The llc Parallel Language and its Implementation on Dado2

Russell C. Mills

Columbia University
Computer Science Department

17 May 1990

CUCS-429-89

Abstract

Llc is an extension of C that has been implemented on the massively parallel Dado2 tree-structured MIMD multicomputer. 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. Language features include distributed storage (objects with one element per processor), the ability to pass parallel values as function arguments and to return them from functions, and built-in and user-defined reduction operators. The paper first describes the synchronous model of hierarchically parallel computations used by llc. It next describes the syntax of the llc language and its semantics in detail. It then describes the Dado architecture and the implementation of llc on Dado2. A summary of the of the strengths of llc follows. Finally, several examples of llc programs help to clarify the important features of the language.

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

One of the goals of the research work on the Dado project has been to construct data-parallel languages [7] in which operations executing in parallel can invoke subsidiary data-parallel operations. Toward this end we have designed the llc language and implemented it on Dado2, a 1023-processor tree-structured MIMD [1] prototype machine at Columbia University. Llc is a hierarchically data-parallel extension of the C language [3, 4] in which 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. Language features include globally- and locally-scoped distributed storage (objects with one element per processor), parallel evaluation of expressions and statements with synchronous semantics, the ability to pass parallel values as function arguments and to return them from functions, and built-in and user-defined reduction operators.

An llc program is a single program in which a number of operations may be performed simultaneously; it runs on a number of processors, but these processors share a single global name space. Llc contains no constructs for communications as such; the run-time environment transparently transmits values computed in one processor but used in another.

Llc is the main system-level parallel language (*cf.* [12]) for the Dado machine. It has been implemented efficiently on Dado2; the language can be implemented on other distributed-memory MIMD parallel machines, since it requires only that a tree be logically embeddable in the machine's communication network. In fact, the only major changes required to port the language are the implementation of communication functions called by the translated code.

Experience with the system-level parallel languages PPL/M [10] and ||PSL [12] for Dado influenced the design of llc. The three languages share the notion of distributed storage with one element per physical processor, provide inter-processor communication, and provide mechanisms for selecting the set of processors in which parallel operations occur. However, in llc, but not in PPL/M or ||PSL, distributed storage can be locally scoped as well as globally scoped, inter-processor communication is implicit, not explicit, and the effects of all mechanisms for selecting the domain of parallel operations are lexically scoped rather than having indefinite lifetime.

The paper first describes the synchronous model of hierarchically parallel computations used in llc. It next describes the syntax of the llc language and its semantics in detail. It then describes the Dado architecture and a completed implementation of llc on Dado2. A summary of the of the strengths of llc follows. Finally, several examples of llc programs help to illustrate some important features of the language.

2 Hierarchically Parallel Computations

An llc program executes in a partially-ordered pool of processors with a single minimal processor, the *principal processor*. If P is a processor, we will call a processor greater than P in the partial ordering a *descendant* of P. In effect, llc assumes a tree of processors with the principal processor at the root. Only the principal processor executes *main()*, so at program startup, only the principal processor is active. The principal processor invokes operations in parallel in its descendant processors; each such processor can itself invoke parallel operations in its descendants. Each processor executing part of an llc program has associated with it a *retinue* of descendant processors that receive instructions from it, as well as a subset

of its retinue (the *evaluating retinue*) that actively executes those instructions. Each processor is the *director* (or *directing processor*) of its retinue.

Llc provides SIMD-like execution semantics. All parallel operations execute with synchronous semantics, that is, as if all processors executing the operations performed them in lockstep. Execution is deterministic and repeatable. The language therefore contains no synchronization constructs, since synchronization at each operation is implicit. Actual execution, however, need not be synchronous if each processor is capable of executing its own stored program.

A processor's retinue can now be defined formally in terms of operations performed synchronously in parallel by sets of processors. 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 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, during each operation, 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. Only the principal processor's retinue is fixed; the retinue of any other processor depends on what operations that processor and all its descendants are performing.

3 Distributed Storage

The llc language allows a programmmer to declare objects either in a single processor or (in data-parallel fashion) in each member of the processor's retinue. A *retinue-tuple* is storage created by a declaration (like all objects in C) and consisting of one element in each processor in the declaring processor's retinue. Clearly, the number of elements in a distributed object depends on the number of processors in the machine executing a program.

To the standard C constructs for defining derived types, namely

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

llc adds a single construct,

- ^ retinue-tuple of

which is a prefix unary operator with precedence that of (*). Retinue-tuple (^) declarations may be mixed with other C type-defining constructs in almost any fashion. A few restrictions apply:

- Structs and unions can not contain retinue-tuple components.
- 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. Figure 1 illustrates some declarations. Notice the C++-style [11] function declarations, which have been included in ANSI C [4].

Figure 1: Some llc Declarations

```
int ^i;              i is a
                         retinue-tuple of
                             int

int (^j)[5];         j is a
                         retinue-tuple cf
                             array of 5
                                 int

char *^demand();
                     demand is a
                         function returning
                             retinue-tuple of
                                 pointer to
                                     char

void pmemcpy(char *^, char *, unsigned int);
                     pmemcpy is a
                         function of (
                             retinue-tuple of
                                 pointer to
                                     char,
                             pointer to
                                 char,
                             unsigned int
                         ) returning
                             void
```

All retinue-tuple declarations obey the scope and extent rules of C. Outside a function, the visibility (lexical scope) of a distributed object is the rest of the file; the object can be **static** or **extern** (global); and its extent is the lifetime of the program. Inside a function, a distributed object 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. Distributed objects 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, since a processor's self storage is director storage to the processor's retinue, and a processor's retinue storage is self storage to members of the retinue.

4 Embedded Code

All parallel computations in llc are invoked through code executed by members of a processor's retinue (*retinue code*) but embedded textually in code executed by that processor (*self code*). Symmetrically, computations in that processor can also be invoked through code embedded textually in code for its retinue. The terms *retinue code* and *self code* are relative, just as *retinue storage* and *self storage* are, since code that is retinue code to a processor is self code to the processor's retinue, and self code for a processor is *director code* to the processor's retinue. The syntactic constructs that embed director and retinue code in self code are the only mechanisms llc provides for the communication of values between processors.

Llc contains a number of constructs that embed retinue code in self code, and that invoke parallel

execution of statements and expressions. The set of processors that evaluate embedded retinue code is exactly the evaluating retinue of the processor executing the surrounding code; thus all parallel operations take place in the evaluating subset of the retinue. The following constructs embed retinue code in self code:

- **Par** statements
- Initializers of retinue-tuples
- Reduction operators and the ^ operator
- Evaluating retinue selectors **with** and **::**
- Function arguments, if the corresponding formal parameters are retinue-tuples
- **Return** statements in functions returning a retinue-tuple

In addition, two constructs embed self code in retinue code:

- The **seq** statement
- The **!^** (*sequential*) operator

The next few sections describe these constructs.

5 Parallel and Sequential Computations

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

par *statement*

All processors in the evaluating retinue execute *statement*, which is any legal Ilc statement not containing **goto**. **par** *statement* is self code (relative to the code textually preceding and following it) containing the embedded retinue code *statement*. As a point of terminology, we will say that the processor executing the code surrounding **par** *statement* executes **par** *statement*, while the members of that processor's evaluating retinue execute *statement*.

Since the subset of processors that execute embedded retinue code is exactly the evaluating retinue, conditional expressions (**&&**, ||, and **?:**), conditional statements (**if** and **switch**), and looping statements (**for**, **while**, and **do...while**) in embedded retinue code affect the evaluating retinue in a natural way. For example, processors in which an **if** condition is false drop out of the evaluating retinue until the **else** clause (if one is present) or the end of the **if** statement. Processors executing a looping statement (**for**, **while**, and **do...while**) in retinue code drop out of the evaluating retinue when they exit the loop; when the evaluating retinue becomes empty, they rejoin the evaluating retinue, and the loop terminates.

Conditional constructs in embedded retinue code 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. Separate paths through conditional and looping constructs are executed concurrently if the code on each path operates only on local values.

While the **par** statement embeds retinue code in self code, Ilc's **seq** statement embeds self code in retinue code embedded in self code. The statement

seq *statement*

which must be contained in a **par** statement, causes *statement* to be executed in the processor executing the **par** statement. In effect, the **seq** statement cancels the enclosing **par**. As another point of terminology, we will say that **seq** *statement* is retinue code containing the embedded self code *statement*, and that the evaluating members of the retinue execute **seq** *statement*, while the retinue's director executes *statement*.

In the statement

seq *statement*

the directing processor executes *statement* only if the set of processors executing **seq** *statement* is nonempty, and then only once. Certainly, the directing processor should not execute *statement* if the evaluating retinue is empty. Otherwise, it would execute *statement* in the code fragment

```
par {
    if (0) {
        seq {
            statement;
        }
    }
}
```

which would be highly counterintuitive. Llc's synchronous semantics dictate that the directing processor execute *statement* only once, rather than once for each processor in its evaluating retinue. Otherwise, the directing processor would have to interleave multiple executions of a single piece of code at an instruction level, rather than a statement level, with unpredictable results.

Notice that a local retinue-tuple declaration is almost equivalent to a declaration of a non-retinue-tuple object within a **par** block enclosing a **seq** block:

```
{
    int ^i;
}
self-code
```

is nearly the same as

```
par {
    int i;
    seq {
        self-code
    }
}
```

the only difference being that in the second example, the processor executes *self code* only if its evaluating retinue is nonempty. Retinue-tuple declarations like the one in the first example are necessary, not just a convenience to the programmer, precisely because the two examples are not identical.

6 Moving Values Between Processors

Self code cannot refer to retinue storage and director storage, even though declarations of the storage may be lexically visible. Instead, references to retinue storage must be contained in embedded retinue code, and references to director storage must be contained in embedded director code. Llc provides two unary operators, the expression analogs of the **par** and **seq** statements, that embed retinue code and director code in self code. Unlike the **par** and **seq** statements, these operators communicate values among self, director, and retinue processors.

The ^ (*retinue*) unary operator, with syntax

^ *expression*

is the expression analog of the **par** statement. Processors in the evaluating retinue evaluate *expression*; the value of the ^ *expression* is one of the values produced. If the evaluating retinue is empty, the value of the expression is undefined. The ^ operator works on all data types--integer, floating-point, and composite. If the directing processor's data formats are different from the retinue's, appropriate format conversion happens automatically, even for structs. An example:

```
/* set i to (the value of j in one processor in the retinue) */
{
    int ^j = f();
    int i = ^j;
}
```

The ^ unary operator, like the * unary operator, is a type-former as well as a unary operator. The use of ^ as a unary operator mirrors its use in declaring derived types. In the above example, *j* is a retinue-tuple of **Int**, so *^j* is an **Int**.

The ^ operator is useful in two situations:

1. in invoking parallel execution in a syntactic context that requires an expression, not a statement, such as the initialization of a **for** loop;

2. in transferring a value from a retinue processor to the director of the retinue (in fact, the ^ operator is the only construct that provides such communication).

Llc's !^ (*director*) unary operator, with syntax

!^ *expression*

is the expression analog of the **seq** statement. It causes the evaluation of its operand in the directing processor, and makes the result available in each evaluating retinue processor. 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. An example:

```
/* set each j to the value of i in the directing processor */
{
    int i = f();
    int ^j;

    par j = !^i;
}
```

Notice that in the **par** statement, references to *j* do not require the ^ operator, since to each retinue processor, j is an **Int**, not a retinue-tuple of **Int**.

The !^ operator is useful in three situations:

1. in invoking sequential execution in the directing processor in a syntactic context that requires an expression, not a statement;

2. in broadcasting a value to retinue processors from the director of the retinue (in fact, the !^ operator is the only construct that transfers values from director to retinue);

3. in using the value returned by a function that returns a retinue-tuple, as described in section(RetinueTupleParameters)

Llc provides no automatic conversion of sequential values to parallel, requiring the programmer to use the ^ and !^ operators even when the translator can determine the unambiguous meaning of an expression. In large part, this design decision reflects the belief that the distinction between sequential and parallel execution is important, and should be reflected in the immediate text of the program. Moreover, the translator must sometimes rely on the !^ and ^ operators to disambiguate expressions, as in the following example:

```
par {
        !^f();
        f();
}
```

In this example, the retinue's director (the processor executing the **par** statement) calls *f*; then each evaluating retinue processor calls *f*. Without the !^ operator, the translator would not be able to distinguish the two uses of *f*. Notice that functions in llc, unlike storage, are accessible in any processor; a function declaration (implicit in this example) makes the function callable in any processor, not just the processors making the declaration.

In order to make the llc implementation efficient, the !^ operator can be used only in retinue code embedded in self code. It is therefore illegal for a function called from embedded retinue code to refer to director storage, even if the text of the function could refer to the storage were the function substituted textually into the embedded retinue code. The reason for this restriction is that a directing processor cannot handle communication with its retinue efficiently if each processor in the retinue can make communication requests that the llc translator cannot anticipate.

7 Reduction Operators

Llc provides a number of *reduction operators*. Each llc reduction operator uses a commutative, associative llc binary operator to combine a set of retinue values pairwise to produce a single self value. Llc borrows its syntax for reduction operators from APL [2]; the language provides the collection of reduction operators shown in figure 2.

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 local results.

A reduction operator's operand is embedded retinue code; the reduction operator itself is self code. Each reduction operator evaluates its operand in the evaluating retinue and combines the values using the corresponding binary operator to produce a single value. Thus, llc's reduction operators implicitly communicate values from a processor's retinue to the processor.

In addition to providing reduction operators corresponding to most of C's binary operators, llc also allows programmers to define new reduction operators. Any function of two parameters can be used as the combining code in a reduction operator. A declaration of the form

function-declarator **default** *default-value;*

declares *function* to be usable as combining code. Then an expression such as

function/ retinue-expression

causes *function* to be applied as a reduction operator to the values of *retinue-expression* in the evaluating retinue. If the evaluating retinue is empty, the result is *default-value*. For example, the declaration

Figure 2: Reduction Operators

+	sum of the operands
*	product of the operands
>?	maximum of the operands (**max** is a built-in binary operator in llc)
<?	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

---

:) default 0;
on of two **int** parameters returning an **int** and with value 0 when reducing over an
).

iluating Retinue

nstructs we have seen that affect a processor's evaluating retinue are C control
l retinue code. In addition to these implicit means for selecting a processor's
o provides an explicit mechanism: the **with** statement and its expression analog,
constructs select the evaluating retinue that a processor has during its own
or expression; thus they define the set of processors that evaluate any retinue
atement or expression. The effects of the **with** statement and the :: operator are
l of the statement (or expression) restores the previous evaluating retinue.

lines the evaluating retinue for a statement:
ssion) *self-statement*
self code containing the embedded retinue code *retinue-expression*. All
uating retinue of the processor executing the **with** statement evaluate
those processors in which the expression is true (nonzero) remain in the
-statement. *Retinue-expression* can also be prefaced by the keyword **all**, in
ssors evaluating *retinue-expression* is the directing processor's entire retinue.

the **with** statement may help to illustrate its meaning. If *retinue-expression* is
or, the statement
sion) **par** *retinue-statement*
nt
ssion) *retinue-statement*

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

is equivalent to

par if (*retinue-expression*) **seq** *self-statement*

The **with** statement has an expression analog,

self-expression **::** *retinue-expression*

As above, the set of those processors where *retinue-expression* is true becomes the evaluating retinue for *self-expression*. The **::** operator is useful in syntactic contexts requiring an expression.

Evaluating retinues nest syntactically; if one **with** statement encloses another, the set of processors that evaluate the inner **with**'s condition is just the set of processors in which the outer **with**'s condition is true. Thus the two statements

with (*retinue-expression-1*)
with (*retinue-expression-2*)
self-statement

and

with (*retinue-expression-1 && retinue-expression-2*)
self-statement

mean the same thing. Functions also inherit their caller's evaluating retinue, so that

with (*retinue-expression*) *body-of-self-function;*

is the same as

with (*retinue-expression*) *self-function();*

which is identical to

self-function() **::** *(retinue-expression);*

Self code can operate on retinue storage only in parallel on each element of the storage, but programs often need to work sequentially with the elements of a distributed object. For example, a program may need to load the elements of an array into a retinue-tuple, where the elements of the array can be manipulated in parallel. To handle this need, llc provides the **?** (*choose-one*) unary operator. The operand of the **?** operator is an expression of numeric or pointer type, and the operator returns the value 0 in all processors but one, while it returns 1 in one processor in which the operand is non-zero. Llc does not specify which processor receives the non-zero value, but guarantees that **?** chooses the same processor each time from a given pool of processors and non-zero values. Like the **!^** operator, the **?** operator can be used only in embedded retinue code, but unlike the **!^** operator, both its operand and its result are retinue values.

The **?** operator is useful for selecting a single data item (for example, one for which the value of some expression is minimized) from a collection of data. Iteration through the elements of a retinue-tuple can also be effected quite easily. If *done* has been declared to be a retinue-tuple of **Int**, the statement

```
for (^(done = 0); ||/(done == 0); ^(done = 1) :: ?(done == 0)) ;
```

sets *done* to 1 in one processor during each loop iteration, and terminates when *done* is 1 in each processor in the **for** statement's evaluating retinue.

9 Retinue-tuple Parameters and Return Values

In llc, retinue-tuples are almost, but not quite, first-class objects. While retinue-tuples cannot be assigned to or operated on directly (their elements can be operated on in parallel), llc allows functions to have retinue-tuple parameters and to return retinue-tuple values. Code that calls a function with a retinue-tuple argument causes each member of the calling processor's evaluating retinue to generate an argument for the function; the processor calling the function then executes it in cooperation with its retinue. Likewise, a function returning a retinue-tuple causes each member of the calling processor's evaluating retinue to generate a return value for the function. Llc uses C++'s syntax for declaring function parameter types and return values (see Figure 1). Functions with retinue-tuple parameters or return values must be declared as such, or the translator will generate incorrect code.

If a function has been declared to have retinue-tuple parameters, the corresponding arguments in a call to the function are retinue code, even though the call itself is self code. To illustrate the use of retinue-tuple parameters, here's a function that uses the parallel memory copy function *pmemcpy* declared in Figure 1 to distribute a string to all evaluating retinue processors.

```
void distribute(char *from)
{
    int len = strlen(from) + 1;
    char *malloc(unsigned int);
    char *^to = malloc(!^len);

    pmemcpy(to, from, len);
}
```

In the text of a function returning a retinue-tuple, any **return** statements are self code executed by the processor executing the function, but the expression returned is retinue code. Using the retinue-tuple of values returned by a function is a standard llc idiom. For example, the following code fragment demonstrates how to use the values returned by the function *demand* from Figure 1):

```
par {
    int (*fp)() = (int (*)())!^demand("f");

    (*fp)();
}
```

In this example, the processor executing the **par** statement calls *demand()*; each processor in the evaluating retinue cooperates with the directing processor to compute a return value, which it stores in *fp*. The !^ operator in this example is essential, since the call to *demand* is director code relative to the use of the function's return value.

Retinue-tuple parameters and return values are very important in llc, because they allow programmers to modularize their code. A typical llc program contains many functions that are purely local, but also contains large chunks of intermingled sequential and parallel code. Since a function executing in a retinue processor cannot refer to storage in its directing processor, code that refers to both retinue and self storage must be part of a function in the retinue's director. Experience with a previous parallel language, PPL/M [10], showed that the inability to break up the intermingled code led to intolerably large procedures and excessive reliance on global distributed variables for communication between functions.

10 Local and Nonlocal Functions

The llc translator can produce substantially better code if it knows that functions called from retinue code or from directing code invoke no retinue code directly or indirectly. A **local** function declaration or call advises the compiler that the specified function contains no retinue code and that it calls no functions that do. 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. 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.

11 Dado2 Architecture

This section summarizes the Dado2 architecture so that we can proceed to a description of a completed implementation of llc on Dado2. For a more complete description of the Dado architecture, see [5, 9].

Dado is a medium-grain, tree-structured, distributed-memory, massively parallel MIMD multicomputer originally intended to accelerate the execution of AI production systems, but applicable to a wider range of problems. The largest version, Dado2, contains 1023 processors connected in a complete binary tree. Dado2 is attached as a coprocessor to a conventional computer, which is the principal processor in an llc program. To the programmer, the host computer and the Dado machine together form an unbalanced binary tree whose root is the host computer; the Dado root processor is the left child of the host.

Each processor consists of an 8-bit microprocessor, 64K bytes of RAM with parity, and a semi-custom I/O chip. Dado2 has two separate communication channels, each a byte wide. The microprocessor ports allow a processor to communicate with its *tree neighbors* (parent, left child, and right child). The I/O chip provides partitionable global communication via a broadcast circuit and a resolve/report circuit (a multi-byte minimum-computing comparator). During each resolve/report operation, the comparator flags a single processor that has the minimum value. Dado2 communication cannot be interrupt-driven; processors must anticipate and participate actively in each byte of all communication.

The Dado2 I/O chip can communicate through all or part of the Dado2 tree. A memory-mapped I/O chip register controls the connections of each I/O chip to its tree neighbors' I/O chips, so that any processor can disconnect itself from its parent or its children from itself. At any point in a program's execution, the I/O chip connections divide the Dado2 machine into a number of connected components. Within a connected component, all communication through the I/O chip is global and synchronous; all processors in the connected component must participate in each communication.

12 Implementation of llc on Dado2

As with the Poker [8] parallel programming language, the heart of the llc implementation is a source-to-source translator that converts the parallel language (llc) to the sequential base language (C). The llc translator converts an llc program for the host-Dado2 ensemble into C code for the host machine and for the Dado2 processors. The generated C code contains calls to interprocessor communications functions [6]. The translator splits each llc function into a C function for the processor executing the llc function and a C function for its retinue. It converts each self statement containing embedded retinue code into a labeled retinue statement and a self statement containing a broadcast by the directing processor of the address (determined at link time) of the retinue statement. At runtime, retinue processors repeatedly read addresses broadcast by their directing processors and execute the corresponding code. In effect, the translator generates SIMD code for a directing processor's retinue. Instead of broadcasting machine instructions, however, the directing processor broadcasts pointers to parallel code of arbitrary complexity.

When a processor enters an llc function containing embedded retinue code, it broadcasts to its retinue the address of the translator-generated parallel code for the function, and at function exit, it broadcasts the address of a **return** statement to force its retinue to leave the corresponding parallel function. This scheme makes it easy to implement retinue-tuple local variables, function arguments, and function return values. Retinue-tuple local variables in an llc function simply become local variables in the corresponding translator-generated retinue function. Retinue-tuple function arguments become locally-computed arguments to a translator-generated function, while retinue-tuple return values become locally-computed return values of a translator-generated function.

Only those processors in the evaluating retinue actively execute retinue code; the translator generates expressions conditioned on a processor's evaluating state. The translator handles llc constructs in the following ways:

- For conditional and looping constructs in embedded retinue code, and for **with** statements and **::** operators, it generates code that saves, manipulates, and restores the processor's evaluating state.

- For the **!^** operator, it generates calls to Dado2 I/O chip broadcast functions: a write for the directing processor, and a corresponding read for the processors in its retinue.

- For the **^** operator, the **?** operator, and the reduction operators **min/**, **max/**, **||/**, and **&&/**, it generates calls to Dado2 I/O chip resolve/report functions: a read for the directing processor, and a corresponding write for the processors in its retinue.

- For other reduction operators, it generates calls to support functions that read values from a processor's children, combine those values with its own value, and (in a retinue processor) write the result to the processor's parent.

- For code in which processors in a directing processor's retinue may themselves invoke parallel operations, the translator generates instructions to manipulate the state of the I/O chip's connections to its tree neighbors. Before such code, the translator places instructions to save the state of the I/O chip's connections to its children and disconnect any children in the current evaluating retinue; after the code, the translator inserts instructions that restore the I/O chip's state.

Because of the Dado2 hardware requirement that a processor's entire retinue participate in all communications while only the evaluating retinue participates in computations, the translator must separate communication from computation, but it must do so without affecting the semantics of the code.

Since communication and computation can be very tightly interwoven, the translator must perform extensive analysis and reorganization of the source code. For this reason, the llc translator is really a compiler, not a simple preprocessor.

13 Related Work

Experience with the system-level parallel languages PPL/M [10] and ||PSL [12] for Dado influenced the design of llc. The three languages share the one-element-per-processor limitation. PPL/M requires the programmer to write explicit communications instructions in parallel code; ||PSL asks the programmer to write these instructions in sequential code. Llc contains no explicit communications instructions.

Among other parallel languages, C* for the Connection Machine [7] is most like llc. Even though llc was developed independently from C*, the two languages share some major features. They both rely heavily on reduction operators and on parallel operations on subsets of distributed objects, because both are natural and powerful constructs that reduce the complexity of programming a massively parallel computer.

Llc differs from C* in the structure of its distributed objects, which are mappings from the set of physical processors to the set of possible values of the element type. In C*, the programmer declares the cardinality of a distributed object, and the compiler allocates a processor to each element of the object; if the target machine does not have enough processors, the program cannot run. The cardinality of a program's distributed objects determines the program's speedup. In llc, the runtime environment determines the cardinality of a distributed object, and for problems in which there are more data than processors, and the number of processors in the target machine determines the program's speedup.

Llc differs from C* in a number of other ways:

- Llc provides multiple levels of parallelism and MIMD execution, while C* provides only a single level.
- Llc provides distributed function arguments and return values, which C* apparently does not.
- Llc allows declarations of distributed objects in sequential code.

14 Conclusion

As part of the Dado project, we have designed an implemented llc, a system-level parallel language consisting of a few simple extensions to the sequential language C. This language combines powerful operators, hierarchically (even recursively) parallel operations, a tight interweaving of sequential and parallel code, efficient implementation, and access to most of the capabilities of the Dado machine.

As a system-level parallel language, llc has many strong points:

1. *Powerful constructs.* Llc's reduction operators and constructs for invoking parallel operations in subsets of the set of attached processors are powerful and allow programmers to write compact, clear programs.
2. *Simplicity.* Llc stays within the spirit of C by providing a single widely applicable construct for declaring distributed objects, and a small set of constructs for controlling parallel execution.

3. *Expressiveness.* Local distributed variables provide convenience; distributed function arguments and return values facilitate modular programming.

4. *Deterministic execution.*

5. *No artificial barrier between sequential and parallel code.* Parallel and sequential code in llc can be tightly interwoven.

6. *Safety.* Llc does not require the programmer to write two sides of each communication. Instead, communication is implicit, and the translator generates matching communication instructions. It is impossible to write a program that deadlocks because a communications instruction has been omitted.

7. *Efficiency.* No language feature is implemented inefficiently on Dado2.

8. *Full use of the machine.* Llc provides full access to the capabilities of the Dado machine. Even tree-neighbor communication, which is not supported explicitly in the language, can be written quite efficiently using parallel constructs in recursive functions.

9. *No MIMD/SIMD distinction.* Llc provides clear SIMD-like semantics for parallel execution in any processor's retinue. Each processor always has a retinue and is in another processor's retinue.

15 Acknowledgements

Many discussions with Richard Reed helped shape llc. Leland Woodbury wrote llcc(1), which coordinates the llc and C preprocessors and compilers, assemblers, linkers, and utilities for the host and Dado processors. Sal Stolfo, my adviser, was very helpful with his support and his ideas. Perry Metzger made important criticisms of earlier versions of llc; Chip Maguire and George Shrier also contributed criticisms. Sally Lewis and Leland Woodbury made many helpful comments on earlier drafts of this paper.

I. A Small Example of Hierarchical Parallelism in llc

```
/*
 * this program computes the level of each processor
 * in the Dado machine
 * It assigns 0 to the principal processor and recursively
 * computes level = parent's level + 1
 */

int ^level;
int setlevel nonlocal();

void
main(int argc, char *argv[])
{
    setlevel(0);
    par gprintf("self %x  level %x\n", self(), level);
}

#pragma self
#pragma retinue
int
setlevel(int n)
{
    par {
        if (parent() == !^self()) level = setlevel(!^n + 1);
    }
    return (n);
}
```

II. An Extended Example in IIc

```
/*
 * Count the number of times some strings
 * appear in a collection of strings.
 * First distribute a collection of strings
 * (read one per line from stdin)
 * among the retinue of the principal processor.
 * Then count the number of times strings
 * passed as arguments to main() occur in the distributed collection.
 */

#include <stdio.h>

/* a string for this program is an array type */
typedef char STRING[1024];

/* each processor stores a list of strings */
typedef struct slist {
    struct slist *next;
    char *data;
} SLIST;

void pmemcpy(char *^to, char *from, int n);
```

```
/* load character strings into Dado, returning pointers to lists */
SLIST *^
load(void)
{
    char *malloc(), *gets();
    STRING line;
    SLIST *^slist = NULL;
    int ^nstrings = 0;

    while (gets(line)) {
        with (?(nstrings == !^min/nstrings)) {
            int len = strlen(line) + 1;

            par {
                SLIST *news;

                if (((news = (SLIST *)malloc(sizeof(SLIST))) == NULL)
                    || ((news->data = malloc(!^len)) == NULL)) {
                    eprintf("can't malloc %d bytes\n",
                            sizeof(SLIST) + !^len);
                    !^exit(1);
                }
                news->next = slist;
                slist = news;
                nstrings++;
            }
            pmemcpy(slist->data, line, len);
        }
    }
    return (slist);
}

/* after loading a set of character strings, */
/* test command-line strings against the set */
main(int argc, char *argv[])
{
    SLIST *^slist = !^load();
    int argn;

    for (argn = 1; argn < argc; argn++) {
        int total = 0;
        STRING ^teststring;

        pmemcpy(teststring, argv[argn], strlen(argv[argn]) + 1);
        par {
            register SLIST *p;

            for (p = slist; p; p = p->next) {
                seq total += +/(strcmp(p->data, teststring) == 0);
            }
        }
        printf("%4d %s\n", total, argv[argn]);
    }
}
```

References

[1] Flynn, M. J.
Very High Speed Computing Systems.
Proceedings of the IEEE 14:1901-1909, 1966.

[2] Iverson, K. E.
A Programming Language.
Wiley, New York, 1962.

[3] Kernighan, B. W., and Ritchie, D. M.
The C Programming Language.
Prentice-Hall, Englewood Cliffs, NJ, 1978.

[4] Kernighan, B. W., and Ritchie, D. M.
The C Programming Language.
Prentice-Hall, Englewood Cliffs, NJ, 1988.

[5] Lerner, M. D., Maguire Jr., G. Q. and Stolfo, S. J.
An Overview of the DADO Parallel Computer.
In *Proceedings NCC '85.* AFIPS, 1985.

[6] Mills, R. C., Radouch, Z., and Maguire Jr., G. Q.
A New Kernel for the DADO2 Parallel Computer.
Technical Report, Department of Computer Science, Columbia University, June, 1987.

[7] Rose, J. R., and Steele Jr., G. L.
C*: An Extended C Language for Data Parallel Programming.
In *Second International Conference on Supercomputing*, pages 2-16. International Supercomputing Institute, Inc., May, 1987.

[8] Snyder, L., and Socha, D.
Poker on the Cosmic Cube: The First Retargetable Parallel Programming Language and Environment.
In *Proceedings of the 1986 International Conference on Parallel Processing.* IEEE, 1986.

[9] Stolfo, S. J., and Miranker, D. P.
The DADO Production System Machine.
Journal of Parallel and Distributed Computing 3(2):269-296, 1986.

[10] Stolfo S. J., Miranker, D. P., and Lerner, M.
PPL/M: The Systems Level Language for Programming the DADO Machine.
Technical Report, Department of Computer Science, Columbia University, 1984.

[11] Stroustrup, B.
The C++ Programming Language.
Addison-Wesley, Reading, MA, 1986.

[12] van Biema, M., Maguire, G. Q. Jr., Lerner, M., and Stolfo, S. J.
The Design and Implmentation of a System Level Language for the DADO Parallel Machine.
In *Twentieth Hawaii International Conference on System Sciences*, pages 152-162. ACM, Kona, Hawaii, January, 1987.