aCe C
Language Reference

John E. Dorband
NASA Goddard Space Flight Center
Greenbelt, MD 20771

Introduction

This document is an introduction to the language aCe C. aCe stands for architecture-adaptive computing environment. aCe C is a superset of ANS C. In this document, the term aCe will be used synonymously for aCe C. This description of aCe C assumes that the reader is knowledgeable of ANS C. The purpose of aCe is to facilitate the writing of parallel programs. This is done by allowing the programmer to explicitly describe the parallelism of an algorithm. The concepts behind aCe C have been gleaned from such parallel languages as APL[3], DAP Fortran[7], Parallel Pascal[4], Parallel Forth[1], C*[5], CM Lisp[6], FGPC[2], and MPL[8].

Typically, a C program has one thread of execution. This is the path that a computer takes through a program while executing it. 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. aCe 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.

Basics

The following is a valid aCe program. Note that it looks no different from a standard C program:

```
int main ( int argc, char **argv ) {
        printf("Hello World\n");
}
```

The difference is that C has only one thread of execution. aCe, however, may have many threads of execution. Each thread may be referenced by name and index. The 'Hello World' program's thread is implicitly named 'MAIN'. This is important when it is necessary to communicate between the 'MAIN' thread and other threads executing concurrently with the 'MAIN'.

Parallelism in aCe is expressed by first defining a set of concurrently executable threads. A group of parallel threads can be viewed as a bundle of executing threads, a cluster of processes, or an array of processors. These three views will be treated synonymously. In aCe, a bundle of threads is defined with the 'threads' statement. The following statement only declares the intent of the programmer to use 10 concurrent threads of executions named 'A' at some point in his code:

```
threads A[10];
```

These threads must be assigned storage before they can execute any code. Each thread will have its own private storage. Variables of a thread can not be accessed directly by any other thread. In aCe, there is no global storage, only storage local to each thread. Storage is declared for a thread by a standard C declaration preceded by the thread's name. All threads of a bundle will be allocated space for any given declaration. The following statement allocates an integer 'aval' for each of the 10 threads previously defined as bundle 'A':

```
A int aval;
```

Once storage has been assigned to a thread, then code may be written that will be executed by the thread. The following is a simple piece of code that adds the ten values of val2 to the ten values of val3 and stores the ten results in the ten locations of val1:

```
threads A[10];
A int val1,val2,val3;
int main () {
        A.{
                val1 = val2 + val3 ;
        }
}
```

Granted, the values of val2 or val3 were never initialized, but that is a different issue. The important point here is that execution started with the function 'main' by the default thread 'MAIN', which transferred control to (forks) the 10 threads of 'A' to add the values of val2 to the values val3 before returning control to 'MAIN'. For parallelism to be useful, the storage of each thread must contain different values. This can be done by different means: 1) read different values into the storage of each thread, 2) copy a built-in value that is unique to each thread into the storage of the thread, or 3) obtain a unique value from another thread. In the first, case the function 'fread' may be used to read values into the storage of a thread.

```
A.{ fread(&val2,sizeof(val2),1,file); }
```

The above fread in the context of the threads of A will read 10 values from the input file and put them in the 10 locations of val2 of the 10 threads of A. The second way of putting a unique value into each of the 10 locations of val2 would be to assign a built in value to val2.

```
A.{  val2 = $$i ;  }
```

The above statement will assign the value of the built-in value '$$i' to val2. The value of '$$i' is the index of the thread. In the case of A, each thread will have a unique index from 0 to 9.

Previously, it was pointed out that a thread only has direct access to storage local to itself. A thread, however, can access storage of another thread indirectly through communication operations. There are two basic communication operations. Under other parallel programming paradigms they are referred to as 'get' and 'put' operations. The following is an example of an aCe 'get' operation:

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

In this example, the value of 'b' is fetch from one thread of A to another thread of A. Remember that the value of $$i is the index of the executing thread and that $$N is the number of threads in the bundle A. Thus the value of (($$i+1))%$$N) is the index of a thread other than the thread performing the 'get' operation. The communication operation uses this value to determine which thread to fetch the value of 'b' from. The following statement is an example of an aCe 'put' operation:

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

In this example, the value of 'a' of the current thread is stored into 'b' of a different thread of A.

In summary, aCe allows the programmer to declare a bundle of parallel threads of execution, allocate the storage of each thread, define code to execute on threads concurrently, and move values between threads. These are the four essential concepts of aCe: execution, storage, code, and communication.

Bundles of threads

In the previous section, the bundle A was declared as a one-dimensional array of threads. A bundle may actually be declared with any number of dimensions. The

following statement declares the bundle B to have three dimensions of sizes 2, 7, and 20 respectively and contain 280 threads:

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

One may also declare bundles of bundles of threads. In the statement below, 'e' is a bundle of bundles of threads. 'e' consists of 100 bundles, each containing 2 bundles, one with 10 threads and the other with 20 threads. One should view each bundle of 'e' as 1 e-thread, 10 c-threads, and 20 d-threads, where the e-thread is the parent thread of the 10 c-threads and 20 d-threads. Thus, there are a total 3,100 threads defined by the following statement:

```
threads { c[10],  d[20]  } e[100] ;
```

Because the definition of a bundle is recursive, a bundle can contain any number of sub-bundles. Note, all bundles are 'descendents' of the bundle 'MAIN'. The primary bundle of a bundle declaration, such as 'e', is an immediate child bundle of 'MAIN'. The following example is a more complex definition meant only to demonstrate the recursive nature of a bundle declaration:

```
threads { s[11], { { u[34],  v[3]  } w[102]  } t[7] [7]  } z[100] ;
```

Execution

All operations that can be performed by the thread, 'MAIN' (i.e., any C code), may be performed by any bundle of threads concurrently. The only operation supported by ANS C, but not by aCe C is 'goto'.

All code that is not labeled with the name of a bundle, by default, will be executed by the lone thread 'MAIN'. The program entry point routine 'main' is run by the thread 'MAIN'. To start code running on a bundle of threads other than 'MAIN', a compound statement must be labeled with the name of that bundle. A compound statement is code enclosed in braces, {}. A compound statement can contain any valid aCe C code. The following code is a compound statement labeled with the bundle name B:

```
B.{ a=b; }
```

This code copies the value at location 'b' to the location 'a', assuming 'a' and 'b' are declared storage locations associated with the bundle B. However, if a conditional statement is executed from within the following labeled compound statement:

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

Some threads of 'B' will copy b to a and some will not, depending on whether 'z' was true or not, respectively. 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’ will execute the code, while all inactive threads will remain idle. Initially, all threads of all bundles are active, and each bundle will only execute code explicitly designated for that bundle. Conditional statements are used to make some threads of a bundle inactive for a portion of code.

A labeled compound statement creates an execution context in which a bundle of threads may execute code. These execution contexts may be nested. In the following example, all threads of B will copy ‘b’ to ‘a’, then all threads of A will copy ‘c’ to ‘d’, and finally all threads of ‘B’ will copy ‘y’ to ‘x’:

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

This would be equivalent to the following statement:

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

Then why would it be necessary for contexts to be nestable? It is only necessary for contexts to be nestable if the execution of one context can implicitly affect the execution of the other. This can happen if the contained context is contained within a condition statement of the containing context, as in the following statement:

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

In this example, if ‘a’ is true for <u>any</u> thread of ‘B’, then ‘y’ is copied to ‘x’ for <u>all</u> threads of ‘A’. Otherwise, if ‘a’ is false for <u>all</u> threads of ‘B’, then ‘y’ will not be copied to ‘x’ for <u>any</u> thread of ‘A’. Putting it another way, if any thread of B is active within the context of A, then all threads of A will be active. The next example is a little more complicated.

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

As with the previous example, ‘x’ is copied to ‘y’ only if ‘a’ is true for at least one thread of B. And ‘t’ is copied to ‘s’ if ‘a’ is false for at least one thread of B. An interesting side effect of this statement is that ‘x’ will be copied to ‘y’ 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.

Routines in aCe must be declared as to which bundle may call them. This is done by preceding the routine’s declaration with the name of the bundle.

```
B int aroutine( int c, int b) { ... code ... }
```

The preceding routine, aroutine, may called within the context of any code executed by ‘B’.

Communication

A conditional expression can also affect communications. Only active threads can initiate a get or put operation. In the following statement, only threads of B for which 'a' is true actually fetch a value from the storage location 'x' of a thread of A:

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

Again, only active threads of 'B' will fetch (or get) values from 'A'. And in the following statement, only active threads of 'B' will store (or put) values into threads of A.

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

Threads of 'A' need not be active to be fetched from or have their storage modified. The following is an example of this:

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

Some threads of 'A' are made inactive; yet, it would execute no differently than the previous example.

More on Concurrent Conditional Execution

As has already been pointed out, one can view conditional execution as the deactivation of threads during conditional code that does not apply, rather than as the action of skipping the execution of code that does not apply. Therefore, 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 the threads are reactivated, as in the case of `switch` structures or `loop-continue` combinations. A conditional statement only applies to the threads that were active when the conditional statement was entered.

The `if` statement may deactivate some of the active threads until the corresponding `else` is reached; then the original threads are restored and those that had been active are deactivated. A loop structure (`for`, `while`, and `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 only until the current iteration of loop has been completed. A `return` statement deactivates all active threads until all threads that entered the routine return or complete the routine.

A continue and break statement in C applies specifically to the control structure that includes it. This applies also to aCe C; in addition, however, the control structure must be executing within the same bundle context as the continue or break statement.

Similarly, return statements must belong to the same bundle context as the calling routine.

Previously it was pointed out that a condition statement executed in one bundle could implicitly affect the threads of another independent bundle. This can happen between threads with common ancestry as well. A parent thread controls its children, and children threads affect their parent. If a thread becomes inactive, all of its children become inactive. And if all the children threads of a parent thread become inactive, the parent thread also becomes inactive.

Thread Identification

Threads within a bundle are identified by system-defined constants. There are two categories under which a thread can be identified: 1) globally among all threads of a bundle and 2) within a dimension of a sub-bundle. There are three variables for each of these categories: 1) the number of threads within the category, 2) the log of the number of threads within the category (if the number is a power of 2), and 3) the sequential identifier of the thread within the category. The following are the definitions of the system constants:

`$$Name` - the name of the bundle of threads (char*).

`$$D` - the number of dimensions of the bundle.

`$$N` - the total number of threads in the bundle (over all sub-bundles).

`$$L` - the log base 2 of the total number of threads in the bundle (equals -1 if $$N is not a power of 2).

`$$i` - the thread identifier of the thread with respect to all the threads of the 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.

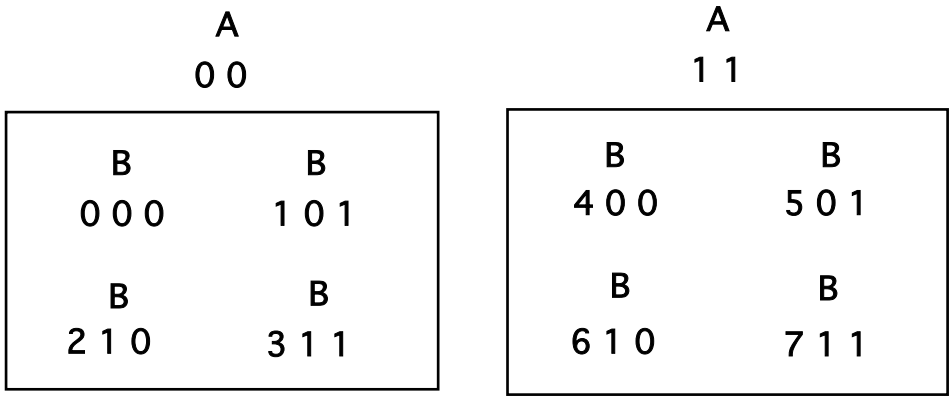
`$$Lx[d]` - the log base 2 of the number of threads of d-th dimension of the sub-bundle (equals -1 if $$Nx[d] is not a power of 2).

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

The following example uses the bundle description:

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

A has 2 threads. In the diagram below, there is a pair of numbers under each **A**. The first number is its value for **$$i,** and the second is its value for **$$ix[0]**. There is a triplet under each thread of **B**. The first number is its value of **$$i**, the second is its value of **$$ix[1]**, and the third is its value of **$$ix[0]**. Note that the threads of 'B' form two sub-bundles, one under A[0] and another under A[1]. A sub-bundle consists of all the threads of a bundle that have the same parent thread (contained in the parent bundle.) For example, 'A' is the parent bundle of 'B' and B[4], B[5], B[6], and B[7] make up a sub-bundle of 'B' whose parent thread is A[1] of bundle 'A' .



The value of the other system constants of **A** are:

$$NAME = "A", $$D = 1, $$N=2, $$L=1,
$$Nx[0]=2, $$Nx[1]=<undefined>,
$$Lx[0]=1, and $$Lx[1]=<undefined>.

The values of the other system constants of **B** are:

$$NAME = "B", $$D = 2, $$N=8, $$L=3,
$$Nx[0]=2, $$Nx[1]=2,
$$Lx[0]=1, and $$Lx[1]=1.

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. I/O in aCe is in order of thread identifier, as in the following example:

```
threads CLX3[1000]
CLX3.{ int A; A=$$i; fwrite(&A,sizeof(int),1,FILEptr); }
```

The thousand values A will be written to the file whose descriptor is located at `FILEptr`. If the above code is contained within a conditional statement, only a subset of the values of A, corresponding to the active threads, will be written to the file.

```
threads CLX3[1000]
CLX3.{ int A; A=$$i;
         if ($$i<4 || $$i>995 )
                fwrite(&A,sizeof(int),1,FILEptr);
}
```

The above code segment will write eight values into the file, four from the first four threads of CLX3 and four from the last four threads of CLX3 in that order.

aCe has another form of I/O, it is 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 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 fastest form of I/O available on the architecture, which may differ from architecture to architecture.

Communications Path Descriptions

Communication between two threads is defined by a communication expression. A communication expression consists of two parts: the *communication path description* or *router expression,* and *the remotely evaluated numeric expression*. In the case of a fetch, the *remotely evaluated numeric expression* must evaluate to a value, while in the case of a send operation, it must evaluate a remote address. The router expression is used to define many concurrent communication paths from threads of one bundle to the threads of another, or even to the same bundle.

In the following example, "`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 threads of type `A` to which the values received from the threads of type B are stored.

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

The threads of B need not be currently active to evaluate the expression `(s+1).` However, a thread of B does need to be a thread that will be fetched from for the expression to be evaluated. Though a thread may have mainly threads fetching from it, the expression will only be evaluated once. In effect, this technique can be used to temporarily reactivate threads.

The router expression describes the path between the remote execution context and the local execution context. If the router expression is the source of a value (a get or fetch operation), the remote context computes a value, and that value is fetched by the local context from the remote thread that is described by the router expression. The previous example demonstrates communications between two single-thread bundles. When multi-thread bundles communicate, each thread of the local context computes an address of a remote thread. This address is based on the router expression. If we modify the above example, we see below that each thread of the local bundle `A` fetches from the corresponding thread of the remote bundle `B`. Note that the system constants `$$i`, which belong to the threads of `A`, is be used to cause the threads of `A` to fetch from the corresponding threads of `B`. This can be viewed as explicitly computing relative thread addresses.

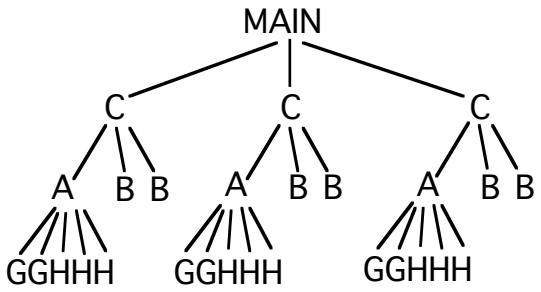
```
threads A[16]; threads B[16];
A.{  int a;  B int b;   a = B[$$i].(b+1);  }
```

Values may be fetched from existing, and yet not necessarily active, threads. If an attempt is made to fetch from a non-existent thread, the result is undefined, possibly due to an incorrectly computed thread identifier. If values are to be fetched from inactive threads, they are temporarily made active. In general, a get operation 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 no matter what the active state of the remote context was prior to the communication. In the above example, `(b+1)` is the remote context for the get operation. In the following example, the router expression will generate invalid addresses for some local threads, therefore a conditional is used to make sure that only valid thread addresses are fetched.

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

There are four types of path descriptions: *absolute* addressing, *universal* addressing, *relative* addressing, and *reduction* addressing ***Absolute*** addressing assumes the path description starts at **MAIN** and proceeds down the tree to the remote thread. Given:

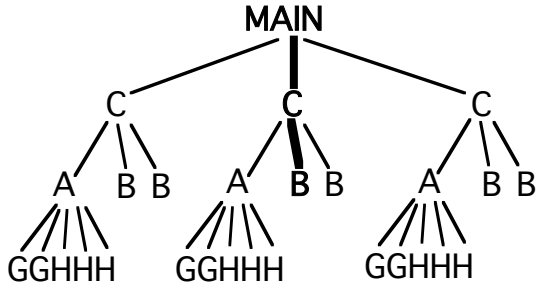
```
threads { { G[2], H[3] } A[1], B[2] } C[3] ;
```



To locate the value of **(b+1)** referenced in the statement below we traverse the graph from **MAIN** to the **x**th thread of **C**. From this node, we then proceed to the **y**th thread of **B**. Note that **x** and **y** may have a different value for each thread of **H** and that the path for each thread of **H** is to a different thread of **B**.

```
H.{  int a,x,y;  B int b;   a = C[x].B[y].(b+1);  }
```

The following diagram shows the highlighted path for an thread of **H** where **x=1** and **y=0**:



Universal addressing is like absolute addressing but need not start at a child of **MAIN**. Universal addressing treats all threads of a bundle as if they were contained in a single one-dimensional bundle. For example, in the following code, threads of **G** are children of threads of **K**, which are children of threads of **J**:

```
threads { { G[3][4] } K[5][7] } J[3][6][7] ;
J.{  int a,x;  G int b;   K[x].G[1][2].b = a;  }
```

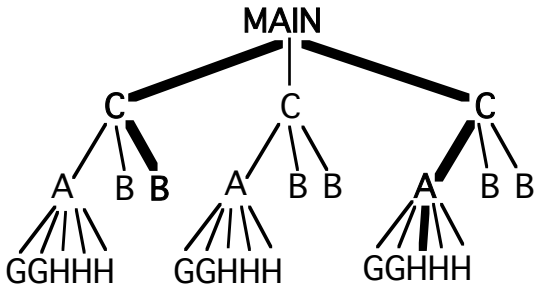
Data of threads of J need to be sent to threads of G, which are not children of that thread of J. However, each thread of type J contains the value of the ID, x, of the thread of K that is the parent of the thread of G where the data is to be sent. Thus, the value of x is used as the index into the one-dimensional array of threads of K as the starting point of the path. The rest of the path is treated as if it were absolute addressing.

Relative addresses are computed with respect to the local thread's and/or its parent's address. If a non-parent is in the path description, the address calculation with respect to the non-parent will be calculated as an absolute address.

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. The following is an example of a path from a thread of H to a thread of B.

```
H.{  int a,x,y;  B int b;   a = .A.C[x].B[y].(b+1);  }
```

The following diagram shows the highlighted path to fetch a remote value from a thread of B (C[0].B[1]) to a local thread of H (C[2].A[0].H[1]) using relative addressing where x=1 and y=1:



The address of C[x] is calculated relative to the parent bundle of the H thread modulo the number of threads of C. Yet, B[y] is treated as an absolute address rather than relative since B is not a parent of H. The bundle A specification does not have an array index because it is not needed. Parents of the local thread do not need indices if they are specifying the path up the bundle hierarchy toward MAIN. Relative path calculations are done modulo the size of the dimensions of the sub-bundle, thus toroidally connecting the sub-bundle's threads. Actually, it was unnecessary to specify the bundle A in the router expression since C is a grandparent of H. The expression could have been written, ".C[x].B[y]".

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. It 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.

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

In the above example note the expression "**a=B.b**". Although **B** is a two-dimensional thread array, 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 ID 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 that being fetch by **H** and assigned to **a**.

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

Global reductions can be performed by this addressing mode by replacing the **=** with operations such as **+=**. In this case, the values **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 (>?=).

Put and Get

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 get from the remote threads. If the expression is on the left side of an assignment, it is a put to the remote threads.

The following is an example of a ***get operation***:

```
H.{  int a,x,y;  B int b;   a = .A.C[x].B[y].b ;  }
```

Reversing the sides of assignment makes it a ***put operation***:

```
H.{  int a,x,y;  B int b;   .A.C[x].B[y].b = a;  }
```

The value of **a** in **H** is sent to the specified remote thread **b** of **B**. If a reduce add operation such as,

```
H.{  int a,x,y;  B int b;   .A.C[x].B[y].b += a;  }
```

is performed and multiple values are sent to the same thread, they are summed together. Since more than one thread can send to the same 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 (-=), multiplication (*=), division (/=), and (&=), or (|=), exclusive-or (^=), minimum (<?=), and maximum (>?=).

A put 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.

```
H.{  int a,x,y; B int b; flag=(.A.C[x].B[y].b = a) ;   }
```

In the above example, the values of **a** in bundle **H** are being sent to the variables **b** in bundle **B**. The value of the flag, after the values have been sent, is TRUE if the specific value of **a** actually reached the requested instance of **b** and FALSE if it did not. There are two reasons a value may not reach its destination: 1) the address of the destination thread is not a valid one, or 2) the put operation is '='. If the put operation is '=', at most one value will be received by any destination thread. Therefore, only one of the source threads that sent to the same destination thread will have its value received and its receive flag set to TRUE.

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 amortizes the cost of computing a path descriptor across multiple uses of the descriptor. A path is declared as follows:

```
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`. Note in the above example, `toB` was computed once but was used in three communications operations. Pointers to or arrays of paths may also be declared.

```
H.{ path(B) toB[4];  @toB[1].b1 = a1 ; }

H.{ path(B) *toB;  @(*toB).b1 = a1 ; }
```

Note, 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 description can not be assigned to a path. Any other path description can be (absolute, universal, or relative).

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 key word `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:

```
generic double sin (double x) { ... C code ... }
```

Generic routines can include inter-processor communication. These are **complex** generic routines. A complex generic routine can have a bundle as an argument. To pass a bundle into a generic routine other than the bundle of the context, one prefixes the argument with the key word **generic**.

```
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.

```
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**.

A generic or a "argument passed" bundle can have multiple dimensions. The dimensions' sizes may either be specified as in case 1 or unspecified as in case 2. Case 1 will only allow bundles with dimensions [2][500][30] to be passed as an argument, while case 2 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.

Case 1:

```
generic(other[2][500][30])
```

Case 2:

```
generic(other[ ][ ][ ])
```

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 **0** to **$$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 dimensions' sizes must be specified and have the same values. For case 3, 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. C is valid even though it has three children. The third child is simply ignored since f matches y and g matches z. D is invalid because k has only one dimension. E is invalid even though children of the right size exist they are not the first two children of E.

The context bundle of the function may also be passed in 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 4.

Case 4:

```
generic ( { y[2], z[4][6] } OTHER ) double Func (double x) { ... C code ... }
```

In Case 4 the context bundle in which 'Func' is called must be of the form of 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. But it would be for, say, a generic FFT routine. Any bundle that has been explicitly declared as a bundle argument or a generic bundle context may be used within the context of the function as any bundle specification can be used.

Realignment of Threads with Processors

If no realignment is specified for a bundle of threads, they will be distributed across the physical processors in a default manner specific to the given architecture. 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.

Actual threads are mapped across physical processors in an order dependent on the architecture. For example, an aCe program running on a PC cluster 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. The constant $$i is the logical thread identifier, and $$ii is the physical thread identifier. If the threads of a bundle are not realigned, then $$i will be equal to $$ii for all threads in the bundle.

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 dimension of the array. For example, if you have a two-dimensional array and you 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. (Refer to the preceding example of the bundle ‘X’.) The realignment keyword is ‘realign’. It is followed by the name of the bundle to be realigned. Next is 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 identifies 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 definitions 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 mapping maps the most significant logical dimension to the least significant physical dimension. This realignment effectively transposes the logical arrangement of threads relative to the physical arrangement. This realign statement is equivalent to:

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

The above realignment description refers to a physical array of threads which could be represented by P[9][5][6] and a logical array of thread represented by L[6][9][5]. The above 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 multi-dimensional array transposing relative to the default arrangement. The next form that will be described 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 above case, the logical array is define as a two-dimensional array, 100x100, and the physical array is a four-dimensional array of dimensions [10][25] [20][2]. The mapping (that follows the array declaration) says 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.

```
threads Y[100][100];
realign Y [100][100] --> [32][32] ; [1][0] --> [0][1] ;;
```

In the above example, the alignment statement is effectively equivalent to:

```
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 find out what the default arrangement of threads are for the specific physical architecture he wants to realign a bundle to.

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 totally virtual illusion 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'. 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. The routines developed under the native programming environment expect parameters and return values of specific types, either scalar or virtual, but not both. Therefore, when a native code routine is declared within aCe, the arguments and parameters must be explicitly designated as either scalar or virtual. Appendix A contains examples of the usage of the keywords `scalar` and `virtual`.

Appendix A: Current Standard aCe Library Calls

A.1 Assert *(see assert.aHr).*

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

A.2 ctype *(see 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 );
```

A.3 Error Number *(see errno.aHr).*

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

A.4 Locale Routines *(see locale.aHr).*

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

A.5 Math Routines *(see 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 *exp);
generic double       ldexp     (double x, int e);
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);
```

A.6 Scan Functions *(see 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 ) ;
```

A.7 Set Jump *(see setjmp.aHr).*

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

A.8 Signals *(see signal.aHr).*

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

A.9 Standard Arguments *(see stdarg.aHr).*

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

A.10 Standard I/O *(see sdtio.h).*

```
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 ... ) ;
```

A.11 Standard Library *(see 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 );
```

A.12 String Routines *(see 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 );
```

A.13 Time Routines *(see time.aHr).*

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

References

1. Dorband, J.E., "MPP Parallel Forth", Proceedings of the First Symposium on the Frontiers of Massively Parallel Scientific Computation, pg. 211-215, 1986.

2. 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, pp 625-628.

3. Iverson, K.E., *A Programming Language*, Wiley, New York, 1962.

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. Rose, J., Steele, G., "C*: An Extended C Language for Data Parallel Programming", Presented at the Second International Conference on Supercomputing, May 1987.

6. Steele, G., Wholey, S., "Connection Machine Lisp: A Dialect of Common Lisp for Data Parallel Programming," August, 1987.

7. *DAP-FORTRAN Language*, International Computers Ltd., TP 6918.

8. *MPL (MasPar Programming Language) Reference Manual*, MasPar Computer Corp., Pt No. 9300-9034-00 Rev A2.