DotPar
A Simple Tutorial
Team 3

Logan Donovan - lrd2127@columbia.edu
Justin Hines - jph2149@columbia.edu
Andrew Hitti - aah2147@columbia.edu
Nathan Hwang - nyh2105@columbia.edu
Sid Nair - ssn2114@columbia.edu

April 12, 2012

1 Introduction

Parallel algorithms tend to be described as operations on collections of values. For instance, "find the minimum neighbor for each vertex", or "sum each row of a matrix" are data-centric operations that can be done in parallel. [1] This ability to operate in parallel over sets of data is often referred to as data-parallelism. It is important that this data-parallelism can be exploited to its full potential. We want to be able to run parallel functions in parallel to maximize the efficiency of the program. Thus, we seek to provide an effective nested data-parallel language.

The following tutorial will give a quick tour of the basics of the language and build up to defining complex array manipulations to enable you to start writing useful programs as soon as possible. To do that, we'll concentrate on the basics: types, arithmetic expressions, arrays, control flow, functions, list comprehensions, and some nifty built-in functions.

2 Overview

2.1 Hello, World

DotPar was designed with simplicity and power in mind. To illustrate this, let's get started with a traditional "Hello, World" program that prints the words "Hello, World" to stdout.

[1] http://www.cs.cmu.edu/ scandal/cacm/node4.html

```
func main:void()
{
    println("Hello, world!");
}
```

Just like in C and Java, the DotPar execution begins with a function named `main`. Thus, every program must contain a main function. In this `main` method, we use the built-in function `print` to print the `string` "Hello, world". A `string` is an `array` of characters. We will explain exactly what `function`s, `string`s, and `char`s in later. But first, let's get this program running.

To compile and run this program there are three steps (assuming a UNIX-like system). First, we should save the function in a file called `helloWorld.par`. DotPar programs are stored in files that have the extension `.par`. Next, we run a compiler on the program file, like so:

```
dotparc helloWorld.par
```

`dotparc` can accept multiple files. So, to compile all the .par files in a directory, you would run:

```
dotparc *.par
```

If the program residing in the file has no syntax errors and is a complete DotPar program then the compiler will produce an executable file:

```
out.parc
```

If there are errors during compilation those will be printed to your console. In this example, running out.parc in the command line with:

```
./out.parc
```

will produce the output:

```
Hello, World
```

We'll now walk you through some key features of the language. As you go along, try writing and compiling these programs yourself.

2.2 Types And Arithmetic Expressions

DotPar contains the concept of types which are used to define a variable before its use. The definition of a variable will assign a store address for the variable and define the type of data that will be held at that address. DotPar only three contains basic types: `number`, `boolean`, and `char`.

2.2.1 Numbers and Basic Arithmetic Operators

All numbers in DotPar are double-precision 64-bit floating point numbers. This basic type is referred to as `number`. This simplicity allows for computations to be straightforward while maintaining incredible precision. We realized that since numbers were so imporant in the language that simplifying everything to high precision and using the parallelization for performance gains would make the language easier to use.

Our next program will illustrate the use of numbers and some basic arithmetic expressions that can be used to manipulate numbers.

```
func main:void()
{
    number a = 1;
    number b = 2;
    println(b);
    println(a / b);
    number c = a + b;
    println(c);
}
```

The first `println` statement may be counterintuitive to what one might expect. This statement will not return `2`, but rather `2.0`. Following this logic, the second print statement should, and does return `.5`. The last print statement's output should now be obvious as `3.0`.

Other basic arithmetic operators include `-` , `*`, and `/` which represent subtraction, multiplication, and division, respectively. There is also a remainder operator, `\%`, which returns the remainder of the division of two numbers.

Note: `+, -, *, /, sqrt, ln, log(num, base), exp(num, exponent), ceil, floor, trunc, round, sin(x), cos(x), tan(x), asin(x), acos(x), atan(x)` are all supported. The trigonometric functions return a value in radians, and the inverse trigonometric functions expect a value in radians.

2.2.2 Booleans and Operators

The boolean data type has only two possible values, `true` and `false`. This uses simple flags that track true/false conditions. While this data type represents one bit of information, but it is treated internally as a 32-bit entity. We decided to have booleans because they are a very useful data type. We implemented them as 32-bit entities because that is how Java does it.

The following program will give an elementary example of the use of booleans, and their uses.

```
func main:void()
{
    boolean b = true;
    println(!b);
```

```
    boolean c = false;
    println(b && c);
    println(c || b);
}
```

Since a boolean tracks a single true/false condition, each print statement will only return either `true` or `false`. The ! symbol is the NOT unary operator and evaluates to the complement of its argument. Since `b` is `true`, the first print statement will be "`false`". The `&&` operator signifies the binary operator AND which returns the conjunction of its two arguments. So the second print statement will return false, since true AND false evaluates to false. The `||` operator signifies the binary operator OR which returns the disjunction of its two operators. So the last print statement will return true, since false OR true evaluates to true.

2.2.3 Char

The char data type is a single 16-bit Unicode character.

The following example will given an elementary example of the use of chars.

```
func main:void()
{
    char c = 'a';
    println(c);
}
```

The above will print out "a", without the quotations. Often chars by themselves are not very useful, as in reality we use a sequence of chars to represent words, or strings. This provides a nice transition to DotPar's arrays.

2.3 Arrays and Control Flow

There are times when writing programs when you may want to store multiple items of same type together. Arrays provide this functionality. They are contiguous bytes in memory, which facilitates easy searching and manipulation of lists of elements. Like everything else in the language, arrays are typed. You can have an array of any type, such as `number`, `boolean`, `char`, or another array. Let's take a look at some examples.

2.3.1 Revisiting Hello, World

```
func main:void()
{
    char[] helloWorld = "Hello, World";
    println(helloWorld);
}
```

The first program we learned to write prints the words "Hello, World." In this equivalent program, we explicitly create an array of type `char` in the first line of `main`. An arrays of chars is also often called a `string`. Arrays are very fundamental building blocks in DotPar. Rather than have many built-in types, DotPar has just a few types and gives users the power to create more complicated types. So, there is no built-in `string` type, so there is instead only an arrays of characters. So, the first line creates a character array with each element being a character in the statement, "Hello, World".

Arrays are where DotPar leverages its power as it can manipulate large sets of data quickly by manipulating arrays, iterating through arrays, and applying functions to arrays.

2.4 Array Iteration

2.4.1 the each method

Our next example program creates an array of type number and demonstrates a method of iterating through an array. The first line creates an array arr of size 10 and populates it with the numbers 1-10. Notice at the end of the line there is a `//`. This is a single-line comment. `//` will cause the compiler to ignore the rest of the line. Comments are a useful to markup text to explain subtleties of the code. They are ignored by the compiler, so they are not evaluated. The second line demonstrates the use of the built-in `each` function, which is used to iterate over an array element by element. For each loops contain all of its statements within curly brackets. This for each iterates through every number in arr and calls the function print on every element. This prints a list of the elements in arr to standard out.

```
func main:void()
{
    number[] arr = [1,2,3,4,5,6,7,8,9,10];  //create the array
    each(arr, func:void(number n) { println(n); });
}
```

2.4.2 the for loop

Another way to write the program above would be to use a `for` loop, which can be used to iterated over the indices of an array by using a counter. So, the variable `i` of type number is created and is initialized to 0. It then checks to ensure that `i` is less than `len(arr)`. `len` is an example of a built-in function that returns a `number` whose value is the length of the array in its argument. In `for` loop we are again calling the `print` function and are passing in `arr[i]`, which is the value in the array at index `i`. The `for` loop then lastly increments `i` by 1.

```
func main:void()
{
```

```
    number[] arr = [1,2,3,4,5,6,7,8,9,10];
    number i;
    for(i = 0; i < len(arr); i = i + 1) {
        println(arr[i]);
    }
}
```

Arrays can be extended to arbitrarily many dimensions. For example, we create a matrix in the example below.

```
func main:void()
{
    number[][] matrix = [ [1,2,3], [4,5,6], [7,8,9] ];
    println(matrix);
}
```

Here we are creating a two-dimensional `3x3` array with the numbers 1 through 9. We have a set of `[ ]` for each dimension we want create. We can fill this array with nested brackets by defining each row within brackets. Try running this program and see what it prints!

2.5 Control Statements

Next, we'll look at conditional statements, random number generation, and the fill statement in DotPar.

Let's take a look at our next example program.

```
func main:void()
{
    /*
    This program creates an array of length 10 and
    fills it with random numbers in the range [0-100).
    It then prints out each element with
    a message declaring it even or odd.
    */

    number[] arr;
    fill(arr, func:number(number index) { return rand(100); }, 10);
    char[] even = is even;
    char[] odd = is odd;

    each(arr, func:void(number element) {
        if (element % 2 == 0) {
            println(element +   + even);
        } else {
            println(element +   + odd);
        }
```

```
    });
}
```

First, notice that the first few lines of the function demonstrate another way to comment in DotPar by using traditional C style comments, `/* ... */` to comment out an entire block of text.

Next, we create an number array arr. The next line calls the built-in function `fill(array, function)` which takes in an array and a function as arguments and inserts values into the array according to the passed in function. In this case we are passing in the `rand(number)` function. `rand` generates random numbers in the range [0, cap). Afterwards, we create two character arrays that contain the text "is even" and " is odd" which will be used later.

Next we have a call to our old friend, the `each` function, that iterates through each element in the array. Within the body of the `each` function we have our first example of `if` statements in DotPar. They follow the form:

```
if (conditional) {
   // code
} elif (conditional) {
   // code
} else {
   // code
}
```

There must be exactly one `if` clause, any number of `elif` clauses, and no more than one `else` clause.

The parenthetical after the `if` is the condition that must be met in order to run the code within the braces. If this condition is true, the code with the braces of the else is run. The conditional we test for in this example is `element \% 2` = 0=. `element` is the value in the array, the `\%` is the arithmetic operation for modulus, the

```
==
```

2.6 List Comprehensions

As a more complicated example, you may want to write something like:

```
number[] foo = [1,2,3,4,5];
number[] squares;
number i;
for(i = 0; i < len(squares) ; i = i + 1){
    if (squares[i] % 2 == 0) {
        squares[i] = exp(foo[i], 2);
    }
}
```

This function creates an array with the square of the even numbers in the original array. But this is rather verbose, which seems unnecessary for a conceptually simple task like this. DotPar provides an easy way to express this logic, borrowing the useful list comprehension from Python and Haskell. We will illustrate the power of list comprehensions by replacing the previous program fragment in one line:

```
number[] squares = [exp(x,2) for number x in [1,2,3,4,5] if (x % 2 == 0)];
```

You can also use an expanded notation for using n arrays, such as

```
number[] foo = [x*y for number x, number y in [1, 3, 9], [1, 2, 3] if (x != 1)];
```

2.7 Working with Functions

In DotPar, functions are first-class, which means they can be assigned to variables or passed to other functions as arguments. A function provides a convenient way to encapsulate some computation, which can then be used without stressing its implementation. All functions are declared with the `func` reserved word, and can be nested within each other.

A function definition has this form:

```
func function-name : return-type(optional parameter declarations)
{
    declarations and statements;
    return return-val;
}
```

2.7.1 Revisiting main

With this in mind, let's take a look at `main` again. As in Java, `main` can accept arguments. However, as in C, the arguments can be omitted.

```
func main: void(char[][] args)
{
    if (len(args) > 0) {
      each(args, func: void(char[] s) { println(s); });
    }
    else {
      println(No arguments passed!);
    }
}
```

Try running this program, passing various arguments and check out the results.

2.7.2 Declaring an Array

Next, let's fill a create a matrix of integers. This example will demonstrate how to pass functions as parameters.

```
func main: void()
{
// This creates an array with 10 rows and 20 columns. The values are not populated,
// so they are random.
    number[10][20] b;
}
```

2.8 Idioms and Parallelism

Our next sample programs will contain more complex array manipulation and introduce some useful functions for arrays. These functions aren't common in imperative languages, and they exhibit some nice properties of DotPar.

2.8.1 Map

`map(array, function)` calls `function(item)` for each of the array's items and returns an array of the return values. For example, let's compute some cubes:

```
func main:void(){
    func cube:number (number value, number index){
        return value*value*value;
    }
    number[] a = [1,2,3,4,5];
    map(a, cube);
}
```

Note that this example could have been computed more succinctly with a list comprehension. However, in some cases, you may find a `map(array, function)` to be clearer. When you run map it returns a transformed list. Whereas when you run each it performs the functions and returns nothing, instead usings the

2.8.2 Reduce

`reduce(array, function, original)` returns a single value constructed by applying the reducing function repeatedly to the reduced value so far and each element of the array. For instance, if you wanted to add all the elements of a numerical array, one could define the function `sum(number a, number b)`, and apply `sum` to the array.

To break things down, we take the first two elements of the array and apply `sum` to them. This returns the sum of the elements, our first result. Then, we take the third element of the array, and apply `sum` to our first result and the third element, obtaining our second result. We continue in this manner, until we have one result and no more elements to reduce.

For a concrete example, let's compute the sum of the numbers 1 to 10000:

```
func main:void(){
   func sum:number (number a, number b){
       return a+b;
   }
   number[] x;
   fill(x, func:number(number index) { return index+1; }, 10000);
   // 0 is the starting value, we do this so that we can return 0
   //if the array is  empty
   //Here we are filling the array with the values 0 - 10000 for
   //the use in the suming function
   println(reduce(x, sum, 0));
}
```

2.8.3 Matrix Multiplication

In this example we introduce nested functions and explore how working with functions can create clearer and better code. We'll also take a look at the feature that gives DotPar its name: implicit parallelism.

```
func matrixMultiply:number[][](number[][] a, number[][] b) {
    func transpose: number[][](number[][] a) {
        number[len(a[0])][len(a)] transposed;
        number i;
        number j;
        for (i<0; i < len(a); i = i + 1){
            for (j<0; j < len(a[0]); j = j+ 1){
                transposed[j][i] = a[i][j];
            }
        }
        return transposed;
    }

    func sum:number(number a, number b) {
          return a + b;
    }

    func dotprod:number(number[] a, number[] b) {
          return reduce([x * y for number x, number y in a, b], sum );
    }

    if (len(a) != len(b[0])) {
          return nil; // one can always return nil in place of a real value
    }

    return [[dotprod(row, col) for number[] col in transpose(b)] for number[] row in a];
```

```
}
```

The function `transpose` returns a transpose of the input matrix. The function `matrixMultiply` takes in arguments which are two two-dimensional arrays, `a` and `b`. The function `dotprod` accepts two one-dimensional arrays and calls the reduce function, which calls the sum function on each `x`, `y` pair which are elements in arrays `a` and `b`. The final line returns a statement that pairs every row in `a` with every column in `b` and calls the function `dotprod`.

2.8.4 Parallelization

These examples demonstrate the power of DotPar's implicit parallelization.

The `map` and `reduce` functions can be parallelized, given certain restrictions. If the functions only access variables passed to them and do not mutate outside state, the mappings and reductions can be automatically parallelized and executed on multiple cores. For instance, the computation done by `map` and `reduce` can be done in $O(n/m)$ time, where n is the size of the array and m the number of processors. In the transpose function from the previous section, each cell in the new matrix can be computed independently of the other, so DotPar will parallelize this multiplication if it is appropriate given the computer's architecture and the nature of the computation.

3 Conclusion

In this tutorial we have covered compiling your programs and writing basic assignment and logical statements all the way through to more complicated features like map, reduce, list comprehensions, and nested functions. With all of the these tools now available, you should now be able to unleash the performance power that DotPar provides for data manipulation.