This paper appeared in: Communications of the ACM, 35(6):21–24, June 1992

A Critique of the Programming Language C*

Walter F. Tichy
Michael Philippsen

University of Karlsruhe
School of Informatics
D-7500 Karlsruhe, F.R.G.

tichy@ira.uka.de
philippsen@ira.uka.de

Phil Hatcher

University of New Hampshire
Department of Computer Science
Durham, NH 03824, U.S.A.

pjh@cs.unh.edu

C* is a data parallel programming language originally developed for the Connection Machine. Efforts are now underway to standardize a revised version of C* [6]. We think that standardization of C* is premature at this time, since the language contains a number of unproven constructs and obvious flaws. We are concerned that standardization of a parallel language now might force its programming model upon future generations of programmers, *even though we already know it is deficient*. The purpose of this note is to make the relevant issues accessible to a wider audience and to make specific recommendations for improving C*.

C* is an extension of ANSI standard C and intended as an "efficient, fairly low-level systems programming language[6]" for parallel computers with distributed memory. Parallelism is expressed directly in the data parallel paradigm. In this paradigm, "parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control[2]". Data parallelism is a synchronous paradigm and therefore well suited to SIMD machines. It has also been implemented successfully on a MIMD machine[5].

As an extension of C, C* inherits most of the drawbacks of its ancestor, but we are not concerned about those here. Neither are we concerned with limiting C* to a synchronous paradigm, even though an asynchronous one would be more general. We are concerned, however, with the principles of programming language design, the programming model underlying C*, and the efficient implementability of C* on both SIMD and MIMD machines. The problems we identified in C* in these areas are discussed below.

1 Parallel Data Types

C* introduces the parallel variable as a new data type. The parallel variable is an array of one or more "parallel" dimensions. All elements of a parallel dimension may be processed simultaneously, while the traditional, "serial" dimensions can only be processed serially. For example, if one wishes to define a two-dimensional array A whose rows can be processed in parallel, but whose columns are processed serially, then one declares the following:

```
shape [N]rowdim;
float:rowdim A[N];
```

The *shape* declaration given here introduces *rowdim* as the name for a storage structure that can hold variables with a parallel dimension of N elements; note the index range declared on the left of the identifier. The second declaration then allocates a variable A with the shape given by *rowdim*. The elements of this variable are again vectors, but with a serial dimension. This time, the index range is on the right. The left and right indexing carries over into accessing arrays: The expression $[i]A[j]$ would select the element in the i-th row and j-th column of A. This notation is unusual, but is intended to provide syntactic clues to the programmer about which dimensions can be processed serially and which in parallel. A minor annoyance is that if the programmer should decide to change a dimension from serial to parallel or vice versa, all index expressions in the program involving the changed type must be switched around accordingly.

While syntax is a matter of taste, non-orthogonality of the new type constructor for parallel variables is a more serious problem. Parallel variables cannot be combined freely with other types. For instance, it is not possible to create parallel variables with records as elements that in turn have parallel variables as components. Perhaps one could argue that this particular restriction is minor for the intended application area of C*. A more serious non-orthogonality concerns pointers. Pointers may not be stored in parallel variables (or in records or other data structures stored in parallel variables). This restriction is unfortunate, since there are many non-numeric applications that could use parallel pointers[1]. Also, omitting them is inconsistent with the spirit of C, where pointers are used frequently. The prohibition against parallel arrays of pointers seems to be motivated partly by the addressing properties of the Connection Machine, and partly by the type structure of C* itself. The earlier version of C* has about ten different variants for each pointer type. The variants reflect whether the pointer itself is stored in a singular or a parallel variable and whether it actually points to a singular or parallel variable, plus some additional variants. The result is that parallel pointers in old C* are exceedingly complicated to program. It appears that the same complexities would arise in new C*, and were omitted for this reason.

The source of these difficulties is quite simple: The orthogonal notions of data type and data layout have been intermixed in C*. A data type determines which operations can be applied to a datum; the layout determines whether the operations can be applied in parallel or serially. These are two separate, independent notions. The set of *what* operations can be applied to a datum should be independent of *how* (in parallel or serially) they can be applied. A better approach appears to be to rearrange a variable of a given type implicitly and automatically to fit a required layout (perhaps with a performance warning from the compiler).

2 No Nested Parallelism

Nested parallelism occurs when a parallel program calls a procedure or another statement which spawns additional parallelism. This feature has been found to be necessary for writing high-level parallel programs[1]. Nevertheless, nested parallelism is not possible in C*. Instead, the nested parallelism must be pushed up to the top level of the program. This property forces programmers to distort otherwise clear programs, prevents the top-down structuring of parallel programs with subprograms, and hinders reusability. We will illustrate these problems with a somewhat longer example.

Suppose we wish to write a program for searching large game trees, such as in chess or checkers. The nodes in the game tree are board positions, where an edge connects two nodes if a single, legal move leads from one node's position to the other's. The task is to write a program for expanding the game tree from a given position down to a certain level. This situation is typical for many search algorithms where the search space is irregular and cannot be given a priori. A clear, recursive outline for building and searching the tree is as follows.

```
void SearchTree(position p) {
    if (p->depth < maxdepth) {
       successors=GenerateMoves(p)
       forall i in length(successors) do
           SearchTree(successors[i])
}}
```

The function *SearchTree* obtains a single board position as parameter and tests whether the maximum search depth has been reached. If not, it calls function *GenerateMoves*, which generates the list of legal successor positions and returns it. *SearchTree* then spawns as many additional invocations of itself as there are successor positions. Note that there may be many simultaneous invocations of *SearchTree* operating simultaneously, but under a synchronous paradigm, they will all perform their various actions, such as calling subprograms, in perfect synchrony. *SearchTree* would also run unchanged under an asynchronous paradigm.

While this program can be transformed to fit C*, the result is not nearly as clear and concise. First, *SearchTree* and *GenerateMoves* must be changed to accept a vector of positions as parameters. Second, *GenerateMoves* must be split into two parts. The first part estimates the number of successors for each position in the vector. These numbers are added and the result is used to allocate a new position vector long enough for storing all successors of the input vector at once. The second part of *GenerateMoves* then fills in the successor positions. Finally, the filled vector is passed to another invocation of *SearchTree*. This transformation illustrates how the parallelism inside *GenerateMoves* and *SearchTree* has to be marshalled and pushed up to the calling procedure. Although this transformation is not particularly hard to program and becomes easier with practice, we believe it would be better performed by the compiler and runtime system. Note also that after the transformation, the idea of multiple threads operating simultaneously has been lost, and a compilation of the transformed program for an MIMD machine might be inefficient due to the forced synchroneity.

The non-nested parallelism of C* is inappropriate for writing clear, maintainable, and portable parallel programs. One might argue that it is still unclear which forms of nested parallelism are appropriate and how to implement them, but that is precisely our point: It is too early to standardize nested parallelism out of existence with C*.

3 Multiple Copies of Each Function

For scalar functions, the programmer must write two versions: one for parallel, the other for sequential contexts. For instance, suppose that we have written a scalar function *abs(x)* that returns the absolute value of an integer. It would be natural to apply *abs* to parallel variables $v1$ and $v2$ of shape *rowdim* thus:

```
with(rowdim) v2 = abs(v1);
```

The **with**-statement in C* activates as many (virtual) processors as there are elements in the given shape. These processors operate on the given parallel variables elementwise. This notation is normally used for all scalar operators and assignment. However, the presence of the call to the scalar function *abs* makes it illegal. To make it legal, the programmer must write a second function *abs* that takes a parallel variable of shape *rowdim* as parameter. Thus, the programmer must write at least two version of each function. Additional versions are needed for additional shapes, or shapes must be passed as parameters and a case analysis performed inside the function. The difficulties of keeping multiple versions of the same function consistent are well known to practicing software engineers.

This discussion points to a problem with the semantics of the **with**-statement. **With** creates multiple processors that operate in parallel, but when they reach a function call, only a single call is actually performed. Inside the function, however, the original processors come back to life. Thus, the parallel context seems to be conceptually suspended for the moment of the call, then resumed inside the procedure. Apparently, the specifics of the procedure call on a SIMD machine are reflected in the language definition. A better, fully consistent view would be to let the **with**-statement create as many processors as before, but let all of them execute the call of the (scalar) function. Since the processors operate synchronously, there is an efficient implementation even on a SIMD machine. A separate, parallel version of each function need not be written. These simplified semantics also accommodate nested parallelism.

4 Control Structures

The control constructs for loops and conditional statements are defined in an awkward way and fully synchronous execution may be too restrictive for efficiency. As an example, consider the following parallel loop in C*.

```
while (|= ( <parallel-condition> )) {
  where ( <parallel-condition> ) {
    statements
  }
}
```

The intent is that multiple processors execute the above loop simultaneously. The whole statement terminates as soon as `<parallel-condition>` evaluates to `false` in all processors. The operator `|=` in the first line, an OR-reduction, expresses this termination. It is awkward to be forced to repeat this condition in the second line. The careful programmer would evaluate the condition only once and then store it into a temporary, in order to prevent unwanted side effects and inefficiency. The repetition could easily be avoided and the compiler be burdened with the required code generation. At least this is how it was in the original C*.

The language definition as it stands also hurts performance on MIMD machines. The problem is the overly synchronous behavior required for loops: all iterations execute in complete lockstep, even if each loop operates on purely private data. This behavior is acceptable on a SIMD machine since the hardware forces that behavior anyway. But on MIMD machines this could hurt performance. Suppose the loops were allowed to run asynchronously, then some "natural" load balancing might occur. That is, suppose one processor executes the first iteration quickly and the next more slowly, while another processor exhibits the opposite behavior and thus the two processors finish in about the same time. With the present language definition, the processors are forced to run fully synchronously, and hence are slowed to the speed of the slowest one.

5 Conclusion

There are many other, small problems in C*, which we will not discuss further. (Among those are (1) that value parameters of only single shape can be passed to functions, (2) that `a += b` has not necessarily the same effect as `a = a + b`, (3) that vector operations are defined for parallel dimensions, but not serial ones, and (4) that the language is defined mostly by example and not by a precise statement of the semantics). While we consider the old C* a first and significant step in the right direction, it is dismaying to see so many of the old problems being carried over into the successor. It seems that elementary principles of language design such as machine-independence, orthogonality of constructs, consistency, and simplicity have not been taken into account sufficiently in new C*.

We believe that a much simpler extension of C suffices to realize data parallelism. One needs to add a single new statement, namely a synchronous *forall*, plus perhaps its asynchronous form. For data structures, one needs to introduce true multidimensional arrays plus pragmas that specify how to lay out the data. Such extensions have been implemented successfully in a compiler for the language Modula-2, targeting the Connection Machine[4, 3]. A simple and consistent extension of Modula-2 avoids all the problems mentioned above, without loss of efficiency. One might, however, call this work an unconfirmed experiment in language design and compiler construction. But this is exactly our point: More time is needed before we can standardize parallel programming languages.

At this time, design and compilation of parallel languages is in an experimental phase, as can be seen by the numerous proposals for such languages, but scant reports on experience with their implementation and use. However, knowledge in this area is increasing rapidly, so it would be imprudent to fix a poorly thought-out extension of a language as influential as C at this time.

References

[1] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, February 1990.

[2] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[3] Michael Philippsen and Walter F. Tichy. Compiling for massively parallel machines. In *Proc. of the Workshop on Code Generation, Schloss Dagstuhl.* Springer Verlag, May 20-24 1991. to appear.

[4] Michael Philippsen, Walter F. Tichy, and Christian G. Herter. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation*, September 1991. (under review).

[5] Michael J. Quinn and Philip J. Hatcher. Data-parallel programming on multicomputers. *IEEE Software*, pages 69–76, September 1990.

[6] Thinking Machines Corporation, Cambridge, Massachusetts. *C* Language Reference Manual*, April 1991.