Language Reference Manual
Matrix Language (ML)
COMS W4115

Kyle Jackson (kdj2109) • Jessica Valarezo (jgv2108) • Jared Greene (jmg2227) • Alex Barkume (ajb2233) • Caroline Trimble (cdt2132)

March 7th, 2016

Contents

1. Introduction

ML is a parallelized programming language with a focus on 1D and 2D operations, designed for computational efficiency and a strong focus on real-time applications. This language has the ability to segment large matrices into rows, columns or individual indices and run operations on them independently on various threads. This ability is particularly applicable to applying computer vision algorithms to images, including filtering, transformations, color space conversions, thresholding, and various others. When a number of these algorithms are applied sequentially to a very large image, their total execution time can be considerably long. As such, our language focuses on reducing the operational time of matrix algorithms. ML will support the .ppm image file type, which represents images as a text file of pixels.

ML's niche is image processing. By allowing the user to import images and quickly convert them into a matrix data type, ML gives a simple basis for image processing to the user. Once given this matrix data type of tuples that represents an image, the user can write a wide-variety of functions that manipulate the rows and/or columns of the matrix by iterating over them. By taking advantage of parallel processing, ML can iterate over multiple rows and/or multiple columns simultaneously, drastically reducing the operational time of image-processing functions. Creating multiple threads, especially those that need to wait and communicate with each other, can cause substantial problems and unwanted consequences. ML is a use-at-your-own-risk language, meaning that ML leaves it up to programmer to understand when and how to use parallel programming. ML provides a `pfor` loop, `asnyc` functions, and a `wait` for specific parallel-processing programming.

2. Types

2.1 Basic Data Types

Type	Description
`int`	32-bit integer value, automatic promotion to float.
`float`	32-bit floating point value.
`bool`	Boolean value: `true` or `false`.
`str`	a text value: multi ASCII character variable.
`matrix`	`Matrix` of `tuple`, `int`, or `float`. Every `tuple` must be of type `int` or `float`, and each `tuple` must be of the same length. All elements of a `matrix` must be of the same type.
`tuple`	A `tuple` is a single-type. Nesting is not allowed.

3. Lexical Conventions

3.1 Identifiers

An identifier can begin with any lowercase or uppercase letter (`^[A-Za-z]`). Thereafter, an identifier can have any combination of lowercase and uppercase letters, numbers, and underscore (`[0-9A-Za-z_]+$`).

3.2 Keywords

Keyword	Description

`#include <file_name>`	Imports declarations and implementations from file_name during compile time. Include statements must be the first lines in the program
`async <function_name>`	See **Parallel Constructs.**
`wait`	See **Parallel Constructs**.
`pfor`	* Every iteration of the for loop is run on a different pthread. *pfor (n; expr1; expr2; expr3) statement* where expr2 is a relational expression.
`for`	* *for(expr1; expr2; expr3) statement* where expr2 is a relational expression.
`if`	* If in if-else statement. *if (expr1) statement* where expr1 is a relational expression
`else if`	* Strings together multiple ifs in if-else statement. *else if (expr1) statement* where expr1 is a relational expression
`else`	* Else in if-else statement. *else statement*
`main`	* Main function. The code within the main function will be executed when the executable runs. The main function always returns type `int`
`return`	Return function value. In the case that the function has spawned threads, waits for all threads to terminate.
`while`	* *while(expr) statement* where expr is a relational expression
`func`	* Defines a function.
`break`	Breaks out of a loop.
`true`	Boolean literal value (True).
`false`	Boolean literal value (False).
`void`	No type

* - curly braces are only necessary if there are multiple statements within the block.

3.3 Comments

```
// Single line comment

/**
 Block comment
 spans multiple lines
**/
```

3.4 Operators

Operators	Description
=	assignment
*	multiplication
/	division
%	modulus
+	addition
-	subtraction
<	less than comparison
>	greater than comparison
>=	greater than or equal to comparison
<=	less than or equal to comparison
==	equality comparison
!=	inequality comparison
&&	logical AND operator
||	logical OR operator
!	logical NOT operator
;	statement separator
()	Access a specific element of tuple

Matrix Operators	Description
,	List of values.
{}	Encloses row or entire matrix.
:	a range of ints, upper bound is exclusive.
[]	Matrix indexing - begins at 0, 0 Matrix[row][column]
+, -, *, /	Matrix and scalar operations
<, <=, >, >=	Compares each entry in matrix to a single constant

	float or int, or to each corresponding entry in another matrix.

3.5 Precedence

Precedence	Expressions and Operators
highest	`<func_name>(arg1, arg2, ...)` `<matrix_name>[index1][index2]`
	`!`
	`* , /`
	`+, -`
	`<, >, <=, >=`
	`==, !=`
	`&&`
	`||`
lowest	`=`

4. Syntax

4.1 Expressions

Assignment Expressions

The assignment operators are binary operators with right-to-left associativity. It is an identifier and an expression separated by an equals sign.

Arithmetic Expressions

The arithmetic operator represent basic mathematical operations with left-to-right associativity.

Matrix Arithmetic Expressions

These represent operations on matrices.

Examples:

```
// Standard matrix scalar multiplication
M1 = M1 * 3;

// Standard matrix multiplication
M1 = M1 * M1;
```

Function Calls

To be able to call a function, it must have been declared and implemented before. The function call will execute using the given arguments and return whatever value was defined as the return type during its declaration.

All argument passing is done by-value, meaning a copy of each argument is made before the function has access to it as a parameter. Therefore, a function may change the values of the function parameters within the scope of the function block, without affecting the arguments in the function call.

The function call will return the value of the data type defined as a return type in the function declaration.

Because our language was designed with matrices and splitting up iterations to execute in parallel, this will help prevent race conditions.

```
function_name(<arguments>);
```

4.2 Declaration and Initialization

Basic Data Type Declaration

Basic data types are declared in the format:

```
type variable_name;
```

Variables can be local, formal arguments in a function, or global. The precedence of these variables is as follows:

Precedence	Variable Type
highest	local
	formal
lowest	global

Basic Data Type Initialization

Basic data types can be initialized in the same line as declaration, in the format:

```
type variable_name = literal;
```

Or, basic data types can be initialized in one of the subsequent lines of the program, in the format:

```
type variable_name;
variable_name = literal;
```

Example:

```
int x = 10;
int y;
y = 15;
float y = 5.0;
str z = “Hello, World!”;
bool b = true;
```

Tuple Declaration

Tuples are declared in the format:

```
type tuple variable_name;
```

However, if a user would like to declare a tuple and initialize it to `0, 0.0, false, "" or {}`, depending on the type, it can be declared in the manner.

```
type tuple variable_name[n] = {};
```

Tuple Initialization

A tuple can be initialized in the same line as its declaration, in the format:

```
type tuple variable_name = (literal, literal, literal,...);
```

A tuple can be initialized to any 1...n literals, all of type `type`. A tuple is not of constant size and, thus, is not declared with a size. The only time a size is specified is in the special case of declaring a tuple and initializing it to n zeros, in the manner `name[n]` (see above).

Tuples are a immutable data type, meaning any operation performed upon them will return a new instance of a tuple with the appropriate manipulations.

Matrix Declaration

Matrices can either be declared in the format:

```
type matrix variable_name;
```

or

```
type matrix variable_name[row][column];
```

where `row` and `column` are integers.

Example:

```
int matrix M0;
int matrix M1[3][4];
```

Matrix Initialization

Similar to basic data types, a matrix can be initialized on the same line as its declaration, or initialized in any subsequent line of the program. Furthermore, because of the two different declaration styles, there are two ways of initializing a matrix.

To utilize the `type matrix name` declaration style, a user must initialize a matrix structurally by listing each row between curly braces. Elements within rows are separated by commas, and rows themselves are separated by commas as well. See below:

```
int matrix M0 = {
        {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
```

```
        {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
        {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
      };

int matrix M1 = {{1,1,1},{2,2,2},{3,3,3}};
```

However, if the user utilizes the `type matrix name[row][column]` declaration style, a user can simply provide elements separated by commas, enclosed in curly braces. See below:

```
int matrix M3[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Users may also use the structural initialization, combined with the `type matrix name[row][column]` declaration style. However, if the [r][c] declaration and the structure do not match up, this will be an error.

```
int matrix M1[3][4] = {
   {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
   {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
   {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};

// 2-D matrix with tuple elements
tuple matrix M1[2][3] = {
    {(255, 255, 255), (255, 255, 255), (255, 255, 255)},
    {(0, 0, 0), (0, 0, 0), (0, 0, 0)}
};

//ERROR:
int matrix M1[2][3] = {
      {1,1,1},
      {2,2,2},
      {3,3,3}
};
```

It is possible to declare and initialize a matrix to just 0s by utilizing the `int matrix variable_name[row][column]` declaration style followed by `{}`. See below:

```
int matrix binaryMatrix[3][2] = {};
```

Accessing a Tuple Element

An element in a tuple is accessed by using its index in brackets following the tuple variable name. If saving in a variable, the type of the variable must match the type of the element (i.e. the type of the tuple).

```
type variable_name = t1[i];
```

Accessing a Matrix Element

An element in a two-dimensional matrix is accessed by using the subscripts, i.e. row index and column index of the matrix. If saving in a variable, the type of the variable must match the type of the element (i.e. the type of the matrix).

```
type variable_name = m0[row][column];
```

Example:

```
int val = m0[2][3];

int tuple pixel = m1[1][2];
printt(pixel);
pixel =
      0 0 0
```

Matrix Primitive Transformations

Primitive transformation can be used in combination to allow the programmer to express any arbitrarily complex matrix transformation. All of these transformations return a `matrix`.

```
// Returns a specified region of a matrix using python-like indexing
// Returns a matrix with 2nd and 3rd column of M1
M1[][1:3];

// Returns a matrix with 2nd and 3rd rows of M1
M1[1:3][];

// Returns a matrix with 1st row of M1
M1[0][];

// Returns matrix without the 1st column of M1
M1[][!0];

// Returns the intersection of 1st 3 columns and 3 rows
M1[0:3][0:3];
```

Examples:

```
// 1 row, 3 columns
int matrix M0 = {{1,2,3}};
printm(M0);
M0 =
      1  2  3

// 1 row, 1 column
tuple matrix M1 = {{(1,2,3)}};
M1 =
      1 2 3

// 1 row, 1 column
// Same as M1, uses whitespace in place of commas when creating tuples
tuple matrix M2 = {{(1 2 3)}};
printm(M2);
M2 =
      1 2 3
```

```
// 2 rows, 3 columns
int matrix M3 = {{1,2,3},{1,2,3}};
printm(M3);
M2 =
      1  2  3
      1  2  3

// 1 row, 3 columns
tuple matrix M3 = {{(255 255 255),(0 0 0),(255 255 255)}};
printm(M3);
M3 =
      255 255 255  0 0 0  255 255 255

// 2 rows, 3 columns
tuple matrix M4 = {{(255 255 255),(0 0 0),(255 255 255)},
      {(255 255 255),(0 0 0),(255 255 255)}};
printm(M4);
M4 =
      255 255 255  0 0 0  255 255 255
      255 255 255  0 0 0  255 255 255

// 1 row, 6 columns
int matrix M5 = happend(M0,M0);
printm(M5);
M5 =
      1 2 3 1 2 3
```

Function Declaration

Functions consist of a function header and a function body. The function header contains the keyword `func`, the `return_type`, which can be any of the fundamental types, including `void`. The header also contains the function name, which must be a valid identifier, and a parameter list enclosed in parenthesis. The function body is enclosed in curly braces.

Examples:

```
// Function with no return type (void)
func void <function_name> (<parameters>){ statement }

// Function with a return type
func return_type <function_name> (<parameters>){ statement }
```

4.3 Statements

Conditional Statements

There are two forms of conditional statements that consist of a `if` and optional `else if and else statements`.

Examples:

```
if ( boolean expression ) { block of statements executed on TRUE
evaluation; }

else if ( boolean expression) { block of statements executed on FALSE
evaluation of boolean expression in preceding if, FALSE evaluation of all
preceding else if statements, and TRUE evaluation of boolean expression }

else { block of statement executed on FALSE evaluation of boolean
expressions of preceding if statement and all preceding else if
statements; }
```

Looping Structures

ML has various looping structures. With `for` and `pfor`, all of the expressions must be present for the loop to execute. In the case of `pfor`, see **Parallel Constructs.**

Examples:

```
while ( expression ) { block of statements }

for ( expression; expression; expression ) { block of statements }

pfor ( k; expression; expression; expression ) { block of statements }
```

Loop Interruption

The entire execution of a looping structure can be terminated with `break.`

Parallel Constructs

async

The `async` statement spawns a new thread for the given statement. The thread will execute a function block of code in parallel.

```
async(<function_call>);
```

When `async` is given an optional `int fid,` it will pair the given function call on the spawned thread with an integer identifier.

```
async(<function_call>, int fid);  // Must use a unique int fid

<variable_name> = async(<function_call>);
```

wait

There are various ways in which the `wait()` statement can be called. In the case that no argument has been specified, `wait()` will wait for all previously spawned threads to finish executing. It will block any subsequent code from executing.

```
wait();
```

The `wait` statement can also wait for a specific function call. The `int fid` is an integer that was previously defined by the programmer to identify a function call when passed to `async`. This will help when there have been various calls to the same function, as a way of differentiating which function call to `wait` on. It will `wait` on all of the threads spawned by the specified function call to finish executing and prevent the execution of code following it.

This version of `wait` can not be paired with `async(<function_call>),` since there is no `int fid` associated with it.

```
wait (<function_name>, int fid);
```

Specifying an `int fid` is optional. In the case that the only argument is a function name, it default to `wait` on the most recent function call.

```
wait (<function_name>);
```

pfor

The `pfor` runs a loop in parallel, assigning its separate iterations to separate threads. Successful parallelization of a loop depends on whether the benefit of running it in parallel outweighs the communication costs. It is up to the user to know whether or not separate iterations are independent.
The user can specify `n` (an integer), the maximum number of threads to spawn, or it will default to a pool of n = 4. The user can not specify more than 4 threads.

```
pfor (n; expression; expression; expression) {statement}
```

return

The `return` is used within the scope of a function, and will wait for all of the threads the function spawned to finish executing before returning a value.

```
return expression;
```

In the case of a void function, `return` will still wait on a function's spawned threads to return.

```
return;
```

5. Standard Library Functions

5.1 Tuple

```
lenT(t);
//Returns the length of the tuple

insert(t, x, i);
//Inserts x at the i^th position of the tuple t

delete(t, i);
//Deletes from the i^th position of tuple t. Produces an error if i is out
of bounds.
```

5.2 Matrix

```
// Return the length of a row in a matrix.
len(M1[0][]);

// Return the length of a column in a matrix.
len(M1[][0]);

// Vertically append two matrices with the same number of columns.
vappend(M1, M2);

// Horizontally append two matrices with the same number of rows.
happend(M1, M2);
```

5.3 FILE I/O

Function	**Description**
`open(<path_to_img_file>)`	Opens a image file with a specified name and returns a matrix representing the image file.
`out(<matrix_name>, <path_to_img_file>)`	Opens a new file and writes the matrix to the image file.

```
// Prints out matrix
func printm(matrix M0);

// Prints out tuple
func printt(tuple T0);

// Prints out non-matrix and non-tuple data type
func print(int a);
```

Opening files, or writing to files, are two functions which can not be done asynchronously.

```
// Read in the image file, returns a matrix
func matrix open(file_path);

// Write matrix M1 to file
func void out(matrix M0, file_path);
```

6. Examples

Example 1

Every line of code is executed sequentially.

```
int main(){
  tuple matrix M1 = {{(255 255 255),(0 0 0),(255 255 255)}};
  tuple matrix M2 = {{M1},{M1},{M1}};
```

```
  printm(M2);
  // Update M2 to an int matrix of 1/0s instead of tuple matrix
  M2 = convertToBinary(M2);
  printm(M2);
  return 0;
}

func matrix convertToBinary(matrix M0){
  int thresh = 128;
  int matrix bMatrix[len(M0[0][])][len(M0[][0])] = {};

  // Outer loop runs iterates over every row
  // 5 threads are spawned
  pfor(5; i=0; i < len(M0[][0]); i++) {
    for (int j = 0; j < len(M0[0][]); j++) {
      if (M0[i][j] > thresh){
        bMatrix[i][j] = 1;
      }
    }
  }
  return bMatrix;
}

Output…
  M2 =
    255 255 255  0 0 0  255 255 255
    255 255 255  0 0 0  255 255 255
    255 255 255  0 0 0  255 255 255
  M2 =
    1 0 1
    1 0 1
    1 0 1
```

Example 2

There are blocks of code running in parallel.

```
int main(){
  tuple matrix M1 = {{(255 255 255),(0 0 0),(255 255 255)}};
  tuple matrix M2 = vappend(M1,M1);
  M2 = vappend(M2, M1);

  printm(M2);

  /** RGB values are converted to binary. The current thread waits for
      convertToBinary to return (see return in parallel constructs) **/
  M2 = convertToBinary(M2);

  // Two independent functions are run on parallel threads
  int sum = async(getSum(M2));
  int avg = async(getAvg(M2));

  // The current thread waits for all parallel threads to terminate
  wait();

  print(sum);
  print(avg);

  return 0;
}

//printm, Standard Library Function
func printm(matrix M){
      for (int i = 0; i < len(M[0][]); i++) {
            for (int j = 0; j < len(M0[][0]); j++){
                  print(M[i][j]);
                  print(" ");
            }
            print("\n");
      }
}

func float getAvg(matrix M0) {
  int sum = getSum(M0);
  int total = len(M0[][0]) * len(M0[0][]);
  return float(sum)/float(total);
}

func int getSum(matrix M0) {
  int sum = 0;
  pfor (5; int i = 0; i < M0[][0]; i++) {
```

```
    for (int j = 0; j < M0[0][]; j ++) {
      sum += M0[i][j];
    }
  return sum;
}
```

Output…

```
6
0.33
```

7. Semantics

Semantic
The user cannot sequence variable declaration. If variables are to be declared on the same line, the user can do so as follows: `<type> <variable_name1>; <type> <variable_name2>; …` In general, ML does not allow sequencing.
Function names cannot be overloaded and must have distinct names
Code blocks under a loop or function must be enclosed in brackets if there are multiple statements within the block.
Code statements must be terminated with a semicolon `;`.
Return statements in functions that cause unreachable lines of code are not allowed.
Functions that do not return a type and are used solely for its side effect must be declared as `void <func_name>`
The main function must always return type `int`.

9. References

1. B.W. Kernighan and D.M. Ritchie. "Appendix A Reference Manual," in The C Programming Language, 2nd edition. Murray Hill, NJ: AT&T Bell Laboratories.
2. Vector Programming Language
   http://www.cs.columbia.edu/~sedwards/classes/2013/w4115-fall/lrms/vector.pdf
3. SMPL Programming Language
   http://www.cs.columbia.edu/~sedwards/classes/2013/w4115-fall/reports/SMPL.pdf