Converse Extensions Library

August 31, 2000

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

1 Introduction **3**

2 Tag Matching **4**

3 Converse Pseudorandom Number Generator **6**

4 Automatic Parameter Marshalling **8**

4.1 CPM Basics 8

4.2 CPM Packing and Unpacking 11

4.3 Inventing New Types of CpmDestinations 14

5 Load Balancing **17**

5.1 Using Converse Load Balancers 17

5.2 How to Write a Load Balancer for Converse/Charm++ 20

5.2.1 Introduction 20

5.2.2 Existing Load Balancers and Provided Utilities 21

5.3 A Sample Load Balancer 21

5.3.1 Minimal Requirements 21

5.3.2 Provided Load Balancing Facilities 25

5.3.3 Finishing the `Test` Balancer 27

6 Futures **30**

7 Converse-POSIX threads **32**

7.1 Pthreads and Converse 32

7.2 Suppressing Name Conflicts 32

7.3 Interoperating with Other Thread Packages 33
7.4 Preemptive Context Switching 33
7.5 Limits on Blocking Operations in main 33
7.6 CpthreadModuleInit 34

8 Parallel Arrays of Threads **35**

8.1 Creating Arrays of Threads 35
8.2 Mapping Functions for Arrays of Threads 35
8.3 Thread Functions for Arrays of Threads 36
8.4 Sending Messages to Threads 36
8.5 Performing Reductions over Array Elements 37

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

Converse Pseudorandom Number Generator

Converse provides three different Linear Congruential Random Number Generators. Each random number stream has a cycle length of 2^{64} as opposed to ANSI C standard's 2^{48}. Also, each of the three random number streams can be split into a number of per processor streams, so that the random number sequences can be computed in parallel, and are reproducible. Furthermore, there is no implicit critical section in the random number generator,and yet, this functionality is thread-safe, because all the state information is stored in the structure allocated by the programmer. Further, this state information is stored in a first class object, and can be passed to other processors through messages. This module of Converse is based on the public-domain SPRNG[1] package developed by Ashok Srinivasan[2] at NCSA.

For minimal change to programs already using C functions `rand()`, `srand()`, and `drand48()`, Converse also maintains a "default" random number stream.

Interface to the Converse Pseudorandom Number Generator module is as follows:

typedef ... CrnStream;
State information for generating the next random number in the sequence.

void CrnInitStream(CrnStream *stream, int seed, int type)
Initializes the new random number stream `stream` of `type` using `seed`. `type` can have values 0, 1, or 2 to represent three types of linear congruential random number generators.

int CrnInt(CrnStream *stream)
Returns an integer between 0 and $2^{31} - 1$ corresponding to the next random number in the sequence associated with `stream`. Advances `stream` by one in the sequence.

double CrnDouble(CrnStream *stream)
Returns an double precision floating point number between 0 and 1 corresponding to the next random number in the sequence associated with `stream`. Advances `stream` by one in the sequence.

float CrnFloat(CrnStream *stream)

[1] URL:`http://www.ncsa.uiuc.edu/Apps/SPRNG/www/`
[2] Email:`ashoks@ncsa.uiuc.edu`

Returns a single precision floating point number between 0 and 1 corresponding to the next random number in the sequence associated with `stream`. Advances `stream` by one in the sequence.

void CrnSrand(int seed)
Specifies a different seed for the default random number stream. Replaces `srand()`.

int CrnRand(void)
Generate the next integer random number from the default random number stream. Replaces `rand()`.

double CrnDrand(void)
Generate the next double precision random number from the default random number stream. Replaces `drand48()`.

Chapter 4

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.

4.1 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 *CpmDestination*s. 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.

4.2 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:

```
 1:    #include "example2.cpm.h"
 2:
 3:    CpmInvokable print_program_arguments(CpmDim argc, CpmStr *argv)
 4:    {
 5:      int i;
 6:      CmiPrintf("The program's arguments are: ");
 7:      for (i=0; i<argc; i++) CmiPrintf("%s ", argv[i]);
 8:      CmiPrintf("\n");
 9:    }
10:
11:    user_main(int argc, char **argv)
12:    {
13:      CpmModuleInit();
14:      CpmInitializeThisModule();
15:      if (CmiMyPe()==0)
16:        Cpm_print_program_arguments(CpmSend(1), argc, argv);
17:    }
18:
19:    main(int argc, char **argv)
20:    {
21:      ConverseInit(argc, argv, user_main, 0, 0);
22:    }
```

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:

```
 1:
 2:    typedef struct { double x,y; } coordinate;
 3:    CpmDeclareSimple(coordinate);
 4:
 5:    void CpmPack_coordinate(coordinate *p)
 6:    {
 7:      CpmPack_double(&(p->x));
 8:      CpmPack_double(&(p->y));
 9:    }
10:
```

```
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.

4.3 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 5

Load Balancing

5.1 Using Converse Load Balancers

This module defines a function **CldEnqueue** that sends a message to a lightly-loaded processor. It automates the process of finding a lightly-loaded processor.

The function **CldEnqueue** is extremely sophisticated. It does not choose a processor, send the message, and forget it. Rather, it puts the message into a pool of movable work. The pool of movable work gradually shrinks as it is consumed (processed), but in most programs, there is usually quite a bit of movable work available at any given time. As load conditions shift, the load balancers shifts the pool around, compensating. Any given message may be shifted more than once, as part of the pool.

CldEnqueue also accounts for priorities. Normal load-balancers try to make sure that all processors have some work to do. The function **CldEnqueue** goes a step further: it tries to make sure that all processors have some reasonably high-priority work to do. This can be extremely helpful in AI search applications.

The two assertions above should be qualified: **CldEnqueue** can use these sophisticated strategies, but it is also possible to configure it for different behavior. When you compile and link your program, you choose a *load-balancing strategy*. That means you link in one of several implementations of the load-balancer. Most are sophisticated, as described above. But some are simple and cheap, like the random strategy. The process of choosing a strategy is described in the manual *Converse Installation and Usage*.

Before you send a message using **CldEnqueue**, you must write an *info* function with this prototype:

void InfoFn(void *msg, CldPackFn *pfn, int *len, int *queueing, int *priobits, unsigned int *prioptr);
The load balancer will call the info function when it needs to know various things about the message. The load balancer will pass in the message via the parameter `msg`. The info function's job is to "fill in" the other parameters. It must compute the length of the message, and store it at `*len`. It must determine the *pack* function for the message, and store a pointer to it at `*pfm`. It must identify the priority of the message, and the queueing strategy that must be used,

storing this information at `*queueing`, `*priobits`, and `*prioptr`. Caution: the priority will not be copied, so the `*prioptr` should probably be made to point to the message itself.

After the user of **CldEnqueue** writes the "info" function, the user must register it, using this:

int CldRegisterInfoFn(CldInfoFn fn)
Accepts a pointer to an info-function. Returns an integer index for the info-function. This index will be needed in **CldEnqueue**.

Normally, when you send a message, you pack up a bunch of data into a message, send it, and unpack it at the receiving end. It is sometimes possible to perform an optimization, though. If the message is bound for a processor within the same address space, it isn't always necessary to copy all the data into the message. Instead, it may be sufficient to send a message containing only a pointer to the data. This saves much packing, unpacking, and copying effort. It is frequently useful, since in a properly load-balanced program, a great many messages stay inside a single address space.

With CldEnqueue, you don't know in advance whether a message is going to cross address-space boundaries or not. If it's to cross address spaces, you need to use the "long form", but if it's to stay inside an address space, you want to use the faster "short form". We call this "conditional packing." When you send a message with **CldEnqueue**, you should initially assume it will not cross address space boundaries. In other words, you should send the "short form" of the message, containing pointers. If the message is about to leave the address space, the load balancer will call your pack function, which must have this prototype:

void PackFn(void **msg)
The pack function is handed a pointer to a pointer to the message (yes, a pointer to a pointer). The pack function is allowed to alter the message in place, or replace the message with a completely different message. The intent is that the pack function should replace the "short form" of the message with the "long form" of the message. Note that if it replaces the message, it should CmiFree the old message.

Of course, sometimes you don't use conditional packing. In that case, there is only one form of the message. In that case, your pack function can be a no-op.

Pack functions must be registered using this:

int CldRegisterPackFn(CldPackFn fn)
Accepts a pointer to an pack-function. Returns an integer index for the pack-function. This index will be needed in **CldEnqueue**.

Normally, **CldEnqueue** sends a message to a lightly-loaded processor. After doing this, it enqueues the message with the appropriate priority. The function CldEnqueue can also be used as a mechanism to simply enqueue a message on a remote processor with a priority. In other words, it can be used as a prioritized send-function. To do this, one of the CldEnqueue parameters allows you to override the load-balancing behavior and lets you choose a processor yourself.

The prototype for **CldEnqueue** is as follows:

void CldEnqueue(int pe, void *msg, int infofn)
The argument `msg` is a pointer to the message. The parameter `infofn` represents a function that can analyze the message. The parameter `packfn` represents a function that can pack the mes-

sage. If the parameter pe is CLD_ANYWHERE, the message is sent to a lightly-loaded processor and enqueued with the appropriate priority. If the parameter pe is a processor number, the message is sent to the specified processor and enqueued with the appropriate priority. **CldEnqueue** frees the message buffer using **CmiFree**.

The following simple example illustrates how a Converse program can make use of the load balancers.

hello.c:

```
#include <stdio.h>
#include "converse.h"
#define CHARES 10

void startup(int argc, char *argv[]);
void registerAndInitialize();

typedef struct pemsgstruct
{
  char header[CmiExtHeaderSizeBytes];
  int pe, id, pfnx;
  int queuing, priobits;
  unsigned int prioptr;
} pemsg;

CpvDeclare(int, MyHandlerIndex);
CpvDeclare(int, InfoFnIndex);
CpvDeclare(int, PackFnIndex);

int main(int argc, char *argv[])
{
  ConverseInit(argc, argv, startup, 0, 0);
  CsdScheduler(-1);
}

void startup(int argc, char *argv[])
{
  pemsg *msg;
  int i;

  registerAndInitialize();
  for (i=0; i<CHARES; i++) {
    msg = (pemsg *)malloc(sizeof(pemsg));
    msg->pe = CmiMyPe();
    msg->id = i;
    msg->pfnx = CpvAccess(PackFnIndex);
    msg->queuing = CQS_QUEUEING_FIFO;
    msg->priobits = 0;
```

```
    msg->prioptr = 0;
    CmiSetHandler(msg, CpvAccess(MyHandlerIndex));
    CmiPrintf("[%d] sending message %d\n", msg->pe, msg->id);
    CldEnqueue(CLD_ANYWHERE, msg, CpvAccess(InfoFnIndex));
    /*    CmiSyncSend(i, sizeof(pemsg), &msg); */
  }
}

void MyHandler(pemsg *msg)
{
  CmiPrintf("Message %d created on %d handled by %d.\n", msg->id, msg->pe,
    CmiMyPe());
}

void InfoFn(pemsg *msg, CldPackFn *pfn, int *len, int *queuing, int *priobits,
    unsigned int *prioptr)
{
  *pfn = (CldPackFn)CmiHandlerToFunction(msg->pfnx);
  *len = sizeof(pemsg);
  *queuing = msg->queuing;
  *priobits = msg->priobits;
  prioptr = &(msg->prioptr);
}

void PackFn(pemsg **msg)
{
}

void registerAndInitialize()
{
  CpvInitialize(int, MyHandlerIndex);
  CpvAccess(MyHandlerIndex) = CmiRegisterHandler(MyHandler);
  CpvInitialize(int, InfoFnIndex);
  CpvAccess(InfoFnIndex) = CldRegisterInfoFn((CldInfoFn)InfoFn);
  CpvInitialize(int, PackFnIndex);
  CpvAccess(PackFnIndex) = CldRegisterPackFn((CldPackFn)PackFn);
}
```

5.2 How to Write a Load Balancer for Converse/Charm++

5.2.1 Introduction

This manual details how to write your own general-purpose message-based load balancer for Converse. A Converse load balancer can be used by any Converse program, but also serves as a *seed* load balancer for Charm++ chare creation messages. Specifically, to use a load balancer, you would pass messages to CldEnqueue rather than directly to the scheduler. This is the default

behavior with chare creation message in Charm++. Thus, the primary provision of a new load balancer is an implementation of the CldEnqueue function.

5.2.2 Existing Load Balancers and Provided Utilities

Throughout this manual, we will occasionally refer to the source code of two provided load balancers, the random initial placement load balancer (`cldb.rand.c`) and the graph-based load balancer (`cldb.graph.c`). The functioning of these balancers is described in the Charm++ Extensions manual load balancing section.

In addition, a special utility is provided that allows us to add and remove load-balanced messages from the scheduler's queue. The source code for this is available in `cldb.c`. The usage of this utility will also be described here in detail.

5.3 A Sample Load Balancer

This manual steps through the design of a load balancer using an example which we will call `test`. The `test` load balancer has each processor periodically send half of its load to its neighbor in a ring. Specifically, for N processors, processor K will send approximately half of its load to (K+1)%N, every 100 milliseconds (this is an example only; we leave the genius approaches up to you).

5.3.1 Minimal Requirements

The minimal requirements for a load balancer are illustrated by the following code.

```
#include <stdio.h>
#include "converse.h"

char *CldGetStrategy(void)
{
  return "test";
}

CpvDeclare(int, CldHandlerIndex);

void CldHandler(void *msg)
{
  CldInfoFn ifn; CldPackFn pfn;
  int len, queueing, priobits; unsigned int *prioptr;

  CmiGrabBuffer((void **)&msg);
  CldRestoreHandler(msg);
  ifn = (CldInfoFn)CmiHandlerToFunction(CmiGetInfo(msg));
```

```
  ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
  CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
}

void CldEnqueue(int pe, void *msg, int infofn)
{
  int len, queueing, priobits; unsigned int *prioptr;
  CldInfoFn ifn = (CldInfoFn)CmiHandlerToFunction(infofn);
  CldPackFn pfn;

  if (pe == CLD_ANYWHERE) {
    /* do what you want with the message; in this case we'll just keep
       it local */
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    CmiSetInfo(msg,infofn);
    CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
  }
  else {
    /* pe contains a particular destination or broadcast */
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    if (pfn) {
      pfn(&msg);
      ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    }
    CldSwitchHandler(msg, CpvAccess(CldHandlerIndex));
    CmiSetInfo(msg,infofn);
    if (pe==CLD_BROADCAST)
      CmiSyncBroadcastAndFree(len, msg);
    else if (pe==CLD_BROADCAST_ALL)
      CmiSyncBroadcastAllAndFree(len, msg);
    else CmiSyncSendAndFree(pe, len, msg);
  }
}

void CldModuleInit()
{
  CpvInitialize(int, CldHandlerIndex);
  CpvAccess(CldHandlerIndex) = CmiRegisterHandler(CldHandler);
  CldModuleGeneralInit();
}
```

The primary function a load balancer must provide is the **CldEnqueue** function, which has the following prototype:

`void` **CldEnqueue**`(int pe, void *msg, int infofn);`

This function takes three parameters: `pe`, `msg` and `infofn`. `pe` is the intended destination of the `msg`. `pe` may take on one of the following values:

- Any valid processor number - the message must be sent to that processor
- `CLD_ANYWHERE` - the message can be placed on any processor
- `CLD_BROADCAST` - the message must be sent to all processors excluding the local processor
- `CLD_BROADCAST_ALL` - the message must be sent to all processors including the local processor

CldEnqueue must handle all of these possibilities. The only case in which the load balancer should get control of a message is when `pe = CLD_ANYWHERE`. All other messages must be sent off to their intended destinations and passed on to the scheduler as if they never came in contact with the load balancer.

The integer parameter `infofn` is a handler index for a user-provided function that allows CldEnqueue to extract information about (mostly components of) the message `msg`.

Thus, an implementation of the **CldEnqueue** function might have the following structure:

```
void CldEnqueue(int pe, void *msg, int infofn)
{
  ...
  if (pe == CLD_ANYWHERE)
    /* These messages can be load balanced */
  else if (pe == CmiMyPe())
    /* Enqueue the message in the scheduler locally */
  else if (pe==CLD_BROADCAST)
    /* Broadcast to all but self */
  else if (pe==CLD_BROADCAST_ALL)
    /* Broadcast to all plus self */
  else /* Specific processor number was specified */
    /* Send to specific processor */
}
```

In order to fill in the code above, we need to know more about the message before we can send it off to a scheduler's queue, either locally or remotely. For this, we have the info function. The prototype of an info function must be as follows:

```
void ifn(void *msg, CldPackFn *pfn, int *len, int *queueing, int *priobits, unsigned
int **prioptr);
```

Thus, to use the info function, we need to get the actual function via the handler index provided to **CldEnqueue**. Typically, **CldEnqueue** would contain the following declarations:

```
  int len, queueing, priobits;
  unsigned int *prioptr;
  CldPackFn pfn;
  CldInfoFn ifn = (CldInfoFn)CmiHandlerToFunction(infofn);
```

Subsequently, a call to `ifn` would look like this:

```
ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
```

The info function extracts information from the message about its size, queuing strategy and priority, and also a pack function, which will be used when we need to send the message elsewhere. For now, consider the case where the message is to be locally enqueued:

```
...
else if (pe == CmiMyPe())
  {
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
  }
...
```

Thus, we see the info function is used to extract info from the message that is necessary to pass on to **CsdEnqueueGeneral**.

In order to send the message to a remote destination and enqueue it in the scheduler, we need to pack it up with a special pack function so that it has room for extra handler information and a reference to the info function. Therefore, before we handle the last three cases of **CldEnqueue**, we have a little extra work to do:

```
...
else
  {
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    if (pfn) {
      pfn(&msg);
      ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    }
    CldSwitchHandler(msg, CpvAccess(CldHandlerIndex));
    CmiSetInfo(msg,infofn);
    ...
```

Calling the info function once gets the pack function we need, if there is one. We then call the pack function which rearranges the message leaving space for the info function, which we will need to call on the message when it is received at its destination, and also room for the extra handler that will be used on the receiving side to do the actual enqueuing. **CldSwitchHandler** is used to set this extra handler, and the receiving side must restore the original handler.

In the above code, we call the info function again because some of the values may have changed in the packing process.

Finally, we handle our last few cases:

```
...
    if (pe==CLD_BROADCAST)
```

```
        CmiSyncBroadcastAndFree(len, msg);
      else if (pe==CLD_BROADCAST_ALL)
        CmiSyncBroadcastAllAndFree(len, msg);
      else CmiSyncSendAndFree(pe, len, msg);
    }
}
```

The above example also provides **CldHandler** which is used to receive messages that **CldEnqueue** forwards to other processors.

```
CpvDeclare(int, CldHandlerIndex);

void CldHandler(void *msg)
{
  CldInfoFn ifn; CldPackFn pfn;
  int len, queueing, priobits; unsigned int *prioptr;

  CmiGrabBuffer((void **)&msg);
  CldRestoreHandler(msg);
  ifn = (CldInfoFn)CmiHandlerToFunction(CmiGetInfo(msg));
  ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
  CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
}
```

Note that the **CldHandler** properly restores the message's original handler using **CldRestoreHandler**, and calls the info function to obtain the proper parameters to pass on to the scheduler. We talk about this more below.

Finally, Converse initialization functions call **CldModuleInit** to initialize the load balancer module.

```
void CldModuleInit()
{
  CpvInitialize(int, CldHandlerIndex);
  CpvAccess(CldHandlerIndex) = CmiRegisterHandler(CldHandler);
  CldModuleGeneralInit();

  /* call other init processes here */
  CldBalance();
}
```

5.3.2 Provided Load Balancing Facilities

Converse provides a number of structures and functions to aid in load balancing (see cldb.c). Foremost amongst these is a method for queuing tokens of messages in a processor's scheduler in a way that they can be removed and relocated to a different processor at any time. The interface for this module is as follows:

```
void CldSwitchHandler(char *cmsg, int handler)
void CldRestoreHandler(char *cmsg)
int CldCountTokens()
int CldLoad()
void CldPutToken(char *msg)
void CldGetToken(char **msg)
void CldModuleGeneralInit()
```

Messages normally have a handler index associated with them, but in addition they have extra space for an additional handler. This is used by the load balancer when we use an intermediate handler (typically **CldHandler**) to handle the message when it is received after relocation. To do this, we use **CldSwitchHandler** to temporarily swap the intended handler with the load balancer handler. When the message is received, **CldRestoreHandler** is used to change back to the intended handler.

CldPutToken puts a message in the scheduler queue in such a way that it can be retrieved from the queue. Once the message gets handled, it can no longer be retrieved. **CldGetToken** retrieves a message that was placed in the scheduler queue in this way. **CldCountTokens** tells you how many tokens are currently retrievable. **CldLoad** gives a slightly more accurate estimate of message load by counting the total number of messages in the scheduler queue.

CldModuleGeneralInit is used to initialize this load balancer helper module. It is typically called from the load balancer's **CldModuleInit** function.

The helper module also provides the following functions:

```
void CldMultipleSend(int pe, int numToSend)
int CldRegisterInfoFn(CldInfoFn fn)
int CldRegisterPackFn(CldPackFn fn)
```

CldMultipleSend is generally useful for any load balancer that sends multiple messages to one processor. It works with the token queue module described above. It attempts to retrieve up to `numToSend` messages, and then packs them together and sends them, via CmiMultipleSend, to `pe`. If the number and/or size of the messages sent is very large, **CldMultipleSend** will transmit them in reasonably sized parcels. In addition, the **CldBalanceHandler** and its associated declarations and initializations are required to use it.

You may want to use the three status variables. These can be used to keep track of what your LB is doing (see usage in cldb.graph.c and itc++queens program).

```
CpvDeclare(int, CldRelocatedMessages);
CpvDeclare(int, CldLoadBalanceMessages);
CpvDeclare(int, CldMessageChunks);
```

The two register functions register *info* and *pack* functions, returning an index for the functions. Info functions are used by the load balancer to extract the various components from a message. Amongst these components is the pack function index. If necessary, the pack function can be

used to pack a message that is about to be relocated to another processor. Information on how to write info and pack functions is available in the load balancing section of the Converse Extensions manual.

5.3.3 Finishing the `Test` Balancer

The `test` balancer is a somewhat silly strategy in which every processor attempts to get rid of half of its load by periodically sending it to someone else, regardless of the load at the destination. Hopefully, you won't actually use this for anything important!

The `test` load balancer is available in charm/src/Common/conv-ldb/cldb.test.c. To try out your own load balancer you can use this filename and SUPER_INSTALL will compile it and you can link it into your Charm++ programs with -balance test. (To add your own new balancers permanently and give them another name other than "test" you will need to change the Makefile used by SUPER_INSTALL. Don't worry about this for now.) The cldb.test.c provides a good starting point for new load balancers.

Look at the code for the `test` balancer below, starting with the **CldEnqueue** function. This is almost exactly as described earlier. One exception is the handling of a few extra cases: specifically if we are running the program on only one processor, we don't want to do any load balancing. The other obvious difference is in the first case: how do we handle messages that can be load balanced? Rather than enqueuing the message directly with the scheduler, we make use of the token queue. This means that messages can later be removed for relocation. **CldPutToken** adds the message to the token queue on the local processor.

```
#include <stdio.h>
#include "converse.h"
#define PERIOD 100
#define MAXMSGBFRSIZE 100000

char *CldGetStrategy(void)
{
  return "test";
}

CpvDeclare(int, CldHandlerIndex);
CpvDeclare(int, CldBalanceHandlerIndex);
CpvDeclare(int, CldRelocatedMessages);
CpvDeclare(int, CldLoadBalanceMessages);
CpvDeclare(int, CldMessageChunks);

void CldDistributeTokens()
{
  int destPe = (CmiMyPe()+1)%CmiNumPes(), numToSend;

  numToSend = CldLoad() / 2;
  if (numToSend > CldCountTokens())
```

```
    numToSend = CldCountTokens() / 2;
  if (numToSend > 0)
    CldMultipleSend(destPe, numToSend);
  CcdCallFnAfter((CcdVoidFn)CldDistributeTokens, NULL, PERIOD);
}

void CldBalanceHandler(void *msg)
{
  CmiGrabBuffer((void **)&msg);
  CldRestoreHandler(msg);
  CldPutToken(msg);
}

void CldHandler(void *msg)
{
  CldInfoFn ifn; CldPackFn pfn;
  int len, queueing, priobits; unsigned int *prioptr;

  CmiGrabBuffer((void **)&msg);
  CldRestoreHandler(msg);
  ifn = (CldInfoFn)CmiHandlerToFunction(CmiGetInfo(msg));
  ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
  CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
}

void CldEnqueue(int pe, void *msg, int infofn)
{
  int len, queueing, priobits; unsigned int *prioptr;
  CldInfoFn ifn = (CldInfoFn)CmiHandlerToFunction(infofn);
  CldPackFn pfn;

  if ((pe == CLD_ANYWHERE) && (CmiNumPes() > 1)) {
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    CmiSetInfo(msg,infofn);
    CldPutToken(msg);
  }
  else if ((pe == CmiMyPe()) || (CmiNumPes() == 1)) {
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    CmiSetInfo(msg,infofn);
    CsdEnqueueGeneral(msg, queueing, priobits, prioptr);
  }
  else {
    ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    if (pfn) {
      pfn(&msg);
      ifn(msg, &pfn, &len, &queueing, &priobits, &prioptr);
    }
```

```
    CldSwitchHandler(msg, CpvAccess(CldHandlerIndex));
    CmiSetInfo(msg,infofn);
    if (pe==CLD_BROADCAST)
      CmiSyncBroadcastAndFree(len, msg);
    else if (pe==CLD_BROADCAST_ALL)
      CmiSyncBroadcastAllAndFree(len, msg);
    else CmiSyncSendAndFree(pe, len, msg);
  }
}

void CldModuleInit()
{
  CpvInitialize(int, CldHandlerIndex);
  CpvAccess(CldHandlerIndex) = CmiRegisterHandler(CldHandler);
  CpvInitialize(int, CldBalanceHandlerIndex);
  CpvAccess(CldBalanceHandlerIndex) = CmiRegisterHandler(CldBalanceHandler);
  CpvInitialize(int, CldRelocatedMessages);
  CpvInitialize(int, CldLoadBalanceMessages);
  CpvInitialize(int, CldMessageChunks);
  CpvAccess(CldRelocatedMessages) = CpvAccess(CldLoadBalanceMessages) =
    CpvAccess(CldMessageChunks) = 0;
  CldModuleGeneralInit();
  if (CmiNumPes() > 1)
    CldDistributeTokens();
}
```

Now look two functions up from **CldEnqueue**. We have an additional handler besides the **CldHandler**: the **CldBalanceHandler**. The purpose of this special handler is to receive messages that can be still be relocated again in the future. Just like the first case of **CldEnqueue** uses **CldPutToken** to keep the message retrievable, **CldBalanceHandler** does the same with relocatable messages it receives. **CldHandler** is only used when we no longer want the message to have the potential for relocation. It places messages irretrievably in the scheduler queue.

Next we look at our initialization functions to see how the process gets started. The **CldModuleInit** function gets called by the common Converse initialization code and starts off the periodic load distribution process by making a call to **CldDistributeTokens**. The entirety of the balancing is handled by the periodic invocation of this function. It computes an approximation of half of the PE's total load (**CsdLength**()), and if that amount exceeds the number of movable messages (**CldCountTokens**()), we attempt to move all of the movable messages. To do this, we pass this number of messages to move and the number of the PE to move them to, to the **CldMultipleSend** function.

Chapter 6

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). 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. This buffer can be filled with data and stored into a future, using CfutureStoreBuffer below. This combination is faster than using CfutureSet directly.

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 7

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.

7.1 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.

7.2 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>
```

7.3 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.

7.4 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.

7.5 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.

7.6 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.

Chapter 8

Parallel Arrays of Threads

This module is CPath: Converse Parallel Array of Threads. It makes it simple to create arrays of threads, where the threads are distributed across the processors. It provides simple operations like sending a message to a thread, as well as group operations like multicasting to a row of threads, or reducing over an array of threads.

8.1 Creating Arrays of Threads

This module defines a data type CPath, also known as an "array descriptor". Arrays are created by the function CPathMakeArray, and individual threads are created using CPathMakeThread:

void CPathMakeArray(CPath *path, int threadfn, int mapfn, ...)
This function initiates the creation of an array of threads. It fills in the array descriptor `*path`. Each thread in the array starts executing the function represented by `threadfn`. The function `mapfn` represents a mapping function, controlling the layout of the array. This parameter must be followed by the dimensions of the array, and then a zero.

void CPathMakeThread(CPath *path, int startfn, int pe)
This function makes a zero-dimensional array of threads, in other words, just one thread.

8.2 Mapping Functions for Arrays of Threads

One of the parameters to CPathMakeArray is a "mapping function", which maps array elements to processors. Mapping functions must be registered. The integer index returned by the registration process is the number which is passed to CPathMakeArray. Mapping functions receive the array descriptor as a parameter, and may use it to determine the dimensions of the array.

unsigned int MapFn(CPath *path, int *indices)
This is a prototype map function, all mapping functions must have this parameter list. It accepts an array descriptor and a set of indices. It returns the processor number of the specified element.

int CPathRegisterMapper(void *mapfn)
Accepts a pointer to a mapping function, and returns an integer index for the function. This number can be used as a parameter to CPathMakeArray.

int CPathArrayDimensions(CPath *path)
Returns the number of dimensions in the specified array.

int CPathArrayDimension(CPath *path, int n)
Returns the nth dimension of the specified array.

8.3 Thread Functions for Arrays of Threads

Thread functions (the functions that the threads execute) must have the following prototype, and must be registered using the following registration function. The integer index returned by the registration process is the number which is passed to CPathMakeArray.

void ThreadFn(CPath *self, int *indices)
This is a prototype thread function. All thread-functions must have these parameters. When an array of threads is created, each thread starts executing the specified thread function. The function receives a pointer to a copy of the array's descriptor, and the array element's indices.

int CPathRegisterThreadFn(void *mapfn)
Accepts a pointer to a thread function, and returns an integer index for the function. This number can be used as a parameter to CPathMakeArray.

8.4 Sending Messages to Threads

Threads may send messages to each other using CPathSend, which takes a complicated set of paramters. The parameters are most easily described by a context-free grammar:

void CPathSend(dest-clause, tag-clause, data-clause, end-clause)
Where:

```
dest-clause :== CPATH_DEST ',' pathptr ',' index ',' index ',' ...
tag-clause  :== CPATH_TAG ',' tag
tag-clause  :== CPATH_TAGS ',' tag ',' tag ',' ... ',' 0
tag-clause  :== CPATH_TAGVEC ',' numtags ',' tagvector
data-clause :== CPATH_BYTES ',' numbytes ',' bufptr
end-clause  :== CPATH_END
```

The symbols `CPATH_DEST`, `CPATH_TAG`, `CPATH_TAGS`, `CPATH_TAGVEC`, `CPATH_BYTES`, `CPATH_END`, and the comma are terminal symbols. The symbols descriptor, index, tag, numtags, tagvector, numbytes, and bufptr all represent C expressions.

The dest-clause specifies which array and which indices the message is to go to. One must provide a pointer to an array descriptor and a set of indices. Any index may be either a normal index, or the wildcard `CPATH_ALL`. Using the wildcard causes a multicast. The tag-clause provides several

notations, all of which specify an array of one or more integer tags to be sent with the message. These tags can be used at the receiving end for pattern matching. The data-clause specifies the data to go in the message, as a sequence of bytes. The end-clause represents the end of the parameter list.

Messages sent with CPathSend can be received using CPathRecv, analyzed using CPathMsgDecodeBytes, and finally discarded with CPathMsgFree:

void *CPathRecv(tag-clause, end-clause)
The tag-clause and end-clause match the grammar for CPathSend. The function will wait until a message with the same tags shows up (it waits using the thread-blocking primitives, see Converse threads). If any position in the CPathRecv tag-vector is `CPATH_WILD`, then that one position is ignored. CPathRecv returns an "opaque CPath message". The message contains the data somewhere inside it. The data can be located using CPathMsgDecodeBytes, below. The opaque CPath message can be freed using CPathMsgFree below.

void CPathMsgDecodeBytes(void *msg, int *len, void *bytes)
Given an opaque CPath message (as sent by CPathSend and returned by CPathRecv), this function will locate the data inside it. The parameter `*len` is filled in with the data length, and `*bytes` is filled in with a pointer to the data bytes. Bear in mind that once you free the opaque CPath message, this pointer is no longer valid.

void CPathMsgFree(void *msg)
Frees an opaque CPath message.

8.5 Performing Reductions over Array Elements

An set of threads may participate in a reduction. All the threads wishing to participate must call CPathReduce. The parameters to CPathReduce are most easily described by a context-free grammar:

void CPathReduce(over-clause, tag-clause, red-clause, data-clause, dest-clause, end-clause)
Where:

```
over-clause :== CPATH_OVER ',' pathptr ',' index ',' index ',' ...
dest-clause :== CPATH_DEST ',' pathptr ',' index ',' index ',' ...
tag-clause  :== CPATH_TAG ',' tag
tag-clause  :== CPATH_TAGS ',' tag ',' tag ',' ... ',' 0
tag-clause  :== CPATH_TAGVEC ',' numtags ',' tagvector
data-clause :== CPATH_BYTES ',' vecsize ',' eltsize ',' data
red-clause  :== CPATH_REDUCER ',' redfn
end-clause  :== CPATH_END
```

The over-clause specifies the set of threads participating in the reduction. One or more of the indices should be `CPATH_ALL`, the wildcard value. All array elements matching the pattern are participating in the reduction. All participants must supply the same over-clause. The tags-clause specifies a vector of integer tags. All participants must supply the same tags. The reducer

represents the function used to combine data pairwise. All participants must supply the same reducer. The data-clause specifies the input-data, which is an array of arbitrary-sized values. All participants must agree on the vecsize and eltsize. The dest-clause specifies the recipient of the reduced data (which may contain `CPATH_ALL` again). The data is sent to the recipient. The results can be received with CPathRecv using the same tags specified in the CPathReduce. The results may be analyzed with CPathMsgDecodeReduction, and freed with CPathMsgFree.

void CPathMsgDecodeReduction(void *msg,int *vecsize,int *eltsize,void *bytes)
This function accepts an opaque CPath message which was created by a reduction. It locates the data within the message, and determines the vecsize and eltsize.

The function that combines elements pairwise must match this prototype, and be registered with the following registration function. It is the number returned by the registration function which must be passed to CPathReduce:

void ReduceFn(int vecsize, void *data1, void *data2)
The reduce function accepts two equally-sized arrays of input data. It combines the two arrays pairwise, storing the results in array 1.

int CPathRegisterReducer(void *fn)
Accepts a pointer to a reduction function, and returns an integer index for the function. This number can be used as a parameter to CPathReduce.

Index

CldEnqueue, 17
CmmFree, 4
CmmGet, 3
CmmNew, 3
CmmProbe, 4
CmmTable, 3
CmmWildCard, 3
CrnDouble, 5
CrnDrand, 6
CrnFloat, 6
CrnInitStream, 5
CrnInt, 5
CrnRand, 6
CrnSrand, 6
CrnStream, 5