AC for the T3D

William W. Carlson and *Jesse M. Draper,* IDA Supercomputing Research Center, Bowie, Maryland, USA

1 Introduction

We have modified the C language slightly to enable programmers to read and write remote memory efficiently with simple assignment statements. To create a programming model in which local data and remote data are differentiated solely by the ways in which they are declared, we have focused on pointers and arrays, the two C constructs which are most closely tied to addresses. The addition of a single keyword gives the programmer the ability to declare a pointer that can point to any memory location in the program's virtual address space or to an array that is distributed among all the local memories of the program. By keeping software constructs to a minimum and enabling programmers to use underlying hardware efficiently, AC is solidly in the C tradition. Using the Free Software Foundation's GNU C Compiler [Stal 94], we have implemented AC on the Cray T3D and have achieved excellent performance results compared to the Cray Standard C compiler.

2 Modifications to C

Adding distributed objects to ANSI C required one syntactic change: the addition of a keyword "dist" as a *type-qualifier* [ANSI 89] in data declarations to indicate that the declared objects are distributed across all the processing elements in the system. Distributed scalars result in a single object on a single PE. In all of the example code throughout the paper, `PROCS` is a special `const int` variable which is initialized to the number of processors in a system segment, and `MYPROC` is a special `const int` variable which, for each processor, is initialized to the index of that processor, between `0` and `PROCS-1`. Although it is not necessary that distributed arrays have a final dimension which is either the symbol PROCS or an integer multiple of that symbol, such declarations make explicit the distribution across processors.

```
dist int y[PROCS];/*one y per PE*/
dist int a[100][PROCS];
     /*one a[100] per PE*/
dist int b[100][12*PROCS];
     /*one b[100][12] per PE*/
dist int odd[PROCS][17];
    /* 17 integers per PE, BUT
       they are SKEWED in their
       mapping by 17/PROCS */
dist int x;   /* one x on entire
                 system */
```

2.1 Distribution of Arrays

As the syntax indicates, the addresses a[100][5] and a[100][6] are on successive processors, with "successive" meaning the next higher numbered processor, with the exception of the last processor, whose successor is processor 0. Note that b[87][5] and b[87][6] are also on successive processors. The local element of b that is one above b[42][5] is b[42][5+PROCS]. In effect, the layout of b is the same as if it had been declared b[100][12][PROCS]. The difference between the two is that references to the latter require three subscripts rather than two. A statement of the form

```
i = a[42][i];
```

will result in the computation of an address on processor `i` and a remote fetch from that address. Similarly, a statement of the form

```
a[i][j] = i;
```

will result in the computation of an address on processor `j` and a remote store to that address.

2.2 Pointers

In order to achieve consistency between arrays and pointers, operations on pointers to distributed objects have several characteristics that differ from those of conventional C pointers. Internally, pointers to distributed objects have two separate components, a processor number and a local address. The processor number is used to determine where the remote reference is to be done, and the local address is used on that processor as if it were in that processor's "local" view. On the T3D, the top 16 bits of a 64-bit value are used to hold the processor number, and the bottom 48 hold the local address. This section discusses several important semantic characteristics of distributed pointers: how they are affected by pointer addition and subtraction; how they are affected by casting to local pointers; and how local pointers are cast to distributed pointers.

It is important to understand a distinction in the ANSI standard term *type qualifier* [ANSI 89] and, specifically, the action of type qualifiers on pointers. The major question is "What is distributed, the pointer itself, or what it points at?" In general, the way to read these declarations is to start with the object

being declared and work out, so the simple declarations are described as follows:

```
int *p;  /* p points at an int */
dist int *pd;  /* pd points at an
                  int in "dist space" */
```

Both of these declarations declare a single value (the pointer) which is allocated on each processor (i.e., local) and is uninitialized. If both pointers have been set to point to reasonable things, dereferencing `p` (i.e., `*p`) results in an access to local memory, whereas dereferencing `pd` results in a remote memory access to "anywhere" in the machine. Continuing to slightly more complex references:

```
int *dist dp;
    /* dp is a pointer which is
       distributed; it points at
       an int which is local */
dist int *dist dpd;
     /* dpd is a pointer which is
        distributed; it points at
        an int which is in dist
        space */
```

In this case each declaration declares a pointer object which is "`dist`" (i.e., a single pointer shared by all processors). That is the meaning of `*dist` in the declarations. Dereferencing `dp` results in a remote memory access to get the value of `dp` (from a single PE), followed by a local memory access using that value. Dereferencing `dpd` results in a remote access to get the value of `dpd`, followed by a second remote access to get the value at that address. Note, then, that the number of `dist` keywords in the declaration indicates the number of remote accesses performed when the object is (fully) dereferenced, and that the order (from "inside-out") indicates which are local and which are remote.

2.2.1 Arrays of Pointers and Pointers to Arrays

Another level of declaration complexity can be obtained by using both pointers and arrays in the same declaration. The important rule in understanding these declarations is that, in the default case, array notation (`[]`) has higher precedence than pointer notation (`*`). Consider the following declarations:

```
int *ap[10]; /* an array of 10
                pointers to int */
dist int *apd[10];
     /* an array of 10 pointers,
        each pointing at an int
        which is in dist space */
int *dist adp[10];
     /* a distributed array of 10
        pointers, each pointing at
        a local int */
dist int *dist adpd[10];
     /* a distributed array of 10
        pointers, each pointing at
        an int in dist space */
```

Each declares an array of 10 pointers. The first two (`ap`, `apd`) of these arrays are "local"; the second two (`adp`, `adpd`) are distributed as described in Section 2.1. A "pointer to a distributed array" requires a declaration using parentheses to break the common precedence rules:

```
dist int (*p)[10];
     /* a single (local) pointer
        to a distributed array of
        10 integers */
```

Such a declaration is quite useful for "striding" through a complex data structure. If the dimension is PROCS, this is a particularly useful context, because increments to p would remain on the same processor. One of AC's advantages is that a programmer with a proper understanding of declarators (as describe above) can create virtually any conceivable data structure. To a large extent this flexibility is possible because the implementation of distributed objects is "faithful" to C.

2.2.2 Pointer Arithmetic

Pointer arithmetic is defined modulo PROCS. For example, successive increments of a distributed pointer increment the processor number until it reaches PROCS and only then affect the offset. Although distributed pointers are not structures, structure notation is useful for defining what happens in pointer arithmetic for two distributed pointers dp and dp1:

```
dist int *dp, *dp1;
dp1 = dp + i;
```

effectively generates the following code

```
dp1.proc = (dp.proc+i)%PROCS;
dp1.loc = dp.loc +
          ((dp.proc+i)/ PROCS)
            * sizeof(*dp);
```

where `proc` and `loc` refer to the components of this pseudo-structure and `sizeof(*dp)` represents the size of what `dp` and `dp1` point at (int in this case). Thus `dp++` points to the same offset on the next processor unless the processor number before the increment is PROCS-1, in which case `dp++` points to the next local address (+= 8 bytes for int's on the T3D) on processor 0. This convention allows a user to step through an entire distributed array one element at a time.

2.2.3 Casting of "Local" Pointers to Distributed Pointers

Whenever a local pointer (one without the `dist` attribute) is cast into a distributed pointer, the processor number portion of the distributed pointer is set equal to the current processor. Therefore, dereferencing the distributed pointer will result in the same value as dereferencing the local pointer.

```
dist int *dp;
int *p;
dp = (dist int *) p;
if (*dp == *p)
    printf("always true!");
```

Because pointers to distributed objects include a processor number, they cannot be statically initialized.

2.2.4 Casting of Distributed Pointers to "Local" Pointers

Whenever a distributed pointer is cast to a local pointer, the processor number of the distributed pointer is lost. Therefore, this operation is dangerous, because the resulting local pointer may point to a different object than the distributed pointer. However, it is useful to get a local pointer for efficient access to the local elements of a distributed array.

```
dist int x[PROCS];
int *p;

p = &x[MYPROC];
     /* p points to x[MYPROC], but
        is more efficient for
        pointer ops */
```

On the T3D all accesses via dist pointers are uncached (they use the prefetch mechanism), and all accesses via local pointers are eligible for caching. In general, this works very well, but cache consistency problems can arise when both a distributed pointer and a local pointer are used to refer to the same object. If a program reads a local pointer and does a store to the dist pointer (on any processor), there may be an inconsistent value in the dereference of the local pointer. For this case, users must use some invalidation mechanism (such as cache_invalid_on() or line_invalid_on(x)).

2.3 Node-level GCC Extensions

Because AC is based on GCC, the GNU C Compiler, AC provides the same extensions to ANSI C that GCC does [Stal94]. While some users shy away from these extensions, several are quite useful in developing high performance programs for the T3D. The first is the use of variably dimensioned automatic arrays and function arguments, much as in Fortran 77. Syntactically, the dimension of an array can be any expression. In the following function:

```
foo (a,n)
  int n;
  double a[n][n];
{
  int i;
  for (i=0;i<n;i++) a[i][i] = 1.0;
```

the array `a` is dimensioned at runtime (and sets the diagonal elements to 1.0). This syntax will work with distributed arrays as well as normal ones.

The other major GCC enhancement of use to (and used by) AC programmers is the GNU "asm()" mechanism, which allows any instruction to be included in user source C code with C expression operands. This extension is obviously machine dependent and requires knowledge of the T3D architecture, specifically the instruction set architecture of the DEC ALPHA chip. The basic format of the instruction is:

```
asm(assembler_string :
output_operands : input_operands);
```

where assembler_string is the actual text to be written to the compiler output; output_operands describes where results of the instruction are written; and input_operands describes the sources of instruction inputs. More specifically, assembler_string is an ascii string with printf-style % arguments. Each % is followed by a number, which refers to the output and input operands, in the order they appear. The output_operand and input_operand segments of the asm specification are comma separated lists of operands, each of which has a "constraint_string" and a C expression value. In general, the "constraint_string" is a collection of potential register or memory classes which are used internally to GCC for the ALPHA. The letter "r" refers to an integer register; the letter "f", to a floating point register; and a number, to a duplicate argument. Thus the asm() statement for a store quad conditional instruction (needed to access the DTB annex on the T3D) would be:

```
int val; int *addr;
asm volatile ("stq_c %0,0(%2)":
 "=r" (val):"0" (val),"r" (addr));
```

In this case, the instruction has an output register which is the same as the input register (the instruction both reads and writes the value) and an input register containing the address. If `val` were allocated by the compiler to register r12 and `addr` to register r7, the compiler would output the instruction:

```
stq_c r12,0(r7)
```

to the assembler output file, which would then be assembled as appropriate. The keyword volatile tells the compiler not to rearrange or eliminate the instruction if `val` is unused after this point in the program. Note that it is crucial to get the input and output arguments of these instructions correct, because the compiler will attempt to optimize and rearrange the user's code as much as possible, and if the constraints are not correct, bad code will result. For example, if the argument `val` was not an output argument, the compiler might attempt to reuse the value `val`, which would be wrong because the `stq_c` instruction changes that register.

3 Performance Analysis

To analyze the performance of our distributed access prototype, we conducted two classes of experiments: a comparison of the low-level transfer performance of AC to highly optimized libraries on the T3D, and a comparison of the implementation of a distributed algorithm under both the distributed model and under a highly efficient message-passing model. These experiments show that our goals are being achieved: the low-level performance is superior to the optimized libraries, achieving very near hardware-limited performance, and the benchmark algorithm shows that one can achieve this performance level

while having a much cleaner representation than is possible with the message-passing model.

3.1 Scalar Performance

AC can produce extremely efficient code for the T3D system in most cases. However, some input programs will produce quite bad results. For maximal efficiency, two simple guidelines should be followed when preparing programs. Fortunately, these guidelines often result in "good" programs whose functionality is clear and therefore easy to understand, maintain, and modify.

1. Each subroutine/function should be of "reasonable" size.
2. Memory usage should encourage reuse.

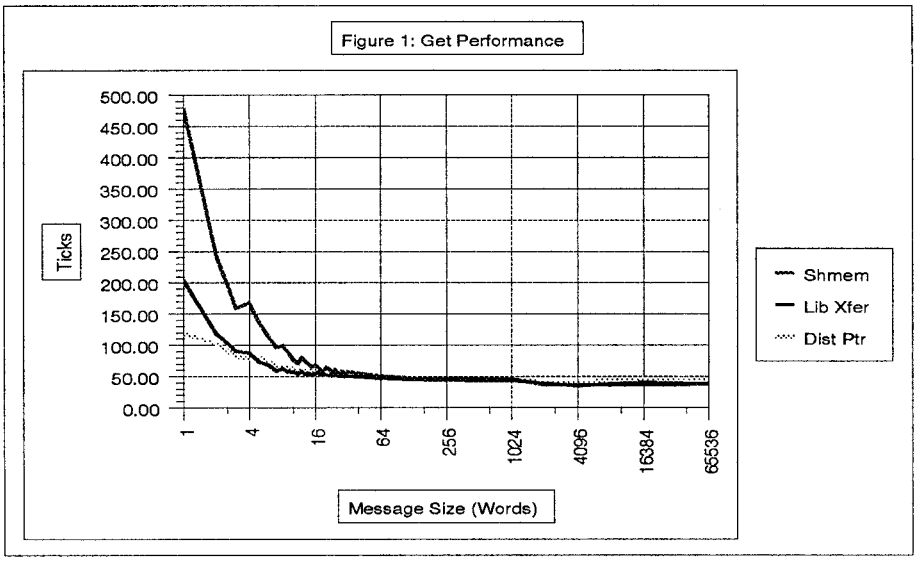
The first rule prevents two big performance problems: too much function call overhead and too much "local" variable state. The first problem occurs when a function has too little work. This can be cured by using the "inline" attribute in GCC, or by simply making each call do a larger amount of work. In general, a function call/return requires about 100 cycles and 10 memory accesses. This means that virtually any function containing a "loop" is big enough, but short (1-10 statement) sequential functions will incur a large overhead. On the other end of the spectrum are gigantic functions. Not only are these difficult to maintain, but they also, in general, contain too much state for the 32 registers in the DEC ALPHA chip. Probably anything with more than 20 local variables or longer than 200 lines should be broken up. Note that both tiny and huge subroutines will compile and run correctly; the overall program will just run much faster if subroutines are "reasonably" sized.

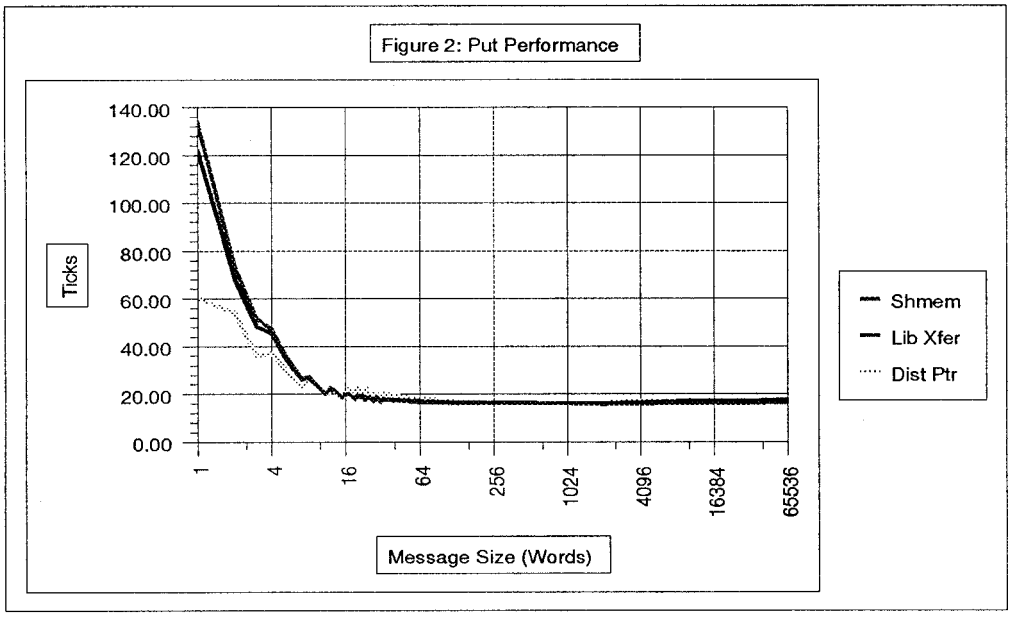
The second rule recognizes the presence of cache memory on the DEC ALPHA chip. Because cache memory is between 8 and 30 times faster than main memory, it is important to use it effectively for high performance. This is mainly achieved by structuring algorithms to allow reuse of data in units of the cache size (8Kbytes for the T3D) or, if that is not possible, using stride-one access to larger data structures. While this paper is not the appropriate place to describe all such algorithms in depth, there is a large body of research [DDSv91] in so-called "blocking" algorithms which may be helpful to users of the T3D.

Our experience shows that when users follow these guidelines, performance achieved by AC and the T3D can be very high. Typical experience shows a 50% to 300% increase in the performance of programs so structured and compiled by AC, when compared to the Cray Standard C compiler, version 4.0.3. When such performance is not seen, it is almost always due to function size problems or a memory-bound (i.e., noncaching) program.

3.2 Get/Put Performance

Figures 1 and 2 show the "get" and "put" performance of three systems: distributed declarations, the Cray provided "shmem" library, and a highly optimized library written locally. In each case, a block of N words was transferred from an originating processor to a remote processor repetitively until the performance measures stabilized. All values are given in "ticks per word", which refers to the transfer rate for a machine with 150MHz clock cycles. For small values of N our mechanism significantly outperforms both of the library versions, and for large values of N the three approaches are generally of equal performance. Therefore, we would expect to see performance advantages using this model when transfer granularity is small and equivalent performance when it is large. Note that these transfers were all dependent on each other, so no compiler optimization was possible. While the put rate is at the hardware limit of the system, the get rate for single word transfers can be reduced to approximately 70 ticks per word if concurrency is visible to the compiler. This, again, is quite close to the hardware limit of the system.





3.3 Benchmark Algorithm Comparison

One AC user has implemented the "blocks-in-a-box" puzzle as described in [BCG 82]. This is a fairly complex benchmark which maintains both a distributed stack of partial solutions to the puzzle on each processor, and a tree structure among the processors for load balancing. There are two implementations: one which uses a very efficient message-passing model, and another which uses the distributed declarations described in this paper. The code for the distributed case is not only considerably cleaner than the message-passing code, but also about 10% faster on average. The latter point is consistent with the data reported above because the "state" passed between processors is 16 words in length or smaller. Therefore, for at least this case, our new model has achieved its goals of equaling or surpassing the hardware limited library performance while providing a convenient and easy to understand programming model for users. In addition, because of the improved optimization, instruction scheduling, and register allocation of the AC compiler, the node-level section of this program runs 5 times faster than when compiled with version 4.0.3 of the Cray Standard C compiler.

4 Using AC

Instructions for acquiring and installing AC can be obtained from the authors. A technical report is available [CaDr 95], and email can be sent to wwc@super.org or jdraper@super.org. Bug reports can also be sent to the authors at the same addresses.

4.1 Command Line

To compile an AC program for the T3D, issue the following command:

```
% ac -O2 -fprocs-N file.c
```

The `-fprocs-N` flag tells the compiler and linker to produce code for N processors. Currently users must compile for a specified number of processors; that requirement may vanish in the future, but specifying the number of PE's will continue to produce better code since the compiler can use constants in performing address calculation and pointer arithmetic. It is possible to use other optimization levels, but level 2 is enough to gain the distinct performance advantages offered by GCC's instruction scheduler.

4.2 Debugging Support

When invoked with the -g option, AC generates code that is compatible with CRI's TotalView debugger. Users can set breakpoints in source code, single-step source code, and examine both local and global variables by name. It is also possible to examine assembly code and registers.

5 Conclusions

We have designed and implemented extensions to the C programming language that enable programmers to read and write remote memory efficiently with familiar C syntax. Our performance results on both node-level programs and remote stores and fetches compare quite favorably with those of the Cray Standard C compiler and libraries on the Cray T3D.

6 Acknowledgments

The Split-C work at UC Berkeley [CDG 93] inspired us to add similar features to AC, and David Culler and several of his colleagues were generous with their time in discussing approaches to the problem of distributed objects. We would like to thank a number of users who have bravely tested AC by writing actual applications rather than just test programs. We also appreciate the cooperation of Cray Research, Inc., which

has provided essential documentation and worked with us to improve the compatibility of AC programs with Cray tools.

7 References

[ANSI 89]American National Standards Institute, *American National Standard for Information Systems--Programming Language--C*, 1989, sec. 3.5.4.1.

[BCG 82] Berlekamp, Elwyn R., John H. Conway, and Richard K. Guy, *Winning Ways for Your Mathematical Plays, Volume 2: Games in Particular*, Academic Press, New York, 1982, p. 736.

[CaDr 95] Carlson, William W., and Jesse M. Draper, *AC for the T3D*, Technical Report SRC-TR-95-141, Supercomputing Research Center, February 23, 1995.

[CDG 93]Culler, David E., Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick, "Parallel Programming in Split-C," in *Proceedings of Supercomputing '93*, Portland, OR, November 15-19, 1993, pp. 262-273.

[DDSv91]Dongarra, Jack J., Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Press, Philadelphia, 1991.

[Stal 94]Richard M. Stallman, *Using and Porting GNU CC*.