Converse
Overview

February 11, 1998

The 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

0.1 The Converse Tutorial

Converse is a library of subroutines for parallel processing. It provides subroutines for message transmission, memory sharing, creating threads, and many of the other fundamental needs of a parallel programmer.

The purpose of this tutorial is to introduce the reader to the Converse programming model, and to familiarize the reader with the terminology used throughout the Converse manual.

0.2 Processors, Nodes, and Threads

A program that uses the Converse library is designed to be executed by multiple computers simultaneously. Suppose, for example, you have a network of 10 Sparc IPXs. You might choose to run your program on all 10. Each workstation would run the same executable file. Each workstation would proceed more or less independently, and they'd talk to each other using messages. You choose which computers to use when you start the program. This set cannot be changed while the program is running.

When you have a machine with multiple CPUs, Converse keeps those CPUs as independent from each other as possible. For example, if you have a network of 5 Sparc 20s, with 2 CPUs in each Sparc 20, you have a network of 10 processors. Converse will treat this network almost exactly like a network of 10 separate workstations.

There is one exception to this rule: two processors that happen to reside in the same workstation can share memory with each other. They *only* do this if they ask for it *explicitly*, using the Converse memory-sharing primitives. Unless they do, they function independently. A set of processors that are capable of sharing memory with each other are termed a "node".

Converse numbers the processors and the nodes, starting at zero. If you have a network of 5 Sparc 20s with 2 CPUs each, then you will have 10 processors, numbered 0-9. You will have 5 nodes, numbered 0-4. The layout will be straightforward.

```
Node 0 = processors 0 and 1
Node 1 = processors 2 and 3
Node 2 = processors 4 and 5
Node 3 = processors 6 and 7
Node 4 = processors 8 and 9
```

When processors send messages to each other, they refer to each other by processor number. Subroutines are provided so that code can determine how many processors are present, how many nodes, which processors are part of which nodes, and so forth. Subroutines are also provided so that code can figure out which processor is running it.

Converse provides subroutines to create threads. Each thread is owned by the processor that created it, and never leaves that processor. There can be as many threads in a processor as you have memory to hold. Threads are not preemptive: they context-switch only when you explicitly call the *yield* operator, or when a thread suspends/blocks.

Terminology: Each physical CPU is referred to as a "processor". The word "node" designates a group of processors that are capable of sharing memory with each other. Therefore, each "node" contains one or more "processors". Each "processor" contains one or more "threads".

0.3 Messages, the Scheduler, and Initialization

Messages are commands. Every message contains a function pointer [1] and some data. When you send a message to a processor, you're commanding it to call a specific function, and you're telling it what parameter to pass to the function.

Every processor contains a thread, the "scheduling thread", whose job is to receive and perform these commands. The scheduling thread continuously executes this loop:

```
loop
  receive a message.
  handle the message.
end
```

The phrase "handle the message" means call the function-pointer, and pass in the data. The function that is called is termed the "handler function" or just the "handler". The loop that the

[1] This is half-true — in truth, they actually contain a network-transmittable equivalent of a function-pointer.

scheduling thread executes actually has a name of its own, it is called "the scheduler". There are unusual circumstances in which you may call the scheduler directly.

You must write code not unlike the following code to initialize your Converse program:

```
void myinit(int argc, char **argv)
{
  /* perform miscellaneous initializations */
  /* send some messages, start some threads */
}

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

ConverseInit makes sure that every processor calls the initialization function (myinit). Then, it starts the scheduling thread on every processor. Normally, there is nothing else in main — initialization code goes in the initialization function you write (myinit). Note that ConverseInit never returns [2].

Note that while the initialization function is runs on every processor, the processors don't all have to do the same thing. Just use if-statements to control which processor does what. Suppose you want to initialize by sending just one message (not one per processor, just one). You could do this:

```
void myinit(int argc, char **argv)
{
  /* If I am the processor numbered zero, send one message */
  if (CmiMyPe()==0) { send one message. }
}
```

Many Converse programs don't use threads. Or, to put it more accurately, in many programs, the scheduling thread is the only thread. These programs are message-driven: the processor

[2] There are exceptions to this rule.

does nothing but wait for messages and handles them. These programs work as follows: the initialization function (myinit) sends a message. That message triggers a handler, and the handler sends more messages. Those messages trigger more handlers, which send more messages, and so forth. The cascade of messages continues until the computation is done. This is an elegant way to structure many programs.

Some programmers prefer a more traditional approach, where each processor is running a "main thread". In that case, your initialization function would create a thread (since the initialization function is running on every processor, this will lead to one thread on each processor). Those threads can then interact with each other by sending messages. Of course, messages are still commands. So if a thread T1 on processor P1 wants to interact with another thread T2 on processor P2, then T1 will have to send a message that *affects* T2. For example, T1 can send a command to processor P2, ordering P2 to insert some data into one of T2's data structures. Normally, if thread interaction is what is desired, then one writes a simple abstraction layer on top of the Converse message-commands. Converse already includes several libraries that contain abstractions of this sort. For example, we provide PVM/MPI style messaging, and a variant of Halstead's futures.

bf Terminology: Messages contain function-pointers and data. The function pointed to is called the message's "handler". The thread that will receive the message, and call the function pointer, is called the "scheduling thread". The process of calling the function pointer and passing in the data is "handling the message". The receive/handle loop being performed by the scheduling thread is the "scheduler".

0.4 Global Variables and Pseudo Globals

You have to be careful when using C global variables. Usually, global variables are shared among multiple processors. Suppose I write a global variable declaration like this (not inside any function):

```
int x;
```

If I run it on my 5 Sparc 20s, the compiler will create 5 copies of the variable X. Processors 0 and 1 will share a copy of X, and processors 2 and 3 will share a copy, and so forth. This is not

Converse's choice, it is a function of how the C compiler stores data. Sometimes, though, you don't want sharing. Often, you just want each processor to have its own data. To make this possible, we created "Pseudo Globals". We wrote a set of macros that extend C with this feature. One of these macros is CpvDeclare. CpvDeclare stands for "Converse Private Variable Declare":

```
CpvDeclare(int, y);
```

This macro declares a global variable that isn't shared. To access this variable, you have to use a macro CpvAccess:

```
printf("%d\n", CpvAccess(y));
```

CpvAccess can also be used on the left-hand-side of an assignment statement:

```
CpvAccess(y) = CpvAccess(y) + 1;
```

Note that when we say these variables aren't shared, we mean that multiple processors don't share a Cpv variable. However, if you create threads, then all the threads on the processor will share the Cpv variable. If you want thread-private data, you need CtvDeclare (Converse Thread-Private Variable Declare):

```
CtvDeclare(int, x);
```

You can also use CsvDeclare (Converse Shared Variable Declare) if you want to make it explicit that you want sharing. Keep in mind that no matter what, that's only as shared as the hardware permits.

In short, these macros allow you to specify the level of sharing you want. The full set of macros is defined in the Converse manual.

Terminology: "Csv variables" are pseudo-globals where one copy exists per node. "Cpv variables" are pseudo-globals where one copy exists per processor. "Ctv variables" are pseudo-globals where one copy exists per thread.

0.5 Registering Handlers

Actually, when we said that Converse messages contained function pointers, we simplified a bit. Actually, to use Converse, you have to number the handler-functions in your program. The message contains a *function number*, not an actual function pointer.

This numbering process is typically done during initialization, using a Converse function named "CmiRegisterHandler".

```
void handler_A(void *data) { ... }
void handler_B(void *data) { ... }
void handler_C(void *data) { ... }

CpvDeclare(int, handler_A_number);
CpvDeclare(int, handler_B_number);
CpvDeclare(int, handler_C_number);

void myinit(int argc, char **argv)
{
  /* perform miscellaneous initializations */
  CpvAccess(handler_A_number) = CmiRegisterHandler(handler_A);
  CpvAccess(handler_B_number) = CmiRegisterHandler(handler_B);
  CpvAccess(handler_C_number) = CmiRegisterHandler(handler_C);
  /* send some messages, start some threads */
}

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

```
}
```

The three handler functions you see are the functions that can be invoked by messages. Each one will be assigned a number. These numbers will be stored in Cpv variables. We choose Cpv variables more out of tradition than anything else. Programmers who use Converse for a while will discover that it's usually simplest, safest, and fastest to give each processor its own copy of the data. Shared variables are only needed in rare circumstances. In this case, shared variables would work. However, we like to make it a habit of using Cpv variables, because they are much easier to reason about and the code almost always comes out correct.

Now that the functions have been numbered, and the numbers have been stored in pseudo-globals, we can put those numbers into messages any time we wish.

Terminology: "Registering Handlers" is the process of assigning numbers to handler-functions, so that those numbers can be put into messages. "Handler Numbers" are the numbers that were assigned to the handler-functions.

0.6 The Bytes of a Message

Messages are contiguous sequences of bytes. The first few bytes are always the "header", which contains (among other things) the handler number. The rest is the user-data area. The size of the header is a Converse-defined constant: *CmiMsgHeaderSizeBytes*. The data area can be as large or small as desired. When sending a message, you perform these steps:

- You allocate a buffer for the message, including the header.
- You fill the user-data part of the buffer.
- You put the handler-number in header part of the buffer.
- You call a Converse message-sending function.

This whole process is simplified if you have a struct definition that describes the contents of the message. Suppose you want to send three integers. You would write this code:

```
struct msg_with_three_ints
{
  char header[CmiMsgHeaderSizeBytes];
  int i1;
  int i2;
  int i3;
};

void send_msg_with_three_ints(int pe, int handler, int i1, int i2, int i3)
{
  struct msg_with_three_ints msg;
  msg.i1 = i1;
  msg.i2 = i2;
  msg.i3 = i3;
  CmiSetHandler(&msg, handler);
  CmiSyncSend(pe, sizeof(msg), &msg);
}
```

The struct definition describes a block of memory containing a converse message-header followed by three ints. The function that follows it is used to send a message of this type. It allocates the memory for the message buffer by declaring a local variable. It then fills the user-data area with three assignment statements. It sets the handler number using the Converse function *CmiSetHandler*, and then it sends a copy of the message using *CmiSyncSend*.

Caution: The "sync" in the function name "CmiSyncSend" does *not* mean that the system waits for the message to be delivered. None of the Converse messaging functions wait for the message to be delivered. It only means that the function waits until the message has been put into the network. There are "async" variants of this function that return before the message has even been fed into the network. Those variants are more complex to use, since you have to be careful and not alter the message buffer until the system is done pulling the data out of it.

When the message arrives, the scheduling thread will get a copy of it. It will see the handler number in the message, and will invoke the corresponding handler function. The handler function will receive a pointer to a copy of the message as an argument:

```
  void handlerfn(msg_with_three_ints *msg)
  {
    CmiPrintf("I just received three ints: %d %d %d\n",
msg->i1, msg->i2, msg->i3);
  }
```