|  |
| --- |
| University of North florida |
| Go & Scala Sort Tools |
| Application Documentation |
|  |
| **Thomas Trask** |
| **9/10/2013** |

|  |
| --- |
| This document covers the program flow and execution of the 6 applications generated to study the multithreading performance of the GO and Scala programming languages. |

Contents

[Project Overview 4](#_Toc367452496)

[Motivation 4](#_Toc367452497)

[Overview 4](#_Toc367452498)

[Environments 4](#_Toc367452499)

[Programs 5](#_Toc367452500)

[Program Flow – Client/Server 5](#_Toc367452501)

[Program Flow – Multicore 6](#_Toc367452502)

[Folder Structure 6](#_Toc367452503)

[Program Compilation 7](#_Toc367452504)

[Go 7](#_Toc367452505)

[Scala 7](#_Toc367452506)

[Pre-Execution Requirements 7](#_Toc367452507)

[Go 7](#_Toc367452508)

[Scala 7](#_Toc367452509)

[Program Input Parameters 7](#_Toc367452510)

[Multicore 7](#_Toc367452511)

[Client 7](#_Toc367452512)

[Server 8](#_Toc367452513)

[Stock 8](#_Toc367452514)

[Program Execution 8](#_Toc367452515)

[Multicore 8](#_Toc367452516)

[Go 8](#_Toc367452517)

[Scala 8](#_Toc367452518)

[Server 8](#_Toc367452519)

[Go 9](#_Toc367452520)

[Scala 9](#_Toc367452521)

[Client 9](#_Toc367452522)

[Go 9](#_Toc367452523)

[Scala 9](#_Toc367452524)

[Stock 9](#_Toc367452525)

[Go 9](#_Toc367452526)

[Scala 9](#_Toc367452527)

[Time Metric Output 10](#_Toc367452528)

[Format 10](#_Toc367452529)

[Example 10](#_Toc367452530)

Project Overview

# Motivation

As new, popular, and interesting programming languages emerge out of the dust heap that is theoretical language design, they attract the attention of programmers, both nascent and seasoned. Programmers, always focused on both the speed and efficiency of their new darling language, often wonder how it compares to current languages, such as C or Java. It is with this motivation that this project was undertaken. Multi-threading performance of the GO and SCALA programming languages will be compared analytically, using the non-trivial task of sorting a large data set.

# Overview

As stated above, the purpose of this project is to compare the multithreading performance of both Scala and Go on 2 disparate systems; each language will be tested on a 64-core Beowulf Cluster, along with a 14-machine client/server system by creating custom applications in each language that will

1. generate a large set of data
2. Split the data into n number of servers/threads available to the application
3. Send each smaller dataset to their respective server/thread for processing.
4. Once each data set is processed, it is sent back to the host process/machine for recombination.
5. Once all data has been sent back to the host thread/machine, it recombines the data.
6. Time metrics will be measured for each application’s sorting and recombination of the large data set mentioned above.

It will then be analyzed to determine the following:

* How individual language performance differs based on how many servers/cpu cores are used
* How individual language performance differs based on how large the data set is.
* How comparative language performance differs when the data set size and resources are equal.

# Environments

As stated above, GO and SCALA applications will be developed for the following environments:

* A 64-core SMP Computer
  + The computer is to be accessed via SSH at the address cisatlas.ccec.unf.edu
* A 14-machine Beowulf Cluster
  + The client machine is to be accessed via SSH at the address Uranus.ccec.unf.edu
  + From the client machine, the 13 servers may be accessed via SSH using the following command:
    - ssh compute-0-<machine number>

Programs

Applications will be created to run against both a 64-core SMP computer and a 14-computer Beowulf cluster. Applications will have as close to the following control flow as their respective languages allow.

# The 4 Programs

The four programs that will be outlined in this program include the list below. All four applications will need to be written in both languages in order to compare data.

### Multicore

This application is designed to run on the ATLAS SMP machine and leverages the multitude of CPU cores that it has to generate and sort a large data set.

### Server

This application is designed to run on the URANUS ‘compute-‘ nodes and is used to generate & sort a subset of data on each node. This is to be run in conjunction with the Client application

### Client

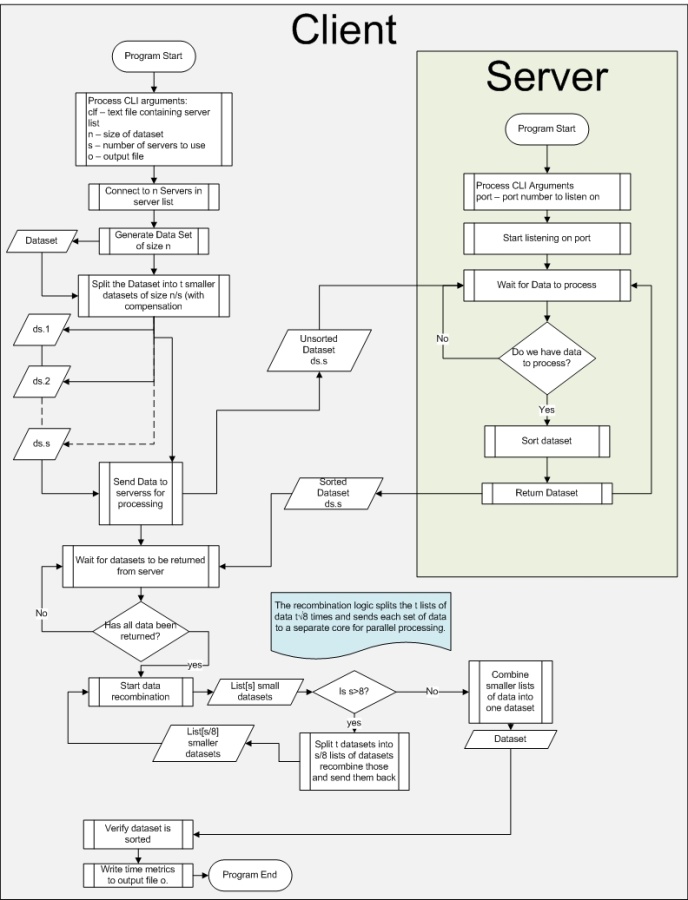
This application is designed to run on the URANUS root node and will be used to field requests by the user to generate a dataset of size N and send it off to a subset of ‘compute-‘ nodes on the URANUS server. This is to be run in conjunction with the Server application.

### Stock

This application is used as a “control” performance benchmark to determine how leveraging multithreading/cluster computing affects the performance of sorting. It is designed to work in any environment.

* For the GO stock application, the sort.Ints() function will be used.
* For the SCALA stock application the scala.util.Sorting.quickSort() function will be used.

# Program Flow – Cluster



# Program Flow – Multicore

## C:\Users\lc503694\Documents\Multicore.jpg

# Folder Structure

The programs for the 2 languages are split into their respective folders, Go-MulitProc and Scala-Multiproc. Each language’s main folder contains both the /src & /bin folders. If the /bin folder is not present in the Scala application, it means that the application still needs to be compiled. The /src folder contains the following folders:

* Client – Contains code to run the Go/Scala Client application
* Server – Contains code to run the Go/Scala Server application
* Multicore – Contains code to run the Go/Scala Multicore application.
* Stock – Contains code to run the Go/Scala “Stock” applications on both the client/server & multicore systems.

# Program Compilation

## Go

A .go file can be compiled and run using the following syntax:

* go build somefile.go
* ./somefile

Alternatively, any .go file can be executed without directly compiling it by using the following syntax:

* go run somefile.go

## Scala

Before execution, the Scala applications need to be compiled. The easiest way to compile the SCALA application set is to run the ‘build.sh’ build script located in Scala’s root directory. You can do this by using the following command:

* <root directory>/Scala-Multiproc/build.sh

To compile the Scala applications manually, the following command can be run from the command line while Scala’s root directory:

* scalac -language:implicitConversions -feature -deprecation src/multicore/\*.scala src/client/\*.scala src/math/\*.scala src/optional/\*.scala src/rpc/\*.scala src/server/\*.scala src/utility/\*.scala src/stock/\*.scala -d bin

# Pre-Execution Requirements

Before any applications are executed, the following commands must be issued

## Go

* export GOPATH=<root directory>/Go-Mulitproc/

## Scala

* export CLASSPATH=<root directory>/Scala-Multiproc/bin/

# Program Input Parameters

For each environment, the input parameters for each application are identical.

## Multicore

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | Definition | Requirement | Default Value | Required |
| n | Dataset size | Int32 | 100 | No |
| t | Number of threads to use during process | Int32 | Total number of threads available | No |
| o | Output file | String | Go: out.go.multicore.csv  Scala: out.scala.multicore.csv | No |

## Client

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | Definition | Requirement | Default Value | Required |
| n | Dataset size | Int32 | 10000 | No |
| clf | List of servers for processing | String | clients.txt | No |
| S | Number of servers to use during process | Int32 | Total number of servers available via server file | No |
| o | Output file | String | Go: out.go.client.csv  Scala: out.scala.server.csv | No |

## Server

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | Definition | Requirement | Default Value | Required |
| port | Port number that the server will listen on | Int32  Go: “<value>”  Scala:”:<value>” | Go: “9000”  Scala: “:1602” | No |
| name | Scala Only – Unique name of the server running on the system. NOTE: This name must be unique within client/server system. For example, using 2 servers, you can name one server ‘server1 and the other server ‘server2. | String:  ‘<value> |  | Yes |

## Stock

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Parameter | Definition | Requirement | Default Value | Required |
| n | Dataset size | Int32 | 100000000 | No |
| o | Output file | String | Go: out.go.stock.csv  Scala: out.scala.stock.csv | No |

# Program Execution

The following commands can be executed from the linux command line to run each application.

## Multicore

### Go

The Go application can be executed using the following command

* go run <root directory/Go-Multiproc/src/Multicore/mc.go [-t <threadcount> -n <dataset size> -o <output file>]
  + *go run ~/Go-Multiproc/src/Multicore/mc.go -t 1 -n 20000 –o out.csv*

### Scala

The Scala application can be executed using the following command

* scala [-J-Xmx4g] multicore.Multicore [-t <threadcount> -n <dataset size> -o <output file>]
  + *scala multicore.Multicore –t 64 –n 10000 –o out.scala.csv*

|  |
| --- |
| The optional parameter [-J-Xmx4g] is to set the maximum Java Virtual machine available memory limit to 4gb. This is required for datasets > 10 million. |

## Server

|  |
| --- |
| **Note: All Servers that you intend to execute against should be started before executing the client. They will run until manually closed by the user.** |

### Go

The Go Server application can be started using the following command.

* go run <root directory/Go-Multiproc/src/server/server.go [-p <port>]
  + *go run ~/Go-Multiproc/src/server/server.go –port 9000*

### Scala

The Scala server application can be started using the following command:

* scala server.ServerShell –name <server name> [-p :<portNumber>]
  + *scala server.ServerShell –name ‘server1 –port :1602*

## Client

### Go

The Go client application can be started using the following command:

* go run <root directory/Go-Multiproc/src/client/client.go [-s <server count> -n <dataset size> -clf <server list> -o <output file name>
  + *go run ~/MP/Go-Multiproc/src/Client/client.go -s 1 -n 200 -clf clients.txt*

### Scala

The Scala client application can be started using the following command:

* scala [-J-Xmx4g] client.ClientShell [-s <server count> -n <dataset size> -clf <server list> -o <output file name>
  + *scala client.ClientShell –s 2 –n 50000*

|  |
| --- |
| The optional parameter [-J-Xmx4g] is to set the maximum Java Virtual machine available memory limit to 4gb. This is required for datasets > 10 million. |

## Stock

### Go

The Go stock application can be started using the following command:

* go run <root directory/Go-Multiproc/src/stock/stock.go [-n <dataset size> -o <output file>]
  + *go run ~/Go-Multiproc/src/stock/stock.go –n 1000 –o out.go.csv*

### Scala

The Scala stock application can be started using the following command:

* scala stock.Stock [-n <dataset size> -o <output file>]
  + scala stock.Stock –n 1000 –o out.scala.csv

# Time Metric Output

## Format

Time metrics will be output in the following comma-separated format and be saved to the application’s respective output file:

|  |  |
| --- | --- |
| Language | Format |
| Go (Non-Stock) | (Server/ThreadCount), Dataset Size,SortTime(ns), CompileTime(ns) |
| Go (Stock) | Dataset Size, Total Time (ns) |
| Scala (Non-Stock) | Execution DateTime, (Server/ThreadCount), Dataset Size, SortTime(ms), CompileTime (ms), CompileTime (ms) |
| Scala (Stock) | Execution DateTime, Dataset Size, Total Time(ms) |

## Example

Below is an example of what to expect from a Scala non-stock application’s output:



Appendix

# Server List

Each Client application reads from its server list from a file (default: clients.txt). The following are examples of how these files should look

## Go

### Format

For each server, the format should be the following:

* <IP>:<Port>

### Example

127.0.0.1:9000  
127.0.0.2:9000  
127.0.0.3:9000

## Scala

### Format

For each server, the format should be the following:

* <ClientName> <IP> <Port>

### Example

Client1 127.0.0.1 9000  
Client2 127.0.0.2 9000  
Client3 127.0.0.3 9000