Llc Reference Manual

Russell C. Mills

Columbia University
Computer Science Department

17 May 1990

CUCS-432-89

Abstract

Llc is an extension of C for hierarchically parallel processing on distributed-memory parallel processors. In an llc program, a single controlling processor invokes operations in parallel in subsets of a set of attached processors, which themselves can invoke parallel operations in remaining processors. This manual succintly describes the language syntax and the implementation-independent aspects of its semantics.

This research was conducted as part of the Dado project. It was supported in part by the New York State Science and Technology Foundation NYSSTF CAT(88)-5 and by a grant from Hewlett-Packard. The author is an AT&T Graduate Fellow.

1 Introduction

Llc is an extension of C for hierarchically parallel processing on distributed-memory parallel processors. In llc, a single controlling processor invokes operations in parallel in subsets of a set of attached processors, which themselves can invoke parallel operations in remaining processors.

2 Retinues and Evaluating Retinues

An llc program executes on a partially-ordered pool of processors with a single minimal processor, the *principal processor*--in effect, a tree with the principal processor at the root. Each processor executing llc code can invoke computations that proceed in parallel in a subset of the processors higher in the partial ordering. At program startup, only the principal processor is active.

Each processor executing part of an llc program has associated with it a *retinue* of processors higher in the partial ordering that receive instructions from it, as well as an *evaluating retinue* of processors actively executing these instructions. Each processor is the *director* (or *directing processor* of its retinue.

The principal processor's retinue is the remainder of the processors. If Q is a processor in P's evaluating retinue for an operation M, Q's retinue, and its evaluating retinue, during its evaluation of M is the subset of P's retinue greater than Q in the partial order on the set of processors and not greater than any processor Q' in P's evaluating retinue that is itself greater than Q. In other words, each processor takes control of as many descendant processors as it can reach, but it cannot reach past a processor taking control of its own descendants.

Code executed by a processor is called *self code*. Code executed by a subset of a processor's retinue is called *retinue code*. Code executed by a processor's director is called *director code*. These terms are relative, since a processor's self code is director code to the processor's retinue, and a processor's retinue code is self code to members of the retinue. Retinue code can be embedded in self code in the following places:

- The **par** statement (section 4)
- The initializer of a retinue-tuple (section 2)
- The operand of a reduction operator (section 11 and the ^ operator (section 5)
- An evaluating retinue selector **with** (section 8) or **::** (section 9)
- A function argument, if the corresponding formal parameter is a retinue-tuple
- A **return** statement in a function returning a retinue-tuple

Director code can be embedded in self code in the following places:

- The **seq** statement (section 6)
- The operand of the !^ (*sequential*) operator (7).

A processor's retinue and evaluating retinue are preserved across function calls. That is, if a processor is executing a function **f**, and calls **g**, the processor's retinue and evaluating retinue at entry to **g** are the same as they were during the processor's execution of **f**. Further, the processor's retinue and evaluating retinue are restored at exit from **g**.

Llc has no explicit communications primitives, but some llc constructs cause communication of values from one processor to another. Different processors in the machine executing an llc program need not have the same data formats, but llc automatically performs appropriate format conversions:

- Llc converts atomic types (char, short, int, long, float, double) from one format to the other; this conversion may entail a loss of precision or of data.
- Llc converts pointer types, but a communicated pointer does not point to the same storage as the original pointer, since all pointers in llc point to storage in the processor storing the pointer.
- Llc converts struct types by converting each component. Llc converts array components by converting each element of the array.
- In general, llc converts union types by converting the first component. If a union is a component of a struct, and the preceding component is of integral type, llc treats that integer as a type tag for the union, and converts the union by converting the indexed component.

Llc has SIMD-like execution semantics, explained in more detail in section(4). Parallel code executes as if all processors in a given processor's evaluating retinue executed all operations synchronously.

3 Declarations

Llc allows a programmer to declare *retinue-tuples* of objects. A retinue-tuple contains one element in each processor in the declaring processor's retinue. To the standard ANSI C constructs for declaring derived types, namely,

- () function returning
- [] array of
- * pointer to

llc adds a single construct,

- ^ retinue-tuple of

The ^ operator is a prefix unary operator with precedence that of (*), and can be combined with the other declarator-forming constructs subject to the following restrictions:

- A struct or union can not contain retinue-tuples.
- There are no pointers to retinue-tuples.
- There are no retinue-tuples of retinue-tuples.
- There are no retinue-tuples of functions.

However, llc functions can have retinue-tuples as parameters, and can return retinue-tuples of any type that C functions can return.

Declarations of retinue-tuples obey the scope and extent rules of C. Outside a function, the visibility (lexical scope) of a retinue-tuple is the rest of the file; the object can be static or global; and its extent is the lifetime of the program. Inside a function, a retinue-tuple can be declared at the beginning of any block; it is visible only in the block in which it is declared; and it is static, or it is created at block entry and destroyed at block exit. Retinue-tuples in recursive functions are handled correctly.

Storage declared in a processor's retinue is called *retinue storage*, while storage declared in the processor itself is called *self storage*, and storage declared in a processor's director is called *director storage*. These terms are relative, just as the terms *self code*, *retinue code*, and *director code* are. Retinue storage and director storage can not be referred to in self code, even though declarations of the storage may be lexically visible. Instead, references to retinue storage must be embedded in retinue code, and references to director storage must be embedded in director code.

Names that do not refer to storage--names of functions, **typedef** names, and **struct** and **union** tags and components-- can be used wherever they are lexically visible. The compiler evaluates constant expressions appearing in declarations of **struct** and **union** types only once, in the context of the processor executing the declaration.

4 Par statement

Llc uses the **par** construct to invoke parallel execution:

par *statement*

All processors in the evaluating retinue execute *statement*, which is any legal llc statement not containing **goto**, and is retinue code. The execution semantics of *statement* are synchronous on an operator-by-operator basis; *statement* executes as if all processors in the evaluating retinue executed all operations in lockstep.

Conditional constructs in retinue code (the ||, **&&**, and **?:** operators, and the **if** and **switch** statements) execute as if the various paths through the code were evaluated in textual order, each path being evaluated in parallel by the appropriate set of processors:

- Processors in which the left-hand side (lhs) of a || operator is true drop out of the evaluating retinue until the end of the right-hand side (rhs).
- Processors in which the lhs of a **&&** operator is false drop out of the evaluating retinue until the end of the rhs.
- Processors in which the first operand of a **?:** is true execute the second operand, then drop out of the evaluating retinue until the end of the expression; processors in which the first operand is false drop out of the evaluating retinue until the end of the second operand, then evaluate the third operand.
- Similarly, processors in which an **if** condition is false drop out of the evaluating retinue until the **else** clause or the end of the **if** statement.
- After executing the control expression of a **switch** statement, all processors drop out of the evaluating retinue. A **case** label with value equal to that of the control expression, or a **default** label if no **case** label equals the value of the control expression, restores a processor to the evaluating retinue, while a **break** statement removes a processor from the evaluating retinue. The end of the **switch** restores the pre-**switch** evaluating retinue.

Looping constructs (**while**, **do...while**, and **for**) in parallel code execute by contracting the evaluating retinue at each loop iteration until the evaluating retinue is empty; when the evaluating retinue is empty, the loop terminates, and the pre-loop evaluating retinue is restored. A **break** statement removes a processor from the evaluating retinue until the end of the loop; a **continue** statement removes a processor from the evaluating retinue until the end of the current loop iteration.

5 ^ operator

The ^ (*retinue*) unary operator is the expression analog of the **par** statement. Processors in the evaluating retinue evaluate the operand; the value of the ^ expression is one of the values from the evaluating retinue. If the evaluating retinue is empty, the value of the expression is undefined. The ^ operator can be applied to an operand of any type, even composite, and the type of the result is the type of the operand.

6 Seq statement

Llc uses the **seq** construct to invoke sequential execution in the directing processor during parallel execution in the director's retinue. The statement

seq *statement*

when embedded in a **par** statement causes *statement* to be executed in the processor invoking the **par** statement, provided that the set of processors invoking the **seq** statement is not empty.

Notice that a declaration of self storage inside **par** *statement* is equivalent to a declaration of retinue storage just outside **par** *statement*. The program fragment

```
{
    int ^i;
    statement
}
```

is equivalent to

```
par {
    int i;
    seq statement
}
```

7 !^ operator

The !^ (*director*) prefix unary operator is the expression analog of the **seq** statement. The expression

!^*expression*

causes evaluation of *expression* in the director of the set of processors evaluating !^*expression*, provided that set is not empty. The type of !^*expression* is the type of *expression*. If *expression* is a call to a function that returns a retinue-tuple, the value of !^*expression* in each processor is the value returned by the function in that processor. Otherwise, *expression* returns a single value, which is also the value of !^*expression*.

8 With statement

Llc's **with** construct defines the evaluating retinue:

with (*retinue-expression*) *self-statement*

All processors in the pre-**with** evaluating retinue of the processor executing the **with** statement evaluate *retinue-expression*. The processor executing the **with** statement then executes *self-statement*, which is self code, but which may contain retinue code or call functions that contain retinue code. The evaluating retinue for *self-statement* is the subset of processors where *retinue-expression* is true. The end of the **with** statement restores the pre-**with** evaluating retinue. *Retinue-expression* can be prefaced by the keyword **all**, in which case the evaluating retinue for *retinue-expression* is the entire retinue. If **all** is included, *retinue-expression* may be omitted, and is implicitly **1**.

9 :: operator

The **with** statement has an expression analogue, the :: (*with*) operator:

self-expression :: *retinue-expression*

All processors in the evaluating retinue evaluate *retinue-expression*; the evaluating retinue for *self-expression* is the subset of processors where *retinue-expression* is true. The end of the :: expression restores the pre-:: evaluating retinue. As in the **with** statement, *retinue-expression* can be prefaced by the keyword **all**. The precedence of the :: operator is between that of assignment operators and the sequencing operator (,).

10 ? operator

The llc **?** (*select*) prefix unary operator allows the programmer to single out one processor. The **?** operator can be used only in retinue code, and its operand *expression* must be of numeric type. The value of

?*expression*

is 0 in all processors but one, and 1 in one processor in the evaluating retinue where *expression* is nonzero. Llc does not specify which processor receives the non-zero value, but guarantees that **?** chooses the same processor each time from a given set of values in a set of processors.

11 Reduction operators

A reduction operator is a unary operator that operates pairwise on a set of values to produce a single result. Llc provides a number of built-in reduction operators, described below, and also allows programmers to define their own. The precedence of reduction operators in llc is that of the unary operators in C.

11.1 Built-in operators

Llc provides the collection of reduction operators shown in figure 1. Each built-in reduction operator is the unary counterpart of a commutative, associative binary operator.

Two of these reduction operators stretch a semantic point for syntactic convenience. Strictly speaking, || and **&&** are not commutative operators in C, since they do not evaluate their right operands if the value of the expression can be determined from the left operand. The reduction operators ||/ and **&&/**, however, evaluate all their operands in parallel and produce the logical OR or AND of all the values.

11.2 User-defined operators

Llc provides reduction operators corresponding to most of C's binary operators, but it also allows programmers to define new reduction operators. Any function of two arguments may be used as the combining code in a reduction operator. A declaration of a function named *f* with

function-declarator **default** *constant-expression;*

or a definition of *f* with

function-declarator **default** *constant-expression function-body*

creates a new unary reduction operator **f/**. Then an expression of the form

f/*retinue-expression*

causes *f* to be applied as a reduction operator to the values of *retinue-expression* in the evaluating retinue of processors. If the evaluating retinue is empty, the result is the declared default value.

Figure 1: Reduction Operators

+/	sum of the operands
*/	product of the operands
max/	maximum of the operands (**max** is a built-in binary operator in llc)
min/	minimum of the operands (**min** is a built-in binary operator in llc)
|/	bitwise OR of all operands
&/	bitwise AND of all operands
^/	bitwise exclusive OR of all operands
||/	1 if any operand evaluates to a non-zero value; 0 otherwise
&&/	1 if all operands evaluate to a non-zero value; 0 otherwise

12 Local and nonlocal functions and function calls

An llc compiler can often produce substantially better code if it knows whether functions called from retinue code or from directing code contain retinue code or call functions that do. A **local** function declaration or call advises the compiler that the specified function or call does not invoke retinue code; a **nonlocal** declaration or call advises the compiler that the function or call does invoke retinue code. The **local** and **nonlocal** keywords modify a standard function declarator, and are placed before the parentheses; the syntax mirrors that of the **volatile** and **const** pointer modifiers or ANSI C. For example, the declaration

```
int f nonlocal(int i);
```

declares f to be a nonlocal function of one **int** argument, which returns an **int**. The program fragment

```
{
    int atoi(), (*fp)() = atoi;
    char *s;

    par {
        int i = !^(*fp) local(s);
    }
}
```

advises the compiler that the call to **fp* invokes no retinue code.