UNH C* User Manual

May 24, 1994

This is a user document for UNH C*. Unfortunately, this document is currently highly incomplete.

Contents

1 C* Language Overview

The following is intended to provide an extremely brief overview of the features of the C* language. Hopefully, this will allow a knowledgeable C programmer to quickly acquire enough knowledge of C* to intelligently read this document. Detailed information is best obtained from [4, 2].

1.1 Introduction

As the name implies, C* is a derivative of the ANSI standard C language [1]. It was created by Thinking Machines Corporation for use with the Connection Machine, a single-instruction multiple-data model (SIMD) parallel machine according to Flynn's taxonomy [3]. C* version 6.0 [4] is a data-parallel language and may fairly be considered the second such language, created by Thinking Machines, bearing the name C*. The earlier, now obsolete, language [5] was also a data-parallel enhancement of C, but the semantics of parallel context were fundamentally different.

Broadly speaking, parallel languages may express parallel operation in two ways: control parallelism (multiple instruction, multiple data: "MIMD" model) and data parallelism ("SIMD" model). Languages which implement control parallelism provide a means for several discrete program segments to execute simultaneously. Data parallel languages allow only a single instruction stream to be active, but provide some means for that stream to operate on more than one piece of data concurrently. Languages have been created which implement either or both models in a hybrid model.

Control parallelism has never been a serious alternative however, since it has several major problems. Parallelism derived from multiple discrete program segments can only scale to as many processors as there are concurrently running segments. At best, this represents a constant time speed up over a conventional implementation. Further, multiple independent thread control requires careful attention to interprocess communication and synchronization. These issues have always been difficult in the realm of operating systems.

Data parallelism obviates the difficulty with synchronization, since the behavior of a single instruction stream is deterministic. Communications between concurrently active processing elements is accomplished implicitly through simple data access. Potentially, a data parallel program can be scaled to use as many processors as there are data elements in the computation.

Figure 1 will serve as a sample C* fragment throughout the balance of this section. The program implements the Jacobi algorithm to find the steady-state temperature distribution on an insulated two-dimensional plate, given constant boundary conditions.

1.2 Parallel Data

The logical form of all parallel data in C* is a mesh with an arbitrary number of ranks. Proximity in the logical mesh implies proximity as data is mapped to physical processors, though no particular mapping strategy is guaranteed.

1.2.1 Declarations

The characteristics of a mesh of parallel data are defined using the type specifier `shape` (Figure 1, line 5). While a shape does not declare actual parallel data, it is not strictly like a C `struct` either. A shape may be passed as a parameter and used for some limited operations. The most important aspects of a shape are that it provides a means of tagging some data as being parallel (Figure 1 lines 6-9) and of subsequently specifying parallel operations on that data (line 13).

It is also possible to declare shapes with specific dimensions left undefined "`[]`" or with no dimensions defined at all. Such undefined characteristics may be defined at run time using intrinsic functions. In this manner, a program may dynamically adapt to an input size.

```
#include <math.h>
#define SIZE    128       /* Resolution of grid */
#define TEMP    50.0      /* Arbitrary cut-off  */
#define EPSILON 0.1

shape [SIZE][SIZE]cell;
bool:cell  active;       /* 0 if cell boundary; 1 otherwise */
float:cell change;       /* Change in temperature */
float:cell new;          /* Newly calculated temperature */
float:cell old;          /* Previous temperature */

main () {
  float maxerr;          /* Largest change in temp. over grid */
  int   cool_cells;      /* Number of cells with temp. < TEMP */
  with (cell) {          /* Initialize grid */
     where ((!pcoord(0)) ||  (!pcoord(1)) || (pcoord(1) == (SIZE-1))) {
        active = 0;
        old = new = 0.0;
     } else where (pcoord(0) == (SIZE-1)) {
        active = 0;
        old = new = 100.0;
     } else {
        active = 1;
        new = 50.0;
     }
     do {               /* Compute steady-state temperatures */
        where (active) {
           old = new;
           new = ([.-1][.]old + [.+1][.]old +
                  [.][.+1]old + [.][.-1]old) / 4.0;
           change = fabs(old-new);
           maxerr = >?= change;
        }
     } while (maxerr > EPSILON);
     cool_cells = (+= (new < TEMP));
  }
  printf ("There are %d cells cooler than %5.2f degrees\n",
     cool_cells, TEMP);
}
```

Figure 1: Example C* program.

1.2.2 Scalar Accesses

Scalar accesses[1] of parallel data appear very much like conventional C array accesses, except that the subscript expression prefaces the name of the vector. Such accesses may be used to initialize parallel data or extract results in scalar context.

Interestingly, scalar accesses of parallel data are not restricted to scalar context. If a parallel data element is accessed from parallel context using strictly scalar data, then only one reference can possibly be implied.

1.2.3 Parallel Accesses

Logically enough, parallel data accesses may only appear in parallel context. Such accesses are normally of the data immediately activated by the shape of the parallel context. The most common forms of parallel data accesses are local (Figure 1 line 16,18-19,21-22,26-30) and remote (Figure 1 line 27-28). The remote accesses are distinguished by the presence of a *left-index* expression, appearing as subscripts to the left of parallel vector names.

The most common type of local access is simply to name the parallel variable and, by virtue of the context, all active copies of that variable are implied to take part in the operation.

Most remote parallel accesses refer to data which is located relative to a currently active piece of data. When the "." symbol appears in a parallel address calculation, the index of each active local data element is implied.

An additional special type of parallel access is accomplished through the `pcoord()` pseudo-function. Using this function, and the index of a rank in the currently active shape, the index of each currently active shape element (the value of ".") may be obtained (lines 14 and 17).

1.3 Spreading and Consolidating Data

1.3.1 Broadcasts

A broadcast occurs whenever an assignment contains a scalar right-hand side and a parallel left-hand side (Figure 1 line 16,18-19,21-22,33). While the scalar right-hand side will normally be simple global or scalar data, an active parallel data item may be broadcast by referencing it with a scalar left-index.

1.3.2 Reductions

Consolidations of parallel data to a single scalar value are known generically as reductions. C* provides control over the method of the consolidation by overloading the C assignment operators, such as "`+=`" and "`*=`" for a global sum and global product respectively. To be recognized as a reduction, the operand to the right must be a parallel expression. Unlike the normal C versions, C* reductions may appear in both binary and unary form. In binary form they perform the specified data reduction followed by an assignment to the left-hand side. In unary form, they perform an in-place reduction (Figure 1 line 33).

C* also includes special maximum and minimum value reduction operators, "`<?=`" and "`>?=`" (Figure 1 line 30). Syntactically, these operators behave exactly like the other reductions.

[1] C* has unfortunately created subtle variations of the meaning of the term "scalar." In C*:

- Scalar Data - unlike the conventional C meaning, this implies all conventional sequential data, including structures and vectors.
- Scalar Context - sequential operating context. Essentially, this is sequential C.
- Scalar Access - discrete access of a single piece of parallel data, either from scalar or parallel context.

1.4 Control Flow Model

1.4.1 Entering Parallel Context

The control flow model in C* is entirely sequential. At no time in a C* program is more than one thread of control active. Parallel context provides a means of specifying when operations are to occur simultaneously for all elements of a given shape.

Entry to parallel context is accomplished by use of the `with` statement. The statement accepts a shape name as a parameter, and initiates parallel operation for the data associated with that shape. The context of the `with` statement lasts for either a single statement or a "{}" enclosed block.

In our sample, we activate the parallel variables `active, change, new,` and `old` by referring to shape `cell` in a `with` statement (line 13).

1.4.2 Restricting a Computation

It is possible to restrict the items of a shape for which a computation is carried out. This is done using the `where` statement (lines 14, 17 and 25). The `where` statement is only valid within a parallel context. A `where` statement must be composed of at least one piece of active parallel data (otherwise, it would be constant for all values). Data elements which satisfy the test found in the `where` statement are allowed to continue computation within the statement. Precisely those elements not satisfying the `where` condition may be separately activated using the `else` form (lines 17 and 20).

The effect of the `where` terminates with the end of the `where` statement.

References

[1] American National Standards Institute, New York, NY. *Programming Language—C, ANSI X3.159-1989*, 1989.

[2] Thinking Machines Corporation. *C* Programming Guide Version 6.0.* Thinking Machines Corporation, Cambridge, MA, 1990.

[3] M. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[4] J. Frankel. C* language reference manual. Thinking Machines Corporation, 1991.

[5] J. Rose and G. Steele Jr. C*: An extended C language for data parallel programming. Technical Report PL 87–5, Thinking Machines Corporation, Cambridge, MA, 1987.