1 DESCRIPTION

Dice is a distributed systems language which optimizes the execution of data intensive algorithms by distributing work over multiple systems. Dice simplifies the distribution process by allowing users to simply specify a block of code and the data on which to execute, and the Dice compiler handles parsing, distributing, and re-collecting the data from the code, which is passed back to the user. This model allows for implementation of more complex structures as well, such as embedding multiple blocks of code, each of which iterates on the data passed back by the function or by only distributing until a certain condition is met. Our target users have access to multiple servers on which they wish to perform data-intensive computing tasks in parallel; they are already familiar with the principles of parallel programming and prefer a tool that is more lightweight than Hadoop is.

2 MOTIVATION

Allowing a programmer to design one’s application such that it could be distributed amongst multiple workers is enticing in an age of increasing data velocity and of embarrassingly parallel computing tasks that can leverage commodity and cloud computing. It is even more relevant when one considers the potential ramifications of utilizing wasted computation time on idle phones, computers, and servers.

The most established systems for distributed computing are Google's implementation of the MapReduce programming paradigm and Hadoop, an open source implementation. Both systems are designed for low latency by using data locality; they both apply functions at the location in which the data is stored rather than send the data over the network. To enable this, both systems rely heavily on a distributed file system (Google's on GFS and Hadoop on HDFS). Implementing a distributed file system is beyond the scope of this course, which emphasizes programming language and compiler design. For this reason, we eschew data locality optimization in our distribution protocol. Dice will distribute both data and instructions to workers via a proprietary data transfer protocol. It will support data ingestion from the local filesystem and potentially MongoDB, a widely used web scale database. MongoDB provides a database operation that runs map-reduce on computations on results of MongoDB-distributed queries. We aim to address the main shortcoming of its built-in map-reduce utility by allowing flexible numbers of workers despite limited sources of data.

We will dice up the code for you!

Dice draws inspiration from pubCrawl (Fall 2013 PLT language). Dice will iterate on pubCrawl compiling designated portions of the source code into LLVM bytecode which will then be distributed to various workers. LLVM’s versatility will allow the programmer to harness machines powered with...
different CPU ISAs, such as Acorn RISC Machine (ARM) and Intel/AMD’s x86-64, which collectively power a majority of phones and computers, while ignoring the underlying implementation on each CPU architecture.

3 SUMMARY OF GOALS

- Create a simple, intuitive language that allows a developer to write code to be distributed amongst several machines simultaneously
- Design the language such that the developer does not need to know how worker machines are monitored or how the data and code are distributed
- Use our language to implement data parallel algorithms (such as distributed merge sort) and brute force hashing applications (such as bitcoin mining) more concisely than what is possible with C or Java
- Design and build an application that will accept incoming requests on worker machines that the host will communicate with
- Develop experience working with LLVM by compiling the distributed portions of Dice code to LLVM Bytecode / Intermediate Representation (IR)

4 DOMAIN FEATURES

- Syntax to designate portions of code to be distributed
- Flexible array structures that behave as lists or arrays, depending on optimization
- Easily access data from database or local text files and operate on them
- Ensure data is shared correctly amongst worker computers
- Use LLVM to allow the code to be cross-platform
- Add verification functions to allow the programmer to affirm workers are doing work properly

5 LANGUAGE DESIGN

Dice will allow for an easier distribution of code across multiple computers. We want to remove the responsibility of figuring out how to distribute information and code across multiple computers and instead let the coder simply define the functions that need to be distributed and provide some logic to how the code should be distributed. Initially, Dice will utilize programmer-provided IP addresses of the distributed machines to be used. Dice will also provide an optimized distribution algorithm for properly sharing code and information amongst multiple computers. All of the distributed code will be compiled to LLVM bytecode and then shared amongst the machines. LLVM bytecode will allow the programmer to utilize ARM-based processors on mobile phones in the distribution cluster.
6 Code Syntax

6.1 Primitive Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int, double, float, char, long</td>
<td>Typical primitive data types. Equivalent to C++ type structure</td>
</tr>
</tbody>
</table>

Array Designated by C style []. See below for Array related syntax and functions

<table>
<thead>
<tr>
<th>void</th>
<th>Refers to a function with no return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>bool is an enum with three potential values: true, false, null</td>
</tr>
<tr>
<td>null</td>
<td>A reference can be pointed to nothing via null</td>
</tr>
<tr>
<td>char[]</td>
<td>Strings in this language follow the same convention as C</td>
</tr>
</tbody>
</table>

6.2 Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
</table>
| distribute | Distributes a function or operation over some iterated input.  
Follows this syntax: 
[verified] distribute <function> over <input array> [until <bool condition>] [withrange <double value>] [into <new array>] | No |
| verified | Tells the compiler that the function should be run with the output of each result check against a programmer defined verify function | Yes |
| over     | Defines an array structure that the function must operate on for each element | No |
| until    | Defines a stop condition for the distribution call. This allows distribute to work as if it were a while-loop | Yes |
| into     | Defines a name for the new array that all data from the distribute call will be put into | Yes |
| withrange | Defines the size of buckets that will be inputted into the function. This will be a double value such as 0.1 to indicate each bucket will be roughly 10% of the original array and each function should operate on that bucket (see merge-sort example below) | Yes |
**range**
When the withrange option is used, range can specify to a specific section of an array. (see merge-sort example below) Yes

**index**
When withrange is omitted, this is used to indicate the current position in the array. When withrange is included, this indicates the current buckets in the array. Yes

**block**
Defines a function that will be distributed amongst multiple workers No

**verify**
Defines a function to check that a block has provided the correct output Yes

**main**
Defines a method that will be called when the program is run No

**(<type>)**
Typecasting is allowed in the language and it is specified using the same syntax as C or Java

**(<var>)**

**(* *)** Designates comments

### 6.3 ARRAY PRIMITIVES

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int[] a = new int[];</td>
<td>This is the syntax for defining an empty array. An array defined this way has flexible length. To get a rigid structure, pass an integer into the array constructor</td>
</tr>
<tr>
<td>a.append(&lt;item&gt;)</td>
<td>Appends an item to the end of the array</td>
</tr>
<tr>
<td>a.prepend(&lt;item&gt;)</td>
<td>Prepends an item to the beginning of the array</td>
</tr>
<tr>
<td>a.length</td>
<td>Provides the length of the array</td>
</tr>
<tr>
<td>a.map(&lt;function&gt;(params))</td>
<td>Calls a function with the associated parameters on each item of the array</td>
</tr>
<tr>
<td>a.intoBuckets(&lt;double&gt;)</td>
<td>Separates an array into discreate buckets each with a proportion equal to the double provided. The double is $0 &lt; double \leq 1$</td>
</tr>
<tr>
<td>a.pop</td>
<td>Returns the first element of the array and removes it</td>
</tr>
<tr>
<td>a.peek</td>
<td>Show the first element of the array</td>
</tr>
<tr>
<td>a.split(&lt;delimiter&gt;)</td>
<td>Splits the array into an array of array on some char[] delimiter</td>
</tr>
</tbody>
</table>
### 6.4 I/O

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print(&lt;char[]&gt;)</code></td>
<td>Prints a string to console. Can be a string literal.</td>
</tr>
<tr>
<td><code>read(&lt;char[]&gt;)</code></td>
<td>Reads in a file located at the passed parameter and returns a char[] with the data in that file</td>
</tr>
<tr>
<td><code>write(&lt;array&gt;, &lt;char[]&gt;)</code></td>
<td>Write the contents of an array into memory</td>
</tr>
</tbody>
</table>
7 Code Examples

7.1 Example Using Distributed IsPrime

```c
block bool isPrime(int val) {
    int p;
    if (!(val & 1) || val < 2) return val == 2;

    /* comparing p*p <= n can overflow */
    for (p = 3; p <= val/p; p += 2)
        if (!(val % p)) return 0;
    return 1;
}

verify bool isPrime(int val, int otherVal) {
    int temp = isPrime(val);
    if(temp == otherVal)
        return true;
    else
        return false;
}

void main() {
    char[] numbers = read('primelist.txt');
    char[] numberList = numbers.split('
');
    char[] newNumberList;

    verified distribute isPrime(val:numberList[index])
        over numberList (*until index >= numberList.length*)
        into newNumberList;

    newNumberList.map(print);
}
```
7.2 Example Using Distributed Hello World

```java
block void HelloWorld(int val) {
    print("hello world\n");
}

void main() {
    int[] arr = new int[300];

    // This will print out
    distribute HelloWorld over arr;
}
```
7.3 **Example Using Distributed Merge Sort**

```java
block void merge (int[] a, int n, int m) {
    int i, j, k;
    int x[] = new int[n];
    for (i = 0, j = m, k = 0; k < n; k++) {
        int val = 0;
        if(j == n)
            val = a[i++];
        else if( i == m)
            val = a[j++];
        else if(a[j] < a[i])
            val = a[j++];
        else
            val = a[i++];
        x[k] = val;
    }
    for (i = 0; i < n; i++) {
        a[i] = x[i];
    }
}

block void merge_sort (int[] a, n) {
    if (n < 2)
        return;
    int m = a.length / 2;
    merge_sort(a, m);
    merge_sort(a + m, n - m);
    merge(a, n, m);
    return a;
}

void merge(int[][] a) {
    int[] final = new int[];
    bool hasMore = true;
    while(hasMore) {
        hasMore = false;
        int minVal = null;
        for(int i = 0; i < a.length; i++) {
            if(a[i].length > 0) {
                hasMore = true;
                if(minVal == null or a[i].peek() < minVal) {
                    minVal = a[i];
                    a[i].pop();
                }
            }
        }
        final.append(minVal);
    }
}

void main () {
    int a[] = {4, 65, 2, -31, 0, 99, 2, 83, 782, 1};

distribute merge_sort(a[range], a[range].length)
    over a
    withrange 0.1
    into partially_sorted;

da.intoBuckets(0.1).map(merge).map(print);

    return 0;
}
```