

Indexical Imperative Languages for Data-Parallel Programming

Weichang Du

Department of Mathematics and Computer Science

University of New Brunswick, Saint John, N.B. Canada E2L 4L5

wdu@unb.ca

Abstract

Indexical programming languages have three essential characteristics: *intensionality*, *dimensionality*, and *context-switching* operations, which distinguish them from other programming language styles. In this paper, we first argue that, in certain restricted forms, existing data-parallel languages (DPLs) also have the characteristics of indexical languages, and hence we can give indexical interpretations to constructs of DPLs. We then show that indexical imperative languages (IILs) with synchronous semantics can be considered as *high-level* DPLs. By high level, we mean that IILs have both architecture(SIMD)-independent formal semantics when used for application programming, and SIMD-based operational semantics when used for data-parallel programming. In particular, in terms of SIMD computations, we show that indexical constructs in IILs have at least the same expressivity as parallel constructs of existing architecture- or machine-specific DPLs such as C* and MPL.

1 Introduction

Most existing DPLs are extensions of sequential languages, such as C* [Fra91], HP Fortran [Hig92], and MPL [Mas92]. The meanings of data-parallel programs in these languages are defined based on the SIMD operational semantics. Thus, to be able to write programs in DPLs, programmers have to know in a certain level of detail about how computations are carried out on the underlying SIMD architecture. In some case, programmers even need to know how the underlying machine works when programming in a machine-specific language such as MPL.

These architecture- or machine-dependent DPLs cause two problems for writing parallel programs. First, they do not support parallel programming at the application level. That is, when

writing an application program, the programmer is concerned with using parallelism to improve the program's performance on parallel computers, but does not need to know how the program runs on a particular parallel architecture such as SIMD. This can not be done in the current parallel programming practice, including data-parallel programming, because the semantics of DPLs are bound with SIMD operational semantics, or even with some particular machine features. Secondly, program portability is another problem. Parallel programs written for one parallel machine in a language with specially designed parallel constructs cannot run on another machine, even if parallel constructs of the latter's language have similar behaviors. It is not only the problem about porting programs between different architecture-types such as between SIMD and MIMD, but also between different machines with the same architecture type such as MasPar and the Connection Machine.

We believe that what we need to solve the problems are *high level* parallel languages (HLPLs) that provide a conceptual level of parallelism independent of their operational semantics. All constructs in HLPLs should have formal semantics which are not based on any parallel operational model. In application programming, the programmer can write parallel programs in HLPLs without considering the parallel operational semantics. But because of the parallelism inherited in HLPLs, parallelism can be expressed in the program implicitly. This is the idea of implicitly parallel programming. On the other hand, HLPLs should have the expressive power, in terms of parallel operational semantics of their constructs, to express the parallel operations of existing and future architecture- or machine-specific parallel languages. In other words, to guarantee that HLPL programs potentially have at least the same performance when implemented on a particular parallel machine as the parallel language specially designed for the machine, there must exist a mapping, in terms of operational semantics, from the parallel constructs of the specific one to that of HLPLs. Otherwise, it is impossible for HLPLs to compete performance with the machine-specific languages. This is the idea of explicitly parallel programming.

Indexical programming languages have three essential characteristics, that is, *intensionality*, *dimensionality*, and *context-switching* operations. In an indexical program, the **intensionality** means that the meanings of program constructs implicitly depend on contexts on a pre-defined or user-defined context space. It is realized by the context space definition and the indexical semantics based on the defined context space. The **dimensionality** means the variation patterns in which the meanings of program constructs vary on the context space. It is realized by the dimensionality declarations and the derivation rules for dimensionality combinations. The **context-switching** operations are those operations that switch contexts from one to another to specify the relationships between the meanings of program constructs in different contexts. They are realized by the definitions of the indexing scheme for the context space and the primitive context-switching operators

based on the scheme.

We consider that indexical languages as candidates for high-level parallel programming, because they not only have parallelism inherited from the formal indexical semantics, that is, **context parallelism**, but also have indexical constructs that can be given a parallel operational semantics and can be mapped to parallel constructs of existing parallel languages.

The indexical programming paradigm is independent of language styles. It can be considered as a model for enriching conventional programming languages with indexical semantics, such as those indexical functional and logic languages which have been designed and developed based on the indexical paradigm [WA85] [Pla92] [OW89].

In Section 2, we describe the idea of indexical imperative languages as the semantic enrichment of existing imperative languages, such as C, with indexical semantics. In Section 3, we show the mapping from existing data-parallel languages to indexical languages. We analyze the semantics of some existing imperative DPLs, namely C* and MPL. We give indexical interpretations to their parallel constructs, and show that, based on the interpretations, the DPLs have the characteristics of indexical languages in some restricted forms. In Section 4, we show the mapping from indexical imperative languages to DPLs. We give SIMD-based operational semantics to indexical constructs, and define the indexical operations that have the same operational semantics as the parallel constructs of the DPLs.

2 The Idea of Indexical Imperative Languages

To enrich imperative languages with indexical semantics, we can use a similar idea to the way that we enriched functional and logic languages. That is, we give indexical semantics to basic constructs in imperative languages. By doing so, programs built up by the basic constructs also have indexical semantics.

Given a base *context space* C which can be pre-defined or user-defined, the environment of an indexical imperative program is a *state space* consisting of an indexed set of states, each of which is a local environment associated with a context on the context space. Formally, we define the state space as

$$SS = C \rightarrow S$$

$$S = (D_l \longrightarrow D_r)$$

where D_l is the domain of locations (l-values) and D_r is the domain of data values (r-values). In terms of computation, the state space represents a kind of multi-thread computation, each thread

is represented by the state change in a local environment in a context. If the base context space has cardinality 1, this is just the sequential computing model.

In imperative languages, we consider the basic constructs to be expressions and statements. In an indexical imperative program, an expression has a value in *every* context, which is computed based on the local environment associated with that context. The semantics of an expression in an indexical imperative language is

$$C \longrightarrow D_l$$

or

$$C \longrightarrow D_r$$

An expression may have different values in different contexts.

A statement in an indexical imperative language has the semantics

$$SS \longrightarrow SS$$

We can define the denotational semantics of a statement in two levels. At the first level, we define the global impact of a statement (or command) in a *particular* context, that is, given a context c how the state space changes after a statement execution in the context c . Notice a statement execution in a context may refer to and/or change state(s) in other contexts because of context-switching operations. In general, we can define the first level semantic function for a statement as

$$\llbracket st \rrbracket (ss, c) = ss'$$

where ss' is a new state space created based on the given state space ss and the impact of the statement st in context c only. At the second level, we define the global impact of a statement execution in *every* context as the union of the new state space created by the statement execution in each context. In general, we can define the second level semantic function for a statement as

$$\llbracket st \rrbracket ss = \bigcup_{c \in C} \llbracket st \rrbracket (ss, c)$$

where the union of two new state spaces ss_1 and ss_2 with respect to the original state space ss is defined by

$$ss_1 \cup ss_2 = \lambda c \in C \lambda l \in D_l \begin{cases} ss_1(c)(l) & ss_1(c)(l) = ss_2(c)(l) \\ ss_1(c)(l) & ss_2(c)(l) = ss(c)(l) \\ ss_2(c)(l) & ss_1(c)(l) = ss(c)(l) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Informally, given a context c and a location l in context c , the union of two new state spaces always takes the one that has updated value in terms of the original value in the location. The *undefined*

value is caused by the update conflict between two statement executions in two different contexts which update the same location in a context. We may also define this update conflict in a different way than giving the undefined value, for instance, giving ss_1 a higher priority to override whatever changed in ss_2 .

For the semantics of an expression, because of possible side effect caused by the expression evaluation, we can define its semantic function to produce a pair of *value* and *state space*, where *value* is an intension from the context space to the domain of l-values or r-values. Notice that the l-value of an expression in a context is a pair consisting of the l-value in that context and the context itself, because when we refer to a location we also need to tell in what context the location is located.

For example, in a context c , an assignment statement $x = x + 1$ will update the content of location x in context c , and a procedure call $proc(x, y)$ will be executed in the local environment associated with context c .

In terms of computation, in a program, a statement may *not* be executed in the computation thread corresponding to every context. There are two kinds of conditions that make a statement be executed in a context or not. A *data-dependent* conditional only depends on the local environment in a context and is independent of the context itself, which we also call a *pointwise conditional* because it is a pointwise extension of the sequential conditional statement. For example, below is a data-dependent conditional

```
if (a > 0) { x = x + 1 };
```

In a context c , the assignment statement $x = x + 1$ is executed only when the value of variable a in context c at the current state is positive. A *context-dependent* conditional only depends on a context itself and is independent of local environment in the context, which we also call a *synchronic conditional*. For example, below is a context-dependent conditional, where in a context c $index(this_context)$ gives the index of c provided the underlying context space is indexed by integers

```
if (index(this_context) > 0) { x = x + 1 };
```

The assignment statement $x = x + 1$ is executed only when the index of context c is positive. The two conditionals are orthogonal to each other, but they can be combined to specify if a statement will be executed at a computing point in a context. For example,

```
if (index(this_context) > 0 && a > 0) { x = x + 1 };
```

From the global point of view, in an indexical imperative program, a statement nested in pointwise and/or synchronic conditionals may be executed in a *subset* of contexts in which the conditions are satisfied.

Formally, we can define the semantics of a conditional statement as follows

$$\llbracket \text{'if' } exp \text{ st} \rrbracket (ss, c) = \begin{cases} \llbracket st \rrbracket (ss, c) & \text{if value}(\llbracket exp \rrbracket ss)(c) \\ ss & \text{otherwise} \end{cases}$$

An indexical imperative program without context-switching operations is trivial. It is equivalent to independent executions of the same sequential program with possibly different sets of inputs in different contexts. For example, assume the context space of the following program consists of 5 contexts, which are indexed by integers from 1 through 5,

```
main() {
    int n,
    scanf("%d", &n);
    printf("%d\n", n*n);
}
```

When $n = 2, 4, 6, 8, 10$ in contexts 1 through 5, then output of the program will be 4,16,36,64,100. In a better case, when synchronic conditionals are involved, the program may produce different results, even if the input is constant in every context. For example,

```
main() {
    int n = index(this_context);
    printf("%d\n", n*n);
}
```

will output values 1,4,9,16,25. But this variety is static, meaning that we can transform the program into equivalent sequential programs, each of which corresponds to a context, and run them independently. In short, it is context-switching operators, or indexical operators, that give the real expressive power to indexical imperative languages. Context-switching operators specify communications between computation threads in different contexts.

Let cs be a context-switching operator which switches from context a to context b , let exp be an expression, and let st be a statement. Consider the meanings of context-switching operations.

1. $cs(st)$ means that the statement st is executed at context b , which may change the local environment of context b but not a , if there is not further context-switching operator involved in st . Formally, its semantics can be defined as

$$\llbracket cs(st) \rrbracket (ss, c) = \llbracket st \rrbracket (ss, I_{cs}(c))$$

where $I_{cs}(c)$ is the context that cs switches to from c ,

2. $cs(exp)$ in context a has the value of exp in context b , i.e. exp is evaluated in the local environment associated with context b . Formally, its semantics can be defined as

$$\llbracket cs(exp) \rrbracket ss = (\lambda c \in C \text{ value}(\llbracket exp \rrbracket ss)(I_{cs}(c)), \bigcup_{c \in C} \text{state}(\llbracket exp \rrbracket (ss, I_{cs}(c))))$$

For example, consider the differences between the following assignment statements with context-switching operator cs applied to.

1. $x = cs(y);$
 2. $cs(x) = y;$
 3. $cs(x = y);$
1. The variable x in context a is assigned the value of variable y in the current local environment in context b . So, the local state in context a changes, but not the state in b .
 2. The variable x in context b is assigned the value of variable y in the current local environment in context a . So, the local state in context b changes, but not the state in a .
 3. The statement is executed in context b , where both x and y are taken from in the local environment in context b , nothing affects the state in context a except that the assignment is initiated in context a .

In general, we can define the semantics of an assignment statement as follows.

$$\llbracket exp_l \text{ ' = ' } exp_r \rrbracket (ss, c) = [\text{value}(\llbracket exp_r \rrbracket ss)(c) / \text{location}(\text{lvalue})] ss(\text{context}(\text{lvalue}))$$

where

$$\text{lvalue} = \text{value}(\llbracket exp_l \rrbracket ss)(c)$$

$$[a/x]ss(c) = \lambda c' \in C \lambda l \in D_l \text{ if } (c = c') \wedge x = l \text{ then } a \text{ else } ss(c')(l)$$

The most important semantic issue of indexical imperative languages that distinguishes them from indexical declarative languages is how the program's state space is referred and updated during the computation. State transition in sequential languages takes a well-defined, unified form because there is only one computation thread. However, because of communications among multiple computation threads in indexical imperative languages, when we switch from context a to context b , we have to define the semantics that at what point in the state history of the local environment in context b , the expression is evaluated or the statement is executed. This is a common semantic problem for imperative languages which allow multiple computation threads.

In this paper, we define indexical imperative languages using synchronous semantics. That is, the state space, i.e. the global environment consisting of all local environments, of a statement execution is the one immediately after the last statement execution. Formally, we can define this synchronous semantics as follows

$$\llbracket st_1; st_2 \rrbracket (ss, c) = \llbracket st_2 \rrbracket (\llbracket st_1 \rrbracket ss, c)$$

This semantic definition can be easily understood for programs that do not have conditionals, because in this case computation threads in contexts contain the same sequence of operations. However, for the conditional statements, to guarantee the correctness of the semantics, we need to serialize their components. That is, we define that the components must be executed in order. Formally, we can define the semantics of conditionals with alterative components as follows.

$$\llbracket 'if' exp st_1 st_2 \rrbracket (ss, c) = \begin{cases} \llbracket st_1 \rrbracket (ss, c) & \text{if value}(\llbracket exp \rrbracket ss)(c) \\ \llbracket st_2 \rrbracket (ss', c) & \text{otherwise} \end{cases}$$

where

$$ss' = \llbracket 'if' exp st_1 \rrbracket ss$$

For example, for *if A then B else C*, we first evaluate *A*, then execute *B*, and then execute *C* at each context. If in a context, expression *A* has value false, the execution of *B* in the context does not have effect on the state space. Similarly, if in a context, *A* has true value, the execution of *C* in the context does not have effect on the state space.

In the above discussions, we have generally referred to a context space *C* that the semantics of an indexical program is based on. The context space of an indexical program can be built-in in the underlying indexical language, such as the one dimensional integer context space of Lucid, or can be user-defined, such as in Indexical Lucid [FJ91], ML-Lucid [Pla92], and uLucid [Du93]. The user-defined context space in an indexical program can be defined statically, that is, it is fixed at compile time, such as those in ML-Lucid and uLucid. A context space can also be defined dynamically, that is, it may vary during the program computation. Dynamic context spaces can be defined in two ways, by local dimension declarations such as in Indexical Lucid, or by dynamic allocations which can be done in indexical imperative languages.

Dimensionality can be defined in many levels given a base context space, and the definitions can be related to the shape of the context space. In the most general case, we have at least the “bottom” and “top” levels of dimensionality, that is, a construct can be constant or non-constant on the context space. When the context space consists of several orthogonal dimensions, dimensionality can be defined in terms of varying in certain dimensions or not. Dimensionalities of constructs in

indexical imperative programs can be either explicitly specified by the programmer or analyzed by the compiler.

We have designed a C-based indexical language *C-Limpid* according to the above ideas. The name *Limpid* refers to a language design model for enriching conventional languages with indexical semantics. In *Limpid*, we take a conventional typeful language, and extend its syntax and semantics as follows.

- User-defined context space.

The context space of a *Limpid* program can be defined to consist of one or more general dimensions. By general we mean that they are not necessary linear integer dimensions and can have any structures. Each of the dimensions is declared by using a type expression. In other words, a value with the type is a coordinate of the dimension for some context in the context space. Each context is indexed by an n -tuple of such coordinates, where n is the number of dimensions. The size of each dimension also can be specified by giving a predicate on the coordinates of the dimension. Currently we only allow static context space declarations.

- Dimensionality declaration.

For indexical imperative languages, the programmer can declare dimensionality for any variable or not at all. The variables whose dimensionalities cannot be analyzed will have maximum dimensionality.

- Primitive context-switching operators

For each declared dimension, there are two primitive indexical operators, an index variable and a generic context-switching operator. The index variable has the same name as the dimension's. In a context, the index variable has the value which is the coordinate of the dimension. Given a dimension d , in a context c , the generic context-switching operator $atd(x, i)$ switches context to c' along dimension d where c' 's coordinate for dimension d is the same as the value of i in context c , and others are the same as those of c . The operator is overloaded, that is, it evaluates x if x is an expression or executes x if x is a statement in context c' .

The following is a C-Limpid program for rotating a raster picture on a two dimensional grid about an angle of 90 degree clockwise. One solution is to divide the picture in four quadrants, exchange their data cyclically and iterate the procedure at half grain size until the pixel-level is reached [Bra89]. Figure 1 illustrates the algorithm stepwise. In the initial picture, the *arrow* points to the lower left. Divided into four quadrants, the image data is cyclically being exchanged. The

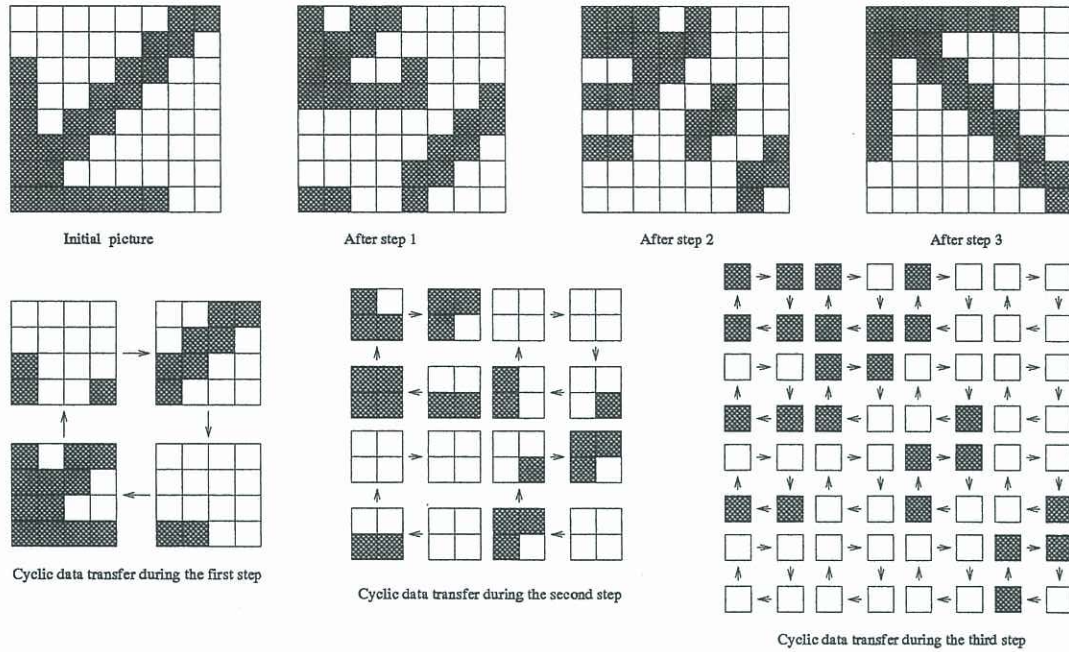


Figure 1: Illustrating the image rotation algorithm

next recursive steps will be in parallel with decreasing side length. Thus for n pixels, after $\log_2(n)$ steps, the picture is rotated.

```
#define size = 1024

dimension int row; element row >= 0 && row < size;
dimension int col; element col >= 0 && col < size;
int picture;
dimensionality [row,col] picture;
int right(int x, int n) {return atrow(x, row+n);}
int left(int x, int n) {return atrow(x, row-n);}
int up(int x, int n) {return atcol(x, col+n);}
int down(int x, int n) {return atcol(x, col-n);}
int rotate(int pic, int picsize) {
    int x, y, size2;
    size2 = picsize / 2;
    x = row % picsize;
    y = col % picsize;
    if (x < size2 && y < size2) return right(pic, picsize);
    else if (x >= size2 && y >= size2) return left(pic, picsize);
```

```

    else if (x >= size2 && y < size2) return up(pic, picsize);
    else return down(pic,picsize);
}

main(){
    int picsize = size;
    scanf("%d", &picture);
    while (picsize > 0)
    {
        picture = rotate(picture, picsize);
        picsize = picsize / 2;
    };
    printf("%d", picture);
}

```

3 Indexical Characteristics of Data-Parallel Languages

It is not surprising that existing data-parallel languages have the three indexical characteristics, intensionality, dimensionality, and context-switching operations, when we give indexical interpretations to their parallel constructs.

We can think of the processor space of a SIMD machine as the underlying context space for all the programs to be run on the machine. Given a program, each processor corresponds to a context and executes the computation thread associated with the context. The SIMD operational semantics of DPLs corresponds to the synchronous semantics of indexical imperative languages. Or in fact, here we *borrow* the synchronous semantics from DPLs. On the other hand, the formal semantics described above is also a framework for defining the denotational semantics of the DPLs.

In the following subsections, we discuss the indexical characteristics, their realizations, and restrictions in the DPLs. We use C* and MPL as examples to illustrate the points.

3.1 Intensionality

Intensionality in a programming language means that the meaning of a basic program construct depends on an implicit context in a given context space, or it is a mapping (intension), from the context space to its all possible meanings. So, there are two things that make a language have intensionality, the concept of programs' context space and the basic program constructs that have intensional meanings.

In MPL, the context space can be considered to be the processor space of the underlying machine, whose size may be 1k, 2k, 4k, 8k, or 16k. That is, the context space for every MPL program is pre-defined and the same, and it is machine-dependent. The context space is indexed by a single integer (*inproc*) or a pair of integers (*ixproc*, *iypoc*). Thus the shape of the context space can be considered as a one-dimensional array or two-dimensional array. The basic program construct in MPL, which brings parallelism in MPL programs, is plural variables. A plural variable represents a memory location on each context (processor) in the context (processor) space. In different contexts (processors), a plural variable may have different values. In an MPL program, an expression which involves plural variables is a plural expression whose value may also vary in the context space. An assignment to a plural variable assigns value to the variable in each context; it is a plural statement. In general, any statement involving plural variables, expressions, and other plural statements is also a plural statement. In the above sense, we say that in MPL all plural variables and expressions have intensionality in terms of their values, and all plural statements have intensionality in terms of their behaviors.

For example, the following is a plural version of Euclid's classical Greatest Common Divisor algorithm in MPL. Values of x and y may be different on each processor (context).

```
plural int GCD(x,y)
  plural int x,y;
{
  while (x != y )
    if (x > y) x = x - y;
    else      y = y - x;
  return x;
}
```

The context space definition in C* is more interesting. In C*, we consider *shapes* are components of the program's context space. A shape as a kind of *type* in a C* program is user-defined, and it has almost the same form as an array but with left indexing. For example, the following declaration declares three shapes *line*, *plane*, and *cube* with sizes 100, 100×100 , and $100 \times 100 \times 100$, respectively.

```
shape [100]line, [100][100]plane, [100][100][100]cube;
```

A shape declaration may fully specify a shape as above, partially specify a shape which only rank (or dimensionality) declared, or fully unspecified, i.e. only a shape name is given. An unfully specified shape can be instantiated later in the program through assignments and allocations. Shape pointers and shape arrays, each of whose element is a shape, can also be declared. After a shape is fully

specified, parallel variables can be declared with the shape. That is, a variable with a shape type have a value associated with each position of the shape; at different positions the variable may have different values. The variable can also be assigned a value at each position of the shape by a single assignment statement. For example, the following is a C* function to compute the cube roots of 8192 values in parallel.

```
shape [8192]cubes;
double:cubes cuberoot(double:cubes a) {
    double:cubes x, nextX;
    int:cubes active;
    nextX = 1.0;
    active = 1;
    do
        where(active) {
            x = nextX;
            nextX = (1.0/3.0)*((x+x) + a / (x*x));
            active = (fabs(nextX-x)>=0.001);
        }
    while(|=active);
    return nextX;
}
```

The above function definition specifies the parallel computation for the cube root of parallel variable *a* at each position of the shape *cubes*. The computation at a position terminates when the parallel variable *active* with the same shape has become 0 at the position.

In terms of indexical semantics, we consider each shape declared and instantiated in a C* program as an individual context space. The program can have one or more these individual context spaces. The intensionality of parallel variables, and hence parallel expressions and statements, can be defined based on their corresponding shapes or context spaces. That is, a parallel variable has intensionality inside its shape but not globally in the whole program. Interactions between values with different shapes are generally not allowed, but can be done in certain ad hoc ways, such as shape casting and some sort of context switching. Globally, we can still consider that a C* program has a global context space consisting of multiple shapes, each of which is a dimension of the context space. Here the concept of a dimension is a general sub-context space. In terms of such global context space, the intensionality of parallel variables can be interpreted with the concept of dimensionality, which we discuss next. The context space definition in C* is dynamic, which depends on shape instantiations

and allocations at run-time. Hence the context space of a program is also dynamic, that is, at different computing points, the context space may be different, which gives more difficulty in defining its denotational semantics.

3.2 Dimensionality

In DPLs, there are two classes of constructs: parallel and singular. We say that parallel constructs have dimensionality in the underlying context space, because they may have different meanings in different contexts, which can be virtual processors or shapes. By contrast, a singular construct, which is usually stored, evaluated, or executed on a single host, does not have dimensionality because its value or behavior is irrelevant to the underlying context space. In other words, in every context, its meaning is always the same. In this sense, we say that a singular construct has constancy.

In MPL, a plural variable must be declared explicitly using the keyword “plural”. In terms of dimensionality, we can also treat this as an explicit dimensionality declaration, that is, it declares the variable to have dimensionality in the context space. In contrast, a singular variable is declared with conventional C notation. It implicitly declares that the variable has constancy in the entire computation. Based on these explicit dimensionality declarations of variables, all other constructs in an MPL program can also be classified into the two categories, that is, those involving plural variables, directly or indirectly, are parallel and others are singular. This classification is very important in terms of implementation, because they will be stored, evaluated, or executed on the host or parallel processors accordingly. The implicit rule for combining a plural value with a singular value is similar to that in indexical languages. The singular one will be first promoted to the corresponding plural variable with constant value in every context. In the implementation, this promotion is implemented by broadcasting from the host to every processor. Although the context space of an MPL program is two-dimensional, but the language’s semantics does not define the rule to combine one- and two-dimensional values. This can only be done by using library functions.

Similarly to MPL, in C*, a variable is parallel when it is declared with a shape, otherwise it is singular or scalar. A parallel expression with a shape can be combined implicitly with a scalar expression also by promoting the scalar one with the shape first. C* also does not define in the semantics about how to combine values with the same multidimensional shape but varying in different dimensions. This kind of combination can only be done by explicitly using the library function *spread*. Another level of dimensionality in a C* program is to consider shapes as dimensions in a global context space. In this sense, a parallel expression has the dimensionality as the same as its shape. In other words, we may consider that it is constant in other shapes or dimensions. However, by the language’s semantics, any implicit combination between expressions with different

dimensionalities (shapes) at this level is disallowed. Finally, in C*, control constructs with conditions also have dimensionality, that is, they are exclusively for scalar conditions such as *if* and *while*, or exclusively for parallel conditions such as *where*.

3.3 Context switching operators

As same as in indexical languages, an arithmetic or logic operator in DPLs can be considered as a pointwise operator when it is applied to parallel expressions. In this case, the operator performs in parallel the same operation on the values of the expressions in every processor or position of a shape (i.e. context), without referring to values in other contexts. This is elementwise parallelism in terms of data-parallel programming, or trivial context parallelism in terms of indexical parallel programming.

In order to let parallel elements in different processors on the processor space or in different positions in a shape to communicate with each other, DPLs also provide a kind of communication operators to do the job. Here we consider the communication operators as context-switching operators in terms of indexical semantics. The communication operators have three basic tasks in DPLs:

1. Obtaining data from another context:

This is context switching on an r-value.

2. Sending data to another context:

This is context switching on an l-value.

3. Requesting to execute a command in another context:

This is context switching on a statement.

In MPL, three kinds of context-switching operators are provided, which can be applied to a plural expression. Each kind is in a complex level of context switching.

1. At the simplest level, the operator

$$\text{proc}[i][j].\text{plural_exp}$$

switches from the current context to the context indexed by i and j and evaluates the *plural_exp* in the new context. Here i and j must be singular, i.e. they must be constant in the context space. This context switching is independent of any particular context, or it is an absolute context switching.

2. At the second level, the operator

$$\text{xnetD}[k].\text{plural_exp}$$

switches context from the current context to the context that is k positions away from the current context in direction D , where D is one of the eight directions on a two-dimensional grid: North, North East, East, South East, South, South West, West, and North West. Here k must be singular, i.e. it is constant in the context space. This context switching is context-dependent or synchronic; it is a relative context switching. However, since k is constant, when the operator is applied to a plural expression in every context, it switches the context uniformly for every one. There will be no potential conflict of switching from two contexts to a same context. This context-switching operator is specially designed for MasPar. It is implemented rather efficient on the machine.

3. At the most general level, the operator

$$\text{router}[\text{plural_index_exp}].\text{plural_exp}$$

switches from the current context to any other context depending on the value of *plural_index_exp* in the current context. This is a context-dependent (synchronic) and data-dependent generic context-switching operator. It is context-dependent when there is index variables involved in *plural_index_exp*.

In C*, context-switching operators take a unified form using left-indexing on parallel expressions of a given shape. For example, when the shape is two-dimensional, the context-switching operator

$$[i][j].\text{parallel_exp}$$

switches from the current context to the context indexed by the values of i and j in the current context, where i and j can both be parallel expressions. When i and j are constants in the context space, this is a context- and data- independent context switching, or absolute context switching. When i and j depend on the index variables of the two dimensions, it is a synchronic context switching. For example, the following context-switching operator in C* has the same behavior as the *xnetNW* operator in MPL, when k is scalar or constant in the context space.

$$[\text{pcoord}(0)-k][\text{pcoord}(1)-k].\text{parallel_exp}$$

where the index function *pcoord*(i) gives the coordinate of the i^{th} dimension of the current context. Here the dimensionality of left indexing expressions is an important factor to determine the semantics of context switching.

1. Data- and context-independent context switching on an r-value means the value being broadcast from the context switched to to all contexts where the context-switching operation performed. For example,

```
x = [5][10]y;
```

2. Data- and context-independent context switching on an l-value causes updating conflict in the remote context, since in this case there will be more than one value to be assigned to the same location in the remote context. For example,

```
[5][10]x = y;
```

3. Pure context-dependent context switching on a parallel expression switches context in a uniform way with regular patterns and without causing update conflict. For example,

```
x = [pcoord(0)+1][pcoord(1)+2]y;  
[pcoord(0)+3][pcoord(1)+4]x = y;
```

4. The combinations of data- and context-dependent context switchings are most general, which can switch from any context to any other context. For example,

```
x = [pcoord(0)+i][pcoord(1)+j]y;
```

Neither MPL nor C* allows context-switching operators on statements.

4 Data-Parallel Programming in Indexical Imperative Language

In data-parallel programming, indexical imperative languages, such as C-Limpid, have at least the same expressivity as the DPLs. As a programming example, in this section, we first define the machine-specific operators *xnetD* of MPL using C-Limpid. We then give an MPL version and a C-Limpid version of a parallel matrix multiplication program.

The *xnetD* definitions in C-Limpid are as follows, assume the machine has 2k processors.

```
dimension int X; element X >= 0 && X < 64;  
dimension int Y; element Y >= 0 && X < 33;
```



```

xnetN (int k, <type> exp) dimensionality []k; {return atY(exp,Y-k);}
xnetNE(int k, <type> exp) dimensionality []k; {return atX(atY(exp,Y-k),X-k);}
xnetE (int k, <type> exp) dimensionality []k; {return atX(exp,X-k);}
xnetSE(int k, <type> exp) dimensionality []k; {return atX(atY(exp,Y+k),X-k);}
xnetS (int k, <type> exp) dimensionality []k; {return atY(exp,Y+k);}
xnetSW(int k, <type> exp) dimensionality []k; {return atX(atY(exp,Y+k),X+k);}
xnetW (int k, <type> exp) dimensionality []k; {return atX(exp,X+k);}
xnetNW(int k, <type> exp) dimensionality []k; {return atX(atY(exp,Y-k),X+k);}

```

The following is a data-parallel program in MPL for computing the product of two matrixes.

```

plural double matrix_multiply(A, B)
  plural double A, B;
{
  int i;
  plural double C = 0.0;

  /* precondition the input arrays */
  for (i=0; i<nxproc-1; i++) {
    if (iyproc > i) xnetW[1].A = A;
    if (ixproc > i) xnetN[1].B = B;
  }

  /* now do the multiply inner loop */
  for (i=0; i<nxproc; i++) {
    C += A * B;
    xnetW[1].A = A;
    xnetN[1].B = B;
  }
  return C;
}

```

The following is an equivalent C-Limpid program, using the defined context space $X \times Y$ and the *xnet* context-switching functions.

```

#define limit = 64

```

```

double matrix_multiply(double A, double B)
{
    int i;
    double C = 0.0;

    for(i=0; i<limit-1; i++) {
        if (x > i) xnetW(1, A) = A;
        if (y > i) xnetN(1, B) = B;
    }

    for(i=0; i<limit-1; i++) {
        C += A * B;
        xnetW(1,A) = A;
        xnetN(1,B) = B;
    }
    return C;
}

```

From the above examples, we can see that there is a relatively trivial mapping from C-Limpid programs with the machine-specific restrictions to MPL programs. In other words, for those C-Limpid programs, their implementations on MasPar will have the same performances as the corresponding MPL programs.

5 Concluding Remarks

In the above sections, we have discussed three issues:

1. What are indexical imperative languages?
2. What are applications of indexical imperative languages?
3. Are the existing data-parallel languages also indexical imperative languages?

We believe that indexical programming as a general programming paradigm should also apply to imperative languages. In the author's opinion, indexical semantics is orthogonal to other language semantics including declarative semantics, instead of relying on it. What distinguishes indexical

languages from others are their three characteristics: intensionality, dimensionality and context switching, not the declarativeness.

We also believe that imperative languages can be enriched with indexical semantics in almost the same way as we enriched declarative languages. However, in designing indexical imperative languages, there are some issues related to the imperative semantics which the declarative languages do not have, such as context switching on l-expressions and statements, synchronous or asynchronous semantics, and the meanings of conditionals. This paper has given some thoughts about these issues, but they are neither guaranteed sound nor complete. Further research on these issues are needed.

An immediate application of imperative indexical languages is for data-parallel programming, especially when we give synchronous semantics to the languages. As we discussed in the paper, there is a natural match between indexical imperative languages and data-parallel languages. This match brings at least two good things to data-parallel programming. First, because of their mathematical semantics, indexical imperative languages can be considered as a kind of universal data-parallel languages whose programs have great portability. Secondly, using indexical semantics we can define the formal semantics of data-parallel languages, so that the correctness of their programs can be verified and reasoned.

Indexical languages do not have to be descendants of Lucid or originally designed based on the indexical or intensional paradigm. In this paper, we have argued that existing data-parallel languages, such as MPL and C*, have the three essential characteristics of indexical languages. We believe that we should classify at least the cores of the data-parallel languages as indexical languages. For the indexical or intensional meanings in these languages are obvious when we interpret them based on the intensional paradigm. This classification will help us further argue that indexical or intensional programming is natural and already being used widely without knowing the theory.

References

- [Bra89] T. Braunl. Structured simd programming in parallax. *Structured Programming*, 10(3):121–133, July 1989.
- [Du93] W. Du. Parallel programming in the intensional language ulucid. In *26th Hawaii International Conference on System Science*, Jan. 1993.
- [FJ91] A.A. Faustini and R. Jagannathan. Indexical lucid. In *The 1991 International Symposium on Lucid and Intensional Programming*, pages 19–34, April 1991.

-
- [Fra91] J.L. Frankel. C* language reference manual. Technical report, Thinking Machines Corporation, May 1991.
- [Hig92] High Performance Fortran Forum. High performance fortran language specification. Technical report, Nov. 1992.
- [Mas92] MasPar Computer Corporation. Mpl (maspar programming language) manual. Technical Report 9300-9034-00, Nov. 1992.
- [OW89] M.A. Orgun and W.W. Wadge. A theoretical basis for intensional logic programming. In *The proceedings of the 1991 International Symposium on Lucid and Intensional Programming*, May 1989.
- [Pla92] J A.. Plaice. Ml-lucid, an intensional functional language. In *The 1992 International Symposium on Lucid and Intensional Programming*, pages 54–62, April 1992.
- [WA85] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.