ARCHITECTURE-ADAPTIVE COMPUTING ENVIRONMENT: A TOOL FOR TEACHING PARALLEL PROGRAMMING

John E. Dorband[1] and Maurice F. Aburdene[2]

Abstract

Recently, networked and cluster computation have become very popular. This paper is an introduction to a new C based parallel language for architecture-adaptive programming, aCe C. The primary purpose of aCe (Architecture-adaptive Computing Environment) is to encourage programmers to implement applications on parallel architectures by providing them the assurance that future architectures will be able to run their applications with a minimum of modification. A secondary purpose is to encourage computer architects to develop new types of architectures by providing an easily implemented software development environment and a library of test applications. This new language should be an ideal tool to teach parallel programming. In this paper, we will focus on some fundamental features of aCe C.

Index terms— Network Programming, Parallel Compiler, Parallel Programming Language

Introduction

Parallel and networked computer programming techniques have become popular and important computational tools taught in first year courses [1]. This paper provides an introduction to a new parallel programming language aCe C, a superset of ANS C [2]. We believe aCe C is ideally suited for teaching parallel programming once students have been taught C. In this paper, we assume that the reader is knowledgeable of ANS C. The concepts designed into aCe C have incorporated features found in parallel languages such APL[3], DAP Fortran [4], Parallel Pascal[5], Parallel Forth[6], C* [7], CM lisp [8], FGPC [9], and MPL[10]. aCe has been implemented on top of native C compilers such as gcc and Maspar MPL and on top of message passing libraries such as Cray SHMEM, PVM [11], and MPI [12].

In this paper, we will focus on particular features of aCe C (from now on will refer to it as aCe) with examples. For more details on the language we refer the readers to see the language reference manual [2].

aCe is based on the concept of structured parallel execution. First the programmer designs a virtual architecture that reflects the spatial organization of an algorithm. A virtual architecture may consist of groups or bundles of threads of execution. Code is written reflecting the temporal organization of the algorithm. The code defines what each thread performs, which together with the virtual architecture, defines the algorithm's execution.

aCe is both data-parallel and task-parallel: data parallel in that threads of a bundle execute the same code, and task parallel in that threads of different bundles execute different code.

aCe is architecture-adaptive, because different virtual architectures may be used for different physical architectures, to improve architecture dependent performance with minor changes to the code.

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. 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 [2] 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 easy expression of algorithms in an architecture independent manner.
- to facilitate the programmer's ability to port and implement algorithms on diverse computer architectures.
- to facilitate the programmer's ability to adapt and implement algorithms on diverse computing architectures.
- to facilitate the optimization of algorithms on diverse computing architectures.
- to facilitate development of applications on heterogeneous computing environments and programming environments for new computer architectures.

Basics

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

[1] John E. Dorband, NASA Goddard Space Flight Center ,Greenbelt, MD 20771, dorband@gsfc.nasa.gov

[2] Bucknell University, Electrical Engineering Department, Lewisburg, PA 17837, aburdene@bucknell.edu

```
/*
                Bundle of Hello Worlds
                     aCe program
*/
#include <stdio.aHr>

threads A[10];

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

This program will print "Hello aCe World" 10 times because the printf command is performed by each of the 10 threads of A.

The difference between aCe and C is that while C has only one thread of execution, aCe, may have many threads of execution. Each thread may be referenced by name and index. The 'Hello World' program's primary 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'. Note that 'main' (lower case) is the name of a function.

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 *bundle* 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 statement "threads A[10]" only declares the intent of the programmer to use 10 concurrent threads of executions named 'A' at some later point in the code.

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 of 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 lone 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 fread statement 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 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. In other parallel programming paradigms, these 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 statement

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

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

One may also declare bundles of bundles of threads. In the statement

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

'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 of 3,100 threads defined by the statement.

Since the definition of a bundle is recursive, a bundle can contain any number of sub-bundles. Note that 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 statement

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

demonstrates the recursive nature of a bundle declaration.

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 will by default 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 code. The compound statement labeled with the bundle name B,

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

copies the value at location 'b' to the location 'a'. However, if a conditional statement is executed from within the labeled compound statement,

```
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 '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. For example, in the statement

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

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':

This would be equivalent to the statement

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

Then why would it be necessary for contexts to be nested? It is only necessary for contexts to be nested 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 statement

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

In this example, if 'a' is true for any thread of B, then 'y' is copied to 'x' for all threads of A. Otherwise, if 'a' is

false for all threads of B, then 'y' will 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, '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. Putting it another way, if any thread of B is active within the context of A, then all threads of A will be active.

Functions 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 function, aroutine, may be called from 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 statement

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

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

B.{ if (a) { A[idx].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 or have their storage modified. For example, in the statement

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

all but one thread of A are made inactive; yet, any active thread of B may store (or put) values into any thread of A whether the thread of A is active or not.

Element 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 definitions of the system constants are:

$$Name - the name of the bundle of threads (char*).
$$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.
$$Lx[d] - the log base 2 of the number of threads of d-th dimension of the subbundle (equals -1 if $$Nx[d] is not a power of 2).
$$ix[d] - the thread identifier of the thread in d-th dimension of the bundle/subbundle.

The following example uses the bundle description:

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

```
        A                 A
        0 0               1 1
  +-----------------+-----------------+
  |  B      B       |  B      B       |
  | 0 0 0  1 0 1    | 4 0 0  5 0 1    |
  |  B      B       |  B      B       |
  | 2 1 0  3 1 1    | 6 1 0  7 1 1    |
  +-----------------+-----------------+
```

Figure 1: Bundle of Threads

A has 2 threads. In Figure 1, 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 values of the other system constants of A are:

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

The values of the other system constants of B are:

4

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

Input & 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 ordered sequentially. I/O in aCe is in order of thread identifier. For example,

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

The thousand values of 'buff' 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 'buff', corresponding to the active threads, will be written to the file. And in,

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

The code segment will write eight values into the file, four from the first four threads of C and four from the last four threads of C 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 be read by a bundle with exactly the same geometry as the bundle that wrote. 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 or get, the *evaluated numeric expression* must evaluate to a value, while in the case of a send or put 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

threads A[1]; threads B[1];

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

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. 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 which can be fetched from for the expression to be evaluated. Though a thread may have many 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 gather 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. For more details on path descriptions see[2].

Scatter and Gather

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:

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

Reversing the sides of assignment makes it a scatter 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 variable 'b' of thread B. If a scatter 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 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.

```
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 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.

Pre-computed Paths

A fair amount of time 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 ; }
```

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.

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 also include inter-processor communication. These are complex generic routines. A complex generic routine also can have a bundle as an argument.

Summary

We have introduced a new computer language, aCe C, that is ideally suited to parallel, networked, and cluster computing. We have presented the basics of the language, the concept of threads and bundles of threads, execution of programs, communication, element identification, input and output (I/O), communications path description, scatter and gather, pre-computed paths, and generic routines. These tools and the aCe C language greatly enhance the teaching of parallel programming.

References

[1] Wilkinson, B. and Allen, M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, 1999.

[2] Dorband, J.E., "aCe C Reference", NASA Goddard Space Flight Center, Greenbelt, MD.

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

[4] *DAP-FORTRAN Language*, International Computers Ltd., TP 6918.

[5] 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.

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

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

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

[9] 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.

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

[11] Gropp, W. Lusk, E. and Skjellum, A. *Using MPI*, 2nd Edition, MIT Press, 1999.

[12] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V.S., *PVM - A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Boston, 1994.