

Two Indexical Parallel Programming Techniques

Weichang Du

Department of Computer Science

University of Victoria, B.C. Canada V8W 3P6

Abstract

Programs in indexical programming languages have implicit parallelism called **context parallelism**. The limitation of context parallelism in the programs is the data dependence between values of objects at different contexts. This dependence relationship is solely determined by explicitly uses of indexical operators in the programs, which is called **indexical communication**. Using context parallelism and indexical communication, indexical languages can express not only implicit but also explicit parallelism. The latter is achieved by giving operational interpretations to indexical semantics of the programs. This paper shows two explicitly parallel programming techniques, *systolic programming* and *parallel functional programming*, in the indexical language *mLucid*, which is a multidimensional extension of the programming language Lucid.

1 Introduction

Parallel programming can be classified into two categories: explicitly parallel programming and implicitly parallel programming. In explicitly parallel programming, programs are written with explicit parallelism in so-called parallel programming languages. A parallel programming language provides special constructs for programmers to express parallelism explicitly in their programs. The meaning of the constructs are usually defined by the language's operational semantics, instead of its mathematical semantics. By contrast, in implicitly parallel programming, programs are written in conventional programming languages without explicit constructs for expressing parallelism. It is compilers' task to detect parallelism in the programs.

Explicitly parallel programming has advantages. For application problems that need fast solutions on parallel computers, if compilers cannot produce implementations with desired high-performance, explicitly parallel programming lets programmers directly take control to optimize implementations of their programs.

To express control information on parallelism explicitly in languages with implicit parallelism, some research on programming languages suggested to extend the languages with explicit constructs [Sha87][Hud86] for parallel programming. The extension gives the constructs operational meaning related to parallel processing, but it is different from the languages' original semantics.

Here we are facing a dilemma that, as long as parallel compiling techniques have not been fully developed, programmers have to either write programs with implicit parallelism at risk of losing efficiency, or use explicitly control constructs that are neither a part of their solutions nor

a part of the original languages. A solution for the problem proposed in this paper is that we need a programming language that has implicit parallelism but meanwhile, without changing the language's semantics, it can also be used to express explicit parallelism as programmers desire.

Indexical languages like Lucid can be considered as extensions of conventional programming languages with indexical semantics. Programs in indexical languages have implicit parallelism. This parallelism is inherited from both semantics of the base languages, such as expression parallelism in functional languages, and indexical semantics. Implicit parallelism based on indexical semantics is called **context parallelism**[AF89]. Context parallelism in an indexical program is distinct from that inherited from the base language. It is exploited by evaluating values of objects in the program at different contexts simultaneously. The limitation of context parallelism in the program is the data dependence between values of objects at different contexts. This dependence relationship is solely determined by explicit uses of indexical operators in the program. If we consider that computations for values of objects in a program at the same context constitute a parallel computing task, an indexical operation that switches context from a context to another expresses communication between the tasks corresponding to the two contexts. In this sense, we call the indexical operation **indexical communication** between the two contexts.

In the previous research [Ash85][AF89][AJ89][Ash90], indexical programming has been shown to have great potential for parallel programming. The results described in this paper can be considered as an extension of their work. In this paper, we show that, without extending indexical languages' semantics, context parallelism and indexical communication can be used for expressing explicit parallelism in indexical programs.

In the rest of the paper, we first introduce an indexical language called **mLucid** in Section 2, which is a multidimensional extension of Lucid. In section 3 and 4, we show two indexical parallel programming techniques: systolic programming and parallel functional programming. In Section 5, we give some concluding remarks.

2 mLucid – a Multidimensional Extension of Lucid

mLucid extends Lucid's context domain to an n -dimensional integer space. Each point in the space is indexed by a vector of integers with size n , each of which corresponds to a dimension designated by its position in the vector. Since n is arbitrary, theoretically the vector has infinitely large size. The value of an expression in an mLucid program depends on a point in the n -dimensional space; at different points or contexts the expression may have different values.

mLucid also extends the primitive indexical operators of Lucid, **fby**, **first** and **next**, by adding an extra parameter called **dimension indicator** to the operators. The dimension indicator indicates the dimension in which the operators switch context. Generally speaking, the dimension indicator d can be any expressions which have nonnegative integer values. But to simplify discussions in this paper, we restrict d to be a constant nonnegative integer in our example programs.

To deal with the negative direction in each dimension, mLucid also adds two primitive indexical operators **before** and **prev** (for "previous"), which are symmetric to **fby** and **next**, respectively. In order to give intuitive meaning, we also change the name of operator **first** to **origin**, representing the origin or the center point of a dimension.

Using indexical operators, indexical programming usually eliminates most of explicit references to indices of contexts in programs. However, explicit references sometimes are still not avoidable, especially when there are irregular communication patterns among computations at different contexts in a program. For this purpose, we also define a built-in index operator `index(d)` in mLucid using the primitive indexical operators *fb*y and *before*.

$$\text{index}(d) = \text{fb}y(d)(\text{before}(d)(\text{index}(d)-1, 0), \text{index}(d)+1).$$

At a point p in the space, the operator $\text{index}(d)$ returns p 's coordinate for dimension d .

In a mLucid program, there is an upper bound of dimensions for an expression; outside of the dimensions the expression's value keep constant. We call this set of dimensions **dimensionality** of the expression. For example, a constant in mLucid has dimensionality $\{\}$; a variable whose value represents a vector in dimension 0 has dimensionality $\{0\}$; a variable whose value represents a matrix in dimensions 1 and 2 has dimensionality $\{1,2\}$. The union of dimensionalities of all expressions in an mLucid program is the maximum dimensionality of the program. In other words, the dimensions in the union constitute the n -dimensional space or the context domain of the program, where n is the union's cardinality.

The dimensionality of an expression has run-time property. However, an approximation of the dimensionality of an expression in an mLucid program can be obtained by analyzing the program at compile-time. The analysis is based on the following informally described assertions.

- The dimensionality of a pointwise operation is a subset of the union of the operands' dimensionality.
- The dimensionality of the operation $\text{origin}(d)(x)$ is a subset of x 's dimensionality without d .
- The dimensionality of the operation $\text{next}(d)(x)$ or $\text{prev}(d)(x)$ preserve x 's dimensionality.
- The dimensionality of the operation $\text{fb}y(d)(x,y)$ or $\text{before}(d)(x,y)$ is a subset of the union of dimensionalities of x and y and $\{d\}$.

In mLucid, dimensionalities of input variables must be declared by a dimensionality declaration in the beginning of the program in the form like:

$$\text{dimensionality } x,y:\{1,2\}; z:\{1\}; n\{ \};$$

The declared dimensionality of an input variable is the maximum set of dimensions in which the variable's value can vary.

It has been proved [Du91] that the maximum dimensionality of an mLucid program is a subset of the union of declared dimensionalities of all input variables, and all occurrences of dimension indicators in the indexical operations *fb*y and *before*.

In the following discussions, the term "dimensionality" means its approximation, and the term "mLucid(n)" means all programs in mLucid whose maximum dimensionality consists of n dimensions.

3 Systolic Programming

Inspired by systolic arrays, the term "systolic programming" was originally proposed by [Sha87]. Systolic programming uses ideas extracted from systolic arrays for general parallel programming. In systolic programming, we first define an abstract machine. The machine is an n -dimensional grid; at each gridpoint there is a programmable processor. Each processor consists of a set of input ports and a set of output ports. An input port receives a data stream from an output port at another processor or from the outside world. The value of an output port, which is also a data stream, depends on the values of some of input ports at the same processor. By defining the values of output ports of a processor, the functionality of the processor is defined. By defining the dependence relation between an input port at a processor and an output of another processor, the communication path between the two processors through grid edges is defined. The task of systolic programming is to define functionalities for processors and communication paths among them. The definitions constitute a systolic program.

We may also call systolic programming **multidimensional dataflow programming**. In Lucid, a program specifies a dataflow network. We consider that such a dataflow network in Lucid defines the behavior of a processor in a systolic program; all homogeneous processors are copies of the same dataflow network being distributed in the space.

In the following, we describe a systolic program for solving linear equations using Gauss elimination. To explain this and later examples intuitively, we use the following alternative notations for indexical operators that switch context in dimensions 0, 1, and 2, which we call *temporal*, *horizontal* and *vertical* dimensions, respectively.

original name	new name	original name	new name	original name	new name
origin(0)	first	origin(1)	origin_horizontal	origin(2)	origin_vertical
next(0)	next	next(1)	right	next(2)	up
prev(0)	prev	prev(1)	left	prev(2)	down
fby(0)	fby_time	fby(1)	fby_horizontal	fby(2)	fby_vertical

Given a set of linear equations such as

$$\begin{aligned}
 a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + a_{03}x_3 &= b_0 \\
 a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\
 a_{20}x_0 + a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\
 a_{30}x_0 + a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3
 \end{aligned}$$

the systolic algorithm for solving the equations is as follows[Sha87]. The algorithm has two phases: elimination and back-substitution, which are shown in Figures 1 and 2, respectively.

There are two types of processors in the algorithm: pivot (in circle) and cell (in square). In the elimination phase, the coefficient matrix A is input at the bottom processors; each of the processors receives an input data stream consisting of consecutive elements of a column of A . The vector B as a data stream is input into the pivot processor at the bottom. The functionalities of a pivot processor and a cell processor are shown in Figure 1, respectively. Here we use the temporal indexical operation, $first(Xin)$, (where Xin represents Ain or Bin in the figure,) to access the first element of the input steam Xin and to store it during the rest of the computation. We use the

temporal indexical operation, $next(Xin)$, to access the rest of the stream after removing the first element from the input stream Xin .

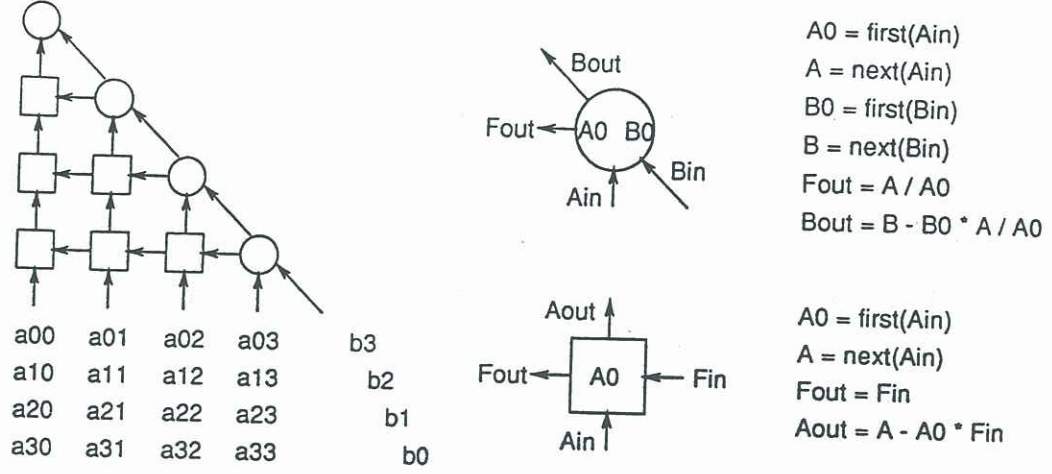


Figure 1: The elimination phase for solving linear equations

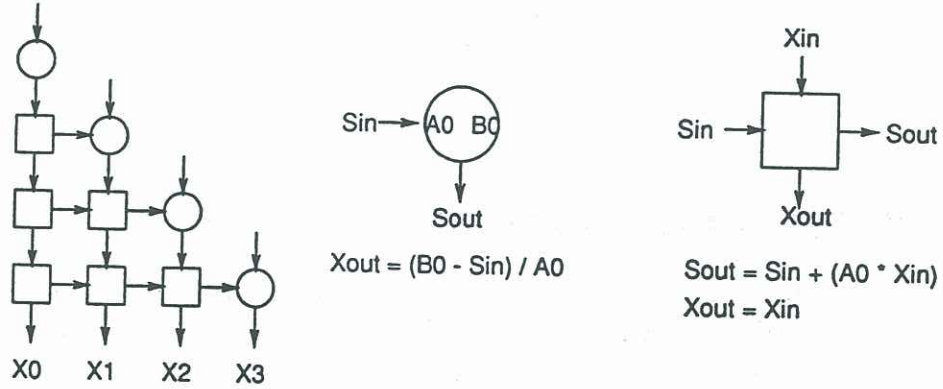


Figure 2: The back-substitution phase for solving linear equations

From the above definition, we can see that at each horizontal line i of processors from bottom up, x_{n-i} is eliminated, where $n+1$ is the number of the equations. Eventually, when the data reaches the top pivot processor of the array ($i = n$), the value of x_0 is computed. In this phase, data flows from bottom up and from right to left.

In the back-substitution phase, the computed x_i flows back from top down. When x_i arrives at a cell processor at horizontal line $n - i - 1$, the processor computes the product of x_i and the stored coefficient A_0 , adds the product to the partial sum of the products received from its left neighbor, and passes the new partial sum to its right neighbor. Eventually, the pivot processor in the line receives the sum of all the products from its left neighboring cell, and uses it to solve x_{i+1} . In this phase, data flows from top down and from left to right. The final solutions of x 's flow out from the processors at the bottom. Figure 2 shows the functionalities of a cell processor and a pivot processor in this phase, respectively.

The following systolic program in mLucid expresses the systolic algorithm.


```

dimensionality A:{0,1}; B:{0}; n:{};

first(origin_vertical(Xout))
where
  Ain = fby_vertical(A, Aout);
  Aout = next(Ain) - first(Ain) * Fin;

  Bin = fby_vertical(B, right(Bout));
  Bout = next(Bin) - first(Bin) * next(Ain) / first(Ain);

  Fin = right(Fout);
  Fout = if diagonal then next(Ain) / first(Ain) else Fin fi;

  Sin = fby_horizontal(0, Sout);
  Sout = Sin + Ain * Xin;

  Xin = up(Xout);
  Xout = if diagonal then (Bin - Sin) / Ain else Xin fi;

  diagonal = n - index(1) eq index(2);
end

```

Figure 3: A systolic Gauss elimination program

4 Parallel Functional Programming

Functional programs have implicit parallelism called expression parallelism. Expression parallelism usually has fine granularity. Several parallel tasks usually have to be scheduled or mapped onto a single processor in a parallel machine during parallel implementation of the program. Since there are no control constructs in functional languages that can express explicit information about how the tasks should be mapped onto the processors, the mapping is up to compilers. However, at the current stage of development of parallel compilers for functional languages, it is difficult for the compilers to schedule the tasks onto processors optimally, even if the programmer knows how to do it.

Motivated by the above problem, an explicitly parallel programming paradigm called *parallel functional programming* was proposed by [Hud86]. The parallel functional programming paradigm is considered as functional programming with control annotations. The control annotations specifies the mapping from expressions in a functional program onto processors. The operational semantics of such annotations are obviously unrelated to functional semantics of the original functional program. Hence the evaluation of such annotated functional program not only depends on its pure functional semantics, but also depends on the operational interpretation for the annotations.

The proposal of parallel functional programming gave us two related impressions. First, a programming language with implicit parallelism, like a functional language, cannot express

control information for parallel programming such as the processor mapping. To express this kind of information, therefore, explicit constructs unrelated with the language's original semantics must be introduced. Secondly, such extended language is not compatible with the original one. Therefore, to evaluate programs in the extended language specially designed interpreters or compilers are needed, otherwise the annotations have to be striped before the programs are evaluated.

Considering mLucid as a semantic extension of functional languages, in the following, we show that, without losing the language's functional semantics, programmers' control information on the processor mapping can be expressed explicitly in programs of a very restrict form of mLucid.

Since Wadge has solved the problem of implementing high-order functions in Lucid [Wad91], Lucid now has at least the same expressive power as other functional languages. Consider a program in mLucid with the following restriction. In the program there are no uses of indexical operators, and all input variables in the program have declared dimensionality {}, i.e. their values are constant at all contexts. Under this restriction, the program is just a pure functional program. The result of the program is the value of the program's defining expression at any context. The program is in mLucid(0).

Consider an expression E in the above program, whose value is constant at all contexts. If we apply an indexical operator to E , such as

$$\text{next}(d)(E)$$

the resultant value of the application is the same as E 's, since E 's value is constant in dimension d . We can conclude from this observation that if we modify the above mLucid(0) program by applying indexical operators like *next* to some of expressions in the program, the modified program has the same functional semantics as the original one, which is still in mLucid(0). It can be proved that the indexical operators that preserve this semantical equivalence are those whose results do not have higher dimensionalities than that of their operands, such as *next*, *prev*, and *origin*.

From the problem-solving point of view, the above additional indexical operations are meaningless. However, we can interpret these indexical operations with operational meanings in terms of the processor mapping.

Given a set of processors, let each of the processors corresponds to a context in the context domain of a mLucid(0) program, and let context-switching in the program represent communication among the processors. The following are some examples of how the mapping can be expressed in mLucid(0) programs. For simplicity, we assume the context domain consists of only dimensions, 1 and 2, i.e. the horizontal and vertical dimensions.

At a context (i,j) corresponding to processor p_{ij} , where i and j are coordinates for dimensions 0 and 1, respectively, we can interpret the evaluation of function application

$$f(e_1, e_2)$$

as that the evaluations for the arguments e_1 and e_2 as well as the application itself are all performed at processor p_{ij} .

For the function application with indexical operations

$$f(\text{left } e_1, \text{right } e_2)$$

we can interpret that the evaluations for the arguments e_1 and e_2 are performed at processors $p_{i-1,j}$ and $p_{i+1,j}$, respectively, but the application is performed at processor p_{ij} .

For the function application with indexical operations

$$(\text{up } f)(\text{left } e_1, \text{right } e_2)$$

we can interpret that the evaluations for the arguments e_1 and e_2 are performed at processors $p_{i-1,j}$ and $p_{i+1,j}$, respectively, and the application is performed at processor $p_{i,j+1}$.

For the function application with indexical operations

$$\text{up } (f(\text{left } e_1, \text{right } e_2))$$

we can interpret that the evaluations for the arguments e_1 and e_2 are performed at processors $p_{i-1,j+1}$ and $p_{i+1,j+1}$, respectively, and the application is performed at processor $p_{i,j+1}$.

We can also define an indexical operator $\text{at}(d)$, which lets a processor corresponding to a context communicate with another processor with an absolute address (i.e. context):

$$\begin{aligned} \text{at}(d)(k, E) = & \text{if } k \leq 0 \text{ then } \text{origin}(d)(E) \\ & \text{else } \text{at}(d)(k-1, \text{next}(d)(E)) \text{ fi;} \end{aligned}$$

where only the positive direction of dimension d is considered. For example, for the function application with indexical operations

$$\text{at}(2)(5, (f(\text{at}(1)(2, e_1), \text{right } e_2)))$$

we can interpret that the evaluations for the arguments e_1 and e_2 are performed at processors $p_{2,5}$ and $p_{i+1,5}$, respectively, and the application is performed at processor $p_{i,5}$.

The above expressing capability for the processor mapping in $\text{mLucid}(0)$ is based on the following observation. A pure functional program has expression parallelism, but not context parallelism. When we add indexical operators to the program, the context parallelism becomes to exist. Although the indexical operations do not alter the functional semantics of the program, their appearances in expressions means that the evaluations of the expressions are switched to different contexts, resulting in context parallelism. In other words, we can think of that the indexical operations *convert* expression parallelism in the functional program into context parallelism in the indexical program. Also, in this case, the shared variable communication in the functional program is *converted* to explicit indexical communication between contexts.

Finally, we give a parallel factorial program in $\text{mLucid}(0)$ [Hud86]. The program is expected to be executed on a finite binary tree of $n = 2^d$ processors. Each processor corresponds to a point i ($0 \leq i < n$) in the one dimensional integer space consisting of dimension 1. The root processor corresponds to point 0; a processor corresponding to a point i has its left and right children corresponding to points $2 * i + 1$ and $2 * i + 2$, respectively. The program splits the computation into two parts at each iteration, and maps the two subtasks onto the children of the current processor by the indexical operations *left* and *right*. The value of *mid* is computed at

the current processor, and then is passed to the children. Figure 4 shows the mapping and flow of data between the tasks at different processors when $k = 5$.

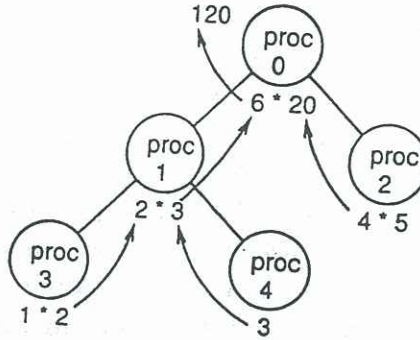


Figure 4: The mapping and dataflow of parallel factorial

```

root(pfac(1,k))
where
  pfac(lo, hi) = if lo eq hi then lo
                elseif lo eq (hi-1) then lo * hi
                else
                  left(pfac(lo, mid)) * right(pfac(mid+1, hi))
                  where mid = (lo + hi)/2; end
                fi;

  root(x) = origin(1)(x);
  left(x)  = if NoProc then x else at(1)(2*index(1)+1, x) fi;
  right(x) = if NoProc then x else at(1)(2*index(1)+2, x) fi;
  NoProc   = 2 * index(1) + 1 > n;
end.

```

Figure 5: A parallel functional program with indexical semantics for factorials

By the indexical semantics of mLucid, it is easy to see that the above program is semantically equivalent to the pure functional program when all the indexical operators and their definitions are striped from the program. In this sense, the indexical operations can also be treated as annotations in terms of pure functional programming. However these annotations preserve the functional semantics of mLucid.

5 Concluding Remarks

Indexical programming has the great potential for both implicitly and explicitly parallel programming. In this paper, we introduced the two parallel programming techniques: systolic programming and parallel functional programming. Systolic programming can be considered as explicitly parallel programming, when the technique is used for specifying systolic arrays or for programming programmable systolic arrays. It can also be considered as implicitly parallel

programming, when the technique is used for problem-solving. Parallel function programming with indexical semantics uses a very restricted form of indexical programming, but it shows that explicitly parallel programming in indexical languages is possible even if the languages have only implicit parallelism.

Other indexical parallel programming techniques are also under investigation by the author [Du91], such as distributed programming and data parallel programming. Indexical distributed programming is a further extension of indexical parallel functional programming, in which we no longer restrict our programs only in mLucid(0). By adding dimensionality to expressions, the distribution of both data and computations on parallel processors can be expressed by indexical programs. Indexical data parallel programming uses the extension and reduction of dimensionalities by indexical operations to express algorithms with data parallelism.

In this paper, mLucid, as an indexical extension of functional languages, is used for illustrating the parallel programming techniques. However, the techniques can also be used in indexical extensions of other styles of languages, such as indexical logic programming languages. For context parallelism and indexical communication exist in all indexical languages.

References

- [AF89] E.A. Ashcroft and A.A. Faustini. Adding intensionality to functional programs. In *The 1989 International Symposium on Lucid and Intensional Programming*, 1989.
- [AJ89] E.A. Ashcroft and R. Jagannathan. An intensional parallel processing language for application programming. In *The 1989 International Symposium on Lucid and Intensional Programming*, 1989.
- [Ash85] E.A. Ashcroft. Ferds – massive parallelism in lucid. In *the Phoenix Conference on Computers and Communications*, pages 16–21. IEEE, 1985.
- [Ash90] E.A. Ashcroft. Parallel programming in lucid. In *The 1990 International Symposium on Lucid and Intensional Programming*, 1990.
- [Du91] Weichang Du. *The Indexical Parallel Programming Paradigm*. PhD thesis, University of Victoria, 1991. In Preparation.
- [Hud86] Paul Hudak. Para-functional programming: A paradigm for programming multiprocessor systems. In *ACM Symp. Princ. Programming Languages*, pages 243–254, 1986.
- [Sha87] Ehud Shapiro. Systolic programming: A paradigm of parallel processing. In Ehud Shapiro, editor, *Concurrent Prolog*, volume 1, pages 207–242. The MIT Press, 1987.
- [Wad91] W.W. Wadge. Hight order lucid. In *The 1991 International Symposium on Lucid and Intensional Programming*, 1991.