Converse Extensions Library

August 18, 1997

Converse Parallel Programming Environment was developed as a group effort at Parallel Programming Laboratory, University of Illinois at Urbana-Champaign. The team consisted of Attila Gursoy, Sanjeev Krishnan, Joshua Yelon, Milind Bhandarkar, Narain Jagathesan, Robert Brunner and Laxmikant Kale.

Contents

Chapter 1

Introduction

The Converse Extensions Library is a collection of modules that have been implemented on top of the Converse API. Each of these modules was deemed potentially useful to other Converse users, thus, we distribute these modules along with Converse as a convenience. *You don't need to read any part of this manual to use Converse.*

Chapter 2

Tag Matching

The message manager is a data structure that can be used to put together runtime systems for languages that support tag-based message retrieval.

The purpose of the message manager is to store, index, and retrieve messages according to a set of integer tags. It provides functions to create tables, functions to insert messages into tables (specifying their tags), and functions to selectively retrieve messages from tables according to their tags. Wildcard tags can be specified in both storage and retrieval.

To use the message manager, you must include `converse.h` and link with the converse library.

In actuality, the term "message manager" is unnecessarily specific. The message manager can store and retrieve arbitrary pointers according to a set of tags. The pointers do *not* necessarily need to be pointers to converse messages. They can be pointers to anything.

typedef struct CmmTableStruct *CmmTable
This opaque type is defined in `converse.h`. It represents a table which can be used to store messages. No information is publicized about the format of a CmmTableStruct.

#define CmmWildCard (-1)
This #define is in `converse.h`. The tag -1 is the "wild card" for the tag-based lookup functions in the message manager.

CmmTable CmmNew();
This function creates a new message-table and returns it.

void CmmPut(CmmTable t, int ntags, int *tags, void *msg)
This function inserts a message into a message table, along with an array of tags. `ntags` specifies the length of the `tags` array. The `tags` array contains the tags themselves. `msg` and `t` specify the message and table, respectively.

void *CmmGet(CmmTable t, int ntags, int *tags, int *ret_tags)
This function looks up a message from a message table. A message will be retrieved that "matches" the specified `tags` array. If a message is found that "matches", the tags with which it was stored are copied into the `ret_tags` array, a pointer to the message will be returned, and the message will be deleted from the table. If no match is found, 0 will be returned.

To "match", the array `tags` must be of the same length as the stored array. Similarly, all the individual tags in the stored array must "match" the tags in the `tags` array. Two tags match if they are equal to each other, or if `either` tag is equal to `CmmWildCard` (this means one can `store` messages with wildcard tags, making it easier to find those messages on retrieval).

void *CmmProbe(CmmTable t, int ntags, int *tags, int *ret_tags)
This function is identical to `CmmGet` above, except that the message is not deleted from the table.

void CmmFree(CmmTable t);
This function frees a message-table t. WARNING: It also frees all the messages that have been inserted into the message table. It assumes that the correct way to do this is to call `CmiFree` on the message. If this assumption is incorrect, a crash will occur. The way to avoid this problem is to remove and properly dispose all the messages in a table before disposing the table itself.

Chapter 3

Automatic Parameter Marshalling

Automatic Parameter Marshalling is a concise means of invoking functions on remote processors. The CPM module handles all the details of packing, transmitting, translating, and unpacking the arguments. It also takes care of converting function pointers into handler numbers. With all these details out of the way, it is possible to perform remote function invocation in a single line of code.

3.1 CPM Basics

The heart of the CPM module is the CPM scanner. The scanner reads a C source file. When it sees the keyword `CpmInvokable` in front of one of the user's function declarations, it generates a *launcher* for that particular function. The *launcher* is a function whose name is `Cpm_` concatenated to the name of the user's function. The launcher accepts the same arguments as the user's function, plus a *destination* argument. Calling the *launcher* transmits a message to another processor determined by the *destination* argument. When the message arrives and is handled, the user's function is called.

For example, if the CPM scanner sees the following function declaration

```
CpmInvokable myfunc(int x, int y) { ... }
```

The scanner will generate a launcher named `Cpm_myfunc`. The launcher has this prototype:

```
void Cpm_myfunc(CpmDestination destination, int x, int y);
```

If one were to call `Cpm_myfunc` as follows:

```
Cpm_myfunc(CpmSend(3), 8, 9);
```

a message would be sent to processor 3 ordering it to call `myfunc(8,9)`. Notice that the *destination* argument isn't just an integer processor number. The possible destinations for a message are described later.

When the CPM scanner is applied to a C source file with a particular name, it generates a certain amount of parameter packing and unpacking code, and this code is placed in an include file named similarly to the original C file: the `.c` is replaced with `.cpm.h`. The include file must be included in the original `.c` file, after the declarations of the types which are being packed and unpacked, but before all uses of the CPM invocation mechanisms.

Note that the `.cpm.h` include file is *not* for prototyping. It contains the C code for the packing and unpacking mechanisms. Therefore, it should only be included in the one source file from which it was generated. If the user wishes to prototype his code, he must do so normally, by writing a header file of his own.

Each `.cpm.h` file contains a function `CpmInitializeThisModule`, which initializes the code in *that* `.cpm.h` file. The function is declared `static`, so it is possible to have one in each `.cpm.h` file without conflicts. It is the responsibility of the CPM user to call each of these `CpmInitializeThisModule` functions before using any of the CPM mechanisms.

We demonstrate the use of the CPM mechanisms using the following short program `myprog.c`:

```
 1:    #include "myprog.cpm.h"
 2:
 3:    CpmInvokable print_integer(int n)
 4:    {
 5:      CmiPrintf("%d\n", n);
 6:    }
 7:
 8:    user_main(int argc, char **argv)
 9:    {
10:      int i;
11:      CpmModuleInit();
12:      CpmInitializeThisModule();
13:      if (CmiMyPe()==0)
14:        for (i=1; i<CmiNumPes(); i++)
15:          Cpm_print_integer(CpmSend(i), rand());
16:    }
17:
18:    main(int argc, char **argv)
19:    {
20:      ConverseInit(argc, argv, user_main, 0, 0);
21:    }
```

Lines 3-6 of this program contain a simple C function that prints an integer. The function is marked with the word `CpmInvokable`. When the CPM scanner sees this word, it adds the function `Cpm_print_integer` to the file `myprog.cpm.h`. The program includes `myprog.cpm.h` on line 1, and initializes the code in there on line 12. Each call to `Cpm_print_integer` on line 15 builds a message that invokes `print_integer`. The destination-argument `CpmSend(i)` causes the message to be sent to the *i*'th processor.

The effect of this program is that the first processor orders each of the other processors to print a random number. Note that the example is somewhat minimalist since it doesn't contain any code for terminating itself. Also note that it would have been more efficient to use an explicit broadcast. Broadcasts are described later.

All launchers accept a *CpmDestination* as their first argument. A *CpmDestination* is actually a pointer to a small C structure containing routing and handling information. The CPM module has many built-in functions that return *CpmDestinations*. Therefore, any of these can be used as the first argument to a launcher:

> **CpmSend(***pe***)** - the message is transmitted to processor *pe* with maximum priority.
>
> **CpmEnqueue(***pe, queueing, priobits, prioptr***)** - The message is transmitted to processor *pe*, where it is enqueued with the specified queueing strategy and priority. The *queueing*, *priobits*, and *prioptr* arguments are the same as for **CqsEnqueueGeneral**.
>
> **CpmEnqueueFIFO(***pe***)** - the message is transmitted to processor *pe* and enqueued with the middle priority (zero), and FIFO relative to messages with the same priority.
>
> **CpmEnqueueLIFO(***pe***)** - the message is transmitted to processor *pe* and enqueued with the middle priority (zero), and LIFO relative to messages with the same priority.
>
> **CpmEnqueueIFIFO(***pe, prio***)** - the message is transmitted to processor *pe* and enqueued with the specified integer-priority *prio*, and FIFO relative to messages with the same priority.
>
> **CpmEnqueueILIFO(***pe, prio***)** - the message is transmitted to processor *pe* and enqueued with the specified integer-priority *prio*, and LIFO relative to messages with the same priority.
>
> **CpmEnqueueBFIFO(***pe, priobits, prioptr***)** - the message is transmitted to processor *pe* and enqueued with the specified bitvector-priority, and FIFO relative to messages with the same priority.
>
> **CpmEnqueueBLIFO(***pe, priobits, prioptr***)** - the message is transmitted to processor *pe* and enqueued with the specified bitvector-priority, and LIFO relative to messages with the same priority.
>
> **CpmMakeThread(***pe***)** - The message is transmitted to processor *pe* where a CthThread is created, and the thread invokes the specified function.

All the functions shown above accept processor numbers as arguments. Instead of supplying a processor number, one can also supply the special symbols CPM_ALL or CPM_OTHERS, causing a broadcast. For example,

```
Cpm_print_integer(CpmMakeThread(CPM_ALL), 5);
```

would broadcast a message to all the processors causing each processor to create a thread, which would in turn invoke `print_integer` with the argument 5.

3.2 CPM Packing and Unpacking

Functions preceeded by the word **CpmInvokable** must have simple argument lists. In particular, the argument list of a CpmInvokable function can only contain cpm-single-arguments and cpm-array-arguments, as defined by this grammar:

```
cpm-single-argument :== typeword varname
cpm-array-argument  :== typeword '*' varname
```

When CPM sees the cpm-array-argument notation, CPM interprets it as being a pointer to an array. In this case, CPM attempts to pack an entire array into the message, whereas it only attempts to pack a single element in the case of the cpm-single-argument notation.

Each cpm-array-argument must be preceeded by a cpm-single-argument of type `CpmDim`. `CpmDim` is simply an alias for `int`, but when CPM sees an argument declared `CpmDim`, it knows that the next argument will be a cpm-array-argument, and it interprets the `CpmDim` argument to be the size of the array. Given a pointer to the array, its size, and its element-type, CPM handles the packing of array values as automatically as it handles single values.

A second program, `example2.c`, uses array arguments:

```
#include "example2.cpm.h"

CpmInvokable print_program_arguments(CpmDim argc, CpmStr *argv)
{
  int i;
  CmiPrintf("The program's arguments are: ");
  for (i=0; i<argc; i++) CmiPrintf("%s ", argv[i]);
  CmiPrintf("\n");
}

user_main(int argc, char **argv)
{
  CpmModuleInit();
  CpmInitializeThisModule();
  if (CmiMyPe()==0)
    Cpm_print_program_arguments(CpmSend(1), argc, argv);
}

main(int argc, char **argv)
{
  ConverseInit(argc, argv, user_main, 0, 0);
}
```

The word `CpmStr` is a CPM built-in type, it represents a null-terminated string:

```
typedef char *CpmStr;
```

Therefore, the function `print_program_arguments` takes exactly the same arguments as `user_main`. In this example, the main program running on processor 0 transmits the arguments to processor 1, which prints them out.

Thus far, we have only shown functions whose prototypes contain builtin CPM types. CPM has built-in knowledge of the following types: char, short, int, long, float, double, CpmDim, and CpmStr (pointer to a null-terminated string). However, you may also also transmit user-defined types in a CPM message.

For each (non-builtin) type the user wishes to pack, the user must supply some pack and unpack routines. The subroutines needed depend upon whether the type is a pointer or a simple type. Simple types are defined to be those that contain no pointers at all. Note that some types are neither pointers, nor simple types. CPM cannot currently handle such types.

CPM knows which type is which only through the following declarations:

```
CpmDeclareSimple(typeword);
CpmDeclarePointer(typeword);
```

The user must supply such declarations for each type that must be sent via CPM.

When packing a value `v` which is a simple type, CPM uses the following strategy. The generated code first converts `v` to network interchange format by calling `CpmPack_typename(&v)`, which must perform the conversion in-place. It then copies `v` byte-for-byte into the message and sends it. When the data arrives, it is extracted from the message and converted back using `CpmUnpack_typename(&v)`, again in-place. The user must supply the pack and unpack routines.

When packing a value `v` which is a pointer, the generated code determines how much space is needed in the message buffer by calling `CpmPtrSize_typename(v)`. It then transfers the data pointed to by `v` into the message using `CpmPtrPack_typename(p, v)` , where `p` is a pointer to the allocated space in the message buffer. When the message arrives, the generated code extracts the packed data from the message by calling `CpmPtrUnpack_typename(p)`. The unpack function must return a pointer to the unpacked data, which is allowed to still contain pointers to the message buffer (or simply be a pointer to the message buffer). When the invocation is done, the function `CpmPtrFree_typename(v)` is called to free any memory allocated by the unpack routine. The user must supply the size, pack, unpack, and free routines.

The following program fragment shows the declaration of two user-defined types:

```

typedef struct { double x,y; } coordinate;
CpmDeclareSimple(coordinate);

void CpmPack_coordinate(coordinate *p)
{
  CpmPack_double(&(p->x));
  CpmPack_double(&(p->y));
}

```

```
11:    void CpmPack_coordinate(coordinate *p)
12:    {
13:      CpmUnpack_double(&(p->x));
14:      CpmUnpack_double(&(p->y));
15:    }
16:
17:    typedef int *intptr;
18:    CpmDeclarePointer(intptr);
19:
20:    #define CpmPtrSize_intptr(p) sizeof(int)
21:
22:    void CpmPtrPack_intptr(void *p, intptr v)
23:    {
24:      *(int *)p = *v;
25:      CpmPack_int((int *)p);
26:    }
27:
28:    intptr CpmPtrUnpack_intptr(void *p)
29:    {
30:      CpmUnpack_int((int *)p);
31:      return (int *)p;
32:    }
33:
34:    #define CpmPtrFree_intptr(p) (0)
35:
36:    #include "example3.cpm.h"
37:    ...
```

The first type declared in this file is the coordinate. Line 2 contains the C type declaration, and line 3 notifies CPM that it is a simple type, containing no pointers. Lines 5-9 declare the pack function, which receives a pointer to a coordinate, and must pack it in place. It makes use of the pack-function for doubles, which also packs in place. The unpack function is similar.

The second type declared in this file is the intptr, which we intend to mean a pointer to a single integer. On line 18 we notify CPM that the type is a pointer, and that it should therefore use CpmPtrSize_intptr, CpmPtrPack_intptr, CpmPtrUnpack_intptr, and CpmPtrFree_intptr. Line 20 shows the size function, a constant: we always need just enough space to store one integer. The pack function copies the int into the message buffer, and packs it in place. The unpack function unpacks it in place, and returns an intptr, which points right to the unpacked integer which is still in the message buffer. Since the int is still in the message buffer, and not in dynamically allocated memory, the free function on line 34 doesn't have to do anything.

Note that the inclusion of the `.cpm.h` file comes after these type and pack declarations: the `.cpm.h` file will reference these functions and macros, therefore, they must already be defined.

3.3 Inventing New Types of CpmDestinations

It is possible for the user to create new types of CpmDestinations, and to write functions that return these new destinations. In order to do this, one must have a mental model of the steps performed when a Cpm message is sent. This knowledge is only necessary to those wishing to invent new kinds of destinations. Others can skip this section.

The basic steps taken when sending a CPM message are:

1. The destination-structure is created. The first argument to the launcher is a CpmDestination. Therefore, before the launcher is invoked, one typically calls a function (like CpmSend) to build the destination-structure.

2. The launcher allocates a message-buffer. The buffer contains space to hold a function-pointer and the function's arguments. It also contains space for an "envelope", the size of which is determined by a field in the destination-structure.

3. The launcher stores the function-arguments in the message buffer. In doing so, the launcher converts the arguments to a contiguous sequence of bytes.

4. The launcher sets the message's handler. For every launcher, there is a matching function called an *invoker*. The launcher's job is to put the argument data in the message and send the message. The *invoker*'s job is to extract the argument data from the message and call the user's function. The launcher uses `CmiSetHandler` to tell Converse to handle the message by calling the appropriate *invoker*.

5. The message is sent, received, and handled. The destination-structure contains a pointer to a *send-function*. The *send-function* is responsible for choosing the message's destination and making sure that it gets there and gets handled. The *send-function* has complete freedom to implement this in any manner it wishes. Eventually, though, the message should arrive at a destination and its handler should be called.

6. The user's function is invoked. The invoker extracts the function arguments from the message buffer and calls the user's function.

The *send-function* varies because messages take different routes to get to their final destinations. Compare, for example, CpmSend to CpmEnqueueFIFO. When CpmSend is used, the message goes straight to the target processor and gets handled. When CpmEnqueueFIFO is used, the message goes to the target processor, goes into the queue, comes out of the queue, and *then* gets handled. The *send-function* must implement not only the transmission of the message, but also the possible "detouring" of the message through queues or into threads.

We now show an example CPM command, and describe the steps that are taken when the command is executed. The command we will consider is this one:

```
Cpm_print_integer(CpmEnqueueFIFO(3), 12);
```

Which sends a message to processor 3, ordering it to call `print_integer(12)`.

The first step is taken by CpmEnqueueFIFO, which builds the CpmDestination. The following is the code for CpmEnqueueFIFO:

```
typedef struct CpmDestinationSend
{
  void *(*sendfn)();
  int envsize;
  int pe;
}
*CpmDestinationSend;

CpmDestination CpmEnqueueFIFO(int pe)
{
  static struct CpmDestinationSend ctrl;
  ctrl.envsize = sizeof(int);
  ctrl.sendfn  = CpmEnqueueFIFO1;
  ctrl.pe = pe;
  return (CpmDestination)&ctrl;
}
```

Notice that the CpmDestination structure varies, depending upon which kind of destination is being used. In this case, the destination structure contains a pointer to the send-function `CpmEnqueueFIFO1`, a field that controls the size of the envelope, and the destination-processor. In a CpmDestination, the `sendfn` and `envsize` fields are required, additional fields are optional.

After CpmEnqueueFIFO builds the destination-structure, the launcher Cpm_print_integer is invoked. Cpm_print_integer performs all the steps normally taken by a launcher:

1. **It allocates the message buffer.** In this case, it sets aside just enough room for one *int* as an envelope, as dictated by the destination-structure's *envsize* field.

2. **It stores the function-arguments in the message-buffer.** In this case, the function-arguments are just the integer 12.

3. **It sets the message's handler.** In this case, the message's handler is set to a function that will extract the arguments and call print_integer.

4. **It calls the send-function to send the message.**

The code for the send-function is here:

```
void *CpmEnqueueFIFO1(CpmDestinationSend dest, int len, void *msg)
{
  int *env = (int *)CpmEnv(msg);
  env[0] = CmiGetHandler(msg);
  CmiSetHandler(msg, CpvAccess(CpmEnqueueFIFO2_Index));
  CmiSyncSendAndFree(dest->pe,len,msg);
}
```

The send-function CpmEnqueueFIFO1 starts by switching the handler. The original handler is removed using using `CmiGetHandler`. It is set aside in the message buffer in the "envelope" space described earlier — notice the use of `CpmEnv` to obtain the envelope. This is the purpose of the envelope in the message — it is a place where the send-function can store information. The destination-function must anticipate how much space the send-function will need, and it must specify that amount of space in the destination-structure field *envsize*. In this case, the envelope is used to store the original handler, and the message's handler is set to an internal function called `CpmEnqueueFIFO2`.

After switching the handler, `CpmEnqueueFIFO1` sends the message. Eventually, the message will be received by `CsdScheduler`, and its handler will be called. The result will be that `CpmEnqueueFIFO2` will be called on the destination processor. Here is the code for `CpmEnqueueFIFO2`:

```
void CpmEnqueueFIFO2(void *msg)
{
  int *env;
  CmiGrabBuffer(&msg);
  env = (int *)CpmEnv(msg);
  CmiSetHandler(msg, env[0]);
  CsdEnqueueFIFO(msg);
}
```

This function takes ownership of the message-buffer from Converse using `CmiGrabBuffer`. It extracts the original handler from the envelope (the handler that calls `print_integer`), and restores it using `CmiSetHandler`. Having done so, it enqueues the message with the FIFO queueing policy. Eventually, the scheduler picks the message from the queue, and `print_integer` is invoked.

In summary, the procedure for implementing new kinds of destinations is to write one send-function, one function returning a CpmDestination (which contains a reference to the send-function), and one or more Converse handlers to manipulate the message.

The destination-function must return a pointer to a "destination-structure", which can in fact be any structure matching the following specifications:

- The first field must be a pointer to a send-function,
- The second field must the an integer, the envelope-size.

This pointer must be coerced to type CpmDestination.

The send-function must have the following prototype:

```
void sendfunction(CpmDestination dest, int msglen, void *msgptr)
```

It can access the envelope of the message using CpmEnv:

```
int *CpmEnv(void *msg);
```

It can also access the data stored in the destination-structure by the destination-function.

Chapter 4

Load Balancing

Converse provides the programmer with a number of Load Balancing strategies in order to distribute work among the processors in the system. The load balancing strategy to be used in the program can be specified at the link time. (Note: *In future versions, we will provide users with the ability to write their own load balancing strategies. However, in this version, the choice is limited to certain predefined load balancing strategies.*) The following functions are provided for interfacing a Converse program with the load balancing module.

void CldNewSeedFromLocal(void *msg, void *ldb, void (*sendfn)(),int queueing, int priolen, int *prioptr)
Hand over the message `msg`, which is freshly generated by the local processor, to the load balancing module. This load balancing module may use the space provided by `ldb` to either send the message off to other processors using `sendfn()`, or may decide to enqueue it locally using the queuing strategy `queueing` and priority specifiers `priolen` and `prioptr`.

void CldNewSeedFromNet(void *msg, void *ldb, void (*sendfn)(),int queueing, int priolen, int *prioptr)
Hand over the message `msg`, which is received from network, to the load balancing module. This load balancing module may use the space provided by `ldb` to either send the message off to other processors using `sendfn()`, or may decide to enqueue it locally using the queuing strategy `queueing` and priority specifiers `priolen` and `prioptr`.

The rationale behind this interface is as follows: A typical load balancing strategy can have different behaviors based on whether the message (`msg`) specifying work to be done is freshly generated by the local processor or has arrived from the network after one or more hops from the generating processor to local processor. (Note: *Message* `msg` *must have its handler field set before it is handed over to the load balancing module.*) If the work is freshly issued, one hands it over to the load balancing module using function `CldNewSeedFromLocal()` otherwise calls `CldNewSeedFromNet()`. Each load balancing strategy may have different use of the parameter `ldb`. Some of the currently implemented language specific load balancing strategies use this area in the message for sending the load information on local processor piggy-backed onto the message periodically; some of them may not need this area at all. (Note: *The only ldb strategy available to Converse users currently, i.e. random, does not use this area and the user could specify* `void * (0))` *there.*)

Some of the parallel languages or libraries built on top of Converse may have the message packing mechanisms which have to be called before any message crosses address boundaries. Therefore the load balancing module cannot use built-in functions such as `CmiAsyncSend()` for sending the work off to another processor. For this reason, the user needs to specify a function `sendfn()` which takes two parameters: `void *msgptr` and `int destPE`. Languages or runtimes using packing can pack the message before sending it to processor `destPE` in `sendfn()`. However, if the load balancing module decides not to send `msg` to any other processor, it may queue it to the processors local queue. For this eventuality, one has to specify the queuing parameters in above calls. They are: `queuing`–a queuing strategy (see `CsdEnqueueGeneral()`); `priolen`–length of the priority field; and `prioptr`–pointer to the priority information. If the user is not using priorities, `priolen` should be 0.

Chapter 5

Futures

This library supports the *future* abstraction, defined and used by Halstead and other researchers.

Cfuture CfutureCreate()

Returns the handle of an empty future. The future is said to reside on the processor that created it. The handle is a *global* reference to the future, in other words, it may be copied freely across processors. However, while the handle may be moved across processors freely, some operations can only be performed on the processor where the future resides.

Cfuture CfutureSet(Cfuture future, void *value, int nbytes)

Makes a copy of the value and stores it in the future. CfutureSet may be performed on processors other than the one where the future resides. If done remotely, the copy of the value is created on the processor where the future resides.

void *CfutureWait(Cfuture fut)

Waits until the future has been filled, then returns a pointer to the contents of the future. If the future has already been filled, this happens immediately (without blocking). If the destroy flag is set, the future is destroyed (freed) in the process. The pointer returned by CfutureWait must eventually be freed using CfutureDestroyValue below. Caution: CfutureWait can only be done on the processor where the Cfuture resides. A second caution: blocking operations (such as this one) can only be done in user-created threads.

void CfutureDestroy(Cfuture f)

Frees the space used by the specified Cfuture. This also frees the value stored in the future. Caution: this operation can only be done on the processor where the Cfuture resides.

void* CfutureCreateValue(int nbytes)

Allocates the specified amount of memory and returns a pointer to it. The memory can be freed by CfutureDestroyValue, just as if it had been extracted from a future. It can also be filled with data and stored into a future, using CfutureStoreBuffer below.

void CfutureStoreValue(Cfuture fut, void *value)

Make a copy of the value and stores it in the future, destroying the original copy of the value. This may be significantly faster than the more general function, CfutureSet (it may avoid copying). This function can *only* used to store values that were previously extracted from other futures, or values that were allocated using CfutureCreateValue.

void CfutureModuleInit()

This function initializes the futures module. It must be called once on each processor, during the handler-registration process (see the Converse manual regarding CmiRegisterHandler).

Chapter 6

Converse-POSIX threads

We have implemented the POSIX threads API on top of Converse threads. To use the Converse-pthreads, you must include the header file:

```
#include <cpthreads.h>
```

Refer to the POSIX threads documentation for the documentation on the pthreads functions and types. Although Converse-pthreads threads are POSIX-compliant in most ways, there are some specific things one needs to know to use our implementation.

6.1 Pthreads and Converse

Our pthreads implementation is designed to exist within a Converse environment. For example, to send messages inside a POSIX program, you would still use the usual Converse messaging primitives.

6.2 Suppressing Name Conflicts

Some people may wish to use Converse pthreads on machines that already have a pthreads implementation in the standard library. This may cause some name-conflicts as we define the pthreads functions, and the system include files do too. To avoid such conflicts, we provide an alternative set of names beginning with the word Cpthread. These names are interchangable with their pthread equivalents. In addition, you may prevent Converse from defining the pthread names at all with the preprocessor symbol SUPPRESS_PTHREADS:

```
#define SUPPRESS_PTHREADS
#include <cpthreads.h>
```

6.3 Interoperating with Other Thread Packages

Converse programs are typically multilingual programs. There may be modules written using POSIX threads, but other modules may use other thread APIs. The POSIX threads implementation has the following restriction: you may only call the pthreads functions from inside threads created with pthread_create. Threads created by other thread packages (for example, the CthThread package) may not use the pthreads functions.

6.4 Preemptive Context Switching

Most implementations of POSIX threads perform time-slicing: when a thread has run for a while, it automatically gives up the CPU to another thread. Our implementation is currently nonpreemptive (no time-slicing). Threads give up control at two points:

- If they block (eg, at a mutex).
- If they call pthread_yield().

Usually, the first rule is sufficient to make most programs work. However, a few programs (particularly, those that busy-wait) may need explicit insertion of yields.

6.5 Limits on Blocking Operations in main

Converse has a rule about blocking operations — there are certain pieces of code that may not block. This was an efficiency decision. In particular, the main function, Converse handlers, and the converse startup function (see ConverseInit) may not block. You must be aware of this when using the POSIX threads functions with Converse.

There is a contradition here — the POSIX standard requires that the pthreads functions work from inside `main`. However, many of them block, and Converse forbids blocking inside `main`. This contradition can be resolved by renaming your posix-compliant `main` to something else: for example, `mymain`. Then, through the normal Converse startup procedure, create a POSIX thread to run `mymain`. We provide a convenience function to do this, called Cpthreads_start_main. The startup code will be much like this:

```
void mystartup(int argc, char **argv)
{
  CpthreadModuleInit();
  Cpthreads_start_main(mymain, argc, argv);
}

int main(int argc, char **argv)
{
  ConverseInit(mystartup, argc, argv, 0, 0);
}
```

This creates the first POSIX thread on each processor, which runs the function mymain. The mymain function is executing in a POSIX thread, and it may use any pthread function it wishes.

6.6 CpthreadModuleInit

On each processor, the function CpthreadModuleInit must be called before any other pthread function is called. This is shown in the example in the previous section.