Converse Programming Manual

December 4, 1998

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 Initialization and Completion **3**

2 Machine Interface and Scheduler **5**

2.1 Machine Model 5

2.2 Defining Handler Numbers 5

2.3 Writing Handler Functions 6

2.4 Building Messages 7

2.5 Sending Messages 7

2.6 Broadcasting Messages 9

2.7 Multicasting Messages 10

2.8 Scheduling Messages 11

2.9 Polling for Messages 12

2.10 Scheduler Callbacks 13

2.11 The Timer 14

2.12 Processor Ids 14

2.13 Global Variables and Utility functions 14

2.13.1 Converse PseudoGlobals 14

2.13.2 Utility Functions 16

2.13.3 Node-level Locks and other Synchronization Mechanisms 17

2.14 Input/Output 18

2.15 Spanning Tree Calls 18

3 Threads **19**

3.1 Basic Thread Calls 19

3.2 Thread Scheduling and Blocking Restrictions 20
3.3 Thread Scheduling Hooks 21

4 Timers, Periodic Checks, and Conditions **23**

Chapter 1

Initialization and Completion

The program utilizing Converse begins executing at `main`, like any other C program. The initialization process is somewhat complicated by the fact that hardware vendors don't agree about which processors should execute `main`. On some machines, every processor executes `main`. On others, only one processor executes `main`. All processors which don't execute `main` are asleep when the program begins. The function ConverseInit is used to start the Converse system, and to wake up the sleeping processors.

typedef void (*CmiStartFn)(int argc, char **argv);
void ConverseInit(int argc, char *argv[], CmiStartFn fn, int usched, int initret)
This function starts up the Converse system. It can execute in one of the modes described below.

Normal Mode: `schedmode=0, initret=0`

When the user runs a program, some of the processors automatically invoke `main`, while others remain asleep. All processors which automatically invoked `main` must call ConverseInit. This initializes the entire Converse system. Converse then initiates, on *all* processors, the execution of the user-supplied start-function `fn(argc, argv)`. When this function returns, Converse automatically calls `CsdScheduler`, a function that polls for messages and executes their handlers (see chapter 2). Once `CsdScheduler` exits on all processors, the Converse system shuts down, and the user's program terminates. Note that in this case, ConverseInit never returns. The user is not allowed to poll for messages manually.

User-calls-scheduler Mode: `schedmode=1, initret=0`

If the user wants to poll for messages and other events manually, this mode is used to initialize Converse. In normal mode, it is assumed that the user-supplied start-function `fn(argc, argv)` is just for initialization, and that the remainder of the lifespan of the program is spent in the (automatically-invoked) function `CsdScheduler`, polling for messages. In user-calls-scheduler mode, however, it is assumed that the user-supplied start-function will perform the *entire computation*, including polling for messages. Thus, ConverseInit will not automatically call `CsdScheduler` for you. When the user-supplied start-function ends, Converse shuts down. This mode is not supported on the sim version. This mode can be combined with ConverseInit-returns mode below.

ConverseInit-returns Mode: `schedmode=1, initret=1`

This option is used when you want ConverseInit to return. All processors which automatically

invoked `main` must call ConverseInit. This initializes the entire Converse System. On all processors which *did not* automatically invoke `main`, Converse initiates the user-supplied initialization function `fn(argc, argv)`. Meanwhile, on those processors which *did* automatically invoke `main`, ConverseInit returns. Shutdown is initiated when the processors that *did* automatically invoke `main` call ConverseExit, and when the other processors return from `fn`. In this mode, all polling for messages must be done manually (probably using CsdScheduler explicitly). This option is not supported by the sim version.

void ConverseExit(void)
This function is only used in ConverseInit-returns mode, described above.

Chapter 2

Machine Interface and Scheduler

This chapter describes two of Converse's modules: the CMI, and the CSD. Together, they serve to transmit messages and schedule the delivery of messages. First, we describe the machine model assumed by Converse.

2.1 Machine Model

Converse treats the parallel machine as a collection of nodes, where each node is comprised of a number of processors that share memory In some cases, the number of processors per node may be exactly one (e.g. Distributed memory multicomputers such as IBM SP.) In addition, each of the processors may have multiple threads running on them which share code and data but have different stacks. Functions and macros are provided for handling shared memory across processors and querying node information. These are discussed in section 2.13

2.2 Defining Handler Numbers

When a message arrives at a processor, it triggers the execution of a *handler function*, not unlike a UNIX signal handler. The handler function receives, as an argument, a pointer to a copy of the message. The message itself specifies which handler function is to be called when the message arrives. Messages are contiguous sequences of bytes. The message has two parts: the header, and the data. The data may contain anything you like. The header contains a *handler number*, which specifies which handler function is to be executed when the message arrives. Before you can send a message, you have to define the handler numbers.

Converse maintains a table mapping handler numbers to function pointers. Each processor has its own copy of the mapping. There is a caution associated with this approach: it is the user's responsibility to ensure that all processors have identical mappings. This is easy to do, nonetheless, and the user must be aware that this is (usually) required.

The following functions are provided to define the handler numbers:

typedef void (*CmiHandler)(void *)

Functions that handle Converse messages must be of this type.

int CmiRegisterHandler(CmiHandler h)
This represents the standard technique for associating numbers with functions. To use this technique, the Converse user registers each of his functions, one by one, using CmiRegisterHandler. One must register exactly the same functions in exactly the same order on all processors. The system assigns monotonically increasing numbers to the functions, the same numbers on all processors. This insures global consistency. CmiRegisterHandler returns the number which was chosen for the function being registered.

int CmiRegisterHandlerGlobal(CmiHandler h)
This represents a second registration technique. The Converse user registers his functions on processor zero, using CmiRegisterHandlerGlobal. The Converse user is then responsible for broadcasting those handler numbers to other processors, and installing them using CmiNumberHandler below. The user should take care not to invoke those handlers until they are fully installed.

int CmiRegisterHandlerLocal(CmiHandler h)
This function is used when one wishes to register functions in a manner that is not consistent across processors. This function chooses a locally-meaningful number for the function, and records it locally. No attempt is made to ensure consistency across processors.

void CmiNumberHandler(int n, CmiHandler h)
Forces the system to associate the specified handler number n with the specified handler function h. If the function number n was previously mapped to some other function, that old mapping is forgotten. The mapping that this function creates is local to the current processor. CmiNumberHandler can be useful in combination with CmiRegisterGlobalHandler. It can also be used to implement user-defined numbering schemes: such schemes should keep in mind that the size of the table that holds the mapping is proportional to the largest handler number — do not use big numbers!

Note: of the three registration methods, the CmiRegisterHandler method is by far the simplest, and is strongly encouraged. The others are primarily to ease the porting of systems that already use similar registration techniques. One may use all three registration methods in a program. The system guarantees that no numbering conflicts will occur as a result of this combination.

2.3 Writing Handler Functions

A message handler function is just a C function that accepts a void pointer (to a message buffer) as an argument, and returns nothing. The handler may use the message buffer until it returns, at which time Converse will automatically free the message buffer. This behavior can be overrided using CmiGrabBuffer:

void CmiGrabBuffer(void **pbuf)
A handler function receives a pointer to a message buffer as an argument. Normally, it is supposed to copy the data out of the message buffer before it returns, at which time Converse automatically frees the message buffer. However, a handler function may use CmiGrabBuffer to claim ownership of the message buffer (and therefore prevent Converse from freeing it). Assuming, for example, that the handler function called its argument `msg`, it would call `CmiGrabBuffer(&msg)`. Afterward,

`msg` contains a pointer to the message, or a copy of it. From that point forward, it is the user's responsibility to free the message using `CmiFree`.

2.4 Building Messages

To send a message, one first creates a buffer to hold the message. The buffer must be large enough to hold the header and the data. The buffer can be in any kind of memory: it could be a local variable, it could be a global, it could be allocated with `malloc`, and finally, it could be allocated with `CmiAlloc`. The Converse user fills the buffer with the message data. One puts a handler number in the message, thereby specifying which handler function the message should trigger when it arrives. Finally, one uses a message-transmission function to send the message.

The following functions are provided to help build message buffers:

void *CmiAlloc(int size)
Allocates memory of size `size` in heap and returns pointer to the usable space. There are some message-sending functions that accept only message buffers that were allocated with CmiAlloc. Thus, this is the preferred way to allocate message buffers.

void CmiFree(void *ptr)
This function frees the memory pointed to by `ptr`. `ptr` should be a pointer that was previously returned by `CmiAlloc`.

#define CmiMsgHeaderSizeBytes
This constant contains the size of the message header. When one allocates a message buffer, one must set aside enough space for the header and the data. This macro helps you to do so.

void CmiSetHandler(int *MessageBuffer, int HandlerId)
This macro sets the handler number of a message to `HandlerId`.

int CmiGetHandler(int *MessageBuffer)
This call returns the handler of a message in the form of a handler number.

CmiHandler CmiGetHandlerFunction(int *MessageBuffer)
This call returns the handler of a message in the form of a function pointer.

2.5 Sending Messages

The following functions allow you to send messages. Our model is that the data starts out in the message buffer, and from there gets transferred "into the network". The data stays "in the network" for a while, and eventually appears on the target processor. Using that model, each of these send-functions is a device that transfers data into the network. None of these functions wait for the data to be delivered.

On some machines, the network accepts data rather slowly. We don't want the process to sit idle, waiting for the network to accept the data. So, we provide several variations on each send function:

- **sync**: a version that is as simple as possible, pushing the data into the network and not returning until the data is "in the network". As soon as a sync function returns, you can reuse the message buffer.
- **async**: a version that returns almost instantaneously, and then continues working in the background. The background job transfers the data from the message buffer into the network. Since the background job is still using the message buffer when the function returns, you can't reuse the message buffer immediately. The background job sets a flag when it is done and you can then reuse the message buffer.
- **sync and free**: a version that returns almost instantaneously, and then continues working in the background. The background job transfers the data from the message buffer into the network. When the background job finishes, it `CmiFree`s the message buffer. In this situation, you can't reuse the message buffer at all. Of course, to use a function of this type, you must allocate the message buffer using `CmiAlloc`.

void CmiSyncSend(unsigned int destPE, unsigned int size, void *msg)
Sends `msg` of size `size` bytes to processor `destPE`. When it returns, you may reuse the message buffer.

void CmiSyncSendAndFree(unsigned int destPE, unsigned int size, void *msg)
Sends `msg` of size `size` bytes to processor `destPE`. When it returns, the message buffer has been freed using `CmiFree`

CmiCommHandle CmiAsyncSend(unsigned int destPE, unsigned int size, void *msg)
Sends `msg` of size `size` bytes to processor `destPE`. It returns a communication handle which can be tested using CmiAsyncMsgSent: when this returns true, you may reuse the message buffer.

void CmiSyncVectorSend(int destPE, int len, int sizes[], char *msgComps[])
Concatenates several pieces of data and sends them to processor `destPE`. The data consists of `len` pieces residing in different areas of memory, which are logically concatenated. The `msgComps` array contains pointers to the pieces; the size of `msgComps[i]` is taken from `sizes[i]`. When it returns, `sizes`, `msgComps` and the message components specified in `msgComps` can be immediately reused.

void CmiSyncVectorSendAndFree(int destPE, int len, int sizes[], char *msgComps[])
Concatenates several pieces of data and sends them to processor `destPE`. The data consists of `len` pieces residing in different areas of memory, which are logically concatenated. The `msgComps` array contains pointers to the pieces; the size of `msgComps[i]` is taken from `sizes[i]`. The message components specified in `msgComps` are `CmiFree`d by this function therefore, they should be dynamically allocated using `CmiAlloc()`. However, the `sizes` and `msgComps` array themselves are not freed.

CmiCommHandle CmiAsyncVectorSend(int destPE, int len, int sizes[], char *msgComps[])
Concatenates several pieces of data and sends them to processor `destPE`. The data consists of `len` pieces residing in different areas of memory, which are logically concatenated. The `msgComps` array contains pointers to the pieces; the size of `msgComps[i]` is taken from `sizes[i]`. The individual pieces of data as well as the arrays `sizes` and `msgComps` should not be overwritten or freed before the communication is complete. This function returns a communication handle which can

be tested using CmiAsyncMsgSent: when this returns true, the input parameters can be reused.

int CmiAsyncMsgSent(CmiCommHandle handle)
Returns true if the communication specified by the given CmiCommHandle has proceeded to the point where the message buffer can be reused.

void CmiReleaseCommHandle(CmiCommHandle handle)
Releases the communication handle `handle` and associated resources. It does not free the message buffer.

void CmiMultipleSend(unsigned int destPE, int len, int sizes[], char *msgComps[])
This function allows the user to send many multiple messages that may be destined for the SAME PE in one go. This is more efficient than sending each message to the destination node separately. This function assumes that the handlers that are to receive this message have already been set. If this is not done, the behavior of the function is undefined.

In the function, The **destPE** parameter identifies the destination processor.

The **len** argument identifies the *number* of messages that are to be sent in one go.

The **sizes**[] array is an array of sizes of each of these messages.

The **msgComps**[] array is the array of the messages.

The indexing in each array is from 0 to len - 1.

Note:

Before calling this function, the program needs to initialise the system to be able to provide this service. This is done by calling the function **void CmiInitMultipleSendRoutine(void)** . Unless this function is called, the system will not be able to provide the service to the user.

2.6 Broadcasting Messages

void CmiSyncBroadcast(unsigned int size, void *msg)
Sends `msg` of length `size` bytes to all processors excluding the processor on which the caller resides.

void CmiSyncBroadcastAndFree(unsigned int size, void *msg)
Sends `msg` of length `size` bytes to all processors excluding the processor on which the caller resides. Uses `CmiFree` to deallocate the message buffer for `msg` when the broadcast completes. Therefore `msg` must point to a buffer allocated with `CmiAlloc`.

void CmiSyncBroadcastAll(unsigned int size, void *msg)
Sends `msg` of length `size` bytes to all processors including the processor on which the caller resides. This function does not free the message buffer for `msg`.

void CmiSyncBroadcastAllAndFree(unsigned int size, void *msg)
Sends `msg` of length `size` bytes to all processors including the processor on which the caller resides. This function frees the message buffer for `msg` before returning, so `msg` must point to a dynamically allocated buffer.

CmiCommHandle CmiAsyncBroadcast(unsigned int size, void *msg)
Initiates asynchronous broadcast of message `msg` of length `size` bytes to all processors excluding the processor on which the caller resides. It returns a communication handle which could be used to check the status of this send using `CmiAsyncMsgSent()`. `msg` should not be overwritten or freed before the communication is complete.

CmiCommHandle CmiAsyncBroadcastAll(unsigned int size, void *msg)
Initiates asynchronous broadcast of message `msg` of length `size` bytes to all processors including the processor on which the caller resides. It returns a communication handle which could be used to check the status of this send using `CmiAsyncMsgSent()`. `msg` should not be overwritten or freed before the communication is complete.

2.7 Multicasting Messages

typedef ... CmiGroup;
A CmiGroup represents a set of processors. It is an opaque type. Group IDs are useful for the multicast functions below.

CmiGroup CmiEstablishGroup(int npes, int *pes);
Converts an array of processor numbers into a group ID. Group IDs are useful for the multicast functions below. Caution: this call uses up some resources. In particular, establishing a group uses some network bandwidth (one broadcast's worth) and a small amount of memory on all processors.

void CmiSyncMulticast(CmiGroup grp, unsigned int size, void *msg)
Sends `msg` of length `size` bytes to all members of the specified group. Group IDs are created using `CmiEstablishGroup`.

void CmiSyncMulticastAndFree(CmiGroup grp, unsigned int size, void *msg)
Sends `msg` of length `size` bytes to all members of the specified group. Uses `CmiFree` to deallocate the message buffer for `msg` when the broadcast completes. Therefore `msg` must point to a buffer allocated with `CmiAlloc`. Group IDs are created using `CmiEstablishGroup`.

CmiCommHandle CmiAsyncMulticast(CmiGroup grp, unsigned int size, void *msg)
Not yet implemented. Initiates asynchronous broadcast of message `msg` of length `size` bytes to all members of the specified group. It returns a communication handle which could be used to check the status of this send using `CmiAsyncMsgSent()`. `msg` should not be overwritten or freed before the communication is complete. Group IDs are created using `CmiEstablishGroup`.

void CmiSyncListSend(int npes, int *pes, unsigned int size, void *msg)
Not yet implemented. Sends `msg` of length `size` bytes to all processors in the list. Group IDs are created using `CmiEstablishGroup`.

void CmiSyncMulticastAndFree(int npes, int *pes, unsigned int size, void *msg)
Not yet implemented. Sends `msg` of length `size` bytes to all processors in the list. Uses `CmiFree` to deallocate the message buffer for `msg` when the broadcast completes. Therefore `msg` must point to a buffer allocated with `CmiAlloc`. Group IDs are created using `CmiEstablishGroup`.

CmiCommHandle CmiAsyncMulticast(int npes, int *pes, unsigned int size, void

***msg)**
Not yet implemented. Initiates asynchronous broadcast of message `msg` of length `size` bytes to all processors in the list. It returns a communication handle which could be used to check the status of this send using `CmiAsyncMsgSent()`. `msg` should not be overwritten or freed before the communication is complete. Group IDs are created using `CmiEstablishGroup`.

2.8 Scheduling Messages

The scheduler queue is a powerful priority queue. The following functions can be used to place messages into the scheduler queue. These messages are treated very much like newly-arrived messages: when they reach the front of the queue, they trigger handler functions, just like messages transmitted with Cmi functions. Note that unlike the Cmi send functions, these cannot move messages across processors.

Every message inserted into the queue has a priority associated with it. Converse priorities are arbitrary-precision numbers between 0 and 1. Priorities closer to 0 get processed first, priorities closer to 1 get processed last. Arbitrary-precision priorities are very useful in AI search-tree applications. Suppose we have a heuristic suggesting that tree node N1 should be searched before tree node N2. We therefore designate that node N1 and its descendants will use high priorities, and that node N2 and its descendants will use lower priorities. We have effectively split the range of possible priorities in two. If several such heuristics fire in sequence, we can easily split the priority range in two enough times that no significant bits remain, and the search begins to fail for lack of meaningful priorities to assign. The solution is to use arbitrary-precision priorities, aka bitvector priorities.

These arbitrary-precision numbers are represented as bit-strings: for example, the bit-string "0011000101" represents the binary number (.0011000101). The format of the bit-string is as follows: the bit-string is represented as an array of unsigned integers. The most significant bit of the first integer contains the first bit of the bitvector. The remaining bits of the first integer contain the next 31 bits of the bitvector. Subsequent integers contain 32 bits each. If the size of the bitvector is not a multiple of 32, then the last integer contains 0 bits for padding in the least-significant bits of the integer.

Some people only want regular integers as priorities. For simplicity's sake, we provide an easy way to convert integer priorities to Converse's built-in representation.

In addition to priorities, you may choose to enqueue a message "LIFO" or "FIFO". Enqueueing a message "FIFO" simply pushes it behind all the other messages of the same priority. Enqueueing a message "LIFO" pushes it in front of other messages of the same priority.

Messages sent using the CMI functions take precedence over everything in the scheduler queue, regardless of priority.

void CsdEnqueueGeneral(void *Message, int strategy, int priobits, int *prioptr)
This call enqueues a message to the scheduler's queue, to be sorted according to its priority and the queueing `strategy`. The meaning of the `priobits` and `prioptr` fields depend on the value of `strategy`, which can be any of the following:

- `CQS_QUEUEING_BFIFO`: the priobits and prioptr point to a bit-string representing an arbitrary-precision priority. The message is pushed behind all other message of this priority.
- `CQS_QUEUEING_BLIFO`: the priobits and prioptr point to a bit-string representing an arbitrary-precision priority. The message is pushed in front all other message of this priority.
- `CQS_QUEUEING_IFIFO`: the prioptr is a pointer to a signed integer. The integer is converted to a bit-string priority, normalizing so that the integer zero is converted to the bit-string "1000..." (the "middle" priority). To be more specific, the conversion is performed by adding 0x80000000 to the integer, and then treating the resulting 32-bit quantity as a 32-bit bitvector priority. The message is pushed behind all other messages of this priority.
- `CQS_QUEUEING_ILIFO`: the prioptr is a pointer to a signed integer. The integer is converted to a bit-string priority, normalizing so that the integer zero is converted to the bit-string "1000..." (the "middle" priority). To be more specific, the conversion is performed by adding 0x80000000 to the integer, and then treating the resulting 32-bit quantity as a 32-bit bitvector priority. The message is pushed in front of all other messages of this priority.
- `CQS_QUEUEING_FIFO`: the prioptr and priobits are ignored. The message is enqueued with the middle priority "1000...", and is pushed behind all other messages with this priority.
- `CQS_QUEUEING_LIFO`: the prioptr and priobits are ignored. The message is enqueued with the middle priority "1000...", and is pushed in front of all other messages with this priority.

Caution: the priority itself is *not copied* by the scheduler. Therefore, if you pass a pointer to a priority into the scheduler, you must not overwrite or free that priority until after the message has emerged from the scheduler's queue. It is normal to actually store the priority *in the message itself*, though it is up to the user to actually arrange storage for the priority.

void CsdEnqueue(void *Message)
This macro is a shorthand for `CsdEnqueueGeneral(Message, CQS_QUEUEING_FIFO,0, NULL)` provided here for backward compatibility.

void CsdEnqueueFifo(void *Message)
This macro is a shorthand for `CsdEnqueueGeneral(Message, CQS_QUEUEING_FIFO,0, NULL)` provided here for backward compatibility.

void CsdEnqueueLifo(void *Message)
This macro is a shorthand for `CsdEnqueueGeneral(Message, CQS_QUEUEING_LIFO,0, NULL)` provided here for backward compatibility.

int CsdEmpty()
This function returns non-zero integer when the scheduler's queue is empty, zero otherwise.

2.9 Polling for Messages

As we stated earlier, Converse messages trigger handler functions when they arrive. In fact, for this to work, the processor must occasionally poll for messages. When the user starts Converse, he can put it into one of several modes. In the normal mode, the message polling happens automatically.

However *user-calls-scheduler* mode is designed to let the user poll manually. To do this, the user must use one of two polling functions: CmiDeliverMsgs, or CsdScheduler. CsdScheduler is more general, it will notice any Converse event. CmiDeliverMsgs is a lower-level function that ignores all events except for recently-arrived messages. (In particular, it ignores messages in the scheduler queue). You can save a tiny amount of overhead by using the lower-level function. We recommend the use of CsdScheduler for all applications except those that are using only the lowest level of Converse, the Cmi. A third polling function, CmiDeliverSpecificMsg, is used when you know the exact event you want to wait for: it does not allow any other event to occur.

void CsdScheduler(int NumberOfMessages)
This call invokes the Converse scheduler, which repeatedly delivers messages to their handlers (i.e. invokes the handler for each message it selects). In each iteration, the scheduler first looks for any message that has arrived from another processor, and delivers it. If there isn't any, it selects a message from the locally enqueued messages, and delivers it. The `NumberOfMessages` parameter specifies how many messages should be processed (i.e. delivered to their handlers). If set to -1, the scheduler continues delivering messages until CsdExitScheduler() is called from a message handler. if `NumberOfMessages` is 0, the scheduler continues delivering messages until it exhausts its supply of messages (i.e. becomes idle) or some handler calls CsdExitScheduler().

int CmiDeliverMsgs(int MaxMsgs)
Retrieves messages from the network message queue and invokes corresponding handler functions for arrived messages. This function returns after either the network message queue becomes empty or after `MaxMsgs` messages have been retrieved and their handlers called. It returns the difference between total messages delivered and `MaxMsgs`. The handler is given a pointer to the message as its parameter.

void CmiDeliverSpecificMsg(int HandlerId)
Retrieves messages from the network queue and delivers the first message with its handler field equal to `HandlerId`. This functions leaves alone all other messages. It returns after the invoked handler function returns.

void CsdExitScheduler(void)
This call causes CsdScheduler to stop processing messages when control has returned back to it. The scheduler then returns to its calling routine.

2.10 Scheduler Callbacks

Scheduler callbacks are provided for programmers who want to utilize in some way the event of scheduler becoming idle, or the scheduler becoming busy again. Typically, these callbacks are useful for determining idle time in complex multilingual programs. The callback function should not contain calls to scheduler related functions. Also, the callback functions should not block indefinitely. Doing so will cause the conditional callbacks to fail. Following functions are provided for controlling the scheduler callbacks.

void CsdStartNotifyIdle(void)
This call causes CsdScheduler to invoke callback functions for idle and busy notifications. Initially, the scheduler does not call any callback functions.

void CsdStopNotifyIdle(void)
This call causes CsdScheduler to stop invoking callback functions for idle and busy norifications.

void CsdSetNotifyIdle(void (* fIdle)(void), void (* fBusy)(void))
This call registers the functions to be called when CsdScheduler becomes idle (`fIdle`) or busy (`fBusy`). These functions can be overridden with another call to CsdSetNotifyIdle.

2.11 The Timer

double CmiTimer(void)
Returns current value of the timer in seconds. This is typically the time spent since the `ConverseInit()` call. The precision of this timer is the best available on the particular machine, and usually has at least microsecond accuracy.

2.12 Processor Ids

int CmiNumPe(void)
Returns the total number of processors on which the parallel program is being run.

int CmiMyPe(void)
Returns the logical processor identifier of processor on which the caller resides. A processor Id is between `0` and `CmiNumPe()-1`.

Also see the calls in Section 2.13.2.

2.13 Global Variables and Utility functions

Different vendors are not consistent about how they treat global and static variables. Most vendors write C compilers in which global variables are shared among all the processors in the node. A few vendors write C compilers where each processor has its own copy of the global variables. In theory, it would also be possible to design the compiler so that each thread has its own copy of the global variables.

The lack of consistency across vendors, makes it very hard to write a portable program. The fact that most vendors make the globals shared is inconvenient as well, usually, you don't want your globals to be shared. For these reasons, we added "pseudoglobals" to Converse. These act much like C global and static variables, except that you have explicit control over the degree of sharing.

2.13.1 Converse PseudoGlobals

Three classes of pseudoglobal variables are supported: node-private, process-private, and thread-private variables.

Node-private global variables are specific to a node. They are shared among all the processes within the node.

Process-private global variables are specific to a process. They are shared among all the threads within the process.

Thread-private global variables are specific to a thread. They are truely private.

There are five macros for each class. These macros are for declaration, static declaration, extern declaration, initialization, and access. The declaration, static and extern specifications have the same meaning as in C. In order to support portability, however, the global variables must be installed properly, by using the initialization macros. For example, if the underlying machine is a simulator for the machine model supported by Converse, then the thread-private variables must be turned into arrays of variables. Initialize and Access macros hide these details from the user. It is possible to use global variables without these macros, as supported by the underlying machine, but at the expense of portability.

Macros for node-private variables:

```
CsvDeclare(type,variable)
CsvStaticDeclare(type,variable)
CsvExtern(type,variable)
CsvInitialize(type,variable)
CsvAccess(variable)
```

Macros for process-private variables:

```
CpvDeclare(type,variable)
CpvStaticDeclare(type,variable)
CpvExtern(type,variable)
CpvInitialize(type,variable)
CpvAccess(variable)
```

Macros for thread-private variables:

```
CtvDeclare(type,variable)
CtvStaticDeclare(type,variable)
CtvExtern(type,variable)
CtvInitialize(type,variable)
CtvAccess(variable)
```

A sample code to illustrate the usage of the macros is provided in Figure 2.1. There are a few rules that the user must pay attention to: The `type` and `variable` fields of the macros must be a single word. Therefore, structures or pointer types can be used by defining new types with the `typedef`. In the sample code, for example, a `struct point` type is redefined with a `typedef` as `Point` in order to use it in the macros. Similarly, the access macros contain only the name of the global variable. Any indexing or member access must be outside of the macro as shown in the

```
File Module1.c

    typedef struct point
    {
         float x,y;
    } Point;


    CpvDeclare(int, a);
    CpvDeclare(Point, p);

    void ModuleInit()
    {
         CpvInitialize(int, a)
         CpvInitialize(Point, p);

         CpvAccess(a) = 0;
    }

    int func1()
    {
         CpvAccess(p).x = 0;
         CpvAccess(p).y = CpvAccess(p).x + 1;
    }
```

Figure 2.1: An example code for global variable usage

sample code (function `func1`). Finally, all the global variables must be installed before they are used. One way to do this systematically is to provide a module-init function for each file (in the sample code - `ModuleInit()`. The module-init functions of each file, then, can be called at the beginning of execution to complete the installations of all global variables.

2.13.2 Utility Functions

To further simplify programming with global variables on shared memory machines, Converse provides the following functions and/or macros. (Note: *These functions are defined on machines other than shared-memory machines also, and have the effect of only one processor per node and only one thread per processor.*)

int CmiMyNode()
Returns the node number to which the calling processor belongs.

int CmiNumNodes()
Returns number of nodes in the system. Note that this is not the same as `CmiNumPes()`.

int CmiMyRank()
Returns the rank of the calling processor within a shared memory node.

int CmiNodeFirst(int node)
Returns the processor number of the lowest ranked processor on node `node`

int CmiNodeSize(int node)
Returns the number of processors that belong to the node `node`.

int CmiNodeOf(int pe)
Returns the node number to which processor `pe` belongs. Indeed, `CmiMyNode()` is a utility macro that is aliased to `CmiNodeOf(CmiMyPe())`.

int CmiRankOf(int pe)
Returns the rank of processor `pe` in the node to which it belongs.

2.13.3 Node-level Locks and other Synchronization Mechanisms

void CmiNodeBarrier()
Provide barrier synchronization at the node level, i.e. all the processors belonging to the node participate in this barrier.

typedef McDependentType CmiNodeLock
This is the type for all the node-level locks in Converse.

CmiNodeLock CmiCreateLock(void)
Creates, initializes and returns a new lock. Initially the lock is unlocked.

void CmiLock(CmiNodeLock lock)
Locks `lock`. If the `lock` has been locked by other processor, waits for `lock` to be unlocked.

void CmiUnlock(CmiNodeLock lock)
Unlocks `lock`. Processors waiting for the `lock` can then compete for acquiring `lock`.

int CmiTryLock(CmiNodeLock lock)
Tries to lock `lock`. If it succeeds in locking, it returns 0. If any other processor has already acquired the lock, it returns 1.

voi CmiDestroyLock(CmiNodeLock lock)
Frees any memory associated with `lock`. It is an error to perform any operations with `lock` after a call to this function.

void *CmiSvAlloc(int size)
Allocates a block of memory of `size` bytes from the heap in node's local memory and returns pointer to the start of this block. (Note: *On machines other than shared-memory machines, this is equivalent to* `CmiAlloc`.)

2.14 Input/Output

void CmiPrintf(char *format, arg1, arg2, ...)
This function does an atomic `printf()` on `stdout`. On machine with host, this is implemented on top of the messaging layer using asynchronous sends.

int CmiScanf(char *format, void *arg1, void *arg2, ...)
This function performs an atomic `scanf` from `stdin`. The processor, on which the caller resides, blocks for input. On machines with host, this is implemented on top of the messaging layer using asynchronous send and blocking receive.

void CmiError(char *format, arg1, arg2, ...)
This function does an atomic `printf()` on `stderr`. On machines with host, this is implemented on top of the messaging layer using asynchronous sends.

2.15 Spanning Tree Calls

Sometimes, it is convenient to view the nodes of the machine as a tree. For this purpose, Converse defines a tree over the nodes. We provide functions to obtain the parent and children of each node. On those machines where the communication topology is relevant, we arrange the tree to optimize communication performance.

int CmiSpanTreeRoot()
Returns the processor number of the root of the spanning tree.

int CmiSpanTreeParent(int procNum)
This function returns the processor number of the parent of `procNum` in the spanning tree.

int CmiNumSpanTreeChildren(int procNum)
Returns the number of children of `procNum` in the spanning tree.

void CmiSpanTreeChildren(int procNum, int *children)
This function fills the array `children` with processor numbers of children of `procNum` in the spanning tree.

Chapter 3

Threads

The calls in this chapter can be used to put together runtime systems for languages that support threads. This threads package, like most thread packages, provides basic functionality for creating threads, destroying threads, yielding, suspending, and awakening a suspended thread. In addition, it provides facilities whereby you can write your own thread schedulers.

3.1 Basic Thread Calls

typedef struct CthThreadStruct *CthThread;
This is an opaque type defined in `converse.h`. It represents a first-class thread object. No information is publicized about the contents of a CthThreadStruct.

typedef void (CthVoidFn)();
This is a type defined in `converse.h`. It represents a function that returns nothing.

typedef CthThread (CthThFn)();
This is a type defined in `converse.h`. It represents a function that returns a CthThread.

CthThread CthSelf()
Returns the currently-executing thread. Note: even the initial flow of control that inherently existed when the program began executing `main` counts as a thread. You may retrieve that thread object using `CthSelf` and use it like any other.

CthThread CthCreate(CthVoidFn fn, void *arg, int size)
Creates a new thread object. The thread is not given control yet. To make the thread execute, you must push it into the scheduler queue, using CthAwaken below. When (and if) the thread eventually receives control, it will begin executing the specified function `fn` with the specified argument. The `size` parameter specifies the stack size in bytes, 0 means use the default size. Caution: almost all threads are created with CthCreate, but not all. In particular, the one initial thread of control that came into existence when your program was first `exec`'d was not created with `CthCreate`, but it can be retrieved (say, by calling `CthSelf` in `main`), and it can be used like any other `CthThread`.

void CthFree(CthThread t)
Frees thread `t`. You may ONLY free the currently-executing thread (yes, this sounds strange, it's historical). Naturally, the free will actually be postponed until the thread suspends. To terminate itself, a thread calls `CthFree(CthSelf())`, then gives up control to another thread.

void CthSuspend()
Causes the current thread to stop executing. The suspended thread will not start executing again until somebody pushes it into the scheduler queue again, using CthAwaken below. Control transfers to the next task in the scheduler queue.

void CthAwaken(CthThread t)
Pushes a thread into the scheduler queue. Caution: a thread must only be in the queue once. Pushing it in twice is a crashable error.

void CthYield()
This function is part of the scheduler-interface. It simply executes { `CthAwaken(CthSelf());` `CthSuspend();` } . This combination gives up control temporarily, but ensures that control will eventually return.

CthThread CthGetNext(CthThread t)
Each thread contains space for the user to store a "next" field (the functions listed here pay no attention to the contents of this field). This field is typically used by the implementors of mutexes, condition variables, and other synchronization abstractions to link threads together into queues. This function returns the contents of the next field.

void CthSetNext(CthThread t, CthThread next)
Each thread contains space for the user to store a "next" field (the functions listed here pay no attention to the contents of this field). This field is typically used by the implementors of mutexes, condition variables, and other synchronization abstractions to link threads together into queues. This function sets the contents of the next field.

3.2 Thread Scheduling and Blocking Restrictions

Converse threads use a scheduler queue, like any other threads package. We chose to use the same queue as the one used for Converse messages (see section 2.8). Because of this, thread context-switching will not work unless there is a thread polling for messages. A rule of thumb, with Converse, it is best to have a thread polling for messages at all times. In Converse's normal mode (see section 1), this happens automatically. However, in user-calls-scheduler mode, you must be aware of it.

There is a second caution associated with this design. There is a thread polling for messages (even in normal mode, it's just hidden in normal mode). The continuation of your computation depends on that thread — you must not block it. In particular, you must not call blocking operations in these places:

- In the code of a Converse handler (see sections 2.2 and 2.3).
- In the code of the Converse start-function (see section 1).

These restrictions are usually easy to avoid. For example, if you wanted to use a blocking operation inside a Converse handler, you would restructure the code so that the handler just creates a new thread and returns. The newly-created thread would then do the work that the handler originally did.

3.3 Thread Scheduling Hooks

Normally, when you CthAwaken a thread, it goes into the primary ready-queue: namely, the main Converse queue described in section 2.8. However, it is possible to hook a thread to make it go into a different ready-queue. That queue doesn't have to be priority-queue: it could be FIFO, or LIFO, or in fact it could handle its threads in any complicated order you desire. This is a powerful way to implement your own scheduling policies for threads.

To achieve this, you must first implement a new kind of ready-queue. You must implement a function that inserts threads into this queue. The function must have this prototype:

void awakenfn(CthThread t);

When a thread suspends, it must choose a new thread to transfer control to. You must implement a function that makes the decision: which thread should the current thread transfer to. This function must have this prototype:

CthThread choosefn();

Typically, the choosefn would choose a thread from your ready-queue. Alternately, it might choose to always transfer control to a central scheduling thread.

You then configure individual threads to actually use this new ready-queue. This is done using CthSetStrategy:

void CthSetStrategy(CthThread t, CthVoidFn awakenfn, CthThFn choosefn)
Causes the thread to use the specified `awakefn` whenever you CthAwaken it, and the specified `choosefn` whenever you CthSuspend it.

CthSetStrategy alters the behavior of CthSuspend and CthAwaken. Normally, when a thread is awakened with CthAwaken, it gets inserted into the main ready-queue. Setting the thread's `awakenfn` will cause the thread to be inserted into your ready-queue instead. Similarly, when a thread suspends using CthSuspend, it normally transfers control to some thread in the main ready-queue. Setting the thread's `choosefn` will cause it to transfer control to a thread chosen by your `choosefn` instead.

You may reset a thread to its normal behavior using CthSetStrategyDefault:

void CthSetStrategyDefault(CthThread t)
Restores the value of `awakefn` and `choosefn` to their default values. This implies that the next time you CthAwaken the specified thread, it will be inserted into the normal ready-queue.

Keep in mind that this only resolves the issue of how threads get into your ready-queue, and how those threads suspend. To actually make everything "work out" requires additional planning: you have to make sure that control gets transferred to everywhere it needs to go.

Scheduling threads may need to use this function as well:

void CthResume(CthThread t)
Immediately transfers control to thread `t`. This routine is primarily intended for people who are implementing schedulers, not for end-users. End-users should probably call `CthSuspend` or `CthAwaken` (see below). Likewise, programmers implementing locks, barriers, and other synchronization devices should also probably rely on `CthSuspend` and `CthAwaken`.

A final caution about the `choosefn`: it may only return a thread that wants the CPU, eg, a thread that has been awakened using the `awakefn`. If no such thread exists, if the `choosefn` cannot return an awakened thread, then it must not return at all: instead, it must wait until, by means of some pending IO event, a thread becomes awakened (pending events could be asynchonous disk reads, networked message receptions, signal handlers, etc). For this reason, many schedulers perform the task of polling the IO devices as a side effect. If handling the IO event causes a thread to be awakened, then the choosefn may return that thread. If no pending events exist, then all threads will remain permanently blocked, the program is therefore done, and the `choosefn` should call `exit`.

There is one minor exception to the rule stated above ("the scheduler may not resume a thread unless it has been declared that the thread wants the CPU using the `awakefn`"). If a thread `t` is part of the scheduling module, it is permitted for the scheduling module to resume `t` whenever it so desires: presumably, the scheduling module knows when its threads want the CPU.

Chapter 4

Timers, Periodic Checks, and Conditions

This module provides functions that allow users to insert hooks, i.e. user-supplied functions, that are called by the system at specified times, or as specific conditions arise.

CcdCallOnCondition

```
void CcdCallOnCondition(condnum,fnp,arg)
    int condnum;
    CcdVoidFn fnp;
    void *arg;
```

This call instructs the system to call the function indicated by the function pointer `fnp`, with the specified argument `arg`, when the condition indicated by `condnum` is raised next. Multiple functions may be registered for the same condition number. A total of 511 conditions, numbered 1 through 511, are supported. Currently, users must make sure that various condition numbers used in a program are disjoint. (In the future, we may provide a call to allocate the next available condition number.)

The system supports a predefined condition, with condition number 1 (CcdPROCESSORIDLE, BUT I AM AM NOT SURE THAT IS EXPORTED TO THE USER PROGRAM), which is raised by the system when there are no entities (ready threads, messages for objects, posted handlers, etc.) in the scheduler's queue.

CcdRaiseCondition

```
void CcdRaiseCondition(condNum)
     int condNum;
```

When this function is called, it invokes all the functions whose pointers were registered for the `condNum` via a *prior* call to `CcdCallOnCondition(..)`. All calls to `CcdCallOnCondition(..)` *during* this function's execution takes effect only after it returns. Once a user-registered function

is called, it loses its registration. So, if users want the function to be called the next time the same condition arises, they must register the function again.

CcdPeriodicallyCall

```
void CcdPeriodicallyCall(fnp, arg)
    CcdVoidFn fnp;
    void *arg;
```

A function registered through this call is called periodically by the system's scheduler. Typically, it would be called every time the scheduler gets control, such as while switching context to a new thread or before scheduling a message for an object, or before calling a posted handler. These functions don't have to be re-registered after every call, unlike the functions for the "conditions".

CcdCallFnAfter

```
void CcdCallFnAfter(fnp, arg, deltaT)
    CcdVoidFn fnp;
    void *arg;
    unsigned int deltaT;
```

This call registers a function via a pointer to it, `fnp`, that will be called at least `deltaT` milliseconds later. The registered function `fnp` is actually called the first time the scheduler gets control after `deltaT` milliseconds have elapsed.