The
Connection Machine
System

C* Release Notes

Versions 6.0 and 6.0.1
December 1990

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, December 1990

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a back-trace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
Usenet Electronic Mail:	ames!think!customer-support
Telephone:	(617) 234–4000 (617) 876–1111

1 About C*, Versions 6.0 and 6.0.1

C*® Versions 6.0 and 6.0.1 are the initial release of a new version of the C* data parallel programming language. This release supersedes all previous releases of C*. Programs written under these releases will not run under the new release. The new release has the following goals:

- To support data-parallel programming idioms that C programmers can understand and use effectively.
- To allow efficient access to all user-visible components of the Connection Machine system (for example, grid and general communication, scans, spreads, and reductions), so that coding in C* is almost as efficient as coding in the CM's parallel instruction set (Paris).
- To allow dynamic behavior.

2 Versions

The two versions of C* included in this release differ as follows:

- C* Version 6.0 works with Version 5.2 of CM system software.
- C* Version 6.0.1 works with Version 6.0 of CM system software.

Note that C version numbers do not correspond to CM System Software version numbers.* There are no differences in these C* versions other than their use with different versions of CM system software. Either Version 6.0 or 6.0.1 is installed as the default in your system. To use the version that is not the default, specify the **`-release`** option when compiling, as described in the *C* User's Guide*.

Back-compatibility mode is no longer required for C*.

To use the compiler for the pre-6.0 version of C*, issue the command **ocs** instead of **cs**. (Check with your system administrator; this version may no longer be available on your system.)

3 Documentation

The documentation for C* Versions 6.0 and 6.0.1 supersedes all previous C* documentation. Besides these release notes, it consists of the following:

- The *C* Programming Guide*, which describes how to program in C*.
- The *C* User's Guide*, which describes how to develop, compile, execute, and debug C* programs on a Connection Machine system.

In addition, a technical report that provides a reference description of the C* language will be available separately. Please note that this technical report includes descriptions of features of the language that are currently unimplemented; these features are listed in Section 4, below.

4 Unimplemented Features

The following features of C* and ANSI C have not yet been implemented:

- safety checking
- parallel bit fields
- parallel enumerated types
- the **physical_index** function
- assertion grammar
- non-constant subscripts for array declarations
- **const** and **volatile**

- scalar versions of the `boolcpy`, `boolmove`, `boolset`, and `boolcmp` functions
- trigrams
- shape axis alignment

In addition, please note that each dimension of a shape must be a power of 2, and the total number of positions in the shape must equal the number of physical processors in the CM, or be a power-of-2 multiple of this machine size.

4.1 Front-End C Compilers Are Not ANSI

The C* compiler works with the C compiler on your VAX or Sun front end. Currently, these compilers (in particular, their C pre-processors) do not conform to the ANSI standard for C. This means that you may be unable to take advantage of certain features of C that are new to the ANSI standard—for example, ANSI libraries. When compliant compilers are released, these features will be available.

This restriction does not apply to function prototyping, which is currently available through the C* compiler.

5 Restrictions

This section discusses known restrictions in the Version 6.0 and 6.0.1 release of C*.

5.1 Programs Can Become Too Big for the VAX Compiler

On a VAX, compiling a C* program may result in one of the following error messages:

```
Out of temporary string space
btw: Branch too far: Try -J flag
```

The problem is that the `..c` file produce by the `cs` compiler is too large for the VAX C compiler. (Using the `-J` flag won't help if you receive the latter message.) The workaround is to break up your source file into smaller files. Also, if you have long functions in your program, try splitting them up into smaller functions.

5.2 The rank Function Doesn't Work with a Segment Bit

The **`rank`** function in the communication library currently does not accept a segment bit for an argument. Specify either **`CMC_start_bit`** or **`CMC_none`** instead.

5.3 The write_to_pvar Function Doesn't Work with bools

The **`write_to_pvar`** function in the communication library currently does not work correctly when the front-end data to be written is an array of **`bools`**.

5.4 Grid Communication Functions Limited to 128 Bits

The grid communication functions have versions with a **`length`** argument that let you transmit data of any length. In the current version, the **`length`** argument must be 128 bits or less. The affected functions are listed below:

```
from_grid
from_grid_dim
to_grid
to_grid_dim
from_torus
from_torus_dim
to_torus
to_torus_dim
```

5.5 Incorrect Line Numbers

Infrequently, incorrect **#line** directives will appear in the intermediate **.c** file. Similarly, the line number may occasionally be incorrect in error messages from the compiler.

We would appreciate hearing from you if you run into either of these problems.

5.6 Problem with Error Recovery

Occasionally the compiler will find an error in a program, print a message, and then be unable to continue processing the program. Instead, it reports an internal compiler error.

We would appreciate hearing from you if you run into either of these problems.

5.7 Negation of Unsigned Constants

The compiler incorrectly optimizes the following code:

```
    unsigned u;
    float f;

    u = 200;
    f = -u;
```

To work around the problem, do not negate the unsigned constant. Instead, do the following:

```
    u = 200;
    f = -200;
```

5.8 Can't Redefine an enum Constant in an Inner Scope

An **enum** constant cannot be re-used as an identifier in a places where the **enum** definition is visible. For example, the compiler currently produces a syntax error for the following legal C code:

```
enum enuma { a };

main()
{
   int a;
}
```

The workaround is to use different names.

5.9 Shape-valued Expressions Are Re-evaluated in Parallel Variable Declarations

When using a shape-valued expression in declaring multiple parallel variables, note that the expression is incorrectly re-evaluated for each parallel variable. For example:

```
int:(s()) i, j;
```

If `s()` has side effects, `i` and `j` may not be allocated in the same shape.

The workaround is to assign the result of the function to a temporary shape, and declare the parallel variables to be of this shape. For example:

```
shape t = s();
{
   int:t i, j;
}
```

5.10 "Short-circuit" Operators Complain about Constant Expressions They Shouldn't Reach

The compiler evaluates constant expressions it shouldn't reach in operators like `||` and `&&`. For example, the following line of code:

```
int i = 1 || 2/0;
```

produces the error message

```
bad constant expression
```

even though the compiler should not evaluate the expression `2/0`.

5.11 Assigning an Address of a Scalar Array to a Pointer to a Scalar Array

Assigning the address of an array to a pointer to a scalar array produces a segmentation fault at run time. For example, the following legal code does not currently work:

```
main ()
{
  int i ;
  int b[2] ;
  int (*a)[2] ;

   a = &b ;

   i = (*a)[0] ;
}
```

For this code to work, compile the program with `-o0` to turn off optimization.

5.12 Warning Message when a Parallel Function Does Not Return a Value

When a parallel function does note return a value, the compiler generates a warning message like the following:

```
"(null)", line 0: warning: CMC_return_val not required in func-
tion foo
```

You can ignore this message.

6 Documentation Errors

6.1 Incorrect Description of send Function

The description of the parameters of the **send** function in Section 14.3.1, page 217, of the *C* Programming Guide,* states the source is a scalar pointer to a parallel variable. In fact, it is a parallel variable. The definition at the beginning of the section and the examples later in the section are correct.

7 Sample Programs

Sample C* programs are available on line. Consult your system administrator for their location on your system.

653899

Supplement to the C* Release Notes, Versions 6.0 and 6.0.1

The restrictions listed below were uncovered too late to be included in the Release Notes for C*, Version 6.0 and 6.0.1. Please add this sheet to the Release Notes.

Send Operations with Wrapping May Produce Incorrect Results

If you want to use a left-indexed send operation with wrapping (using **pcoord** and the **%%** operator), make sure that the second operand of the **%%** operator is equal to the number of positions in the dimension (for example, by using the **dimof** function). Using this syntax with other values may produce incorrect results. For example, the following code produces incorrect results if **n** is not equal to the number of positions in axis 1:

```
int n;
/* ... */
[.][(. + 1) %% n]a = a;
```

The workaround is to use the **make_send_address** and **send** functions from the C* communication library to do the send explicitly.

Local Shapes Are Not Deallocated after goto, continue, break, or return Statement

Shapes allocated within a function or block are not deallocated when you leave the function via a **return** statement, or the block via a **goto**, **continue**, or **break** statement. This could cause a program to run out of VP sets. One workaround is to structure your code so that it exits any block or function containing local shape declarations through the bottom; another is to move the shape declaration out of the function or block.