# A Tutorial Introduction to llc

Russell C. Mills

Columbia University
Computer Science Department

17 May 1990

CUCS-433-89

## Abstract

Llc is an extension of C for hierarchically parallel processing on distributed-memory parallel processors. The language has been implemented for the Dado2 machine, a 1023-processor binary tree machine at Columbia University. This tutorial guides the novice through a series of working llc programs. It assumes no prior knowlege of the language, but makes no attempt to be complete.

Copyright © 1990 Russell C. Mills and The Trustees of Columbia University in the City of New York. All rights reserved.

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

This tutorial introduces the novice to llc, an extension of the C language for hierarchically parallel processing. It explains informally the syntax and semantics of the llc extensions to C. The tutorial is self contained and assumes no prior knowledge of llc, but it makes no attempt to be complete. For a more formal treatment of the language, the reader should consult [1].

Llc assumes a *data-parallel* model of computing with *synchronous semantics*. In llc, a single controlling processor, the *principal processor*, invokes operations in parallel in subsets of a set of attached processors, which can themselves invoke parallel operations in remaining processors. Each processor executing part of an llc program has associated with it a *retinue* of processors that receive instructions from it (their *director*), and an *evaluating retinue* of processors actively executing those instructions. If a retinue processor executes code that invokes parallel computation, that processor becomes the director of its own retinue, consisting of as many processors in its director's retinue as it can reach--but it cannot reach past a processor taking control of its own retinue.

The llc language has been implemented on the Dado machine, a binary tree of processing elements (PEs)[1] each consisting of a processor, local memory, and some communication circuitry. There is no shared memory. On Dado, a processor's retinue consists of its descendants in the tree. The root's retinue is the rest of the processors, while leaf processors have no retinue at all. In the Dado2 machine ( [4]), a complete binary tree of 8-bit processing elements functions as a coprocessor attached to a conventional host computer. [2] provides an overview of the Dado2 architecture and the implementation of llc on Dado2, while [3] discusses the meaning of various llc constructs in this implementation.

This tutorial is organized as a series of working llc programs. Each example introduces a few llc constructs and illustrates their use in conjunction with previously-introduced constructs. A complete translation into pidgin C and a brief informal discussion of the new constructs accompany each example. All these programs are available in the directory /proj/dado/llc/examples; output from these programs has been included directly in the text. Input from the user is in italics, while output from the programs is in typewriter font.

# 2 Getting Started

Let's write a small program that prints the words

```
hello, world
```

The llc program that prints this message is the same as the C program:

```
main()
{
    printf("hello, world\n");
}
```

This isn't really an interesting program; it just illustrates the fact that (almost) any valid C program is also a valid llc program. A more interesting program, *hello.llc*, causes each Dado PE to print the same message:

---

[1]This tutorial uses the terms *processor* and *processing element* interchangeably.

```
/* print a hello message from each Dado PE */
main()
{
    par gprintf("hello, world\n");
}
main()
{
    in all PEs {
        print "hello, world\n";
    }
}
```

The **par** statement causes each PE in the principal processor's retinue to call **gprintf**. Notice that there are no obligatory header files to **#include** in llc programs. Suppose the file *hello.llc* contains this program. To compile and link *hello.llc* for the Dado machine, one uses the *llcc* command, modeled on the the *cc* command:

*llcc -2 hello.llc*

This command converts the input file *hello.llc* into C code for principal (host) and retinue (Dado) processors, compiles the host and Dado C code into *hello.o* and *hello.o51*, and creates host and Dado executables *a.out* and *a.out.e51*. To run the program on Dado, one uses the *llcrun* command, which has no analog in the UNIX[2] operating system:

```
llcrun -2 -n 3 -p -l -D bigdado a.out
```

The option *-2* specifies that the program is to run on a Dado2 machine, *-n 3* specifies that the program is to use 3 Dado PEs instead of the whole machine, *-p* enables printing from Dado PEs, *-l* causes print messages to be labeled with PE numbers, and *-D bigdado* specifies which Dado machine to use (*/dev/bigdado* in this case). The program produces the output

```
[PE 1(0x1)] hello, world
[PE 2(0x2)] hello, world
[PE 3(0x3)] hello, world
```

which could have been redirected to a file, because llcrun sends the output generated by **gprintf()** to **stdout**.

## 3 Simple Communication

Recall that Dado2 functions as a coprocessor attached to a conventional computer, the *host*. Each PE has a unique ID; the host's ID is 0, while Dado2 PEs have non-zero ID's. The following program, *lineage.llc*, prints a Dado PE's ID, its parent's ID, its grandparent's ID, and so forth up to the Dado2 root, which has ID 1.

```
/* print a PE's lineage */
void
main(int argc, char *argv[])
{
    int pe = argc > 1 ? atoi(argv[argc - 1]) : 0;

    while (pe != self()) {
        printf("%d ", pe);
        with (self() == !^pe) pe = ^parent();
    }
    printf("\n");
}
```

The single line

[2]UNIX is a trademark of AT&T Bell Laboratories

```
with (self() == !^pe) pe = ^parent();
```

contains three llc constructs and two llc library functions. The **with** statement, with syntax

**with** *retinue-expression self-statement*

restricts the evaluating retinue for *self-statement* to those processors where *retinue-expression* is true. The **!^** (*sequential*) unary operator causes the evaluation of its operand in the retinue's director and the communication to the retinue of the resultant value. The **^** (*retinue*) unary operator causes the evaluation of its operand in its evaluating retinue, and the communication of one of the resultant values to the director. The **self()** function returns a PE's ID, while the **parent()** function returns the ID of a PE's parent. *Lineage.llc* can therefore be paraphrased as follows:

```
void
main(int argc, char *argv[])
{
    int pe = value (as an integer) of first command-line option;

    while pe is not the principal PE {
        print pe;
        select that PE {
            pe = parent of that PE;
        }
    }
}
```

Running *lineage* produces the output

```
llcrun -2 -D bigdado lineage 723
723 361 180 90 45 22 11 5 2 1
```

## 4 Reduction Operators

*Lineage.llc* has a serious flaw: What happens if the user requests the lineage of a PE not in the machine? In that case the evaluating retinue for the statement

```
pe = ^parent();
```

is empty, and the value assigned to **pe** is undefined. Since **pe** is the controlling variable for the **while** loop, the program can get stuck in an infinite loop, which is what it does on Dado2. The following program, *ancestry.llc*, corrects this problem:

```
/* print a PE's ancestors */
void
main(int argc, char *argv[])
{
    int pe = argc > 1 ? atoi(argv[argc - 1]) : 0;

    if ((||/1 :: (self() == !^pe)) || pe == self()) {
        for (;
             printf("%d ", pe), pe != self();
             pe = (^parent() :: self() == !^pe)) ;
        printf("\n");
    }
}
```

This example introduces two new llc operators. The **||/** (*reduction logical OR*) operator causes the evaluation of its operand in the evaluating retinue, and the communication to the retinue's director of the logical OR of those values. The **::** (*with*) operator is the expression analog of the **with** statement. The **::** operator binds very loosely, so that

```
self() == !^pe
```

modifies the evaluating retinue during the evaluation of

araphrased:

```
= value (as an integer) of last command-line option;

e is any PE with ID equal to pe (
 (;
  pe is not the principal PE;
  pe = parent of PE with ID pe)
rint pe;
```

nothing when given a PE not in Dado:

*D bigdado ancestry 15*

## eclarations

ng the Dado machine's topology from the bottom up, let's try showing it from the top program, *topology.llc*, prints a PE's ID and recursively displays all the PE's children. , starts in the principal PE.

```
dio.h>

E 0
 (!FALSE)
gned char bool;

 topology of Dado from a given PE down */

 , int depth)


 0; i < depth; i++) printf("  ");
 4d (0x%x)\n", id, id);
  parent() == !^id) {
 ^undone;

 ^(undone = TRUE); ||/undone;) {
ith (?undone) {
   print(^self(), depth + 1);
   par undone = FALSE;


 e principal PE */
  char *argv[])

 (), 0);
```

*topology.llc* have declared storage only in the principal PE. *Topology.llc* declares n the statement

o be a *retinue-tuple* of **bool**; each retinue PE has a single **bool** named **undone**.

The ^ operator is used in declarations the same way the C operators [] (*array of*), (), (*function returning*), and * (*pointer to* are. Since this is a default **auto** declaration, **undone** is created in each retinue PE when the director enters the block in which the declaration occurs, and destroyed when the director leaves the block. Notice that the number of objects created by the declaration depends on how big the retinue is. There is no way of knowing or specifying at compile time the number of objects in a retinue-tuple.

The code in *topology.llc* that sequentially processes a retinue-tuple is a characteristic llc idiom. The **?** (*select*) unary operator is useful for enumerating the elements of a retinue-tuple. The **?** operator returns a value of 1 in one PE in the evaluating set where its operand is nonzero, and returns 0 in all other PEs.

The outer **with** statement contains another llc construct, the **all** keyword. Normally only the PEs in the evaluating retinue evaluate the **with** condition, so each **with** or **::** shrinks the evaluating retinue, and evaluating retinues nest syntactically. The **all** keyword, when used in a **with** statement or a **::** expression, expands the evaluating retinue to the entire retinue before evaluating the **with** condition.

Paraphrasing *topology.llc* yields the following pidgin C code:

```
void
print(int id, int depth)
{
    indent 2 * depth spaces;
    print id;
    select all children of PE with ID id {
        sequentially select a child {
            recurse on that PE with depth + 1;
        }
    }
}

main(int argc, char *argv[])
{
    print(principal PE's ID, 0);
}
```

Running *topology* produces the output

*llcrun -2 -n 10 -D bigdado topology*

```
0 (0x0)
  1 (0x1)
    2 (0x2)
      4 (0x4)
        8 (0x8)
        9 (0x9)
      5 (0x5)
       10 (0xa)
    3 (0x3)
      6 (0x6)
      7 (0x7)
```

## 6 More Reduction Operators

A tree machine like Dado can sort in linear time if the cardinality of the set to be sorted is less than or equal to the number of processors in the machine. An llc sorting program is simple to write, and is contained in *sort.llc*.

```
/* load ints into Dado one per PE and sort them */
/* with or without duplicates */

#include <stdio.h>

typedef struct {
    int data;
    unsigned char valid;
} DATA;

DATA ^item;

/* look for uniqueflag (-u), then get data and sort */
void
main(int argc, char *argv[])
{
    char uniqueflag = 0;
    int argn;
    void getdata();
    void sort();

    for (argn = 1; argn < argc; argn++) {
        if (strcmp(argv[argn], "-u") == 0) uniqueflag = 1;
    }
    getdata();
    sort(uniqueflag);
}
```

```
/* get data from stdin and store in Dado, one datum per PE */
void
getdata()
{
    char *gets(), line[256];

    while (gets(line)) {
        if (||/!item.valid) {
            with (?!item.valid) {
                par {
                    item.data = !^atoi(line);
                    item.valid = 1;
                }
            }
        }
        else {
            fprintf(stderr, "%d is not enough PEs\n", +/1);
            exit(1);
        }
    }
}

/* sort the data, putting it on stdout */
void
sort(int flag)
{
    char ^uniqueflag = !^flag;
    char ^unprinted = item.valid;

    while (||/unprinted; ) {
        with (unprinted && (uniqueflag ?
                            (item.data == !^min/item.data) :
                            ?(item.data == !^min/item.data))) {
            printf("%d\n", min/item.data);
            par unprinted = 0;
        }
    }
}
```

This sorting program introduces two new reduction operators, **min/** (*reduction minimum*) and **+/** (*reduction add*). Each llc reduction operator evaluates its operand in the evaluating retinue and combines the values using a commutative, associative operator to yield a single value in the directing PE. The **sort** function in *sort.llc* contains another often-used llc idiom,

```
?(retinue-expression == !^min/retinue-expression)
```

which selects a single retinue PE having the minimum value of *retinue-expression*. This idiom is supported directly by the Dado2 I/O circuitry, and the llc compiler for Dado2 exploits this hardware support.

Notice that the **sort** function declares a retinue-tuple of **char** initialized to the value of **flag**. Using **flag** directly in the **with** condition would be illegal, because **flag** is storage in the directing processor, while the **with** condition is retinue code. The llc compiler does not use implicit **^** and **!^** operators to move values between a processor and its retinue. On the other hand, **typedef** names and **struct** and **union** definitions remain fully visible inside embedded retinue code.

Here is a paraphrase of *sort.llc*:

```
One item per PE; each one consists of an int and a valid flag;

void
main(int argc, char *argv[])
{
    look through command-line options looking for "-u";
    get data;
    sort;
}

void
getdata()
{
    char *gets(), line[256];

    while reading a line from stdin did not return EOF {
        if there is still an item without valid data {
            select an item without valid data {
                in that PE {
                    data = value (converted to int) of input line;
                    valid = true;
                }
            }
        }
        else {
            print error message telling how many PEs there are;
            host (and Dado) exit with error;
        }
    }
}

void
sort(int flag)
{
    uniqueflag is a char in each PE, initialized to flag;
    unprinted is a char in each PE, initialized to item.valid;

    while there are any unprinted data {
        select PEs with unprinted data and either
          all remaining instances of the minimum value or
          a single instance of the minimum value {
              print the minimum value on stdout;
              in those PE(s) {
                  mark the PE(s) as having printed data;
              }
          }
    }
}
```

Running *sort* a few times produces this output:

```
llcrun -2 -n 3 -D bigdado sort
-1
2
-1
-1
-1
2
llcrun -2 -n 3 -D bigdado sort -u
-1
2
-1
-1
2
llcrun -2 -n 3 -D bigdado sort -u
-1
2
-2
3
3 is not enough PEs
```

## 7 Hierarchically Parallel Processing

All the programs we have seen so far have used only a single level of parallelism. In fact, it's hard to write a simple program that effectively utilizes hierarchical parallelism. Here's a program, *level.llc*, that computes the level of each Dado PE. The program assigns level 1 to the children of the principal processor, and assigns level $n + 1$ to the children of a processor at level $n$.

```
/*
 * This program computes level numbers in Dado and then prints them
 * The root gets level 1, its children 2, its grandchildren 3, ...
 * This file need not be compiled with llcc -mm, since setlevel
 * is declared as a nonlocal function before it is used
 */

char ^level;

int self local();
int parent local();
void setlevel nonlocal();

int
main(int argc, char *argv[])
{
    with (parent() == !^self()) {
        par {
            level = 1;
            setlevel();
        }
    }
    par gprintf("self %x  level %x\n", self(), level);
}

#pragma retinue
void
setlevel()
{
    with (parent() == !^self()) {
        par {
            level = !^level + 1;
            setlevel();
        }
    }
}
```

Two features of this program merit comment. First is the **local** and **nonlocal** function declarations. Llc uses the keyword **local** to tell the compiler that a function contains no code that invokes operations in the retinue of the processor executing the function, and that the function calls no functions directly or indirectly that do. Likewise, llc uses the keyword **nonlocal** to warn the compiler that a function may contain or call such code. The llc compiler can generate much more efficient code if it knows which functions called from retinue code are local and which are nonlocal; **local** and **nonlocal** declarations typically override the default assumption that such functions are local. The syntax of a **local** or **nonlocal** declaration mirrors that of a **const** or **volatile** pointer declaration in ANSI C: the **local** and **nonlocal** keywords appear before the parentheses of a function declaration, but after the rest of the declaration.

Second is the **#pragma retinue** preprocessor directive. The llc compiler generates separate executable files for the principal PE and the rest of the Dado PEs. Unlike storage, a function in an llc program can be used anywhere, but it is up to the programmer to ensure that functions called in Dado PEs actually get compiled for them. The **#pragma retinue** directive instructs the compiler to generate code for the next function only for the retinue of the principal PE; the **#pragma self** directive, which is the default, tells the compiler to generate code for the principal PE. Including both directives directs the compiler to generate code for all PEs.

Here's a paraphrase of *level.llc*:

```
char level (in each PE);

int self() is a local function;
int parent() is a local function;
void setlevel() is a nonlocal function;

/*
 * with no #pragma, this function is needed only in the
 * principal PE
 */

int
main(int argc, char *argv[])
{
    select the PEs whose parent is the principal PE {
        in all these PEs {
            level = 1;
            setlevel();
        }
    }
    in all PEs {
        print ID and level;
    }
}

#pragma setlevel is a function needed only in Dado PEs
void
setlevel()
{
    select PEs whose parent is the PE executing this function {
        in all these PEs {
            level = level in parent + 1;
            recurse;
        }
    }
}
```

Running *level* in 7 PEs produces the following output:

```
llcrun -2 -n 7 -p -D bigdado level
self 1  level 1
self 2  level 2
self 4  level 3
self 5  level 3
self 3  level 2
self 6  level 3
self 7  level 3
```

## 8 Functions and More Declarations

Let's combine most of the constructs already discussed with some new declaration syntax. The following program, *count.llc*, distributes character strings to Dado PEs, one per PE, and counts how many times user-supplied test strings appear in the previously-loaded set of strings.

```
/* count the number of times a string appears in a set */
#include <stdio.h>

typedef char STRING[1024];

void pmemcpy(char *^to, char *from, int n);
int ^pstreq(char *^many, char *one);
char *^load();
void test(char *^);
char *promptgets();

main(int argc, char *argv[])
{
    test(!^load());
}

/* load strings into Dado, at most 1 per PE, returning pointers */
char *^
load()
{
    char *malloc();
    STRING line;
    char *^word = NULL;

    while (promptgets("load>", line)) {
        if (||/(word == NULL)) {
            with (?(word == NULL)) {
                int len = strlen(line) + 1;

                par {
                    if ((word = malloc(!^len)) == NULL) {
                        eprintf("can't malloc %d bytes\n", !^len);
                        !^exit(1);
                    }
                }
                pmemcpy(word, line, len);
            }
        }
        else {
            eprintf("%d is not enough PEs\n", +/1);
            exit(1);
        }
    }
    rewind(stdin);
    return (word);
}

/* compare test lines with strings loaded into Dado PEs */
void
test(char *^word)
{
    STRING line;

    while (promptgets("test>", line)) {
        printf("number of occurrences %d\n", +/!^pstreq(word, line));
    }
}
```

```
/* prompt and get a line from stdin */
char *
promptgets(char *prompt, char *line)
{
    char *s;

    printf("%s", prompt);
    if ((s = gets(line)) == NULL) printf("\n");
    return (s);
}

/* compare a string in the directing PE and strings in the retinue */
int ^
pstreq(register char *^many, register char *one)
{
    register int c;

    while (c = *one++) {
        par {
            if (many && *many++ != !^c) many = NULL;
        }
    }
    return (many && *many == 0);
}
```

In llc, functions can take retinue-tuples as arguments and can return retinue-tuple values. Such functions must have the types of their parameters and return values declared before use. If a function takes a retinue-tuple as a parameter, the corresponding argument expression is retinue code, and is treated just as if it were enclosed in a **par** statement. The value returned from a function returning a retinue-tuple can be used only in retinue code, but the call itself is not retinue code: the retinue's director executes the call, so the call must be set off with the !^ operator.

One function that takes a retinue-tuple argument is parallel memory copy, **pmemcpy**. This function copies memory from a PE to its retinue, but can be safely used only for arrays of **char**, since the Dado2 principal PE's data formats are different from those of the retinue PEs.

Notice the use of the function **eprintf()** in *count.llc*, which is a synonym for **fprintf(stderr, ...)**. Since **stderr** is the address of storage in the principal PE, Dado PEs cannot refer to it, but they can call **eprintf**. On the other hand, a program can declare a retinue-tuple of FILEs, but cannot do much that is useful with them, since most of the functions in the standard I/O library exist only for the principal PE.

Finally, look at the call to **exit()** in the function **load()**; it is embedded in retinue code. When is **exit()** called? In llc, the directing processor evaluates the operand of the !^ operator only if the evaluating retinue is nonempty, and if it does evaluate the operand, it does so only once. So the principal processor exits if its current evaluating retinue is not empty, that is, if some PE has failed to allocate enough space.

Here's *count.llc* rephrased:

```
pmemcpy() takes as copy destination a pointer to char in each PE;
pstreq() has as first parameter a pointer to char in each PE;
load() returns a pointer to char in each PE;
test() has as its parameter a pointer to char in each PE;
promptgets() returns a pointer to char;

void
main(int argc, char *argv[])
{
    test() on the value returned by load() in each PE;
}

char *^
load()
{
    line is an array of characters;
    word is a pointer to char in each PE, initially NULL;

    while there is an input line from stdin {
        if there is any PE with word still NULL {
            select one PE with word still NULL {
                int len = length of line + 1 for the NULL at the end;

                in this PE {
                    if we cannot allocate len bytes {
                        print an error message to stderr;
                        the principal PE exits (and so do all PEs);
                    }
                }
                memcpy from principal PE to this PE;
            }
        }
        else each PE has a string,
        but the principal PE still has more to distribute {
            print the number of Dado PEs to stderr;
            exit;
        }
    }
    return the string in each PE;
}

void
test(char *^word)
{
    line is an array of char in the principal PE;

    while there is an input line from stdin {
        print the sum over all PEs of
          the match of the local string
          and the string in the principal PE
        }
}
```

```
char *
promptgets(char *prompt, char *line)
{
    prompt;
    read a line;
}

int ^
pstreq(register char *^many, register char *one)
{
    register int c is a single character of the principal PE's string;

    for each character in the string {
        in all PEs {
            if it matches up to now, but this character mismatches {
                this PE no longer matches;
            }
        }
    }
    return 1 if the PE still matches and is at the end of its string;
}
```

Running *count* on some simple data produces the output:

```
llcrun -2 -D bigdado count
load>first
load>second
load>third
load>first
load>
test>first
number of occurrences 2
test>second
number of occurrences 1
test>third
number of occurrences 1
test>fourth
number of occurrences 0
test>
llcrun -2 -n 3 -D bigdado count
load>first
load>second
load>third
load>fourth
3 is not enough PEs
```

## 9 Overcoming the one-datum-per-PE restriction

An inconvenient feature of llc is the restriction that retinue-tuples contain exactly one element in each retinue PE. Because of this restriction, llc programs are portable between Dado machines with different numbers of PEs, but llc programs cannot declare distributed arrays--it is impossible to distribute the elements of an array of known size among an unknown number of Dado PEs. Llc lets the programmer declare retinue-tuples of arrays, but these do not solve the problem. One portable solution to the problem is to allocate most retinue storage dynamically, and to keep data in linked lists in each PE. The following program, *lists.llc*, is *count.llc* rewritten to use linked lists.

```
/* count the number of times a string appears in a set */

#include <stdio.h>

typedef char STRING[1024];
typedef struct slist {
    struct slist *next;
    char *data;
} SLIST;

void pmemcpy(char *^to, char *from, int n);
int ^pstreq(char *^many, char *one);
SLIST *^load();
void test(SLIST *^);
char *promptgets();

main(int argc, char *argv[])
{
    test(!^load());
}

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

    while (promptgets("load>", 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);
        }
    }
    rewind(stdin);
    return (slist);
}
```

```
/* compare test lines with strings loaded into Dado PEs */
void
test(SLIST *^slist)
{
    STRING line;

    while (promptgets("test>", line)) {
        int total = 0;

        par {
            register SLIST *runner;

            for (runner = slist; runner; runner = runner->next) {
                seq total += +/!^pstreq(runner->data, line);
            }
        }
        printf("number of occurrences %d\n", total);
    }
}

/* promptgets() and pstreq() are the same as in count.llc */
```

*lists.llc* introduces only one syntactic feature of llc, the **seq** statement, which is the syntactic analog of the !^ operator. The **seq** statement in **test** is embedded in a **par** statement containing a **for** loop. What are the semantics of a loop in retinue code? At each execution of the continuation condition for the loop, processors where the condition is false drop out of the evaluating retinue. When the evaluating retinue is empty, the loop terminates, and the pre-loop evaluating retinue is restored.

The following paraphrases the code in *lists.llc*:

```
SLIST is the building block for linked lists of pointers to char;

pmemcpy() takes as copy destination a pointer to char in each PE;
pstreq() has as first parameter a pointer to char in each PE;
load() returns a pointer to SLIST in each PE;
test() has as its parameter a pointer to SLIST in each PE;
promptgets() returns a pointer to char;

void
main(int argc, char *argv[])
{
    test() on the value returned by load() in each PE;
}

SLIST *^
load()
{
    line is an array of characters;
    slist is the head of a list of pointers to char, initially NULL;

    while there is an input line from stdin {
        with one PE with the fewest strings so far {
            int len = length of line + 1 for the NULL at the end;

            in this PE {
                news is a pointer to an SLIST;

                if we cannot allocate space for an SLIST or
                  for a copy of line {
                      print an error message to stderr;
                      the principal PE exits (and so do all PEs);
                  }
                link news at the head of slist;
                increment the number of strings in this PE;
            }
            memcpy from the principal PE to this PE;
        }
    }
    return the head of the list in each PE;
}
void
test(SLIST *^slist)
{
    line is an array of char in the principal PE;

    while there is an input line from stdin {
        total is the number of occurrences of this line in all PEs;

        in all PEs {
            runner traverses the linked list of pointers to char;

            traverse the list {
                in the principal PE add the sum over all PEs of
                  the match of the current local string
                    and the string in the principal PE;
            }
        }
        printf("number of occurrences %d\n", total);
    }
}
```

Notice the code in the line

```
        with (?(nstrings == !^min/nstrings)) {
```

This code selects a single PE with the minimum number of stored strings in which to store the next string. Thus it balances the number of stored strings and the computational load among the available PEs.

Running *lists* on some sample data produces the following output:

```
llcrun -2 -n 3 -D bigdado lists
load>first
load>second
load>third
load>first
load>
test>first
number of occurrences 2
test>second
number of occurrences 1
test>last
number of occurrences 0
test>
```

## 10 Conclusion

This tutorial began with the simplest possible llc program, and has moved rather quickly to a program that uses most of the features of llc, and even does some runtime load balancing. The reader who understands these examples is ready to write ambitious programs on the Dado machine.

# References

[1] Mills, R. C.
*Llc Reference Manual.*
Technical Report, Department of Computer Science, Columbia University, 1989.

[2] Mills, R. C.
*The llc Parallel Language and its Implementation on DADO2.*
Technical Report, Department of Computer Science, Columbia University, 1989.

[3] Mills, R. C.
*Dado2 llc Users' Manual.*
Technical Report, Department of Computer Science, Columbia University, 1989.

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