtt{L}$ abstract machine because MPI can guarantee both portability and efficiency. In particular, our choice is based on the following reasons:

- MPI is an accepted standard that is widely spreading in parallel computing community, thus using MPI guarantees high portability for the $\mathtt{P}^3\mathtt{L}$ compiling tools;
- though much of MPI just standardizes the *common practice* of existing systems (such as PVM [GBD+94], Chimp[BRV91, MPI92], Chameleon [Har93]), MPI has defined advanced features such as user-defined data types, persistent communication ports, powerful collective communication operations, and scoping mechanism for communication that may be very useful for easily implementing complex computations.

As we have already stated, we are not interested in giving here a comprehensive description of the MPI standard, but just in discussing some of MPI mechanisms that are more suitable to be used in the $\mathtt{P}^3\mathtt{L}$ abstract machine.

2.1 Process management

At the moment, MPI provides no support for dynamic spawning of tasks, but it is possible to *structure the program tasks dynamically in different* **groups** *of processes* depending on their communication patterns.

As soon as a program starts execution, all the activated processes are viewed as if they belong to a *special* process *group* that in some sense is the *MPI world*. Starting from this group, MPI supplies many functions to create or to destroy several kind of process subgroups. These subgroups are used to describe the participants to a communication *universe* and give them unique names with respect to that universe of communication.

In practice, groups are ordered sets of processes. Processes within a group are uniquely identified by contiguous integers starting from zero and representing *ranks* in the group. Hence, groups define a scope for process names in point-to-point communication (which ranks are the low-level names for) and fix the sets of processes that can be involved in collective operations. Every instance of a group may be joined to one or more *communication contexts* by means of data objects called *communicators*[1]. Contexts partition the communication space, both for point-to-point and collective operations, providing the ability to have separate safe *universes* of message-passing (i.e., separate resources and management). Every `MPI` communication operations specifies *communicators* and uses the process naming of its related group.

Even if it is stated that this kind of process management was included to support the creation of robust parallel libraries, collective operations can only be used if we structure a program in groups. In fact, as we will see in a subsequent section, collective operations can be applied just within the same context, hence among processes that belong to the same group. Thus, the groups structure must reflect in some way the communication patterns in the program. In other cases, restricting the communication context can be useful just to give the program more protection or to reflect program semantic structure. However, we think that defining a complex groups structure within a program when it is not necessary, may bring unnecessary overhead due to groups and contexts management.

Another interesting point related to the internal processes structure is the ability to partition a single process in several concurrent *lightweight* processes. In fact, although no explicit support for **threads** is provided, `MPI` has been designed so as not to prejudice their use and to allow for thread-safety. The ability of creating threads can be used in writing $\mathtt{P}^3\mathtt{L}$ templates, or in `MPI` implementation itself, to obtain a better performance for $\mathtt{P}^3\mathtt{L}$ programs.

The most direct use of multi-threading is to implement the overlapping of communication and computation tasks. For example, `MPI` implementation may define specific communication threads to realize non-blocking

[1] `MPI` provides both `intra-communicators`, that can be used by processes in the same group to communicate, and `inter-communicators` that can be used to perform point-to-point communication among processes belong to different groups. We are not interested in inter-communicators because they are useful just if processes are grouped according to a complex structure. On the contrary we think that minimizing the number of groups can help us to achieve better performance.

functions. Similarly, within $\mathtt{P}^3\mathtt{L}$ data-parallel templates (see sec. 4.4), we can implement complex communication patterns defining different threads to manage data manipulation and to control the communication itself.

Finally, it is important to consider the `MPI` style of programming. It is claimed that `MPI` is suitable for use by fully general MIMD programs, as well as those written in the more restricted style of **SPMD**. However, it looks evident, especially referring to collective operations, that `MPI` functions are defined according to a SPMD programming style. So, we think this programming style makes easier to write applications, and particularly, our $\mathtt{P}^3\mathtt{L}$ templates (see sec. 4).

2.2 Logical interconnection network

As we stated, $\mathtt{P}^3\mathtt{L}$ abstract machine is a fully connected abstract machine, with a uniform interconnection network. However, in some cases it is useful to include in a $\mathtt{P}^3\mathtt{L}$ template structure some **virtual topologies**. These topologies can be used to model complex communication pattern in a more appropriate way.

`MPI` allows to define *virtual topology* as a logical process arrangement that provides a convenient naming mechanism for the processes of a group (within a communicator). Processes may be arranged in topological patterns described by a graph that reflects the original communication patterns of the processes. A convenient process naming structure can bring benefits for program readability and notational power in message-passing applications. Additionally may assist the runtime system in mapping the processes onto hardware.

As `MPI` provides message-passing between any pair of processes in a group, a *missing link* in the user-defined process graph does not prevent the corresponding processes from exchanging messages. It means rather that the virtual topology gives a non convenient way of naming this pathway of communication. Hence, the virtual topology will represent just *privileged*, that is the most important or the most repeated, communication patterns. This topology will not necessarily reflects all the communication in the program or the real physical network in the target machine.

An interesting feature of virtual topology is that although its description depends only on the application and is machine-independent, the existence of related information may be used as advice by the runtime system. It means that topology can be exploited in the assignement of processes to physical processors and this may help to improve the communication performance on a given machine. However, whether this mapping is exploited and the way it works depends on the particular MPI implementation.

Specifying the virtual topology in terms of a graph is a complete way to describe the communication patterns. However, MPI allows to address explicitly some restricted topologies like rings and two, or higher, dimensional grids or tori that are completely defined by the number of dimensions and the number of processes in each coordinate direction. Thus, the user have to specify less detailed information and efficiency may increase, because the mapping of these structures is usually an easier problem than that of general graphs.

Virtual topologies can be an interesting feature for developing implementation templates for a skeleton based language, such as $\mathtt{P}^3\mathtt{L}$ because most common structure of $\mathtt{P}^3\mathtt{L}$ data-parallel computations are two, or higher, dimensional arrays of virtual processors. Using MPI grids topologies we can define process networks that reflect this virtual process structure. This could result in more program readability and, depending on *topologies* implementation, in a high improvement of template performance. This is particularly true in *data-parallel* templates such as map (see sec.4.4) where we have need an efficient distribution and collection of data.

2.3 Communication mechanisms

Sending and receiving messages is the basic MPI mechanism for communication, so MPI provides a number of basic point-to-point communication functions and several kinds of collective communication operations.

In this section, we just present the most interesting features of these functions for $\mathtt{P}^3\mathtt{L}$ implementation. First of all, we talk about point-to-point communication functions describing their parameters and their main differences with respect to semantics and efficiency, then we discuss collective operations. Finally we talk about data communication management.

2.3.1 Communication parameters and asymmetric communication

The basic point to point communication operations are `send` and `receive` functions that allow the communication of *typed*[2] data with an associated *tag*.

The information these functions need to send messages, or selectively receive them are specified in appropriate fields. These information are called the *message envelope*. In particular, envelope fields are the following [3]:

- *source* (in send function is implicitly determined by the identity of the message sender);
- *destination* (in receive function is implicitly determined by the identity of the message receiver);
- *tag*, an integer value that allows selectivity of message at the receiving end within the specified communication context;
- *communicator*, that specifies the communication context of the communication operation.

A message can be received by a receive operation if its envelope matches the `source`, `tag` and `communicator` values specified by the receive operation. However, the receiver may specify a **wild-card** value for `source` and/or a wild-card value for `tag`, indicating that any source and/or tag are acceptable (it cannot specified a wild-card value for `communicator` and a send operation must specify a unique receiver). This facility allows to implement **many-to-one** asymmetric communication within a communication context.

This is particularly useful for implementing nondeterministic computation. For example, asymmetric communication could be used to implement the *in-process* that manages the input tasks in `P`^{3}`L loop` template (sec.4.5). In fact, input tasks come from two different logical *channels*: the input and the feedback channel. Instead of testing the two different channels repeatedly, the *in-process* could use a unique asymmetric channel to receive tasks of both types. To implement a priority policy, the *in-process* might try first

[2]The type information is necessary for heterogeneous support to perform correct data representation conversions as data is sent from one architecture to another.

[3]A last parameter in the receive function is used to return information on the message just received.

to receive the feedback message (instantiating the source with the appropriate feedback process), and then, if no message is arrived, keep waiting on the asymmetric channel.

In addition to the corresponding envelope, each communication operations specifies a *send/receive buffer* in its own memory from which the message data is taken or it is stored in. The location, size and type of the communication buffer are specified by the first three parameters of the communication function call. The message type has to be an `MPI_DATATYPE`, that is one of the **MPI** predefined types, or a new type defined starting from these. Although the `MPI_DATATYPE` must be specified in each communication operation, the type matching between send and receive buffers is not really strict. First, because there are two general `MPI_DATATYPE` (`MPI_BYTE`, `MPI_PACKED`) that allow to communicate every kind of data without specifying its real type. Moreover, because the receive buffer can be larger than the send buffer, provided that the same `MPI_DATATYPE` is specified. The fact that type matching is not strict may be useful, for example, to receive dynamic data structures without knowing the message size. It suffices that data are structured in convenient way and a receive buffer having the maximum expected size is defined. Talking about data management we will discuss mechanisms to communicate complex or dynamic data.

2.3.2 Communication modes

As we stated above, `MPI` provides a number of point-to-point communication functions with different semantics. The functions provided are both *blocking* and *non blocking* and for each of them four different communication modes are provided.

In case of *blocking* semantics, the send call does not return until the send buffer can be reclaimed and similarly the receive function blocks until the receive buffer contains the message. According to *non-blocking* semantics, a send call returns before the message is copied from the send buffer and a receive call returns before a message is stored into the receive buffer.

In *blocking* semantics, the completion of a send implies that the send buffer can be used again and this allows to implement a protocol minimizing the copying and buffering of data. On the other hand, a *non blocking* semantics can improve system performance by overlapping communication and computation. However, a non-blocking protocol will necessarily end up doing more buffering and copying of data, because of data are buffered into the system.

We are interested for both blocking and non-blocking send/receive functions, because there are some cases in which overlapping computation and communications achieves good performance and other cases in which it is necessary to synchronize processes to preserve execution correctness.

In addition to blocking or non blocking semantics, `MPI` provides four different communication modes:

- in *standard* mode it is up to `MPI` to decide whether outgoing messages will be buffered; thus no assumption can be made in the application program about it or about the time the matching receive starts: if the messages are buffered, the send call may complete before a matching receive is invoked (even in blocking semantics); on the other hand, if buffer space is unavailable, or `MPI` chooses not to buffer them for performance reasons, the send call will not complete until a matching receive has been posted and the data has been moved to the receiver (even using a non blocking semantics);

- in *buffered* mode, the user can guarantee that a certain amount of buffering space is available and the space must be explicitly provided by the application program;

- in *synchronous* mode a rendezvous semantics between sender and receiver is used;

- in *ready* mode user exploits extra knowledge asserting that the matching receive already has been posted; this simplifies the protocol and potentially allows higher performance but can be used only in very particular cases.

Choosing to use non blocking functions, `MPI` provides an additional mechanism to improve communication performance. When communication with the same argument list is repeatedly executed it can be optimized it. This is obtained binding the list of communication arguments to a **persistent communication request** once and, then, repeatedly using the request to initiate and complete messages. A persistent request can be thought of as a *half-channel* (since there is no binding of the send port to the receive port) and it allows reduction of the overhead for communication between process and communication controller.

The *request* is created using a communication initialization function (corresponding to the four different communication modes for sending and to the receive operation) and remains inactive until a *start* function is called on it. It is possible to start more than one request at the time but if the same communication object is used in several concurrent threads, it is the user's responsibility to coordinate calls so that the correct sequence is maintained. The communication is completed by a call to *wait* function, that when completed makes the request inactive. However, the request is not deallocated and it can be activated by a new *start* function call.

As $\mathtt{P}^3\mathtt{L}$ templates are designed to be able to compute stream of data, their implementation processes are structured as cycles each one computing an incoming task (sec.4). Even if the computation clearly depends on the application code each process iterate the same basic computation and the same communication patterns. It is clear that persistent communications can be successfully used in process templates whenever a communication is needed and this can result in highly improving performance.

2.3.3 Collective communication functions

A collective communication is defined as a communication that involves a group of processes, specified by means of a communicator[4]. A collective communication is executed by having all the processes in the group call the corresponding communication routine, with matching arguments. The same function with the same parameters structure have to be called in each process, even if there is just a single originating or receiving process (specified as the *root*). The *root* has a different behavior from the others and some arguments are significant only for the *root* and are ignored by the others. It is clear that this kind of definition suggest, as we stated, an SPMD programming style and further strongly force the group program structure.

Generally, collective functions provided by `MPI` are variations of data distribution functions. Just one (`MPI_barrier`) serves to synchronize processes without passing data, whereas all the other functions make groups of processes exchange data with each other. The collective communication functions are:

[4] `MPI` guarantees that collective and point-to-point functions have different communication contexts even if they are called with the same communicator argument.

- `MPI_Bcast`: to broadcast data from one process to all the others in a group;
- `MPI_Gather`: to gather data from all processes in a group to one;
- `MPI_Scatter`: to scatter data from one to all processes in a group;
- `MPI_All_gather`: to gather data from all processes in a group and buffer it in all processes in the group;
- `MPI_All_to_all`: to send distinct data from each process in a group to each process in the group;
- `MPI_Reduce_scatter`: combined reduce scatter operation;
- `MPI_Scan`: scan operation across processes;
- global reduction operations: to perform operations such as sum, max, min and user-defined functions (the result is returned to all processes or, in a variation, to only one process);

`MPI` does not provide any function that is able to select a set of communication partners that is different from a group. Thus we are forced to use collective operations and define corresponding group structure, even to do a simple `multicast` operation.

Collective operations have syntax and semantics consistent with the syntax and semantics of the point-to-point operations but they have more restrictions. The type-matching conditions are more strict (the amount of data sent must exactly match the amount of data specified by the receiver) and they have only *blocking* version and just one mode analogous to the *standard mode* of the point to point ones. For program correctness, it is important to be careful with synchronization side-effects of collective operations. In fact, a collective communication call (except the barrier) may or may not have the effect of synchronizing all calling processes. Actually the completion of a call just indicates that the caller is now free to access location to the communication buffer but, in general, it does not indicate that other processes in the group have completed or even started the operation.

Summarizing, collective functions could be very useful to express easily several kind of complex communication patterns. Thus, they can improve program readability and if MPI implementation provides specific optimizations, they can result in highly increased communication performance. Finally, regarding $\mathtt{P}^3\mathtt{L}$ implementation, they may help us to define specialized processes to implement *data-parallel* template such as the one for map construct (sec.4.4).

2.3.4 Communication data management

The ability of structuring data to be used within communication operations is an important issue to define expressive power of a *message passing interface*. Some communication interfaces allow to pass only data with the same basic datatype. Others provide functions to pack complex data structures (such as structures combining different datatype or composed by noncontiguous data items) before sending it and unpacking it after receiving it. These restrictions can cause overhead at run time due to packing/unpacking operations or, alternatively, may increase the number of communication operations. On the other hand it is clear that minimizing the number of messages sent helps to save part of the communication overhead due to start up time. Several applications are composed by processes that need to exchange such a complex data.

MPI provides three mechanisms for this purpose, and the choice of which one fits best depends on data characteristics. They are:

- *count parameter* to use in communication routines;
- *packing/unpacking* functions;
- MPI *derived datatypes*.

When data items are stored in *contiguous* memory locations and they have the same basic or *derived* type, it is possible to group them into a single message simply using the *count parameter* in the communication routines. Instead, to pass messages that contain values stored in noncontiguous memory locations and/or that have different types you have to use the remaining two mechanisms.

Using packing and unpacking functions allows to explicitly store noncontiguous data in contiguous memory locations and to copy data from a contiguous buffer into noncontiguous memory locations. Thus, a complex data can be passed in a unique message by packing it at the sender site and unpacking it at the receiver site. These functions are provided for compatibility with previous libraries (such as PVM [GBD+94, SGDM94]) but they may be useful to manage messages that have a variable structure. For instance, a message can be composed by a first part, where information about the message is contained (such as the number of items that compose the second part), and a second part that can be variable. The message can be received in a buffer as large as the maximum expected size and the first part of the message can be unpacked. Finally, also the second part can be unpacked, depending on the contents of the second part.

A second solution is to build `MPI` *derived datatypes* at run-time to define more general, mixed type and noncontiguous communication buffers. These are not typical language *types* but just *opaque* objects that specifies the layout, in memory, of a set of primitives types. Using `MPI` type-constructor functions, the user specifies a sequence of basic datatypes (that are the member types) and a set of displacements (that are the member relative locations in memory). In this way, `MPI` is able to know the datatype structure and use this information to retrieve the right values during communication operations. The most general datatype constructor needs such a *complete* description of each element of the type. However, there are also datatype constructors for dealing with subset of the entries (contiguous, equally spaced or arbitrary) in an array that needs less detailed information.

`MPI` type-constructors can be applied recursively and derived datatypes can replace basic datatypes whenever they can be used.

This allow us to construct really general communication buffers. For instance, since `MPI` does not know the objects declared in the host language, to communicate a general object defined in the host language, it suffices to provide a definition of a communication buffer that mimics the definition of its structure or array section in question.

On the other hand, the most important restriction in applying this approach is that derived datatypes can be built just starting from a defined variable that describe the underlying data structure and selecting elements from it by means of their displacements. It means that message data structure has to be exactly known and in the host language a variable that contains message values has to be defined. Anyway, combining the use of derived datatypes and count parameters allows to pass messages composed with a variable number of complex structures, provided that these are stored in contiguous memory locations.

It is trivial to observe that the choice of the best data management mechanism depends strongly on data structure characteristics but also performance reasons have to be considered. The use of *count parameter* is the only approach that involves no additional overhead but, as we stated, it can be applied just for contiguous data. Packing/unpacking functions have the additional disadvantage of requiring additional memory-to-memory copy operations at both communication sides. Instead, using derived datatypes is up to the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted or whether it can be collected directly from where it resides. The overhead due to derived datatype creation and packing/unpacking functions depends on the implementation. Thus, knowing the costs of these functions may be important to choose he right approach to efficiently send heterogeneous data. Instead, if there are a large number of noncontiguous elements, building a derived datatype will probably involve less overhead than a large number of calls to packing/unpacking functions. In fact, the overhead of creating a derived datatype is incurred only once, while the overhead of calling packing/unpacking functions must be incurred every time the data is communicated.

Going back to $\mathtt{P^3L}$ implementation, $\mathtt{P^3L}$ templates (sec. 4) always communicate to each other complex data structures. In fact, in addition to the actual task, that in turns may be a complex data structure, each message always contains additional information such as task tags or termination marks. Moreover, as we stated above, they are structured as computational cycles that repeatedly communicate data having the same structure. For these reasons, the derived datatype approach seems the best approach to be adopted for template implementation. However, in some special cases, for example when a process has to receive a list of tasks whose length is unknown at compile time (such as in *map* template), we are forced to use packing/unpacking functions. An alternative method consists in restructuring the task appropriately in such a way that we can define a corresponding derived datatype, storing the task in contiguous memory locations, so that processes can communicate it using the appropriate count parameter value.

2.4 Profiling facilities

In `MPI` a standard *profiling interface* is defined and some requirements to `MPI` implementation are depicted in order to define such an interface. Anyway, the purpose of the MPI profiling interface is just to make easy to interface codes of profiling tools with MPI implementation on different architectures. MPI says *nothing* about the way the profiling interface is used. In particular, what information is collected and how the collected information is saved, filtered, or displayed will depend on specific profiling tools implementation.

As we stated, a complete definition of P^{3}L abstract machine needs to have accurate measures of exported mechanisms. Thus, having profiling functions to measure the cost of each MPI functions with respect to the target machine, is a very interesting facility. In particular, it could be particularly useful to have function costs that represent just the execution time for the function code, without having to consider additional time due to operating system multitask management. At the same time, it is important that these costs are defined using an appropriate time unity. We think that a time unity such as microseconds can suffices to measure significant cost values. Finally, we think that the ability to mark interesting part of codes and to measure global execution time of the marked parts could result particularly useful in determining costs of sequential user code defined within P^{3}L templates. As we stated, MPI functions costs and sequential code costs are used to instantiate performance analytical model of P^{3}L templates that allow to define optimized template implementations on the physical target machine.

3 P^{3}L language and support

P^{3}L is a structured parallel programming language whose goals are to allow easy programmability and efficient portability of applications on different architectures. Fundamental for this kind of approach are: the use of a set of structured parallel constructs, i.e., the parallel language; a set of compilation tools which exploit the structured characteristics of the language and guarantee the efficient portability of the applications; an abstract machine, which represents the fundamental mechanisms and characteristics of the underlying architecture.

The parallel constructs which P^{3}L includes are associated to well-known forms of parallelism:

- **pipe**, which implements the *pipeline* parallelism;
- **farm**, which implements *processor farm* parallelism;

- **map**, which implements *data parallel* computing;
- **loop**, which implements *iterative* computations;
- **reduce**, which implements *global reduction* operations;

furthermore there is a skeleton suitable to express **sequential** computation.

The $\mathtt{P}^3\mathtt{L}$ compiler, for a particular target architecture, contains for each parallel construct an *implementation template*. An *implementation template* associated to a parallel construct is composed by: a network of processes, which code is defined by *implementation processes*; a network topology, which is defined by an *implementation mapping*; an analytical performance model. To each parallel construct could be associated more implementation templates, different as concerns the mapping characteristics, but with associated the same interaction scheme. For each implementation the $\mathtt{P}^3\mathtt{L}$ compiler is able to determine the right parameters, such as the parallelism degree and the process grain, based on the characteristics of the construct to implement and some analytical performance models associated with templates. In particular the compiler actions are grouped in three phases, to which correspond three distinct components of the compiler: *front end, middle end, back end.*

The front end parses the source code and constructs an internal representation by using a syntax tree, called *construct tree.*

The middle end fills-in the tree with information concerning with process names, mapping on the abstract machine and the type of data sent by processes. The logical mapping of the constructs is selected from a set of *mapping templates*, called *mapping library.* Each element of the mapping library is associated to a performance formula parametrized respect to the costs of mechanisms of the abstract machine.

The back end generates the executable code for the target architecture, by defining a physical mapping of processes and communication channels on the abstract machine. In particular the executable code is generated by using process templates. It is important to observe that the use of an abstract machine as target architecture allows to realize an implementation independent from the underlying architecture. Then, to achieve portability and efficiency on a new architecture it is necessary to implement only new process templates.

Therefore a fundamental step in the implementation of a P^{3}L compiler on a new architecture is the definition of the implementation templates, for each construct of the language, which allow to exploit the characteristics of the target architecture. This document contains a discussion on the design of the P^{3}L implementation templates on the MPI abstract machine and a description of the implementation of a prototype version of that templates. In particular, first section describes a subset of P^{3}L constructs which we implemented to study the behavior on the target architecture, and the following section describes design and implementation of the corresponding templates.

4 Implementation Templates

In this section, we describe the implementation templates on the MPI abstract machine for a subset of the P^{3}L constructs. The constructs we consider are: *pipeline*, *farm*, *loop*, *map* and *sequential*. For a description of the syntax, semantics and the compiling tool scheme of the constructs see [BDO$^+$95a].

As discussed before an implementation template is built-out of two kind of information. The mapping information (**mapping templates**), which define the parametric network of processes implementing the program constructs and its mapping on the abstract machine; the process execution information of processes in the network (**process templates**), that define the macros which contain the process code. Implementation templates are organized according to the SPMD model. In general, each node executes an SPMD program with the structure in Fig. 1. Here, we can distinguish three parts: **initialization part**, in which all the data structures are initialized; a process specific **computation part**, built-out as a case in which processes with different ranks include different process templates; a **termination part**, in which the MPI exit routine is called.

During initialization each process executes the following operations:

- initialization of `MPI` (`MPI_Init`);
- read the rank of each process in `MPI_COMM_WORLD` (`MPI_Comm_rank`);
- read the total number of processes activated in `MPI_COMM_WORLD` (`MPI_Comm_size`);
- creation of all the groups different from the `MPI_COMM_WORLD`.

After the initialization, each process executes a computation phase which is specific of each type of process. The computation phase, of a given process

```
main ()
{
<definitions>
int my_rank, np;
.....
MPI_Init(&argc, &argv)
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_SIZE, &np)
<groups inizialitation>
.
.
.
switch (my_rank) {
case:  0:  {
#include ``macros_process_1''
break;
}
case:  1:  {
#include ``macros_process_2''
break;
}
...
}  /* end switch */
.
.
MPI_Finalize();
}
```

Figure 1: General structure of a process template

P terminates when P receives a task containing a mark, the `end_mark`, distinguishing the last task to be computed. At the end of computation phase, the termination phase is executed. During termination the process exits from the MPI environment by calling `MPI_Finalize`.

P^{3}L implementation templates use non blocking communication routines to exploit parallelism among communication and computation, whenever it is possible. In fact, if the target architecture is equipped with routing coprocessors, non blocking communication routines can be implemented by threads residing on a routing coprocessor, so increasing parallelism. Another MPI feature that has been exploited in the implementation templates are persistent communication. Persistent communications allow the envelope of a series of messages of the same type to be sent once and for all to a lightweight process residing on a routing coprocessor. Persistent communication are particularly convenient for processes implementing `P`^{3}`L` templates; in fact, all processes work on streams of objects of the same structure, which can be stated once and for all via a persistent communication.

As we have already discussed in section 2.1, groups are logical collections of processes communicating each other by a **communicator**. Collective operations are defined between processes belonging to the same communicator. Then, in `MPI`, to make a collective communication between process $P_1 \ldots P_n$, requires to define a group including all processes $P_1 \ldots P_n$ and only processes $P_1 \ldots P_n$. In fact, if other processes not in $\{P_1 \ldots P_n\}$, are included in the group they are involved in collective communications to which are not interested. This kind of approach allows to implement more efficient collective communications because only processes interested to such as communication are involved by a call. In particular in `MPI` broadcast, or multicast, communications are executed by a call of the `MPI_Bcast` routine from all the involved processes, with the same parameters. The result of the call of this communication routine is broadcasting of data from the root to the other calling processes. We can observe, as already previously discussed, that this is clearely an SPMD programming style.

Finally, another important characteristic in the implementation of process templates concerns with the type of the data sent using `MPI` primitives. In fact, `MPI` needs that messages are of a known **datatype** to be sent and makes available a set of predefined datatypes. Predefined datatypes correspond to sequence of contiguous elements of the same type, e.g., integer or array of characters. To manage structured data `MPI` allows to define new datatypes corresponding to structured data. New `MPI` datatypes for new structured types are defined once after `MPI` inizialization and do not need to be redefined each time a message is sent. Then this method is more efficient than the other, such as pack-unpack of structured objects, which involve additional overhead to pack object each time a message is sent.

4.1 The Seq implementation templates

The implementation template corresponding to the **seq** construct has a unique process, called `seq`. The computation cycle of this process can be summarized as follows:

- receive a task to compute;

- execute the user code on the received task;
- send the computation result to a receiver process.

To do these operations, the process needs an input channel, to receive tasks, and an output channel to send out the results of computations. The mechanism we used to implement communications for this kind of process is based on the use of two process ranks, *sender_rank* and *receiver_rank*, which represent the sender and the receiver process ranks in the current communicator. In the implementation of `seq` process, we used derived datatypes to send structured data, based on the above considerations. The use of derived datatypes was possible because the structured types we send have a static structure, but, as we will see later, this is not true in general, that is there are cases in which it is impossible to use derived datatypes.

4.2 The Pipe implementation template

The implementation template corresponding to the `pipe` construct is composed by a sequence of computation stages. Different stages are connected by communication channels. We observe that each stage could be a sequential construct, or any other $\mathtt{P}^3\mathtt{L}$ construct. In the second case the stage will be implemented by an appropriate template associated with that construct, following the hierarchical composition rule on which $\mathtt{P}^3\mathtt{L}$ is based.

To manage input and output streams, a generic $\mathtt{P}^3\mathtt{L}$ program is defined as `pipe` with a data stream input stage, a computation stage and a data stream output stage, parallel construct are nested into the computation stage. Then, implementing all the others $\mathtt{P}^3\mathtt{L}$ construct requires the implementation of a `pipe` construct with, at least, three stages: a data stream input stage; a computation stage, in which the parallel construct is nested in; and a data stream output stage. In such a structure, the first and the last stage are particular *sequential* constructs which manage input and output operations. To these processes correspond two process templates `seq_no_input` and `seq_no_output`. `seq_no_input` is a process that does not have input stream, symmetrically, `seq_no_output` does not have output stream. The process template corresponding to the central stage of the mapping template, could be a *sequential* process, which, along with its own input and output stream management routines, contains the user defined code. Communication channels are implemented refering process ranks. In particular, each process will refer their sender and receiver processes by using two process ranks, `sender` and `receiver`. The communication routines we used in these processes are non blocking send and receive routines, based on efficiency considerations as discussed for the general structure of processes in section 4. Based on the same considerations of section 4 the task transmission technique we used is to send `MPI` derived datatypes.

4.3 The Farm implementation template

In the structure of the `farm` construct there are three main activities, corresponding each one to a particular process template:

- *tasks distribution*, made by an emitter process towards the worker processes;
- *computing*, made by worker processes;
- *results collection*, made by a collector process.

Beyond these fundamental activities there are two other things that processes have to do:

- *load balancing among workers*;
- *results ordering.*

Load balancing is performed by the `emit` process, *result ordering* by the `col` process. The `emit` process implements the task distribution activities. This process is composed by an *initialization* phase, a *computation* phase and a *termination* phase. In the inizialization phase, the emitter process sends a task to each worker. During the computation phase, it receives new tasks from the input channel and waits for SEQ_SYNC_TAG messages from workers. These messages contain the worker rank in the current communicator and signal that the worker is available for a new task computation. When the emitter process receives the SEQ_SYNC_TAG, from worker i, then it, sends back a new task to that worker (if any is available). During the termination phase, the emitter process waits until all workers have computed the last task. The `emit` process also marks each task with an integer incremental mark. This mark will be used by the collector to preserve the task sending order when they sending out the results. The worker process template, which realizes the *computing* activity, is implemented by the `seq_sync` process. This process is close to a `seq` process except that, at the end of each task computation, a SEQ_SYNC_TAG message is sent to the emitter process. The computation cycle of the `seq_sync` process is composed by the following three phase: *initialization*, in which `MPI` data structures are initialized; computation, that is a cycle in which the process receives tasks, executes the user code and on them sends out the results until a task holding the **end_mark** is received; *termination*, which contains the call to the `MPI_Finalize` routine.

Finally, the *results collection* activity is realized by the collector process template, that is the `coll` process. This process also realizes the *results ordering* operation, sending out the received tasks in increasing order respect to their incremental integer mark.

To implement communications the `emit` process needs n channels, one for each worker. In the same way, as we have seen in the previous section for the `pipe` construct, outgoing channels are implemented using the process ranks of the receiver processes. Also the communication routines which we used are of the same type as in the `pipe` case, based on the similar considerations about efficiency. Finally the method used to send tasks between process uses derived MPI_Datatypes because tasks have a constant structure and sender and receiver processes know that structure at compile time.

4.4 The Map implementation template

The `map` implementation template is similar to the `farm` implementation template we have seen in the previous section. In fact the map construct, gets an input task and decomposes it in independent sub tasks, that are executed in parallel and whose result is a single output data. The mapping template used for the `map` implementation template is as follows. As concerns the process templates, `map_emit`, `map_work`, `map_coll`, they are similar to the farm case, except that they have to manage the *data partitioning* operation that differs from the *task distribution* operation made by the farm construct process template. The data partitioning operation is done by worker processes: the emitter sends the entire task to each worker and each worker extracts its own data partition from the task. This choice has been made for efficiency reasons; in fact, in this way the task partitioning is made in parallel by workers. Then the entire task is sent out by the emitter together with a tag, `num_task`, which indicates the number of partitions that the worker have to execute. The result of the workers computation is sent to the collector together with the tag. The might be more than one tag sent by the emitter process because the `map` construct could be nested recursively. The tags will be used by the collector to extract, from the results received from each worker, the partition holding the result of the computation.

To implement these process and mapping templates, we used *non blocking persistent* communication routines as in the previous case. As we observed in section 4.1, derived datatypes can be used to send objects of structured types, if they have a static structure, that is, if they are composed by a constant number of bytes. In cases in which structured types do not have a static structure (consider a list) it is necessary to pack objects before sending them. In the previous sections, we observed that messages holding tasks have a static structure and we used, to implement the corresponding process templates, structured datatypes to send tasks. As concerns the `map` construct, we can observe that messages sent by the `map_emit` process do not have a constant structure. Then `map_emit` sends tasks to the worker processes by messages holding a tag and a list of tasks, different for each worker. Such a structure does not allow to define a derived datatype, as we observed above. Then in this case we used the pack/unpack primitives to transmit tasks.

To distribute tasks, the `map_emit` process uses collective communications in particular it uses a multicast operation. Therefore, for reasons discussed in section 4, we used groups to implement this operation, in particular we defined a group, called **emit group**, containing the `map_emit` process and the `map_work` processes. When the `emit` process needs to send tasks to the workers it refers to the emit group, on the other hand worker processes refer this group to receive tasks.

4.5 The loop construct

The `loop` construct allows iterative computation of tasks until a particular condition becomes true. The logical structure of this construct, is composed by three stages: *input*, *body* and *output*. Tasks are received at the input stages, sent to the body, computed and sent to the output stage. Tasks reaching the output stage could be sent out to the output stream or sent back towards the input stage, by using a feedback channel. A fundamental task for processes implementing this construct is that they have to guarantee parallel execution of both tasks arriving from the input stream and partial results arriving the internal feedback channel.

The process template associated to the loop construct includes two processes, `loop_in`, `loop_out` and a stage, containing the body of the construct. `loop_in`, `loop_out` perform the input/output activities, respectively. The body stage could be a `seq` stage or any other P^{3}L construct. The `loop_in` process has two input channels, from the input stream sender process and from the `loop_out` process. On the other hand, the `loop_out` process has two output channels, towards the `loop_in` process and towards the output stream process manager. All these channels are implemented by naming the process ranks of, respectively, sender and receiver processes.

The `loop_in` process starts the computation receiving a task from the input channel and sending the task to the nested body construct. Then, it starts a cycle in which it tests its own two incoming channels. This operation is implemented by a call to the receive routine specifing a **wild card** as the receiver process rank. Tasks received from the feedback channel are immediately sent to the nested construct, while tasks received from the input stream are associated with an ordering tag. On the other hand, the `loop_out` process, manages the termination phase. When receives a task, it calls a predefined function in which the $\mathtt{P}^3\mathtt{L}$ compiler has included the user code that determines the termination condition. Tasks verifying the termination condition are stored into a buffer and sent out following their ordering number, tasks that does not verify termination condition are sent, by the feedback channel, to the `loop_in` process. Task transmission is performed in the same way as in the `pipe` construct (see sec. 4.2), by using `MPI_Datatype`; the communication routines used in this case are *non blocking persistent* `MPI_Send/Recv`.

5 Conclusions

This document summarize a preliminary study for implementing $\mathtt{P}^3\mathtt{L}$ on `MPI`Ȧccording to the document structure, we can distinguish two different kind of considerations resulting by our study.

In this first part, we analyzed `MPI` standard mechanisms that seems to be suitable to realize the $\mathtt{P}^3\mathtt{L}$ abstract machine. These mechanisms are:

- *virtual topologies* that allow to define processes networks that directly reflect $\mathtt{P}^3\mathtt{L}$ virtual processor structure;
- *wild-card* values, used for source and tag parameters in point-to-point communication functions, that can be used to implement many-to-one asymmetric communications;
- *no strict type matching* between send and receive buffers that allows to communicate data of variable size;

- *non blocking standard* point-to-point communication functions that allow to overlap communication and computation phases without having to consider any explicit buffer management;
- *persistent* communication functions that allow to optimize communication when the same communication pattern is repeated;
- *collective* operations that allow to easily express complex communication patterns;
- *derived data types* that allow to pass complex data structures among processors minimizing the overhead to copy data buffers and improving readability;
- *count parameter* to be used in communication functions and *packing/unpacking* functions that allow to communicate variable data.

In the second part of the document, we discussed the design and implementation of some prototype $\mathtt{P^3L}$ templates on the `MPI` abstract machine. In particular we, briefly, discussed the $\mathtt{P^3L}$ approach to parallel programming and discussed the implementation templates for some of the fundamental $\mathtt{P^3L}$ constructs, that is: `seq`, `pipe`, `farm`, `map` and `loop`. Then we presented the general structure of the processes and discussed the implementation choices we made.

As `MPI` standard seems to be thought for an SPMD programming model and programming in a SPMD model in `MPI` is more efficient, we realized processes implementing the $\mathtt{P^3L}$ constructs as SPMD processes in which different sections of code are included by conditional macros. As concerns the communication model, we observed that using the `MPI` *non blocking* communication routines to implement $\mathtt{P^3L}$ implementation templates, could be very efficient. Another important feature of `MPI` to implement the $\mathtt{P^3L}$ process templates are *persistent* communications, in fact, they can contribute to reduce the overall communication time, by reducing the start-up time of a single communication routine call.

Groups are an important feature of `MPI`, in particular in `MPI` groups are used to support efficient collective communications, such as broadcast, multicast, gather, scatter. We exploited this characteristic to implement collective communications between processes implementing the $\mathtt{P^3L}$ process templates. In particular we grouped *worker* processes, in the `map` construct, together with the *emit* process, into the same group to implement multicast operations between them. Finally, we observed that `MPI` derived datatypes are useful to efficiently implement task transmission between processes. The fact that we could build a derived datatype allows us to construct this kind of object once at initialization phase and then to send objects of this type without additional computation overhead at sending time. We used `MPI` derived datatypes to send tasks in every case it is possible, that is in cases in which the task structure is regular and static. On the other hand, in cases in which object to send does not have a static structure, such as lists of objects, it is not possible to use derived datatype. When the use of derived datatypes is not possible, `MPI` allows to send objects by packing it. We used the packing/unpacking technique to implement the `map` construct for which messages holding tasks sent between the workers and the collector does not have a static structure.

References

[BDO+95a] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a Structured High-level Parallel Language and its Structured Support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[BDO+95b] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. Summarising an experiment in parallel programming language design. In *Proc. of High Performance Computing and Networking, International Conference and Exhibitions*, Milano, May 1995.

[BRV91] G. Burns, Daoud R., and J. Vaigl. CHIMP concepts. Technical report, Edinburgh Parallel Computing Centre, University of Edinburgh, 1991.

[DDMO+92] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A Methodology for the Development and Support of Massively Parallel Programs. *Future Generation Computer Systems*, 8(1-3), July 1992.

[DOSW95] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. An Introduction to the MPI Standar. Technical Report CS-95-274, University of Tennessee, January 1995.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users'*

Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994.

[Har93] J. C. Hardwick. Chameleon parallel programming tools user manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.

[Mes94] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994. Also available as Technical Report CS-94-230, Computer Science Dept., University of Tennessee, Knoxville, TN, 1994.

[Mes95] Message-Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995. Version 1.1.

[MPI92] CHIMP version 1.0 interface. Technical report, Edinburgh Parallel Computing Centre, University of Edinburgh, May 1992.

[Pac95] Peter S. Pacheco. A User's Guide to MPI. Technical report, University of San Francisco, Department of Mathematics, March 1995.

[Pel93] S. Pelagatti. *A methodology for the development and the support of massively parallel programs.* PhD thesis, Dipartimento di Informatica, Università di Pisa - Italy, March 1993.

[SGDM94] V.S. Sunderam, G.A. Geist, J. Dongarra, and R Manchek. The PVM Concurrent Computing System: Evolution, Experiences and Trends. *Journal of Parallel and Distributed Computing*, 20, 1994.