aCe-ing

Parallel Programming

A Parallel Programming Tutorial Using the aCe C Language

Version 3

John E. Dorband
Maurice F. Aburdene
Michelle L. McCotter
Andrew J. Marbach

July 2003

This research was funded in part by NASA grant NAG5-10821.

Introduction to aCe

We use this tutorial as a means of introducing a new parallel programming language called aCe C. Contrary to what one might expect, the purpose of the tutorial is not to teach yet another programming language; instead, we use aCe to expound on parallel computing in general. The tutorial in no way assumes to be a comprehensive guide to parallel computing. Instead, we give a brief introduction of parallel processing and then use aCe to show the details of a specific parallel programming language.

Parallel Computing

Concurrent programs are programs that contain multiple processes. These multiple processes usually have the ability to communicate, and must deal with synchronization issues. Concurrent programs are further divided into various other groupings. Experts disagree on specific groupings and definitions, and for that reason we do not go into the details of different types of concurrent programs. Instead, we focus on one type of concurrent programs—the parallel program.

A *parallel program* typically consists of two or more tasks that execute concurrently, and that either use shared data or message passing to communicate. The purpose of a parallel program is to solve a problem (usually large) in a shorter amount of time than standard programs allow. The basic philosophy is to divide a large problem into smaller sections. Then, the sections are assigned to different processors, with each processor being allowed to complete its assigned portion of the task. Also, each processor is given access to all the data and software it needs to complete the task. The separate processors complete their assigned tasks at the same time, thus decreasing the amount of time required to solve the large problem.

Several factors motivate parallel programming, some because of advances in the technology industry, and some because of the nature of parallel programming itself. First of all, advances in hardware components facilitate the development of more parallel machines. Also, advances in communication methods better match high-speed computers. Secondly, because of the growing popularity of parallel programming research, more practical parallel architectures have been developed. Because more effort has been focused on developments in parallel programming, more options are now available. Another advantage comes as a direct result of the second: because more options are available, the cost of building parallel machines has decreased. Development of standards in building computer components, also, has decreased the cost of building parallel computers, because single components can be changed, rather than having to build machines from scratch every time. An advantage of parallel machines themselves is that they exhibit higher fault tolerance, meaning that if one processor goes down, the others keep on working. In a standard one-processor machine, if a processor goes down, the program halts completely. On top of all of this, the capabilities of today's machines are stretched to the limit by engineering and scientific applications, and building faster processors does not seem to be solving the problem. In fact, the development of faster processors is slowing down. There is a fundamental need for faster computers, and standard machines do not seem to promise adequate solutions.

Parallel computing, while allowing programmers to develop rapid solutions to large problems, also comes with its share of costs. It requires a substantial amount of communication and control overhead, potentially slowing computation down. Also, after having trained our minds to think purely sequentially for the majority of programming efforts in the past, with parallel programming we must view problems in a parallel manner. It will take programmers some time to become acclimated to new methods. However, focusing future efforts on parallel programming is enticing, because parallel processing seems to enable greater advances in computing, and it will most likely allow more efficient solutions of large problems in the long run. So teaching parallel programming becomes very important.

Classification of Parallel Systems

The most popular taxonomy for parallel systems was created by Flynn [1] in 1966. It is based on the notion of streams of information flowing into processors. A *stream* consists of instructions and data—pieces of information that either guide operation or are operated on by a processor. In Flynn's Taxonomy, the streams are conceptually divided into instructions and data, regardless of whether the information arrives together or separately. Types of parallel machines are classified based on the content of these streams—SISD (single instruction stream, single data stream), SIMD (single instruction stream, multiple data streams), MISD (multiple instruction streams, single data stream), and MIMD (multiple instruction streams, multiple data streams). These different parallel machines are explained as follows.

- SISD—These are conventional serial Von Neumann computers in which there is only one set of instructions, and thus only one instruction processing unit. Each arithmetic instruction leads to one arithmetic operation on an item of data taken from a single stream of data. The result is a single data stream of logically related arguments and results.

- SIMD—This type of computer has one instruction-processing unit, and several data-processing units, sometimes referred to as p*rocessing elements (PEs)*, with each having its own access to data storage. This computer thus retains a single stream of instructions, but has vector instructions that initiate many operations. The instruction-processing unit is responsible for fetching and interpreting instructions. When it encounters a data-processing instruction, it issues the instruction to all PEs (a vector instruction). All PEs then perform the command using their individual registers, and the result is a data vector comprised of many pieces of data from each PE, hence multiple data streams.
- MISD—This type of machine has multiple instructions operating on a single piece of data, simultaneously. According to Flynn, this class has received much less attention than the other classes. Other computing experts argue that pipelined machines are in this category.
- MIMD—These machines have multiple instruction-processing units for the multiple instruction streams. Several data streams result from the multiple processors, hence multiple instruction streams, multiple data streams. Typically, machines of this type have been specifically designed with parallel programming in mind. In the 1970s, machines built with more than one processor were designed to perform totally separate tasks; these were *task parallel* machines, meaning that different processors executed different code at the same time. Machines with processors that execute the same code at the same time are *data parallel* machines. Since the 1970s, one focus of research has been on data parallel machines, i.e. using multiple processors to execute a single program.

As computer use and development continues, discrepancies arise as to which category a particular computer belongs. Nevertheless, Flynn's Taxonomy remains as a widely accepted means of categorizing parallel machines.

The Parallelism of aCe

The focus language of this tutorial, aCe C, is a superset of ANSI C. aCe stands for "architecture-adaptive computing environment." With few exceptions, code for aCe C, hereon referred to simply as aCe, is derived from ANSI C. This tutorial begins with the assumption that the reader is familiar with ANSI C.

The concepts behind aCe have been gleaned from such parallel languages as APL [2], DAP Fortran [3], Parallel Pascal [4], Parallel Forth [5], C* [6], CM Lisp [7], FGPC [8], and MPL [9]. In contrast to most of these languages, which express parallelism in terms of the geometry of data, aCe expresses parallelism in terms of the geometry of execution and control. aCe was designed to simplify the concept of parallel programming by giving the programmer control of only the essential features of parallel computing (execution and communication). All parallel programming models provide computation and communications operations, yet some models include additional operations such as synchronization and mutual exclusion (ensuring that data access maintains its integrity). aCe provides the programmer with the essential elements of computation and communication, while hiding model-dependent operations such as synchronization and mutual exclusion in the runtime system. This facilitates the programmer's ability to express the algorithm with the highest degree of parallelism while minimizing the need to be concerned with architecture-dependent aspects of parallel computing. aCe is architecture-adaptive, because different virtual architectures may be used for different physical architectures to improve architecture-dependent performance. Other languages use libraries to perform system calls for specific architectures; in aCe, system calls are imbedded in the system. It is in these ideas that aCe has the advantage over other parallel languages.

aCe assumes that every algorithm is based on an inherent virtual architecture consisting of one or more bundles of threads. Typically, a C program has one thread of execution. A *thread of execution* is the path that a computer takes through a program while executing the program. In contrast to C programs, aCe programs consist of many threads of execution. Each of the threads computes in parallel. Each thread bundle has its own instruction stream or thread of execution (i.e. SIMD-like execution). That is, the same instruction stream is applied to all threads within a bundle. Thus aCe is data-parallel, because threads of a bundle execute common threads of execution. An individual thread of a bundle does not have to execute all instructions of the bundle's instruction stream. If a conditional control structure is executed and its condition is false for a thread, that specific thread does not execute the corresponding code. Different bundles of threads will most likely execute different instruction streams (MIMD-like execution). In this, aCe is task-parallel, because threads of different bundles carry out different threads of execution.

More sophisticated compilers and run-time environments may be able to infer from the code which portions of the execution thread may be performed concurrently without conflict. However, it is very difficult to perform this task automatically. The aCe language allows the programmer to explicitly express that which can be performed concurrently (i.e. the parallelism), thus eliminating the need for a compiler to second-guess the intents of the programmer.

The purpose of aCe is to facilitate the development of parallel programs by allowing programmers to explicitly describe the parallelism of an algorithm. The goals of aCe are:

- to allow expression of algorithms in an architecture-independent manner.

- to facilitate the programmer's ability to port algorithms between diverse computer architectures.
- to facilitate the programmer's ability to adapt and implement algorithms on diverse computing architectures.
- to facilitate the ease of optimization of algorithms on diverse computing architectures.
- to facilitate development of applications on heterogeneous computing environments and programming environments for new computer architectures.

As you explore the different facets of aCe, think about how aCe exhibits parallelism. You will probably understand aCe the best by doing some of your own experiments. When questions arise about the implementation of aCe, write some of your own programs to test the compiler. Does aCe behave as you expect it should? If so, you understand it; if not, study some more. When you finish the tutorial, we hope that you have an excellent grasp of the capabilities of aCe as a language, but more importantly, that you have a new understanding of what it means for aCe to be a parallel language. In trying to understand broader concepts of parallel programming, it may be helpful to re-read the introduction after finishing the tutorial. Also, for those of you whose interest in parallel computing grows upon completing this tutorial, we include a list of suggested reading (at the end). Have fun learning!

Table of Contents

Chapter 1: Introducing Threads

1.1 A First aCe Program

The following is an aCe program. Note that it looks very similar to a standard C program.

```
/*
   Hello World aCe program

   Maurice F. Aburdene
   John E. Dorband
*/
#include <stdio.aHr>

int main () {
   printf("Hello aCe World\n");
}
```

Program 1.1: Prints "`Hello aCe World`" once.

Program 1.1 prints

```
Hello aCe World
```

once, as a C programmer would expect. The careful observer may notice that the extension of the header file in this program is different than the extension for C programs. This is due to a difference in compilers, which we will discuss in section 1.6 when we discuss compiling aCe programs.

1.2 Adding Threads to the First Program

Here is a modified version Program 1.1.

```
/*
```

```
   Bundle of Hello Worlds
     aCe program

   Maurice F. Aburdene
   John E. Dorband
*/
#include <stdio.aHr>

threads A[4];

int main () {
   A.{ printf("Hello aCe World\n"); }
}
```

Program 1.2: Creates four 'A' threads. Each thread prints "Hello aCe World" once.

Program 1.2 displays the following output.

```
Hello aCe World
Hello aCe World
Hello aCe World
Hello aCe World
```

This happens because of the creation of threads in our program. Before `main()`, we create four threads calling them 'A'. Inside `main()`, we issue a command to print "`Hello aCe World`" for each of the four threads of 'A'.

aCe and C differ in that C has only one thread of execution, and aCe may have many threads of execution. Each thread may be referenced by name and index (discussed in detail in chapter 4). All aCe programs have *at least one* thread. "Where was the thread in the first program?" you ask. Every program has a primary thread that is *implicitly* named 'MAIN'. This is important when it becomes necessary to communicate between the 'MAIN' thread and other threads executing concurrently with the 'MAIN'. Also note an important distinction: '`main()`' (lower case) is the name of a function, and it is *not* the same as 'MAIN' (upper case).

1.3 Multi-dimensional Arrays of Threads

In the previous section (section 1.2), the bundle 'A' is declared as a one-dimensional array of threads. One of the advantages of aCe, however, is that the language allows thread bundles to be declared with any number of dimensions. Like queues, binary trees, arrays, etc. in other languages, this feature allows a programmer to create structures of thread bundles that logically fit her program. The statement

```
threads B[2][3];
```

declares the bundle ‘B’ to have two dimensions of sizes two and three, respectively. She could instead use a declaration like

```
threads B[6];
```

which has the same number of threads as the previous two-dimensional bundle; however, she may want to set the threads up in two dimensions because, for her particular program, the structure makes more sense that way. Here is a visual image of what is happening with the two-dimensional bundle.

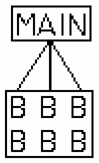


Figure 1.1: A two-dimensional bundle of ‘B’ threads created by the declaration `threads B[2][3];`.

Initially, we have one bundle of threads—namely ‘MAIN’—with one thread in the bundle. When the program creates the ‘B’ threads, the program forks. The ‘B’ threads become children of the parent thread ‘MAIN’, and they exist together in the same bundle, depicted by the box around the ‘B’ threads. (Note: Technically, there should be six lines coming from ‘MAIN’, one to each ‘B’ thread. Since representing a two-dimensional bundle on paper is difficult, the reader will have to use her imagination to fill in the missing lines.)

Here is a modified version of program 1.2.

```
/*
   Two-Dimensional Bundle of Hello Worlds
   aCe program

   Maurice F. Aburdene
   John E. Dorband
*/

#include <stdio.aHr>

threads A[4][2];

int main () {
   A.{ printf("Hello aCe World\n"); }
}
```

Program 1.3: Creates a two-dimensional bundle of threads; each thread prints "`Hello aCe World`".

The '`threads`' command creates a two-dimensional bundle of threads named '`A`'. In any threads declaration, the number of sets of square brackets indicates the number of dimensions, and the number inside each set of square brackets indicates the number of threads in that dimension. In this bundle, the first dimension has four threads, and the second has two. In total, there are 4x2=8 threads.

When program 1.3 is compiled and run, it produces the following output.

```
Hello aCe World
Hello aCe World
Hello aCe World
Hello aCe World
Hello aCe World
Hello aCe World
Hello aCe World
Hello aCe World
```

This demonstrates that no matter how many dimensions a thread bundle possesses, any command that the program issues on the threads of a particular bundle is executed on every thread of that bundle. (The only time a thread does not execute a command that it receives occurs when that thread is inactive. We elaborate on active and inactive threads in chapter 3.) Note that we list eight lines of output from program 1.3, each saying "`Hello aCe World`", for one of the eight threads of '`A`'.

Exercise 1.1: How many threads are in the bundle declared as follows?

```
threads B[2][7][20];
```

Solution: There are 2 x 7 x 20 = 280 threads.

1.4 Bundles of Bundles of Threads

In aCe, one can also declare bundles of bundles of threads. Let's say we want to create bundle of two threads, called '`A`', and another bundle of two threads, called '`B`'. We could do this in the code that follows.

```
threads A[2];
threads B[2];
```

Say, however, that we want to set these bundles up so that 'B' is a sub-bundle of 'A'. To do this, we use a nesting pattern in the declaration. We write down the normal threads declaration for the bundle of threads that we want as the parent bundle—the 'A' threads in this case.

```
threads A[2];
```

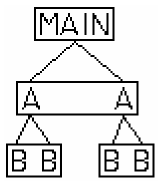
In between the word 'threads' and the name of our bundle of threads ('A'), we place a set of curly brackets, like this

```
threads { } A[2];
```

Then, we fill in the curly brackets with 'B[2]'—the name of the bundle of threads that we want as a sub-bundle of our 'A' bundle, followed by the number of threads (in brackets) that we want in each sub-bundle of 'B' threads.

```
threads {B[2]} A[2];
```

Thus we have created bundles of threads in a structure that looks something like the diagram shown in figure 1.2.



<u>Figure 1.2:</u> 'A' is a bundle of two threads, and a bundle of bundles of 'B' threads.

Note that each sub-bundle of 'B' threads contains two threads, and since there are two sub-bundles of 'B' threads (one for each 'A' thread), then there is a total of four 'B' threads across the entire bundle of 'B' threads.

One can declare any number of bundles and sub-bundles of threads. For example, we can write the declaration

```
threads {B[3],C[4]} A[2];
```

'A' is a bundle of bundles of threads. 'A' consists of two bundles, each with a sub-bundle of three 'B' threads, and a sub-bundle of four 'C' threads. Each bundle of 'A' is basically one 'A' thread, three 'B' threads, and four 'C' threads. Figure 1.3 is a visual representation of the threads declaration.

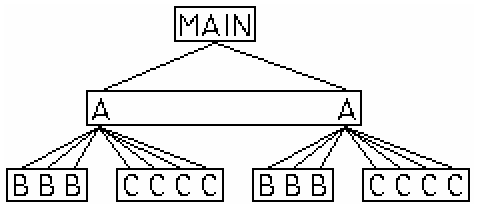


Figure 1.3: 'A' is a bundle of bundles of threads ('B' and 'C'). 'A' is a sub-bundle of 'MAIN', and 'B' and 'C' are sub-bundles of 'A'.

As you can see, a bundle of threads can contain any number of sub-bundles, since the definition of a bundle is recursive. Remember, all bundles are sub-bundles of the primary bundle 'MAIN'. We refer to sub-bundles as children, and to bundles where the sub-bundles come from as parents. The statement

```
threads {{D[2][3],{F[3]} A[2]} C[2]} E[5];
```

demonstrates the recursive nature of a bundle declaration. We have the 'E' threads as the top-most bundle. Under each 'E' thread, there are two 'C' threads. Then, under each 'C' thread, there are two 'A' threads and six 'D' threads (the 'D' threads are two-dimensional). Under each 'A' thread, there are three 'F' threads. You can test your comprehension of this concept by doing the following exercise.

Exercise 1.2: How many 'D' threads result from this threads declaration?

```
threads {{D[2][3],C[2]} B[2]} A[2];
```

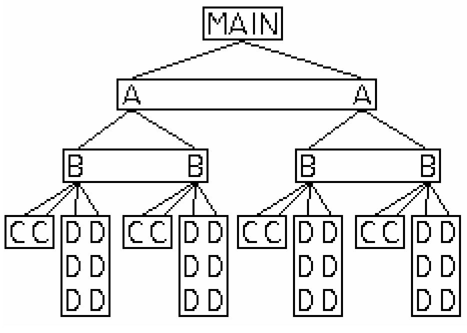
How many 'B' threads are there? How many threads (all names) are there total?

Solution: There are 2x3x2x2 = 24 'D' threads, 2x2 = 4 'B' threads, and 2x3x2x2 + 2x2x2 + 2x2 +2 = 38 threads in all.

Exercise 1.3: Construct a diagram of the threads resulting from this declaration:

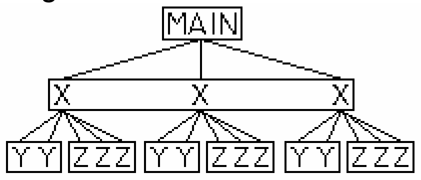
```
threads {{D[2][3],C[2]} B[2]} A[2];
```

Solution:

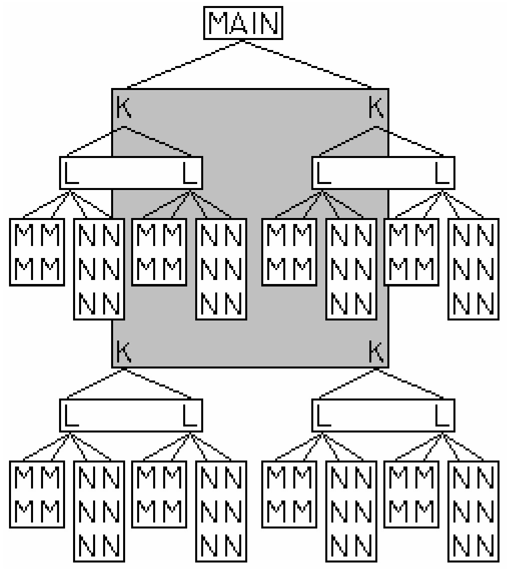

MAIN
A A
B B B B
C C D D D D D D C C D D D D D D C C D D D D D D C C D D D D D D

Exercise 1.4: Write the threads declaration indicated by the following diagrams.

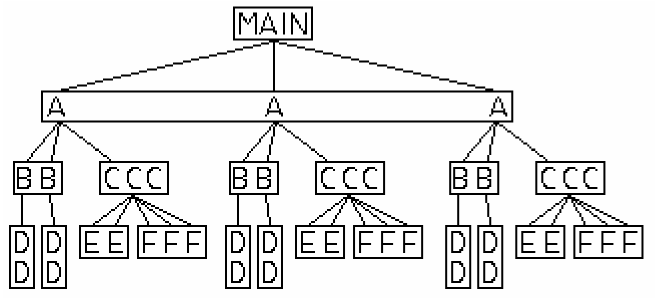
a. *Hint: Write down "*`threads {   } X[3];`*" first. Then fill in the brackets as you move "down" the tree, putting in new "levels" from right to left.*



b.



c.



<u>*Solution:*</u> a. `threads {Y[2],Z[3]} X[3];`
b. `threads {{M[2][2],N[3][2]} L[2]} K[2][2];`
c. `threads {{D[1][2]} B[2],{E[2],F[3]} C[3]} A[3];`

1.5 A Discussion of Parallel Programming

One of the main goals of a parallel program is to solve a problem in a reduced amount of execution time. This can be done so long as the solution process does not require absolutely sequential steps.

In aCe, parallelism is expressed by first defining a set of *concurrently executable threads*. A group of parallel threads can also be viewed as a *bundle of executing threads* or an *array of processing elements*. These three views will be treated synonymously. In aCe, a bundle of threads is defined with the '`threads`' statement. In program 1.2, the statement "`threads A[4]`" declares a bundle of executing threads called '`A`'. This statement declares the intent of the programmer to use four concurrent executions named '`A`' at some later point in the code. If the threads created were executed sequentially, instead of concurrently, the program would take four times as long to complete. Execution time is reduced greatly when the threads are executed concurrently (in parallel).

1.6 Thread Storage

One of the four main concepts in aCe is *storage*. In aCe, there is no global storage, only storage local to each thread. When threads are created, they must be assigned storage before they can execute any code. Storage is declared by a standard C declaration, preceded by the name of the bundle of threads. All threads of a bundle are allocated space for any given declaration.

The following statement allocates an integer '`store`' for each of the four threads of bundle '`A`'.

```
        A int store;
```

A single thread, which we refer to as a *processing element (PE),* only has direct access to storage local to itself. So here, each 'A' thread can only directly access its own `store` variable. A PE can access storage of another PE indirectly, but we will wait until we discuss thread communication in chapter 4 to elaborate on indirect access.

Once storage is assigned to a PE, then code may be written that can be executed by the PE. The following is a sample piece of code that adds the four values of 'y' to the four values of 'z' and stores the four results in the four locations of 'x'.

```
threads A[4];
/* Create three integer storage values for each thread of A
   */
A int x=1, y=2, z=3;

int main() {
   A.{ x = y + z; }
}
```

Here, execution starts with the function 'main()' by the single thread 'MAIN', which transfers control to the four threads of 'A' (via forks). Each 'A' thread adds the values of 'y' to the values of 'z', and stores them into the values of 'x'. Then control returns to 'MAIN'. For parallelism to be useful, the storage of each thread must contain different values. Above, the four values of 'x' are the same; the same is true for 'y' and 'z'. So the example above is not the most useful, but it illustrates the idea of thread storage.

There are methods of assigning different values to variables of the same name, but we do not yet have the tools to do this. Different methods will be discussed after we introduce the system constants in chapter 2, and when we talk about thread communication in chapter 4.

There is one other idea to keep in mind. Code belonging to each bundle of threads is executed on the storage, or data, of each thread in the bundle. All storage belongs to some specific PE, and can only be acted upon by that PE. The storage values, 'x', 'y', and 'z' in the preceding code were created only for the bundle of 'A' threads, and not for any other bundle. So 'MAIN' does not have any storage (in the preceding code), and each PE of 'A' can only access its own storage.

1.7 Printing Stored Values

By printing the stored values from the code before, we see how aCe deals with thread storage. The following program is a modification of the code from section 1.6.

```
/*
   Storage and Printing of Bundle of Threads with Data
   aCe program

   Michelle L. McCotter
*/
#include <stdio.aHr>

threads A[4];
/* Create three integer storage values for each thread of A
   */
A int x=1, y=2, z=3;

int main () {
   A.{
      x = y + z;
      /* Command MAIN thread to print header line */
      MAIN.{ printf("x y z\n-----\n"); }
      /* Command each thread to print its stored values */
   printf("%d %d %d\n",x,y,z);
   }
}
```

Program 1.4: Initializes and prints storage values for threads.

There are two new ideas here: printing the storage values of each processing element, and issuing a print command to the 'MAIN' thread. First, observe the output from program 1.4.

```
x y z
-----
5 2 3
5 2 3
5 2 3
5 2 3
```

After assigning the sum of 'y' and 'z' to 'x' we issue the following command.

```
MAIN.{ printf("x y z\n-----\n"); }
```

This is a command *only* to the 'MAIN' thread. If you recall from section 1.2, each aCe program has a primary thread that is *implicitly* named 'MAIN'. We can issue commands to the 'MAIN' thread as we can any other thread, but because there is only one thread named 'MAIN', whatever command we give to 'MAIN' only gets executed. In program 1.4, since we issue a print command to 'MAIN', the program prints the output we ask it to ("x y z" on one line, and "-----" on the next line), and it only prints that output one time.

After we ask the program to print the header lines, each PE then prints the values of 'x', 'y', and 'z', each PE's values per line. Note that all of the 'x' variables have the same values, as do 'y' and 'z'.

At this point, the reader may be wondering about concurrent execution. *If the threads of 'A' are concurrently executable, meaning that each thread executes the same line of code at the same time, then how do the PE's each print out values on different output lines (making execution seem sequential instead)?* Execution is actually concurrent, and we deal with this issue when we discuss input/output in chapter 3.

1.8 Compiling and Executing aCe Programs

As mentioned previously, aCe has a different compiler than standard C. The aCe compiler requires header files for aCe programs to be terminated by '.aHr', and aCe program files to be terminated by '.aCe'. (Recall that C header files are terminated by '.h', and C program files are terminated by '.c'.)

To compile an aCe program (assuming the aCe compiler has been installed on your machine), type "ace <filename>" at the prompt. The compiler will create an executable for the program named "a.out". No matter what the program is named, the aCe compiler, by default, will name the executable "a.out".

To execute the program, simply type "a.out" at the prompt.

Exercise 1.5: Using any editor you choose, enter the "Hello World" program from section 1.1. After you have finished entering the code, save your file as "hello.aCe". Then compile and run your program. The output should be the same as the output listed in section 1.1.

Exercise 1.6: Repeat exercise 1.5 for any of the other programs listed so far in this chapter. The expected output for each program should appear wherever the programs are located in the chapter.

The aCe compiler allows you to create an executable with a name other than "`a.out`" if you wish. Consider an aCe program file named "`run.aCe`". To create an executable named "`run`", at the prompt, enter the following:

```
[<username>@rhino]$ ace run.aCe –o run
```

This tells the compiler to name the executable "`run`" instead of "`a.out`".

Exercise 1.7: Compile and run any of the programs listed so far in this chapter, naming the executables something other than "`a.out`".

1.9 Summary

- aCe code is, with minor exceptions, the same as code for standard C.

- All aCe programs have at least one primary thread of execution, called '`MAIN`'. Other threads can be created by issuing a command similar to the following

  ```
  threads A[4];
  ```

  where '`A`' is the name of a set of four concurrently executable threads.

- The primary thread of execution, '`MAIN`', is *not* the same is the function '`main()`'.

- Threads can have any number of dimensions desired. Each dimension is indicated by a set of square brackets, and the number of threads in each dimension is indicated by the number inside the square brackets.

- One can declare bundles of bundles of threads. In fact, due to the recursive nature of bundles, a bundle can have any number of sub-bundles.

- A processing element (PE) is another name for a single thread within a bundle of threads.

- One of the four essential aCe concepts is storage. aCe does not have global data storage. Each PE must be allocated its own storage before it can execute any code. Storage is allocated by issuing a command similar to the following

  ```
  A int value;
  ```

  where '`A`' is the name of the bundle of threads to be allocated storage, and '`value`' is the name of the data storage to be allocated to each PE.

- A main goal of parallel programming is to reduce the amount of execution time. This can be done as long as the solution algorithm does not require sequential execution.

- aCe exhibits parallelism by first defining a bundle of executing threads. Then, storage is allocated to each thread, and code can be executed on each thread.

- To execute a command on each thread of a particular bundle, say a bundle called '`A`', the following format is used

  ```
  A.{ /* execution code goes here */ }
  ```

  The commands between the curly braces are executed on each thread of '`A`'. So if '`A`' has four threads, and there is a '`printf(…)`' command between the curly braces, the code in the '`printf(…)`' command is displayed four times.

- When a command of the following format is issued

  ```
  MAIN.{ printf("Print one line.\n"); }
  ```

  the program will only print "`Print one line.`" one time. This is because there is only one primary thread named '`MAIN`'.

- To compile an aCe program, enter the following at the prompt:

  ```
  [<username>@rhino]$ ace <filename>
  ```

The compiler will create an executable named ‘`a.out`’. To create an executable with a different name, issue the following command at the prompt:

```
[<username>@rhino]$ ace <filename> -o <executableName>
```

The compiler will create an executable named `<executableName>`.

- To execute an aCe program, simply enter the name of the executable at the terminal prompt.

Chapter 2: System Constants

2.1 System Constants: An Initial Description

Understanding some inherent values of threads is helpful before we elaborate on more complicated aCe concepts. Hence, we discuss the system constants here. Threads within a bundle are identified by these system-defined constants. There are two categories under which a thread can be identified:

1. Globally, among all threads of a bundle/sub-bundle
2. Within one dimension of a bundle/sub-bundle

Both of the previous categories have three variables:

1. The number of threads within the category
2. The log of the number of threads within the category (only if the number is a power of 2)
3. The sequential identifier of the thread within the category

The definitions of the system constants are as follows:

`$$Name`	--	the name of the bundle/sub-bundle of threads
`$$D`	--	the number of dimensions of the bundle/sub-bundle
`$$N`	--	the total number of threads in the particular bundle/sub-bundle (over all bundles by that same name)
`$$L`	--	the log base 2 of the total number of threads in the bundle/sub-bundle (equals `-1` if `$$N` is not a power of 2)
`$$i`	--	the logical thread identifier; the unique identifier of the thread with respect to all the threads of the bundle/sub-bundle
`$$ii`	--	the physical thread identifier (where the thread is executing within an architecture)
`$$Nx[d]`	--	the number of threads of d-th dimension of the bundle/sub-bundle
`$$Lx[d]`	--	the log base 2 of the number of threads of `d`-th dimension of the bundle/sub-bundle (equals `-1` if `$$Nx[d]` is not a power of 2)

$$ix[d] -- the logical thread identifier of the thread in the d-th dimension of the bundle/sub-bundle

2.2 Thread Identifiers -- $$i and $$ii

In later sections, we discuss all of the system constants; however, we will focus on just two in this section. Although all threads of a particular bundle have the same name, each thread has two unique numbers that identify it--$$i and $$ii. $$i is the value of a thread's logical identifier; it identifies the thread with respect to the other threads in that same bundle. $$ii is the value of a thread's physical identifier; it distinguishes a thread's execution location in an architecture. In many cases, $$ii = $$i, because a thread's physical thread identifier is the same as its logical one. In a chapter 7, we show examples in which the logical identifiers have different values than the physical identifiers in a thread bundle.

Let's look at a simple program that creates a one-dimensional bundle of threads and displays the values of $$i and $$ii for each thread in the bundle.

```
/*
   Sample aCe program
   Prints one-dimensional bundle of threads
   Prints the number of each thread in each thread's
   location in the bundle

   Michelle L. McCotter
   Maurice F. Aburdene
   John E. Dorband
*/
#include <stdio.aHr>
#define size 4
threads A[size] ;

int main () {
   A.{
      MAIN. {printf ("$$i values\n----------\n");}
      printf (" %d",$$i);
      MAIN. {printf ("\n\n$$ii values\n----------\n");}
      printf (" %d",$$ii);
      MAIN. {  printf ("\n"); }
   }
   return 0;
}
```

Program 2.1: Prints the number of each thread in each thread's location in the one-dimensional thread bundle.

Program 2.1 has the following output.

```
$$i values
----------
 0  1  2  3

$$ii values
----------
 0  1  2  3
```

Notice that we declare 'A' as a one-dimensional array of processing elements (remember from chapter 1 that an array of processing elements is the same as a bundle of threads), and the `$$i` and `$$ii` values print out as just that—two bundles, each having one row of four elements. Notice that the value of `$$i` is equal to the value of `$$ii` for each of the threads of 'A'. As mentioned before, `$$i` often has the same value as `$$ii` for a particular thread in a bundle. Also observe that the value of the first thread begins with `0`, and since there are four threads, the last value of `$$i` (or `$$ii`) is `3`.

Now, here's a program similar to program 2.1, but with a two-dimensional bundle of threads.

```
/*
   Sample aCe program
   Prints two-dimensional bundle of threads
   Prints the number of each thread in each thread's
   location in the   bundle

   Michelle L. McCotter
   Maurice F. Aburdene
   John E. Dorband
*/
#include <stdio.aHr>
#define size 4
threads A[size][size] ;

int main () {
   A.{
      MAIN.{printf ("   $$i values\n---------------\n"); }
      printf (" %2d%c",$$i, ($$ix[0]==(size-1))?'\n':' ');
   /* %2d means there will be at least 2 digits places
   printed when $$i prints--this is for lining digits up
   nicely; 3rd argument in previous command prints and end
   of line character if the thread identifier in dimension
   0 is equal to size - 1;  else (indicated by ':') it
   prints a space */
      MAIN.{printf ("\n   $$ii values\n---------------\n");
   }
   printf (" %2d%c",$$i, ($$ix[0]==(size-1))?'\n':' ');
   MAIN. {  printf ("\n"); }
   }
return 0;
}
```

Program 2.2: Prints the number of each thread in each thread's location in the two-dimensional thread bundle.

Here is the output for program 2.2.

```
   $$i values
---------------
  0   1   2   3
  4   5   6   7
  8   9  10  11
 12  13  14  15

   $$ii values
---------------
  0   1   2   3
  4   5   6   7
  8   9  10  11
 12  13  14  15
```

Here, we have two 4x4 arrays, corresponding to our two-dimensional bundle of threads with four threads in each dimension. Notice that no thread has the same value of `$$i` (or `$$ii`) as another thread. Also notice that `$$i` = `$$ii` again. This is often but not always the case, as we will see in chapter 7.

2.3 System Constants for '**MAIN**'

Let's take a look at the system constants for '`MAIN`'. The following program prints out the values for each of the system constants.

```
/*
   Sample aCe program
   Prints each of the system constants for 'MAIN'

   Michelle L. McCotter
*/
```

```
#include <stdio.aHr>

int main () {
   MAIN.{
      printf("System Constants for 'MAIN'\n");
            printf("--------------------------\n");
      printf("$$Name\t=  %s\n",$$Name);
      printf("$$D\t=  %d\n",$$D);
      printf("$$N\t=  %d\n",$$N);
      printf("$$L\t=  %d\n",$$L);
      printf("$$i\t=  %d\n",$$i);
      printf("$$ii\t=  %d\n",$$ii);
      printf("$$Nx[0]\t=  %d\n",$$Nx[0]);
      printf("$$Lx[0]\t=  %d\n",$$Lx[0]);
      printf("$$ix[0]\t=  %d\n",$$ix[0]);
   }
   return 0;
}
```

Program 2.3: Prints system constants for 'MAIN'.

When we execute Program 2.3, the following output results:

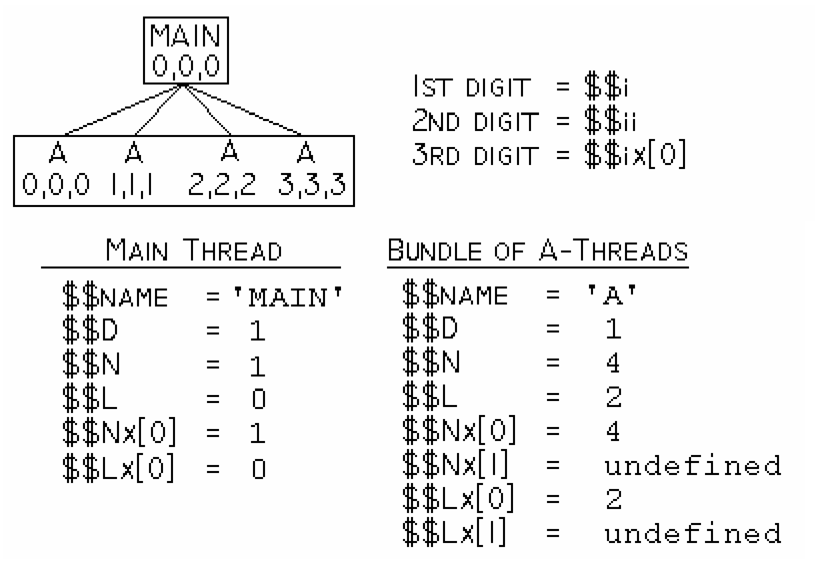
```
System Constants for 'MAIN'
---------------------------
$$Name  =  MAIN
$$D     =  1
$$N     =  1
$$L     =  0
$$i     =  0
$$ii    =  0
$$Nx[0] =  1
$$Lx[0] =  0
$$ix[0] =  0
```

The name of 'MAIN' is as we would expect it to be–'MAIN'. `$$D = 1` shows that 'MAIN' is one-dimensional, and `$$N = 1` shows that there is only one thread in the bundle. `$$L = 0` makes sense, since `$$N = 1 = `2^0. The logical thread identifier of the single thread within the bundle of threads called 'MAIN' (`$$i`) is `0`, as is the physical identifier (`$$ii`). The thread identifier of the thread in the `0`th dimension of the bundle (`$$ix[0]`) is also `0`. Because we begin indexing dimensions at zero, `$$Nx[0]` refers to the first and only dimension that 'MAIN' has; this value is `1`, as we would expect. `$$Lx[0] = 0` makes sense, since `$$Nx[0] = 1 =` 2^0.

2.4 System Constants in One Dimension

Figure 2.1 displays the system constants for two bundles—‘MAIN’, and a one-dimensional array of processing elements created by the following command.

```
threads A[4];
```



MAIN THREAD

\$\$NAME	=	'MAIN'
\$\$D	=	1
\$\$N	=	1
\$\$L	=	0
\$\$Nx[0]	=	1
\$\$Lx[0]	=	0

BUNDLE OF A-THREADS

\$\$NAME	=	'A'
\$\$D	=	1
\$\$N	=	4
\$\$L	=	2
\$\$Nx[0]	=	4
\$\$Nx[1]	=	undefined
\$\$Lx[0]	=	2
\$\$Lx[1]	=	undefined

Figure 2.1: System constants for all bundles of threads created by the declaration `threads A[4];` .

We declare four threads called ‘A’. In figure 2.1, we have two bundles of threads (indicated by two boxes); one bundle is ‘MAIN’, and the other bundle, which is also a sub-bundle of ‘MAIN’, is ‘A’. The first digit under each ‘A’ thread represents the value for `$$i`, the second digit represents the value for `$$ii`, and the third digit represents the value for `$$ix[0]` (the thread identifier of the thread in the 0th dimension; remember that dimensions begin counting at zero, and not at one). These three system constants are listed out individually for each thread in the bundles (‘MAIN’ and ‘A’), because they are not always equal to each other, and each thread within a particular bundle has a different value for each of these constants. Each of these threads have equal values for `$$i` (also `$$ii`) and `$$ix[0]`, because the threads exist in one dimension. As the dimensions increase, which we see in section 2.5, these values become more interesting. In the lower portion of figure 2.1 are two lists of system constants, one for ‘MAIN’, and one for ‘A’. These constants are only listed once, because they are the same for each of the threads in a particular bundle.

Exercise 2.1: Write an aCe program that creates a one-dimensional bundle of two threads. You can start with the program listed in section 2.3. Print out the system constants for each thread in the bundle. (You can print out both values of `$$Name` at a time, then both values of `$$D`, etc.) Keep in mind that if you would like something to print out only once, you can issue the command

```
MAIN.{ printf(/*print code here */); }
```

Try to write the program without looking at the solution. If you get stuck, look to the solution for help, but do your best to figure it out on your own.

Solution:

```
/*
   Sample aCe program
   Prints each of the system constants for all
      threads

   Michelle L. McCotter
*/
#include <stdio.aHr>
threads A[2];

int main () {
   MAIN.{
      printf("System Constants for 'MAIN'\n");
      printf("---------------------------\n");
      printf("$$Name\t=  %s\n",$$Name);
      printf("$$D\t=  %d\n",$$D);
      printf("$$N\t=  %d\n",$$N);
      printf("$$L\t=  %d\n",$$L);
      printf("$$i\t=  %d\n",$$i);
      printf("$$ii\t=  %d\n",$$ii);
      printf("$$Nx[0]\t=  %d\n",$$Nx[0]);
      printf("$$Lx[0]\t=  %d\n",$$Lx[0]);
      printf("$$ix[0]\t=  %d\n",$$ix[0]);
   }

   A. {
      MAIN. {
         printf("\nSystem Constants for 'A'\n");
         printf("------------------------\n");
         printf("$$Name\t=");
      }
      printf("  %s",$$Name);
      MAIN.{ printf("\n$$D\t="); }
      printf("  %d",$$D);
```

```
          MAIN.{ printf("\n$$N\t="); }
          printf("  %d",$$N);
          MAIN.{ printf("\n$$L\t="); }
          printf("  %d",$$L);
          MAIN.{ printf("\n$$i\t="); }
          printf("  %d",$$i);
          MAIN.{ printf("\n$$ii\t="); }
          printf("  %d",$$ii);
          MAIN.{ printf("\n$$Nx[0]\t="); }
          printf("  %d",$$Nx[0]);
          MAIN.{ printf("\n$$Lx[0]\t="); }
          printf("  %d",$$Lx[0]);
          MAIN.{ printf("\n$$ix[0]\t="); }
          printf("  %d",$$ix[0]);
          MAIN.{ printf("\n"); }
      }
      return 0;
  }
```

Program 2.4: Solution to exercise 2.1. Prints out system constants for all threads.

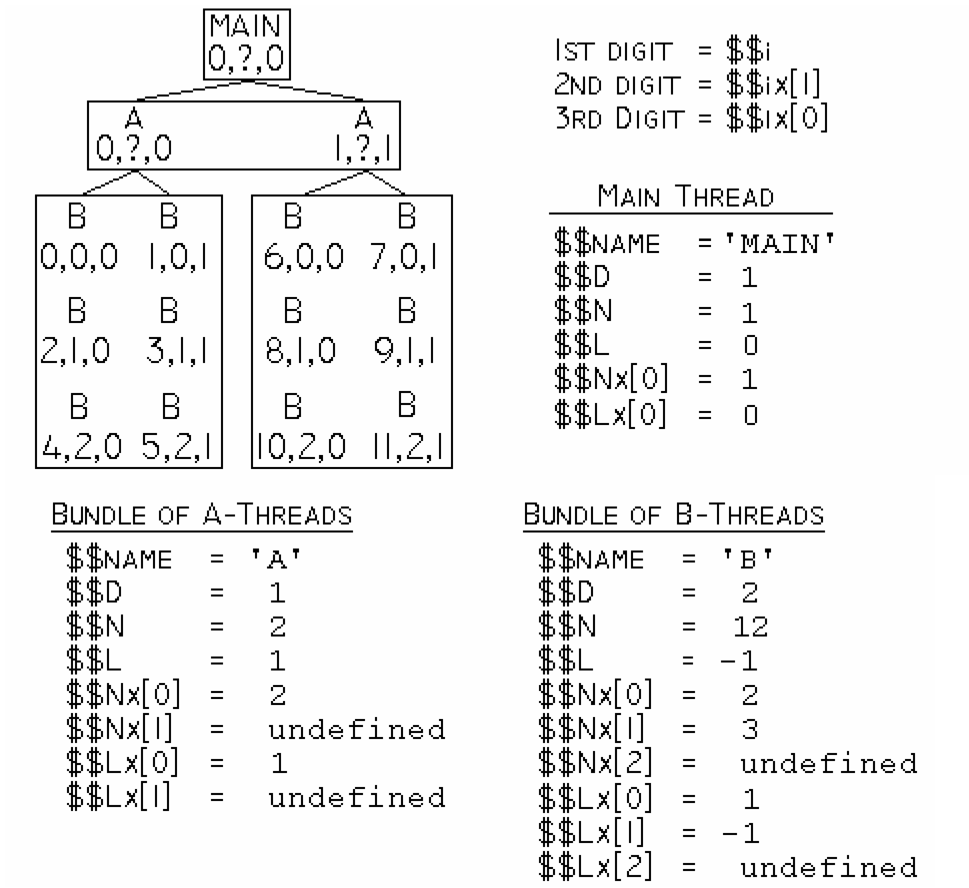
Exercise 2.2: Enter the program listed in the solution to exercise 2.1. Change the threads declaration so that it is two-dimensional. Execute the program, and study the output. What do you notice?

Exercise 2.3: Repeat exercise 2.2 for a threads declaration of three dimensions. Use different numbers of threads in each dimension.

2.5 System Constants for Bundles of Bundles of Threads

Here is a visual example for the following threads declaration.

```
threads {B[3][2]} A[2];
```



1ST DIGIT = $$i
2ND DIGIT = $$ix[1]
3RD DIGIT = $$ix[0]

MAIN THREAD

$$NAME	=	'MAIN'
$$D	=	1
$$N	=	1
$$L	=	0
$$Nx[0]	=	1
$$Lx[0]	=	0

BUNDLE OF A-THREADS

$$NAME	=	'A'
$$D	=	1
$$N	=	2
$$L	=	1
$$Nx[0]	=	2
$$Nx[1]	=	undefined
$$Lx[0]	=	1
$$Lx[1]	=	undefined

BUNDLE OF B-THREADS

$$NAME	=	'B'
$$D	=	2
$$N	=	12
$$L	=	-1
$$Nx[0]	=	2
$$Nx[1]	=	3
$$Nx[2]	=	undefined
$$Lx[0]	=	1
$$Lx[1]	=	-1
$$Lx[2]	=	undefined

Figure 2.2: System constants for all bundles of threads created by the declaration `threads {B[3][2]} A[2];` .

You should understand most of these constants by now. We are choosing to point out a few things that appear here and not in previous examples. In aCe, as in standard C, the compiler indexes the dimensions of an array from right to left. For the '`B`' threads, dimension `0` is the dimension with two threads, and dimension `1` is the dimension with three threads. Observe that `$$L=-1`. This is because the total number of threads of '`A`' is six, and six is not a power of two. Also, `$$Lx[0]=-1`, because `$$Nx[0]`=3, and three is not a power of two. `$$Lx[1]=1`, because `$$Nx[0]=2`, and `2`=2^1. Even though we named the rows first and columns second when we create the '`B`' threads, (i.e. `B[3][2]` for `3` rows and `2` columns), `$$ix[0]` refers to the column index for each '`B`' thread, and `$$ix[1]` refers the row index. This convention may seem contrary to intuition, but it is the method that standard C uses to implement arrays. In C, the least significant dimension is furthest to the right; likewise, the leftmost dimension is the most significant. The last item we wish to draw your attention to is `$$ix[2]`. This constant is undefined because '`A`' has two dimensions; if you recall correctly, dimensions are counted beginning with zero, so even though there are two dimensions, `$$ix[2]` does not make sense because there is no dimension with index two.

Exercise 2.4: Write a program to print out the system constants depicted in figure 2.2. For now, when issuing commands to the 'B' threads, use the following syntax.

```
A.{
  /* commands to 'A' threads */
}
B.{
/* commands to 'B' threads */
}
```

In chapter 3, we discuss the possibility of using other syntax when giving commands to bundles of threads.

Solution:

```
/*
   Sample aCe program
   Creates 2 bundles of threads
   Prints each of the system constants for all
      threads

   Michelle L. McCotter
*/
#include <stdio.aHr>
threads {B[3][2]} A[2];

int main () {

   MAIN.{
      printf("System Constants for 'MAIN'\n");
      printf("--------------------------\n");
      printf("$$Name\t=  %s\n",$$Name);
      printf("$$D\t=  %d\n",$$D);
      printf("$$N\t=  %d\n",$$N);
      printf("$$L\t=  %d\n",$$L);
      printf("$$i\t=  %d\n",$$i);
      printf("$$ii\t=  %d\n",$$ii);
      printf("$$Nx[0]\t=  %d\n",$$Nx[0]);
      printf("$$Lx[0]\t=  %d\n",$$Lx[0]);
      printf("$$ix[0]\t=  %d\n",$$ix[0]);
   }

   A. {
      MAIN. {
         printf("\nSystem Constants for 'A'\n");
         printf("------------------------\n");
         printf("$$Name\t=");
      }
      printf("  %s",$$Name);
      MAIN.{ printf("\n$$D\t="); }
      printf("  %d",$$D);
      MAIN.{ printf("\n$$N\t="); }
      printf("  %d",$$N);
      MAIN.{ printf("\n$$L\t="); }
      printf("  %d",$$L);
      MAIN.{ printf("\n$$i\t="); }
      printf("  %d",$$i);
      MAIN.{ printf("\n$$ii\t="); }
      printf("  %d",$$ii);
      MAIN.{ printf("\n$$Nx[0]\t="); }
      printf(" %2d",$$Nx[0]);
      MAIN.{ printf("\n$$Lx[0]\t="); }
      printf(" %2d",$$Lx[0]);
      MAIN.{ printf("\n$$ix[0]\t="); }
      printf(" %2d",$$ix[0]);
      MAIN.{ printf("\n"); }
   }

   B.{
      MAIN.{
         printf("\nSystem Constants for 'B'\n");
            printf("------------------------\n");
         printf("$$Name\t=");
      }
      printf("  %s",$$Name);
      MAIN.{ printf("\n$$D\t="); }
      printf(" %2d",$$D);
      MAIN.{ printf("\n$$N\t="); }
      printf(" %2d",$$N);
      MAIN.{ printf("\n$$L\t="); }
      printf(" %2d",$$L);
      MAIN.{ printf("\n$$i\t="); }
      printf(" %2d",$$i);
      MAIN.{ printf("\n$$ii\t="); }
      printf(" %2d",$$ii);
      MAIN.{ printf("\n$$Nx[0]\t="); }
      printf(" %2d",$$Nx[0]);
      MAIN.{ printf("\n$$Nx[1]\t="); }
      printf(" %2d",$$Nx[1]);
      MAIN.{ printf("\n$$Lx[0]\t="); }
      printf(" %2d",$$Lx[0]);
      MAIN.{ printf("\n$$Lx[1]\t="); }
      printf(" %2d",$$Lx[1]);
      MAIN.{ printf("\n$$ix[0]\t="); }
      printf(" %2d",$$ix[0]);
      MAIN.{ printf("\n$$ix[1]\t="); }
      printf(" %2d",$$ix[1]);
      MAIN.{ printf("\n"); }
   }
```

```
   return 0;
}
```

Program 2.5: Solution to exercise 2.4. Prints out system constants for all threads. (There are bundles of bundles of threads in this program.)

Exercise 2.5: For each threads declaration listed in parts (a) through (f), determine the values of the system constants listed in (i) through (viii). Do this for the thread specified by name and index number (`$$i`).

a. `threads B[1][2];`
   `(thread 'B', $$i = 1)`
b. `threads B[2];`
   `(thread 'B', $$i = 1)`
c. `threads A[2][2][2];`
   `(thread 'A', $$i = 7)`
d. `threads X[3][4];`
   `(thread 'X', $$i = 9)`
e. `threads {{D[2],C[3]} B[2]} A[3];`
   `(thread 'B', $$i = 2)`
f. `threads {L[3][3],K[3][4],J[2]} M[2];`
   `(thread 'K', $$i = 19)`

i. `$$N`
ii. `$$D`
iii. `$$L`
iv. `$$Nx[0]`
v. `$$Nx[1]`
vi. `$$ix[0]`
vii. `$$ix[1]`
viii. `$$ix[2]`

Solution:

	a	b	c	d	e	f
i	2	2	8	12	6	24
ii	2	1	3	2	1	2
iii	1	1	3	-1	-1	-1
iv	2	2	2	4	2	4
v	1	0	2	3	0	3
vi	1	1	1	1	0	3

vii	0	0	1	2	1	1
viii	undef	undef	1	undef	undef	undef

2.6 Thread Storage Revisited

In section 1.7, we talked about thread storage, and assigning values the variables of each thread. In that section, we only had the tools to assign the same value to a particular variable for all threads. Now, using the system constants, here is an example program that assigns different values to the particular thread variables.

```
/*
   Sample aCe Program
   Creates a bundle of threads
   Creates storage for each thread
   Uses system constants to give storage variables
   different values for each thread

   Michelle L. McCotter
*/
#include <stdio.aHr>
threads A[5];

int main () {
   A. {
      int a,b,c;
      a = $$i;
      b = ($$i+1)%$$N;
      c = (a + b)%$$N;
      /* %$$N assures that b and c will have only values
         0,1,2,3, or 4 */
      printf("Thread %d: a = %d, b = %d, c = %d\n",
         $$i,a,b,c);
   }

   return 0;
}
```

Program 2.6: Uses system constants to give storage variables different values for each thread.

The program output shows that variables of the same name were each assigned different values. So all the ‘a’ variables have different values, as do the ‘b’ and ‘c’ variables.

```
Thread 0: a = 0, b = 1, c = 1
Thread 1: a = 1, b = 2, c = 3
Thread 2: a = 2, b = 3, c = 0
Thread 3: a = 3, b = 4, c = 2
Thread 4: a = 4, b = 0, c = 4
```

In chapter 4, after we discuss thread communication, we show another method of assigning different values to a particular variable of a thread bundle.

2.7 Summary

- Each thread that is created has a set of system-defined constants.
- The values of `$$i` and `$$ii` are unique to each thread within a bundle.
- Often, the value of `$$i` (the logical identifier of a single thread) is the same as the value of `$$ii` (the physical identifier of a single thread) for a particular thread within a bundle/sub-bundle.
- Thread identifiers and thread dimensions begin indexing at zero.
- In aCe, as in standard C, the compiler indexes the dimensions of an array from right to left. Even though we name the rows first and columns second when we create a two-dimensional bundle of threads, `$$ix[0]` refers to the column index for each thread, and `$$ix[1]` refers to the row index.

Chapter 3: Thread Execution

3.1 Operations

One of the four essential concepts of aCe is execution. We have introduced storage and system constants. In this chapter, we will focus on execution and in chapter 4 we will address communication. All operations that can be executed by the thread '`MAIN`' may be performed concurrently by any bundle of threads. '`MAIN`' can perform most operations that are in standard C; the only operation supported by ANSI C, but not by aCe is '`goto`'.

Those programmers who are already familiar with standard C should have an excellent grasp of what can generally be executed in aCe. Because we are assuming that people reading this tutorial are already familiar with standard C, we will not go into general executable statements. Due to the concept of threads, however, there is a difference in code execution for aCe; this difference is due to the option of having *active* and *inactive threads*. When active threads within a bundle execute a command, all of the active threads execute the same command at the same time. Actually, aCe gives the *illusion* that the same command is executed by all active threads at the same time. However, actual execution in aCe may not be, and probably is not, absolutely concurrent. The important feature of aCe is that when active threads within a bundle execute a command, the programmer has the illusion that all of the active threads execute the same code at the same time. Also, the results of execution are the same as if the program had executed each command for a bundle of threads concurrently. This enables aCe to be used on machines with a variety of processors; even on single-processor machines, aCe gives the illusion of absolutely concurrent execution. Concurrent execution for bundles of threads is a particular attribute of aCe that distinguishes it from standard C.

3.2 Active and Inactive Threads

In a data-parallel context, we view conditional execution as the activation and deactivation of threads for which the current code does not apply, rather than as the action of skipping the execution of code that does not apply. Conditional execution on a bundle of threads deactivates all threads for which the condition is false. The inactive threads remain inactive until either the conditional structure is exited or reactivated, as in the case of switch structures or continue statements. A conditional statement only applies to the threads that were active when the statement was entered.

We show active threads and inactive threads with the use of an example. In the code that follows

```
B int a,b,z;
B.{
   if (z) {
      a = b;
   }
}
```

some threads of ‘B’ will copy ‘b’ to ‘a’ while some will not, depending on whether or not ‘z’ is true for each ‘B’ thread. During the copy operation, all threads for which ‘z’ is true are said to be *active*, and all threads for which ‘z’ is false are said to be *inactive*.

Within the context of a segment of code for bundle ‘B’, all active threads of ‘B’ execute the code, while all inactive threads remain idle. Initially, all threads of all bundles are active, and each bundle only executes code explicitly designated for that bundle. Code is explicitly designated for a bundle of threads via execution contexts, which we discuss in section 3.4.

Conditional statements are used to make some threads of a bundle active or inactive for a portion of code. The `if` statement may deactivate some of the active processing elements (individual threads) until the corresponding `else` is reached; then the original PE’s are restored, and those that had been active are deactivated. A loop structure (`for`, `while`, `do-while`) deactivates more and more of the active threads as the threads fail the loop condition. A `break` statement deactivates all active threads until the corresponding loop or switch structure has been exited. A `continue` statement deactivates all active threads until only the current iteration of the loop has been completed. A `return` statement deactivates all active threads until all threads that entered the routine return or complete the routine.

The following program makes use of active and inactive threads.

```
/*
   Sample aCe Program
   Creates one-dimensional bundle of threads
   Demonstrates making threads inactive

   Michelle L. McCotter
*/
#include <stdio.aHr>
/* define values to make active/inactive idea more clear */
#define ACTIVE   1
#define INACTIVE 0
```

```
threads A[4];

int main() {
   int counter = 0;
   A.{
      int state = ACTIVE;
      int a = 0;
      /* all threads are initially active */

      while (state == ACTIVE) {
         MAIN.{ printf("\nLoop # %d\n--------\n",counter);}
         printf("Thread %d is active.\n",$$i);
         MAIN.{ counter++; }
         if ((a-($$i)) == 0)
            state = INACTIVE;
         a++;
      }

      MAIN.{ printf("\nWhile loop finished\n"
                 "------------------\n"); }
      printf("Thread %d is active.\n",$$i);
   }

   return 0;
}
```

Program 3.1: Creates a bundle of threads and uses a `while` statement to make more and more of the threads inactive.

Program 3.1 creates four 'A' threads. As with all aCe programs, the threads are initially active. One must write some kind of conditional code to make them inactive. In program 3.1, we use a `while` statement to do this. As the threads execute through the loop, we "turn off" more and more of them. You can see the effect this has in the program's output that follows.

```
Loop # 0
--------
Thread 0 is active.
Thread 1 is active.
Thread 2 is active.
Thread 3 is active.

Loop # 1
--------
Thread 1 is active.
Thread 2 is active.
Thread 3 is active.
```

```
Loop # 2
--------
Thread 2 is active.
Thread 3 is active.

Loop # 3
--------
Thread 3 is active.

While loop finished
-------------------
Thread 0 is active.
Thread 1 is active.
Thread 2 is active.
Thread 3 is active.
```

As we stated, all threads are initially active. Executions through the `while` loop make more threads inactive. When the `while` loop is finished, all threads are active again, even though we do not explicitly reassign '`ACTIVE`' to '`state`'. Actually, the '`state`' variable is just used as a condition for the `while` loop; the variable itself does not indicate whether or not a thread is active. A thread is active if it executes the line of code at the current location of the bundle of threads. We know that the four '`A`' threads are active after the `while` loop finishes, because all threads print out that they are active, and a thread must be active to execute code.

In the output above, it might seem that we could have made it more clear which threads were inactive in a particular loop. We might have done this, say, by printing out which threads were inactive, instead of just printing the active ones. However, for a thread to print, it must be active. So we cannot use a conditional statement to command the inactive threads to print. Only active threads can print.

Try the following exercises to test your comprehension of active and inactive threads.

Exercise 3.1: The following program uses a loop to make threads inactive. For each iteration through the loop, list the active threads (by `$$i`).

```
#include <stdio.aHr>
threads A[4];

int main() {
   int counter = 0;
   A int i;
   A.{
      for (i=1; i<5; i++) {
      MAIN.{
        printf("\nLoop # %d\n"
                                  "---------\n",counter);
                     }
                     if (($$i)%i != 0)
                     printf("$$i=%d\n",$$i);
                       MAIN.{ counter++; }
                   }
              }
              return 0;
        }
```

Solution:

Loop # 0	Loop # 1	Loop # 2	Loop # 3
none	$$i = 1	$$i = 1	$$i = 1
	$$i = 3	$$i = 2	$$i = 2
			$$i = 3

Exercise 3.2: Write a program that creates a bundle of five threads. Then, use successive iterations through a conditional loop to make each thread of the bundle the only active thread for that iteration. For each iteration, print out the loop number and the logical identifier of the active thread.

Solution:

```
/*
  Sample aCe Program
  Creates one-dimensional bundle of threads
  Demonstrates making threads inactive

  Michelle L. McCotter
*/
#include <stdio.aHr>
#define size 5
threads A[size];

int main() {
   int counterMAIN = 0;
   A int i;
   A.{
      /* all threads are initially active */
      for (i=0; i < size; i++) {
         MAIN.{ printf("\nLoop # %d\n"
                "--------\n",counterMAIN); }
         if ($$i == i)
            printf("Thread %d is active.\n",$$i);
         MAIN.{ counterMAIN++; }
      }

      MAIN.{ printf("\nFor loop finished\n"
             "------------------\n"); }
      printf("Thread %d is active.\n",$$i);
   }
   return 0;
}
```

Program 3.2: Solution to exercise 3.2. Uses a `for` loop to make one thread of the ‘`A`’ threads active at a time; does this for each ‘`A`’ thread.

3.3 Execution of Bundles of Bundles of Threads

A single thread controls whether or not its children (its sub-bundles) can be active. If a thread of a bundle becomes inactive, all of that thread’s children become inactive. If all of the thread’s children become inactive, then the parent thread also becomes inactive. We show both of these cases in this section, one by example, and one with exercise 3.3.

Study the following program (a modified version of program 3.1)

```
/*
   Sample aCe Program
   Demonstrates how a parent thread controls its sub-
      bundles
   If the parent is inactive, all children become inactive.

   Michelle L. McCotter
*/
#include <stdio.aHr>
/* define values to make active/inactive idea more clear */
#define ACTIVE   1
#define INACTIVE 0
threads {B[2]} A[3];

int main() {
   int counter = 0;
   A.{
      int state = ACTIVE;
      int a = 0;
      /* all threads are initially active */

      while (state == ACTIVE) {
         MAIN.{printf("\nLoop # %d\n--------\n",counter);}
         printf("Thread A[%d] is active.\n",$$i);
         B.{ printf("Thread B[%d] is active.\n",$$i); }
         MAIN.{ counter++; }
         if ((a-($$i)) == 0)
            state = INACTIVE;
         a++;
      }

      MAIN.{ printf("\nWhile loop finished\n"
                    "-------------------\n"); }
      printf("Thread A[%d] is active.\n",$$i);
      B.{ printf("Thread B[%d] is active.\n",$$i); }
   }

  return 0;
}
```

Program 3.3: Demonstrates how parent threads control their sub-bundles. If a parent thread becomes inactive, then all of its sub-bundles become inactive.

In program 3.3, we declare a bundle of three 'A' threads, with each 'A' thread having a bundle of two 'B' threads. You can see this in the following diagram.

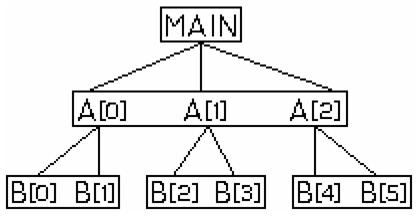


Figure 3.1: The bundles of threads declared by `threads {B[2]} A[3];`

You can see in the output from program 3.3 that when one of the 'A' threads becomes inactive, then both of its 'B' threads also become inactive.

```
Loop # 0
--------
Thread A[0] is active.
Thread A[1] is active.
Thread A[2] is active.
Thread B[0] is active.
Thread B[1] is active.
Thread B[2] is active.
Thread B[3] is active.
Thread B[4] is active.
Thread B[5] is active.
```

```
Loop # 1
--------
Thread A[1] is active.
Thread A[2] is active.
Thread B[2] is active.
Thread B[3] is active.
Thread B[4] is active.
Thread B[5] is active.

Loop # 2
--------
Thread A[2] is active.
Thread B[4] is active.
Thread B[5] is active.

While loop finished
-------------------
Thread A[0] is active.
Thread A[1] is active.
Thread A[2] is active.
Thread B[0] is active.
Thread B[1] is active.
Thread B[2] is active.
Thread B[3] is active.
Thread B[4] is active.
Thread B[5] is active.
```

Remember that the word '`ACTIVE`' in the program is just used as an argument in the conditional statement. It is not actually used to make a thread active. A thread is active if it passes the conditional statement and executes the code inside the conditional block. If execution of a line of code does not take place, the particular thread is inactive. When the conditional block ends, all threads become active again. In program 3.2, we are only concerned with making the '`A`' threads active and inactive, to see how they affect the '`B`' threads; thus we do not use the '`ACTIVE`', '`INACTIVE`', or '`state`' variables with the '`B`' threads.

Program 3.3 shows how making a parent thread inactive causes its children to be inactive as well. By doing exercise 3.3, you can see how a parent thread becomes inactive if all of its children are inactive.

Exercise 3.3: Using any text editor, enter this modified version of program 3.3. Then execute it. What do you notice? *(Note: The thread bundles structure is the same as the structure in program 3.3.)*

```
/*
   Sample aCe Program
  Demonstrates how a child thread
     controls its parent bundles
   If the parent is inactive, all children
   become inactive.

   Michelle L. McCotter
*/
#include <stdio.aHr>
/* define values to make active/inactive
idea more clear */
#define ACTIVE   1
#define INACTIVE 0
threads {B[2]} A[3];

int main() {
  int counter = 0;
  B.{
    int state = ACTIVE;
    int b = 0;
    /* all threads are initially active */

    while (state == ACTIVE) {
      MAIN.{ printf("\nLoop # %d\n"
         "--------\n",
         counter); }
      printf("Thread B[%d] "
      "is active.\n",$$i);
      A.{ printf("Thread A[%d] "
          "is active.\n",$$i); }
      MAIN.{ counter++; }
      if ((b-($$i)) == 0)
     state = INACTIVE;
      b++;
    }

    MAIN.{ printf("\nWhile loop finished\n"
           "------------------"
           "\n"); }
    printf("Thread A[%d] "
       "is active.\n",$$i);
    A.{ printf("Thread B[%d] "
        "is active.\n",$$i); }
    MAIN.{ printf("\n"); }
  }
  return 0;
}
```

3.4 The Execution Context

An *execution context* is the area of code over which a bundle of threads has permission to execute code. Unless stated otherwise (using code), the execution context of ‘MAIN’ is the entire program. When a bundle of threads is created, the programmer must create an execution context for that bundle of threads; otherwise, none of the threads will be able to execute any code.

In program 3.1 from section 3.2, we included some code like the following.

```
threads A[4];
int counter = 0;

int main () {
   A.{
      /* code to execute on 'A' threads here */
   }

   return 0;
}
```

Here, the execution context for the 'A' threads is the block of code between 'A.{ ' and the corresponding '}'. Any time we issue a command like

```
MAIN.{ /* code to execute here */ }
```

we create an execution context for 'MAIN'. After the execution context for 'MAIN' ends (signified by the '}' corresponding to the first '{' after 'MAIN'), the execution context reverts back to the 'A' threads.

Execution contexts may be nested. We have seen this before when 'MAIN' had an execution context inside of 'A'. We see it again after we make a change to the following statement.

```
B.{ a = b; }
A.{ d = c; }
B.{ x = y; }
```

Before we present an alternate version of this statement, we should first understand what it does. Initially, all threads of 'B' copy 'b' to 'a', then all threads of 'A' copy 'c' to 'd', and finally all threads of 'B' copy 'y' to 'x'. This is equivalent to the statement

```
B.{
   a = b;
   A.{ d = c; }
   x = y;
}
```

We initially create an execution context for 'B' in which all threads copy 'b' to 'a'. Then, inside the execution context for 'B', we create a one-line execution context for 'A'. After that ends, the execution context, by default, returns to 'B'.

It is only necessary for contexts to be nested if the execution of one context can implicitly affect the execution context of the other. This can happen if the contained context is contained within a condition statement of the containing context, as in the statement

```
B. {
   if (a)
      A.{ x = y; }
}
```

If ‘a’ is true for any thread of ‘B’, then ‘y’ is copied to ‘x’ for all threads of ‘A’. Otherwise (then ‘a’ is false for every thread of ‘B’) ‘y’ is not be copied to ‘x’ for any thread of ‘A’.

The next example is a little more complicated.

```
B.{
   if (a)
      A.{ x = y; }
   else
      C.{ s = t; }
}
```

As with the previous example, ‘y’ is copied to ‘x’ only if ‘a’ is true for at least one thread of ‘B’. Also, ‘t’ is copied to ‘s’ if ‘a’ is false for at least one thread of ‘B’. An interesting side effect of this code is that ‘y’ will be copied to ‘x’ for all threads of ‘A’ and ‘t’ will be copied to ‘s’ for all threads of ‘C’ as long as ‘a’ is true for some threads and false for others. This statement may be more understandable when you see that there are three possible scenarios.

1. At least one thread of ‘B’ is active, and at least one thread of ‘B’ is inactive, making all threads of ‘A’ and ‘C’ active.
2. All threads of ‘B’ are active, making all threads of ‘A’ active, and all threads of ‘C’ inactive.
3. All threads of ‘B’ are inactive, making all threads of ‘A’ inactive, and all threads of ‘C’ active.

To determine whether or not you understand this concept, try the following exercises.

Exercise 3.4: In the following segment of code, explain what will happen, based on execution contexts.

```
J.{
  if (p || q)
    K.{ x = y;}
  else if (r)
    L.{ s = t;}
  else
    M.{ u = v;}
}
```

Solution: There are 3 main situations that could happen.

1. At least one of ‘p’ or ‘q’ is true for any thread of ‘J’, and both ‘p’ and ‘q’ are false for at least one thread of ‘J’. Then all threads of ‘K’ will be active and one of three situations will happen:
   a. ‘r’ is true for at least one thread of ‘J’, and false for at least one thread of ‘J’. Then all threads of ‘L’ and ‘M’ will be active.
   b. ‘r’ is true for all threads of ‘J’. Then all threads of ‘L’ will be active, and all threads of ‘M’ will be inactive.
   c. ‘r’ is false for all threads of ‘J’. Then all threads of ‘M’ will be active, and all threads of ‘L’ will be inactive.
2. At least one of ‘p’ or ‘q’ is true for all threads of ‘J’. Then all threads of ‘K’ will be active, and all threads of ‘L’ and ‘M’ will be inactive.
3. Both ‘p’ and ‘q’ are false for all threads of ‘J’. Then all threads of ‘K’ will be inactive, and one of the three situations listed in situation 1 (a, b, or c) will happen.

Exercise 3.5: In the following segment of code

```
A.{
  if (a || b) {
    B.{ p = q; }
    if (a)
      C.{ r = s; }
    else
      D.{ t = u; }
  }
  else
    E.{ v = w; }
}
```

what situation is necessary for the threads of

a. ‘E’ to be active?
b. ‘B’ to be active?
c. ‘B’ to be active, and ‘C’ and ‘E’ to be inactive?
d. ‘B’ to be active, and ‘C’, ‘D’ and ‘E’ to be inactive?
e. ‘B’, ‘C’, and ‘D’ to be active, but not ‘E’?
f. ‘B’, ‘D’, and ‘E’ to be active, but not ‘C’?

Solution:

a. ‘a’ and ‘b’ are false for all threads of ‘A’.

b. 'a' or 'b' is true for at least one thread of 'A'.
c. 'b' is true for all threads of 'A'.
d. Not possible
e. 'a' and 'b' are never both false for all threads of 'A', and 'a' is false for at least one but not all threads of 'A'.
f. 'b' is true for at some threads of 'A' and false for others, and 'a' is never true for any thread of 'a'.

Exercise 3.6: Write a segment of code that fulfills the following conditions:

1. Assume the existence of thread bundles 'A', 'B', 'C', and 'D'.
2. Assume the existence of variables 'a', 'b', 'c', 'd', 'p', 'q', 'r', 's', 't', and 'u'.
3. Have all of the condition statements occur inside the execution context of 'A'.
4. If both 'a' and 'b' are true for any thread of 'A', then all threads of 'B' are active, and copy 'q' to 'p'.
5. If both 'a' and 'b' are true for all threads of 'A', then all threads of 'C' and 'D' are inactive.
6. If both 'b' and 'c' are true for any thread of 'A', then
   a. if 'd' is true for any thread of 'A', then all threads of 'C' are active and copy 's' to 'r'.
   b. if 'd' is false for any thread of 'A', then all threads of 'D' are active and copy 't' to 'u'.
7. If 'a' and 'b', or 'b' and 'c', are false for any thread of 'A', then all threads of 'C' are active and copy 'r' to 's'.

Solution:

```
A.{
   if (a && b)
      B.{ p = q; }
   else if (b && c) {
      if (d)
         C.{ r = s; }
      else
         D.{ u = t; }
   }
```

```
            else
               C.{ s = r; }
         }
```

3.5 Input and Output (I/O)

I/O is defined as an order sequence of data that is to be placed in a file. The data can be placed in the file concurrently, but the order within the file will be properly ordered sequentially. In general in aCe, all active threads of a bundle execute the same command at the same time; however, when commands are given in which resources outside of the program itself are needed (i.e. printing), and concurrent execution is not possible (i.e. printing the values of a variable to the screen), aCe execution happens sequentially in order of thread identifier. Observe the following code.

```
threads A[100];
A.{
   fileP = fopen("data","w");  /* open file for writing */
   fprintf(fileP,"$$i= \t%3d\n",$$i);  /* print to file */
   fclose(fileP);  /* close file */
}
```

The preceding code opens the file named ‘`data`’, prints the one hundred values of `$$i` to that file (located at ‘`fileP`’), and closes the file. Note that each ‘`A`’ thread is commanded to open and close the file. This is because each thread of ‘`A`’ needs to know the location of the file, and this location value is obtained by opening the file. Each thread that opens a file needs to close that file when the file is no longer needed.

If the preceding code is contained within a conditional statement, only a subset of the values of `$$i`, corresponding to the active threads, will be written to the file.

```
threads A[100];
A.{
   fileP = fopen("data","w");  /* open file for writing */
   if ($$i > 90)
      fprintf(fileP,"$$i=\t%d\n",$$i);  /* print to file */
   fclose(fileP);  /* close file */
}
```

The code segment will write nine values into the file, from thread `91` through thread `99`.

The following program is an example of I/O in aCe.

```
/*
```

```
   Sample aCe Program
   Creates a two-dimensional bundle of threads
   Prints $$i, $$i[0], and $$i[1] of each thread to a file

   Michelle L. McCotter
*/
#include <stdio.aHr>
#define size 2
threads A[size][size+1];

int main() {
   A.{
      FILE *fileP;  /* Create a pointer to a file */
      int var = 0;
      int ident = $$i;
      MAIN.{ printf("\n-->Values before writing to "
                  "file:\n"); }
      printf("    ident = %d, var = %d\n",ident,var);
      fileP = fopen( "identfile","w+");
      fwrite(&ident,sizeof(ident),1,fileP);
      fclose(fileP);

      fileP = fopen("identfile","r+");
      fread(&var,sizeof(var),1,fileP);
      fclose(fileP);

      MAIN.{ printf("\n-->Values after reading from"
                  "file:\n"); }
      printf("    ident = %d, var = %d\n",ident,var);
   }
   return 0;
}
```

Program 3.4: Prints the values of a variable to a file for each thread in a bundle, then reads the values from the file and assigns them to a new variable.

Program 3.4 creates a two-dimensional bundle of threads, assigns the thread identifiers to the variable '`ident`', and writes the values of '`ident`' to the file "`identfile`". Then, it reads the values from "`identfile`" into the variables '`var`'. We print out the values of the variables before and after the I/O, to allow you to see the effect of reading and writing. Following is the program's output.

```
-->Values before writing to file:
    ident = 0, var = 0
    ident = 1, var = 0
    ident = 2, var = 0
    ident = 3, var = 0
    ident = 4, var = 0
    ident = 5, var = 0
```

```
-->Values after reading from file:
    ident = 0, var = 0
    ident = 1, var = 1
    ident = 2, var = 2
    ident = 3, var = 3
    ident = 4, var = 4
    ident = 5, var = 5
```

aCe does not allow two bundles of threads to write to the same file. If this is attempted, one of the bundles of threads will write over what the other has already written.

aCe has another form of I/O, called *fast I/O.* The fast I/O routines `ffopen`, `ffread`, `ffwrite`, and `ffclose` correspond to the standard I/O routines `fopen`, `fread`, `fwrite`, and `fclose`, except that their use is very machine dependent. Files written by fast I/O must be read by fast I/O. Also, files written with fast I/O must also be read by a bundle of threads with exactly the same geometry as the bundle that wrote it. The reason for this is that fast I/O is intended to be implemented with the fast form of I/O available on the architecture, which may differ from architecture to architecture.

3.6 Summary

- One of the four essential concepts in aCe is execution.

- All operations that can be executed by '`MAIN`' can be executed by any bundle of threads.

- aCe code, with minor exceptions, is the same as code for standard C.

- aCe gives the programmer the *illusion* that every active thread executes each line of code (within the bundle's execution context) at the same time. Execution merely appears to be absolutely concurrent, when, in fact, it is not. The importance is that the programmer believes that execution for a bundle of threads is concurrent, and the end result of execution is as though execution were entirely concurrent.

- The only operation supported by ANSI C but not by aCe is '`goto`'.

- Conditional statements are used to make individual threads of a bundle active or inactive. They can also be used to make whole bundles of threads active or inactive.

- Active threads execute code given to them; inactive threads do not.

- When a thread has any number of sub-bundles, if that thread becomes inactive, then all of the sub-bundles also become inactive. Also, if all of the sub-bundles become inactive, then the parent thread also becomes inactive.

- The execution context is the area of code over which a particular bundle of threads has permission to executed commands.

- Execution contexts may be nested, but do not have to be. So although thread bundle '`B`' may be a sub-bundle of thread bundle '`A`', the execution context of '`B`' can be, but does not have to be, nested inside the execution contest of '`A`'.

- I/O in aCe is in order of thread identifier. Data is placed in a file concurrently, but the data in the file are properly ordered sequentially.

- aCe does not allow two different bundles of threads to write to the same file. If this is attempted, one of the bundles will write over what the other has already written.

Chapter 4: Thread Communication

4.1 Thread Communication Within a Bundle of Threads

Another of the four essential aCe concepts is communication, which we discuss in this chapter. In section 1.6, we mentioned the idea of thread storage access. A thread can only directly access its own storage, not the storage of any other thread, even if the threads are in the same bundle. However, there are methods that allow indirect access to storage of other threads. For now, we focus on thread storage access within the same bundle of threads.

Indirect access to thread storage requires communication between threads. In aCe, there are two basic communication operations. Other programming languages refer to them as ‘`get`’ and ‘`put`’ operations. Following is an example of ‘`get`’ in aCe.

```
A. {
   int a,b,c;
   b = ($$i+1) * ($$i-1);
   c = ($$i+1)%$$N;
   a = A[c].b;
}
```

In this code, the value of ‘`b`’ is fetched from one thread of ‘`A`’ to another thread of ‘`A`’. This operation is depicted in figure 4.1.

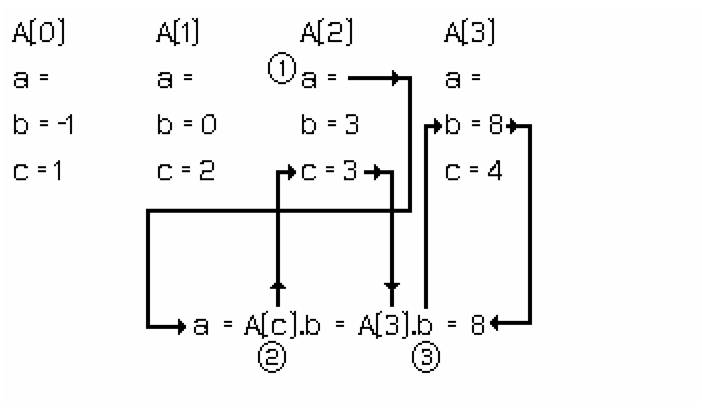


Figure 4.1: An aCe ‘`get`’ operation.

In aCe, all threads of a particular bundle are executing at the same time. For simplicity, we have shown the steps of the 'get' operation for only one of the threads of 'A'. We arbitrarily show steps for A[2]. (1) When the program reaches the line of code

```
a = A[c].b
```

it knows that the value of 'a' for the 'A' thread of index 2 will come from the bundle of threads called 'A' (indicated by the 'A' in A[c].b). The value of the 'c' in the A[c].b comes from the 'c' of the current thread. That thread is the one for which $$i=2; we represent that thread in figure 4.1 as A[2]. (2) Thus c=3 for the current thread. (3) So together, A[c]=A[3] means that we're going to look to the thread A[3] for the value of 'b' that we want. Because b=8 for A[3], then a=A[c].b=8.

An aCe 'put' operation works in a similar manner. Observe the code for a 'put' operation.

```
A.{
   int a,b,c;
   a = ($$i-1) * ($$i+1);
   c = ($$i+1)%$N;
   A[c].b = a;
}
```

With this code, the value of 'a' of the current thread is stored in 'b' of a different thread. The 'put' operation is depicted in figure 4.2.

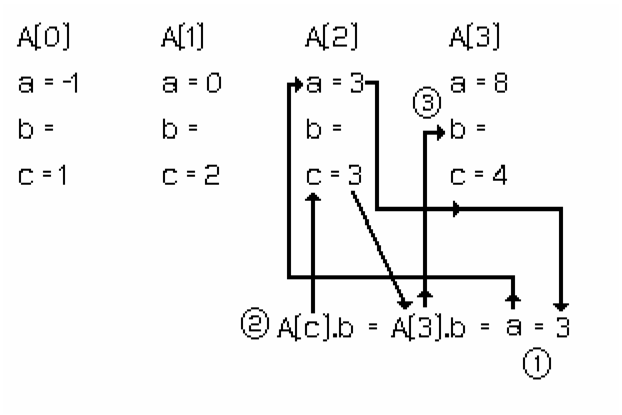


Figure 4.2: An aCe 'put' operation.

We are assigning the value of 'a' of the current thread to A[c].b. (1) The computer determines the value of 'a' for the current thread (in figure 4.2, that thread is A[2]). (2) Then, the computer determines the value of 'c' in A[c].b, and finds that `A[c].b` = `A[3].b`. (3) Finally, the computer stores (puts) the value of ‘`a`’ of the current thread (`a=3`) in the location `A[3].b`.

The following program demonstrates thread communication by using a ‘`put`’ operation to compute the sum of all of the logical thread identifiers of bundle ‘`A`’.

```
/*
  Computes sum of all $$i for a bundle of threads
  Example of thread communication
  Sample aCe Program

  Michelle L. McCotter
*/
#include <stdio.aHr>
#define SIZE 8
threads A[SIZE];

int main() {
   int sum;
   /* Here, we use thread communication to compute the sum
   of the eight logical 'A' thread identifiers--this is
   equivalent to computing the sum of the first seven
   integers */
   A.{
      int sum=0;
      A[0].sum += $$i;
      MAIN.{ printf("The sum of the %d logical thread\n"
                "identifiers in the bundle is %d.\n",
                SIZE,A[0].sum); }
   }
   /* Now, we use a common method to compute the sum of the
   first seven integers */
   sum = (SIZE-1)*(SIZE)/2;
   printf("\nThe sum of the first %d integers "
          "is %d.\n",SIZE-1,sum);

   return 0;
}
```

Program 4.1: Uses two different methods to compute the sum of the first seven integers.

In program 4.1, we use thread communication to compute the sum of the first seven integers (equal to the first eight logical thread identifiers). We also use a known formula ($\texttt{sum}_n$`=(n)(n+1)/2, with n=SIZE-1`) to do the same thing, to show that our computation with thread communication works. Here is the program’s output.

```
The sum of the 8 logical thread
```

```
identifiers in the bundle is 28.

The sum of the first 7 integers is 28.
```

Notice that the sum of the first eight logical thread identifiers is equal to the sum of the first seven integers. (This is because the indexing of threads begins with zero, and not one.)

Exercise 4.1: Write a program that uses thread communication (either '`get`' or '`put`'—it can be done both ways) to compute `9!` Also include a computation using a standard method to compare your thread's computation with the actual value. (By the way, `9!=362880`.)

Solution:

```
/*
   Computes factorials using threads
   Uses thread communication
   Sample aCe Program

   Michelle L. McCotter
*/
#include <stdio.aHr>
#define size 10
threads A[size];

int main() {
int factorial=1,i;

   A.{
      int factorial=1,i,ident;
      ident = $$i;
      if ($$i != 0)
         A[0].factorial *= ident;
      if ($$i == 0)
         printf( "\nUsing the 'put' method:\n"
                "-->%d!, computed via\n"
                  "   thread communication, "
                  "is %d.\n",size-1,factorial);

      if ($$i == 0) {
         factorial=1;
       for (i=1; i < size; i++)
         factorial *= A[i].ident;
      printf( "\nUsing the 'get' method:\n"
              "-->%d!, computed via \n"
              "thread communication, "
```

```
                "is %d.\n",size-1,factorial);
      }
   }

   for (i=1; i<size; i++)
      factorial = factorial * i;
   printf( "\nUsing the standard method:\n"
            "-->%d!, computed via a \n"
            " standard method, is %d.\n",
            size-1,factorial);
   return 0;
}
```

Program 4.2: Computes 9! twice using thread communication methods of '`get`' and '`put`', and again using a standard method.

4.2 Communications Path Descriptions

Communication between two threads is defined by a communication expression. A communication expression consists of two parts:

1. the *communication path description*, or *router expression*, and
2. the *remotely evaluated numeric expression*, or *remote expression.*

In the case of the '`get`' operation, discussed in section 4.1 and listed again here,

```
A.{
   int a,b,c;
   b = ($$i+1) * ($$i-1);
   c = ($$i+1)%$$N;
   a = A[c].b;
}
```

we focus on the statement `a=A[c].b`. In this statement, `A[c]` is the router expression, and '`b`' is the remote expression. In most cases, the '`c`' in the router expression indicates the thread for which `$$i=c`. However, if we have a multi-dimensional bundle of threads like this one

```
threads B[2][3];
```

we write the router expression in two different ways, either

`B[x].z` or `B[x][y].(z).`

The first method indicates the ‘`B`’ thread with `$$i=x`; the second indicates the ‘`B`’ thread with `$$ix[0]=x` and `$$ix[1]=y`. (Note: In the declaration

```
threads B[2][3];
```

dimension zero has three threads (is referred to second), and dimension one has two threads (is referred to first). However, if one chooses to use a router expression with two indices, the index for dimension zero goes first, and not second.) If you ever use the second method, be sure to enclose the remote expression, even if it is just a single variable.

Although router expressions are not all that difficult to understand, we provide exercise 4.2 for you to test your comprehension.

Exercise 4.2: Identify the (a) communication expression, the (b) router expression, and the (c) remote expression in the following examples.

i. `B[b].c = a`
ii. `G[b+1].c = d`
iii. `x = Y[0].(y+2)`
iv. `y = A[$$ix[0]].(z+1)`
v. `z = A[B[c].d].a`
vi. `D[k][l].n = m`

Solution:

	(a)	(b)	(c)
i	`B[b].c`	`B[b]`	`c`
ii	`G[b+1].c`	`G[b+1]`	`c`
iii	`Y[0].(y+2)`	`Y[0]`	`y+2`
iv	`A[$$ix[0]].(z+1)`	`A[$$ix[0]]`	`z+1`
v.	`A[B[c].d].a`	`B[c], A[B[c].d]`	`d,a`
vi.	`D[k][l].n`	`D[k][l]`	`n`

Part v. is a trick question. There are two router expressions here, one nested inside the other. This goes to show that router expressions can indeed be nested.

4.3 Thread Communication Between Different Thread Bundles

Here, we show the method of thread communication between two bundles with the use of an example. In the next section, we will discuss communication path descriptions.

Consider the bundles of threads declared the following line of code.

```
threads {B[3]} A[2];
```

The structure of these bundles of threads is shown in figure 4.4.

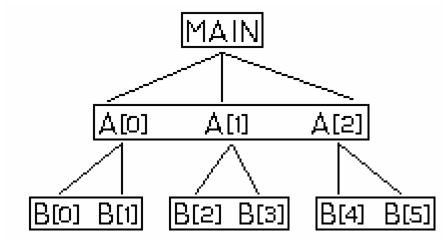


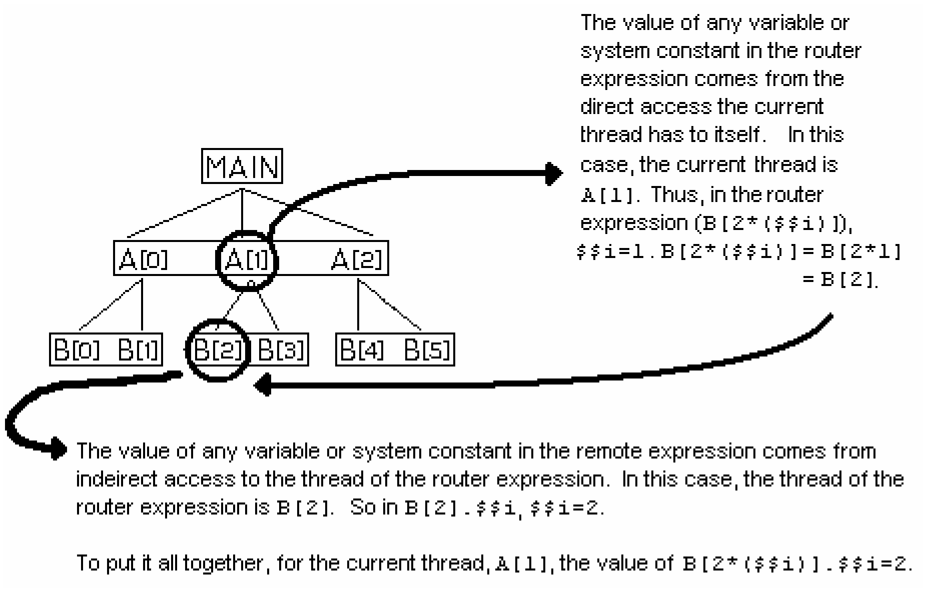
Figure 4.3: Bundles of threads declared by `threads {B[3]} A[2];`

Let's say we declare an integer storage variable called '`sum`' for each of the '`A`' threads, and we want this variable to be the sum of the thread identifiers in each '`A`' thread's sub-bundle of '`B`' threads. (i.e. '`sum`' for the left-most '`A`' thread would be equal to the sum of the `$$i` values for each of the '`B`' threads "underneath" of it; so `sum=0+1=1`). To do this, we write the following code.

```
A.{ sum = B[2*($$i)].$$i + B[2*($$i)+1].$$i; }
```

This code says that '`sum`' for each '`A`' thread will be obtained from a sum of two values. The first of these values comes from the '`B`' thread with twice the logical identifier as the current '`A`' thread, and the specific value is the logical identifier of that '`B`' thread. The second of these values comes from the '`B`' thread with a logical identifier two times the logical identifier of the current '`A`' thread, plus one; the specific value is the logical identifier of this "next" '`B`' thread.

If this process is not yet clear, perhaps figure 4.5 will make it more so.



The value of any variable or system constant in the router expression comes from the direct access the current thread has to itself. In this case, the current thread is `A[1]`. Thus, in the router expression (`B[2*($$i)]`), `$$i=1`. `B[2*($$i)]=B[2*1]` `=B[2]`.

The value of any variable or system constant in the remote expression comes from indeirect access to the thread of the router expression. In this case, the thread of the router expression is `B[2]`. So in `B[2].$$i`, `$$i=2`.

To put it all together, for the current thread, `A[1]`, the value of `B[2*($$i)].$$i=2`.

Figure 4.4: Shows the sources of values in the communication expression.

For this example, the following table shows the values of each computation for each of the threads of ‘`A`’.

	A[0]	A[1]	A[2]
$$i	0	1	2
2*($$i)	0	2	4
B[2*($$i)].$$i	B[0].$$i	B[2].$$i	B[4].$$i
B[2*($$i)].$$i	0	2	4
2*($$i)+1	1	3	5
B[2*($$i)+1].$$i	B[1].$$i	B[3].$$i	B[5].$$i
B[2*($$i)+1].$$i	1	3	5
sum	1	5	9

Computation values for thread communication.

You can try the following exercises to test your comprehension of thread communication between different bundles.

Exercise 4.3: What is wrong with the following code? *(Hint: Look at the communication expression.)*

```
threads A[2];
threads B[3];
A int a,c;
B int b;

int main() {
  A.{
   c = $$i;
   B.{ b = ($$i+1)%($$N); }
   a = B[b].$$i;
 }
 return 0;
}
```

Solution: The router expression is incorrect. ‘b’ is a value directly accessible only to the ‘B’ threads, and since we are inside the execution context of ‘A’, the value between the brackets of ‘B[...]’ should be a value accessible to the ‘A’ threads.

Exercise 4.4: Write a program to compute 9! using thread communication between a bundle of threads and ‘MAIN’. Do this by sending values from your bundle of threads into a storage variable of ‘MAIN’.

Solution:

```
/*
   Computes factorials using threads
   Uses thread communication between bundles
   Sample aCe Program

   Michelle L. McCotter
*/
#include <stdio.aHr>
#define size 10
threads A[size];

int main() {
   int factorial=1,i;

   for(i=1; i<size; i++)
      factorial = factorial * A[i].$$i;
   printf("The factorial of %d, computed via\n"
          "thread communication, is %d.\n",
          size-1,factorial);
```

```
  return 0;
}
```

Program 4.3: Uses thread communication between 'MAIN' and another bundle of threads to compute 9!.

Exercise 4.5: Write a program that creates a bundle of three threads. Then, print the system constants for that bundle without declaring an execution context for that bundle. It may be easiest to print out all system constants for each thread, one thread at a time, rather than a particular system constant for all threads, one thread at a time (as we did in chapter 2).

Solution:

```
/*
   Sample aCe program
   Prints each of the system constants for bundle
      of threads

   Michelle L. McCotter
*/
#include <stdio.aHr>
#define size 3
threads A[size];

int main () {
   int i=0,j=0;

   for (i=0; i<size; i++) {
      printf("\nSystem Constants for Thread"
             " A[%d]\n",A[i].$$i);
      printf("-------------------"
             "-------------\n");
      printf("$$Name\t=  %s\n",A[i].$$Name);
      printf("$$D\t= %2d\n",A[i].$$D);
      printf("$$N\t= %2d\n",A[i].$$N);
      printf("$$L\t= %2d\n",A[i].$$L);
      printf("$$i\t= %2d\n",A[i].$$i);
      printf("$$ii\t= %2d\n",A[i].$$ii);
      printf("$$Nx[0]\t= %2d\n",A[i].$$Nx[0]);
      printf("$$Lx[0]\t= %2d\n",A[i].$$Lx[0]);
      printf("$$ix[0]\t= %2d\n",A[i].$$ix[0]);
  }
```

```
      return 0;
   }
```

Program 4.4: Uses thread communication to print system constants for a bundle of threads.

4.4 Communications Path Descriptions for Communication Between Bundles

In section 4.2, we talked about a communication expression, and we gave examples of the router expression and remote expression for communication *within* a bundle of threads. Now, we would like to talk about the communication expression for communication *between* different bundles.

In the code

```
threads A[1];
threads B[1];
A.{
   int t;
   B int s;
   t = B[0].(s+1);
}
```

the ‘`A`’ and ‘`B`’ threads are single-thread bundles. `B[0]` is the router expression, `(s+1)` is the remote expression that is executed on the threads of ‘`B`’, and ‘`t`’ is the variable in each of the ‘`A`’ threads to which the values received from the ‘`B`’ threads are stored. The router expression (`B[0]`) describes the path between the remote execution context (the ‘`B`’ threads) and the local execution context (the ‘`A`’ threads). If the router expression is the *source* of a value (i.e. a gather or fetch operation like it is here), the remote context computes a value (the value of ‘`s`’ for the thread indicated by `B[0]`, plus one). That value is fetched by the local context from the remote thread that is described by the router expression.

The threads of ‘`B`’ need not be currently active to evaluate the expression `(s+1)`. If a value is to be fetched from an inactive ‘`B`’ thread, that thread is temporarily made active.

Since ‘`A`’ and ‘`B`’ are single-thread bundles, we could write the communication expression as `B.(s+1)` instead of `B[0].(s+1)`, because there is only one ‘`B`’ thread; there is no need to make a distinction (i.e. with the `0` in `B[0]`) about the ‘`B`’ thread from which we want to fetch the value of `(s+1)`.

With multi-thread bundles, we can still write the communication expression without an index. By default, an expression like `B.(s+1)` is evaluated the same as an expression with an index of zero (`B[0].(s+1)`). If we want to refer to threads other than the thread with `$$i=0`, we must include an index in the router expression.

Values may be fetched from existing but not necessarily active threads. If an attempt is made to fetch a value from a non-existing thread, the result is undefined. As mentioned before, if values are to be fetched from inactive threads, those threads are temporarily made active. In general, a gather operation (to be discussed in a later section) activates all threads that will be fetched from, and deactivates threads that will not be fetched from for the duration of the remote context's execution. This happens no matter what the active state was prior to the communication. In the following example, if the conditional expression were eliminated, the router expression would generate invalid addresses for some local threads. Thus a conditional statement is used to ensure that only valid thread addresses are fetched.

```
#define size 16
threads E[size];
threads F[size];
E.{
   int a;
   F.{ int b = $$i*8; }
   if ($$i+2<size)
      a = F[$$i+2].(b+1);
}
```

You should understand communication expressions at this point. If you do not, go back and read sections 4.2 and 4.4 again. You can do any of the exercises presented so far in this chapter to test your comprehension.

4.5 Conditional Expressions

A conditional expression can also affect communications. Only active threads can initiate a '`get`' or '`put`' operation. In the statement

```
B.{
   if (a)
      b = A[c].x;
}
```

only threads of '`B`' for which '`a`' is true actually fetch a value from the storage location '`x`' of a thread of '`A`'. In the statement

```
B.{
   if (a)
      A[c].y = b;
}
```

only active threads of 'B' will store (or 'put') values into threads of 'A'.

Threads of 'A' need not be active to be fetched from ('get') or have their storage modified ('put'). For example, in the statement

```
A.{
   if ($$i == 0) {
      B.{
         if (a)
            A[c].y = b;
      }
   }
}
```

all but one thread of 'A' are made inactive by `if` statement; yet all threads of 'B' (which are active, i.e. for which 'a' is true) will store ('put') values into threads of 'A'. To allow you to see this better, we include an example program that does this.

```
/*
   Sample aCe Program
   Demonstrates active and inactive threads
      in a 'get' operation

   Michelle L. McCotter
*/
#include <stdio.aHr>
threads A[4];
threads B[6];

int main() {
   A int a;
   B int b;

   A.{
      /* initialize all a values to -1 so that we can
         easily see when a values are modified later */
      a = -1;
      /* The Conditional Statement that follows makes all
         but one A thread inactive */
      if ($$i == 2) {
      /* only active threads of A will print out */
         printf("Active Thread:  $$Name = %s, "
                "$$i = %d, a = %d\n",
                $$Name,$$i,a);
         a = $$i;
         B.{
            b = $$i;
            MAIN.{ printf("\n"); }
            /* only active threads of B will print out */
            printf("Active Thread:  $$Name = %s, "
                " $$i = %d, b = %d\n",$$Name,$$i,b);
            A[b].a = $$i;
         }
      }
      MAIN.{ printf("\n"); }
      /* only active threads of A will print out */
      printf("Active Thread:  $$Name = %s, $$i = %d, "
             "a = %d\n",$$Name,$$i,a);
   }
   return 0;
}
```

Program 4.5: Demonstrates modification of data of inactive threads via a ‘`put`’ operation from active threads.

Program 4.4 has the following output.

```
Active Thread:  $$Name = A, $$i = 2, a = -1

Active Thread:  $$Name = B, $$i = 0, b = 0
Active Thread:  $$Name = B, $$i = 1, b = 1
Active Thread:  $$Name = B, $$i = 2, b = 2
Active Thread:  $$Name = B, $$i = 3, b = 3
Active Thread:  $$Name = B, $$i = 4, b = 4
Active Thread:  $$Name = B, $$i = 5, b = 5

Active Thread:  $$Name = A, $$i = 0, a = 0
Active Thread:  $$Name = A, $$i = 1, a = 1
Active Thread:  $$Name = A, $$i = 2, a = 2
Active Thread:  $$Name = A, $$i = 3, a = 3
```

Initially, only one thread of ‘`A`’ prints, showing that only one thread of ‘`A`’ is active. While only one thread of ‘`A`’ is active, all threads of ‘`B`’ are active, shown by all the ‘`B`’ threads printing out their values for ‘`b`’. ‘`B`’ threads then perform ‘put’ operations to modify the values stored in ‘`a`’ of the ‘`A`’ threads. Even though all but one thread of ‘`A`’ are inactive, we see by the later printout of ‘`A`’ threads that all of the ‘`a`’ values for the ‘`A`’ threads were modified. (If they had not been modified, they would have been equal to `-1`.)

4.6 Summary

- A thread can only access its own storage directly; it cannot directly access the storage of any other thread, even if the threads are in the same bundle.

- In aCe, there are two basic communication operations: ‘`get`’ and ‘`put`’.

- A communication expression defines communication between two threads. The expression consists of two parts: the router expression and the remote expression.

- A conditional expression can also affect communications. Only active threads can initiate communication. Other threads need not be active to be fetched from or have their storage modified.

Chapter 5: Path Descriptions

5.1 Definition

Before further discussion of thread communication, we must understand the different types of addressing used in aCe, known as path descriptions. A *path description* is simply a description of the path to a particular thread. In aCe, there are four types of path descriptions: *absolute addressing, universal addressing, relative addressing,* and *reduction addressing.* We will discuss each of these types in detail, one type at a time.

5.2 Absolute Addressing

Absolute addressing assumes that the path description starts at '`MAIN`', and proceeds down the tree to the remote thread. Given

```
threads {{D[2],E[4]} C[1],{F[3]} B[2]} A[3];
```

we have a thread bundles structure looks like figure 5.1.

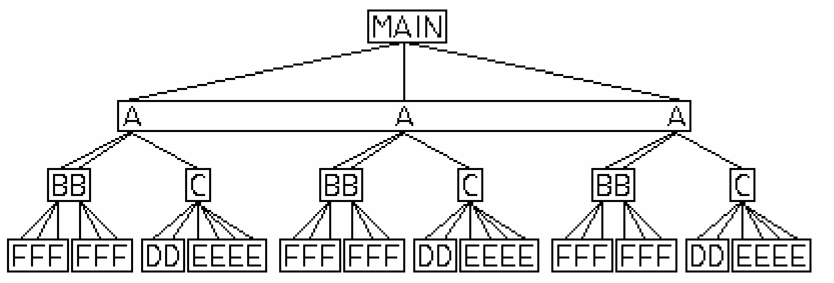


Figure 5.1: Bundles of threads created with the threads declaration
`threads {{D[2],E[4]} C[1],{F[3]} B[2]} A[3];`

For the statement

```
E.{
   int a,x1,x2,x3;
   F int f;
```

```
      a = A[x1].B[x2].F[x3].(f+1);
   }
```

to locate the value of (f+1), we traverse figure 5.1 from 'MAIN' to the x1-th thread of 'A'. From this node, we then proceed to the x2-th thread of 'B'. 'x2' is the logical thread identifier of the 'B' thread, modulo the number of 'B' threads under each 'A' thread. Finally, we end at the x3-th thread of 'F'. Similar to 'x2', 'x3' is the logical thread identifier of the 'F' thread, modulo the number of 'F' threads under each 'B' thread. Note that 'x1', 'x2', and 'x3' have a different value of each thread of 'E' and that the path for each thread of 'E' is to a different thread of 'F'. The following diagram shows the highlighted path for a thread of 'E' where x1=1, x2=0, and x1=2.

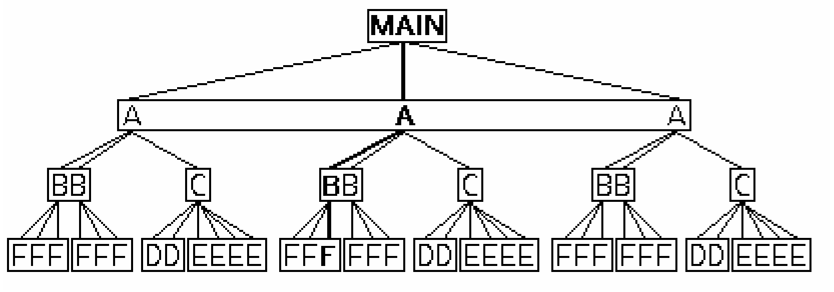


Figure 5.2: The boldface path displays the path indicated by the absolute path description A[1].B[0].F[2].(f+1)

5.3 Universal Addressing

Universal addressing is like absolute addressing but needs not start at a the child of 'MAIN'. In a way, universal addressing initially treats all threads of a bundle as if they were contained in a single one-dimensional bundle; no matter what the local thread is, a programmer can directly refer to another bundle of threads, without having to include the path to that bundle from the local bundle. For example, in the following code, threads of 'E' are children of threads of 'C', which are children of threads of 'A'.

```
threads {{D[2],E[4]} C[1],{F[3]} B[2]} A[3];
B.{
   int x1,x2,b;
   E int e;
   C[x1].E[x2].e = b;
}
```

Figure 5.3 depicts the layout of the bundle of threads.

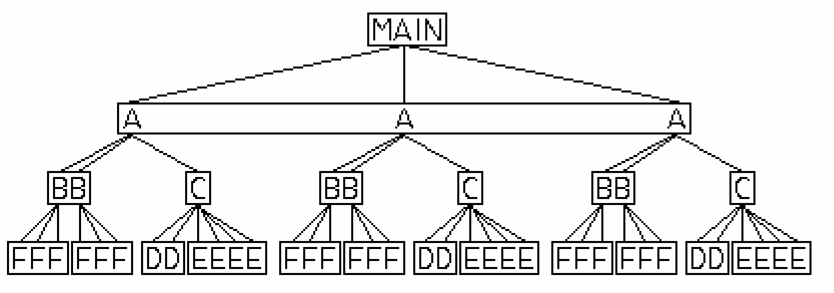


Figure 5.3: Thread bundles declared by
`threads {{D[2],E[4]} C[1],{F[3]} B[2]} A[3];`

Inside the execution context of ‘`B`’, data of threads of ‘`B`’ need to be sent to threads of ‘`E`’, which are not children of that thread of ‘`B`’. However, each thread of ‘`B`’ contains the value of the thread identifier, ‘`x1`’, of the thread of ‘`C`’ that is the parent of the thread of ‘`E`’ (with identifier ‘`x2`’) where the data is to be sent. Thus, the value of ‘`x1`’ is used as the index into the one-dimensional array of threads of ‘`C`’ as a starting point of the path. Even though the local bundle is the ‘`B`’ threads, we can directly refer to a child bundle of the ‘`C`’ threads because universal addressing allows us to do this. The important part of universal addressing is that the index value of the first bundle of threads specified (i.e. the ‘`C`’ threads) comes from the `$$i` values across the entire bundle of ‘`C`’ threads. The rest of the path (after `C[x1]`) is treated as if it were absolute addressing, and the indices for these bundles are the relative `$$i` values within the specific sub-bundle.

With `x1=1` and `x2=2`, the communication expression `C[x1].E[x2].e` refers to the value of ‘`e`’ of the ‘`E`’ thread indicated by the boldface path in figure 5.4.

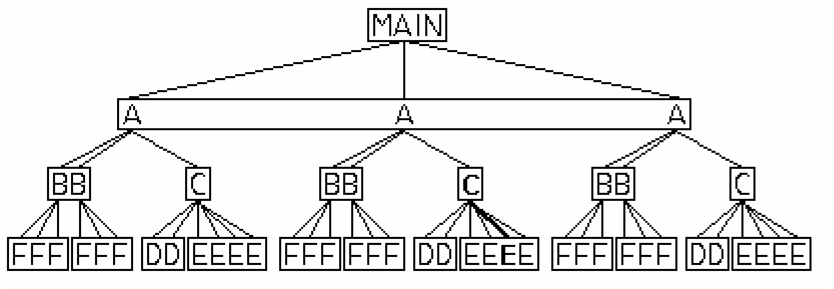


Figure 5.4: The boldface path displays the path indicated by the universal path description `C[0].E[2].e`.

Note that the universal path description that we are using as an example does not start at a child of ‘`MAIN`’ (an ‘`A`’ thread). Instead, it starts with a ‘`C`’ thread. The difference between universal addressing and absolute addressing lies in this idea: absolute addressing must always start at a child of ‘`MAIN`’.

We include the following program as an example of universal addressing. The structure of the threads in the program corresponds to the structure found in figure 5.4.

```
/*
   Sample aCe Program
   Demonstrates universal addressing

   Michelle L. McCotter
*/
#include <stdio.aHr>
threads {{D[2],E[4]} C[1],{F[4]} B[2]} A[3];

int main() {
   B.{
      /* do everything in this execution context for each
         thread of B */
      int x1,x2,sizeC,sizeE;
      E int e = -1;  /* initialize to -1 so we can see
                  which ones do not get modified */
      sizeC = C.$$N;
      sizeE = E.$$N/C.$$N;
      x1 = ($$i)%(sizeC);
      x2 = ($$i)%(sizeE);
      C[x1].E[x2].e = C[x1].E[x2].$$i;  /* assign a $$i of
                           E to the corre-
                           sponding e of E*/
      E.{ printf("$$i = %2d, C[%d].E[%d].e = %2d\n",
               $$i,$$i/B.sizeE,$$i%(B.sizeE),e); }
                  /* print out values of e for each thread
                  of E */
   }
   return 0;
}
```

Program 5.1: Assigns `$$i` of the current '`B`' thread to the '`e`' of an '`E`' thread, using universal addressing. Prints out values of '`e`' for every '`E`' thread.

Program 5.1 prints the following output.

```
$$i =  0, C[0].E[0].e =  0
$$i =  1, C[0].E[1].e = -1
$$i =  2, C[0].E[2].e = -1
$$i =  3, C[0].E[3].e =  3
$$i =  4, C[1].E[0].e =  4
$$i =  5, C[1].E[1].e =  5
$$i =  6, C[1].E[2].e = -1
$$i =  7, C[1].E[3].e = -1
$$i =  8, C[2].E[0].e = -1
```

```
$$i =  9, C[2].E[1].e =  9
$$i = 10, C[2].E[2].e = 10
$$i = 11, C[2].E[3].e = -1
```

Program 5.1 should be a good example of how universal addressing works. Six of the '`e`' values have been modified (i.e. they are no longer `-1`). This is as we would expect, because the universal addressing occurs within the execution context of '`B`', and there are six '`B`' threads. There is one minor detail to note: we use a modular operation when computing '`x1`' and '`x2`', because the compiler would give an error if there were an attempt to fetch data from threads that did not exist. Using the mod operator (so long as it is used correctly) ensures that all of our universal address are valid.

5.4 Relative Addressing

Relative addressing is a bit more difficult to explain than universal or absolute addressing. Relative addresses are computed with respect to the local thread's address and/or the address of one or more of its parents. A relative router expression is prefixed with a period (`.`) and is assumed to start at the local thread's bundle, or one of its parents (meaning a parent thread, a grandparent thread, etc.) Using the same threads structure as with our previous examples, the following code includes an example of a path from a thread of '`E`' to a thread of '`F`'.

```
threads {{D[2],E[4]} C[1],{F[3]} B[2]} A[3];
E.{
   int e,x1,x2,x3;
   F int f;
   e = .C.A[x1].B[x2].F[x3].(f+1);
}
```

Figure 5.5 shows the highlighted path (using relative addressing) to fetch a remote value from a thread of '`B`' (with absolute path description `A[0].B[1].F[1]`) to a local thread of '`E`' (with absolute path description `A[0].C[0].E[0]`), where `x1=0`, `x2=1` and `x3=1`.

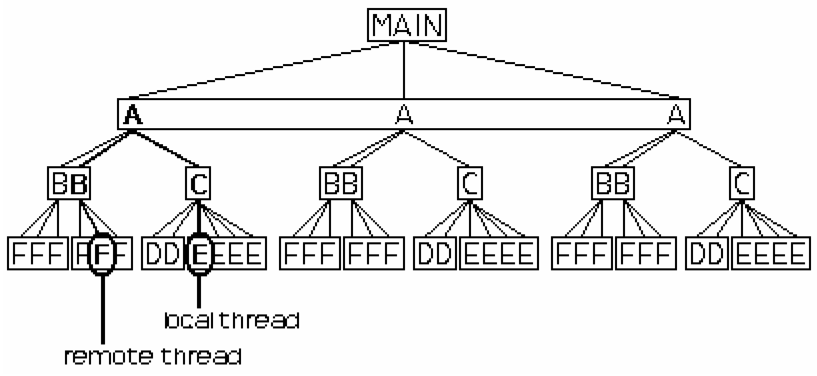


Figure 5.5: Relative path description for `.C.A[0].B[1].F[1].(f+1)` from a local thread with absolute address `A[0].C[0].E[0]`.

In this example, the local thread is shown in figure 5.5, and the relative path description is `.C.A[x1].B[x2].F[x3].(f+1)`. When determining the absolute address for relative router expression, we speak in terms of bundle levels. Since we have chosen to depict bundles of threads far in a tree-like structure, the idea of levels should be straightforward. '`MAIN`' is at top, '`A`' is above '`B`' and '`C`'. '`B`' and '`C`' are two levels below '`MAIN`', one level below '`A`', etc. A given bundle of threads, in general, is one level below its parent bundle.

Relative path calculations are done modulo the size of the dimensions of the sub-bundle, thus toroidally connecting the sub-bundles of threads. As we describe relative addressing in detail, you will come to see how this is true. The actual remote thread referred to by a relative router expression depends on the specific local thread, as well as the values inside the brackets of each thread bundle (i.e. '`x1`', '`x2`', and '`x3`'). When '`x1`', '`x2`', and '`x3`' are not zero, they serve in shifting the absolute address of the remote thread; when they are zero, there is no shift. We discuss one portion of the relative router expression at a time.

1. `F[x3]`—This is just the address of the remote thread within the '`F`' bundle. It has nothing to do with determining which '`F`' bundle is referred to (other portions of the relative router expression do that, and they will be discussed momentarily). Since the '`F`' bundles are at the same level as the '`E`' bundles, we determine the corresponding absolute address for the '`F`' portion of the relative address as follows:
   a) Determine `$$i` for the local thread. In this case, there are twelve '`E`' threads, so `$$i` can have values from the set `{0,1,…,11}`. For the local thread indicated in figure 5.5, `$$i=0`.
   b) Determine `$$N` for the bundle containing the remote thread. With our example, `$$N=3`.
   c) Using `$$i` from part a) and `$$N` from part b), determine the absolute address of the '`F`' thread by computing the value of

F[(x3+$$i)%$$N]. In this case, F[(1+0)%3]=F[1]. At this point, F[1] could refer to any of the threads circled in figure 5.6.

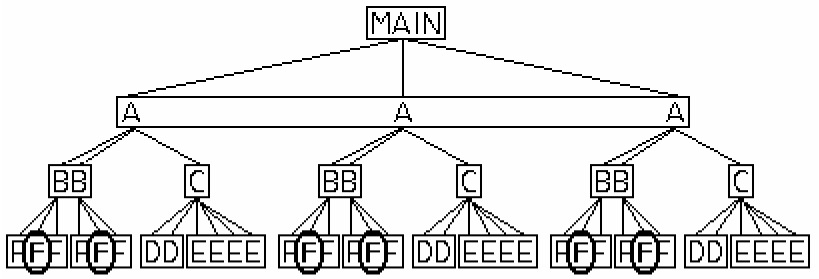


Figure 5.6: Possible 'F' threads referred to by F[x3]. The specific 'F' thread will be determined as we determine possible 'B' threads and a distinct 'A' thread.

2. B[x2]—This is the relative address of the parent thread of the 'F' thread indicated by F[x3]. By "relative" address, we mean the address of the 'B' thread within *one* sub-bundle of 'B' threads, and not the address with respect to all 'B' threads. So since there are only two 'B' threads in any one bundle, B[x2] refers to either B[0] or B[1], no matter what value 'x2' is. Since the 'B' threads are one level above the 'E' threads, we no longer use $$i of the 'E' thread. Rather, we use the $$i of the parent of the 'E' thread (the local thread). To determine the absolute address of the 'B' portion of the relative address, do as follows:
   a) Determine the $$i of the parent thread of the local thread, one level up from the local thread (i.e. the $$i of the 'B' threads). In this case, $$i=0.
   b) Determine $$N for the parent bundle of the remote thread one level up from the remote thread. In this case, $$N=2.
   c) Using $$i from part a) and $$N from part b), determine the absolute address of the 'B' thread by computing the value of B[(x2+$$i)%$$N]. In this case, B[(1+0)%2]=B[1]. Taking B[1] from what we just determined, and F[1] from what we determined in part 1, B[1].F[1] could refer to any of the threads circled in figure 5.7.

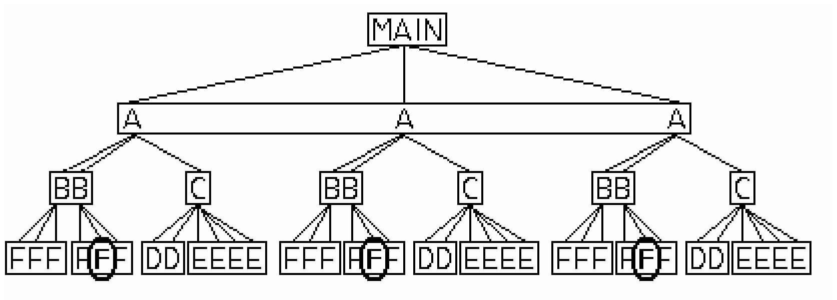


Figure 5.7: Possible 'F' threads referred to by `B.[x2].F[x3]`. The specific 'F' thread will be determined as we determine possible a distinct 'A' thread.

3. `A[x1]`—This is the relative address of the parent thread of the 'B' thread indicated by `B[x2]`. As we explained in part 2, this means that `A[x1]` refers to either `A[0]`, `A[1]`, or `A[2]`. Since the 'A' threads are two levels above the 'E' threads, we use `$$i` of the 'A' thread, the `$$i` of the parent of the 'E' thread two levels above the 'E' thread. To determine the absolute address of the 'A' portion of the relative address, do as follows:
   a) Determine the `$$i` of the parent thread of the local thread, two levels up from the local thread (i.e. the `$$i` of the 'A' threads). In this case, `$$i=0`.
   b) Determine `$$N` for the parent bundle of the remote thread one level up from the remote thread. In this case, `$$N=3`.
   c) Using `$$i` from part a) and `$$N` from part b) Determine the absolute address of the 'A' thread by computing the value of `A[(x1+$$i)%$$N]`. In this case, `A[(0+0)%3]=A[0]`. Taking `A[0]` from what we just determined, `B[1]` from part 2, and `F[1]` from part 1, `A[0]. B[1].F[1]` refers to the remote thread circled in figure 5.8.

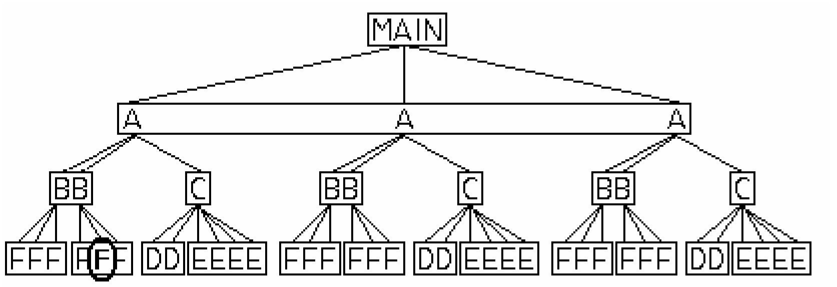


Figure 5.8: The 'F' thread referred to by `A[x1].B[x2].F[x3]`, for the local thread with the absolute address of `A[0].C[0].E[0]`,

4. .C—The parent of the local thread is a 'C' thread, indicated by '.C'. Parents of the local thread do not need indices if they are specifying the path up the bundle hierarchy toward 'MAIN'. Actually, it is unnecessary to specify the bundle 'C' in the router expression since 'A' is a grandparent, and thus also considered a parent, of 'E'. Instead, the expression could be written as '.A[x1].B[x2].F[x3]'.

Thus for the local thread of 'E' with absolute path description A[0].C[0].E[0], the remote thread of 'F' denoted by A[0].B[1].F[1] has absolute path description A[0].B[1].F[1]. It may seem that we did a lot of work for an address that turned out to be so simple. So let us consider a more complicated example. We will use the same thread bundles structure, with relative router expression .A[5].B[3].F[7] and absolute local address A[1].C[0].E[2]. Although we choose not to include all our work as we did in the previous example, by following the same pattern as we used previously, we find that the absolute address of the remote thread is A[0].B[0].F[1]. We show this visually in figure 5.9.

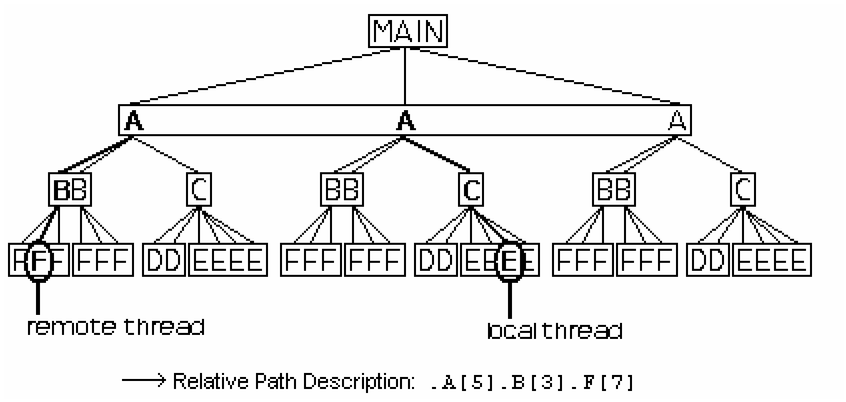


Figure 5.9: Relative path description for .A[5].B[3].F[7].(f+1) from a local thread with absolute address A[1].C[0].E[2].

Exercise 5.1: For the local thread with absolute address A[1].C[0].E[2], go through each portion of the relative address .A[5].B[3].F[7] (following the steps we outlined previously to compute the absolute address of the remote thread (A[0].B[0].F[1].)

There are a two other things to keep in mind with relative addressing. For one, the remote thread does not have to be at the same level as the local thread. In the preceding example, we could have used the relative router expression

.A[5].B[3] instead. All absolute addresses would have been calculated the same. Second, the remote thread cannot be at a level lower than the level of the local thread. If you try to write a program that uses relative addressing to reference a thread at a lower level than the level of the local thread, the compiler will give you a linking error.

Relative addressing can seem quite complicated. If you do exercise 5.2 by adhering to the general ideas that follow, you will be able to see the complexity of relative addressing, and you will also be able to test how well you understand the concept.

- The relative router expression can be divided into different threads. Consider one thread at a time. For clarity, let us refer to these different divisions as destination threads. (i.e. We can divide the relative router expression .A[5].B[3], into A[5] and B[3].)
- When we calculate the absolute address for each thread in the router expression (i.e. each destination thread), we need to determine three values:
  a) 'N'—the number of threads in the destination thread bundle (i.e. When we are computing the address of the 'B' thread, B[3] is our destination bundle, there are three threads in the 'B' bundle.)
  b) 'x'—the displacement value, which is indicated by the number in brackets (i.e. With B[3], the displacement value is 3.)
  c) '$$i'—the thread identifier of the local thread or one of the parents of the local thread, whichever thread is at the same level as the level of the destination thread bundle.
- The index of the absolute address for each destination bundle is determined by: `index=(x+$$i)%N`.

Exercise 5.2: Using the same threads structure (declared by `threads {{D[2],E[4]} C[1],{F[3]} B[2]} A[3];`), fill in the table below. Refer to figure 5.10 for help.

Local Thread	Remote Thread	Remote Thread
(Abs Address)	(Rel Address)	(Abs Address)
A[0].C[0].E[0]	.A[1].B[1].F[1]	a)
A[2].C[0].E[2]	.A[2].B[5].F[8]	b)
A[2].C[0].E[2]	.A[5].B[2]	c)
A[1].C[0].E[2]	d)	A[2].B[1]
A[0].B[1]	.A[2].C[0].E[1]	e)
f)	.A[5].C[6].E[4]	A[2].C[0].E[3]
A[2].C[0]	g)	A[1].B[1]
A[1].B[0]	h)	undefined

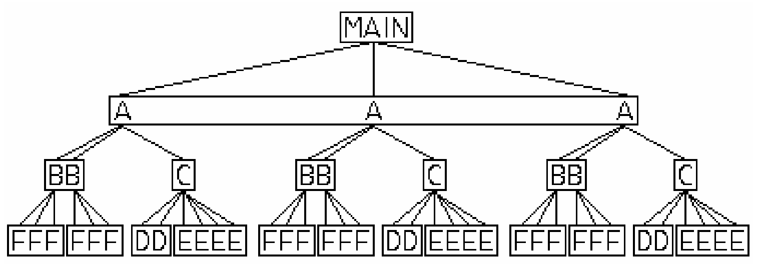


Figure 5.10: Thread bundles created by the declaration `threads {{D[2],E[4]}C[1],{F[3]}B[2]}A[3];`

Solution:

a) `A[1].B[1].F[1]`
b) `A[1].B[1].F[0]`
c) `A[5].B[0]`
d) `.A[1+3m].B[2n]`, where `m,n` ∈ **N**
e) `undefined`
f) `A[0].B[1].F[0]`
g) `A[2+3m].B[1+2n]` , where `m,n` ∈ **N**
h) any relative address to an '`E`' thread

Exercise 5.3: This exercise will help you understand how different relative addresses affect the same local thread. Let the numbered instructions guide you as you complete each part of the table that follows.

1) After filling in parts a) and b), make a prediction about part c). Test your prediction. Were you right?
2) Now fill in part d). Notice how many threads there are in each sub-bundle of the '`F`' threads. Make a prediction for part e) and test it.
3) Fill in part f). What will the absolute address be for part g)? *(Hint: Patterns with the '`F`' threads correspond to patterns with the '`B`' threads.)*
4) Using the trends you've discovered with the previous portions of this exercise, make a guess for part h). Was your guess correct?
5) For part i), what is your guess for the absolute address? *(Hint: Patterns with the '`A`' threads correspond to patterns with the '`F`' threads and '`B`' threads.)*

Local Thread	Remote Thread	Remote Thread
(Abs Address)	(Rel Address)	(Abs Address)
A[0].C[0].E[0]	.A[0].B[0].F[0]	a)
A[0].C[0].E[0]	.A[0].B[0].F[1]	b)
A[0].C[0].E[0]	.A[0].B[0].F[2]	c)
A[0].C[0].E[0]	.A[0].B[0].F[3]	d)
A[0].C[0].E[0]	.A[0].B[0].F[5]	e)
A[0].C[0].E[0]	.A[0].B[1].F[0]	f)
A[0].C[0].E[0]	.A[0].B[2].F[0]	g)
A[0].C[0].E[0]	.A[0].B[5].F[7]	h)
A[0].C[0].E[0]	.A[5].B[3].F[8]	i)

Solution: a) `A[0].B[0].F[0]`
b) `A[0].B[0].F[1]`
c) `A[0].B[0].F[2]`
d) `A[0].B[0].F[0]`
e) `A[0].B[0].F[2]`
f) `A[0].B[1].F[0]`
g) `A[0].B[0].F[0]`
h) `A[0].B[1].F[1]`
i) `A[2].B[1].F[2]`

Exercise 5.4: This exercise will help you understand the effect of the same relative address on different local threads. Use the following threads declaration.

```
threads {{D[1],E[5]} C[1],{F[3]} B[2]} A[3];
```

to complete the following table. Figure 5.11 is included to assist you.

1) After filling in parts a) and b), make a prediction about part c). Test your prediction. Were you right?
2) Fill in parts d) and e). Predict the answers for parts f) and g). Were your predictions correct?
3) Complete h) and i). What should part j) be? Test your prediction.
4) Finally, fill in parts k) and l). Guess the solution for part m), then check your answer.

Local Thread	Remote Thread	Remote Thread
(Abs Address)	(Rel Address)	(Abs Address)
A[0].B[0].F[0]	.A[0].C[0].E[0]	a)
A[0].B[0].F[1]	.A[0].C[0].E[0]	b)
A[0].B[0].F[2]	.A[0].C[0].E[0]	c)
A[0].B[1].F[2]	.A[0].C[0].E[0]	d)
A[1].B[0].F[0]	.A[0].C[0].E[0]	e)
A[1].B[1].F[1]	.A[0].C[0].E[0]	f)
A[2].B[0].F[0]	.A[0].C[0].E[0]	g)
A[0].B[0].F[0]	.A[0].C[0].E[2]	h)
A[0].B[0].F[1]	.A[0].C[0].E[2]	i)
A[0].B[0].F[2]	.A[0].C[0].E[2]	j)
A[0].B[1].F[0]	.A[1].C[0].E[0]	k)
A[0].B[1].F[1]	.A[1].C[0].E[0]	l)
A[0].B[1].F[2]	.A[1].C[0].E[0]	m)

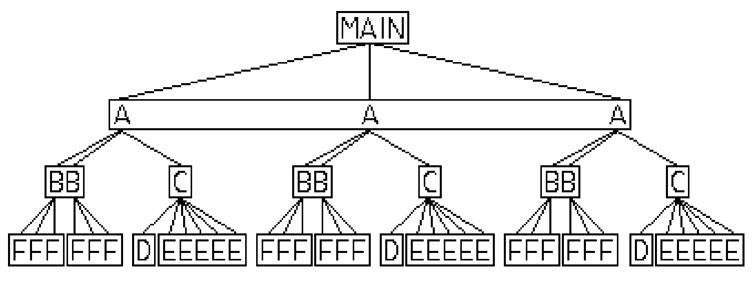


Figure 5.11: Thread bundles created by the declaration `threads{{D[1],E[5]}C[1],{F[3]}B[2]}A[3]`

Solution:
a) `A[0].C[0].E[0]`
b) `A[0].C[0].E[1]`
c) `A[0].C[0].E[2]`
d) `A[0].C[0].E[0]`
e) `A[1].C[0].E[1]`
f) `A[1].C[0].E[2]`
g) `A[2].C[0].E[2]`
h) `A[0].C[0].E[2]`
i) `A[0].C[0].E[3]`
j) `A[0].C[0].E[4]`
k) `A[2].C[0].E[1]`
l) `A[2].C[0].E[2]`
m) `A[2].C[0].E[3]`

5.5 Reduction Addressing

Reduction addressing can be viewed as all-to-all communications. Every thread of the source bundle sends a value to every thread of the destination bundle. This mode of addressing is useful as well as efficient for performing global reductions (to be defined in a few paragraphs). Reduction addressing is also efficient if a value is needed from the threads of a bundle by the threads of another if the value is the same for all threads. Reduction addressing is performed by designating the name of the bundle from which a value is to be reduced.

Observe the following code.

```
threads H[100];
threads B[300][400];
H.{
   int a;
   B int b=z;
   a = B.b;
}
```

In this code, note the expression ‘a=B.b’. Although ‘B’ is a two-dimensional thread bundle, the path from ‘B’ is designated by its name alone and no indices. The expression will get the value of ‘b’ of the active thread of ‘B’ with smallest logical identifier value and distribute it to ‘a’ of all the active threads of ‘H’. The next example is functionally equivalent to the previous, except that the values of ‘b’ are being sent to ‘a’ rather than being fetched by ‘H’ and assigned to ‘a’.

```
threads H[100];
threads B[300][400];
B.{
   H int a;
   int b=z;
   H.a = b;
}
```

Exercise 5.5: Using any editor you choose, enter the following program.

```
/*
     Sample aCe Program
     Demonstrates the use of reduction
   addressing

   Michelle L. McCotter
*/
#include <stdio.aHr>
```

```
threads A[3];
threads B[3][2];

int main() {
   A int a = 0;
   B.{
      if ($$i >= 0)
         A.a = $$i;
   }
   A.{ printf("a = %d\n",a); }
   return 0;
}
```

Program 5.2: Demonstrates the use of reduction addressing.

Compile and execute the program. What do you notice about the values of 'a' that print out? Change the value in the if statement to something larger than zero, and execute the program. What do you notice?

Solution: Each value of 'a' is equal to the value of the smallest thread identifier of the active 'B' threads.

A *global reduction* occurs when multiple data are combined using a combination operator (as in +=, -=, &=, |=, etc.). Global reductions can be performed by this addressing mode by replacing the '=' with operations such as '+='. In this case, the values of 'b' of all active threads of 'B' are summed and added to 'a' of all threads of 'H'.

```
threads H[100];
threads B[300][400];
B.{
   H int a;
   int b=z;
   H.a += b;
}
```

Global reductions are limited to the following operations: addition (+=), subtraction (-=), and (&=), or (|=), exclusive or (^=), minimum (<?=), and maximum (>?=). You should keep in mind that the 'and', 'or', and 'exclusive or' operations result from binary bitwise functions. For example, the assignment '&=' is determined by (a&b&c&d ...). '&' is the binary bitwise 'and' function (i.e. 1100&1010=1000). The 'or' and 'exclusive or' functions are similar.

Exercise 5.6: Experiment with various global reductions, making sure you understand what is going on in your programs.

Exercise 5.7: As a review of addressing types, can you identify the type of addressing for each of the following addresses?

a. `threads {{C[2]}B[3]}A[4];`
   `A[x1].B[x2].C[x3].c`

b. `threads {{C[2]}B[3]}A[4];`
   `C.c`

c. `threads {D[2],{C[2]}B[3]}A[4];`
   `.B[x1].C[x2].c`

d. `threads {{E[2],D[2]}C[2]}A[1];`
   `E.$$i`

e. `threads {{E[2],D[2]}C[2]}A[1];`
   `C[x1].E[x2].e`

f. `threads {{C[2]}B[3]}A[4];`
   `.C.c`

g. `threads {{C[2]}B[3]}A[4][3];`
   `A[3][2].$$ix[0]`

h. `threads {{C[2]}B[3]}A[4][3];`
   `C[x1].c`

Solution:

a. absolute
b. reduction
c. relative
d. reduction
e. universal
f. relative
g. absolute
h. universal

5.6 Summary

- A path description is a description of the path to a particular remote thread. There are four types of path descriptions: absolute, universal, relative, and reduction.

- Absolute addressing assumes that the path starts at '`MAIN`' and proceeds down the thread bundles tree to the remote thread.

- Universal addressing is like absolute addressing, but needs not start at a child of ‘`MAIN`’.

- Relative addresses are computed with respect to the local thread’s address and the address of one or more of its parents. A relative router expression is prefixed with a period (`.`) and is assumed to start at the local thread’s bundle, or at one of the local thread’s parent bundles.

- Reduction addressing can be viewed as all-to-all communications. Every thread of the source bundle sends a value to every thread of the destination bundle.

- A *global reduction* occurs when multiple data are combined using a combination operator (as in `+=`, `-=`, `&&`, `||`, etc.).

- A scatter operation sends data to the threads of a remote bundle. This happens when the router expression is on the left side of an assignment statement.

- A gather operation fetches data from the threads of a remote bundle. This occurs when the router expression is on the right side of an assignment statement.

- In communications operations, computing the identifiers of remote threads takes a fair amount of time. aCe allows a programmer to define a path description as a variable that is computed at run time. A path is computed once at its assignment statement, and it is used (as many times as necessary) by placing a ‘@’ symbol before the name of the path (each time the path is used).

Chapter 6: More on Thread Communication

6.1 Scatter and Gather Operations

Now that we have discussed different types of addressing, we use a variety of path descriptions in more examples of thread communication. Here, we discuss scatter and gather operations.

Data is fetched from or sent to the threads of a remote bundle depending on whether the router expression is on the right or left side of an assignment. If the router expression is on the right side of an assignment, it is a *gather* from the remote threads. If the expression is on the left side of an assignment, it is a *scatter* to the remote threads.

The following is an example of a *gather operation.*

```
threads {{D[1],E[5]} C[1],{F[3]} B[2]} A[3];
E.{
   int a,x,y;
   B int b;
   a = .C.A[x].B[y].(b+1);
}
```

The value of '`b+1`' of the remote thread of '`B`' is assigned to '`a`' of the current thread of '`E`'. Reversing the sides of an assignment makes it a *scatter operation.*

```
threads {{D[1],E[5]} C[1],{F[3]} B[2]} A[3];
E.{
   int a,x,y;
   B int b;
   .C.A[x].B[y].(b+1) = a;
}
```

The value of '`a`' in '`E`' is sent to the specified remote thread '`b+1`' of '`B`'. If a scatter add operation such as

```
threads {{D[1],E[5]} C[1],{F[3]} B[2]} A[3];
E.{
   int a,x,y;
   B int b;
   .C.A[x].B[y].(b+1) += a;
}
```

is performed, and multiple values are sent to the same thread, they are summed together. Since more than one thread can send a value to the same destination thread, the data will either have to be combined, or some will be lost. Therefore, when data is sent to another thread, there are several options for combination into the remote location, such as addition (+=), subtraction (-=), and (&=), or (|=), exclusive or (^=), minimum (?<=), and maximum (?>=).

No matter what type of addressing is used, a scatter operation returns a flag to the expression in which it is contained, rather than a value. This flag indicates whether or not the value being sent actually was received at the destination thread. For example, in this code

```
threads {{D[1],E[5]} C[1],{F[3]} B[2]} A[3];
E.{
   int a,x,y;
   B int b;
   flag = (.C.A[x].B[y].(b+1) = a);
}
```

the values of ‘a’ in the bundle ‘E’ are being sent to the variables ‘b’ in bundle ‘B’. The value of the flag, after the values have been sent, is TRUE (1) if the specific value of ‘a’ actually reached the requested instance of ‘b’, and FALSE (0) if it did not. There are two reasons a value may not reach its destination:

1. the address of the destination thread is not valid, or
2. the scatter operation is ‘=’.

If the scatter operation is ‘=’, at most one value will be received by any destination thread. Therefore, only one of the source threads that send to the same destination thread will have its value received and its receive flag set to TRUE.

Here is a sample program to show you how the flag works.

```
/*
   Sample aCe Program
   Demonstrates the flag in a scatter operation

   Michelle L. McCotter
*/
#include <stdio.aHr>
#define sizeA 3
#define sizeB 2
#define sizeC 1
#define sizeD 2
#define sizeE 4
#define sizeF 3
threads {{D[sizeD],E[sizeE]} C[sizeC],{F[sizeF]} B[sizeB]}
        A[sizeA];

int main() {
   F int f;

   E.{
      int VAR,x1,x2,x3,X1abs,X2abs,X3abs,addr,flag;
      int temp;  /* temp is used for temporary storage
                 throughout program */
      VAR = $$i;
      x1 = 0;
      x2 = 0;
      x3 = 0;
      addr = .A[x1].B[x2].F[x3].$$i;
      F.{ f = -1; }
      flag = (.A[x1].B[x2].F[x3].f = VAR);
      if ($$i == 0) printf("$$i Rel Address     flag  VAR”
                     "f  addr  Abs Address\n"
                        "----------------------------"
                     "--------------------------\n",
                           x1,x2,x3,x1,x2,x3);
      X3abs = (addr)%(sizeF); /* relative thread identifier
                     of F thread (w/in F bundle
                     under each A thread) */
      temp  = (addr)/(sizeF);
      X2abs = (temp)%(sizeB); /* thread identifier of the
                        B thread */
      temp  = (temp)/(sizeB);
      X1abs = (temp)%(sizeA);
      printf("%2d  .A[%d].B[%d].F[%d] %3d%5d%4d%4d   "
             " A[%d].B[%d].F[%d]\n",
             $$i,x1,x2,x3,flag,VAR,
             .A[x1].B[x2].F[x3].f,addr,
             X1abs,X2abs,X3abs);
   }
   MAIN.{ printf("\n"); }
   F.{
      MAIN.{ printf("\nB threads\n---------\n"); }
      printf("$$i = %2d  f = %3d\n",$$i,f);
   }
   return 0;
}
```

Program 6.1: Demonstrates the flag in a scatter operation.

Program 6.1 prints out the following output.

```
$$i Rel Address      flag  VAR  f  addr  Abs Address
------------------------------------------------------
 0  .A[0].B[0].F[0]   1     0   0   0    A[0].B[0].F[0]
 1  .A[0].B[0].F[0]   1     1   1   1    A[0].B[0].F[1]
 2  .A[0].B[0].F[0]   1     2   2   2    A[0].B[0].F[2]
 3  .A[0].B[0].F[0]   0     3   0   0    A[0].B[0].F[0]
 4  .A[0].B[0].F[0]   1     4   4  10    A[1].B[1].F[1]
 5  .A[0].B[0].F[0]   1     5   5  11    A[1].B[1].F[2]
 6  .A[0].B[0].F[0]   1     6   6   9    A[1].B[1].F[0]
 7  .A[0].B[0].F[0]   0     7   4  10    A[1].B[1].F[1]
 8  .A[0].B[0].F[0]   1     8   8  14    A[2].B[0].F[2]
 9  .A[0].B[0].F[0]   1     9   9  12    A[2].B[0].F[0]
10  .A[0].B[0].F[0]   1    10  10  13    A[2].B[0].F[1]
11  .A[0].B[0].F[0]   0    11   8  14    A[2].B[0].F[2]
```

```
B threads
---------
$$i =  0  f =   0
$$i =  1  f =   1
$$i =  2  f =   2
$$i =  3  f =  -1
$$i =  4  f =  -1
$$i =  5  f =  -1
$$i =  6  f =  -1
$$i =  7  f =  -1
$$i =  8  f =  -1
$$i =  9  f =   6
$$i = 10  f =   4
$$i = 11  f =   5
$$i = 12  f =   9
$$i = 13  f =  10
$$i = 14  f =   8
$$i = 15  f =  -1
$$i = 16  f =  -1
$$i = 17  f =  -1
```

The first table lists several values for each ‘E’ thread. It lists the relative address and corresponding absolute address for each thread, so that you can see which absolute addresses correspond to the relative address for each thread. The ‘addr’ value listed is simply the $$i of the ‘F’ threads corresponding to the absolute address. The table also lists values for ‘flag’, ‘VAR’, and ‘f’. ‘VAR’ is the value sent from each ‘E’ thread to the ‘f’ of some ‘F’ thread, and ‘f’ is the value each ‘F’ thread received from an ‘E’ thread. If flag=1, then VAR=f, and if flag=0, then VAR≠f.

The second table lists the values of ‘f’ for each ‘F’ thread. In program 6.1, we initialize each ‘f’ with a value of -1, so that we can clearly see whether or not this value is modified with the scatter operation. The individual threads that have modified values for ‘`f`’ correspond to the variable ‘`addr`’ in the first table.

Exercise 6.1: Study the output of program 6.1. Can you predict the flag values for each thread of ‘`E`’, based on the relative addresses? (Hint: What happens when the remote addresses of two different threads refer to the same thread? In other words, what happens when two local threads attempt to write to the same remote thread?)

Solution: If two or more local threads attempt to send values to the same remote thread, then the remote thread will receive the value from the local thread with the smallest `$$i`. The flag value for that smallest `$$i` will be `1` (since the value was received), and it will be `0` for the other local threads (that tried to send a value to the same remote thread).

Exercise 6.2: If, in program 4.5, `sizeF` was `2` instead of `3`, and the relative address for each thread were `.A[0].B[0].F[0]`, which threads of ‘`E`’ would have `flag=1`? Which would have `flag=0`? We include a diagram to help you see the structure of the bundles of threads. Remember that the method of addressing used here is relative addressing.

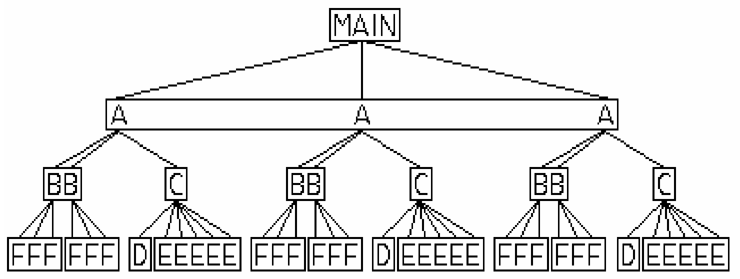


Figure 6.1: Bundles of threads created by the declaration
```
threads {{D[1],E[5]} C[1],
        {F[3]} B[2]} A[3];
```

If the relative address for each thread were `.A[1].B[1].F[1]`, which threads of ‘`E`’ would have `flag=1`? Which would have `flag=0`? What if the relative address were `.A[0].B[0].F[1]`?

Solution: No matter what relative address is used, as long as it is of the form `.A[x1].B[x2].F[x3]`, the `flag` values are as follows.

```
$$i  flag
-----------
 0    1
 1    1
 2    0
 3    0
 4    1
 5    1
 6    0
 7    0
 8    1
 9    1
10    0
11    0
```

Can you explain why changing the values of '`x1`', '`x2`', or '`x3`' in the relative address does not affect the flag values? *(Hint: If you change the x-values, how does this change the remote threads?)*

6.2 Pre-computed Paths

A fair amount of time spent in a communications operation may be spent computing the identifiers of the remote threads involved in the communications. The ability to define a path description as a variable that is computed at run time allows a path description to be pre-computed and optimized once, and yet used many times. This minimizes the cost of computing a path descriptor, especially when the descriptor is used multiple times. A path is declared in the following manner.

```
threads H[4];
threads {B[5]} C[5];
H.{
   path(B) toB;
   int a0,a1,a2,x,y;
   B int b0,b1,b2;
   toB=C[x].B[y].;
   @toB.b0=a0;
   @toB.b1=a1;
   @toB.b2=a2;
}
```

The path ‘`toB`’ is a path from the local context of ‘`H`’ to the remote context of ‘`B`’. It is created by using the keyword ‘`path`’. Inside the parentheses is the name of the bundle of threads for which the path will be used. The path is given a value with an assignment statement. The address in the assignment statement must be terminated by a period (`.`), or a compiler error will result. Each time ‘`toB`’ is used, the programmer must place an ‘@’ symbol before the name of the path. Note in the example code that ‘`toB`’ is computed once (for each thread of ‘`H`’) but is used in three communications operations.

Many engineering applications require the computation of the sum of nearest neighbors. The following program uses four different pre-computed paths, named ‘`North`’, ‘`South`’, ‘`East`’, and ‘`West`’, to compute the sum of the nearest neighbors for each PE in a two-dimensional array of processing elements.

```
/*
   aCe program
   Sum of nearest neighbors using pre-computed paths

   Maurice F. Aburdene
   John E. Dorband
*/
#include <stdio.aHr>
#define size 8
threads A[size][size];

int main () {
   A.{
      path(A) North, South, East, West ; /* define paths */
      int a,b;
      a = $$i;

      /* Computing the paths */
      /* Relative addressing is wrap-around, so the
         "East" thread of the last thread in a row is
         actually the first thread in that same row.  There
         is a similar pattern for North, South, and West.
      */

      North = .A[-1][ 0]. ;
      South = .A[ 1][ 0]. ;
      East  = .A[ 0][ 1]. ;
      West  = .A[ 0][-1]. ;

      /* Performing sum using the paths for a subset of
         local threads */
      b = @North.a + @South.a + @East.a + @West.a ;
```

```
        MAIN.{ printf("\n  $$i for Bundle of 'A'Threads\n"
                         "------------------------------\n"); }
        printf ( "%3d%c",a,($$ix[0]==(size-1))?'\n':' ');
        MAIN.{ printf("\n  Sums of Surrounding Points\n"
                         "------------------------------\n"); }
        printf ( "%3d%c",b,($$ix[0]==(size-1))?'\n':' ');
    }
    return 0;
}
```

Program 6.2: Uses pre-computed paths to compute the sum of the nearest neighbors.

Program 6.2 has the following output.

```
  $$i for Bundle of 'A' Threads
-------------------------------
  0   1   2   3   4   5   6   7
  8   9  10  11  12  13  14  15
 16  17  18  19  20  21  22  23
 24  25  26  27  28  29  30  31
 32  33  34  35  36  37  38  39
 40  41  42  43  44  45  46  47
 48  49  50  51  52  53  54  55
 56  57  58  59  60  61  62  63

  Sums of Surrounding Points
-------------------------------
 72  68  72  76  80  84  88  84
 40  36  40  44  48  52  56  52
 72  68  72  76  80  84  88  84
104 100 104 108 112 116 120 116
136 132 136 140 144 148 152 148
168 164 168 172 176 180 184 180
200 196 200 204 208 212 216 212
168 164 168 172 176 180 184 180
```

The first array that prints out is the array comprised of the logical thread identifiers of the 'A' threads. In the second array, for each thread in 'A', we compute the sum of the North, East, South, and West logical thread identifiers. (Note: The logical identifier for the individual thread itself is not included in the sum.) For example, the value printed out for A[0][0] (72) is obtained by adding the identifiers of the North thread (56), the East thread (1), the South thread (8) and the West thread (7). 56+1+8+7=72.

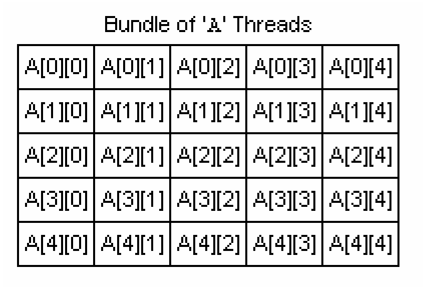
6.3 Using Pre-Computed Paths to Build a Pascal Matrix

Pascal's matrix plays an important role in signal processing, image processing, image coding, and statistics. We can use pre-computed paths to create Pascal's matrix, as the following program demonstrates.

```
/*
   Sample aCe program
   Program to form Pascal's matrix
   Two-dimensional bundle of threads

   Maurice F. Aburdene
   John E. Dorband
*/
#include <stdio.aHr>
#define size 5
threads A[size][size] ;

int main () {
   A int a, i;
   A path(A) North, East;
   A.{
      a=0;
      /* set thread A[0][0].a to 1 */
      if ( ($$ix[0]==0) && ($$ix[1]==0) ) a=1;

      /* $$ix[0] gives the column address */
      /* $$ix[1] gives the row address */
         for(i=1;i<(2*size-1);i++) {
            /* selects a i-th diagonal of the matrix */
            if ( ($$ix[0]+$$ix[1])==i ) {
               /*.A[0][-1]. is relative addressing, which
               is toroidal or wrap around.   Therefore,
               the previous column of the 1st column is
               the last column */
               North = .A[0][-1].;
                  /* fetches value from previous column,
                     same row */
               East  = .A[-1][0].;
                  /* fetches value from previous row, same
                     column */
               a = @North.(a) + @East.(a);
            }
         }
      MAIN.{ printf("%dx%d Pascal Matrix\n"
                "-----------------\n",size,size); }
      printf ( "%2d%c",a,($$ix[0]==(size-1))?'\n':' ');
   }
   return 0;
}
```

Program 6.3: Creates Pascal's matrix using a two-dimensional bundle of threads and relative addressing.

The program creates a bundle of threads, laid out as follows.

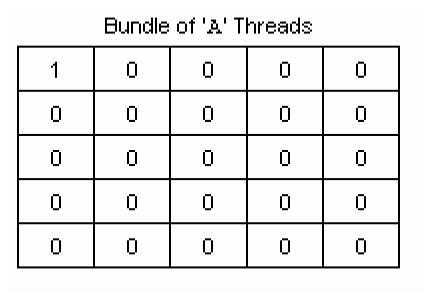


Bundle of 'A' Threads

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]
A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]
A[2][0]	A[2][1]	A[2][2]	A[2][3]	A[2][4]
A[3][0]	A[3][1]	A[3][2]	A[3][3]	A[3][4]
A[4][0]	A[4][1]	A[4][2]	A[4][3]	A[4][4]

Figure 6.2: Structure of the bundle of '`A`' threads created by program 6.3.

Each of these '`A`' threads has a storage location named '`a`'. Initially, all of those storage locations equal zero. In program 6.3, we first set `A[0][0].(a)=1`, like this.

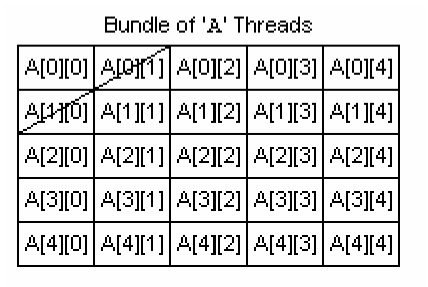


Bundle of 'A' Threads

1	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Figure 6.3: Setting `A[0][0].a = 1`.

Then, we enter the `for` loop. In this loop, we have a counter, '`i`', that is used to make different '`A`' threads inactive, depending on how many times we have gone through the loop. Each time through the loop, we modify the values of '`a`' of the threads for which `$$ix[0]+$$ix[1]=i`. For the first time through the loop (`i=1`), the only active threads are the ones lying along the following diagonal.

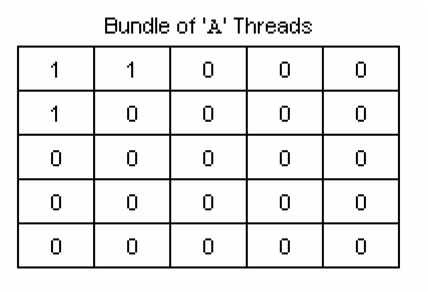


Bundle of 'A' Threads

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]
A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]
A[2][0]	A[2][1]	A[2][2]	A[2][3]	A[2][4]
A[3][0]	A[3][1]	A[3][2]	A[3][3]	A[3][4]
A[4][0]	A[4][1]	A[4][2]	A[4][3]	A[4][4]

Figure 6.4: Diagonal of active threads for the first time through the `for` loop in program 6.3.

Only the threads lying along the diagonal are active during this pass through the `for` loop. After this pass through the loop, the ‘`a`’ values for the bundle of ‘`A`’ threads are as follows.

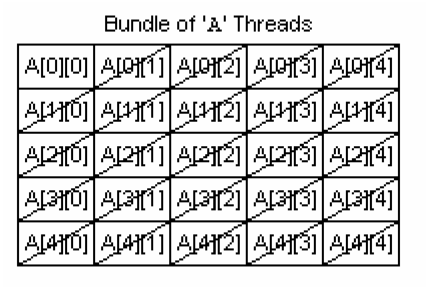


Bundle of 'A' Threads

1	1	0	0	0
1	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Figure 6.5: Values of ‘`a`’ for the ‘`A`’ threads after one pass through the for loop in program 6.3.

As the program progresses, the ‘`a`’ values along the following diagonals are progressively updated during successive passes through the `for` loop.

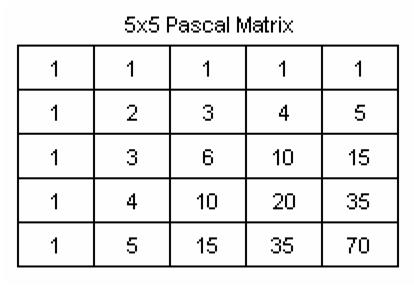


Bundle of 'A' Threads

A[0][0]	A[0][1]	A[0][2]	A[0][3]	A[0][4]
A[1][0]	A[1][1]	A[1][2]	A[1][3]	A[1][4]
A[2][0]	A[2][1]	A[2][2]	A[2][3]	A[2][4]
A[3][0]	A[3][1]	A[3][2]	A[3][3]	A[3][4]
A[4][0]	A[4][1]	A[4][2]	A[4][3]	A[4][4]

Figure 6.6: Diagonals corresponding to the ‘`A`’ threads with `($$ix[0]+$$ix[1]=i)` in the `for` loop for program 6.3.

Here is the Pascal matrix that the program creates.



5x5 Pascal Matrix

1	1	1	1	1
1	2	3	4	5
1	3	6	10	15
1	4	10	20	35
1	5	15	35	70

Figure 6.7: 5x5 Pascal matrix created by program 6.3.

Exercise 6.3: Using any editor you choose, enter program 4.8. Copy the two `printf` statements from the end of the program, and paste them inside the `for` loop, after the terminating bracket of the `if` statement. Compile and execute the program to see how the Pascal matrix is created.

6.4 Arrays of Paths and Path Pointers

aCe allows the declaration of arrays of paths and pointers to paths. Arrays of paths may also be declared as follows.

```
threads H[4];
threads {B[5]} C[5];
H.{
   path(B) toB[4];
   toB[0] = C[x-1].B[y-1].;
   toB[1] = C[x-1].B[y+1].;
   toB[2] = C[x+1].B[y-1].;
   toB[3] = C[x+1].B[y+1].;
   @toB[1].b1 = a1;
}
```

aCe allows pointers to paths as well. Two cases follow, demonstrating two different assignment statements for path pointers.

```
Case #1:                      Case #2:
threads H[4];                 threads H[4];
threads {B[5]} C[5];          threads {B[5]} C[5];
H.{                           H.{
   path(B) toB;                  path(B) toB[4];
   path(B) *toBP;                path(B) *toBP;
   int a1;                       int a1;
   B int b1;                     B int b1;
   toBP = &toB;                  toBP = &toB[2];
```

```
                 @(*toBP).b1 = a1;            @(*toBP).b1 = a1;
              }                            }
```

In case #1, we create a path and a path pointer. We then compute the address of the path, and assign that address to the path pointer. In case #2, we create an array of paths and a path pointer. Then, we assign the address of one element of the array to the path pointer, because, after all, each address location is set up to contain a path. We do not include an example here, but we could also explicitly assign the value of one path pointer to another.

An array element from a path array need not be enclosed in parentheses when used, but a pointer to a path or any address expression that points to a path does. A reduction address description cannot be assigned to a path. A path assignment with any other path description is permitted (i.e. absolute addressing, universal addressing, or relative addressing).

6.5 Summary

- A scatter operation sends data to the threads of a remote bundle. This happens when the router expression is on the left side of an assignment statement.

- A gather operation fetches data from the threads of a remote bundle. This occurs when the router expression is on the right side of an assignment statement.

- In communications operations, computing the identifiers of remote threads takes a fair amount of time. aCe allows a programmer to define a path description as a variable that is computed at run time. A path is computed once at its assignment statement, and it is used (as many times as necessary) by placing a '@' symbol before the name of the path (each time the path is used).

- When a path assignment is made, the address (on the right side of the assignment statement) must be terminated by a period and semi-colon (`.;`).

- aCe allows a programmer to create arrays of paths and path pointers. When using a pointer to a path or an address expression that points to a path, the pointer or address expression (which comprise the router expression) must be enclosed in parentheses. Parentheses are not necessary when using path array elements.

Chapter 7: Routines

7.1 Bundle-Specific Routines

In aCe, we refer to functions as *routines.* When we create a function, we must specify the bundles of threads for which it may be used. aCe does not allow multiple bundle specifiers; it allows routines to be bundle-specific, meaning that a routine can be written for use only by one bundle of threads, and it allows routines to be generic, meaning that a routine can be written for use by any bundle of threads in a program. In this section, we focus on *bundle-specific routines*.

The following program makes use of routines for printing out the system constants, as we did in chapter 2.

```
/*
   Sample aCe program
   Program to print out the thread identifiers using
      bundle-specific routines

   Maurice F. Aburdene
   John E. Dorband
   Michelle L. McCotter
*/
#include <stdio.aHr>
threads A[4];

A void aCe_print_header(void)
{
   /* prints a header line for the sys constants */
   if ($$i == 0)
      printf("\nSystem constants for the `%s' threads\n"
             "-----------------------------------"
             "\n",$$Name);
}

A void aCe_print_char(char *s,char *a)
{
   /* prints char strings from each thread of a bundle */
   if ($$i==0) printf("%s =  ",s);
      /* Only one thread prints "a =" */
   printf("  %s",a);
      /* each thread prints its value of *a in the order of
```

```
         its identifier */
   if ($$i==0) printf("\n");
      /* Only one thread prints "\n" */
}

A void aCe_print_int(char *s,int a)
{
   /* prints an integer from each thread of a bundle */
   if ($$i==0) printf("%s =  ",s);
      /* Only one thread prints "a =" */
   printf("%2d ",a);  /* each thread prints its value of a
      in the order of its identifier */
   if ($$i==0) printf("\n");
      /* Only one thread prints "\n" */
}

A void aCe_printEOL(void)
{
   /* prints one end of line character */
   if ($$i == 0)
      printf("\n");
}

int main () {
   A.{
      aCe_print_header();
      aCe_print_char("$$Name",$$Name);
      aCe_print_int("$$D    ",$$D);
      aCe_print_int("$$N    ",$$N);
      aCe_print_int("$$L    ",$$L);
      aCe_print_int("$$i    ",$$i);
      aCe_print_int("$$ii   ",$$ii);
      aCe_print_int("$$Nx[0]",$$Nx[0]);
      aCe_print_int("$$Nx[1]",$$Nx[1]);
      aCe_print_int("$$Lx[0]",$$Lx[0]);
      aCe_print_int("$$Lx[1]",$$Lx[1]);
      aCe_print_int("$$ix[0]",$$ix[0]);
      aCe_print_int("$$ix[1]",$$ix[1]);
      aCe_printEOL();
   }
   return 0;
}
```

Program 7.1: Creates a one-dimensional bundle of threads and prints the system constants for that bundle.

In this program, we declare the bundle of threads (`threads A[4]`) before we create any routines. This is not always necessary. However, if a routine is created for a particular bundle of threads, the threads declaration that includes that bundle of threads must appear before the routine in a program. In program 7.1, after we declare the bundle of ‘`A`’ threads, we create four routines, specific to the bundle of ‘A’ threads. Then, in the function *main()*, we use the routines. Note the output for the program, as follows.

```
System constants for the `A' threads
------------------------------------
$$Name =    A  A  A  A
$$D     =   1  1  1  1
$$N     =   4  4  4  4
$$L     =   2  2  2  2
$$i     =   0  1  2  3
$$ii    =   0  1  2  3
$$Nx[0] =   4  4  4  4
$$Nx[1] =   0  0  0  0
$$Lx[0] =   2  2  2  2
$$Lx[1] =   0  0  0  0
$$ix[0] =   0  1  2  3
$$ix[1] =   1  2  3  0
```

In chapter 2, we print the system constants for a one-dimensional bundle of threads with program 2.4. We do the same thing here, though program 7.1 is more clear.

Exercise 7.1: Write a program that uses bundle-specific routines to create and print a 6x6 Pascal matrix. *(Hint: Refer to program 6.3 if you get stuck.)*

Solution:

```
/*
   Sample aCe program
   Program to form Pascal's matrix
   Two dimensional bundle of threads
   Uses simple bundle-specific routines

   Maurice F. Aburdene
   John E. Dorband
   Michelle L. McCotter
*/
#include <stdio.aHr>
#define size 6
threads A[size][size];
A int a;

A void createMatrix(void) {
   /* creates a Pascal Matrix */
   path(A) North, East;
   int i;
   a=0;
   /* set thread A[0][0].a to 1 */
   if ( ($$ix[0]==0) && ($$ix[1]==0) ) a=1;
   for(i=1;i<(2*size-1);i++) {
      /* selects a i-th diagonal of the matrix */
      if ( ($$ix[0]+$$ix[1])==i ) {
         North = .A[0][-1].;
         East  = .A[-1][0].;
         a = @North.a + @East.a ;
      }
   }
}

A void printMatrix(void) {
   MAIN.{ printf("  %dx%d Pascal Matrix\n"
                 "----------------------"
                "\n",size,size); }
   printf ("%3d%c",a,
           ($$ix[0]==(size-1))?'\n':' ');
}

int main () {
   A.{
      createMatrix();
      printMatrix();
   }
   return 0;
}
```

Program 7.2: Uses simple bundle-specific routines to create a `6x6` Pascal matrix.

Programs 7.1 and 7.2 use *simple bundle-specific routines,* "bundle-specific" meaning that the routines are created for use by only one bundle, and "simple" meaning that the routine is executed without inter-processor communication. Bundle-specific routines that use inter-processor communication are called *complex bundle-specific routines*. Complex bundle-specific routines can have a bundle of threads as one of its arguments, but the bundle must be a generic bundle. This means that any bundle could be used as an argument. Complex bundle-specific routines with bundle-specific arguments are not allowed in aCe. In fact, they do not make sense because any routine can create an execution context for any bundle of threads, without that bundle having to be passed into the routine as an argument. So it does not make sense to create a complex bundle-specific routine with an argument that is bundle-specific. However, it *does* make sense to have a *generic bundle* as an argument in a complex bundle-specific routine. We provide an example this after we discuss generic routines in the next section.

7.2 Simple Generic Routines

Most routines are specific to only one bundle. Some, however, are useful to many bundles. These are referred to as *generic routines*. A generic routine is declared by preceding it with the keyword '`generic`' in place of a specific bundle name. Trigonometric functions are examples of such routines. There is nothing about a sine function, for example, that makes it inherently specific to any bundle. A sine function can be applied to all threads of a bundle, and is trivially parallel (i.e. requires no communication between threads.) This makes it a *simple generic routine*, and allows it to be executed within the context of any bundle. An example of a simple generic routine is the following.

```
generic double sin(double x) { /* C code */ }
```

We can use simple generic routines in a modified version of program 7.1 to print out the system constants for any bundle of threads, as shown in the program that follows.

```
/*
   Sample aCe program
   Program to print out the thread identifiers
      using generic routines

   Maurice F. Aburdene
   John E.Dorband
   Michelle L. McCotter
*/
#include <stdio.aHr>
threads {B[3]} A[4];

generic void aCe_print_header(void)
{
/* prints a header line for the sys constants */
   if ($$i == 0)
      printf("\nSystem constants for the `%s' threads\n"
             "----------------------------------"
             "\n",$$Name);
}
```

```
generic void aCe_print_char(char *s,char *a)
{
/* prints char strings from each thread of a bundle */
   if ($$i==0) printf("%s =  ",s);
      /* Only one thread prints "a =" */
      printf("  %s",a);
      /* each thread prints its value of *a in the
         order of its identifier */
   if ($$i==0) printf("\n");
      /* Only one thread prints "\n" */
}

generic void aCe_print_int(char *s,int a)
{
/* prints an integer from each thread of a bundle */
   if ($$i==0) printf("%s =  ",s);
      /* Only one thread prints "a =" */
   printf("%2d ",a);
   /* each thread prints its value of a in the order
      of its identifier */
   if ($$i==0) printf("\n");
   /* Only one thread prints "\n" */
}

generic void aCe_printEOL(void)
{
/* prints one end of line character */
   if ($$i == 0)
      printf("\n");
}

int main () {
   A.{
      aCe_print_header();
      aCe_print_char("$$Name",A.$$Name);
      aCe_print_int("$$D    ",A.$$D);
      aCe_print_int("$$N    ",A.$$N);
      aCe_print_int("$$L    ",A.$$L);
      aCe_print_int("$$i    ",A.$$i);
      aCe_print_int("$$ii   ",A.$$ii);
      aCe_print_int("$$Nx[0]",A.$$Nx[0]);
      aCe_print_int("$$Nx[1]",A.$$Nx[1]);
      aCe_print_int("$$Lx[0]",A.$$Lx[0]);
      aCe_print_int("$$Lx[1]",A.$$Lx[1]);
      aCe_print_int("$$ix[0]",A.$$ix[0]);
      aCe_print_int("$$ix[1]",A.$$ix[1]);
      aCe_printEOL();
   }
   B.{
      aCe_print_header();
      aCe_print_char("$$Name",B.$$Name);
      aCe_print_int("$$D    ",B.$$D);
      aCe_print_int("$$N    ",B.$$N);
      aCe_print_int("$$L    ",B.$$L);
      aCe_print_int("$$i    ",B.$$i);
      aCe_print_int("$$ii   ",B.$$ii);
      aCe_print_int("$$Nx[0]",B.$$Nx[0]);
      aCe_print_int("$$Nx[1]",B.$$Nx[1]);
      aCe_print_int("$$Nx[2]",B.$$Nx[2]);
      aCe_print_int("$$Lx[0]",B.$$Lx[0]);
      aCe_print_int("$$Lx[1]",B.$$Lx[1]);
      aCe_print_int("$$Lx[2]",B.$$Lx[2]);
      aCe_print_int("$$ix[0]",B.$$ix[0]);
      aCe_print_int("$$ix[1]",B.$$ix[1]);
      aCe_print_int("$$ix[2]",B.$$ix[2]);
      aCe_printEOL();
  }
  return 0;
}
```

Program 7.3: Uses generic routines to print out the system constants of bundles of bundles of threads.

When using generic routines in a program, the threads declaration does not have to appear before the generic routines, but it can. Note that we use the generic routines for both bundles of threads that we create. We could have used the routines for the ‘`MAIN`’ thread, or any other bundle of threads we may have chosen to create.

7.3 Bundle-Specific Routines Revisited

In this section, we return to our discussion of bundle-specific routines. As we stated in section 7.1, it does not make sense to allow bundle-specific arguments in a bundle-specific routine, since any routine can create an execution context for any bundle of threads. However, it *does* make sense to allow generic bundles as arguments. Say we are writing a program in which we want to allow only ‘`MAIN`’ to issue print commands. Say, also, that we want to be able to print out system constants for other bundles of threads. We can write a function that is specific to ‘`MAIN`’ that prints out system constants for a generic bundle of threads. Note the example program that follows.

```
/*
   Sample aCe program
   Program to print out the thread identifiers
      using bundle-specific routines with generic arguments

   Maurice F. Aburdene
   John E.Dorband
   Michelle L. McCotter
*/
#include <stdio.aHr>
threads {B[3]} A[4];

MAIN void aCe_print_header(generic(OTHER))
{
   /* prints a header line for the sys constants */
   if ($$i == 0)
      printf("\nSystem constants for the '%s' threads\n"
             "-----------------------------------"
             ",OTHER.$$Name);
}

MAIN void aCe_print_char(char *s,char *a,int ident)
{
   /* prints char strings from each thread of a bundle */
   if (ident==0) printf("\n%s =  ",s);
      /* Only one thread prints "a =" */
   printf("  %s",a);
      /* each thread prints its value of *a in the
         order of its identifier */
}

MAIN void aCe_print_int(char *s,int a,int ident)
{
   /* prints an integer from each thread of a bundle */
   if (ident==0) printf("\n%s =  ",s);
      /* Only one thread prints "a =" */
   printf("%2d ",a);
      /* each thread prints its value of a in the order
         of its identifier */
}

MAIN void aCe_printEOL(int ident)
{
   /* prints one end of line character */
   if (ident==0)
      printf("\n");
}

MAIN void aCe_printSysConst(generic(OTHER)) {
   int i;
   aCe_print_header(OTHER);
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_char("$$Name",OTHER.$$Name,i); }
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$D    ",OTHER.$$D,i);     }
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$N    ",OTHER.$$N,i);     }
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$L    ",OTHER.$$L,i);     }
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$i    ",OTHER.$$i,i);     }
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$ii   ",OTHER.$$ii,i);    }
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$Nx[0]",OTHER.$$Nx[0],i);}
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$Nx[1]",OTHER.$$Nx[1],i);}
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$Lx[0]",OTHER.$$Lx[0],i);}
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$Lx[1]",OTHER.$$Lx[1],i);}
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$ix[0]",OTHER.$$ix[0],i);}
   for (i=0; i<OTHER.$$N; i++) {
      aCe_print_int("$$ix[1]",OTHER.$$ix[1],i);}
   for (i=0; i<OTHER.$$N; i++) {
      aCe_printEOL(i);                          }
}

int main () {
  aCe_printSysConst(A);
  aCe_printSysConst(B);
  return 0;
}
```

Program 7.4: Uses bundle-specific routines with generic bundle arguments to print out system constants of different bundles of threads.

Each one of the `aCe_print` routines was a bundle-specific designed only for the ‘MAIN’ thread. By passing a generic bundle in as an argument in the function `aCe_printSysConst`, we allow ‘`MAIN`’ to print the system constants of any bundle of threads. No other bundle besides ‘`MAIN`’ is allowed to use the print functions that we have designed in this program, although other threads could use the `printf` command.

As you might expect, the output is as follows. It is very similar to the output for program 7.3. The only difference is that we did not write code to enable program 7.3 to print out all of the system constants for various thread bundles with different dimensions.

```
System constants for the 'A' threads
------------------------------------
$$Name =    A  A  A  A
$$D     =   1  1  1  1
$$N     =   4  4  4  4
$$L     =   2  2  2  2
$$i     =   0  0  0  0
$$ii    =   0  0  0  0
$$Nx[0] =   4  4  4  4
$$Nx[1] =   0  0  0  0
$$Lx[0] =   2  2  2  2
$$Lx[1] =   0  0  0  0
$$ix[0] =   0  1  2  3
$$ix[1] =   1  2  3  0


System constants for the 'B' threads
------------------------------------
$$Name =    B  B  B  B  B  B  B  B  B  B  B  B
$$D     =   1  1  1  1  1  1  1  1  1  1  1  1
$$N     =  12 12 12 12 12 12 12 12 12 12 12 12
$$L     =  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
$$i     =   0  0  0  0  0  0  0  0  0  0  0  0
$$ii    =   0  0  0  0  0  0  0  0  0  0  0  0
$$Nx[0] =   3  3  3  3  3  3  3  3  3  3  3  3
$$Nx[1] =   0  0  0  0  0  0  0  0  0  0  0  0
$$Lx[0] =  -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
$$Lx[1] =   0  0  0  0  0  0  0  0  0  0  0  0
$$ix[0] =   0  1  2  0  1  2  0  1  2  0  1  2
$$ix[1] =   1  2  0  1  2  0  1  2  0  1  2  0
```

7.4 Complex Generic Routines

Now, we return to our discussion of generic routines can also include inter-processor communication. These are *complex generic routines*. A complex generic routine can also have a bundle of threads as one of its arguments. To pass a bundle into a generic routine other than the bundle of the execution context, one prefixes the argument with the keyword ‘`generic`’, as in the following code.

```
generic double func(generic(OTHER), double x, OTHER int y)
{
   int a,b;
   OTHER int c,d;
   a = OTHER[b].d;
}
```

The bundle passed into the routine is ‘OTHER’. Note that variables ‘x’, ‘a’, and ‘b’ belong to the context bundle, and variables ‘y’, ‘c’, and ‘d’ belong to the bundle passed in as ‘OTHER’. To pass a bundle description into a complex generic routine, the bundle name is the argument, as in the code that follows.

```
threads A[200];
threads B[400];
A.{
   double a,b;
   B int c;
   a = func(B,b,c);
}
```

Function ‘func’ is called by bundle ‘A’ and is passed the bundle ‘B’. Note that ‘b’ is a variable of bundle ‘A’ and ‘c’ is a variable of bundle ‘B’. We include the following program, building off of previous programs in the chapter, to demonstrate the use of complex generic routines.

```
/*
   Sample aCe program
   Program to print out the thread identifiers
      using complex generic routines

   Maurice F. Aburdene
   John E.Dorband
   Michelle L. McCotter
*/
#include <stdio.aHr>
threads {B[3]} A[4];

generic void aCe_print_header(void)
{
   /* prints a header line for the sys constants */
   if ($$i == 0)
      printf("\nSystem constants for the '%s' threads\n"
             "-----------------------------------"
             "\n",$$Name);
}

generic void aCe_print_char(char *s,char *a)
{
   /* prints char strings from each thread of a bundle */
   if ($$i==0) printf("%s =  ",s);
      /* Only one thread prints "a =" */
   printf("  %s",a);
      /* each thread prints its value of *a in the
         order of its identifier */
   if ($$i==0) printf("\n");
      /* Only one thread prints "\n" */
}

generic void aCe_print_int(char *s,int a)
{
   /* prints an integer from each thread of a bundle */
   if ($$i==0) printf("%s =  ",s);
      /* Only one thread prints "a =" */
   printf("%2d ",a);
      /* each thread prints its value of a in the order
         of its identifier */
   if ($$i==0) printf("\n");
   /* Only one thread prints "\n" */
}

generic void aCe_printEOL(void)
{
   /* prints one end of line character */
   if ($$i == 0)
      printf("\n");
}

generic void aCe_printSysConst(generic(OTHER)) {
   OTHER.{
      aCe_print_header();
      aCe_print_char("$$Name",$$Name);
      aCe_print_int("$$D    ",$$D);
      aCe_print_int("$$N    ",$$N);
      aCe_print_int("$$L    ",$$L);
      aCe_print_int("$$i    ",$$i);
      aCe_print_int("$$ii   ",$$ii);
      aCe_print_int("$$Nx[0]",$$Nx[0]);
      aCe_print_int("$$Nx[1]",$$Nx[1]);
      aCe_print_int("$$Lx[0]",$$Lx[0]);
      aCe_print_int("$$Lx[1]",$$Lx[1]);
      aCe_print_int("$$ix[0]",$$ix[0]);
      aCe_print_int("$$ix[1]",$$ix[1]);
      aCe_printEOL();
   }
}

int main () {
   aCe_printSysConst(A);
   aCe_printSysConst(B);
   return 0;
}
```

Program 7.5: Uses simple and complex generic routines to print system constants of different bundles of threads.

Program 7.5 has the same output as program 7.4. We list the output in section 7.3.

Exercise 7.2: Practice writing a variety of aCe routines. Either write your own new programs, or modify programs shown in the tutorial by replacing our code with bundle-specific or generic routines.

A generic or an "argument passed" bundle can have multiple dimensions. The sizes of the dimensions may either be specified as in case one, or unspecified as in case two.

```
Case 1:  generic(OTHER[2][500][30])
Case 2:  generic(OTHER[][][])
```

Case one will only allow bundles with dimensions `[2][500][30]` to be passed as an argument, while case two will allow any three-dimensional bundle to be passed. A bundle being passed as an argument must have the same number of dimensions as the generically declared bundle.

If the generically declared bundle does not specify any dimensions, then a bundle with any number of dimensions may be passed, but it will be treated as a one-dimensional bundle. The index value of this array may range from zero to the value of `$$N-1` of the bundle. A generically declared bundle may also specify the child bundles it must have to be passed. All bundles of the passed bundle must have at least as many child bundles and the same number of dimensions as the argument bundle. Also, the sizes of the dimensions must be specified and have the same values. For case three, '`A`' and '`C`' are valid passed bundles for '`OTHER`'; bundles '`B`', '`D`', and '`E`' are not.

```
threads {a[2],b[4][6]} A[200];
threads {d[2],e[5][6]} B[400];
threads {f[2],g[4][6],h[5]} C[300];
threads {j[2],k[4],m[5]} D[300];
threads {n[2],p[5],q[4][6]} E[300];
Case 3:  generic{{y[2],z[4][6]} OTHER)
```

'`A`' is an exact match. '`B`' is invalid because '`e`' does not have the same dimension size as the generic function requires. '`C`' is valid, even through it has three children. Remember, all bundles of the passed bundle must have *at least as many* child bundles. For '`C`', the third child is simply ignored since '`f`' matches '`y`' and '`g`' matches '`z`'. '`D`' is invalid because '`k`' only has one dimension. '`E`' is invalid even though children of the right size exist; the reason is that they are not the *first* two children of '`E`'.

The context bundle (meaning the bundle of the current execution context) may also be passed to the routine by name. This is done using the same syntax as declaring a bundle to be passed by argument, except it replaces the keyword '`generic`' preceding the routine declaration as in case four.

```
Case 4:  generic{{y[2],z[4][6]} OTHER)
```

```
double FUNC(double x) { /* C Code */ }
```

In case four, the context bundle in which ‘`FUNC`’ is called must be of the form ‘`OTHER`’, as if it were a passed bundle. Without this capability, the context bundle would not be able to be referenced explicitly. This is not a problem with functions like trigonometric functions that require no inter-thread communications. However, it would be for, say, a generic FFT routine (Fast Fourier Transform routine). Any bundle that has been explicitly declared as a bundle argument or as a generic bundle context may be used within the context of the function as any bundle specification can be used.

Exercise 7.3: Experiment with generic routines by writing your own programs, and observing what compiler errors result.

7.5 The Virtual/Scalar Distinction

So far, the programmer has been presented with a view of a variable declared over a bundle as having an independent value for every thread of that bundle. This can lead to extreme inefficiency of both memory and compute cycles if implemented to its logical conclusion. The aCe language is designed to maintain the illusion that each thread has its own storage for each variable, but is implemented in a way that takes advantage of variables that have the same value in every thread (a scalar value). This is done by recognizing variables that are assigned only constants or other scalar variables. These variables are designated as ‘`scalar`’. If each thread had its own storage location for a particular variable, say ‘`i`’ for a loop counter, this would lead to great inefficiencies. So aCe does not provide separate storage for every variable for each thread. If a given variable is scalar, however, this does not mean that there is exactly one storage location for that variable. That would lead to inefficiencies as well. aCe compromises between the two ideas, providing a program with as many storage locations for scalar variables as necessary, perhaps more than one but less than the total number of threads (the number of instances of that variable).

Most other variables are designated as ‘`virtual`’. It would not be necessary to make the programmer aware of this in general, except for the fact that aCe interfaces with the *native programming environment*. By native programming environment, we mean a language other than aCe that is native to a machine, like ANSI C, for example. aCe allows a programmer to create links to C routines in an aCe program. However, the routines developed under the native programming environment (i.e. C routines) expect parameters and return values of specific types, either scalar or virtual, but not both. C does not implement variables in the same way as aCe does, with the same flexibility for scalar values. Therefore, when a native code routine is declared within aCe (to link aCe to a C routine, for example), the arguments and parameters must be explicitly designated as either scalar or virtual. Variables used in aCe routines created for an aCe program do not have to be designated as either scalar or virtual; the aCe compiler takes care of this on its own.

7.6 Summary

- A *routine* is simply another name for a function.

- Routines in aCe can either be bundle-specific routines or generic routines. Bundle-specific routines can only be called and used by one bundle of threads; generic routines can be used by any bundle of threads.

- Simple generic routines do not involve inter-processor communication. Complex generic routines, on the other hand, do.

- A complex routine (bundle-specific or generic) can have, but does not need to have, another bundle of threads as one of its arguments.

- In a complex routine, a generic bundle (used as an argument) can have multiple dimensions. The sizes of the dimensions may or may not be specified.

- aCe gives the programmer the *illusion* that each thread has its own copy of every variable. However, with scalar variables, there are most likely fewer copies of the variable than there are instances of the variable (one instance for each thread), but there can be more than one copy, in order to make execution more efficient when different processors are involved.

- When linking an aCe program to routines in the native programming environment, the arguments and parameters in the aCe declaration must be explicitly designated as either scalar or virtual.

Chapter 8: Thread Alignment

8.1 Virtual Processor Alignment

When a bundle of threads is created, the threads are distributed across the physical processors in a default manner specific to the given architecture. If no realignment is specified for the bundle of threads, the default distribution will be unchanged, and the values of `$$i` (the logical thread identifier) and `$$ii` (the physical thread identifier) will be equal for every thread in the bundle, which we have seen in every discussion of the system constants so far. This often does not represent a very effective arrangement given the communications patterns of a specific algorithm. The realignment statement allows for the rearrangement of the threads across the physical processors relative to the default arrangement. It is with this realignment statement that the values of `$$i` and `$$ii` can differ.

Actually, threads are mapped across physical processors in an order dependent on the architecture. For example, an aCe program running on a PC bundle running MPICH as its message passing protocol may distribute threads in the following manner: given that there are '`n`' threads to be distributed over '`p`' processors and '`d=n/p`', the first '`d`' threads will be allocated to the first processor, the second '`d`' threads to the second processor, and so on until all threads have been allocated. If '`n`' is not evenly divisible by '`p`', then '`d=n/p+1`', and the last processor will have '`n mod p`' threads. The idea of realignment is to re-map the logical thread identifier to a different physical thread identifier.

8.2 Realignment of Threads with Processors

Rearrangement is the permuting of logical thread relative to a fixed arrangement of physical threads. In its simplest form, this is done by defining an array of logical threads, and permuting the dimensions of the array. For example, if we have a two-dimensional array, and we swap the two dimensions, this is equivalent to performing a transpose on the elements of the array. The following example shows how a transpose array of logical threads is defined.

```
threads X[32][32];
realign X[32][32]-->[32][32]; [0]-->[1]; [1]-->[0];;
```

Realignment is described by a *realignment descriptor statement*. The realignment keyword is '`realign`', as shown in the preceding example code. It is followed by the name of the bundle to be realigned. Next are two sets of dimensions separated by an arrow (`-->`). The first set of dimensions describes an array that represents how the logical identifiers are to be viewed. And the second represents how the physical identifiers are to be viewed. In this example, both the logical and the physical identifiers are represented by a `32x32` array of threads. Following the array dimensions are the mappings. These mappings map one or more logical dimensions to one or more physical dimensions. In this case, there are two mappings, each consisting of one logical dimension mapped to one physical dimension. The first mapping maps the least significant logical dimension to the most significant physical dimension. The second maps the most significant logical dimension to the least significant physical dimension. Finally, the realignment statement is terminated by two semi-colons (`;;`). This realignment effectively transposes the logical arrangement of threads relative to the physical arrangement. This realign statement is equivalent to the following code.

```
realign X[32][32]-->[32][32]; [0][1]-->[1][0];;
```

The following example uses C-like code to show how logical threads are mapped to physical threads.

```
threads C[27][10];
realign X[6][9][5]-->[9][5][6];
[0]-->[1]; [1]-->[2]; [2]-->[0];;
```

The realignment description refers to a physical array of threads which could be represented by `P[9[5][6]` and a logical array of threads represented by `L[6][9][5]`. The previous dimension mapping describes how to change the ordering of the dimensions of a multi-dimensional array (a generalized multi-dimensional transpose). This description is equivalent to the following C-like code describing the mapping of physical threads to logical threads.

```
int L[6][9][5], P[9][5][6];
for (I2=0;, I2<6; I2++) {
   for (I1=0; I1<6; I1++) {
      for (I0=0; I0<6; I0++) {
         L[I0][I2][I1] maps_to P[I2][I1][I0];
      }
   }
}
```

This simple form of the alignment statement allows for any form of multidimensional array transposing relative to the default arrangement. The next form that we describe allows for changing from an array with one number of dimensions to one of a different number of dimensions.

```
threads Y[100][100];
realign Y[100][100]-->[10][25][20][2];
[0][1]-->[1][3][2][0];;
```

In the preceding case, the logical array is defined as a `100x100` two-dimensional array, and the physical array is a four-dimensional array of dimensions `10x25x20x2`. The mapping (that follows the array declaration) says to map the transpose of the logical array to an array of the physical thread whose dimensions have been reordered to be `[20][10][25][2]`. This mapping statement appears to request the reordering of the physical threads. In reality, however, the physical threads will remain in the default order, and only the logical threads will be reordered so that mapping from logical to physical is consistent with the mapping statement.

Note that the original and the final arrays of threads are the same size. This is not necessary. If the physical thread array size is larger than the logical, the array is padded with inactive threads that never become active. If the logical array is larger than the physical array size, then the final array descriptor is given an additional most significant dimension large enough to make the physical array just larger than the logical. In the following code, realignment statement #1 is effectively equivalent to realignment statement #2.

```
threads Y[100][100];
Realignment Statement #1:
   realign Y[100][100]-->[32][32]; [1][0]-->[0][1];;
Realignment Statement #2:
   realign Y[100][100]-->[10][32][32]; [1][0]-->[2][0][1];;
```

There is one final form of the alignment state. The realignment description describes not only how an array may be rearranged or reformed into another array, but it can also be used to describe how sub-arrays may be reformed into different sub-arrays. An alignment statement may consist of multiple mappings that reorder subsets of the dimension, some of which have no explicitly defined size.

```
threads Z[640][480];
realign Z[640][480]-->[][][64][32];
[1]-->[3][1]; [0]-->[2][0];;
```

The compiler will fill in the blank sizes. Dimensions with blank sizes may only be used as most significant dimensions in a mapping array. The compiler will translate the above realignment statement into the following.

```
realign Z[640][480]-->[10][15][64][32];
[1]-->[3][1]; [0]-->[2][0];
```

For realignment to be of significant usefulness, the programmer will need to determine the default arrangement of threads for the specific physical architecture to which she wants to realign a bundle.

8.3 Summary

- When a programmer creates a bundle of threads, the threads are distributed across physical processors in a default manner specific to the given architecture.
- Values of `$$i` and `$$ii` are equal, unless a programmer issues a realignment statement.
- A *realignment descriptor statement* describes realignment in a program. The keyword is '`realign`', and the statement is always terminated by two semi-colons (`;;`).
- With realignment, it is not necessary for original and final arrays to be the same size.
- Realignment descriptions can also describe how sub-arrays may be reformed into different sub-arrays.

Chapter 9: Applications

9.1 A Review of aCe

We include a chapter on applications of aCe to give you a review of the ideas we have covered in the tutorial. The programs we have chosen to include are not entirely comprehensive, but each program represents the implementation of various aCe concepts.

9.2 Building a Binary Tree

aCe can be used quite easily to build a binary tree. Instead of using C to create node structures with different data objects, in aCe we use a bundle of threads with different variables. Study the program that follows.

```
/*
   Sample aCe program
   Program to create a binary tree ( (2^N)-1 nodes )
   Sum children's values and parent's value for each parent
      using pre-computed paths
   One-dimensional bundle of threads

   Maurice F. Aburdene
   John E. Dorband
   Michelle L. McCotter
*/
#include <stdio.aHr>
#define N 4 /* 15 nodes */
threads A[(1<<N)-1] ;

int main () {
   A int level, id, P, L, R, sum0, sum1;
   A path(A) Parent, Left, Right ;

   A.{
      /* Create Tree */

      /* Determine to which level of the tree a thread
         belongs */
      id = $$i+1;
```

```
level=0;
/* NOTE >>=1 is integer shift right one bit or divide
   by 2 */
while ( (id&1)==0 ) { /* for the least significant
   one bit of id */
   level++;
   id>>=1;
}
/* how many least significant zeros in id */

/* L and R stand for left and right child
   respectively; if L = -1, there is no left child
   (same for R & right child) */
/* leaf nodes are at level zero; root node is at the
   highest level */
L=R=-1;
if ( level>0 ) {
   int a = ( 1 << (level - 1) );
   L = $$i - a;
   R = $$i + a;
}

/* communicate to children who its parent is */
P = -1; /* -1 means there is no parent */
if ( level > 0 ) {
   A[L].P = $$i ;
   A[R].P = $$i ;
}

/* Pre-compute paths to parent and left and right
   children */
Parent = A[P]. ;
Left   = A[L]. ;
Right  = A[R]. ;
/* sum values in children to parent's value */
{
   int i;
   sum0=$$i;
   for(i=0;i<(N-1);i++) {
      sum1 = 0;
         if (level==i) {
            @Parent.sum1 += sum0;
         }
      sum0 += sum1;
   }
   MAIN.{ printf("\n$$i  Lv   L   R   P   Sum\n"
                 "--------------------------\n"); }
   printf("%3d %3d %3d %3d %3d %5d\n",
          $$i,level,L,R,P,sum0);
}
```

```
   }
   return 0;
}
```

Program 9.1: Uses a one-dimensional bundle of threads to create a binary tree. Uses pre-computed paths to sum the logical thread identifiers of each parent thread and all of its children.

Program 9.1 creates a binary tree structure like the diagram in figure 9.1. Each node of the tree is represented by one of the threads in the bundle of fifteen threads. We show each node by a box that encloses the logical thread identifier of the node in standard numerical format, and again in binary format. The binary value should allow you to more clearly see how program 9.1 assigns the left and right children to each parent. Observe the tree structure, and study program 9.1 to make sure you understand how the binary tree is created.

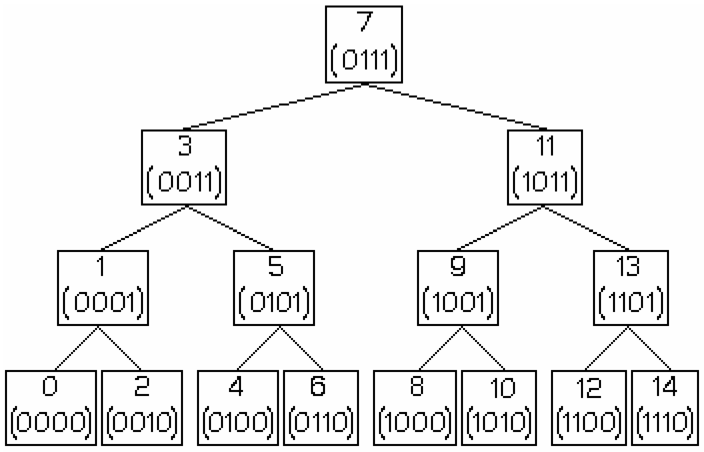


Figure 9.1: Demonstrates the structure of a binary tree created by program 9.1.

The output for program 9.1 is as follows.

```
$$i  Lv   L   R   P   Sum
--------------------------
  0   0  -1  -1   1     0
  1   1   0   2   3     3
  2   0  -1  -1   1     2
  3   2   1   5   7    21
  4   0  -1  -1   5     4
  5   1   4   6   3    15
  6   0  -1  -1   5     6
  7   3   3  11  -1   105
  8   0  -1  -1   9     8
  9   1   8  10  11    27
 10   0  -1  -1   9    10
 11   2   9  13   7    77
 12   0  -1  -1  13    12
 13   1  12  14  11    39
 14   0  -1  -1  13    14
```

By comparing the output with figure 9.1, you will see that each thread's value for '`P`' is, indeed, correct. The value for '`Sum`' is obtained by computing the sum of each thread's logical thread identifier and the logical thread identifiers of all of that thread's children. By "children", we mean any node in the tree that is a "descendant" of a given thread.

The facets of aCe demonstrated by program 9.1 are

- the ability of the programmer to use bundles of threads in creating structures that best represent a problem solution,
- the creation of execution contexts for different bundles of threads ('`A`' and '`MAIN`'),
- the use of thread storage,
- the use of binary operations (which we did not go over in the tutorial),
- the application of universal addressing
- the creation and application of pre-computed paths, and
- the use of active and inactive threads.

9.3 Building a Ring

Thus far, we have not shown a ring structure. Here, we use thread communication to build a ring. Observe the program that follows.

```
/*
   Communication on a ring
   aCe program
   Maurice F. Aburdene
   John E. Dorband
   Michelle L. McCotter
*/
#include <stdio.aHr>
threads A[10];

int main () {
   A.{
      int Left, Right;
      /* Communicates from thread that has the next higher
         id */
      Left = .A[1].$$i;
      /* Communicates from thread that has the next lower
         id */
      Right = .A[-1].$$i ;
      MAIN.{ printf("\n$$i  L  R\n"
                    "---------\n"); }
      printf("%3d %2d %2d\n",$$i,Left,Right);
   }
   return 0;
}
```

Program 9.2: Uses a one-dimensional bundle of threads and thread communication to create a ring structure.

Program 9.2 creates a ring structure that looks something like figure 9.2.

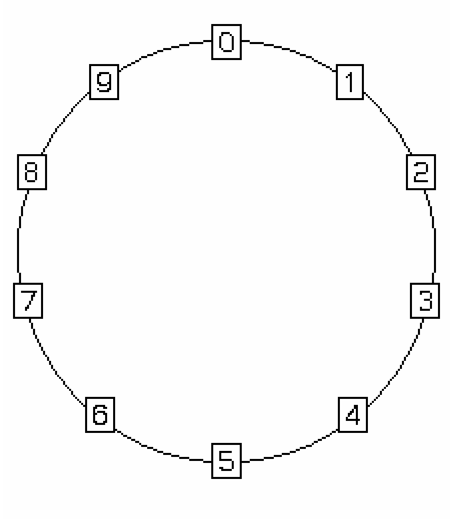


Figure 9.2: The ring structure created by program 9.2.

Although we do not do anything with the ring structure itself, program 9.2 demonstrates how one could use aCe to create a ring. The program has the following output.

```
$$i  L  R
---------
  0  1  9
  1  2  0
  2  3  1
  3  4  2
  4  5  3
  5  6  4
  6  7  5
  7  8  6
  8  9  7
  9  0  8
```

Note that the output matches the diagram.

Now, we include a more complicated program that uses an election algorithm with a ring structure. Observe the following program.

```
/*
   Election algorithm on a ring
   aCe program

   Maurice F. Aburdene
   John E. Dorband
*/
#include <stdio.aHr>
#include <random.aHr>
threads A[10];

int main () {
   A.{
      int address, highest, active, h0, h1, direction,
          done, finish, count, total, sent ;
      highest=address=$$i;
      done=0;
      srand( 477957 );
      active=count=0;
      if ( drand() > 0.5 ) {
         direction=0; active=1;
      }
      MAIN.{ printf("\nHt "); }
      printf("%2d ",highest);
      MAIN.{ printf("\n"); }
      MAIN.{ printf("\nAc "); }
      printf("%2d ",active);
      MAIN.{ printf("\n"); }
      finish=0; sent=0;
      while( ! done ) {
         MAIN.{ printf("\n"); }
```

```
if ( active && (direction==0) && (!sent) ) {
   if ( drand() > 0.5 )
      direction=1;
   else direction=-1;
}

h0=-1;
/* Condition for senders */
if ( active && (direction == 1) )  {
   .A[1].h0 = highest ;
   count++;
   sent=1;
   active=0;
}
/* Condition for receivers */
MAIN.{ printf("\nh0 "); }
printf("%2d ",h0);
if ( (h0>-1) ) {
   /* if a processor received something */
   if ( h0 == address )
      finish=1;
   else if ( h0 > highest ) {
      highest = h0 ;
      direction=1 ;
      active=1;
      sent=1;
   }
   else if ( ! sent ) {
      direction=0 ;
      active=1;
   }
}
MAIN.{ printf("\nHt "); }
printf("%2d ",highest);
MAIN.{ printf("\nFn "); }
printf("%2d ",finish);
MAIN.{ printf("\nAc "); }
printf("%2d ",active);
MAIN.{ printf("\n"); }

h0=-1;
/* Condition for senders */
if ( active && (direction == -1) )  {
   .A[-1].h0 = highest ;
   count++;
   sent=1;
   active=0;
}
/* Condition for receivers */
MAIN.{ printf("\nh1 "); }
printf("%2d ",h0);
```

```
            if ( (h0>-1) ) {
               /* if a processor received something */
               if ( h0 == address )
                  finish=1;
               else if ( h0 > highest ) {
                  highest = h0 ;
                  direction=-1 ;
                  active=1;
                  sent=1;
               }
               else if ( ! sent ) {
               direction=0 ;
               active=1;
               }
            }

            A.done |= finish ;
            MAIN.{ printf("\nHt "); }
            printf("%2d ",highest);
            MAIN.{ printf("\nFn "); }
            printf("%2d ",finish);
            MAIN.{ printf("\nAc "); }
            printf("%2d ",active);
            MAIN.{ printf("\n"); }
         }
         total=0;
         A.total+=count;
         MAIN.{ printf("\nTotal messages sent = %5d\n",
                        A.total); }
      }
      return 0;
   }
```

Program 9.3: Uses a bundle of threads to build a ring, and then uses an election algorithm for message passing in the ring.

Program 9.3 uses a bundle of threads to create a ring, similar to program 9.2. Then, the program uses an election algorithm to do message passing. We choose not to go into detail for program 9.3, but we include it as an example of an aCe application. If you are interested, you can study the program to get an idea of how the election algorithm works, and you can enter and execute the program on your machine to experiment with the output.

The features of aCe shown in program 9.3 are

- the ability of the programmer to use bundles of threads in creating structures that best represent a problem solution,
- the creation of execution contexts for different bundles of threads (‘A’ and ‘MAIN’),
- the use of thread storage,

- the implementation of thread communication using relative addressing, and
- the use of active and inactive threads.

9.4 Conclusion

By this point, you should understand aCe fairly well. As we conclude our explanation of aCe and parallel computing, we would like to draw your attention to one more thing. Throughout the tutorial, you have experienced one of the fundamental strengths of aCe, perhaps without even realizing it. In every program we have included, and in every program we have asked you to write, we have never had to worry about issues of synchronization, or about performing system calls. This is because the aCe compiler takes care of all those details, allowing you to focus on the ideas of parallelism itself, rather than all the details that go into parallel computing. In a way, this makes parallel programming much easier.

If you are interested in understanding parallel programming even better, you may go back and read the introduction again, or you may peruse our list of suggest reading that follows.

Suggested Reading

- Andrews, Gregory R. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley Longman, Inc. 2000.
- Foster, Ian. Designing and Building Parallel Programs. Addison-Wesley, Inc., Argonne National Library, and the NSF Center for Research on Parallel Computation. http://www-unix.mcs.anl.gov/dbpp/. (31 July, 2002).
- Pagadala, Sashi. System Simulation and Image Enhancement Using aCe. MSEE Thesis, Bucknell University, 2002.

Standard aCe C Library

Assert (assert.aHr)

(The libraries for 'assert' are not currently implemented.)

ctype (ctype.aHr)

```
generic int isalnum   ( int c );
generic int isalpha   ( int c );
generic int iscntrl   ( int c );
generic int isdigit   ( int c );
generic int isgraph   ( int c );
generic int islower   ( int c );
generic int isprint   ( int c );
generic int ispunct   ( int c );
generic int isspace   ( int c );
generic int isupper   ( int c );
generic int isxdigit  ( int c );
generic int tolower   ( int c );
generic int toupper   ( int c );
```

Error number (errno.aHr)

(The libraries for 'errno' are not currently implemented.)

Locale Routines (locale.aHr)

(The libraries for 'locale' are not currently implemented.)

Math Routines (math.aHr)

```
generic double cos   ( double x );
generic double sin   ( double x );
generic double tan   ( double x );
generic double acos  ( double x );
generic double asin  ( double x );
generic double atan  ( double x );
generic double atan2 ( double x, double y );
generic double sinh  ( double x );
generic double cosh  ( double x );
generic double tanh  ( double x );
generic double exp   ( double x );
generic double log   ( double x );
generic double log10 ( double x );
generic double frexp ( double value, int *e );
generic double ldexp ( double value, int exp );
generic double modf  ( double value, double *iptr );
generic double pow   ( double base, double exp );
generic double sqrt  ( double x );
generic double floor ( double x );
generic double ceil  ( double x );
generic double fmod  ( double x, double y );
generic double fabs  ( double x );
```

Scan Functions (scan.aHr)

```
generic int     scan_and   (int V,    char M );
generic int     scan_or    (int V,    char M );
generic int     scan_xor   (int V,    char M );
generic int     scan_add   (int V,    char M );
generic float   f_scan_add (float V,  char M );
generic double  d_scan_add (double V, char M );
generic int     scan_min   (int V,    char M );
generic float   f_scan_min (float V,  char M );
generic double  d_scan_min (double V, char M );
generic int     scan_max   (int V,    char M );
generic float   f_scan_max (float V,  char M );
generic double  d_scan_max (double V, char M );
```

Set Jump (setjmp.aHr)

(The libraries for 'setjmp' are not currently implemented.)

Signals (signal.aHr)

(The libraries for 'signal' are not currently implemented.)

Standard arguments (stdarg.aHr)

(The libraries for 'stdarg' are not currently implemented.)

Standard I/O (sdtio.aHr)

```
generic scalar FILE*  fopen   ( scalar char *filename,
                                scalar char *mode) ;
generic scalar int    fflush  ( scalar FILE* stream ) ;
generic scalar int    fclose  ( scalar FILE* stream ) ;
generic scalar int    fread   ( virtual void *virtual ptr,
                                scalar size_t size,
                                scalar size_t nobj,
                                scalar FILE* stream ) ;
generic scalar int    fwrite  ( virtual void *virtual ptr,
                                scalar size_t size,
                                scalar size_t nobj,
                                scalar FILE* stream ) ;
generic virtual int   printf  ( scalar char * scalar format,
                                virtual ... ) ;
generic virtual int   sprintf ( virtual char *virtual buff,
                                scalar char * scalar format,
                                virtual ... ) ;
generic virtual int   fprintf ( scalar  FILE *scalar  file,
                                scalar char * scalar format,
                                virtual ... ) ;
```

Standard Library (stdlib.aHr)

```
generic double         atof    ( const char*  s );
generic int            atoi    ( const char*  s );
generic long           atol    ( const char*  s );
generic double         strtod  ( const char* s, char** endp );
generic long           strtol  ( const char* s, char** endp,
                                 int base );
```

```
generic unsigned long  strtoul  ( const char* s, char** endp,
                                   int base );
generic scalar void    abort    ( void );
generic scalar void    exit     ( scalar int status );
generic int            abs      ( int n );
generic long           labs     ( long n );
generic div_t          div      ( int  num, int  denom );
generic ldiv_t         ldiv     ( long num, long denom );
```

String Routines (string.aHr)

```
generic char*  strcpy   ( char*  s, char* ct );
generic char*  stnrcpy  ( char*  s, char* ct, int n );
generic char*  strcat   ( char*  s, char* ct );
generic char*  strncat  ( char*  s, char* ct, int n );
generic int    strcmp   ( char* cs, char* ct );
generic int    strncmp  ( char* cs, char* ct, int n );
generic char*  strchr   ( char* cs, char   c );
generic char*  strrchr  ( char* cs, char   c );
generic size_t strspn   ( char* cs, char* ct );
generic size_t strcspn  ( char* cs, char* ct );
generic char*  strpbrk  ( char* cs, char* ct );
generic char*  strstr   ( char* cs, char* ct );
generic size_t strlen   ( char* cs );
generic char*  strerror ( int n );
generic char*  strtok   ( char*  s, char* ct );
generic void*  memcpy   ( void*  s, void* ct, int n );
generic void*  memmove  ( void*  s, void* ct, int n );
generic int    memcmp   ( void* cs, void* ct, int n );
generic void*  memchr   ( void* cs, int    c, int n );
generic void*  memset   ( void*  s, int    c, int n );
```

Time Routines (time.aHr)

(The libraries for 'time' are not currently implemented.)

More Applications – Discrete Cosine Transform

One-Dimensional Discrete Cosine Transform

In the image, video, voice and signal processing fields, a discrete cosine transform (DCT) is commonly used to compress data. In digital image processing, the DCT is used to compress an image and is an important part of the JPEG standard.

The data compression process is fairly straightforward. A DCT is performed and creates a transform coefficient matrix. Only the most significant coefficients are stored or transmitted. Data can be recreated using the inverse transform with the stored coefficients. However, unless all of the coefficients are used, this method is lossy. The tradeoff between size and quality is controlled by the amount of coefficients used.

We will begin by looking at the implementation of the one-dimensional DCT. There are many similar formulas for DCTs. Each has its advantages. For this section we use the recursive definition for the one-dimensional DCT as shown below:

$$Y(k) = \frac{2}{N}\gamma_k(-1)^k\sum_{n=0}^{N-1} y(N-1-n)\cos[k(n+\frac{1}{2})\frac{\pi}{N}],$$

$$k = 0,1,...,N-1.$$

Similarly, the inverse discrete cosine transform (IDCT) is:

$$y(n) = (-1)^n\sum_{k=0}^{N-1} y(N-1-k)\sin[(k+1)(n+\frac{1}{2})\frac{\pi}{N}],$$

$$n = 0,1,...,N-1.$$

$$\gamma_k = \frac{1}{2} \text{ if } k = 0, \text{ or } \gamma_k = 1, \text{otherwise.}$$

In the following sections, we will look at many different ways of implementing the two-dimensional transform in aCe. While many of the programs are similar, the differences are noteworthy.

Program B.1 shows one method of implementing the DCT and IDCT using aCe.

```
/*
```

```
Discrete  Cosine  Transform  (DCT)  and  Inverse  Discrete  Cosine  Transform
    (IDCT)
___________________________________________________________

Maurice.F. Aburdene
Andrew Marbach
John E. Dorband
5/30/03
____________________________________________________________

DCT formula for N samples in y[]:


Y(k)= 2/N * Gk * (-1)^k * SUM(n=0->N-1){ y(N-1-n) * cos [ k * (n+0.5) *
    pi/N ] }

    Gk = 1/2, if k = 0;  Gk = 1, otherwise;
    k= 0,1, ... N-1;
____________________________________________________________

IDCT formula for N samples in Y[]:


y(k)= (-1)^k * SUM (n=0->N-1){ Y(N-1-n) * sin [ (k + 0.5) * (n+1) * pi/N ]
    }
    k= 0,1, ... N-1;
____________________________________________________________
*/

#include <stdio.aHr>
#include <math.aHr>
#define N 4            /*defines number of samples*/
#define PI   (4*atan(1))

threads y[N];
threads Y[N];

int main (){

    y double temp, sum=0.0, value=0.0, data;
    y int n;           /*initializes bundle variables*/
    Y double temp, sum=0.0, value=0.0;
    Y int n;

    y.{
        data = (float) $$i;    /*initializes data*/
    }


    printf("\n\n y[]= { ");
    y.{ printf("%f\t", data); }/*print data in y[]*/
    printf("}\n\n");

/*_____________________Start DCT______________________*/

    Y.{
    value = 2./N;              /*scaling factor*/
    value = value*pow(-1,$$i);

    if ($$i==0)                /*gamma coefficient*/
        value=value * 0.5;

    for (n=0; n<N; n++){       /*loop to use all of the samples*/

    temp=y[N-1-n].(data);  /*get y[] data, multiply by cos term*/

    temp=temp*cos($$i*(n+.5)*PI/(float)N);
    sum = sum + temp;          /*add iteration to the sum*/

    /* un-commment to test
    printf("Y[%d] n= %d temp=%f sum=%g\n", $$i,n,temp,sum);
    MAIN.{printf("\n");}  */

    }

    value = value * sum;       /*final value calculation*/

    }


    printf("DCT - \n Y[]= { ");
    Y.{ printf("%f\t", value); }      /*print values of Y[]*/
    printf("}\n\n");


/*________________End DCT, Start IDCT__________________*/


    y.{
    value=pow(-1,$$i);     /*scaling factor*/

    for (n=0; n<N; n++){       /*loop to use all samples*/

        temp=Y[N-1-n].(value); /*get y[] data, multiply by sine*/

        temp = temp * sin((n+1)*($$i+.5)*PI/(float)N);

        sum = sum + temp;      /*add iteration to sum*/


        /* un-commment to test
        printf("y[%d] n= %d temp=%f sum=%g\n", $$i,n,temp,sum);
        MAIN.{printf("\n");}  un-comment to test   */

    }

    value = value * sum;       /*final value calculation*/

    }

    printf("IDCT - \n y[]= { ");
    y.{ printf("%f\t", value); }      /*print values of Y[]*/
    printf("}\n\n");

}
```

Program B.1: **Uses a one-dimensional bundle of threads to perform a discrete cosine transform and then inverts back using the inverse discrete cosine transform.**

Program B.1 uses two bundles of threads (y and Y) to represent the original data and the transform coefficients respectively. Each transform coefficient thread 'gets' the data from each sample data thread. This particular algorithm is fairly simple and can be improved upon. Improvement will come in the form of number of multiplications and additions or in terms of thread communications.

The output of program B.1 can be seen below. Notice that the inverse of the transform is the same as the original values. In this case, all coefficients are used in the IDCT and the conversion is lossless.

```
 y[]= { 0.000000     1.000000      2.000000     3.000000}

DCT -
 Y[]= { 1.500000     -1.577161     0.000000     -0.112085}

IDCT -
 y[]= { -0.000000    1.000000      2.000000     3.000000}
```

Two-dimensional Discrete Cosine Transform

To compute the two-dimensional DCT of an image, first split the data array into its constituent rows. Next, a one-dimensional DCT is performed on each of the rows. Then another one-dimensional DCT is performed on the columns of the coefficients. Finally, the data is recombined to create the two-dimensional transform matrix.

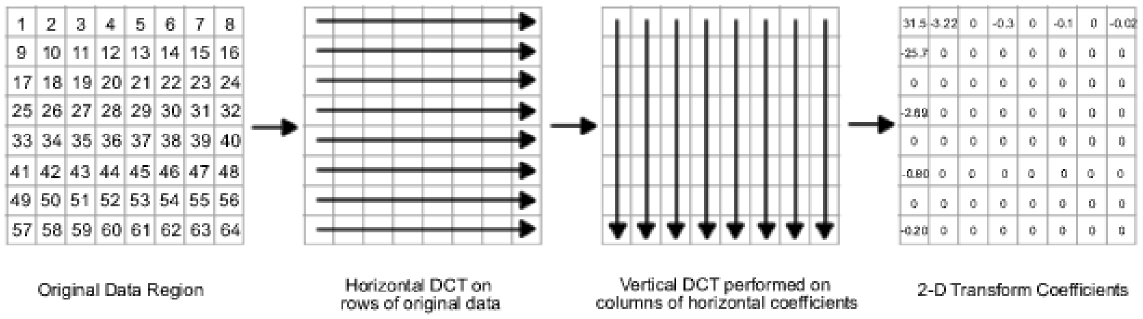

Original Data Region
Horizontal DCT on rows of original data
Vertical DCT performed on columns of horizontal coefficients
2-D Transform Coefficients

Figure B.2: The two-dimensional DCT process on an 8x8 image.

Program B.2 shows a way to implement both the two-dimensional DCT and the two-dimensional inverse DCT. The program uses functions to call a one-dimensional DCT and IDCT multiple times. A one-dimensional DCT is set up to work across the rows. In order to perform the vertical transform, the data values are transposed. Once the vertical transform is performed, the results are transposed back to the original position. The process is similar for the IDCT.

In the following sections (B.2 – B.7) we will present five additional ways of performing a two-dimensional DCT. The method presented in program B. 2 uses two threads, each using functions to perform the operations.

```
/*  2-D Discrete Cosine Transform and Inverse DCT
aCe Program

Maurice F. Aburdene
Andrew J. Marbach
John E. Dorband
____________________________________________________________

DCT formula for N samples in y[]:


Y(k)= 2/N * Gk * (-1)^k * SUM(n=0->N-1){ y(N-1-n) * cos [ k * (n+0.5)
* pi/N ] }

        Gk = 1/2, if k = 0;  Gk = 1, otherwise;
        k= 0,1, ... N-1;
____________________________________________________________

IDCT formula for N samples in Y[]:


y(k)= (-1)^k * SUM (n=0->N-1){ Y(N-1-n) * sin [ (k + 0.5) * (n+1) * pi/N ]
    }
        k= 0,1, ... N-1;
____________________________________________________________
*/
#include <stdio.aHr>
#include <math.aHr>
#define N 4
#define PI   (4*atan(1))
threads y[N][N];
threads Y[N][N];

/*initialize bundle variables*/
y double temp, sum, value=0.0, data=0.0;
y int n;
Y double temp, sum, value=0.0;
Y int n;


generic void dataprint (double input) {

        /* prints out formatted data values*/
        printf("%f\t%c", input,($$ix[0]==(N-1))?'\n':' ');
        MAIN.{printf("\n\n");}
}

generic double inputdata () {

        /*Generates values*/
        double temp;
        temp=$$i;
        return temp;
}

Y double dct (){

        sum=0.0;

        /*scaling factor calculation*/
        value = 2./N;
        value = value*pow(-1,$$i%N);

        /*gamma coefficient calculation*/
        if ($$i%N==0)
             value=value * 0.5;
        /*loop to use all of the samples*/
        for (n=0; n<N; n++){

                /*get y[] data, multiply by cos term    */
                temp=y[$$i/N][N-1-n].(data);
                temp=temp*cos(($$i%N)*(n+.5)*PI/(float)N);

                /*add current iteration to the sum*/
                sum = sum + temp;

        }
        /*final value calculation*/
        value = value * sum;
        return value;

}

y double idct() {
        sum=0.0;
        /*scaling factor calculation*/
        value=pow(-1,$$i);
        /*loop to use all of the sample points*/
        for (n=0; n<N; n++){

                /*get y[] data, multiply by sine value*/
                temp=Y[$$i/N][N-1-n].(value);
                         temp   =   temp   *   sin   ((n+1)*(($$i%N)+.5)*
       PI/(float)N);

                /*add current iteration to the sum*/
                sum = sum + temp;

        }

        /*final value calculation*/
        value = value * sum;
        return value;


}

int main (){

        y.{
                /*initialize, set, and print data*/
                data=inputdata();
                MAIN.{printf("Original Data=\n");}
                dataprint(data);
        }
        Y. {
                /*first dct on rows of data, save as data*/
                y[$$i/N][$$i%N].data=dct();

                /*transpose rows of data*/
                y.{data = y[$$ix[0]][$$ix[1]].data;}

                /*perform second dct on columns*/
                dct();

                /*transpose data and value back*/
                y.{data = y[$$ix[0]][$$ix[1]].data;}
                value=Y[$$ix[0]][$$ix[1]].value;

                /*print out Y.values after DCT_2D*/
                MAIN.{printf("DCT_2D(y)=\n");}
                dataprint(value);
        }
        y. {
                /*perform first idct*/
                Y[$$i/N][$$i%N].value=idct();

                /*transpose values in Y*/
                Y.{value = y[$$ix[0]][$$ix[1]].value;}

                /*perform second idct*/
                idct();

                /*transpose values in Y back*/
                Y.{value = y[$$ix[0]][$$ix[1]].value;}

                /*print out y.values*/
                MAIN.{printf("IDCT_2D(Y)=\n");}
                dataprint(value);
        }
}
```

Program B.2: **A two-dimensional DCT program using 2-two-dimensional bundles calling a bundle specific functions.**

The output of program B.2.

```
Original Data=
0.000000        1.000000        2.000000        3.000000
4.000000        5.000000        6.000000        7.000000
8.000000        9.000000        10.000000       11.000000
12.000000       13.000000       14.000000       15.000000


DCT_2D(y)=
7.500000        -1.577161       -0.000000       -0.112085
-6.308644       0.000000        0.000000        -0.000000
0.000000        -0.000000       -0.000000       0.000000
-0.448342       -0.000000       0.000000        -0.000000


IDCT_2D(Y)=
0.000000        1.000000        2.000000        3.000000
4.000000        5.000000        6.000000        7.000000
8.000000        9.000000        10.000000       11.000000
12.000000       13.000000       14.000000       15.000000
```

Two-Dimensional Discrete Cosine Transform

In the second version of the two-dimensional DCT there is just one two-dimensional bundle, but each thread has two arrays of data. One is used for the original data and one for the transform. This method copies data from one thread to all of the threads as well as uses system constants to access other threads. Look for the $$ix [] calls in the program where the first version used only modulus and division.

```
/*  2-D Discrete Cosine Transform and Inverse DCT
aCe Program

Maurice F. Aburdene
Andrew J. Marbach
John E. Dorband
____________________________________________________________

1D DCT formula for N samples in y[]:


Y(k)= 2/N * Gk * (-1)^k * SUM(n=0->N-1){ y(N-1-n) * cos [ k * (n+0.5)
* pi/N ] }

        Gk = 1/2, if k = 0;  Gk = 1, otherwise;
        k= 0,1, ... N-1;
____________________________________________________________

1D IDCT formula for N samples in Y[]:


y(k)= (-1)^k * SUM (n=0->N-1){ Y(N-1-n) * sin [ (k + 0.5) * (n+1) * pi/N ]
    }
        k= 0,1, ... N-1;
____________________________________________________________
*/

#include <stdio.aHr>
#include <math.aHr>

#define N 4
#define PI   (4*atan(1))

threads Y[N][N];

/*initialize bundle variables*/

generic double inputdata( void ) {
    /*Generates values*/
    return $$i ;
}

generic void dataprint(double input)
    /* prints out formatted data values*/
    printf("%f\t%c", input,($$ix[0]==(N-1))?'\n':' ');
    MAIN.{ printf("\n\n"); }
}

generic double DCTrow ( double IMAGE[N][N] )
{

    double temp, sum=0.0, value;
    int i;


    /*scaling factor calculation*/
    value = 2.0/N;
    value = value*pow(-1,$$ix[0]);


    /*gamma coefficient calculation*/
    if ($$i%N==0)
        value=value * 0.5;
    /*loop to use all of the samples*/
    for (i=0; i<N; i++){

        /*get y[] data, multiply by cos term    */
        temp=IMAGE[$$ix[1]][N-1-i];
        temp=temp*cos(($$ix[0])*(i+.5)*PI/(float)N);


        /*add current iteration to the sum*/
        sum = sum + temp;

    }

    /*final value calculation*/
    value = value * sum;

    return value;
}


generic double IDCTrow ( double IMAGE[N][N] )
{

    double temp, sum=0.0, value;
    int i;

    /*scaling factor calculation*/
    value = pow(-1,$$ix[0]);

    /*loop to use all of the samples*/
       for (i=0; i<N; i++){

        /*get y[] data, multiply by cos term*/
        temp=IMAGE[$$ix[1]][N-1-i];
        temp = temp * sin( (i+1.0) * (($$ix[0])+.5) * (PI/(float)N));

        /*add current iteration to the sum*/
        sum = sum + temp;
    }

    /*final value calculation*/
    value = value * sum;

    return value;
}



void main (){

    Y double image [N][N];
    Y double image2 [N][N];
    Y double value=0.0;
    Y double data;
    Y int i,j;

    /*initialize, set, and print data*/
    Y.{
        data = inputdata();
        dataprint(data);

        /* copies data to all threads */
        for ( i=0; i<N; i++ ){
            for ( j=0; j<N; j++ ){
                image[i][j] = Y[i][j].data;
            }
        }

        /* compute a DCT value for each thread */
        value=DCTrow(image);

        /* transpose values */
        value = Y[$$ix[0]][$$ix[1]].value;

        /* copies data to all threads */
        for ( i=0; i<N; i++ ){
            for ( j=0; j<N; j++ ){
                image2[i][j] = Y[i][j].value;
            }
        }

        /* compute a DCT value for each thread */
        value = DCTrow ( image2 );
        /* transpose values */
        value = Y[$$ix[0]][$$ix[1]].value;
        /* print result of 2D DCT */
        dataprint(value);
    }

    Y.{

        /* copies data to all threads */
        for ( i=0; i<N; i++ ){
            for ( j=0; j<N; j++ ){
                image2[i][j] = Y[i][j].value;
            }
        }

        /* compute an inverse DCT value for each thread */
        data=IDCTrow(image2);
        /* transpose values */
        data = Y[$$ix[0]][$$ix[1]].data;

        /* copies data to all threads */
        for ( i=0; i<N; i++ ){
            for ( j=0; j<N; j++ ){
                image2[i][j] = Y[i][j].data;
            }
        }

        /* compute an inverse DCT value for each thread */
        data=IDCTrow(image2);
        /* transpose values */
        data = Y[$$ix[0]][$$ix[1]].data;
        /* print result of inverse 2D DCT*/
        dataprint(data);
}
```

Program B.3: **A two-dimensional DCT program using 2 two-dimensional bundles calling bundle specific functions.**

The output of program B.3 is similar to program B.2. The first data section is the original data, the second section is the transform coefficients and the final section is the inverse of the transform coefficients.

```
0.000000        1.000000        2.000000        3.000000
4.000000        5.000000        6.000000        7.000000
8.000000        9.000000        10.000000       11.000000
```

```
12.000000        13.000000       14.000000       15.000000

7.500000         -1.577161       -0.000000       -0.112085
-6.308644        0.000000        0.000000        -0.000000
0.000000         -0.000000       -0.000000       0.000000
-0.448342        -0.000000       0.000000        -0.000000

0.000000         1.000000        2.000000        3.000000
4.000000         5.000000        6.000000        7.000000
8.000000         9.000000        10.000000       11.000000
12.000000        13.000000       14.000000       15.000000
```

Two-Dimesnional Discrete Cosine Transform

The third method of performing a two-dimensional DCT uses two bundles, each with two arrays of stored data. This method also used relative addressing.

```
/*  2-D Discrete Cosine Transform and Inverse DCT
aCe Program

Maurice F. Aburdene
Andrew J. Marbach
John E. Dorband
___________________________________________________________________________
    ____

1D DCT formula for N samples in y[]:


Y(k)= 2/N * Gk * (-1)^k * SUM(n=0->N-1){ y(N-1-n) * cos [ k * (n+0.5)
* pi/N ] }

    Gk = 1/2, if k = 0;  Gk = 1, otherwise;
    k= 0,1, ... N-1;
___________________________________________________________________________

1D IDCT formula for N samples in Y[]:


y(k)= (-1)^k * SUM (n=0->N-1){ Y(N-1-n) * sin [ (k + 0.5) * (n+1) * pi/N ]
    }
    k= 0,1, ... N-1;
___________________________________________________________________________
*/

#include <stdio.aHr>
#include <math.aHr>

#define N 4
#define PI   (4*atan(1))

threads Y[N][N];
threads y[N][N];

/*initialize bundle variables*/
```

```
generic double inputdata( void ) {
    /*Generates values*/
    return $$i ;
}

generic void dataprint(double input) {

    /* prints out formatted data values*/
    printf("%f\t%c", input,($$ix[0]==(N-1))?'\n':' ');
    MAIN.{ printf("\n\n"); }
}

generic double DCTrow ( double IMAGE[N][N] )
{

    double temp, sum=0.0, value;
    int i;

    /*scaling factor calculation*/
    value = 2.0/N;
    value = value*pow(-1,$$ix[0]);

    /*gamma coefficient calculation*/
    if ($$i%N==0)
        value=value * 0.5;

    /*loop to use all of the samples*/
    for (i=0; i<N; i++){

        /*get y[] data, multiply by cos term    */
        temp=IMAGE[$$ix[1]][N-1-i];
        temp=temp*cos(($$ix[0])*(i+.5)*PI/(float)N);

        /*add current iteration to the sum*/
        sum = sum + temp;
         }

    /*final value calculation*/
    value = value * sum;

    return value;
}


generic double IDCTrow ( double IMAGE[N][N] )
{

    double temp, sum=0.0, value;
    int i;

    /*scaling factor calculation*/
    value = pow(-1,$$ix[0]);

    /*loop to use all of the samples*/
    for (i=0; i<N; i++){

        /*get y[] data, multiply by cos term */
        temp=IMAGE[$$ix[1]][N-1-i];
        temp = temp * sin( (i+1.0) * (($$ix[0])+.5) * (PI/(float)N));

        /*add current iteration to the sum*/
        sum = sum + temp;
         }

    /*final value calculation*/
    value = value * sum;
    return value;
}

void main (){

    Y double value=0.0;

    /*initialize, set, and print data*/
    Y.{
        double image [N][N];
        double image2 [N][N];
        double data;
        int i,j;

        data = inputdata();
        dataprint(data);

        /* copies data to all threads */
        for ( i=0; i<N; i++ ){
            for ( j=0; j<N; j++ ){
                image[i][j] = Y[i][j].data;
            }
        }

        /* compute a DCT value for each thread */
        value=DCTrow(image);

        /* transpose values */
        value = Y[$$ix[0]][$$ix[1]].value;

        /* copies data to all threads */
        for ( i=0; i<N; i++ ){
            for ( j=0; j<N; j++ ){
                image2[i][j] = Y[i][j].value;
            }
        }

        /* compute a DCT value for each thread */
        value = DCTrow ( image2 );
        /* transpose values */
        value = Y[$$ix[0]][$$ix[1]].value;
        /* print result of 2D DCT */
        dataprint(value);
    }

    y.{
        double image [N][N];
        double image2 [N][N];
        double value=0.0;
        double data;
        int i,j;

        /* copies data to all threads */
        for ( i=0; i<N; i++ ){
            for ( j=0; j<N; j++ ){
                image2[i][j] = Y[i][j].value;
            }
        }

        /* compute an inverse DCT value for each thread */
        data=IDCTrow(image2);
        /* transpose values */
        data = y[$$ix[0]][$$ix[1]].data;

        /* copies data to all threads */
        for ( i=0; i<N; i++ ){
            for ( j=0; j<N; j++ ){
                image2[i][j] = y[i][j].data;
            }
        }

        /* compute an inverse DCT value for each thread */
        data=IDCTrow(image2);
        /* transpose values */
        data = y[$$ix[0]][$$ix[1]].data;
        /* print result of inverse 2D DCT*/
        dataprint(data);
    }

}
```

Program B.4: **A fourth two-dimensional DCT program using two bundles each with two arrays of data.**

The output of program B.4 is the same as program B.3.

2-D Discrete Cosine Transform Version 4

The fourth version of the DCT program uses just one bundle of NXN threads each containing two arrays with N threads. This program uses ‘for’ loops to copy the data from one thread to all threads. Here is program B.5.

```
/*  2-D Discrete Cosine Transform and Inverse DCT
aCe Program

Maurice F. Aburdene
Andrew J. Marbach
John E. Dorband
___________________________________________________________________________

1D DCT formula for N samples in y[]:


Y(k)= 2/N * Gk * (-1)^k * SUM(n=0->N-1){ y(N-1-n) * cos [ k * (n+0.5)
* pi/N ] }

        Gk = 1/2, if k = 0;  Gk = 1, otherwise;
        k= 0,1, ... N-1;
___________________________________________________________________________

1D IDCT formula for N samples in Y[]:


y(k)= (-1)^k * SUM (n=0->N-1){ Y(N-1-n) * sin [ (k + 0.5) * (n+1) * pi/N ]
    }
        k= 0,1, ... N-1;
___________________________________________________________________________
*/

#include <stdio.aHr>
#include <math.aHr>

#define N 4
```

```
#define PI   (4*atan(1))

threads Y[N][N];

/*initialize bundle variables*/

generic double inputdata( void ) {
    /*Generates values*/
    return $$i ;
}

generic void dataprint(double input) {

    /* prints out formatted data values*/
    printf("%f\t%c", input,($$ix[0]==(N-1))?'\n':' ');
    MAIN.{ printf("\n\n"); }
}

generic double DCTrow ( double IMAGE[N] )
{

    double temp, sum=0.0, value;
    int i;

    /*scaling factor calculation*/
    value = 2.0/N;
    value = value*pow(-1,$$ix[0]);

    /*gamma coefficient calculation*/
    if ($$ix[0]==0)
    value=value * 0.5;

    /*loop to use all of the samples*/
    for (i=0; i<N; i++){

    /*get y[] data, multiply by cos term*/
    temp=IMAGE[N-1-i];
    temp=temp*cos(($$ix[0])*(i+.5)*PI/(float)N);

    /*add current iteration to the sum*/
    sum = sum + temp;
}

    /*final value calculation*/
    value = value * sum;

    return value;
}


generic double IDCTrow ( double IMAGE[N] )
{
    double temp, sum=0.0, value;
    int i;

    /*scaling factor calculation*/
    value = pow(-1,$$ix[0]);

    /*loop to use all of the samples*/
    for (i=0; i<N; i++){

        /*get y[] data, multiply by cos term*/
        temp=IMAGE[N-1-i];
        temp = temp * sin( (i+1.0) * ($$ix[0]+.5) * (PI/(float)N));

        /*add current iteration to the sum*/
        sum = sum + temp;
    }

    /*final value calculation*/
    value = value * sum;
    return value;
}

void main (){

    /*initialize, set, and print data*/
    Y.{
        double image [N];
        double image2 [N];
        double value=0.0;
        double data;
        int i,j;

        data = inputdata();
        dataprint(data);

        /* copies data to all threads */
        for ( j=0; j<N; j++ ){
            image[j] = Y[$$ix[1]][j].data;
        }

        /*
        for ( j=0; j<N; j++ ){
            dataprint(image[j]);
        }
        */

        /* compute a DCT value for each thread */
        value=DCTrow(image);

        /* copies transposed data to all threads */
        for ( j=0; j<N; j++ ){
            image2[j] = Y[j][$$ix[1]].value;
        }

        /* compute a DCT value for each thread */
        value = DCTrow ( image2 );
        /* transpose values */
        value = Y[$$ix[0]][$$ix[1]].value;
        /* print result of 2D DCT */
        dataprint(value);

        /* INVERSE DCT */

        /* copies data to all threads */
        for ( j=0; j<N; j++ ){
            image2[j] = Y[$$ix[1]][j].value;
        }

        /* compute an inverse DCT value for each thread */
        data=IDCTrow(image2);

        /* copies transposed data to all threads */
        for ( j=0; j<N; j++ ){
            image2[j] = Y[j][$$ix[1]].data;
        }

        /* compute an inverse DCT value for each thread */
        data=IDCTrow(image2);
        /* transpose values */
        data = Y[$$ix[0]][$$ix[1]].data;
        /* print result of inverse 2D DCT*/
        dataprint(data);
    }
}
```

Program B.5: **A fifth two-dimensional DCT program.**

The output of program B.6 is similar to the output of program B.3.

Two-Dimensional Discrete Cosine Transform

The fifth method uses one bundle with two-two-dimensional arrays of data. Program B.6 also uses relative addressing.

```
/*  2-D Discrete Cosine Transform and Inverse DCT
/*  2-D Discrete Cosine Transform and Inverse DCT
aCe Program

Maurice F. Aburdene
Andrew J. Marbach
John E. Dorband
___________________________________________________________________________
1D DCT formula for N samples in y[]:


Y(k)= 2/N * Gk * (-1)^k * SUM(n=0->N-1){ y(N-1-n) * cos [ k * (n+0.5)
* pi/N ] }

    Gk = 1/2, if k = 0;  Gk = 1, otherwise;
    k= 0,1, ... N-1;
___________________________________________________________________________

1D IDCT formula for N samples in Y[]:


y(k)= (-1)^k * SUM (n=0->N-1){ Y(N-1-n) * sin [ (k + 0.5) * (n+1) * pi/N ]
    }
    k= 0,1, ... N-1;
___________________________________________________________________________
*/

#include <stdio.aHr>
#include <math.aHr>

#define N 4
#define PI   (4*atan(1))

threads Y[N][N];

/*initialize bundle variables*/

generic double inputdata( void ) {
    /*Generates values*/
    return $$i ;
}

generic void dataprint(double input) {
    /* prints out formatted data values*/
    printf("%f\t%c", input,($$ix[0]==(N-1))?'\n':' ');
    MAIN.{ printf("\n\n"); }
}

generic double DCTrow ( double IMAGE[N] )
{

    double temp, sum=0.0, value;
    int i;


    /*scaling factor calculation*/
    value = 2.0/N;
    value = value*pow(-1,$$ix[0]);

    /*gamma coefficient calculation*/
    if ($$ix[0]==0)
        value=value * 0.5;


    /*loop to use all of the samples*/
    for (i=0; i<N; i++){

        /*get y[] data, multiply by cos term    */
        temp=IMAGE[N-1-i];
        temp=temp*cos(($$ix[0])*(i+.5)*PI/(float)N);

        /*add current iteration to the sum*/
        sum = sum + temp;
    }

    /*final value calculation*/
    value = value * sum;

    return value;
}


generic double IDCTrow ( double IMAGE[N] )
{

    double temp, sum=0.0, value;
    int i;

    /*scaling factor calculation*/
    value = pow(-1,$$ix[0]);

    /*loop to use all of the samples*/
    for (i=0; i<N; i++){

        /*get y[] data, multiply by cos term*/
        temp=IMAGE[N-1-i];
        temp = temp * sin( (i+1.0) * ($$ix[0]+.5) * (PI/(float)N));

        /*add current iteration to the sum*/
        sum = sum + temp;
         }

    /*final value calculation*/
    value = value * sum;

    return value;
}


void main (){
        /*initialize, set, and print data*/
         Y.{
        double image [N];
        double image2 [N];
        double value=0.0;
        double data;
        int i,j;

        data = inputdata();
        dataprint(data);

        /* copies data to all threads */
        for ( j=0; j<N; j++ ){
            image[j] = Y[$$ix[1]][j].data;
        }

        /*
        for ( j=0; j<N; j++ ){
            dataprint(image[j]);
        }
        */

        /* compute a DCT value for each thread */
        value=DCTrow(image);
        dataprint(value);

        /* transpose values */
        value = Y[$$ix[0]][$$ix[1]].value;

        /* copies data to all threads */
        for ( j=0; j<N; j++ ){
            image2[j] = Y[$$ix[1]][j].value;
        }

        /* compute a DCT value for each thread */
        value = DCTrow ( image2 );
        /* transpose values */
        value = Y[$$ix[0]][$$ix[1]].value;
        /* print result of 2D DCT */
        dataprint(value);

        /* INVERSE DCT */

        /* copies data to all threads */
        for ( j=0; j<N; j++ ){
            image2[j] = Y[$$ix[1]][j].value;
        }

        /* compute an inverse DCT value for each thread */
        data=IDCTrow(image2);
        /* transpose values */
        data = Y[$$ix[0]][$$ix[1]].data;

        /* copies data to all threads */
        for ( j=0; j<N; j++ ){
            image2[j] = Y[$$ix[1]][j].data;
        }

        /* compute an inverse DCT value for each thread */
        data=IDCTrow(image2);
        /* transpose values */
        data = Y[$$ix[0]][$$ix[1]].data;
        /* print result of inverse 2D DCT*/
        dataprint(data);
    }
}
```

Program B.6: **A fifth two-dimensional DCT program using one bundle, with two-two-dimensional arrays per thread..**

The output of program B.6 is similar to program B.3.

More Applications

Parameter Passing

In this chapter, more applications using aCe are presented. First we present a program that uses a call by reference routine.

```
/*  Call by Reference
aCe Program

Maurice F. Aburdene
Andrew J. Marbach
*/

#include <stdio.aHr>
#define N 4
threads A[N];

generic void modify (generic(OTHER),OTHER int *b){

    /*modify *b, and print*/
    OTHER.{
        *b=$$i+1;
    }
}

int main () {

    A int a=0;          /*initialize a to 0*/
    A.{printf("%d\t",a);}  /*print a*/
    printf("\n");

    modify(A,&a);       /*modify a in function, call by reference*/

    printf("\n");       /*print modified a*/
    A.{printf("%d\t",a);}
}
```

Program C.1: **Uses a function to modify a passed parameter.**

This program is pretty straight forward. The `modify` routine is passed a value by reference and then modifies it. Since the parameter is passed by reference, the value is changed in the main program as well. The output for program C.1 is below.

```
0         0         0         0
1         2         3         4
```

BMP Transformations

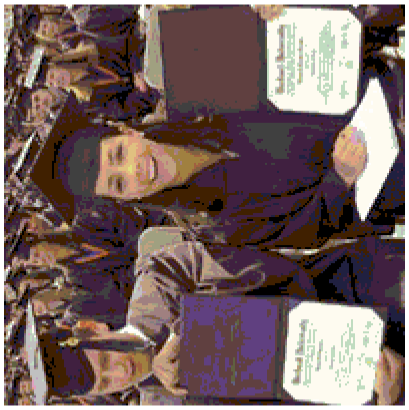
This application takes a BMP image file, reads the data into bundle variables. The image is transformed and then printed to a new file. The transformation function uses universal addressing. An array of chars are used to hold the header information, while the transformation is taking place on the rest of the data. This program works with 8-bit (256 color) bitmaps. Char's are used to store the information because they store 8-bits of information.

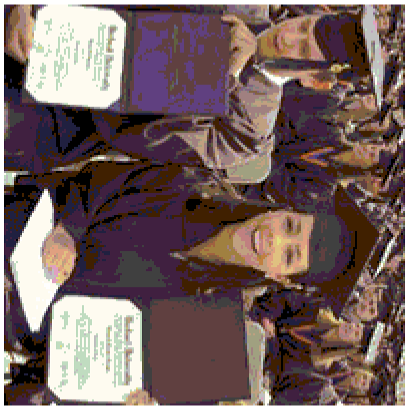
```
/*BMP Transformation Program
aCe program

Andrew J. Marbach
7.01.03
*/

#include <stdio.aHr>
#include <stdlib.aHr>
#include <math.aHr>
#define X 200
#define COLORS 256

threads Y[X][X];

generic void charprint(unsigned char input) {

    /* prints out formatted data values*/
    printf("%i\t%c", input,($$ix[0]==(X-1))?'\n':' ');
    MAIN.{printf("\n\n");}
}

generic char rotateccw (generic(Y), Y unsigned char value) {

    /*rotates the data stored in the bundle counter-clockwise*/
    unsigned char temp;
    temp=Y[X-$$ix[0]-1][$$ix[1]].value;
    return temp;
}

generic char rotatecw (generic(Y), Y unsigned char value) {

    /*rotates the data stored in the bundle counter-clockwise*/
    unsigned char temp;
    temp=Y[$$ix[0]][X-$$ix[1]-1].value;
    return temp;
}

generic char horizflip (generic(Y), Y unsigned char value) {

    /*flips the data stored in the bundle horizontally*/
    unsigned char temp;
    temp=Y[$$ix[1]][X-$$ix[0]-1].value;
    return temp;
}

generic char vertflip (generic(Y), Y unsigned char value) {

    /*flips the data stored in the bundle vertically*/
    unsigned char temp;
    temp=Y[X-$$ix[1]-1][$$ix[0]].value;
    return temp;
}


int main (){

    /*initialize variables*/
    FILE *fid;
    Y unsigned char image, vert, horiz, cw, ccw;
    unsigned char header[(54+COLORS*4)];

    /*open original image file*/
    fid=fopen("grads.bmp","r+");

    /*reader header information*/
    fread(&header, (54+4*COLORS),1,fid);

    Y.{
        /*Open original image, load in to image*/
        fread(&image,sizeof(image),1,MAIN.fid);
        charprint(image);

        /*perform transformations*/
        cw=rotatecw(Y,image);
        ccw=rotateccw(Y,image);
        vert=vertflip(Y,image);
        horiz=horizflip(Y,image);
    }

    fclose(fid);

    /*write the horizontally fliped image to a file*/
    fid=fopen("gradshoriz.bmp","w+");
    fwrite (header,sizeof(header),1,fid);
    Y.{fwrite(&horiz,sizeof(horiz),1,MAIN.fid);}
    fclose(fid);

    /*write the vertically fliped image to a file*/
    fid=fopen("gradsvert.bmp","w+");
    fwrite (header,sizeof(header),1,fid);
    Y.{fwrite(&vert,sizeof(vert),1,MAIN.fid);}
    fclose(fid);

    /*write the clockwise rotated image to a file*/
    fid=fopen("gradscw.bmp","w+");
    fwrite (header,sizeof(header),1,fid);
    Y.{fwrite(&cw,sizeof(cw),1,MAIN.fid);}
    fclose(fid);

    /*write the couter-clockwise image to a file*/
    fid=fopen("gradsccw.bmp","w+");
    fwrite (header,sizeof(header),1,fid);
    Y.{fwrite(&ccw,sizeof(ccw),1,MAIN.fid);}
    fclose(fid);

    return 0;
}
```

Program C.2: **Reads in a BMP image file, transforms the image, writes transformed images to a file.**

Sample input and output:

grads.bmp





gradsccw.bmp gradscw.bmp

gradshoriz.bmp

gradsvert.bmp

BMP and Discrete Cosine Transform

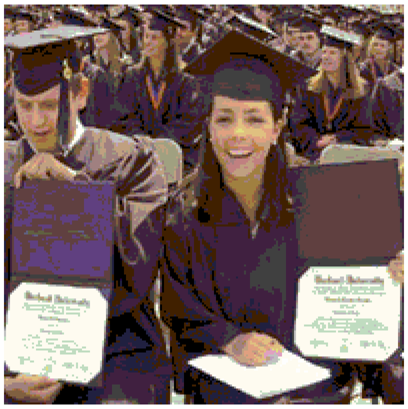
This application combines the use of BMP files as in the previous program, but also adds in DCT functionality. Program C.3 uses a cutoff to choose at what point the transform coefficients should be dropped. While this algorithm does not save space, dropping the coefficient shows how lossy the image would be when neglecting coefficients below the cutoff.

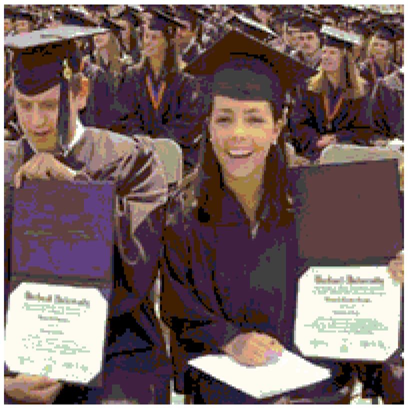
```
/* BMP & DCT-IDCT Program
aCe program

Andrew J. Marbach
7.1.03
*/

#include <stdio.aHr>
#include <stdlib.aHr>
#include <math.aHr>
#define X 200
#define COLORS 256
#define PI (4*atan(1))
#define CUTOFF .001
threads Y[X][X];
FILE *fid;


generic void charprint(unsigned char input) {

    /* prints out formatted data values*/
    printf("%i\t%c", input,($$ix[0]==(X-1))?'\n':' ');
    MAIN.{printf("\n\n");}
}
```

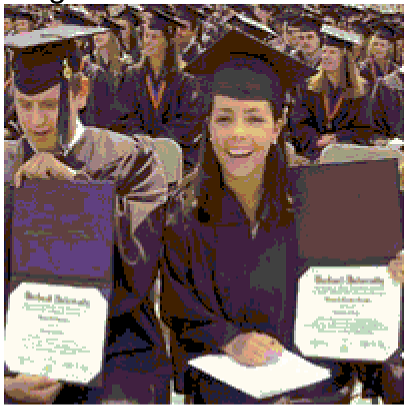
```
generic double DCTrow (generic(Y), Y double value)
{
    double temp, sum=0., value;
    int n;

    /*scaling factor calculation*/
    value = 2./X;
    value = value*pow(-1,$$i%X);

    /*gamma coefficient calculation*/
    if ($$i%X==0)
         value=value * 0.5;

    /*loop to use all of the samples*/
    for (n=0; n<X; n++){

        /*get y[] data, multiply by cos term    */
        temp=Y[$$i/X][X-1-n].value;
        temp=temp*cos(($$i%X)*(n+.5)*PI/(float)X);

        /*add current iteration to the sum*/
         sum = sum + temp;

    }
    /*final value calculation*/
    value = value * sum;
    return value;
}

generic double IDCTrow(generic(Y), Y double value) {

    double temp, sum=0., value;
    int n;

    /*scaling factor calculation*/
    value=pow(-1,$$i);

    /*loop to use all of the sample points*/
    for (n=0; n<X; n++){

        /*get y[] data, multiply by sine value*/
        temp=Y[$$i/X][X-1-n].(value);
            temp = temp * sin((n+1)*(($$i%X)+.5)*PI/(float)X);

        /*add current iteration to the sum*/
            sum = sum + temp;

    }

    /*final value calculation*/
    value = value * sum;
    return value;
}


int main (){

    Y double transform, inverse=0;
    Y unsigned char image, outchar;
    Y unsigned int x;

    unsigned char rows, columns;
    unsigned char header[(54+COLORS*4)];

    fid=fopen("grads256.bmp","r+");
    fread(&header, (54+4*COLORS),1,fid);
    Y.{

        /*Open original image load in to image*/
        fread(&image,sizeof(image),1,MAIN.fid);

        /*perform two dimensional DCT, with transposition*/
        transform = DCTrow(Y,(double)image);

        transform = Y[$$ix[0]][$$ix[1]].transform;
        transform = DCTrow(Y,transform);
        transform = Y[$$ix[0]][$$ix[1]].transform;


        /*drop insignificant transform coefficients*/
        if (fabs(transform) < CUTOFF)
            transform = 0.;

        /*perform two dimensional IDCT, with transposition*/
        inverse = IDCTrow(Y,transform);

        inverse = Y[$$ix[0]][$$ix[1]].inverse;
        inverse = IDCTrow(Y,inverse);
        inverse = Y[$$ix[0]][$$ix[1]].inverse;

        /*type cast inverse*/
        x=inverse+.5;
    }

    fclose(fid);
    /*openoutputfile, save header*/
    fid=fopen("newgrads256.bmp","w+");
    fwrite (header,sizeof(header),1,fid);

    Y.{
        outchar=x;
        /*write recreated image to a new file*/
        fwrite(&outchar,sizeof(outchar),1,MAIN.fid);
    }

    fclose(fid);

    return 0;
}
```

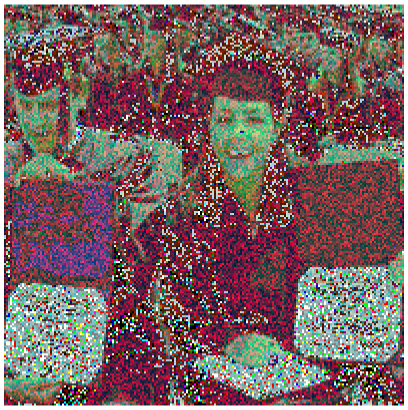
Program C.3: **Reads in a BMP image file, performs DCT, removes some coefficients, performs IDCT, writes new BMP file.**

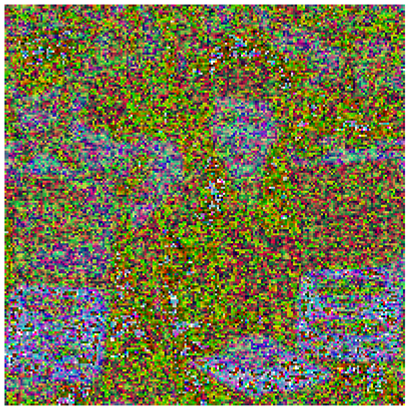
Sample outputs with various transform coefficient CUTOFF values:

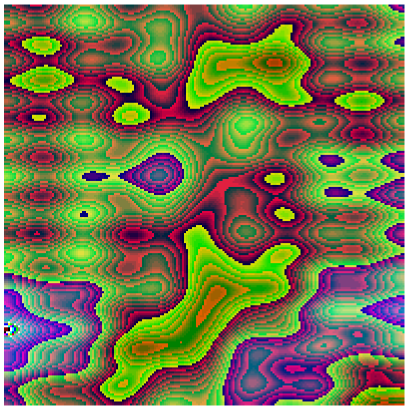

Original – CUTOFF = 0


CUTOFF = .001


CUTOFF = .01


CUTOFF = .1


CUTOFF = 1


CUTOFF = 10

References

Sources used for Most of the Tutorial (not cited)

[A] Dorband, John E. Architecture-Adaptive Computing Environment. NASA GSFC. http://newton.gsfc.nasa.gov/aCe/. (31 July, 2002).

[B] Dordband, John E. and Aburdene, Maurice F. Architecture-Adaptive Computing Environment: A Tool for Teaching Programming, Frontiers in Education, 2002. FIE 2002. 32nd Annual, Volume 3, 2002. pg.. 247-252.

Other Sources Cited in the Tutorial

[1] Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pg. 94, 1972.

[2] Iverson, K.E., A Programming Language, Wiley, New York, 1962.

[3] DAP-FORTRAN Language, International Computers Ltd., TP 6918.

[4] Reeves A.P., Bruner J.D., The Language Parallel Pascal and other Aspects of the Massively Parallel Processor, School of Electrical Engineering, Cornell University, December, 1982.

[5] Dorband, J.E., "MPP Parallel Forth", Proceeding of the First Symposium on the Frontiers of Massively Parallel Scientific Computation, pg. 211-215, 1986.

[6] Rose, J., Steele, G., "C*: An Extended C Language for Data Parallel Programming", Presented at the Second International Conference on Supercomputing, May, 1987.

[7] Steele, G., Wholey, S., "Connection Machine Lisp: A Dialect of Common Lisp for Data Parallel Programming," August, 1987.

[8] Hamet, L.E., Dorband, J.E., A Generic Fine-grained Parallel C, Proceedings of the Second Symposium on the Frontiers of Massively Parallel Computation, October 1988, Fairfax, VA, pg. 625-628.

[9] MPL (MasPar Programming Language) Reference Manual, MasPar Computer Corp., Pt No. 9300-9034-00 Rev A2.