CUBIX: Programming Hypercubes without Programming Hosts

JOHN SALMON*

Abstract. Typically, application programs for hypercubes consist of two parts, a master process running on the host and a server running in the nodes of the hypercube.

CUBIX adopts a different viewpoint. Once a program is loaded into the hypercube, that program assumes control of the machine. The host process only serves requests for operating system services. Since it is no more than a file server, the host program is universal; it is unchanged from one application to the next.

This programming model has some important advantages.

- Program development is easier because it is not necessary to write a separate program for the host.
- Hypercube programs are easier to develop and debug because they can use standard I/O routines rather than machine-dependent system calls.
- Hypercube programs can often be run on sequential machines with minimal modification.

Versions of CUBIX exist for both crystalline and amorphous applications. In crystalline applications operating system requests occur synchronously. Requests are either "singular" or "multiple" according to whether all nodes request the same or distinct actions. In amorphous applications requests occur completely asynchronously. The host process serves each request independently. Care is taken so that many processors may simultaneously access the same file.

1. Introduction

CUBIX was created to make programming hypercubes easier. It's goal is to eliminate significant duplication of effort on the part of hypercube programmers, and to make the hypercube environment appear much more familiar to application programmers. It is also intended to make hypercube programs more easily portable to sequential machines as

* Physics Department, California Institute of Technology, Pasadena, CA 91125

well as between different brands of hypercubes.

The motivation for CUBIX can probably best be understood by sitting down with one's favorite hypercube and trying to get each of the nodes to perform a trivial task involving input and output to the console. For example, have each processor identify itself, and multiply its processor number by a number entered on the console, printing an informative message like:

"I am processor 17 and 3 times 17 is 51."

in response to the number 3 being entered. This is an extraordinarily difficult exercise because the nodes of the hypercube do not have direct access to the operating system facilities available on the host. One can not, for instance, execute a "scanf" in the nodes to obtain data from the console. Instead, the host (intermediate host, cube manager, control processor, etc.) must allocate a buffer, read data from the console into it, pass the contents of the buffer to the nodes, read a message for each node containing the results of that node's calculation, format those messages and print the results. Programming this exercise requires two programs, one for the host and one for the nodes of the cube, often compiled with different compilers and different compiler options.

This example is obviously frivolous, but it illustrates an important shortcoming in hypercube programming environments. Maintaining and debugging "real" programs is unnecessarily difficult for exactly the same reason as in the exercise: it is too hard to use the host's operating system. Debugging is extremely difficult because programs cannot be easily modified to produce output tracing the flow of control. Additionally, when a program is modified, it often requires separate but coordinated changes to both the node program and the host program. The necessary coordination is a rich source of minor bugs.

A further deficiency in the hypercube environments is the duplication of effort involved in this programming style. Each programmer is forced to reinvent a host-cube protocol which resembles, functionally at least, the protocols that have been written hundreds, if not thousands, of times already. After writing a few protocols, each programmer tends to develop a characteristic signature. Programmers quickly learn to reuse their 'main' routines, but by then, their time has already been wasted.

Finally, after expending the effort to develop an application on the hypercube, the programmer finds that the program will not run on a sequential machine. The I/O protocol designed for the cube is completely foreign to the sequential machine. Even though the bulk of the application would operate correctly by linking with a very simple library of dummy communication routines, the host program and node program must be "glued" back together. Maintaining an evolving code intended to run on both sequential machines and hypercubes is quite difficult for this reason. (Note that the program, once glued, no longer runs on the hypercube.)

All these deficiencies can be traced to a single source. Hypercubes are viewed as high-speed peripherals attached to a host computer which controls their operation. As peripherals go, they are extremely flexible and programmable, but control, nevertheless, resides in the host. The host loads programs and data into the cube, which then computes and eventually returns results which are expected, in number and length, by the host. In more sophisticated applications, the cube analyzes various tokens passed by the host and may perform different computations based on their values. This general organizational style is familiar to most hypercube programmers.

2. A diff

Th
trol the c
to the co
the cube
of-day c
more tha
such req
serves th
programn

It is
as the sys
learning a
program
quence th
sage, add
the progra

High
the CUB
designed
example,
various fo
Section 1

```
#i

m
{

}
```

3. The cat

It is
computer
additions.
the use of
markedly b
up in the n

How
differ

2. A different perspective

The basic idea behind CUBIX is that the program running in the cube should control the operation of the associated program running on the host. This is exactly opposite to the common style of programming discussed above. In CUBIX, tokens are passed from the cube to the host requesting activities like opening and closing files, reading the time-of-day clock, reading and writing the file system, etc. The host program does nothing more than read requests from the cube, act on them and return appropriate responses. All such requests are generated by subroutine calls in the cube. The host program which serves the requests is universal; it is unchanged from one application to the next, and the programmer need not be concerned with its internal operation.

It is convenient to give the cube subroutines the same names and calling conventions as the system calls they generate on the host. This relieves the programmer of the task of learning a new lexicon of system calls. Any operation he would have performed in a host program can be encoded in a syntactically identical way in the cube. It is of no consequence that the subroutine called in the cube will actually collect its arguments into a message, add a token identifying the request, and send the message to the host for action. All the programmer sees is a call to, e.g., *write(fd, ptr, cnt)*.

High-level utilities are often written in terms of a set of standard system calls. Since the CUBIX system calls have the usual names and calling sequences, system utilities designed for the sequential host computer can be readily ported to the hypercube. For example, the C Standard I/O Library can be compiled and linked with CUBIX allowing various forms of formatted and unformatted buffered I/O. Under CUBIX, the exercise of Section 1 would be programmed as:

```
#include <stdio.h>

main()
{
   int entry, pnum;

   pnum = /* machine dependent specification of local processor number */;
   scanf("%d", &entry);
   fmulti(stdout);                /* see section 4. */
   printf("I am processor %d, and %d times %d is %d\n",
      pnum, entry, pnum, pnum*entry);
   exit(0);
}
```

3. The catch

It is highly optimistic to think that a set of system calls designed for a sequential computer can be sufficient for use in a parallel environment without modifications or additions. In fact, the requirements of the parallel environment do force one to restrict the use of some routines and also to add a few additional ones. The details differ markedly between crystalline and amorphous environments. The two cases will be taken up in the next two sub-sections. In both cases, the issue addressed is the same:

How does one resolve the problem that different processors may need to do different things?

3.1 The crystalline case

Crystalline programs are characterized by uniformity from processor to processor and a computation that proceeds in loose lock-step. Synchronization is maintained by enforcing a rendez-vous whenever data is communicated between processors. Since loose synchronization is the norm in crystalline programs, it is not unreasonable to demand that system calls be made loosely synchronously. That is, it is permissible to call system subroutines whenever all communication channels are free. Furthermore, when a system call is made in one processor, it must be made in all processors at the same time, and with identical arguments. (Exceptions will be discussed shortly.) This neatly resolves the problem of how to deal with disparate requests from different nodes; such an event is declared to be in error.

Of course, there are times when different nodes need to request different actions from the host. The short program in Section 2 contains an example in which each processor attempts to print a different string. CUBIX adds two system calls, *mread* and *mwrite*, to the usual set to allow for distinct I/O operations to be performed by different processors. Both must be called loosely synchronously, but they may have different arguments in each node. Their effect is as follows:

mread(fd, ptr, cnt) causes cnt_0 bytes to be read from the file referred to by file descriptor *fd*, into the memory of processor 0 starting at ptr_0. The next cnt_1 bytes are read from the file into the memory of processor 1 starting at ptr_1, etc. Subscripts refer to the value of the argument in the corresponding processor.

mwrite(fd, ptr, cnt) behaves like *mread*, except that data is copied from the memory of the various processors to the file.

In C programs, it is much more common to use the the Standard I/O Library rather than to use system calls like *read, write, open* and *close* directly. Thus, it is crucial to enhance the Standard I/O Library so users can take advantage of *mread* and *mwrite* along with the usual system calls. In the Standard I/O Library, I/O is directed to *streams*, declared as pointers to type FILE. In CUBIX, streams have a new attribute called *multiplicity*. That is, streams can be in either the *singular* or *multiple* state. The functions, *fmulti(stream)* and *fsingl(stream)* are provided, which change the multiplicity of their argument to multiple and singular, respectively. Singular streams behave in the usual way, and are bound by the usual rules of loose synchronization and identical arguments. Multiple streams form the standard I/O interface to *mread* and *mwrite*. They allow the programmer to read and write data which is distinct in each node of the hypercube. Since output is buffered, queueing data for output to multiple streams need not be synchronous.

On the other hand, flushing the buffer must be done explicitly, and it must be synchronous. Flushing a multiple stream causes the data stored in each processor's buffer to appear in order of increasing processor number. The buffer associated with a stream may be flushed by calling one of *fflush, fclose* or *exit*, simultaneously in all the nodes of the hypercube. Since the programmer has control over when buffers are flushed, he can control,in detail, the appearance of his program's output. For example, the code fragment:

```
fmulti(stdout);
printf("hello\n");
fflush(stdout);
printf("goodbye\n");
fflush(stdout);
```

```
printf("CUBIX ");
printf("is flexible\n");
fflush(stdout);
```

produces the following output when executed in all processors loosely synchronously:

```
hello
hello
...
hello
goodbye
goodbye
...
goodbye
CUBIX is flexible
CUBIX is flexible
...
CUBIX is flexible
```

Multiple input streams are not quite as flexible as output streams because the data must be available to the program when the input routine returns. This is in contrast to output routines which do not guarantee that the data has appeared on the output device upon return from the function. Thus, when input functions like *scanf* and *getc* are applied to multiple streams, each node reads as much of the input stream as necessary and then passes control on to the next node in sequence. The function, *ungetc*, when applied to multiple input streams replaces the last character read by the last processor.

3.2 The amorphous case

Amorphous (i.e. non-crystalline) programs are naturally asynchronous. It would be extremely inconvenient for the programmer to synchronize his calculation every time he wished to produce output or interact with the operating system. The processors in an amorphous CUBIX program are treated as though they are executing separate and independent processes. There is no notion of singular I/O, and there are no requirements of loose synchronization or identical arguments. Most system calls behave in a completely straightforward way when used in an amorphous CUBIX program, but the programmer must beware of asking for system resources too frequently. With currently available hosts, it would be easy to swamp the host's operating system if every node were to simultaneously request the same resource.

There is some difficulty, however, in maintaining numerous open files. If the host's operating system allowed CUBIX to allocate several hundred file descriptors, CUBIX could simply return a distinct file descriptor to every process that requested one. Unfortunately, there is a limit of about twenty simultaneously open files, so the CUBIX host program must remember what files are already open and avoid reopening them. There is still a limit of about twenty simultaneously open file *names*, which means that the programmer usually cannot open a different file for each processor in the cube.

When a file is opened by a processor, that processor's pointer into the file is unchanged by the activity of other processors. Each processor maintains some information about the files it has opened, including the current offset at which to begin the next read or write operation. When a read or write request is sent to the host, this information is sent as well, so the host can "seek" to the correct place before reading or writing the data. Thus, each processor has complete control over the location of each byte it writes into the file. Using this system requires considerable care on the part of the programmer to keep processors using the same file from destroying one another's data. Nevertheless, such care often results in programs whose output is repeatable, so that the order of the data in output files does not depend on tiny variations in processor speed, etc. Aside from difficulty of use, there is another important disadvantage. In order for several processors to share a file, it must make sense for that file to have multiple pointers into it. This is simply not true of devices like terminals, to which data may only be appended.

The UNIX operating system provides for file output in *append* mode, in which each datum is placed at the end of the file, regardless of the offset of the process' current file pointer. CUBIX supports the same idea. Placing output files in append mode is a simple way of guaranteeing that data will not be lost because of several processors writing to the same offset. Output to files in append mode may also be directed to a terminal or other serial device. Append mode has the disadvantage that each record in the file must usually be tagged to indicate its originator. A system to automatically tag each record and record a "table–of–contents" at the end of the file upon closure is under development.

4. Experience with CUBIX

A crystalline version of CUBIX has been running at Caltech since early 1986. A version for amorphous applications was implemented about six months later. Since its introduction, CUBIX has become quite popular, and systems are now operating on the Caltech Mark II and Mark III machines as well as the Intel IpSC and the NCUBE. The prevailing attitude among users is that use of CUBIX is vastly simpler than the old host-cube protocols (even among persons not in the author's immediate family). Several programs have been written for which the same code can be compiled and run on a sequential machine, as well as a hypercube running CUBIX.

CUBIX's most significant drawback seems to be the increased code size in node programs. All computation that would have been done on the host is now done in the nodes of the hypercube. Although it is not any slower to perform inherently sequential tasks simultaneously in many processors, a copy of the code must reside in each processor. It is important to realize that both Standard I/O routines like *printf,* which usually does not appear in non-CUBIX programs, and application-dependent sequential code, which would have appeared in the host program, must now be included in the code that runs in every node. The size of this code can be significant, and reduces the amount of space available for data. The code and data linked by a call to *printf,* for example, requires about 6 kbytes on each node in our implementation. Measures can be taken to reduce the size of application-dependent sequential code. For example, filters can be used with UNIX pipes to massage the data prior to sending it into the cube, or after getting it back. So far, we have not needed more generality than that provided by simple input and output filters. Nevertheless, the possibility remains that in subsequent versions of CUBIX, application programs in the cube could call application-dependent subroutines linked into the host program.

5. Conclusion

Adopting the viewpoint that the program running in the nodes of the hypercube should control the behavior of the host program has some extremely desirable consequences.

- It is possible to write a universal host program which accepts commands generated by subroutine calls in the nodes of the hypercube.
- Given a universal host program, programmers only write one program (the one for the nodes) for any application, eliminating considerable labor and an annoying source of bugs.
- All details of the host-cube interface are hidden from the application programmer. Operating system services are obtained by system calls identical to those used on the host.
- Since applications require only one program to operate on the hypercube, it is usually a simple matter to run them on a sequential machine as a special case.
- Since operating system interaction is, for the most part, the same as in sequential programs, there is considerably less to learn before one can begin writing significant hypercube programs.