FM: Fast Messaging on the Cray T3D

Vijay Karamcheti Andrew Chien

Concurrent Systems Architecture Group
Department of Computer Science
University of Illinois at Urbana-Champaign

The FM library contains fast messaging primitives which exploit special features of the Cray T3D hardware to provide extremely low latency for short messages. FM provides an order of magnitude lower latency than Cray's PVM and achieves performance comparable to SHMEM get while providing a message-passing interface. There are two distinct sets of primitives which make use of the fetch-and-increment and atomic swap hardware respectively. The fetch-and-increment implementation optimizes for lowest possible latency and is suitable for situations with light communication traffic. The atomic swap version eliminates output contention at the cost of slightly higher latency, but by doing so can deliver robust performance even for dynamic and irregular traffic loads. Detailed analysis of the FM messaging implementations is available as a technical report from `http://www-csag.cs.uiuc.edu/projects/communication/cray-messaging.html`.

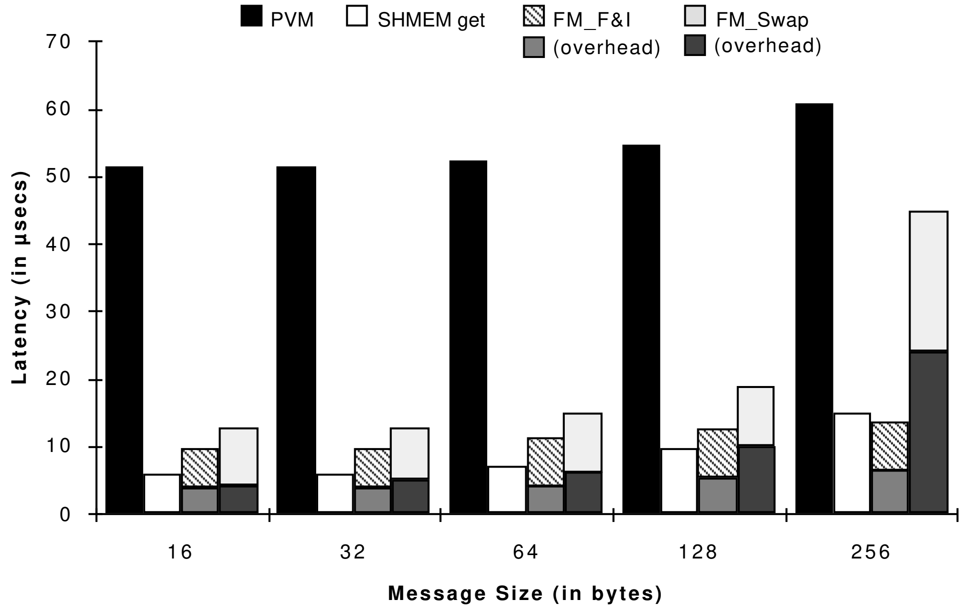

Figure 1: Comparison of communication latency.

Figure 1 compares the performance of the FM primitives for various message sizes with corresponding functionality built using the fast PVM send/receive primitives and SHMEM get. All times (except the SHMEM case) correspond to the latency from send through receive for a message. SHMEM times correspond to transferring data from a remote node via a get operation.

Note that although send/receive actually implements greater functionality than SHMEM get (i.e., buffer management), the FM primitives obtain comparable performance. The figure also shows the fraction of the communication latency which arises from software overhead (for PVM synchronous send/receive and SHMEM get, the overhead is the same as latency). The FM primitives overlap communication and computation; consequently, the non-overhead portion of latency can be effectively eliminated.

The remainder of this document describes the messaging primitives and their use.

Fetch-and-Increment Messaging

The fetch-and-increment register is used to index a preallocated set of message buffers. The source node obtains a valid index by performing a fetch-and-increment operation with respect to the destination node, then transfers data using remote stores. Message completion at the receiver is detected by a tag at the end of the message buffer. These messaging primitives provide low latency, but in situations where there is output contention, the remote stores can saturate a nodes memory bandwidth, backing traffic up into the network and degrading communication performance.

Each message send is associated with execution of a handler at the destination node (similar to active messages) The interface consists of three primitives **FMf_send_4**, **FMf_send_4i**, and **FMf_send** which send messages and **FMf_extract_1** and **FMf_extract** which invoke the message handlers for one or all pending messages respectively. Before using the library, **FM_initialize** must be called to initialize the messaging layer and the buffer pool. The maximum message size and the size of the buffer pool can be changed prior to initialization using the **FM_set_parameter** function. More detailed description and usage examples are given in the accompanying manual pages. Note that messages are not processed at the destination unless explicit calls to **FMf_extract** are made.

Atomic-Swap Messaging

The atomic swap register and prefetch hardware together are used to construct a distributed queue implementation of messaging. On a send, messages are linked into a distributed message queue for the destination node, but the message data remains at the source. When the destination node receives the message, it retrieves the message from the source memory. The prefetch hardware is used to hide the latency of both the atomic swap and remote loads. Transmitting data lazily (pulling messages) eliminates output contention (fan-in) problem, but incurs a small increase in latency. The substantial benefit is that the send rate to a node is not limited by memory bandwidth. Consequently, these primitives provide better performance for irregular and dynamic traffic patterns.

Each message send is associated with execution of a handler at the destination node (similar to active messages) The interface consists of three primitives **FMs_send_4**, **FMs_send_4i**, and **FMs_send** which send messages and **FMs_extract_1** and **FMs_extract** which invoke the message handlers for one or all pending messages, implicitly dequeueing them. The user must ensure that either an **FMs_extract** call or an **FMs_complete_send** call follows the last send in a program, otherwise the data corresponding to the last send will not be accessible from the destination. Before using the library, **FM_initialize** must be called to initialize the messaging layer and the buffer pool. The current implementation limits the maximum message size to 4096 bytes. More detailed description and usage examples are given in the accompanying manual pages.

1 Usage Notes

1. The fetch-and-increment primitives or swap primitives can be used independently without any problems. However, note that **FMf_extract** and **FMs_extract** only extract messages sent using the corresponding send primitives.

2. The current implementation encodes the destination node number as an 8-bit quantity. Thus, T3D configurations larger than 256 nodes are not currently supported. Please let us know if this represents a problem.

3. When using the FM library, the fetch-and-increment registers and the prefetch hardware cannot be used for any other purpose. In addition, the fetch-and-increment messaging primitives make use of entries 22 and 23 of the DTB Annex, while the swap messaging primitives use annex entries 17–21.

NAME

FM_initialize, FM_set_parameter – Initialization of the messaging library

SYNOPSIS

#include <fm.h>
void FM_initialize(void);
void FM_set_parameter(int param_id, int value);

DESCRIPTION

FM_initialize must be called on all nodes before any other functions from the messaging library are used. It initializes global variables used by the library functions, and also allocates message buffers used by the Fetch-and-increment messaging primitives.

FM_set_parameter enables users to configure the messaging library by changing default values of certain parameters. All parameters accept integer values and are as follows (the parameter identifiers correspond to macros defined in **fm.h**):

MAX_MSG_SIZE_FINC	The largest size of the messages (in bytes) which will be transferred using the fetch-and-increment messaging primitives. The parameter value must be a power of 2. The default value of this parameter is 1024 bytes.
MSG_BUFFER_SIZE_FINC	The size of the preallocated message buffer pool used by the fetch-and-increment messaging primitives. Since the message pool is not extended during the program run, this parameter must be set so that it can accomodate differences in the sending and receiving rates of the nodes. The parameter value must be a power of 2. The default size of the buffer pool is 512 buffers.

USAGE

FM_initialize must be called before any memory requests are made by the node code. This ensures that the buffers on all nodes are allocated at the same addresses.

Any **FM_set_parameter** calls must occur before **FM_initialize** and should be consistent across all the nodes. For example, to set the maximum message size transferred using the fetch-and-increment primitives to 1024 bytes:

```
main() {
  ...
  FM_set_parameter(MAX_MSG_SIZE_FINC, 1024);
  FM_initialize();
  ...
}
```

NAME

FMf_send_4 – Send 4-word message using fetch-and-increment mechanism

SYNOPSIS

#include <fm.h>
typedef void FM_shandler();
void FMf_send_4(int node, FM_shandler *fptr, ...);

DESCRIPTION

FMf_send_4 sends a four word message to the remote **node**, calling the handler **fptr** and passing up to 4 64-bit word arguments. The call

FMf_send_4(node, fptr, i1, i2, i3, i4);

sends a four-word message to **node** and causes

(*fptr)(i1, i2, i3, i4);

to be invoked. **FMf_send_4** is declared as a varargs function, but really accepts up to four words of arguments in addition to **node** and **fptr**, and passes them into the remote function. The implementation of **FMf_send_4** avoids any local memory references by operating completely from registers. The four arguments are supplied in registers (r18 through r21) and according to the Alpha calling convention can be any combination of integer and double values. Note that **FMf_send_4** always transmits a four word message, and thus there is no performance advantage from sending fewer than four arguments to the remote function.

USAGE

For example, if each processor wanted to send the values of some variables to its right neighbor (assuming the processors are organized as a linear array):

```
int recvd_neighbor_vals = 0;
void store_neighbor_values(int i1, int i2, double d3) {
  nbr_var1 = i1; nbr_var2 = i2; nbr_var3 = d3;
  recvd_neighbor_vals = 1;
}

{
  int var1, var2; double var3;
  ...
  FMf_send_4( (MY_PE+1)%N_PES, store_neighbor_values, var1, var2, var3 );
  while ( recvd_neighbor_vals != 1 ) FMf_extract();
  ...
}
```

FMf_extract extracts pending messages and invokes their handlers.

Although the remote function handler can contain arbitrary computation, a possibility of deadlock exists if the handler itself involves a message send. Deadlock can be prevented by restricting remote handlers to perform only local computation. Alternatively, a large value of the **MSG_BUFFER_SIZE_FINC** will make deadlock very unlikely.

NAME

FMf_send_4i – Send a 4-word indirect message using fetch-and-increment mechanism

SYNOPSIS

#include <fm.h>
typedef void FM_shandler();
void FMf_send_4i(int node, FM_shandler **fptr, ...);

DESCRIPTION

FMf_send_4i is nearly identical to **FMf_send_4** except that it performs an indirect function call on the remote **node**: a pointer to a pointer to a remote function, **fptr**, is passed in the message. The call

FMf_send_4i(node, fptr, i1, i2, i3, i4);

sends a four-word message to **node** and causes

(fptr)(fptr, i1, i2, i3, i4);**

to be invoked. As with **FMf_send_4**, up to four words of arguments in addition to **node** and **fptr** can be supplied.

USAGE

In some circumstances, **FM_send_4i** enables one pointer to refer to both the function and its calling environment. For example, in the following code fragment, a short message send can be used to initialize an arbitrary structure on the remote node:

```
struct Object {
  FM_shandler *fptr; /* initialized to init_object_vars */
  int          fld1;
  int          fld2;
  int          fld3;
  int          fld4;
};

void init_object_vars((Object *)obj, int i1, int i2, int i3, int i4) {
  obj->fld1 = i1; obj->fld2 = i2; obj->fld3 = i3; obj->fld4 = i4;
}

{
  ...
  FMf_send_4i( node, obj, var1, var2, var3, var4 );
  ...
}
```

To achieve the above initialization without **FMf_send_4i**, five arguments (the pointer of the structure and the four initialization arguments) would need to be transmitted.

NAME

FMf_send – Send a longer message using fetch-and-increment mechanism

SYNOPSIS

#include <fm.h>
typedef void FM_lhandler(void *, int);
void FMf_send(int node, FM_lhandler *fptr, void* buf, int byte_count);

DESCRIPTION

FMf_send sends a message to the remote **node**, calling the handler **fptr** and passing the buffer, **buf**, as argument. The call

FMf_send(node, fptr, buf, len);

will allocate a buffer of size **len** on the remote **node**, transfer the source buffer **buf** into the allocated buffer, invoke

(*fptr)(*allocated_buffer***, len);**

and finally deallocate the buffer. Note that since the storage corresponding to the allocated buffer is reclaimed on exit from the message handler, any data required after the handler exit must be explicitly copied to its final destination.

The maximum size of messages which can be transmitted using this primitive is limited by the value of the **MAX_MSG_SIZE_FINC** parameter. This parameter can be changed during initialization time using the function **FM_set_parameter**.

USAGE

To transmit more than 4 words from one node to another,

```
void destination_handler(void *buf, int len) {
  ...
  /* compute using buffer contents */
  ...
  /* copy whatever portion is necessary */
}

{
  int buf[256];
  ...
  FMf_send(node, destination_handler, (void *)buf, 256*sizeof(int));
  ...
}
```

NAME

FMf_extract_1, FMf_extract – Extract pending messages and invoke handlers

SYNOPSIS

#include <fm.h>
int FMf_extract_1(void);
int FMf_extract(void);

DESCRIPTION

FMf_extract∗ retrieves messages sent using the **FMf_send∗** primitives from the message buffers and invokes the appropriate message handlers. Note that no handlers get invoked unless a call to **FMf_extract∗** is made. A call to **FMf_extract_1** dequeues and processes one pending message while a call to **FMf_extract** processes all pending messages. The return value is 1 if some messages were pending and 0 otherwise.

USAGE

The **FMf_extract∗** functions can be used to build a variety of wait mechanisms. For example, in the code fragment below, a node initiates an operation on a remote node and processes all incoming messages while waiting for its completion.

```
int sync_variable;
void sync_handler(void) { sync_variable = 1; }

void ping(int i1, int i2, int req_node, RT_shandler *fptr) {
  ...
  /* computation using i1, i2 */
  ...
  FMf_send_4(req_node, fptr);
}

{
  ...
  sync_variable = 0;
  FMf_send_4( node, ping, arg1, arg2, MY_PE, sync_handler );
  while (sync_variable != 1) FMf_extract();
  ...
}
```

In the fetch-and-increment based messaging scheme, the fetch-and-increment value acts as an index into a preallocated buffer pool. Therefore, the sender node blocks up when the destination node's buffer is all filled up. It is the responsibility of the programmer to ensure either that **FMf_extract∗** is called frequently or that **MSG_BUFFER_SIZE_FINC** is set to a large enough value that sender nodes do not need to block.

NAME

FMs_send_4, FMs_send_4i, FMs_send – Send message using atomic-swap mechanism

SYNOPSIS

#include <fm.h>
typedef void FM_shandler(...);
typedef void FM_lhandler(void *, int);
void FMs_send_4(int node, FM_shandler *fptr, ...);
void FMs_send_4i(int node, FM_shandler **fptr, ...);
void FMs_send(int node, FM_lhandler *fptr, void* buf, int byte_count);

DESCRIPTION

The swap-based send primitives have exactly the same interface as their fetch-and-increment counterparts and almost the same semantics. The only difference corresponds to when a send operation is considered complete. In the fetch-and-increment case, returning from a send call implies that the message data has actually been sent to the remote node. In this case, it just means that the send operation has been initiated.

A pending send operation gets completed when either another send, or a **FMs_extract**, or a **FMs_complete_send** call is encountered.

The maximum message size which can be transmitted using **FMs_send** is restricted to 4096 bytes. This arises from the implementation encoding the message length as a 12-bit quantity.

USAGE

It is essential that either a **FMs_extract**∗ or a **FMs_complete_send** follow the last send on a node. This ensures that all messages sent from a node actually finish getting enqueued into the message queue of the appropriate destination. Without this, the destination will not be able to extract the message corresponding to the send.

NAME

FMs_complete_send – Complete any pending sends issued using atomic-swap based primitives

SYNOPSIS

#include <fm.h>
void FMs_complete_send(void);

DESCRIPTION

FMs_complete_send ensures that any previously issued sends have been attached to the distributed message queues of the corresponding destination nodes. Without this, the destination will not be able to extract the message corresponding to the send.

USAGE

Depending on the synchronization structure of the program, **FMs_complete_send** may or may not be required. If the synchronization structure is such that it requires all messages sent from the node to be received by their destination nodes, then the programmer must ensure that the last send in the program is followed by either a call to **FMs_extract** or a call to **FMs_complete_send**. For example,

```
{
  ...
  FMs_send(node, remote_function, buf, len);      /* last send */
  ...
  FMs_complete_send();                            /* completes previous send */
  ...
}
```

NAME

FMs_extract_1, FMs_extract – Extract pending messages and invoke handlers

SYNOPSIS

#include <fm.h>
int FMs_extract_1(void);
int FMs_extract(void);

DESCRIPTION

FMs_extract_1 and **FMs_extract** provide exactly the same interface as **FMf_extract_1** and **FMf_extract**. They extract one or all pending messages sent using **FMs_send**∗ primitives, and invoke the appropriate message handlers.

USAGE

Usage is similar to **FMf_extract**∗ except for one minor difference. Unlike **FMf_extract**∗ which needs to be called frequently to prevent sender nodes from blocking, **FMs_extract_1** or **FMs_extract** need be called only when the receiver is ready to process pending messages.