BIP Messages User Manual for BIP 0.94

Loïc PRYLLI

June 8, 1998

Abstract

BIP stands for Basic Interface for Parallelism. It is a message-passing system implementation on top of Myrinet. It achieves one Gigabit/s bandwidth and less than 5 μs latency. This manual describes the whole BIP API, as well as the tools used to compile and run BIP programs on a Myrinet platform. The BIP API definition is still evolving in function of user requirements. Still backwards compatibility has always been maintained until now. The last version of the manual can be found at: `http://www-bip.univ-lyon1.fr/bip.html#doc`.

Contents

1 Introduction

Our objective is to provide for the Myrinet [BCF+95, Myr95, SBSS93, LS92, FDCF94] network a basic interface for message-passing, targeted towards parallel computing. Of course there are standard API well established: MPI [SOHL+95] and PVM [GBD+94], but to begin with we choose to implement our own simpler interface: BIP (for Basic Interface for Parallelism). One of BIP main goal is to really achieve the maximal performance of the hardware, but with a sufficient abstraction level, to hide the hardware details.

Currently BIP messages has only been implemented for cluster of x86/Linux workstations, linked by Myrinet boards with the LANAI4.1 processor.

The base principle of BIP messages is to implement all communication in a user-level library with zero-memory copies and direct access the network hardware without system calls. Our current implementation achieves less than 5 microseconds latency for small messages and more than 125Mbytes/s bandwidth (so more than 1Gigabit/s if you take Giga=10^9).

2 BIP message-passing model and semantics

2.1 Overview of the BIP API

The BIP API consists in several variants of send and receive primitives that should meet any need, with both blocking and non-blocking versions.

BIP ensures reliable and ordered transmission of messages in the absence of network fault (cf. 2.2).

It is an error to send a message to yourself.

All messages consist of a contiguous array of words (multiple of 4 bytes, properly aligned in memory). Buffers can be located anywhere in the process address space: either “global” data, or data allocated with `malloc`, or data on the stack.

2.2 Reliability

In general a Myrinet network is reliable enough so that error recovery is not strongly necessary. Still sometimes a network error can happen, in this case it will be detected. Either an application terminates correctly, or it may be aborted in case of network error. In any case, BIP ensures that no messages can be silently lost and no corrupt message can arrive to the application.

Implementing error recovery is ongoing work. If the gain is worth, we will make it a compile-time option, because from our experience, we have never been able to show evidence of a network error on our platform.

2.3 Quick start example

Here is a small example before going into the details of BIP:

```
#include "bip.h"

int main(int argc,char*argv[])
{
```

```
  int token;

  bip_init();
  if (bip_mynode == 0) {
      token = 333;
      printf("token start on 0\n");
      bip_send(1,&token,1);
      bip_recv(&token,1);
      printf("token arrived\n");
  } else {
      bip_recv(&token,1);
      printf("token %d received on %d\n",token, bip_mynode);
      bip_send((bip_mynode+1)%bip_numnodes,&token,1);
  }
  return 0;
}
```

This example can be run on 3 machines (here lhpca, lhpcb, lhpcc) by doing:

```
% bipcc token.c -o token
% bipconf lhpca lhpcb lhpcc
% bipload token
token start on 0
token 333 received on 1
token 333 received on 2
token arrived
%
```

2.4 BIP "short" and "long" messages semantics

BIP calls have some semantic differences between short and long messages, the same send and receive calls are used for both, so in simple cases, the user can ignore the semantic distinctions.

"long messages" sends and receives have a rendez-vous semantic where a receive need to be posted before or at least "not too long" after the matching send has begun. This is a requirement similar to the "ready send" mode of MPI. It is in fact a bit more permissive, the precise restriction is that the receive should be posted no longer than about *50 ms* after the send (after which a message blocked on the network could be dropped in some cases), but note that the receive should *preferably* be posted before or "very soon" after the send to avoid blocking communication paths in the Myrinet network (which could severely affect performance of other communications).

On the opposite "short" messages are stored into an circular queue, so that the send calls will not block even if no matching receive has been posted, (except if the destination receive buffers are full, the amount of buffering can be controlled with `bip_taginit`).

The limit between "short" and "long" messages is specified by `BIPSMALLSIZE` (and depending on the release is between 100 and 400 bytes).

2.5 BIP queues

Each message is sent to a particular queue of a particular node. Each different queue on a node is identified with a *tag* between 0 and `NTAGS-1` (`NTAGS` is 200 in the current release). The "token" example shown above use the default queue by not specifying any `tag`.

Communication buffers for small messages are allocated at initialization for each receive queue, you can override the default amount of buffers independantly for each queue before the call to `bip_init` with the primitive `bip_taginit`.

As small messages are buffered on reception, it is not mandatory for the user to receive these messages in the sending order. On the contrary, for a long message, the application must be ready to receive such a message when it is sent, and messages for other queues cannot be received before the user has provided a buffer for such a long message.

On one process send and receive functions are completely independent. At one time, you can have at most one send call in progress, and one receive call per tag posted. Receives on different tags, and a send can be done at the same time in different threads and are thread-safe (but note that blocking calls will not automatically generate a thread switch). Two receives with the same tag, or two sends cannot be done concurrently by two threads.

3 Definition of BIP variables and functions

Each C module using BIP must include the file `bip.h`.

3.1 Basic primitives and variables of BIP

`extern int bip_mynode,bip_numnodes`

respectively give the logical number of the current process and the total number of processes.

`void bip_init(void)`

should be called once before any other BIP primitive except `bip_taginit`, and will initialize the system. Note that `bip_mynode` and `bip_numnodes` are only valid after this call.

`int bip_tisend(int dst,int tag,int *buf, int length)`

this is an asynchronous tagged send call, `length` is the size of the message to be sent in *words* (not bytes), `buf` points on the message data, the destination node number is given by `dst`, the message is sent to queue `tag`. This function returns immediately but the buffer should not be changed until the send has completed as ensured by `bip_swait`, in the meantime you cannot do any other send or isend communication calls.

`void bip_swait(int id)`

waits for the last send request to complete, it requires as argument the value returned by the last `bip_tisend`. When the function returns, the buffer can be used for something else, and a new send call can be done.

`int bip_stest(int id)`

tests if the last `bip_tisend` has completed, takes the `id` returned by isend. Returns 1 if the buffer can be reused, 0 otherwise.

`int bip_tirecv(int tag,int *buf, int maxlength)`

this is an asynchronous receive call. `tag` identifies the receive queue on which this call applies, `buf` gives the target buffer where the message will be stored at completion, `maxlength` is the maximum size the buffer is able to receive. The first message that arrives (or has already arrived) for this queue from the network will go into this buffer, it is an error if a message longer than this size arrives. The buffer contents are invalid until completion (as told by `bip_rwait`), and should not be modified until that. This function returns an identificator that should be passed to `bip_rwait`. This function returns immediately but you cannot do any other recv or irecv call with the same `tag` until the receive has completed (so after calling `bip_rwait/bip_rtest`).

`int bip_rwait(int id)`

waits for an asynchronous receive to complete, it requires as argument the value returned by the corresponding `bip_tirecv`. It returns the size (in words) of the message that has been received and stored in the user buffer.

`int bip_rwaitx(int id,int *node)`

same thing as `bip_rwait` but it returns the source node of the message in the variable pointed by `node`.

`int bip_rtest(int id)`

similar to `bip_rwait`, but if the reception has not completed, it does nothing and returns `-1`. It never blocks.

`int bip_rtestx(int id,int *node)`

similar to `bip_rtest` expect that if the receive has completed, it also gives the source node of the message in `node`.

`int bip_tprobe(int tag)`

if a *short* message can be received without blocking, the call return a non-null value else it returns 0. It cannot be used to detect the presence of a *long* message (will always return `0` if a long message is the first to arrive).

`int bip_tnrecv(int tag, int *buf,int maxlength)`

It is a non-blocking receive, if a *short* message is ready on the queue given by `tag`, it does not block and is equivalent to `bip_trecv`, else it is a “no-op” call and returns `-1` (not zero because that is a valid message size). *Warning: you* *cannot receive a* long *message with this call: it will always return -1 if a long message is arriving to the head of the queue*

`int bip_tnrecvx(int tag, int *buf,int maxlength,int *node)`

similar to `bip_tnrecv`, expect that if a message is received, the source of the message is also returned via `node`.

`int *bip_tgetmsg(int tag, int *length, int *node)`

This function receives a message on queue `tag` and returns a pointer to the contents (the message is in a statically allocated buffer), the message size is returned via the parameter `length`, and the source of the message is returned in `node`. When the user has finished using the message contents, it must free it explicitely via `bip_tfreemsg(tag)`.

`int *bip_tngetmsg(int tag, int *length,int *node)`

similar to `bip_tgetmsg` except it is non-blocking. If no message is available, it returns a null pointer and has no side-effect.

`void bip_tfreemsg(int tag)`

frees the older buffer received by `bip_tgetmsg` or `bip_tngetmsg` on queue `tag`. There is a static circular queue of buffers for each `tag` and buffers of a queue are freed, in the order they were received. You can receive several messages via `bip_tgetmsg` before freeing them (in the limit of the buffer queue for the corresponding `tag`. You can not do any other kind of receive call on this queue (such as `bip_trecv`) before having freed the buffers of all messages received via this call for the corresponding `tag`.

`int bip_taginit(int tag, int nbufs, int size)`

Allows to override the default buffer size allocated to the queue specified by `tag`, the buffer will be able to store `nbufs` messages of length `size` without any receive occuring. If the message are smaller, they can be more of them. But note that you have to take into account for each message not only the payload but also a few words (currently 4) of head information for internal purpose. This procedure can be called at most one time for each queue, and it must be called before the `bip_init` function. This procedure is optional, default values are taken, if not explicitely defined.

3.2 Blocking calls

The following functions can be semantically defined in terms of the previous one. They are provided both for convenience and also because their implementation is sometimes faster that using the basic primitives of BIP.

`int bip_trecv(int tag, int *buf,int maxlength)`

blocking receive, semantically equivalent to `bip_rwait(bip_tirecv(tag,buf,maxlength)`.

`int bip_trecvx(int tag, int *buf,int maxlength,int *node)`

blocking receive, semantically equivalent to `bip_rwaitx(bip_tirecv(tag, buf,maxlength)`.

`int bip_tsend(int dst, int tag, int *buf,int length)`

blocking send, semantically equivalent to `bip_swait(bip_tisend(dst, tag, buf,length)`.

3.3 Non tagged message

There exists an "untagged" version of all BIP calls to maintain compatibility with previous versions, and for convenience for applications that just need one queue.

The functions are:

`bip_isend(int dst, int *buf, int length)`

equivalent to `bip_tisend(dst,0,buf,length)`

`int bip_irecv(int *buf, int maxlength)`

equivalent to `bip_tirecv(0,buf,maxlength)`

`int bip_nrecv(int *buf,int maxlength)`

equivalent to `bip_tnrecv(0,buf,maxlength)`

`int bip_nrecvx(int *buf,int maxlength,int *node)`

equivalent to `bip_tnrecvx(0,buf,maxlength,node)`

`int *bip_getmsg(int *length, int *node)`

equivalent to `bip_tgetmsg(0,length,node)`

`int *bip_ngetmsg(int *length, int *node)`

equivalent to `bip_tngetmsg(0,length,node)`

`void bip_freemsg(void)`

equivalent to `bip_freemsg(0)`

`int bip_probe(void)`

equivalent to `bip_tprobe(0)`

`int bip_recv(int *buf,int maxlength)`

equivalent to `bip_trecv(0,buf,maxlength)` `int bip_recvx(int *buf,int maxlength,int *node)`

equivalent to `bip_trecvx(0,buf,maxlength,node)`

`int bip_send(int dst, int *buf,int length)`

equivalent to `bip_tsend(dst, 0, buf,length)`.

3.4 Comments

Note that a `bip_tirecv` *must* be followed by a `bip_rwait/bip_rtest`. On the other hand, the `bip_swait/bip_stest` is not mandatory after a `bip_tisend`, but you should not reuse the buffer or issue another send before you are sure the message has been received at the other side (for example because you have received an answer).

Note that in the current implementation, passing a value of `maxlength` much larger than the final message length can have a significant impact on performance.

4 BIP utilization

4.1 Note for MPI

The scripts describes below are used for pure BIP programs as well as for MPI programs, when using the MPI-BIP implementation.

4.2 Warning

As BIP relies on direct access to the hardware, bugs in either the BIP library or in programs using BIP may crash the system, corrupt the system memory, so *use BIP at your own risk*. We have tried to design BIP so that these problems are very unlikely to occur by accident, but as the Copyright says: *the software is provided as-is, without any express or implied warranty. In no event will the author be held liable for any damages.*

But still note we are confident enough to run regularly BIP programs on a machine that is the Web and mail server for our team, and have never experiment any crash.

4.3 General requirements

To use BIP you must ensure the Myricom IP driver (or the BIP IP driver) has been shutdown (with `ifconfig myri0 down` for instance, it must still be present so do not "rmmod" it). This must be done on all nodes you want to use for BIP, but BIP will work with no problem with other nodes running the IP driver on the same Myrinet network.

The scripts `bipload`, `biproute`, `bipconf` can be run from any machine even one that does not have a Myrinet board, but you must be sure that you can "rsh" from this machine to every BIP node. So you should have another network under IP in addition to the Myrinet network on the BIP nodes. Check either you have a suited `/etc/hosts.equiv` file or adapt your `.rhosts` file.

All Myrinet and BIP software should be available at the same location on every machine (for instance by NFS). The scripts also assume you have the same NFS home directory mounted on every machine accessible by the same path.

The Myrinet board used by BIP should be the unit 0 (so in fact the first `/dev/mlanaix` on which the user has permissions access).

If any one on these conditions are not fulfilled, look at the following section 5 to see how to configure and start a BIP application manually.

4.4 Configuration

To use BIP successfully, you just need to ensure that the directory of Myricom programs (probably something like `.../myrinet/bin/intel_linux`), and the directory with BIP scripts utilities (something like `.../bip/bin`) are in your `PATH` environment variable.

Eventually you can try to define the `MYRI_HOME` environment variable to the toplevel Myricom software directory if BIP scripts cannot find automatically the Myricom utilities and libraries, but this is normally not required.

The structure of the BIP distribution should not be changed, so that the scripts are able to find the files they need.

4.5 Creating the routing database

The first time you want to use BIP or each time the Myrinet network topology change you have to run the program `biproute` passing as arguments the names of the machines you may want to use, `biproute` will determine the Myrinet network sub-topology consisting of the machines you want to use as BIP nodes in an application (note that `biproute` relies on the Myricom tool `print_routes` and the Myricom MCP to accomplish this task).

The results of `biproute` will be stored in the file `$HOME/.bip/bipdatabase`.

Example: `% biproute lhpca lhpcb lhpcc lhpcd lhpcg lhpch`

4.6 Specifying a BIP configuration

Before running a BIP application, you need to specify a set of machines on which to run it with `bipconf`. This will create a configuration file read at run-time by each node. Just pass as arguments the names of the machines to use. The order of the arguments will determine the numbering of the processes in the BIP application.

Example: `%bipconf lhpca lhpcd`

`bipconf` store the configuration info in `$HOME/.bip/bipconf`, you can use the option `-f <file>` to store the information in an another file (in this latter case you will then have to pass the same option to `bipload` also, the rationale is to be able to keep some often used configuration files).

4.7 Compiling the application

The simplest way to compile an application is to use `bipcc`, which is a script that will take care of adding the appropriate "include" directory, library directory in the search path as well as adding all BIP and Myrinet libraries required, all arguments are passed as is to the real compiler.

For instance in the bip/test directory:
bipcc jeton.c -o jeton will create the executable jeton.

For MPI programs, just add -lmpi at the end of the compilation line at link time.

If you want to use directly the normal C compiler,

- add the .../bip/include directory to the include search path (option -I).
- add the .../bip/lib directory to the library search path (option -L)
- add the myrinet/lib/intel_linux directory to the library search path (option -L)
- link the program with the libraries -lbip, -ldio, -lLanaiDevice.

4.8 Running the application

Syntax: bipload [-sz <n>] [-f <configfile>] [-q] prog [args] ....

To start an application, use the script bipload, pass to it the name of the executable and the application arguments. If you want to use a different executable for each BIP node, you will have to start them manually on each machine in the configuration.

The option -sz <n> of bipload means the use of only the n first machines of the configuration defined with bipconf, so n must be less or equal to the number of machines specified with bipconf. The option -q means to suppress a few messages that are normally written at initialization and termination by the BIP library. The option -f <file> allows to specify explicitly the file describing the configuration of BIP nodes to use for the run instead of using the default file creating by bipconf: $HOME/.bip/bipconf.

When starting an application manually the effect of the option -q and -sz can be achieved by setting the environment variable BIP_OPTS. For instance BIP_OPTS=n5q is equivalent to having the option -q and -sz 5. The option -f is replaced with the environment variable BIPCONF.

5 Definition of the BIP configuration file format

You do not need to read or understand this section if the general requirements of the last section are fulfilled, but in case you have any problem, it could be useful to check the contents of the configuration file that have been automatically generated.

The configuration of a BIP application relies on a configuration file that must be read by every process of the application, this file is by default $HOME/.bip/bipconf, but can be overridden by the environment variable BIPCONF.

You may write manually the configuration file for some peculiar configuration, for instance using a unit different of 0 or if you have problems using the scripts biproute, bipload, and bipconf. The only strong restrictions of BIP is that two machines must be separated by at most 4 switches, and only one board can be used by BIP on each machine (this last restriction could be removed, but it has not been done yet as it seems unlikely to be really useful).

The configuration file is organized by lines:

- the first line contains the number of nodes n for the BIP application.
- Then for each node there is:
  - one line giving the hostname of the machine that will be this node and the unit of the Myrinet board to use for communication,
  - and n lines which are the routes in Myrinet topology to reach the other nodes, these routes consist in 1 to 4 numbers depending on the number of switches to cross. As a special case a line with a single `256` means no switch (and is also used to fill the route to itself).

So in summary a configuration file will contain $(1 + n * 1 + n)$ lines. Moreover all blank lines and lines starting with `#` are ignored and can be used for comments. See file `bip/examples/bipconf.ex` to see an example of a configuration file, file `bip/examples/bipconf.noswitch` is an example when using two nodes linked directly without any switch.

At BIP initialization (in fact in `bip_init`), a process first gets its BIP logical number by searching the name of the machine (as given by `uname -n`) in the configuration file, reads the following routing information, and initialize the `bip_mynode` and `bip_numnodes` variables. And then will execute a small synchronization algorithm to synchronize with the other nodes, so that the processes can be started in any order and with any delay between them.

6 Installation

Read the README file in BIP distribution to see how to install the "dio" kernel module on each machine.

Install the bip directories anywhere in your file-system and just add `bip/bin` directory to your search path. That's all if all goes well `:-)`.

References

[BCF+95] Boden, Cohen, Feldermann, Kulwik, Seitz, Seizovic, and Su. MYRINET: A Gigabit per second Local Area Network. *IEEE-Micro*, 15:29–36, February 1995.

[FDCF94] R. Felderman, A. DeSchon, D. Cohen, and G. Finn. ATOMIC: A high speed local communication architecture. *Journal of High Speed Networks*, 1994. IOS Press.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Mancheck, and Vaidy Sunderam. *PVM: Parallel Virtual Machine*. Scientific and Engineering Computation. MIT Press, 1994.

[LS92] Charles L. Seitz. Mosaic C : An experimental fine-grain multicomputer. Technical report, California Institute of Technology, Pasadena, CA, 1992.

[Myr95] Myricom. Myrinet link and routing specification, 1995. http://www.myri.com/myricom/document.html.

[SBSS93] Charles L. Seitz, Nanette J. Boden, Jakov Seizovic, and Wen-King Su. The design of Caltech Mosaic C Multicomputer. In *Proceedings of the University of Washington Symposium on Integrated Systems*, 1993.

[SOHL+95] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, 1995.