Abstract

Building complex SPMD code in an efficient and portable way is nowadays a challenge, especially when there is no uniformity of tools and libraries across platforms. The Fast Messages (FM) and the Portability Library (PL) were both designed to provide the basis of an abstract enough framework for C, so that problems can be coded and ported to any supported platform with no more than a few changes in the makefiles and a recompilation. The FM library provides a message passing communications library built around the Berkeley Active Messages library. The PL library provides the primitives for host to node communication for problem initialization and results collection, as well as other miscellaneous and potentially non-portable primitives. This technical report contains the documentation for both libraries.

1 Introduction

It is unrealistic to think that a complex framework coded in an SPMD paradigm can survive long enough to be useful, unless it is written in a portable way that allows it to migrate to different platforms. Massively parallel machines have a very short life span compared to other platforms and become obsolete — and no longer usable — in a few years. Even so, new ideas in algorithm design have to be coded and tested, because it is only the empirical evidence what makes them acceptable with a good degree of confidence. That is the motivation behind the development of the Fast Messages (FM) and the Portability Library (PL) libraries. Both of them put together provide the basis of an abstract enough framework in which to code message passing SPMD code in C.

Fast Messages (FM) is a communication library based on services provided by the Berkeley’s Active Messages (AM) library or any other similar library. It provides higher level primitives than those in the AM library, all of them intended for communication between processes (processors on parallel machines.) It provides facilities for sending and receiving unicast messages as well as a simple interface for dealing with errors that may occur during the user computation. As of this writing, the FM library has been implemented and tested under the following AM-like primitive versions, machines and operating systems:

- A TCP/IP version of Berkeley’s AM, that works over TCP/IP on Sun machines running Solaris. This version is not intended for performance — which it would not achieve — but rather, for testing purposes. A version of the program under development can be compiled and run with this version until the program is ready for a “production” version based on a real parallel machine.

- A CM-5 version of CMAML, that works on CM-5 machines running CMost.

The Portability Library (PL), which really started standing for Parallel/Portable Library, is a library of primitives for those services that, although provided in most or all systems, are achieved in different ways or through different system-specific function calls. By grouping all those functions together and assigning
them a single name across systems, portability can be preserved as soon as a version of the PL library is available. As of this writing, the PL library has been implemented and tested under the following machines and operating systems:

- An emulated version, that works on Sun machines running Solaris. This version is not intended for performance—which it would not achieve—but rather, for testing purposes. A version of the program under development can be compiled and run with this version until the program is ready for a "production" version based on a real parallel machine.

- A CM-5 version, that works on CM-5 machines running CMost.

The complete documentation for both libraries follows. The rest of this document is also online at the URL:

http://www.prolangs.rutgers.edu/software

where it will be kept up-to-date if there are changes in the future.
Fast Messages Communication Library

This document contains a complete description of the Fast Messages (FM) communications library. Along with a description of the library and an annotated list of primitives, there is a section on how to use the library and another on how to customize certain internal characteristics to suit your debugging needs or memory constraints. Two non-trivial example programs are also included as well as a references section. This document is written in a generic way and thus covers no specific version of the library.

INDEX OF THIS PAGE

- Fast Messages Communication Library
  - What is Fast Messages?
  - Description of the FM library
    - The Data Descriptor: ddesc
    - The FM primitives
  - Using the FM library
  - FM User Requirements and Error Reporting
    - User Requirements
    - Error Reporting
  - Debugging Options and Internal Characteristics
    - Debugging Options
    - Internal Characteristics
  - Example 1
  - Example 2
  - References

What is Fast Messages?

Fast Messages (FM) is a communication library based on services provided by the Berkeley’s Active Messages (AM) library or any other similar library. It provides higher level primitives than those in the AM library, all of them intended for communication between processes (processors on parallel machines.) It provides facilities for sending and receiving unicast messages as well as a simple interface for dealing with errors that may occur during the user computation. As of this writing, the FM library has been implemented and tested under the following AM-like primitive versions, machines and operating systems:

- A TCP/IP version of Berkeley’s AM, that works over TCP/IP on Sun machines running Solaris. This version is not intended for performance -which it would not achieve- but rather, for testing purposes. A version of the program under development can be compiled and run with this version until the program is ready for a "production" version based on a real parallel machine.

- A CM-5 version of CMAML, that works on CM-5 machines running CMost.
Similar versions of the Portability Library (PL) exist. See the documentation about this library for more information.

Also, it is expected that the library will run without changes (perhaps some minor changes) on any other system where the Berkeley’s AM library exists.

---

**Description of the FM library**

The FM library is a fairly simple library. It has only one data structure and eleven primitives.

**The Data Descriptor: ddesc**

This is the descriptor of the unit of data that will be transmitted between processes. A descriptor can be thought of as the way to encode a primitive object: it puts together a piece of information (the data), with a description of what that information means (the header.) Every such object, which constitutes a message, is assigned a type (the id), which is a convenient way of distinguishing different classes of messages in what respects to their meaning on the receiving end. The receiver of a message can ask for a certain type of message by using its assigned id on the sending side. Also, if there is no need for distinction, the receiver can ask for any type of message, thus bypassing the id mechanism.

This is what a Data Descriptor looks like:

```c
struct ddesc {
    byte2 id;
    void *header;
    ulong hlen;
    void *data;
    ulong dlen;
};
```

You should ignore the types of the fields. They will be defined under different architectures so that the fields have the appropriate size. id is guaranteed to be at least 16 bits unsigned; header and data to be valid pointers to any data size; and hlen and dlen to be at least 32 bits unsigned.

id should describe the type of the message.

header is a pointer to the header of the message, which will be hlen bytes long.

data is a pointer to the data of the message, which will be dlen bytes long.

More fields may be added to the structure in the future, so you should not assume that the structure is any size -rather than sizeof(struct ddesc)- or that the fields are a certain number, or arranged in a certain order.

Applications or other libraries that build on top of FM may use the ddesc descriptor as a way of tracking the type of their data.
The FM primitives

Here is a list of the eleven primitives in the FM library. A C prototype precedes each primitive description.

void fm_enable(void);

Should be called before any other FM functions, probably at the very beginning of the program execution, but always after the PL `pl_enable()` call.

---

void fm_disable(void);

Should be called after all other FM functions for a graceful ending, probably right before the end of the program execution, but always before the PL `pl_disable()` call.

---

int fm_procs();

Returns the number of processes running.

---

int fm_my_proc();

Returns the number of process the calling process is, between 0 and fm_procs()-1.

---

int fm_poll(void);

Should be called within any loop of computation. Must be called within any loops waiting for an event that requires a network operation. No program should allow a long computation without calling `fm_poll()` periodically if out of band communication is expected. The return value depends on the node that makes the call. A value of zero signals that no error occurred during the communication. A value greater than zero is only returned by node zero. That means that one or more error reports were received from the other nodes during the poll and the worst severity of those is returned. A value less than zero is only returned by any node other than node zero. It signals a request from node zero to terminate.

---

void fm_terminate(int severity);

Request for termination. Should be called only from node zero. All other nodes will receive the "termination request" by the return value of their next call to `fm_poll()`. The interpretation of the severity of the request is application dependent. It should be higher for worse errors and typically the same as the severity of the error that originated the request. The action taken could range from immediate termination to termination after all local pending computations are ended and reported.
struct ddesc *fm_descriptor(void *header, ulong hlen, void *data, 
ulong dlen, byte2 id);

Creates a "data structure" descriptor (necessary for sending it.) A data structure is considered to be two 
parts (possibly) laying in non-contiguous memory locations: header and data. The meaning of those 
parts is application dependent. id is a user defined identifier. It can be used on the receiving end to 
identify the data structure. Returns a pointer to the descriptor. Deallocation of such a data structure 
should be done with a simple free.

void fm_free(struct ddesc *dd);

Free the memory allocated to a "data structure" descriptor. Should be used to free a ddesc pointer (and 
all it points to) that was returned by an fm_receive(...) call.

int fm_send(vnn_t destnode, struct ddesc *desc);

Sends a "data structure" to another node. The structure descriptor has to be created before, using 
fm_register(...). Returns non-zero in case an error occurred during the transmission. Since 
congestion is the most likely cause of the error, multiple retries could be attempted. This call does not 
return until the message is totally received by the destination node or an error occurs. The message will 
be queued at the destination, waiting for an fm_receive(...) call.

struct ddesc *fm_receive(byte2 id, int wait);

Receives a "data structure" identified by its id from some other node (not directly specified.) If id is 
zero, any structure will be returned if one is available. In this case, structures are returned in the order 
they arrived. A poll is done to ensure that no pending messages are missed. If the parameter wait is 
non-zero, the call will block until the requested data structure is received, in case it has not already been 
received before. Returns a pointer to the data structure descriptor. A null pointer signals an error or the 
data structure not available (for the no-wait option.) Note that an error is possible even in the case of the 
"wait" option.

void fm_error(char *errormsg, int severity);

Signals an error to the network. It can be invoked from node zero. Node zero will receive the "error 
request" by the return value of the next call to fm_poll(). The interpretation of the severity of the error 
is application dependent. It should be higher for worse errors; 1 is the starting number for the least 
severe error. 0 can be used to signal an event that is not an error but is to be logged as soon as possible 
in some versions of FM, at the end of the computation in others; for example, for debugging purposes. 
In all cases, the calling node information will be included in the actual message reported, including a 
time stamp.
Using the FM library

To use the FM library, you need to include the file `fm.h` before any FM function call. The best way to do that without having to change source files for all different versions across platforms, is to include the line:

```c
#include <fm.h>
```

and then make sure that your `makefile` includes a definition like:

```bash
CFLAGS=-I<fm-directory>
```

Where `<fm-directory>` is the directory where the file `fm.h` is. That directory may also have to include the file `am.h` for the AM definitions.

If you are invoking the C compiler directly from the command line, you should add the option `-I<fm-directory>` to the invocation line.

Also, you will need to link your object code with the library `libpar.a`. For that purpose, you can add a line like:

```bash
LIBS=-L<lib-directory> -lpar
```

in your `makefile`, and have all your linking command invocations include it by adding `${LIBS}` as appropriate. Once more, if you are invoking the C compiler directly from the command line, you should add the option `-L<lib-directory>` to the invocation line.

Be aware that many compilers will not pass on the linking options to the linker correctly, unless they appear after the source or object file names that you are also including. It is always safe to include the linking options last (or right before the `-o <target>` if you are naming your target file.)

It has been so far customary to have a single directory, `libpar`, include all the parallel machine-specific files and the single library `libpar.a`. That directory includes all FM and PL (see the PL library documentation) specific files, and maybe the sources and the `makefile` to generate the `libpar.a` library file. In that case, and assuming that your `libpar` directory is `~/libpar` the options above become:

```bash
CFLAGS=-I~/libpar
LIBS=-L~/libpar -lpar
```

and similarly for the command line invocation options.

---

**FM User Requirements and Error Reporting**

**User Requirements**
As it has been noted in the primitive description above, the FM library requires that the user calls \texttt{fm_enable()} before using any of the FM functions. All processing nodes have to make the call, even if they will not be involved in any communication. That may happen if you want to use less processing nodes than you have been allocated.

Also, the FM library requires that the user calls \texttt{fm_disable()} before any processing node terminates. All processing nodes have to make the call, even if they were not involved in any communication.

Both the \texttt{fm_enable()} and the \texttt{fm_disable()} functions perform an implicit synchronization, so that all processing nodes will return from the calls at about the same time. Executing \texttt{fm_disable()} early will effectively block the processing node until the rest of the nodes are done.

**Error Reporting**

The FM library is intended to work even in those environments where message logging is not possible - or would be too slow - during the user computation (between the \texttt{fm_enable()} and the \texttt{fm_disable()} calls.) For that reason, warnings (severity zero errors) and errors may or may not be reported as they occur. In order to provide some error trace-ability, error messages are queued in node zero during the progress of the user computation. When the computation ends, they are all sent over to the front end (if applicable) and printed there. This is a sample error log:

```
FM messages:
  5.830000: [0] - n00: warning at node 0
  6.850000: [0] - n00: warning at node 0
  7.120000: [2] - n02: error at node 2
  25.590000: [0] - n00: messg: 160s/110r; bytes: 9122s/6325r; warns 0
  4.059999: [0] - n01: messg: 75s/122r; bytes: 4263s/6960r; warns 0
  24.340000: [0] - n02: messg: 127s/130r; bytes: 7242s/7342r; warns 0
<end of messages>
```

The first fractional number indicates the busy time (may be the elapsed time in some implementations) in seconds at the reporting node, since the computation started (call to \texttt{fm_enable()}).

The second number, between square brackets, contains the severity of the message. A severity of \texttt{[0]} means that the error is really a warning or some other condition that FM or the user program wanted to report.

The third number, preceded by the character "n", is the node at which the condition occurred.

The rest of the line from the colon is the message reported by the node.

Note that only the first \texttt{MAX_MSGLEN} characters of an error message are reported and that only the first \texttt{MAX_MSG} error messages are reported. See the next section to find out what those limits are and how to change them.

The last three (in general as many as processing nodes) messages are an example of the zero-severity errors generated automatically by FM when the FMPROFILE option is on. See the next section for details.
Debugging Options and Internal Characteristics

Debugging Options

A number of different options have been embedded in the FM code, to suit different debugging needs and internal data structure sizes. They are all located at the beginning of the file fm.c, which contains some or all of the code for the FM library. Depending on the platform for which the FM library is compiled, the options provided may or may not make sense. That mostly depends on whether the `printf` function is available or not at the processing nodes. Special care has been taken to isolate the handler specific prints from the other ones, in case `printf` cannot be used inside handler functions.

These are the possible options (they are all activated by defining the macro with the name described):

WARNINGS
If defined, send warning messages to node zero. That applies to all internally generated warning conditions, which have to do with recoverable errors like out-of-queue-space situations or transmission errors. If all errors returned by the FM functions are covered and you are sure that your queues are big enough for your needs and you will not run out of memory because of queued messages that are waiting to be read, you can leave this option off. It may be useful to have it on at all times, both to be aware of network errors that were corrected by the FM library or your program, or in case you have one or more of the queue sizes too small for your needs.

FMPROFILE
If defined, gather and report statistics about node communication. The total number of messages and bytes sent and received by each node, as well as the number of internal warnings generated, is computed and reported as a severity zero message during the call to `fm_disable()`. These messages, once reported, will look like:

```
FM messages:
  25.590000: [0] - n00: messg: 160s/110r; bytes: 9122s/6325r; warns 0
  4.059999: [0] - n01: messg: 75s/122r; bytes: 4263s/6960r; warns 0
  24.340000: [0] - n02: messg: 127s/130r; bytes: 7242s/7342r; warns 0
<end of messages>
```

There will be as many lines as processors (three in this example.) First the number of sent and received messages is reported, in this order. Then the number of sent and received bytes, in this order. Then the number of warnings generated. This count is specially useful in case you disable the WARNINGS option above. Even when the warnings are not logged, at least they will be counted. If a count other than zero is detected, you should probably activate the WARNINGS option to find out what kind of problem is causing it. The information may prove useful if you are playing with the internal queue sizes, for instance.

DEBUG_MSG
If defined, log the message generation at the point where it happens. This may be useful if your program is running out of error message queue space or crashes before the error log is generated. May not work if your environment does not allow logging from the processing nodes while the computation is being performed. Always useful for preliminary testing with emulated versions.
DEBUG_TRACE
If defined, print trace messages, but do not trace handlers. This option will generate a complete trace log from all your FM calls. More than one trace message will be generated per call in those cases where interaction with another node is necessary. This trace should be useful when looking for the causes of deadlocks in the user code. Note that no trace is generated for handlers, since printing from them may not be possible in certain environments. You can trace them separately with the following option. A raw hex/ascii dump of the transmitted data is also possible; see below.

DEBUG_HANDL
If defined, trace handlers. You can activate this option if your environment allows prints from handler code. This should be true at least in the emulated versions.

DEBUG_DUMP
If defined, dump sent and received Data Descriptors, and received messages. That could be useful if you are uncertain of your code and want to check what the receiving end is really receiving (and very useful if you are porting the FM library to another architecture...) The dump is done in the typical hexadecimal/ascii format, ten bytes per line. Unprintable characters are displayed as a dot (.) -codes less than 32 or greater than 126. On the sender side, the associated ddesc is dumped; on the receiving side, the ddesc (which should be the the same as the one sent if the communication is working), the header and the data memory blocks are dumped. The DEBUG_TRACE option should be activated for this option to take effect.

The way used so far to choose the desired set of options, has been to comment or uncomment the #define lines at the beginning of the fm.c file. Though a little ad-hoc, this approach is fast and effective. If modifying the FM code bothers you (not until you have port it), you can comment all the #define lines out, and then choose the ones you need with the invocation options that you pass the C compiler. The option -D<MACRO> will define the macro MACRO as if it were defined in the first line of your compiled code. You can add as many of these as needed either in your makefile (in the definition of the CFLAGS variable) or directly in your C compiler invocation line.

Internal Characteristics

The FM library has a number of internal queues, both to store data to be sent or data received and not yet processed by the user. Also, one of the eight guaranteed PL timers is used for logging purposes. All the internal queues have to be static to avoid unnecessary overhead when accessing them and -specially- to avoid having to call any heap manipulation functions, since some of the accesses are done from handler code that may have been invoked from interrupt code.

The exact explanation of each macro follows. Note that each macro name is followed by its recommended value in parenthesis and that the queue sizes are one more than the actual number of elements they may hold at once:

PL_TIMER (7)
PL timer number to use for time logging. One of the eight guaranteed timers provided by the PL library is used for time logging. The rest of the timers are free for the user to use. Nevertheless, you may want to read this timer to find out the busy (most likely) time since the fm_enable() call started the FM library.
MAX_GET (41)
Maximum number of pending gets at a given node. This is the number of messages that may be pending to be received in the next fm_poll() call. This value is not critical unless your Active Messages version uses interrupts to invoke handlers. If that is the case, this is the maximum number of pending receive requests at any given time. Make sure you make this value big enough (the total number of computation nodes is big enough) so that it is never exceeded. Otherwise, expect communication errors.

MAX_GMS (41)
Maximum number of pending message gets. This is similar to the previous, but applies to error message reporting to node zero.

MAX_INQ (401)
Maximum size of the input message queue. This is the maximum number of messages that may be queued while they wait for an fm_receive(...) call. Note that calling fm_receive(...) for specific id values may leave messages with other id values waiting for a long time.

MAX_MSG (201)
Maximum number of pending error messages to node zero or (at node zero) maximum number of error messages to queue. The latter value is the critical one. Up to the first MAX_MSG-1 error messages will be logged. Any subsequent error messages will be lost.

MAX_MSGLEN (80)
Maximum error message length. Up to MAX_MSGLEN-1 characters will be reported from every error message (a null character is needed at the end of each message.)

You can modify any of the internal characteristics by changing the #define line to the appropriate value. For some of the queue sizes, you may have to do a little of experimentation if you want to set them to the minimum possible value. All the values suggested are very conservative and thus, quite wasteful in terms of memory.

---

**Example 1**

Here is an example of a client/server scheme. Node zero generates MESSAGES requests that are sent to each of the other nodes in a circular way. The other nodes respond to the requests as they come along, whereas node zero queues the replies until all requests are sent. After node zero has sent all requests, it sends an end-of-computation message that every other node acknowledges. Then all replies are processed. Provision is made in case some of the end-of-processing acknowledgments are mixed with the replies.

The computation performed is very simple. Node zero sends out the first MESSAGES integers, and the other nodes return the square of the integer received.

Two types of messages are used: ID_DATA and ID_END. They are used to distinguish between data messages, that contain either a request or a reply, from control messages, that contain either an
end-of-computation or its acknowledgment.

The program parses an optional argument from the invocation line. If the argument is supplied, it should be the number of processors to use, out of the total number of processors. If no argument is specified, all processors are used.

For this program, you can safely ignore the PL-related calls, `pl_enable()` and `pl_disable()`.

```c
/* fm.c test main for the communication functions */
#include <stdio.h>
#include <stdlib.h>
#include "fm.h"
#include "pl-n.h"
#define MESSAGES 198
/* Message ID */
#define ID_DATA  1
#define ID_END   2

void main(int argc, char *argv[]) {
  int n;
  vnn_t d;
  struct ddesc *dd;

  int type[MESSAGES];
  int data[MESSAGES];

  int nodes;

  pl_enable();
  fm_enable();

  if(argc==2)
    nodes=atoi(argv[1]);
  else
    nodes=fm_procs();

  if(!fm_my_proc()) {
    /* ****************** */
    /* Code for node zero */
    /* ****************** */
    printf("This is node zero\n");

    /* Send some requests... */
    d=1;
    for(n=0;n<MESSAGES;n++) {
      type[n]=n;
      data[n]=n+1;

      dd=fm_descriptor(type+n,sizeof(int),data+n,sizeof(int),ID_DATA);
      if(!dd)
        printf("dd null!!!\n");
      fm_send(d,dd);
      printf("%d...",*((int *)(dd->data)));
      fflush(stdout);
```
free(dd);

/* Cycle through all nodes */
d=d+1;
if(d>=nodes)
    d=1;
/* ... And an end-of-requests message to each node */
for(d=1;d<nodes;d++) {
    dd=fm_descriptor(type,sizeof(int),data,sizeof(int),ID_END);
    if(!dd)
        printf("dd null!!\n");
    fm_send(d,dd);
    printf("[EOF%d...]",d);
    fflush(stdout);
    free(dd);
}

/* Collect the replies */
printf("\n\nCollecting replies:\n");
for(n=1;n<nodes;)
    /* Accept any message */
    dd=fm_receive(0,1);
    if(dd->id==ID_END) {
        printf("Received: id=%1d\n", dd->id);
        fflush(stdout);
        n++;
    }
    else {
        printf("Received: id=%1d, head=%2d, data=%4d\n", dd->id,
                *((int *)(dd->header)), *((int *)(dd->data))
        );
        fflush(stdout);
        fm_free(dd);
    }
else {
    /* ******************** */
    /* Code for other nodes */
    /* ******************** */
    printf("This is node %d",fm_my_proc());
    do {
        int *pi;
        dd=fm_receive(0,1);
        if(!dd)
            printf("dd null!!\n");
        if(dd->id==ID_DATA) {
            /* Return the square of the number sent */
            int *pi=(int *)dd->data;
            printf("Received %d...\n",*pi);
            fflush(stdout);
            *pi=(*pi) * (*pi);
            fm_send(0,dd);
            printf("Reply sent for %d...\n",*pi);
            fflush(stdout);
        }
    }
Example 2

Here is an example of an all-to-all communication. All processes behave in the same way, except for node zero, that will pass along a request to terminate generated at some other node (or even node itself), by means of a `fm_terminate(severity)` call. The `severity` used is the same reported by the last call to `fm_poll()`.

The program uses three probabilities, `PROB_MSG`, `PROB_WAR` and `PROB_ERR`, scaled up to `PROB_TOTAL`, to figure out when to generate certain events. Those are, respectively, sending a message to a random node, generating a warning, or generating an error.

Each pass through the loop receives and prints incoming messages, generates one or more of the events above, and waits for `SLEEP` seconds. See the code for details about the randomly generated messages and events.

The program parses an optional argument from the invocation line. If the argument is supplied, it should be the number of processors to use, out of the total number of processors. If no argument is specified, all processors are used.

For this program, you can also safely ignore the PL-related calls, `pl_enable()` and `pl_disable()`.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "fm.h"
#include "pl-n.h"

#define PROB_TOTAL 999 /* Scale of probabilities */
#define PROB_MSG 500 /* Probability to send a message */
#define PROB_WAR 17 /* Probability to generate a warning */
#define PROB_ERR 3 /* Probability to generate an error */
#define SLEEP 0.01 /* Number of seconds to sleep between tries */
```
void main(int argc, char *argv[])
{
    int r;
    vnn_t d;
    struct ddesc *dd;
    int head;
    char data[160];
    int nodes;
    double tt1,tt2;

    /* ****************** */
    /* Code for all nodes */
    /* ****************** */
    pl_enable();
    fm_enable();

    if(argc==2)
        nodes=atoi(argv[1]);
    else
        nodes=fm_procs();

    if(fm_my_proc()<nodes) {
        /* Use only 'nodes' nodes */

        printf("This is node %d
",fm_my_proc());

        tt1=pl_timer_busy(0);
        srand( (unsigned)(tt1*fm_my_proc()) );
        while(1) {
            r=fm_poll();
            /* Request to terminate at node zero */
            if(r>0) {
                printf("Sending a request to terminate, severity %d\n",r);
                fm_terminate(r);
                break;
            }
            /* Request to terminate at other nodes */
            if(r<0) {
                printf("Request to terminate received, severity %d\n",r);
                break;
            }

            /* Message received */
            while( (dd=fm_receive(0,0)) ) {
                printf("From node %d: %s\n",*((int *)dd->header), (char *)dd->data);
                fm_free(dd);
            }

            /* Send a message? */
            if( rand()%PROB_TOTAL<PROB_MSG ) {
                d=rand()%nodes;
                head=fm_my_proc();
                switch(rand()%5) {
                    case 0:
                        sprintf(data,"Isn’t %d a cool number?",rand()%100);
                        break;
                    case 1:
                        sprintf(data,"%d is definitely a cool number.",rand()%100);
                        break;
                    case 2:
                        sprintf(data,"How come %d is a cool number?",rand()%100);
                        break;
                }
            }
        }
    }
}
break;
case 3:
    sprintf(data,"How can you say that %d is a cool number?",rand()%100);
    break;
case 4:
    sprintf(data,"%d is definitely not a cool number.",rand()%100);
    break;
}

dd=fm_descriptor(&head,sizeof(int),data,strlen(data)+1,rand()%9+1);
fm_send(d,dd);
free(dd);

/* Send a warning? */
else if( rand()%PROB_TOTAL<PROB_WAR ) {
    printf("*** Producing a warning **\n");
    sprintf(data,"warning at node %d",fm_my_proc());
    fm_error(data,0);
}
/* Send an error? */
else if( rand()%PROB_TOTAL<PROB_ERR ) {
    printf("*** Producing an error **\n");
    sprintf(data,"error at node %d",fm_my_proc());
    fm_error(data,rand()%9+1);
}

/* Sleep for some time while polling */
pl_timer_clear(0);
pl_timer_start(0);
do {
    fm_poll();
    pl_timer_stop(0);
    tt2=pl_timer_elapsed(0);
    pl_timer_start(0);
} while( tt2<SLEEP );

printf("Communication done.\n");
}

fm_disable();
pl_disable();

References

- References to HTML. URL: http://extreme.chem.rpi.edu/orgchem/html.html.
- Portability Library (PL) Documentation.
- Building Communication Paradigm with the CM-5 Active Message layer (CMAM). T. Eicken and D. Culler.
Portability Library

This document contains a complete description of the Portability Library (PL). Along with a description of the library and an annotated list of primitives, there is a section on how to use the library. A non-trivial example program is also included as well as a reference section. This document is written in a generic way and thus covers no specific version of the library.

INDEX OF THIS PAGE

- Portability Library
  - What is the Portability Library?
  - Description of the PL library
    - The PL Programming Paradigm
    - PL Host Primitives
    - PL Node Primitives
  - Using the PL library
  - PL User Requirements
  - Example
  - References

What is the Portability Library?

The Portability Library (PL), which really started standing for Parallel/Portable Library, is a library of primitives for those services that, although provided in most or all systems, are achieved in different ways or through different system-specific function calls. By grouping all those functions together and assigning them a single name across systems, portability can be preserved as soon as a version of the PL library is available. As of this writing, the PL library has been implemented and tested under the following machines and operating systems:

- An emulated version, that works on Sun machines running Solaris. This version is not intended for performance -which it would not achieve- but rather, for testing purposes. A version of the program under development can be compiled and run with this version until the program is ready for a "production" version based on a real parallel machine.

- A CM-5 version, that works on CM-5 machines running CMost.

Similar versions of the Fast Messages Library (FM) exist. See the documentation about this library for more information.

Description of the PL library
The PL Programming Paradigm

There are two versions of the PL library, one intended for the host node, one intended for the processing nodes. The host node is the node that will set up the computation, by reading the problem definition from the user, and send the problem information to the processing nodes. Once the desired computation has been performed, it will also collect the solution and generate the user output. The problem information may be some processed form of the user input, if that processing is to be done sequentially. In that case, that preprocessing could be performed by the host node. In the same way, the user output may be some processed or compiled form of the processing nodes output, if that processing or compilation is to be done sequentially. That post-processing could also be performed by the host node.

The processing nodes are the processes or -more likely- processors that will compute the desired output from the problem definition coming from the host node. They can -and probably will- communicate with each other, by means of the FM library mentioned above. Once their output is ready, it will be sent to the host node to generate the user output.

The PL library contains primitives to perform the communication between the host node and the processing nodes as well as some other miscellaneous functions only present on the node version. The names of the primitives look the same both for the host node and the processing nodes. Nevertheless, they are named differently internally to allow linking with a single library file. For this reason, a different header file must be included for each version. The host program will use:

```
#include <pl-h.h>
```

whereas the processing nodes will use:

```
#include <pl-n.h>
```

That should be easy to remember, as "h" stands for host and "n" stands for node.

Because of the way that the PL library is written, the host node code is split into two different programs (and executables.) One of them will be used to set up the computation by generating the input to the processing nodes. The other will be used to generate the user output by reading and displaying a form of the output coming from the processing nodes. In order to make references to both programs easier, from here on, the first part of the node program will be called the input host program, whereas the second part of the host program will be called the output host program. Since the processing nodes code is another program, that yields a total of three different pieces of code for a single problem.

The reason to divide up the host node code in two pieces is to allow using a generic "host program" to interact with the processing nodes, in those systems where this is the preferred execution model. For instance, on the CM-5 using the CMMD communications library, programs that follow this model of execution are called host-less programs. These programs have a number of advantages mostly related to I/O efficiency and simplicity that will not be discussed here.

PL Host Primitives

Here is a list of the PL host primitives. The ones that generate output are intended for the input host program; the ones that collect input are intended for the output host program. In both cases, the following line should be included before using any of the functions:
A list of the primitives, preceded by their prototype follows. They all behave in much the same way as
the C standard input/output functions, except for the fact that the FILE* they operate on is implicit.
Also, all functions return zero on success or non-zero in case of an error.

```c
#include <pl-h.h>
int pl_open_read(void)
Open a data stream between the processing nodes and the host node. All data written by the processing
nodes using the symmetric function calls will be read from this stream. No more than one
pl_open_read() call should be used in the output host node program.
```

```c
int pl_open_write(void)
Open a data stream between the host node and the processing nodes. All data written on this stream will
be read by the symmetric function calls on the processing nodes. No more than one pl_open_write() call should be used in the input host node program.
```

```c
int pl_read(char *buffer, int nbytes)
Read a block of data coming from the processing nodes. Since no specific order in the way the
processing nodes write is guaranteed, the caller should be able to select the correct size for the memory
block, if more than one is used. That can be done easily by prefixing every block written by another
fixed-size block that describes it.
```

```c
int pl_write(char *buffer, int nbytes)
Write a block of data to be read by all processing nodes. The data blocks written in this way will be
broadcasted to all nodes in the order written.
```

```c
int pl_close_read(void)
Close the data stream established between the processing nodes and the host node. No more than one
pl_close_read() call should be used in the output host program.
```

```c
int pl_close_write(void)
Close the data stream established between the host node and the processing nodes. No more than one
pl_close_write() call should be used in the input host program.
```
**PL Node Primitives**

Similarly as with the PL Host Primitives, the following line should be included in the processing nodes program before using any of the functions:

```c
#include <pl-n.h>
```

A list of the primitives, preceded by their prototype follows. The I/O functions behave in much the same way as the C standard input/output functions, except for the fact that the `FILE*` they operate on is implicit. This is similar to the PL Host Primitives above. In this case also, all I/O functions return zero on success or non-zero in case of an error.

```c
void pl_enable(void)
```

Should be called before any other PL Node functions, probably at the very beginning of the processing nodes program execution, and always before the FM `fm_enable()` call.

```c
void pl_disable(void)
```

Should be called after all other PL functions for a graceful ending, probably right before the end of the processing nodes program execution, and always after the FM `fm_disable()` call.

```c
int pl_open_read(void)
```

Open a data stream between the host node and the processing nodes. All data written by the host node using the symmetric functions will be read from this stream. No more than one `pl_open_read()` call should be used in the processing nodes program.

```c
int pl_open_write(void)
```

Open a data stream between the processing nodes and the host node. All data written on this stream will be read by the symmetric functions on the host node. No more than one `pl_open_write()` call should be used in the processing nodes program.

```c
int pl_read(char *buffer, int nbytes)
```

Read a block of data coming from the host node. Since blocks written by the host node are broadcasted, all processing nodes have to read all blocks, even if the information they contain is not intended for them.
int pl_write(char *buffer, int nbytes)

Write a block of data to be read by the host node. Since no specific order in the host node input stream is guaranteed, the caller processing node should indicate the host node what the size of the memory block will be, if more than one is used. That can be done easily by prefixing every block written by another fixed-size block that describes it.

int pl_close_read(void)

Close the data stream established between the host node and the processing nodes. No more than one pl_close_read() call should be used in the processing nodes program.

int pl_close_write(void)

Close the data stream established between the processing nodes and the host node. No more than one pl_close_write() call should be used in the processing nodes program.

void pl_disablei(void)

Disable system interrupts. Should be used before accessing any information that could be modified by any kind of handler or interrupt driven procedure and also be accessed by regular processing nodes program code.

void pl_enablei(void)

Restore the system interrupts state before the last pl_disablei() call. Both interrupt handling functions are intended to support the implementation of the FM library. There is no reason why a normal user program should use these function calls.

int pl_printf(char *format, ...)

Similar to the printf(...) C standard input/output library function. It also takes a format string and a variable number of arguments. Should be used to generate log output from the processing nodes program. It may be a null function in systems that do not allow any logging from the processing nodes. Where available, it will identify the calling node as part of the message logged.

int pl_error(char *format, ...)

Similar to the pl_printf(...) function above, except that it will also abort the processing nodes
These functions are used to clear, start and stop timer `timer` respectively. All timers should be cleared before starting them. It is possible to restart a timer once it has been stopped. That can be used to accumulate different measured times. Clearing the timer provides a way to measure times independently. Every timer should be stopped before reading its value with the functions below. Up to eight timers are guaranteed in any implementation, numbered from zero to seven. Note that timer seven (by default) is used by the FM library. Check the FM library documentation for details.

Return the busy or elapsed value (respectively) of the timer `timer`. All times returned are in seconds. The fraction of a second is given as the decimal part of the double returned. The busy value measures the time that the system assigned to the processing nodes program in the given calling node, between the `pl_timer_start(...)` and `pl_timer_stop(...)` calls. The elapsed value measures the total time elapsed between the same function calls. The two times will be different in multitasking environments, when more than one process per node may be running at once. Not all systems may be able to provide both times; in that case, the values returned will be the same for both functions. The precision of the timers is within $\frac{1}{\text{PL_TIMER_PREC}}$ seconds, i.e. the internal clock runs at $\text{PL_TIMER_PREC}$ ticks per second.

Using the PL library

As it was mentioned before, a complete program -the executable code that solves a problem on a parallel machine- is made up from three different subprograms, so far referred as: the input host node program, the output host node program and the processing nodes program. Each program contains the set of instructions to be executed in a different part of the total computation. Precisely:

- The input host node program is executed first in a single processor. It will read the user input and generate the output to be broadcasted to all the processing nodes. It does not necessarily execute in parallel with the processing nodes.

- The processing nodes program executes next on all processing nodes at once. It will read all the input host node program output on each of the processing nodes, and perform the desired computation. Communication among the processing nodes is possible using the primitives of the FM library. As part of the computation, the processing nodes should generate some output that will be read by the output host node program. The execution of the parallel program ends when all individual processing nodes finish execution, i.e. a barrier is established so that they will all end at
Once.

Note that all processing nodes are required to read all the input coming from the host node. Undesired input can be thrown away as it is read, but it has to be read.

- The output host node program executes last, also in a single processor. It will read the processing nodes output and generate the user output. It will not necessarily execute in parallel with the processing nodes.

The input host node program might not have to be executed if no input is required by the processing nodes. Also, the output host node program might not have to be executed if no user output is required. In any of both cases, this is correct as long as no PL communication functions are used between processing nodes program and the other inexistent program; i.e. between the input host node program and the processing nodes program, or between the processing nodes program and the output host node program.

Note that the distinction of the different programs is not totally inherent to the PL library. Depending on the system, each program may have to be compiled or linked differently. On real parallel environments, at least the processing nodes program will have a different linking process.

It has been so far customary to name each of the programs in the same way, but with a different postfix: "-host1" for the input host node program, "-host2" for the output host node program, and "-node" for the processing nodes program. The example included with this document also follows this convention.

To use the appropriate version of the PL library, you need to include the file pl-h.h or pl-n.h before any PL function call. The best way to do that without having to change source files for all different versions across platforms, is to include the line (supposing the host version):

```
#include <pl-h.h>
```

and then make sure that your makefile includes a definition like:

```
CFLAGS=-I<pl-directory>
```

Where `<pl-directory>` is the directory where the files pl-h.h and pl-n.h are.

If you are invoking the C compiler directly from the command line, you should add the option `-I<pl-directory>` to the invocation line.

Also, you will need to link your object code with the library libpar.a. For that purpose, you can add a line like:

```
LIBS=-L<lib-directory> -lpar
```

in your makefile, and have all your linking command invocations include it by adding `$(LIBS)` as appropriate. Once more, if you are invoking the C compiler directly from the command line, you should add the option `-L<lib-directory>` to the invocation line.

Be aware that many compilers will not pass on the linking options to the linker correctly, unless they appear after the source or object file names that you are also including. It is always safe to include the
linking options last (or right before the -o <target> if you are naming your target file.)

It has been so far customary to have a single directory, libpar, include all the parallel machine-specific files and the single library libpar.a. That directory includes all PL and FM (see the FM library documentation) specific files, and maybe the sources and the makefile to generate the libpar.a library file. In that case, and assuming that your libpar directory is ~/libpar the options above become:

CFLAGS=-I~/libpar
LIBS=-L~/libpar -lpar

and similarly for the command line invocation options.

---

**PL User Requirements**

As it has been noted in the primitive description above, the PL library requires that the user calls `pl_enable()` before using any of the PL functions in the node version. All processing nodes have to make the call, even if they will not be involved in the computation. In addition, the output generated by the input host node program has to be read by all processing nodes, even if none of them needs all the data that is being broadcasted to them. All irrelevant input can be thrown away as it is read, but it has to be read.

Also, the PL library requires that the user calls `fm_disable()` before any processing node terminates. All processing nodes have to make the call, even if they were not involved in any computation.

---

**Example**

Here is a simple example of how all three different programs required to solve a problem in parallel fit together. Note that the recommended naming convention is followed. The `pl-host1.c` input host node program, writes the integers 0 through 255 as the information to send to the processing nodes. The integers are written directly in the machine representation, which will only work if the internal integer representation is the same on the host as it is on the processing nodes. That is the likely case (and true in both implemented versions of the PL library,) but may not work for all architectures. It is used here to make the example simple.

Then the `pl-node.c` processing nodes program reads that information, and generates the square of each number. Each processor reads all numbers and generates the squares of eight of those numbers in parallel. Processor zero computes the squares of the first eight numbers, 0 through 7; processor one computes the squares of the following eight, 8 through 15; and so on. A total of 32 processors is required to compute the squares of 256 numbers. Note in the code below how all processing nodes read all their input, even though most of it is not used.

Finally, the computed information is written by the `pl-host2.c` output host node program. The order in which the computed squares is read by the output program is not guaranteed to be the same as the order in which they were generated, and definitely not guaranteed to be the same as the order in which the
input numbers used to compute the squares, were written by the input host node program. If the user wanted the solution in order, every squared number (as written by the processing nodes program) could be preceded (or followed) by the number itself. Then the output from the processing nodes could be stored in an array, using the number as the index for the square value. Printing the values of the array in order, would generate the output sorted.

In the case of the two existing implementations of the PL library, the output will be generated in perfect order. That is because the two implementations arrange the output coming from the processing nodes in the order it comes from each node, and one node at a time, starting from node zero. Nevertheless, other implementations of the PL library may arrange the output from the processing nodes in the order it is received (or any other order), thus quite likely generating a different output for each run. The user of the PL library should not assume any specific order whatsoever.

File: pl-host1.c

/* Test program for the PL library */
/* Host initialization */
#include "pl-h.h"

main()
{
    int n;
    pl_open_write();
    printf("Writing the information for the nodes:

");
    /* Write a couple of numbers to be read by the processing nodes */
    for(n=0;n<256;n++) {
        pl_write(&n, sizeof(int));
        printf("%d ",n);
    }
    printf("\n");
    pl_close_write();
}

File: pl-host2.c

/* Test program for the PL library */
/* Host results collection */
#include "pl-h.h"

main()
{
    int n,i;
    pl_open_read();
    printf("Reading the information written by the nodes:

");
    /* Read out all numbers written by the processing nodes */
    do {
        n=pl_read(&i, sizeof(int));
        if(!n)
            printf("%d ",i);
        printf("%d ",i);
    } while(n);
}
while(!n);
printf("\n");
pl_close_read();
}

File: pl-node.c

/* Test program for the PL library */
/* Node program. Requires 32 processing nodes to generate all output */

#include "pl-n.h"
#include "fm.h"

main()
{
    int node;
    int n,i,ii;
    pl_enable();
    fm_enable();
    node=fm_my_proc();
    pl_open_read();
    pl_open_write();

    /* All nodes read all the information */
    for(n=0;n<256;n++) {
        pl_read(&i, sizeof(int));
        /* And every node processes a piece of it */
        if(n/8==node) {
            ii=i*i;
            pl_write(&ii, sizeof(int));
        }
    }

    pl_close_write();
    pl_close_read();
    fm_disable();
    pl_disable();
}

References

- References to HTML. URL: http://extreme.chem.rpi.edu/orgchem/html.html.
- Fast Messages (FM) Library Documentation.
- CM-5 Technical Summary. Thinking Machines Corporation.
- CMMD User’s Guide. Thinking Machines Corporation.
- Using DJM on the CM-5. Thinking Machines Corporation.

Javier Elices  
CoRE 346, +1 908-445-4070  
elices@cs.rutgers.edu

This page, last updated: October 20, 1995.