Parallel Programming Laboratory
University of Illinois at Urbana-Champaign

Converse Programming Manual

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. The most recent version of Converse has had inputs from Milind Bhandarkar, Laxmikant Kale, Robert Brunner, Terry Wilmarth, Parthasarathy Ramachandran, Krishnan Varadarajan, and Jeffrey Wright.

Version 5.4 (Release 1)

University of Illinois
CHARM++/CONVERSE Parallel Programming System Software
Non-Exclusive, Non-Commercial Use License

Contact Information:
The best contact path for licensing issues is by e-mail to `kale@cs.uiuc.edu` or send correspondence to:

Prof. L. V. Kale
Dept. of Computer Science
University of Illinois
201 N. Goodwin Ave
Urbana, Illinois 61801 USA
FAX: (217) 333-3501

Contents

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.

void CmiAbort(char *msg)

This function can be used portably to abnormally terminate a CONVERSE program. Before termination, it prints a message supplied as `msg`.

void CmiAssert(int expr)

This macro terminates the Converse program after printing an informative message if `expr` evaluates to 0. It can be used in place of `assert`. In order to turn off `CmiAssert`, one should define $CMK\_OPTIMIZE$ as 1.

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 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 for any purpose, but is responsible for eventually deleting 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. The returned pointer point to the message header, the user data will follow it. See CmiMsgHeaderSizeBytes for this.

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. For example, if one want to allocate an array of 100 int, he should call the function this way: `‘‘char *myMsg = CmiAlloc(100*sizeof(int) + CmiMsgHeaderSizeBytes)’’`

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.
- **send 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 CmiFrees the message buffer. In this situation, you can’t reuse the message buffer at all. To use a function of this type, you must allocate the message buffer using CmiAlloc.
- **node**: a version that send a message to a node instead of a specific processor. This means that when the message is received, any “free” processor within than node can handle it. $\boxed{\beta}$

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 CmiSyncNodeSend(unsigned int destNode, unsigned int size, void *msg) $\boxed{\beta}$

Sends *msg* of size *size* bytes to node *destNode*. 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.

void CmiSyncNodeSendAndFree(unsigned int destNode, unsigned int size, void *msg) $\boxed{\beta}$

Sends *msg* of size *size* bytes to node *destNode*. 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. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent.

CmiCommHandle CmiAsyncNodeSend(unsigned int destNode, unsigned int size, void *msg) $\boxed{\beta}$

Sends *msg* of size *size* bytes to node *destNode*. It returns a communication handle which can be tested using CmiAsyncMsgSent: when this returns true, you may reuse the message buffer. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent.

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 CmiFreed 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. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent.

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[]) $\boxed{\beta}$

This function allows the user to send 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 CmiInitMultipleSendRoutine. 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 CmiSyncNodeBroadcast(unsigned int size, void *msg)

Sends *msg* of length *size* bytes to all nodes excluding the node 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 CmiSyncNodeBroadcastAndFree(unsigned int size, void *msg)

Sends *msg* of length *size* bytes to all nodes excluding the node 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 CmiSyncNodeBroadcastAll(unsigned int size, void *msg)

Sends *msg* of length *size* bytes to all nodes including the node 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.

void CmiSyncNodeBroadcastAllAndFree(unsigned int size, void *msg)

Sends *msg* of length *size* bytes to all nodes including the node 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. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent. *msg* should not be overwritten or freed before the communication is complete.

CmiCommHandle CmiAsyncNodeBroadcast(unsigned int size, void *msg)

Initiates asynchronous broadcast of message *msg* of length *size* bytes to all nodes excluding the node on which the caller resides. It returns a communication handle which could be used to check the status of this send using CmiAsyncMsgSent. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to 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. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent. *msg* should not be overwritten or freed before the communication is complete.

CmiCommHandle CmiAsyncNodeBroadcastAll(unsigned int size, void *msg)

Initiates asynchronous broadcast of message *msg* of length *size* bytes to all nodes including the node on which the caller resides. It returns a communication handle which could be used to check the status of this send using CmiAsyncMsgSent. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to 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)

(**Note:** 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. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to 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)

Sends *msg* of length *size* bytes to *npes* processors in the array *pes*.

void CmiSyncListSendAndFree(int npes, int *pes, unsigned int size, void *msg)

Sends *msg* of length *size* bytes to *npes* processors in the array *pes*. Uses CmiFree to deallocate the message buffer for *msg* when the multicast completes. Therefore, *msg* must point to a buffer allocated with CmiAlloc.

CmiCommHandle CmiAsyncListSend(int npes, int *pes, unsigned int size, void *msg)

Initiates asynchronous multicast of message *msg* of length *size* bytes to *npes* processors in the array *pes*. It returns a communication handle which could be used to check the status of this send using CmiAsyncMsgSent. If the returned communication handle is 0, message buffer can be reused immediately, thus saving a call to CmiAsyncMsgSent. *msg* should not be overwritten or freed before the communication is complete.

2.8 Reducing Messaging

Reductions are operations for which a message (or user data structure) is contributed by each partecipant processor. All these contributions are merged according to a merge-function provided by the user. A Converse handler is then invoked with the resulting message. Reductions can be on the entire set of processors, or on a subset of the whole. Currently reductions are only implemented on processors sets. No equivalent exists for SMP nodes.

There are eight functions used to deposit a message into the system, summarized in Table 2.1. Half of them receive as contribution a Converse message (with a Converse header at its beginning). This message must have already been set for delivery to the desired handler. The other half (ending with "Struct") receives a pointer to a data structure allocated by the user. This second version may allow the user to write a simpler merging function. For instance, the data structure could be a tree that can be easily expanded by adding more nodes.

	global	**global with ID**	**processor set**	**CmiGroup**
message	CmiReduce	CmiReduceID	CmiListReduce	CmiGroupReduce
data	CmiReduceStruct	CmiReduceStructID	CmiListReduceStruct	CmiGroupReduceStruct

Table 2.1: Reductions functions in Converse

The signatures for the functions in Table 2.1 are:

void CmiReduce(void *msg, int size, CmiReduceMergeFn mergeFn);
void CmiReduceStruct(void *data, CmiReducePupFn pupFn, CmiReduceMergeFn mergeFn, CmiHandler dest, CmiReduceDeleteFn deleteFn);
void CmiReduceID(void *msg, int size, CmiReduceMergeFn mergeFn, CmiReductionID id);
void CmiReduceStructID(void *data, CmiReducePupFn pupFn, CmiReduceMergeFn mergeFn, CmiHandler dest, CmiReduceDeleteFn deleteFn, CmiReductionID id);
void CmiListReduce(int npes, int *pes, void *msg, int size, CmiReduceMergeFn mergeFn, CmiReductionID id);
void CmiListReduceStruct(int npes, int *pes, void *data, CmiReducePupFn pupFn, CmiReduceMergeFn mergeFn, CmiHandler dest, CmiReduceDeleteFn deleteFn, CmiReductionID id);
void CmiGroupReduce(CmiGroup grp, void *msg, int size, CmiReduceMergeFn mergeFn, CmiReductionID id);
void CmiGroupReduceStruct(CmiGroup grp, void *data, CmiReducePupFn pupFn, CmiReduceMergeFn mergeFn, CmiHandler dest, CmiReduceDeleteFn deleteFn, CmiReductionID id);

In all the above, *msg* is the Converse message deposited by the local processor, *size* is the size of the message *msg*, and *data* is a pointer to the user-allocated data structure deposited by the local processor. *dest* is the CmiHandler where the final message shall be delivered. It is explicitly passed in "Struct" functions only, since for the message versions it is taken from the header of *msg*. Moreover there are several other function pointers passed in by the user:

void * (*mergeFn)(int *size, void *local, void **remore, int count)

Prototype for a CmiReduceMergeFn function pointer argument. This function is used in all the CmiReduce forms to merge the local message/data structure deposited on a processor with all the messages incoming from the children processors of the reduction spanning tree. The input parameters are in the order: the size of the local data for message reductions (always zero for struct reductions); the local data itself (the exact same pointer passed in as first parameter of CmiReduce and similar); a pointer to an array of incoming messages; the number of elements in the second parameter. The function returns a pointer to a freshly allocated message (or data structure for the Struct forms) corresponding to the merge of all the messages. When performing message reductions, this function is also responsible to updating the integer pointed by *size* to the new size of the returned message. All the messages in the *remote* array are deleted by the system; the data pointed by the first parameter should be deleted by this function. If the data can be merged "in-place" by modifying or augmenting *local*, the function can return the same pointer to *local* which can be considered freshly allocated. Each element in *remote* is the complete incoming message (including the converse header) for message reductions, and the data as it has been packed by the pup function (without any additional header) for struct reductions.

void (*pupFn)(pup_er p, void *data)

Prototype for a CmiReducePupFn function pointer argument. This function will use the PUP framework to pup the *data* passed in into a message for sending across the network. The data can be either the same data passed in as first parameter of any "Struct" function, or the return of the merge function. It will be called for sizing and packing. (**Note:** It will not be called for unpacking.)

void (*deleteFn)(void *ptr)

Prototype for a CmiReduceDeleteFn function pointer argument. This function is used to delete either the data stucture passed in as first parameter of any "Struct" function, or the return of the merge function. It can be as simple as "free" or as complicated as needed to delete complex structures. If this function is NULL, the data structure will not be deleted, and the program can continue to use it. Note: even if this function is NULL, the input data structure may still be modified by the merge function.

CmiReduce and *CmiReduceStruct* are the simplest reduction function, and they reduce the deposited message/data across all the processors in the system. Each processor must to call this function exactly once. Multiple reductions can be invoked without waiting for previous ones to finish, but the user is responsible to call CmiReduce/CmiReduceStruct in the same order on every processor. (**Note:** CmiReduce and CmiReduceStruct are not interchangeable. Either every processor calls CmiReduce or every processor calls CmiReduceStruct).

In situations where it is not possible to guarantee the order of reductions, the user may use *CmiReduceID* or *CmiReduceStructID*. These functions have an additional parameter of type CmiReductionID which will uniquely identify the reduction, and match them correctly. (**Note:** No two reductions can be active at the same time with the same CmiReductionID. It is up to the user to guarantee this.)

A CmiReductionID can be obtained by the user in three ways, using one of the following functions:

CmiReductionID CmiGetGlobalReduction()

This function must be called on every processor, and in the same order if called multiple times. This would generally be inside initialization code, that can set aside some CmiReductionIDs for later use.

CmiReductionID CmiGetDynamicReduction()

This function may be called only on processor zero. It returns a unique ID, and it is up to the user to distrubute this ID to any processor that needs it.

void CmiGetDynamicReductionRemote(int handlerIdx, int pe, int dataSize, void *data)

This function may be called on any processor. The produced CmiReductionID is returned on the specified *pe* by sending a message to the specified *handlerIdx*. If *pe* is -1, then all processors will receive the notification message. *data* can be any data structure that the user wants to receive on the specified handler (for example to differentiate between requests). *dataSize* is the size in bytes of *data*. If *dataSize* is zero, *data* is ignored. The message received by *handlerIdx* consists of the standard Converse header, followed by the requested CmiReductionID (represented as a 4 bytes integer the user can cast to a CmiReductionID, a 4 byte integer containing *dataSize*, and the *data* itself.

The other four functions (CmiListReduce, CmiListReduceStruct, CmiGroupReduce, CmiGroupReduceStruct) are used for reductions over subsets of processors. They all require a CmiReductionID that the user must obtain in one of the ways described above. The user is also responsible that no two reductions use the same CmiReductionID simultaneously. The first two functions receive the subset description as processor list (*pes*) of size *npes*. The last two receive the subset description as a previously established CmiGroup (see 2.7).

2.9 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. $\boxed{\beta}$

A recent addition to CONVERSE scheduling mechanisms is the introduction of node-level scheduling designed to support low-overhead programming for the SMP clusters. These functions have "Node" in their names. All processors within the node has access to the node-level scheduler's queue, and thus a message enqueued in a node-level queue may be handled by any processor within that node. When deciding about which message to process next, i.e. from processor's own queue or from the node-level queue, a quick priority check is performed internally, thus a processor views scheduler's queue as a single prioritized queue that includes messages directed at that processor and messages from the node-level queue sorted according to priorities.

void CsdEnqueueGeneral(void *Message, int strategy, int priobits, int *prioptr)

This call enqueues a message to the processor's 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 are explained below.

void CsdNodeEnqueueGeneral(void *Message, int strategy, int priobits, int *prioptr)

This call enqueues a message to the node-level 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 CsdNodeEnqueue(void *Message)

This macro is a shorthand for

```
CsdNodeEnqueueGeneral(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 CsdNodeEnqueueFifo(void *Message)

This macro is a shorthand for

```
CsdNodeEnqueueGeneral(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.

void CsdNodeEnqueueLifo(void *Message)

This macro is a shorthand for

```
CsdNodeEnqueueGeneral(Message, CQS_QUEUEING_LIFO,0, NULL)
```

provided here for backward compatibility.

int CsdEmpty()

This function returns non-zero integer when the scheduler's processor-level queue is empty, zero otherwise.

int CsdNodeEmpty()

This function returns non-zero integer when the scheduler's node-level queue is empty, zero otherwise.

2.10 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.

In each iteration, a 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.

void CsdScheduleForever(void)

Extract and deliver messages until the scheduler is stopped. Raises the idle handling converse signals. This is the scheduler to use in most Converse programs.

int CsdScheduleCount(int n)

Extract and deliver messages until n messages have been delivered, then return 0. If the scheduler is stopped early, return n minus the number of messages delivered so far. Raises the idle handling converse signals.

void CsdSchedulePoll(void)

Extract and deliver messages until no more messages are available, then return. This is useful for running non-networking code when the networking code has nothing to do.

void CsdScheduler(int n)

If n is zero, call CsdSchedulePoll. If n is negative, call CsdScheduleForever. If n is positive, call CsdScheduleCount(n).

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.

2.11 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)

```
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

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 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.

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 processors/nodes of the machine as a tree. For this purpose, CONVERSE defines a tree over processors/nodes. We provide functions to obtain the parent and children of each processor/node. On those machines where the communication topology is relevant, we arrange the tree to optimize communication performance. The root of the spanning tree (processor based or node-based) is always 0, thus the CmiSpanTreeRoot call has been eliminated.

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.

int CmiNodeSpanTreeParent(int nodeNum)

This function returns the node number of the parent of *nodeNum* in the spanning tree.

int CmiNumNodeSpanTreeChildren(int nodeNum)

Returns the number of children of *nodeNum* in the spanning tree.

void CmiNodeSpanTreeChildren(int nodeNum, int *children)

This function fills the array *children* with node numbers of children of *nodeNum* in the spanning tree.

2.16 Isomalloc

It is occasionally useful to allocate memory at a globally unique virtual address. This is trivial on a shared memory machine (where every address is globally unique); but more difficult on a distributed memory machine (where each node has its own separate data at address 0x80000000). Isomalloc provides a uniform interface for allocating globally unique virtual addresses.

Isomalloc can thus be thought of as a software distributed shared memory implementation; except data movement between processors is explicit (by making a subroutine call), not on demand (by taking a page fault).

Isomalloc is useful when moving highly interlinked data structures from one processor to another, because internal pointers will still point to the correct locations, even on a new processor. This is especially useful when the format of the data structure is complex or unknown, as with thread stacks.

void *CmiIsomalloc(int size)

Allocate size bytes at a unique virtual address. Returns a pointer to the allocated region.

CmiIsomalloc makes allocations with page granularity (typically several kilobytes); so it is not recommended for small allocations.

void CmiIsomallocFree(void *doomedBlock)

Release the given block, which must have been previously returned by CmiIsomalloc. Also releases the used virtual address range, which the system may subsequently reuse.

After a CmiIsomallocFree, references to that block will likely result in a segmentation violation. It is illegal to call CmiIsomallocFree more than once on the same block.

void CmiIsomallocPup(pup_er p,void **block)

Pack/Unpack the given block. This routine can be used to move blocks across processors, save blocks to disk, or checkpoint blocks.

After unpacking, the pointer is guaranteed to have the same value that it did before packing.

int CmiIsomallocLength(void *block);

Return the length, in bytes, of this isomalloc'd block.

int CmiIsomallocInRange(void *address)

Return 1 if the given address may have been previously allocated to this processor using Isomalloc; 0 otherwise. CmiIsomallocInRange(malloc(size)) is guaranteed to be zero; CmiIsomallocInRange(CmiIsomalloc(size)) is guaranteed to be one.

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`.

CthThread CthCreateMigratable(CthVoidFn fn, void *arg, int size)

Create a thread that can later be moved to other processors. Otherwise identical to CthCreate.

This is only a hint to the runtime system; some threads implementations cannot migrate threads, others always create migratable threads. In these cases, CthCreateMigratable is equivalent to CthCreate.

CthThread CthPup(pup_er p,CthThread t)

Pack/Unpack a thread. This can be used to save a thread to disk, migrate a thread between processors, or checkpoint the state of a thread.

Only a suspended thread can be Pup'd. Only a thread created with CthCreateMigratable can be Pup'd.

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 CthAwakenPrio(CthThread t, int strategy, int priobits, int *prio)

Pushes a thread into the scheduler queue with priority specified by `priobits` and `prio` and queueing strategy `strategy`. Caution: a thread must only be in the queue once. Pushing it in twice is a crashable error. `prio` is not copied internally, and is used when the scheduler dequeues the message, so it should not be reused until then.

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.

void CthYieldPrio(int strategy, int priobits, int *prio)

This function is part of the scheduler-interface. It simply executes
{`CthAwakenPrio(CthSelf(),strategy,priobits,prio);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 2.9). 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 2.9. 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, int strategy, int priobits, int *prio);

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, CthAwkFn 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 as specific conditions arise. These conditions differ from UNIX signals in that they are raised synchronously, via a regular function call; and that a single condition can call several different functions.

The system-defined conditions are:

CcdPROCESSOR_BEGIN_IDLE Raised when the scheduler first finds it has no messages to execute. That is, this condition is raised at the trailing edge of the processor utilization graph.

CcdPROCESSOR_STILL_IDLE Raised when the scheduler subsequently finds it still has no messages to execute. That is, this condition is raised while the processor utilization graph is flat.

CcdPROCESSOR_BEGIN_BUSY Raised when a message first arrives on an idle processor. That is, raised on the rising edge of the processor utilization graph.

CcdPERIODIC The scheduler attempts to raise this condition every few milliseconds. The scheduling for this and the other periodic conditions is nonpreemptive, and hence may be delayed until the current entry point is finished.

CcdPERIODIC_10ms Raised every 10ms (at 100Hz).

CcdPERIODIC_100ms Raised every 100ms (at 10Hz).

CcdPERIODIC_1second Raised once per second.

CcdPERIODIC_10second Raised once every 10 seconds.

CcdPERIODIC_1minute Raised once per minute.

CcdPERIODIC_10minute Raised once every 10 minutes.

CcdPERIODIC_1hour Raised once every hour.

CcdPERIODIC_12hour Raised once every twelve hours.

CcdPERIODIC_1day Raised once every day.

CcdQUIESCENCE Raised when the quiescence detection system has determined that the system is quiescent.

CcdSIGUSR1 Raised when the system receives the UNIX signal SIGUSR1. Be aware that this condition is thus raised asynchronously, from within a signal handler, and all the usual signal handler restrictions apply.

CcdSIGUSR2 Raised when the system receives the UNIX signal SIGUSR2.

CcdUSER The system never raises this or any larger conditions. They can be used by the user for application-specific use. All conditions from CcdUSER to CcdUSER+256 are so available.

int 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.

int CcdCallOnConditionKeep(condnum,fnp,arg)

As above, but the association is permanent– the given function will be called again whenever this condition is raised.

Returns an index that may be used to cancel the association later.

void CcdCancelCallOnCondition(int condnum, int idx)

void CcdCancelCallOnConditionKeep(int condnum, int idx)

Delete the given index from the list of callbacks for the given condition. The corresponding function will no longer be called when the condition is raised. Note that it is illegal to call these two functions to cancel callbacks from within ccd callbacks.

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` or `CcdCallOnConditionKeep`.

void CcdCallFnAfter(fnp, arg, msLater)

```
CcdVoidFn fnp;
void *arg;
unsigned int msLater;
```

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

Chapter 5

Converse Client-Server Interface

This note describes the CONVERSE client-server (CCS) module. This module enables CONVERSE programs to act as parallel servers, responding to requests from (non-CONVERSE) programs across the internet.

The CCS module is split into two parts– client and server. The server side is the interface used by a CONVERSE program; the client side is used by arbitrary (non-CONVERSE) programs. The following sections describe both these parts.

A CCS client accesses a running CONVERSE program by talking to a `server-host`, which receives the CCS requests and relays them to the appropriate processor. The `server-host` is charmrun for net- versions, and is the first processor for all other versions.

5.1 CCS: Starting a Server

A CONVERSE program is started using

```
charmrun pgmname +pN charmrun-opts pgm-opts
```

charmrun also accepts the CCS options:

`++server`: open a CCS server on any TCP port number

`++server-port=`*port*: open the given TCP port as a CCS server

`++server-auth=`*authfile*: accept authenticated queries

As the parallel job starts, it will print a line giving the IP address and TCP port number of the new CCS server. The format is: "ccs: Server IP = *ip*, Server port = *port* $", where *ip* is a dotted decimal version of the server IP address, and *port* is the decimal port number.

5.2 CCS: Client-Side

A CCS client connects to a CCS server, asks a server PE to execute a pre-registered handler, and receives the response data. The CCS client may be written in any language (see CCS network protocol, below), but a C interface (files "ccs-client.c" and "ccs-client.h") and Java interface (file "CcsServer.java") are available in the charm include directory.

The C routines use the skt_abort error-reporting strategy; see "sockRoutines.h" for details. The C client API is:

void CcsConnect(CcsServer *svr, char *host, int port);

Connect to the given CCS server. svr points to a pre-allocated CcsServer structure.

void CcsConnectIp(CcsServer *svr, int ip, int port);

As above, but a numeric IP is specified.

int CcsNumNodes(CcsServer *svr);

int CcsNumPes(CcsServer *svr);

int CcsNodeFirst(CcsServer *svr, int node);

int CcsNodeSize(CcsServer *svr,int node);
These functions return information about the parallel machine; they are equivalent to the Converse calls CmiNumNodes, CmiNumPes, CmiNodeFirst, and CmiNodeSize.
void CcsSendRequest(CcsServer *svr, char *hdlrID, int pe, unsigned int size, const char *msg);
Ask the server to execute the handler hdlrID on the given processor. The handler is passed the given data as a message. The data may be in any desired format (including binary).
int CcsSendBroadcastRequest(CcsServer *svr, const char *hdlrID, int size, const void *msg);
As CcsSendRequest, only that the handler hdlrID is invoked on all processors.
int CcsSendMulticastRequest(CcsServer *svr, const char *hdlrID, int npes, int *pes, int size, const void *msg);
As CcsSendRequest, only that the handler hdlrID is invoked on the processors specified in the array *pes* (of size *npes*).
int CcsRecvResponse(CcsServer *svr, unsigned int maxsize, char *recvBuffer, int timeout);
Receive a response to the previous request in-place. Timeout gives the number of seconds to wait before returning 0; otherwise the number of bytes received is returned.
int CcsRecvResponseMsg(CcsServer *svr, unsigned int *retSize,char **newBuf, int timeout);
As above, but receive a variable-length response. The returned buffer must be free()'d after use.
int CcsProbe(CcsServer *svr);
Return 1 if a response is available; otherwise 0.
void CcsFinalize(CcsServer *svr);
Closes connection and releases server.

The Java routines throw an IOException on network errors. Use javadoc on CcsServer.java for the interface, which mirrors the C version above.

5.3 CCS: Server-Side

Once a request arrives on a CCS server socket, the CCS server runtime looks up the appropriate handler and calls it. If no handler is found, the runtime prints a diagnostic and ignores the message.

CCS calls its handlers in the usual Converse fashion– the request data is passed as a newly-allocated message, and the actual user data begins CmiMsgHeaderSizeBytes into the message. The handler is responsible for CmiFree'ing the passed-in message.

The interface for the server side of CCS is included in "converse.h"; if CCS is disabled (in conv-mach.h), all CCS routines become macros returning 0.

The handler registration interface is:
void CcsUseHandler(char *id, int hdlr);
int CcsRegisterHandler(char *id, CmiHandler fn);
Associate this handler ID string with this function. hdlr is a Converse handler index; fn is a function pointer. The ID string cannot be more than 32 characters, including the terminating NULL.

After a handler has been registered to CCS, the user can also setup a merging function. This function will be passed in to CmiReduce to combine replies to multicast and broadcast requests.
void CcsSetMergeFn(const char *name, CmiReduceMergeFn newMerge);

Associate the given merge function to the CCS identified by id. This will be used for CCS request received as broadcast or multicast.

These calls can be used from within a CCS handler:
int CcsEnabled(void);
Return 1 if CCS routines are available (from conv-mach.h). This routine does not determine if a CCS server port is actually open.
int CcsIsRemoteRequest(void);
Return 1 if this handler was called via CCS; 0 if it was called as the result of a normal Converse message.
void CcsCallerId(skt_ip_t *pip, unsigned int *pport);
Return the IP address and TCP port number of the CCS client that invoked this method. Can only be called from a CCS handler invoked remotely.

void CcsSendReply(int size, const void *reply);
Send the given data back to the client as a reply. Can only be called from a CCS handler invoked remotely. In case of broadcast or multicast CCS requests, the handlers in all processors involved must call this function.
CcsDelayedReply CcsDelayReply(void);
Allows a CCS reply to be delayed until after the handler has completed. Returns a token used below.
void CcsSendDelayedReply(CcsDelayedReply d,int size, const void *reply);
Send a CCS reply for the given request. Unlike CcsSendReply, can be invoked from any handler on any processor.

5.4 CCS: system handlers

The CCS runtime system provides several built-in CCS handlers, which are available in any CONVERSE job:

ccs_getinfo Takes an empty message, responds with information about the parallel job. The response is in the form of network byte order (big-endian) 4-byte integers: first the number of parallel nodes, then the number of processors on each node. This handler is invoked by the client routine CcsConnect.

ccs_killport Allows a client to be notified when a parallel run exits (for any reason). Takes one network byte order (big-endian) 4-byte integer: a TCP port number. The runtime writes "die
n" to this port before exiting. There is no response data.

perf_monitor Takes an empty message, responds (after a delay) with performance data. When CMK_WEB_MODE is enabled in conv-mach.h, the runtime system collects performance data. Every 2 seconds, this data is collected on processor 0 and sent to any clients that have invoked perf_monitor on processor 0. The data is returned in ASCII format with the leading string "perf", and for each processor the current load (in percent) and scheduler message queue length (in messages). Thus a heavily loaded, two-processor system might reply with the string "perf 98 148230 100 385401".

5.5 CCS: network protocol

This information is provided for completeness and clients written in non-C, non-Java languages. The client and server APIs above are the preferred way to use CCS.

A CCS request arrives as a new TCP connection to the CCS server port. The client speaks first, sending a request header and then the request data. The server then sends the response header and response data, and closes the connection. Numbers are sent as network byte order (big-endian) 4-byte integers– network binary integers.

The request header has three fields: the number of bytes of request data, the (0-based) destination processor number, and the CCS handler identifier string. The byte count and processor are network binary integers (4 bytes each), the CCS handler ID is zero-terminated ASCII text (32 bytes); for a total request header length of 40 bytes. The remaining request data is passed directly to the CCS handler.

The response header consists of a single network binary integer– the length in bytes of the response data to follow. The header is thus 4 bytes long. If there is no response data, this field has value 0.

5.6 CCS: Authentication

By default, CCS provides no authentication– this means any client anywhere on the internet can interact with the server. $authfile$, passed to '++server-auth', is a configuration file that enables authentication and describes the authentication to perform.

The configuration file is line-oriented ASCII text, consisting of security level / key pairs. The security level is an integer from 0 (the default) to 255. Any security levels not listed in the file are disallowed.

The key is the 128-bit secret key used to authenticate CCS clients for that security level. It is either up to 32 hexadecimal digits of key data or the string "OTP". "OTP" stands for One Time Pad, which will generate a random key when the server is started. This key is printed out at job startup with the format "CCS_OTP_KEY> Level i key: *hexdigits*" where i is the security level in decimal and *hexdigits* is 32 hexadecimal digits of key data.

For example, a valid CCS authentication file might consist of the single line ”0 OTP”, indicating that the default security level 0 requires a randomly generated key. All other security levels are disallowed.

Chapter 6

Converse One Sided Communication Interface

This chapter deals with one sided communication support in converse. It is imperative to provide a one-sided communication interface to take advantage of the hardware RDMA facilities provided by a lot of NIC cards. Drivers for these hardware provide or promise to soon provide capabilities to use this feature.

CONVERSE provides an implementation which wraps the functionality provided by different hardware and presents them as a uniform interface to the programmer. For machines which do not have a one-sided hardware at their disposal, these operations are emulated through converse messages.

CONVERSE provides the following types of operations to support one-sided communication.

6.1 Registering / Unregistering Memory for RDMA

The interface provides functions to register(pin) and unregister(unpin) memory on the NIC hardware. The emulated version of these operations do not do anything.

int CmiRegisterMemory(void *addr, unsigned int size);

This function takes an allocated memory at starting address *addr* of length *size* and registers it with the hardware NIC, thus making this memory DMAable. This is also called pinning memory on the NIC hardware, making remote DMA operations on this memory possible. This directly calls the hardware driver function for registering the memory region and is usually an expensive operation, so should be used sparingly.

int CmiUnRegisterMemory(void *addr, unsigned int size);

This function unregisters the memory at starting address *addr* of length *size*, making it no longer DMAable. This operation corresponds to unpinning memory from the NIC hardware. This is also an expensive operation and should be sparingly used.

For certain machine layers which support a DMA, we support the function void *CmiDMAAlloc(int size);

This operation allocates a memory region of length *size* from the DMAable region on the NIC hardware. The memory region returned is pinned to the NIC hardware. This is an alternative to *CmiRegisterMemory* and is implemented only for hardwares that support this.

6.2 RDMA operations (Get / Put)

This section presents functions that provide the actual RDMA operations. For hardware architectures that support these operations these functions provide a standard interface to the operations, while for NIC architectures that do not support RDMA operations, we provide an emulated implementation. There are three types of NIC architectures based on how much support they provide for RDMA operations:

- Hardware support for both *Get* and *Put* operations.

- Hardware support for one of the two operations, mostly for *Put*. For these the other RDMA operation is emulated by using the operation that is implemented in hardware and extra messages.
- No hardware support for any RDMA operation. For these, both the RDMA operations are emulated through messages.

There are two different sets of RDMA operations

- The first set of RDMA operations return an opaque handle to the programmer, which can only be used to verify if the operation is complete. This suits AMPI better and closely follows the idea of separating communication from synchronization. So, the user program needs to keep track of synchronization.
- The second set of RDMA operations do not return anything, instead they provide a callback when the operation completes. This suits nicely the charm++ framework of sending asynchronous messages. The handler(callback) will be automatically invoked when the operation completes.

For machine layer developer: Internally, every machine layer is free to create a suitable data structure for this purpose. This is the reason this has been kept opaque from the programmer.

void *CmiPut(unsinged int sourceId, unsigned int targetId, void *Saddr, void *Taadr, unsigned int size);

This function is pretty self explanatory. It puts the memory location at *Saddr* on the machine specified by *sourceId* to *Taddr* on the machine specified by *targetId*. The memory region being RDMA'ed is of length *size* bytes.

void *CmiGet(unsinged int sourceId, unsigned int targetId, void *Saddr, void *Taadr, unsigned int size);

Similar to *CmiPut* except the direction of the data transfer is opposite; from target to source.

void CmiPutCb(unsigned int sourceId, unsigned int targetId, void *Saddr, void *Taddr, unsigned int size, CmiRdmaCallbackFn fn, void *param);

Similar to *CmiPut* except a callback is called when the operation completes.

void CmiGetCb(unsigned int sourceId, unsigned int targetId, void *Saddr, void *Taddr, unsigned int size, CmiRdmaCallbackFn fn, void *param);

Similar to *CmiGet* except a callback is called when the operation completes.

6.3 Completion of RDMA operation

This section presents functions that are used to check for completion of an RDMA operation. The one sided communication operations are asynchronous, thus there needs to be a mechanism to verify for completion. One mechanism is for the programmer to check for completion. The other mechanism is through callback functions registered during the RDMA operations.

int CmiWaitTest(void *obj);

This function takes this RDMA handle and verifies if the operation corresponding to this handle has completed.

A typical usage of this function would be in AMPI when there is a call to AMPIWait. The implementation should call the CmiWaitTest for all pending RDMA operations in that window.

Chapter 7

Random Number Generation

Converse includes support for random number generation using a 64-bit Linear Congruential Generator (LCG). The user can choose between using a supplied default stream shared amonst all chares on the processor, or creating a private stream. Note that there is a limit on the number of private streams, which at the the time of writing was 15,613.

struct CrnStream;

This structure contains the current state of a random number stream. The user is responsible for allocating the memory for this structure.

7.1 Default Stream Calls

void CrnSrand(int seed);

Seeds the default randon number generator with `seed`.

int CrnRand(void);

Returns the next random number in the default stream as an integer.

int CrnDrand(void);

Returns the next random number in the default stream as a double.

7.2 Private Stream Calls

void CrnInitStream(CrnStream *dest, int seed, int type);

Initializes a new stream with its initial state stored in `dest`. The user must supply a seed in `seed`, as well as the `type` of the stream, where the `type` can be 0, 1, or 2.

double CrnDouble(CrnStream *genptr);

Returns the next random number in the stream whose state is given by `genptr`; the number is returned as a double.

double CrnInt(CrnStream *genptr);

Returns the next random number in the stream whose state is given by `genptr`; the number is returned as an integer.

double CrnFloat(CrnStream *genptr);

Returns the next random number in the stream whose state is given by `genptr`; the number is returned as a float. (Note: This function is exactly equivalent to `(float) CrnDouble(genptr);` .)

Chapter 8

Converse Persistent Communication Interface

This chapter deals with persistent communication support in converse. It is used when point-to-point message communication is called repeatedly to avoid redundancy in setting up the message each time it is sent. In the message-driven model like charm, the sender will first notify the reciver that it will send message to it, the receiver will create handler to record the message size and malloc the address for the upcoming message and send that information back to the sender, then if the machine have one-sided hardware, it can directly put the message into the address on the reciver.

Converse provides an implementation which wraps the functionality provided by different hardware and presents them as a uniform interface to the programmer. For machines which do not have a one-sided hardware at their disposal, these operations are emulated through converse messages.

Converse provides the following types of operations to support persistent communication.

8.1 Create / Destroy Persistent Handler

The interface provides functions to crate and destroy handler on the processor for use of persistent communication.

Persistenthandle CmiCreatePersistent(int destPE, int maxBytes);

This function creates a persistent communication handler with dest PE and maximum bytes for this persistent communication. Machine layer will send message to destPE and setup a persistent communication. A buffer of size maxBytes is allocated in the destination PE.

PersistentReq CmiCreateReceiverPersistent(int maxBytes);
PersistentHandle CmiRegisterReceivePersistent(PersistentReq req);

Alternatively, a receiver can initiate the setting up of persistent communication. At receiver side, user calls CmiCreateReceiverPersistent() which returns a temporary handle type - PersistentRecvHandle. Send this handle to the sender side and the sender should call CmiRegisterReceivePersistent() to setup the persistent communication. The function returns a PersistentHandle which can then be used for the persistent communication.

void CmiDestroyPersistent(PersistentHandle h);

This function destroys a persistent communication specified by PersistentHandle h.

void CmiDestroyAllPersistent();

This function will destroy all persistent communication on the local processor.

8.2 Persistent Operation

This section presents functions that uses persistent handler for communications.

void CmiUsePersistentHandle(PersistentHandle *p, int n)

This function will ask Charm machine layer to use an array of PersistentHandle "p"(array size of n) for all the following communication. Calling with p=NULL will cancel the persistent communication. n=1 is for sending message to each Chare, n¿1 is for message in multicast-one PersistentHandle for each PE.