A Critique of the llc Parallel Language and Some Solutions

Russell C. Mills

Columbia University
Computer Science Department

17 May 1990

CUCS-430-89

Abstract

Llc is an extension of C that has been implemented on the Dado2 machine at Columbia university. 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. Llc allocates one element of a parallel object per physical processor. Removing this restriction allows programs to use parallel vectors of arbitrary size without reference to the number of processors in the machine. A program in the resulting language, mpc, contains a single main process. Each mpc process can create sets of attached processes statically or dynamically by declaring arrays of process type, and can invoke operations in parallel in these processes. Mpc retains much of llc's power while adding generality, clarity, and portability.

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

The llc language [1] is an extension of C that has been implemented on the Dado2 parallel computer at Columbia university. 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 paper points out some of the shortcomings of llc, both as a system-level and as a more general-purpose parallel programming language, and proposes a new language, mpc, that addresses these problems, while retaining most of the features and power of llc.

Llc allocates one element of a parallel object per physical processor. Removing this restriction allows programs to use parallel vectors of arbitrary size without reference to the number of processors in the machine. A program in the resulting language, mpc, contains a single main process. Each mpc process can create sets of attached processes statically or dynamically by declaring arrays of process type, and can invoke operations in parallel in these processes. The new language adds to the capabilities of llc the explicit creation of parallel objects of various sizes and shapes, and hierarchical parallelism created by declarations rather than the runtime environment.

2 Critique of llc

As a system-level parallel language, llc has many strong points: it has simple, powerful constructs, provides deterministic execution, raises no artificial barriers between sequential and parallel code, does not require the programmer to write message-passing code, and provides full and efficient use of the Dado machine. Nevertheless, llc has a number of problems.

One set of problems stems from the requirement that each processor contain a single element of each parallel object (*retinue-tuple* in llc terminology). with the conseqeuent impossibility of specifying the cardinality of a retinue-tuple. Thus a program text does not express the complete meaning of the program, and it is difficult to reason about an llc program. Another consequence is that oversized problems--those for which the target machine does not have a processor for each data element in the problem--do not fit well into llc. A programmer can create retinue-tuples of arrays, but must iterate over the elements of the arrays explicitly. Thus iteration over the data elements of an oversized problem must contain two levels of iteration, an outer level iterating over the processors in the machine, and an in one iterating over the data in a processor. Similarly, parallel operations on these data elements must contain a level of iteration and a level of parallel execution. Furthermore, it is impossible to dimension arrays properly in each processor without knowing how many processors the target machine contains. Worse, while it is not difficult to calculate the number of processors in a given processor's *retinue* (set of attached descendant processors) -- +/1::all does it -- +/1 is not a constant expression, and so cannot be used in array bounds. Thus a program must use dynamic storage allocation, or the programmer, not just the compiler, must know how many processors are in the target machine.

A similar set of problems stems from the dynamic calculation of a processor's retinue. The number of processors controlling subsidiary parallel operations is determined by the number of processors in the parallel machine, the partial ordering on the set of processors, and each processor's data. Again, a program's text does not express the complete meaning of the program. A programmer wanting to have, say, 16 computations in parallel, each controlling its own parallel computation, must explicitly select which 16 processors should execute those computations. On Dado2, a programmer typically uses the library function self() to select a level of the tree.

While the single-element-per-processor restriction and the data-dependency of hierarchical parallelism create problems for the llc programmer, they do provide an advantage in the form of potential efficiency at the expense of program elegance and portability. Llc gives the programmer complete control of the placement of data and independently parallel tasks. A programmer can distribute subtasks in order to maximize effective parallelism, and for oversized problems, can assign data to selected processors in order to balance the processing load.

A further problem arises because retinue-tuples are not arrays, so the language does not allow a program to iterate over the elements of a retinue-tuple with array subscripts or pointers. It is in fact possible to iterate over the elements of a retinue-tuple, but the code required to do so is clumsy:

```
{
        int ~not_done = TRUE;

        while (||/not_done) {
                with (?not_done) {
                        user code;
                        not_done = FALSE;
                }
        }
}
```

Llc suffers from a number of other problems as well:

- Although llc is suitable for hierarchically parallelizable computations, the language does not express any notion of hierarchy in its data declarations. Thus there are no retinue-tuples of retinue-tuples.

- One can legally communicate a pointer using !^ and ,/, and llc allows the communicated pointer to be dereferenced, but in general, dereferencing a communicated pointer is meaningless. If p is a pointer, the types of p and !^p should be different, and !^p should point to the processor where *p resides, as well as the address within that processor. The problem of communicating pointers is related to the difficulties in iterating through the elements of a retinue-tuple.

- Llc makes no provision for shared memory. In fact, the design of the language forbids it, since pointers cannot point to objects in other processors. While shared memory cannot be implemented on Dado, a richer language might offer a shared-memory construct consistent in spirit with the language subset implementable on Dado.

- Since the llc compiler cannot know how a program decomposes hierarchically, it cannot create different executable images for different processors. All Dado processors must receive all distributed code. Since memory at each PE is limited, and since Dado has no virtual memory, it is wasteful and limiting not to produce different executable images for processors that execute different pieces of code.

3 Mpc

This section proposes a new language, named mpc (*m*ultiply *p*arallel C), which retains most of the features of llc, and which solves the problems described in the previous section. In mpc, objects to be operated on in parallel are not tied to specific processors. By removing the one-element-per-processor restriction, mpc allows programs to use parallel vectors of arbitrary size without reference to the number of processors in the machine. An mpc program contains a single principal process, created when the program begins executing. Each mpc process can create sets of attached processes statically or dynamically, and can invoke operations in parallel in these processes. A set of processes is created simply by a declaration of an array of objects of process type, and has the same lifetime as an ordinary object: for a local declaration, the block in which it is declared, and, for a static or global declaration, the duration of the program. Since process sets are created by declarations rather than the interaction of data with the execution environment, mpc programs are clearer and more portable than llc programs.

The design of mpc owes much to C* for the Connection Machine [2], although mpc is not an extension of C*. Mpc retains much of the syntax and flavor of llc. There follows below a brief description of mpc syntax and semantics where mpc and llc differ. For an example of mpc code, see Appendix I.

3.1 Processes

An mpc program consists of a number of *processes*. Each process is created by a declaration of a process object. A process type is akin to a C++ class, and is declared much like a class, except that separate statements can declare different components of a process type. A process declaration with a process *tag* (like a **struct** or **union** tag in C) declares the name of the process type; the process tag becomes a typedef name, and can be used in further declarations. Later declarations of pieces of the process type use only the process tag. Names of process types obey the usual C scoping rules, so it is possible to redeclare a process tag in an inner block, in which case it becomes the name of a new process type.

As in C++, process types have member functions, which are declared with C++ syntax. Functions belonging to no process type implicitly belong to the (unnamed) type of the principal process, of which there is but a single instance, namely the principal process.

A declaration of part of a process type can also declare objects of process type; the syntax is identical to that used for declaring objects of **struct** or **union** types. A declaration of a process or of an array of processes creates a number of processes of that type. A set of processes created by a single declaration is called a *process set*, and is the domain of parallelism in mpc.

An llc program with only a single level of parallel operations corresponds roughly to an mpc program with a single named process type and a single array of processes, one for each processor in the principal processor's retinue.

3.2 Parallel Code

Parallel operations over a process set in mpc are invoked with the the **par** statement together with the name of a process set. Thus if *pset* is the name of a process set,

par (*pset*) *parallel-statement*

causes the parallel execution of *parallel-statement* in all processes in *pset*.

Mpc allows a programmer to make local declarations of parallel objects over process *sets*, while it requires global declarations of parallel objects to be over process *types*. Why the distinction? Because a local declaration of a parallel object should be equivalent to a local declaration of a sequential object within a **par** statement, which act over process sets. Furthermore, a local declaration of a parallel object effectively invokes a parallel operation, namely the allocation of stack space for the object, while a global declaration of a parallel object causes the compiler to set aside space for that object in the data space of each instance of the process.

Mpc provides llc's set of reduction operators, but the mpc program must specify the domain of application of the operator:

reduction-operator process-set-name . parallel-expression

3.3 Parallel Arguments and Return Values

Mpc functions can have parallel arguments and return parallel values, as can llc functions. The declaration syntax required for parallel arguments and return values is more complicated in mpc than in llc, because a function may be called from different places with different process sets. Here is an example:

```
int (p pset[]).f(int);  pset is an array of
                            process type p
                        f is a
                            function of
                                int
                            returning
                                pset of
                                    int
```

3.4 Addressability

When one process refers to the value of a pointer in another process, the resulting pointer has as part of its value the location of the second process, as well as the address of the object within that process. In analogy with C++, which gives a pointer to a class member a type containing the member's class, a communicated pointer has as part of its type the process type pointed to. This convention makes objects within a process set addressable from the parent process, thus making it possible to iterate easily through the elements of a parallel object. As a special case, addressable processes make shared memory possible, provided the communication network of the machine is sufficiently highly connected.

3.5 An Implementation Scheme for Dado

Not all of mpc can be implemented efficiently on Dado, since on a tree-structured machine, general pointers between processors cannot be handled well. However, with some restrictions on inter-process pointers, which would still allow a process to iterate through process sets it creates, it should be possible to implement most of mpc.

The translation scheme envisioned for mpc on Dado does not assume an infinite supply of virtual processors supported by hardware. Instead, it maps a number of processes to each physical processor and converts parallel code into loops over the processes assigned to a processor. Provided that the number of processes assigned to each processor is fairly large, this should provide a measure of load balancing. Furthermore, the mpc compiler will be able to create different executable programs for different processors, to the extent that the pattern of process creation is statically determined. This will save precious memory in a massively parallel MIMD machine.

4 Conclusion

This paper points out some of the shortcomings of llc, both as a system-level and as a more general-purpose parallel programming language, and proposes a new language, mpc, that addresses these problems, while retaining most of the features and power of llc. The new language adds to the capabilities of llc

- the explicit creation of parallel objects of various sizes and shapes,
- hierarchical parallelism created by declarations rather than the runtime environment,
- the capability to iterate through elements of parallel objects, even on a limited machine like Dado,
- and general interprocess pointers on a parallel machine with a more complete interconnection network.

I. An Example in llc and mpc

This section presents a simple but illustrative example coded in llc and in mpc. Each program computes the minimum of a set of maxima of a set of integers. First is the llc code.

```
static int ^i;

int main(int argc, char *argv[])
{
    initialize();
    with (id() >= 16 && id() < 32) {
        printf("min max %d\n", min/maximum());
    }
    return (0);
}

int (^maximum) ()
{
    return (max/i);
}
```

In contrast, the mpc code is more transparent:

```
process q {
    int i;
};

process p {
    int maximum();
    q qset[16];
} pset[16];

main(int argc, char *argv[])
{
    initialize();
    printf("min max %d\n", min/pset.(maximum()));
}

int p::maximum()
{
    return (max/qset.(i));
}
```

References

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

[2] 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.