Hahn Chong - hc2361
Fred Clark Jr. - fc2413
Rotem David - rd2499
Robert Tolda - rmt2131
Jose Rodriguez - jgr2128 - Team Leader

Proposal - PB & J
(Parallel Boxes and Jam)

1 Introduction

Distributed computing is the use of multiple autonomous computers which work in concert to achieve a desired output.

PB&J is designed for developers to distribute a job amongst multiple computers/processors in a minimal amount of code needed while still being readable.

In PB&J, developers define a program which is run on both the master and slave servers. While a portion of the program is only run on the master, the slave waits for messages from the master as to what functions to run on what data sets.

It's the responsibility of the master server to distribute the job to the slave servers. A developer making a program in PB&J can use the spread & jam statements to perform these actions.

The final application would be run in a fashion similar to:
```
master-box$ programName -m (m for master)
 slave-box$ programName 192.0.0.1 (ip address of master)
```

2 Language Description

2.1 Types

2.1.1 Primitive types
PB&J supports the following primitive types:

long	Basic numeric type.
double	Floating point value and is declared by adding a decimal point to an integer.
boolean	True or false expression.
string	Sequence of characters.

2.1.2 Collection types
PB&J also supports more complex data types, known as collection types, that follow the syntactical structure of JSON.

Array	A fixed position of sequential data types.
Map	A key/value based data structure. Where all keys are of the same type but none are duplicated.

2.2 Lexical Structure

2.2.1 Comments

In PB&J there are only single line comments. Comments are generated one line at a time, by placing three consecutive periods (...).

Example:
…This is a comment in PB&J.
… This is also a comment in PB&J.
.. . This is not a valid comment in PB&J

2.2.2 Operators

Yeah we got some of those.

PB&J operator (@):
Spread: @ is used to identify an explicit argument to spread amongst the slave servers with
Jam: @ is used to identify the parameter in which to deliver the spread result.

Assignment operators : *identifier* <- *expr*
The assignment operator consists of <- to represent you are injecting the left identifier to the right expression.

Comparison operators
We're dropping the == operator as to not create confusion.

=	Structural comparison
===	Physical comparison

2.2.2.5 String concatenation

The ~ (tilde) is used to concatenate data types into a string.

Examples:

Expression	*Evaulation*
4 ~ 2	“42”
“Hello “ ~ 0.0	“Hello 0.0”

2.2.5 Identifiers

An identifier may be created using any combination of digits, and upper or lowercase ASCII characters. An identifier may not be one of the reserved keywords. Two identifiers are if same, if and only if, they have identical unicode characters for each letter of digit.

2.2.6 Keywords
Two keywords, formed from ASCII characters, are reserved and can not be used as identifiers.

jam spread for

2.3 Blocks and Statements

2.3.1 Blocks
Blocks for methods and statements are defined using braces { }

2.3.2 spread Statement
spread: func
The spread statement is used to distributed a collection type amongst registered slave machines. The function acts as a callback returning each time a slaving reports results, but only blocks the call for the first result.

Parameters
func: A reference to a function that runs on the spread collection.

2.3.3
jam: func
The jam statement is used to block on a spread statement until each slave reports a result. It'll then return the results of the spread or the result of the given function.

Parameters
func: (optional) a function to run on the resulting collection set.

3 Examples of syntax

3.1 Example 1
Add all integers in a given array.

```
... Print sum of longs in an Array.
master() {
      ... Inside context of Master
      var myList <- [1, 2, 3, 4, 5, 6];

      ... Spread splits a collection type equally
      ... across already registered slaves
      ... join is a blocking call to wait on
      ... results from the spread

      var result <- jam: add(@) spread: add(@myList);

      ... Print result.
      print("Result " ~ result ~ "\n");
}

add(myList) {
      ... Anonymous inner function
      var result <- 0;
      for(var value <- #values) {
            result <- result + value;
      }
      result ->
}
```

3.2 Example 2
Nieve prime factorization

```
master(bigPrime){ ... Runtime argument.
      var searchStarts <- [#slaves.length];
      var iterations <- bigPrime / 3 / #slaves.length;

      for(var m <- 0; m < #slaves.length; m <- m + 1) {
            searchStarts[i] <- iterations * m;
      }

      var result <- spread: factor(@searchStarts, bigPrime, iterations);
            ...lack of join means that first non-null
            ...answer is taken as the answer or stops when all return.
      print "Result: " ~ result;
}

factor(mylist, prime, iterations) {
      var start <- mylist[0];
      for( i <- start. i < iterations + start. i <- i + 1) {
            if(prime % i = 0 ) { ...it is not prime
                  i ->
            }
      }
      null -> ...no evidence that it is not prime
}
```