|  |
| --- |
| University of North florida |
| Multiprocessing |
| GO vs SCALA |
|  |
| **Thomas Trask** |
| **9/10/2013** |

|  |
| --- |
| An overview of the performance differences between the GO and SCALA programming languages when processing large datasets using multiprocess/multicomputer systems. |

Contents

[1. Project Overview 4](#_Toc374273040)

[1.1. Motivation 4](#_Toc374273041)

[1.1. Background 4](#_Toc374273042)

[1.2. Multiprocessing 4](#_Toc374273043)

[1.1.1. Multiprocessing In Go 4](#_Toc374273044)

[1.1.2. Multiprocessing In Scala 4](#_Toc374273045)

[1.2. Sorting 4](#_Toc374273046)

[1.2.1. Sorting in Go 4](#_Toc374273047)

[1.1.1. Sorting in Scala 5](#_Toc374273048)

[1.3. Hypothesis 5](#_Toc374273049)

[1.3.1. Individual Application Performance 5](#_Toc374273050)

[1.1.1. Sorting vs Recombination 5](#_Toc374273051)

[1.1.2. Cluster Performance vs SMP Performance 5](#_Toc374273052)

[1.1.3. Go VS Scala 6](#_Toc374273053)

[2. Testing 6](#_Toc374273054)

[2.1. The 4 Programs 6](#_Toc374273055)

[2.1.1. Multicore 6](#_Toc374273056)

[2.1.2. Server 6](#_Toc374273057)

[2.1.3. Client 6](#_Toc374273058)

[2.1.4. Stock 6](#_Toc374273059)

[2.2. Environments 6](#_Toc374273060)

[2.3. Program Flow 6](#_Toc374273061)

[2.4. Testing Methodology 7](#_Toc374273062)

[2.5. Performance Measurement 7](#_Toc374273063)

[2.5.1. Sorting and Recombination 7](#_Toc374273064)

[3. Test Results 8](#_Toc374273065)

[3.1. A note on Test Results 8](#_Toc374273066)

[3.2. Individual Application Performance 8](#_Toc374273067)

[3.2.1. Go – SMP Machine 8](#_Toc374273068)

[3.2.2. Go – Beowulf Cluster 9](#_Toc374273069)

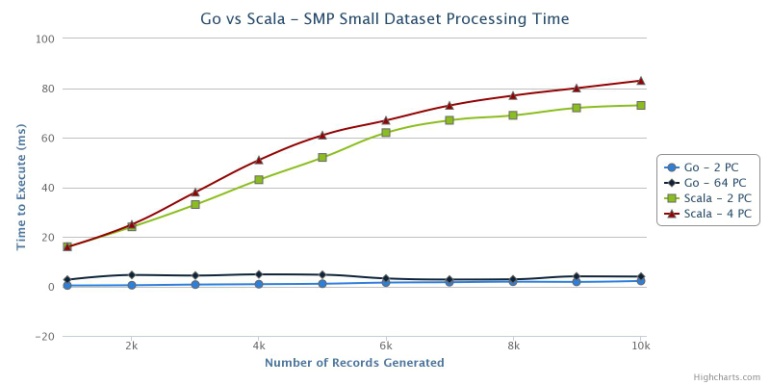
[3.2.3. Go – SMP vs Cluster 10](#_Toc374273070)

[3.2.4. Scala - SMP Machine 10](#_Toc374273071)

[3.2.5. Scala – Beowulf Cluster 11](#_Toc374273072)

[3.2.6. Scala - SMP vs Cluster 13](#_Toc374273073)

[3.3. GO vs SCALA 14](#_Toc374273074)

[3.3.1. SMP Machine 14](#_Toc374273075)

[3.3.2. Beowulf Cluster 14](#_Toc374273076)

[4. Conclusion 15](#_Toc374273077)

[4.1. Go 15](#_Toc374273078)

[5. Further Research 15](#_Toc374273079)

[6. References 15](#_Toc374273080)

[Appendix A: Detailed Program Flow 16](#_Toc374273081)

[Program Flow – Cluster 16](#_Toc374273082)

[Program Flow – Multicore 17](#_Toc374273083)

[Appendix B: Aggregate Data Results 18](#_Toc374273084)

[Appendix C: Ancillary Testing 18](#_Toc374273085)

[B1: General Language Performance 18](#_Toc374273086)

[B1.1: Number Generation 18](#_Toc374273087)

[B1.2: Function Calls 18](#_Toc374273088)

[C2: Cross-Resource Communication 19](#_Toc374273089)

[C3: Threaded Combination Dataset Size 19](#_Toc374273090)

# 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 was compared analytically, using the non-trivial task of sorting a very large data set.

## Background

## Multiprocessing

The idea of application multiprocessing involves a software application executing multiple code paths concurrently, with access to the same set of resources. This, in theory, will result in the application completing a desired task in less time than it would have if it had executed all code paths sequentially. In reality, this is not necessarily the case every time. As will be seen, implementing multiprocessing incurs a performance penalty due to the extra processing steps needed to implement it. For small datasets, this overhead can represents a significant portion of processing time, making an application perform slower than had it not leveraged multiprocessing. For larger datasets, the overhead often pales in comparison to the performance advantage to processing data concurrently.

### Multiprocessing In Go

#### Local Multiprocessing

Go’s approach to multithreading revolves around **goroutines**, Go functions that, at runtime, “[executes] concurrently in the same address space as other goroutines. A Go application consists of one or more Goroutines.”

For all intents, when executed, a goroutine is a tiny application that communicates with a host application. When a Go application (a host goroutine) calls a child goroutine, the child goroutine’s processing is multiplexed onto a system thread and processed concurrently with the host goroutine. If available, the child goroutine will execute on a different CPU than the host CPU. The host and child goroutines can communicate with each other using **channels**, Go types that facilitates communication using a message-passing interface. Communication between goroutines can be both synchronous and asynchronous. When the child goroutine has finished executing, it sends an “exit” message back to its host application.

#### Non-Local Multiprocessing

Go applications can communicate across networks or other I/O connections through remote procedure call (RPC). In RPC, a “client” application initializes a connection to a “host” application by means of a TCP/UNIX/IP connection request. Communication is accomplished by serializing and deserializing functions and parameters that are registered on both the client and host applications. This approach is very similar to that of Java’s RPC interface. For this purpose, the Go client application will be the application that receives some dataset to sort, splits the dataset, sends it to a set of server applications for processing, and recombines the datasets when they are returned.

### Multiprocessing In Scala

#### Local Multiprocessing

Scala implements concurrency through the use of Actors and greatly resembles Go’s goroutine package. A Scala application can create an Actor object that lives within the application but executes on a separate process using a message-passing interface. A host Scala application can communicate with its Actor child processes by sending them messages in the form of Case Classes and Case Functions. The child Actor communicates with the host in the same manner; the host listens for Case Classes & functions for responses from the child Actor.

#### Non-Local Multiprocessing

For non-local multiprocessing, Scala leverages the RemoteActor model. As its name implies, RemoteActor uses much of the same syntax and communication ideas that Scala’s Actor class does. The main difference is that, when instantiated, the RemoteActor requires a remote address that points to a Scala application capable of communicating with the application. RemoteActor uses RPC functionality, similar to that of Go.

## Sorting

### Sorting in Go

As outlined by its documentation, Go uses a generic quicksort algorithm “makes one call to data.Len to determine [the length of the dataset to be sorted], and O(n\*log(n)) calls to data.Less [the comparison function] and data.Swap.” As implied, Go uses a single processing thread to sort a dataset.

### Sorting in Scala

As discussed above, Scala implements a functional programming style. Scala’s sorting algorithm consists of one tail-recursive function that sorts a dataset in O(n\*log(n)) time. Scala’s compiler is specifically designed to optimize tail-recursive functions; when compiled, the sorting algorithm is compiled into an iterative loop that traverses an input dataset, sorting it in-place.

## Hypothesis

### Individual Application Performance

In the target applications, leveraging multiprocessing involves additional steps beyond those leveraged by sequential processing:

* The application must allocate the resources needed for multiprocessing (GO goroutines/scala actors)
* The application must determine which additional resources are available.
* The application then must split its input data across the multiple resources.
* The application then must send the data to each individual resource and wait for it to be returned.
* Finally, the application must process the returned data and regroup it back into one contiguous dataset.

With these added processing complexities taken into consideration, it is believed that applications using multiprocessing execution that sort a small amount of data should execute either as fast or slower than those that do not use multiprocessing. It is believed that, as both the number of resources as well as the size of the dataset to process increases, the processing overhead will be dwarfed by the processing benefits gained through leveraging multiple resources. Furthermore, if this proves to be true, it is believed that for each combination of resources and dataset sizes, there is a point of equilibrium where the applications that leverage multiprocessing will perform equivalently to the applications that do not.

For the experiment, the researchers were able to specify any number of resources to use. For the SMP machine, it is hypothesized that the application will see no performance increase once the number of resources requested by the application exceeds the number of resources (CPU cores) available to the system.

As will be discussed later, two of the performance criteria being measured will be how quickly each individual block of data is sorted and how quickly each application is able to recombine these sorted blocks into one contiguous, sorted data set. The number of smaller datasets to recombine is equal to that of the number of resources that are being used; i.e. if 64 CPUs are used on the SMP machine for a test, there will be 64 datasets to recombine after they’re all sorted. Therefore, it is hypothesized that, as the number of resources increases, the time it takes to recombine the smaller datasets into one larger dataset will increase. It remains to be seen whether this increase in processing time will offset the decrease in processing time gained by leveraging more resources.

### Sorting vs Recombination

As outlined above, the two most-resource intensive portions of the application’s process are when individual datasets are sorted and when these datasets are recombined. As stated earlier it is hypothesized that, given a large enough dataset, sorting will be positively impacted with the addition of more resources. However, the number of resources leveraged during the recombination process is essentially static, whereas the number of datasets to process increases with the number of resources used to sort the data. Because of this, it is hypothesized that, as the number of resources used in an application increases, the time it takes to recombine the set of sorted datasets will increase. It is possible that this decrease in performance will cancel out those improvements gained by sorting large datasets with more resources.

### Cluster Performance vs SMP Performance

One of the defining attributes of an SMP machine is that all of its processing units are connected and share I/O and memory resources via a physical system bus running at 6.0GB/s. Conversely, one of the defining attributes of a Beowulf cluster is that its resources communicate across a high-speed network bus running at 1.0Gb/s. Because of the significant speed difference between the two busses, it is hypothesized that, when using the same number of resources, the SMP machine will outperform the cluster.

#### A note on computational equivalence

It is important to note that the Beowulf cluster’s applications will use the maximum number of CPUs available to each cluster machine (4 CPUs). In order to equitably compare the 2 systems, the number of “effective” CPUs will be used. For instance, if we use 2 client machines in the Beowulf cluster, this should be considered equitable to the SMP machine using 8 CPU resources, not 2.

### Go VS Scala

As discussed previously, GO applications are compiled pre-execution. Conversely, Scala applications are compiled as they are executed using the Java JIT compiler. Go applications are able to pass parameters to functions byref, meaning that they can pass a dataset’s memory address to a function, without having to create a new instance of the dataset. On the other hand, Scala applications must pass function parameters by-value, meaning that a copy of an input parameter must be created and passed to each function that uses it. See Appendix B1.2 for more information regarding the speed difference between Go and Scala when calling functions and passing parameters to them. The Go programming language is a statically-typed, imperative language. When a variable is created in Go, such as an 32-bit integer, the runtime only needs to allocate the 32-bit memory space for the integer. Scala is an object-oriented programming language; all variable types derive from the Scala/Java “Object” class. This means that, when a 32-bit integer is created, the runtime must create the integer variable as well as generate all resources associated with the Object class. See Appendix B1.1 for more information regarding the speed difference between Go and Scala when generating integers.

Because of all of the language differences outlined above it is hypothesized that, in all instances, the GO applications will outperform their Scala counterparts.

# Testing

## The 4 Programs

Four applications were created to run against both a 64-core SMP computer and a 14-computer Beowulf cluster owned by UNF. All four applications outlined below were written in both Scala and Go.

### Multicore

This application is designed to run on the ATLAS SMP machine and leverages the multitude of CPU cores available 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.

## Environments

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
  + <Add technical information about cluster>
* 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>
  + <Add technical information about cluster>

## Program Flow

As outlined below, the main gist of each application contains 7 parts:

1. Generate a list of data
2. Split the data into small pieces
3. Send it to resources to sort
4. Get the data back from said resources
5. Recombine the data
6. Verify that the data is sorted and log any relevant performance measures.

More detailed information on the program flow of each separate application is located in Appendix A.

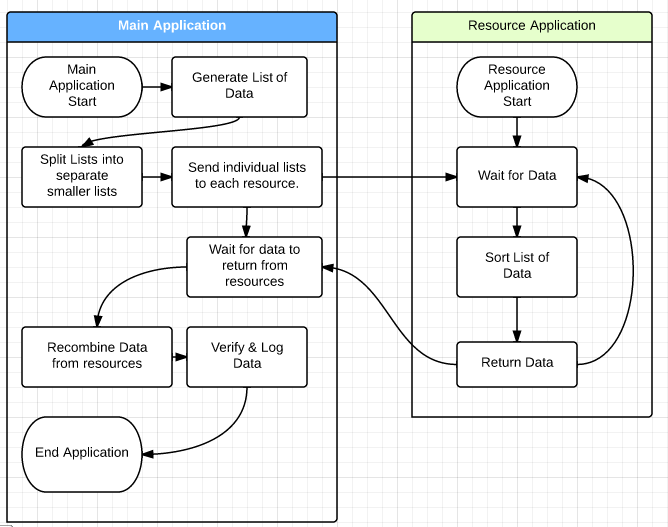


Figure Process flow for Threaded Sorting

## Testing Methodology

Each application will be executed using the following scheme:

* ~$ <application name> <number of threads/servers> <number of ints to sort> [<extraneous inputs>]
  + Application Name - The name of the application being executed.
  + Number of threads/server:
    - For the SMP machine, this includes using {2, 4,8,16,32,64,128,256} threads.
    - For the Cluster, this includes using {2,4,6,8,10,12} servers.
  + Number of Ints to sort - This includes a range of numbers ranging from 10 thousand(k) to 100 million(m) in the following scheme: 1x, 2x, 3x… 10x where x is some power of 10 ranging from 10^4 (10k) to 10^8 (100m). For instance, the first range of numbers would include {10k, 20k, 30k… 100k}; the second range of numbers would include {100k, 200k… 1m).

Each scheme will be executed ~100 times, to provide a large enough data set to accurately map performance of each application.

## Performance Measurement

For each application’s execution, the **total time** to execute the sorting and recombination of the data requested was measured, as well as the individual **sorting** **time** and **recombination time**. The data was then imported into a relational database for analysis. The following aggregate time measurements were calculated for each permutation: median, mean, mean average, Q1/Q3 mean, minimum/maximum.

### Sorting and Recombination

As stated above, two of the performance criteria measured for each application will be the time it takes to sort each individual dataset as well as the time it takes to recombine all of the datasets. Both applications use the QuickSort sorting method to sort each dataset.

Traditionally, dataset recombination would be accomplished by iterating through each sorted set of data, determining which dataset currently had the lowest value, and copying that value to the result dataset. This method provides the application with a O(n^2) performance metric; as the number of datasets increases, the time it takes to recombine them increases exponentially. During testing, this method disproportionately impacted SMP applications, as they generate a much larger data set than Cluster applications. A better, faster way of recombining the datasets was developed. It uses the following method: For numbers of datasets greater than some number, split the datasets into two smaller sets of data and recombine those, in parallel. Over a set of ⌈⌉ iterations, where r is the number of resources used and n is an arbitrary number used to split the datasets. For testing, n=4 was used Further testing should be done to determine if this is the most useful value to use when splitting the number of datasets. Further research should be conducted to determine the most advantageous value for n. The following figure illustrates this method:

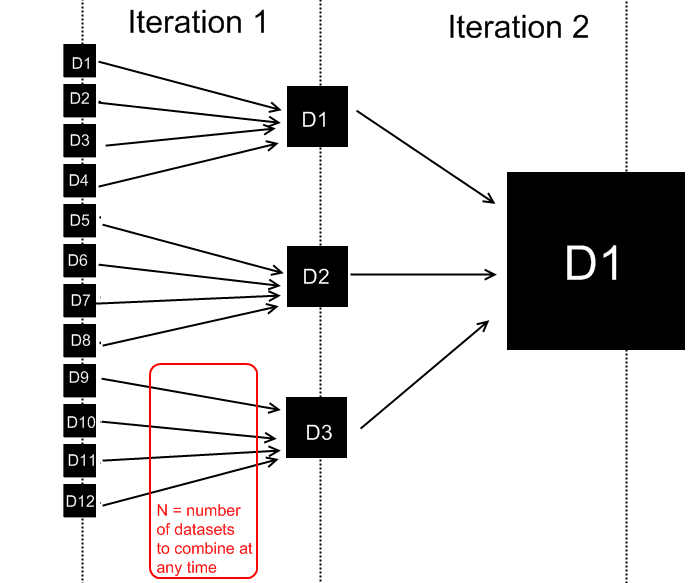


Figure Fork-Join Data Recombination

This method of recombination improved the recombination time considerably; the application’s comparison times performed at O(n log(n)) where n is the number of datasets used. However, it did increase the number of resources used during recombination.

# Test Results

## A note on Test Results

In general, each test results represented below represent the mean average of over 100 data points for that particular configuration. Each data set’s mean average was calculated by calculating the average of the “middle half” of the data point set: those values between the Q1 and Q3 percentile of the set. This method of assessment was chosen to minimize irregular application performance skewing average data.

## Individual Application Performance

### Go – SMP Machine

#### Threading Comparison

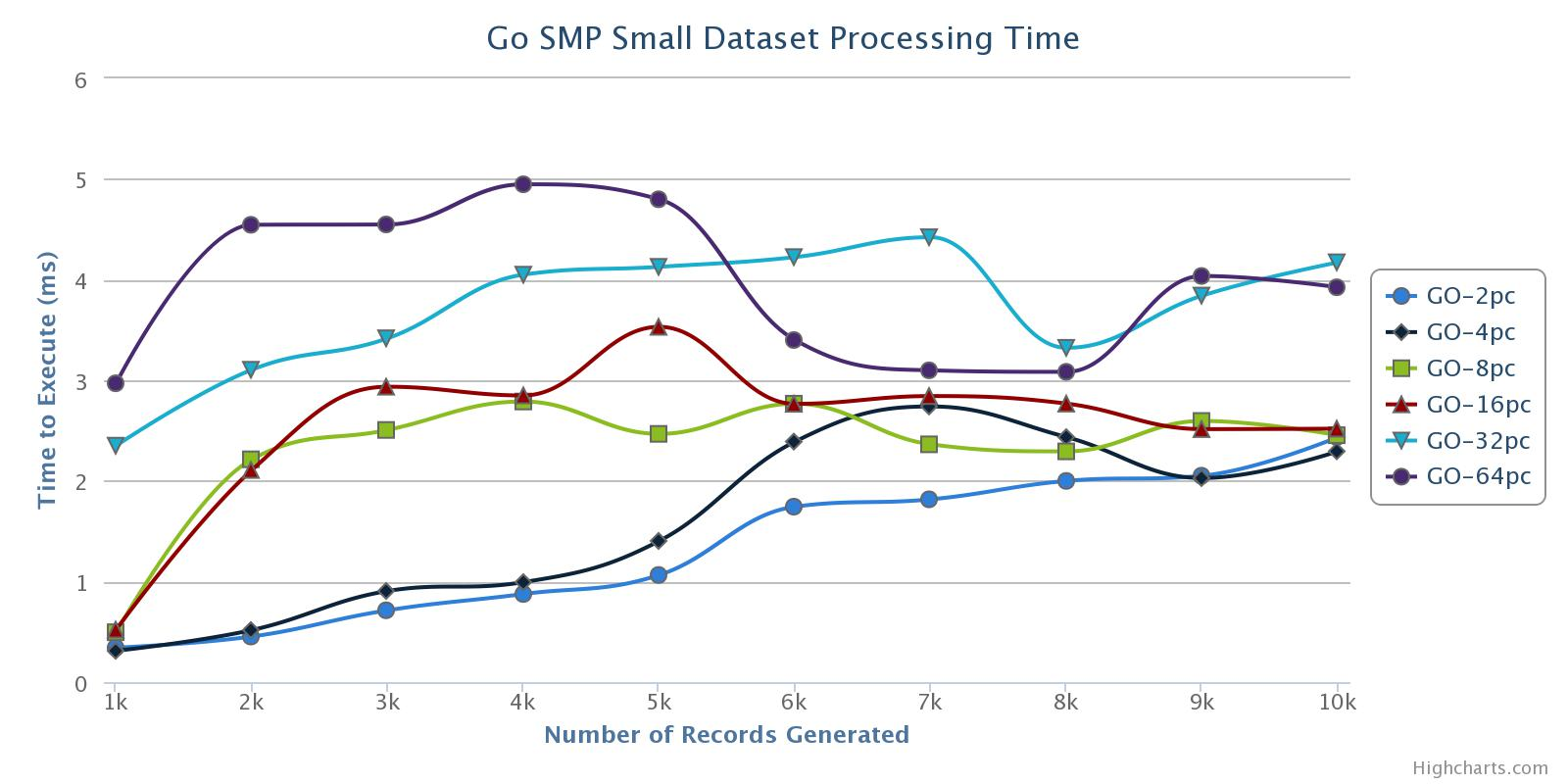
When dealing with smaller datasets, using a greater number of resources may not be advantageous. Figure 1 shows that initially, applications that leverage fewer resources (“low-resource applications”) outperformed those that leverage a greater number of resources (“higher-resourced applications”), sometimes by a significant margin. It is important to note that all applications returned a sorted data set in less than 5ms, regardless of how many resources were allocated. It can be seen, however, that as the dataset size increases, the performance gap is closing between all applications.

Figure : Go SMP Performance - Small Dataset Size

As the Dataset size increases, the performance gap between low-resource applications and high-resource application begins to close.

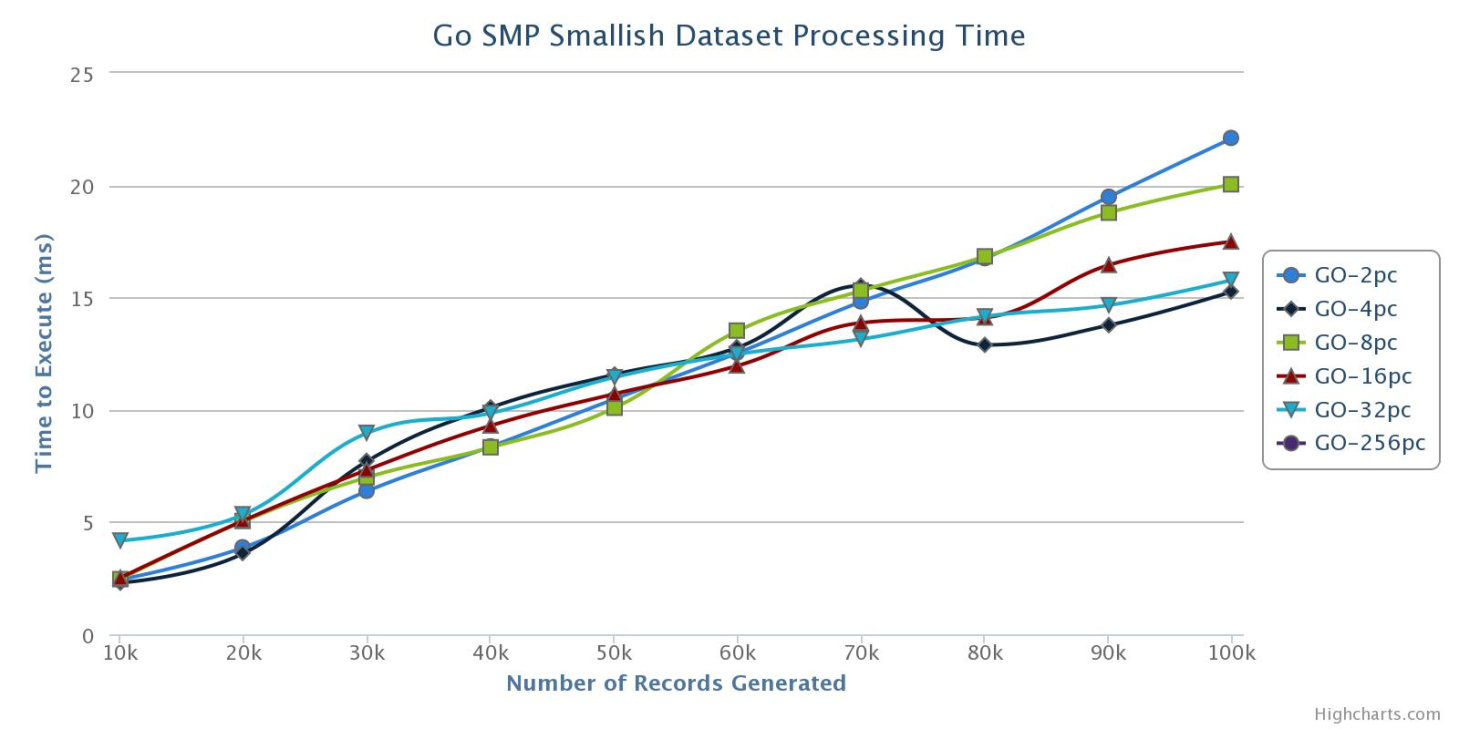


Figure : Go SMP Small Dataset Performance

When the applications process mid-size datasets (100k-1m), the performance advantages of using multiple cores begins to pay off, as evident in Figure 5.

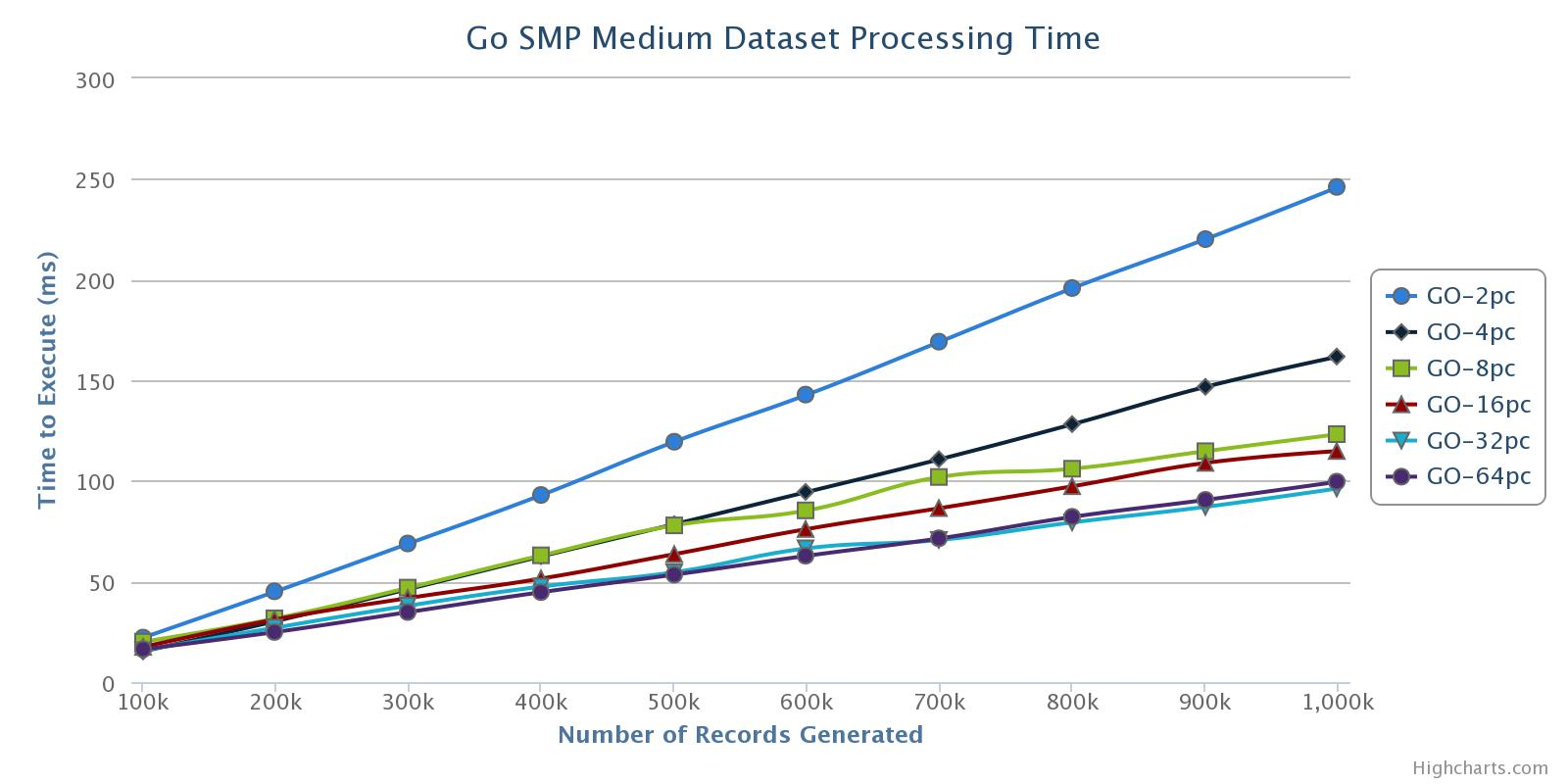


Figure : Go SMP Dataset Processing Time

Similarly, when the applications process very large data sets (10m-100m), increases in resources used directly correlates with decreases in processing time.

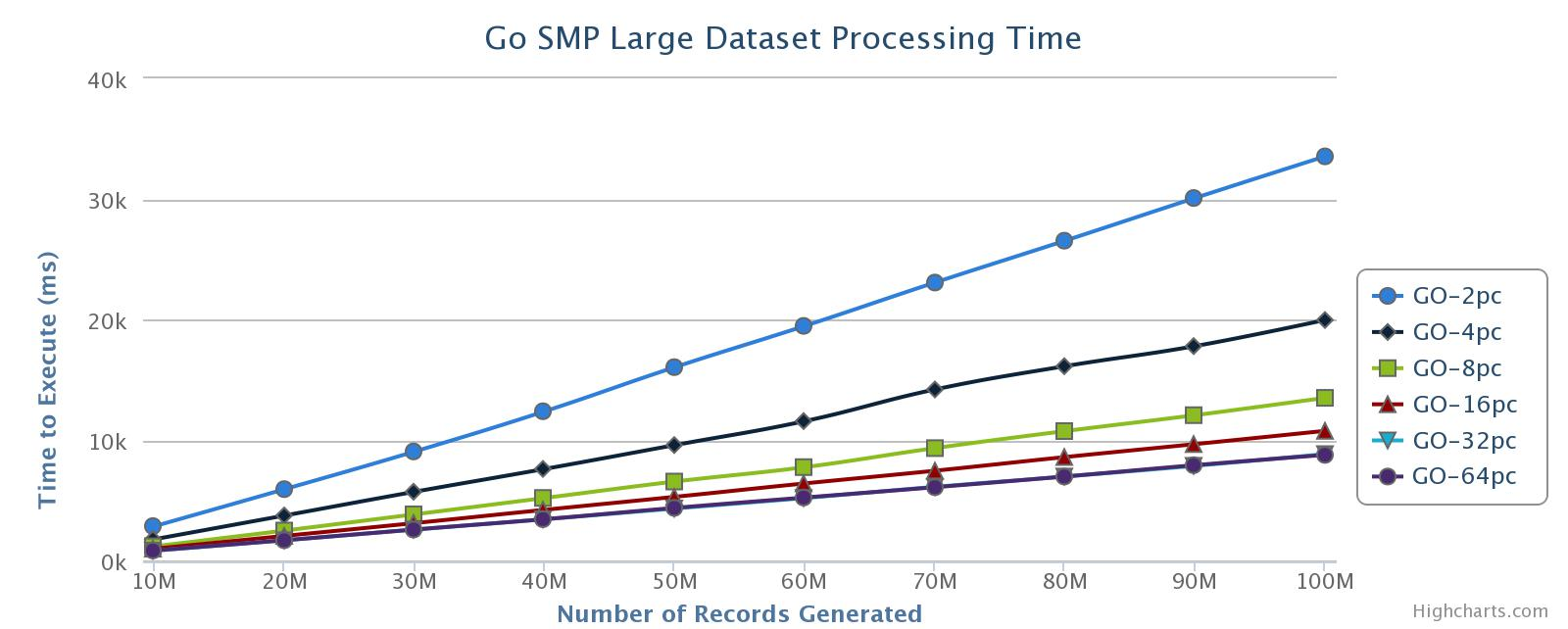


Figure : Go SMP Large Dataset Processing Time

#### Threading Versus Stock

To measure performance against a “control” system, an application was developed that uses Go’s built-in sort.Ints() function. As can been seen in Figure 7, Go’s stock sorting algorithm performs just a smidge better than the multithreaded GO application when using 1 CPU core. The slight disparity in performance should be attributed to the extra effort the multithreading application goes to create an extra CPU resource and marshal the data to and from it.

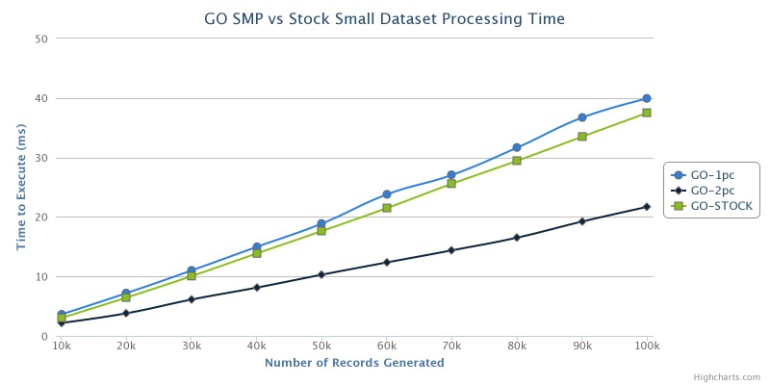


Figure : Go Stock vs Multithreaded performance

#### The Importance of a Higher Number of Resources

The above graphs allude to the fact that, as a dataset’s size increases, an application should seek to leverage a greater number of resources. This is true in the general sense; more CPUS a happier application makes. However, at least in the case of the current configuration, there is no significant performance gained by using more than 32 CPUs, as seen in Figure 8.

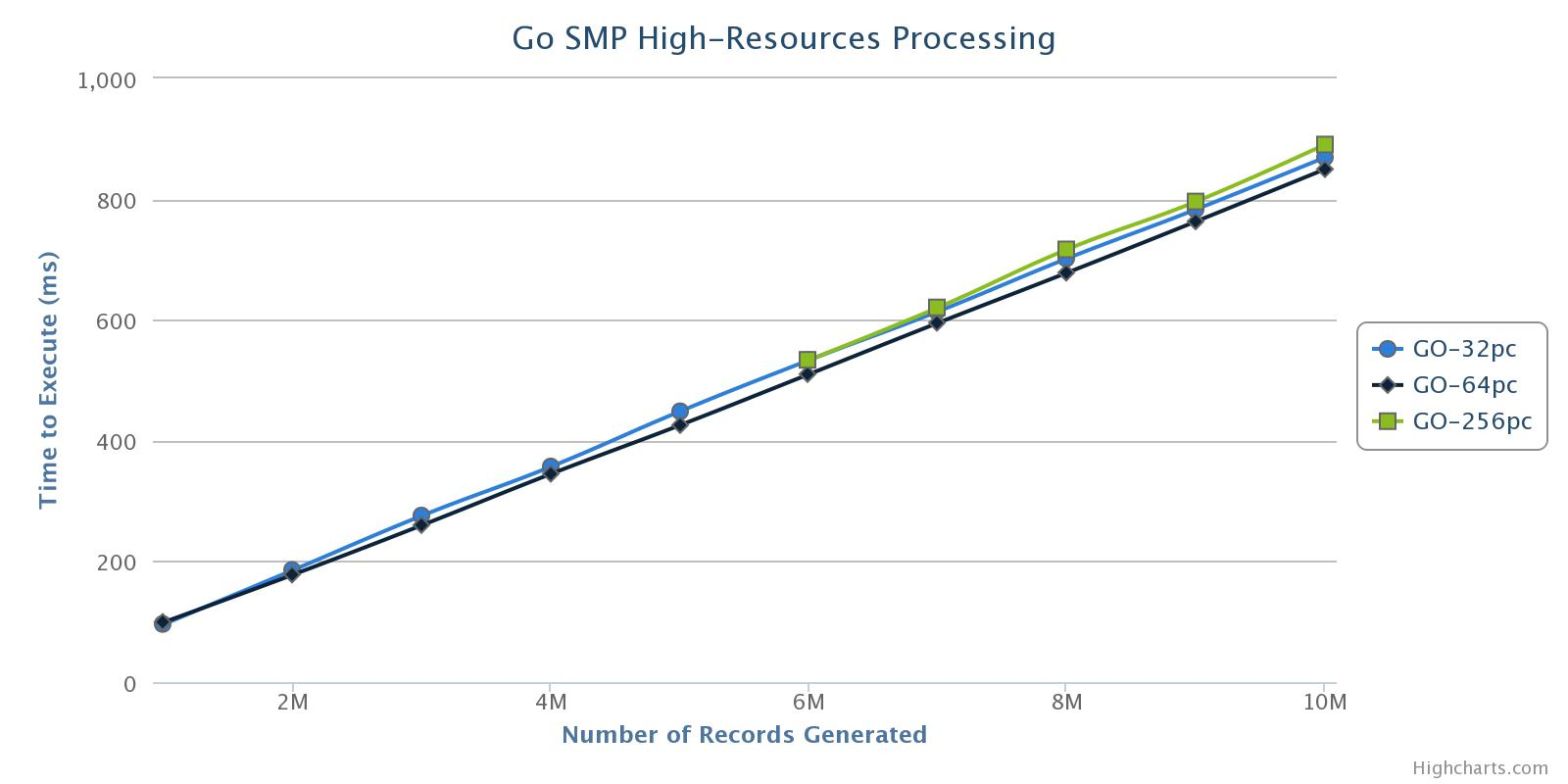


Figure : G0 SMP High-Resource Proecssing

### Go – Beowulf Cluster

#### Cluster Processing

As discussed above, the UNF Uranus server consists of 1 “master” computer that is used to communicate with 13 “slave” computers via a high-speed network bus. For these tests, “resource” refers to each slave computer being used.

#### Cluster Comparison

Figure 9 shows how each resource configuration performs when sorting smaller datasets (1k-10k records). As expected, for very small datasets, performance suffers when leveraging extra processing resources. However, sorting performance quickly converges as datasets increase in size; all resource configurations using more than 2 servers finished within the same 1ms range when sorting 10k records.

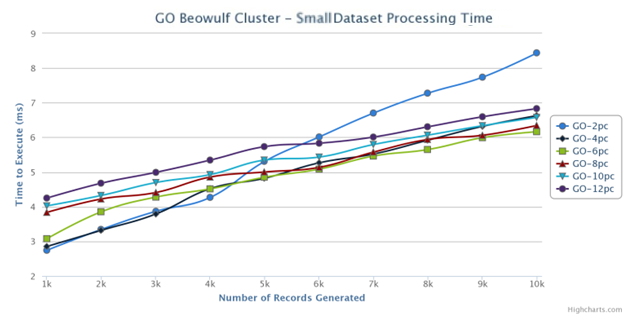


Figure : Go - Cluster Small Datasets Processing Time

As dataset sizes increase, there is a significant drop-off in the benefits gained from leveraging more server resources. Figure 10 shows the results of the Go Beowulf Cluster applications when processing medium-sized (100k-1m) datasets. There is less than a 3% performance difference between the processing time of 100k records when using 4 server resources (304ms) compared to using 12 server resources (294ms).

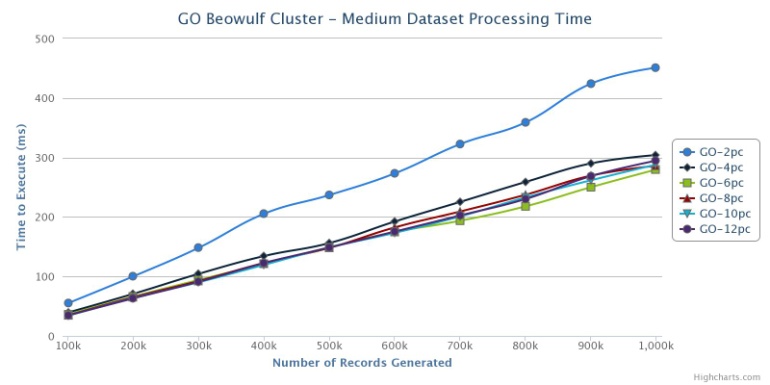


Figure : Go - Cluster Medium Dataset Processing Times

When processing large size (10-100m) datasets, an almost equivalent phenomenon occurs; no significant performance gain is seen when using more server resources. Figure 11 shows the performance results when processing large datasets. The performance difference between 6 servers (26234ms), 8 servers (27462ms), & 10 servers (26899ms) is less than 4%.

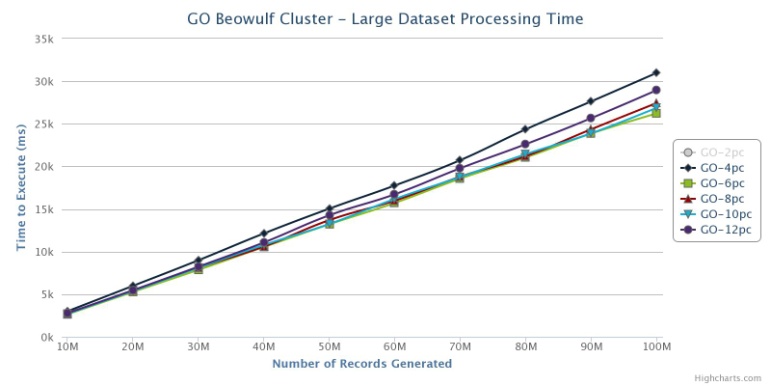


Figure : Go Cluster Large Dataset Processing Time

#### Cluster vs Stock

Figure 12 shows the performance results between select GO Cluster resource configurations and the stock GO sorting algorithm, sort.Ints(). Its performance falls squarely between a 2-server and 4-server cluster configuration. The fact that the 2-PC configuration is slower than the stock behavior can probably be attributed to the relatively higher latency and slower throughput present in communicating via a network bus as opposed to a system bus or crossbar.

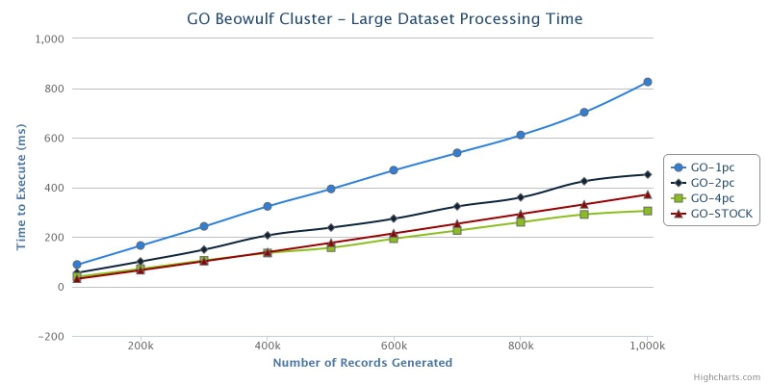


Figure : Go Cluster vs Stock Processing Time

### Go – SMP vs Cluster

As outlined in the testing specifications, when comparing cluster performance to SMP performance, the comparison had to take into account that each server would be fully utilizing all of their available CPU resources. This, in essence, meant that a 12-comptuer cluster should be assumed to be operating at an “effective” 48-core capacity. As Figure 13 shows, the fastest cluster configuration (12-PCs) is significantly slower than even a 2-CPU SMP machine. As noted earlier, the relatively higher latency and lower throughput present in the Beowulf cluster is probably the main reason that the clustered application performs so significantly slower than the SMP applications.

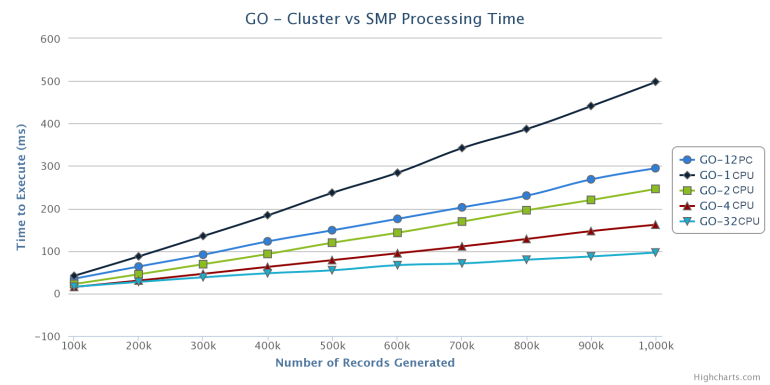
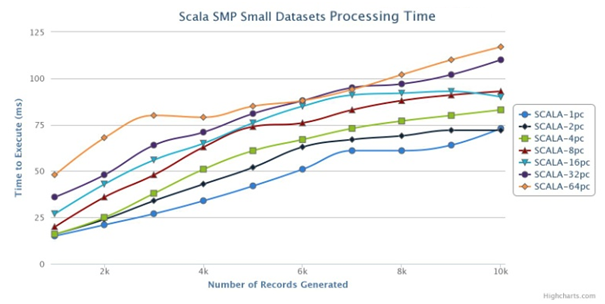


Figure : Go SMP vs Cluster Performance

### Scala - SMP Machine

#### Threading Comparison

For small datasets (1-10k), the Scala SMP applications performed partially as hypothesized. As figure 14 indicates, applications leveraging fewer resources outperformed those leveraging more resources. However, according to the graph, there doesn’t seem to be any indication that the applications performance will converge. On the contrary, each application’s performance seems to be linearly bound to the dataset size being processed.

Figure : Scala SMP Small Dataset Total Processing Times

Unfortunately, this trend in decreased performance when using more resources continues for both medium and large data sets. Figure 15 shows Scala application performance when sorting medium-sized datasets (100k-1m). What can be seen from the graph is that application performance increased incrementally when using 2 CPUs compared to using one CPU. Application performance when using 2 CPUs is equivalent to application performance when using 4 CPUs. Using more resources than 4 results in a degradation of performance.

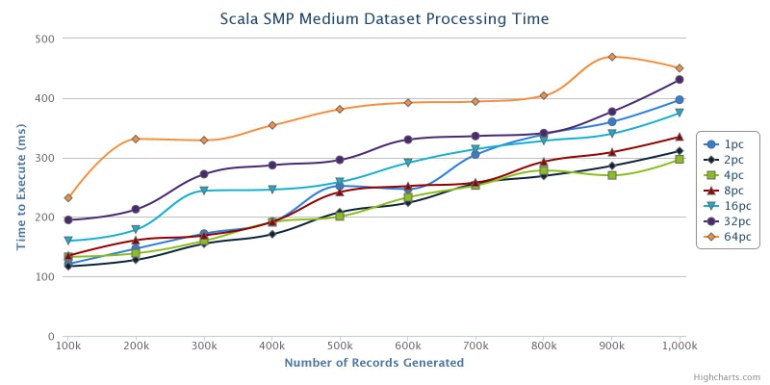


Figure : Scala SMP Medium Dataset Total Processing Time

Figure 16 shows Scala application performance for large datasets (10-100m). It can be seen that application performance peeks when leveraging 4/8 CPUs and then slowly degrades as the number of resources increases.

#### SMP - Sorting vs Combination

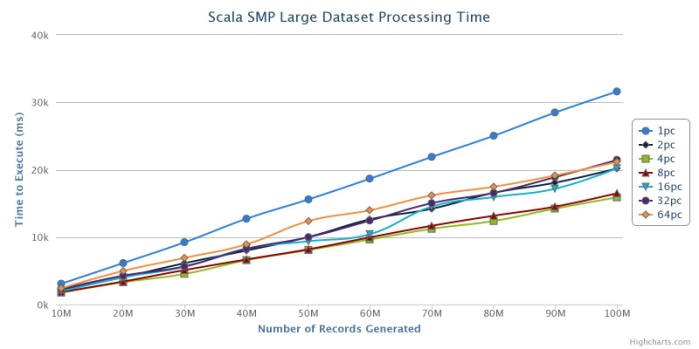


Figure Scala SMP Large Dataset Processing Time

It was discussed previously that two of the main time sinks for the application’s sorting algorithm would be the actual sorting of each small data set and the recombination of the small datasets into one larger data set. These 2 operations represent the whole of the execution time for each Scala application. As Figure 17 indicates, Scala application sorting performance increases when leveraging more resources.

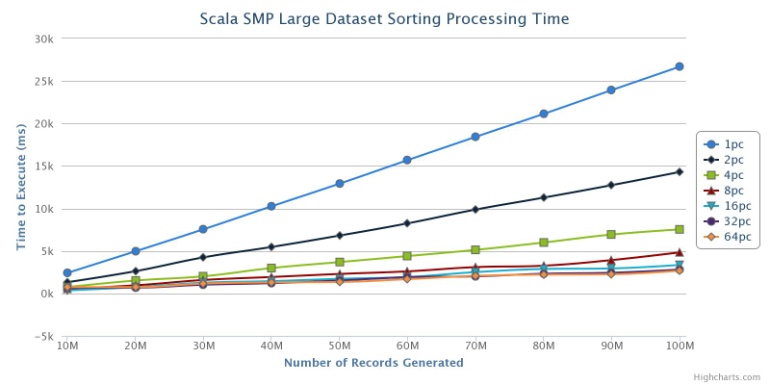


Figure 17: Scala SMP Sorting Processing Time

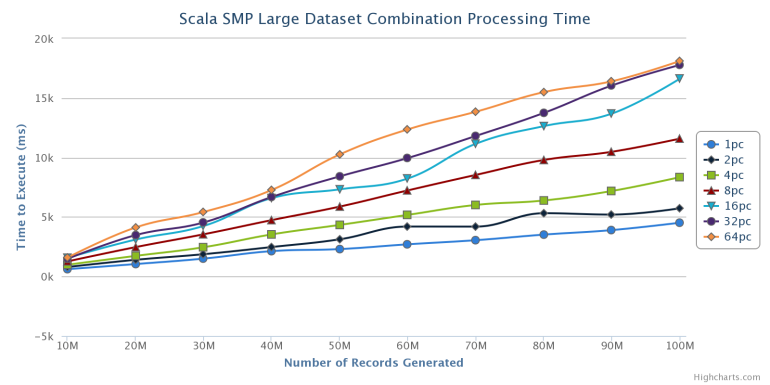
The exact opposite phenomenon is seen when combination time is analyzed. Figure 18 shows the time it takes to recombine large datasets in each SMP resource configuration. As explained in section 2.5.1, recombination performance is degraded as the number of resources was increased. 

Figure 18: Scala SMP Combination Processing Time

#### SMP versus Stock

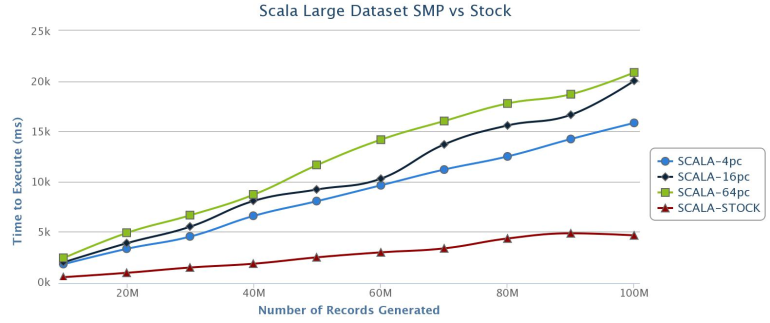


Figure Scala Large dataset SMP versus stock processing time

Figure 19 shows the difference in processing time between the Scala SMP application and the stock Scala sorting algorithm. As is discussed in section 1.2.2, the stock Scala sort algorithm sorts a given dataset using the quicksort algorithm essentially in-line, making no child function calls and is optimized at compile-time for efficiency. It is believed that this optimization, along with the elimination of any function-call overhead, causes the stock sorting algorithm to outperform all resource-heavy configurations.

### Scala – Beowulf Cluster

#### Cluster Processing

As discussed above, the UNF Uranus server consists of 1 “master” computer that is used to communicate with 13 “slave” computers via a high-speed network bus. For these tests, “resource” refers to each slave computer being used.

#### Cluster Comparison

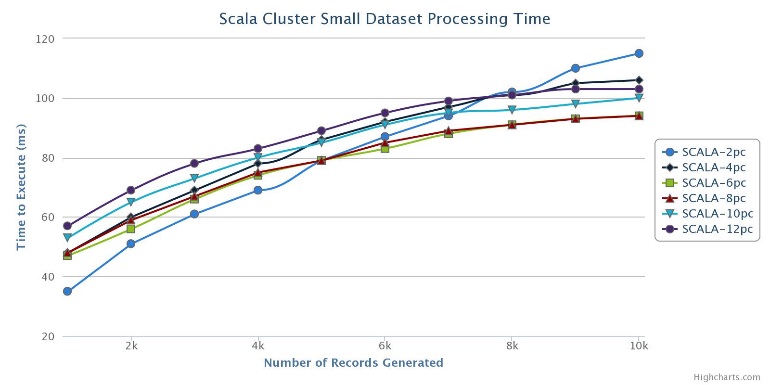


Figure - Scala Cluster Small Dataset Processing Time

Figure 20 represents how Scala performs when sorting smaller datasets across multiple resources. Performance seems to follow a logarithmic path. Initially, lower resource applications seem to outperform higher-resource applications, as expected. Also as expected, applications leveraging more resources show improved relative performance as the size of the sorted dataset increases. Mirroring the Scala SMP application results, performance peaked when leveraging 8 server resources and began to degrade as more servers were leveraged.

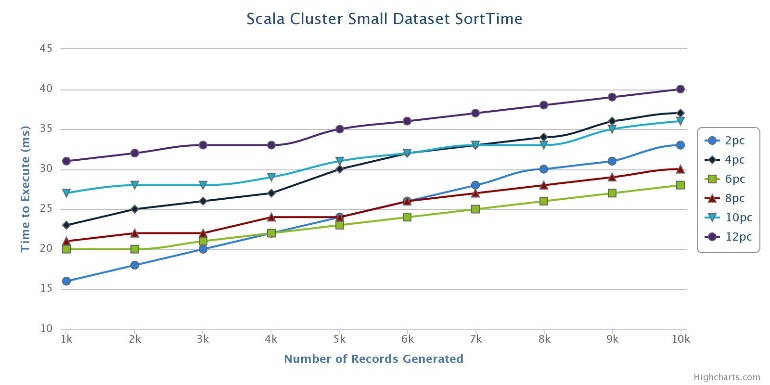


Figure Scala Cluster Small Dataset Sort Time

Figure 21 represents the sort time for various resource configurations when processing smaller data sets. As expected, little benefit is seen when leveraging larger resources. This lack of performance improvement can be attributed to the communication overhead experienced when using more and more non-local resources.

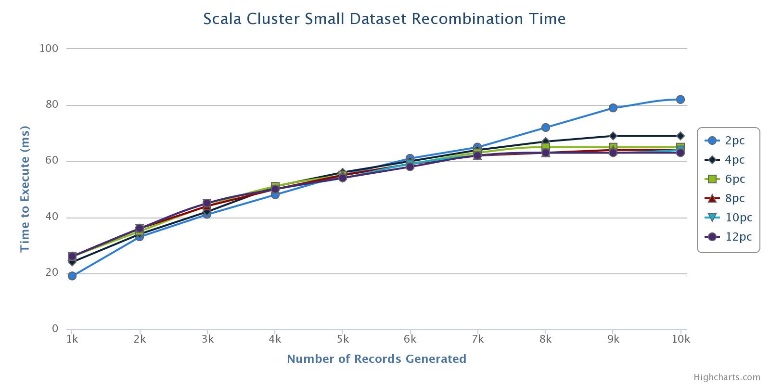


Figure Scala Cluster Small Dataset Recombination Time

Figure 22 illustrates the recombination time when using multiple resources for sorting. It was hypothesized that recombination time would increase as the number of resources increased. At least for the instance of smaller datasets, this is not the case. A reason may be that the number of datasets is sufficiently small enough so as to not impact recombination. Since all recombination is done on the host machine and the dataset sizes are similar and low.

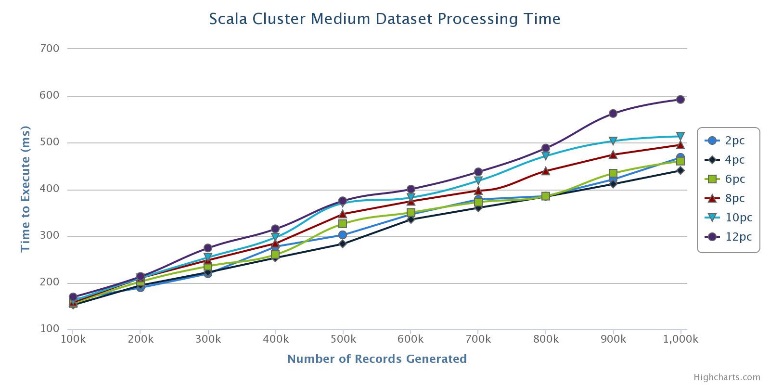


Figure Scala Cluster Medium Dataset Processing Time

As can been seen in Figure 23, performance peeks when using 4 resources and continues to drop as more resources are added. As can be seen in Figures 24 and 25, this is due to the slowdown caused by dataset recombination of the larger resourced applications outweighing the benefits gained from distributed sorting.

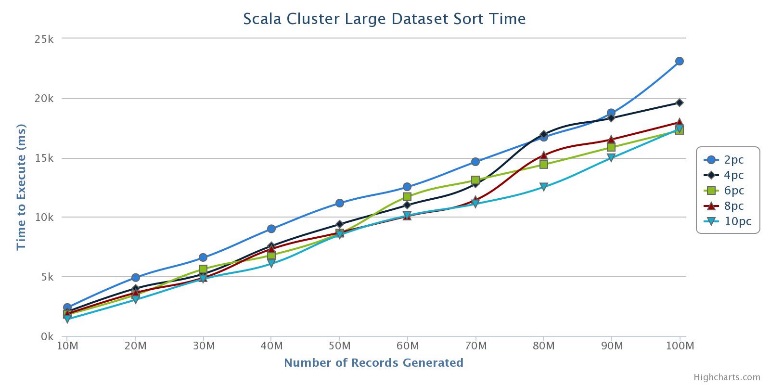


Figure Scala Cluster Large Dataset Sort Performance

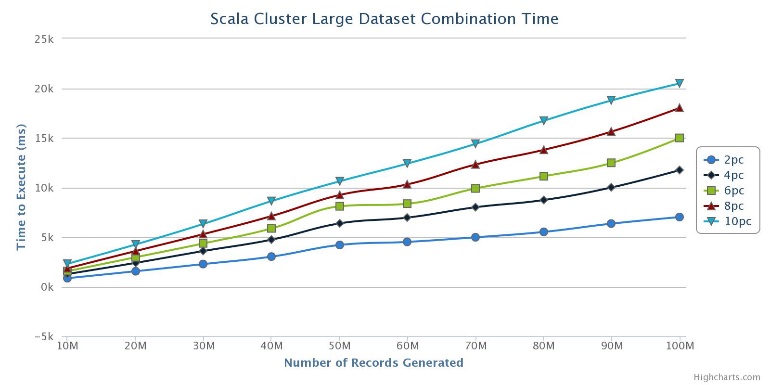


Figure Scala Cluster Large Dataset Combination Performance

As dataset sorting processing time decreased, albeit sporadically, when more resources were added, recombination time increased linearly, outpacing those gains. The non-linearity of the sorting performance is curious. More research needs to be conducted to determine the cause.

#### Cluster vs Stock

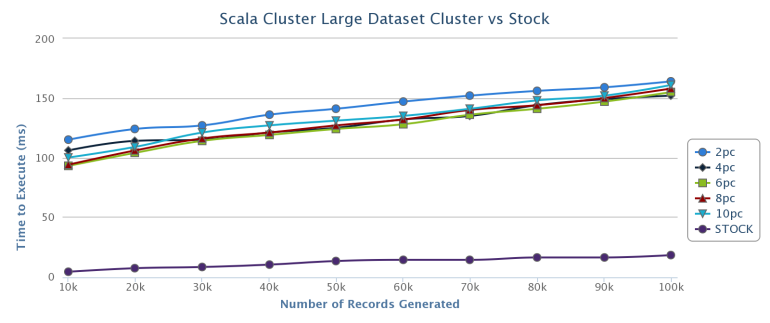


Figure Scala Cluster vs Stock Large Dataset Performance

As Figure 26 demonstrates, the Scala stock sorting algorithm performs significantly better than any of the cluster configurations, on average by 500%. As appendix B1.2 indicates that making function calls degrades performance significantly. In contrast, the Scala stock quicksort algorithm sorts the dataset using one tail-recursive function calls which allow it to sort the dataset essentially in-line, as is explained in section 1.2.2. It is believed that these two issues caused the performance disparity between the distributed application and the stock application.

### Scala - SMP vs Cluster

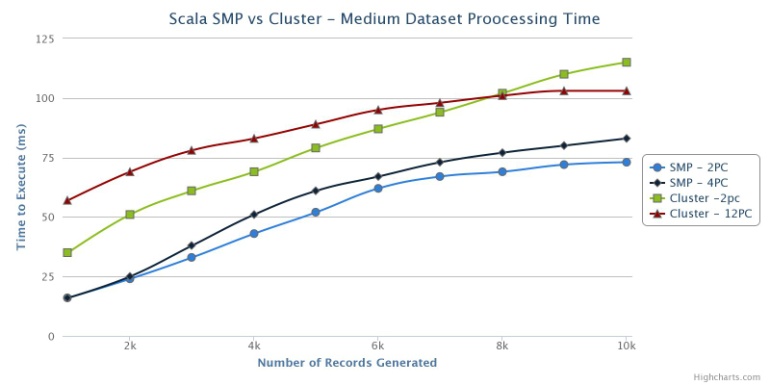


Figure Scala SMP vs Cluster Small dataset performance

Figure 27 shows a comparison between SMP and Cluster applications when sorting large datasets. For each, the best possible configuration was chosen: a 2-CPU SMP application and a 2PC cluster application. As can be seen, the SMP machine outperforms the Cluster with aplomb.

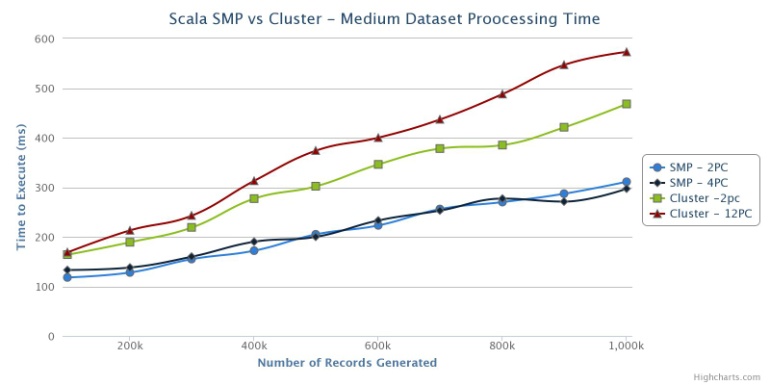


Figure Scala SMP vs Cluster Medium Dataset Performance

Figure 28 shows the continuation of this trend; the two best SMP configurations outperform the two best Cluster configurations by a significant margin. The same story can be told for large datasets, as illustrated by Figure 29.

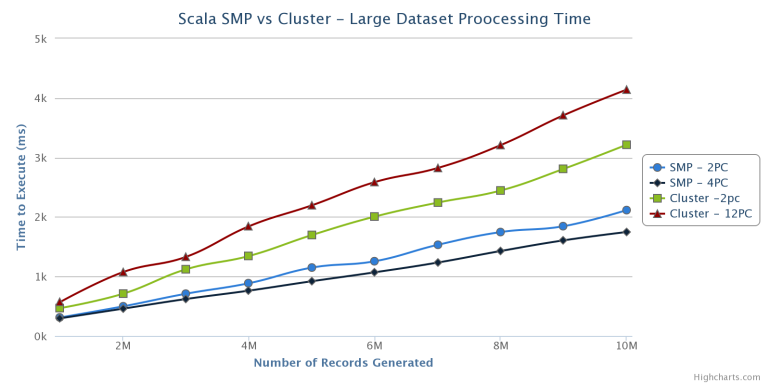


Figure Scala SMP vs Cluster Large Dataset Performance

## GO vs SCALA

### SMP MachineD:\Downloads\chart (7).jpeg

Figure Go vs Scala SMP Small Dataset Performance

As can be seen in Figure 30, GO is orders of magnitude faster than Scala. This is probably due to the performance differences covered in Appendix B1; for lower payloads, Scala is significantly slower than Go in many aspects.

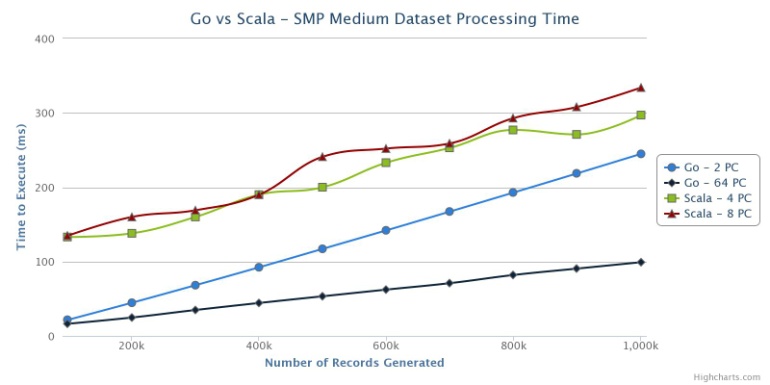


Figure Go vs Scala SMP medium Dataset

Figure 31 shows the performance difference between Go and Scala when processing medium-sized datasets. Go still outperforms Scala by 300-1000%; even the 2-CPU GO configuration outperforms Scala’s best efforts. The significant performance gap is shrinking, however.

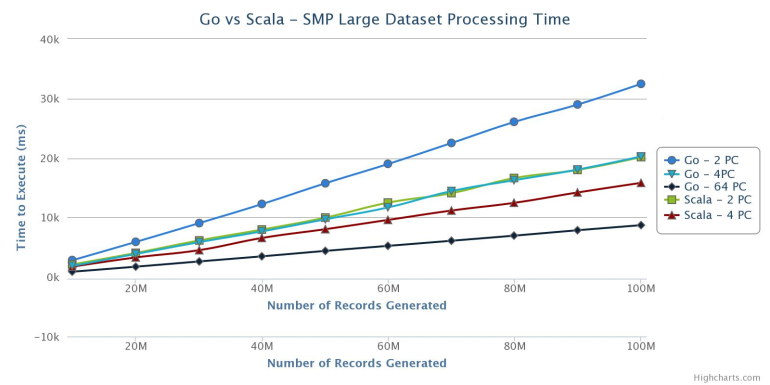


Figure Go vs Scala SMP Large Dataset

Figure 32 shows how GO and Scala SMP applications perform on large datasets. Go’s 64-CPU application still outperforms Scala’s highest performing application, by 100-120%. Of great note is that both Scala’s 2-CPU and 4-CPU configuration seems to outperform their Go counterparts, by ~20%. In fact, Scala’s 2-CPU configuration performs on-par with Go’s 4-CPU configuration. For all other configurations, Go outperforms Scala.

### Beowulf Cluster

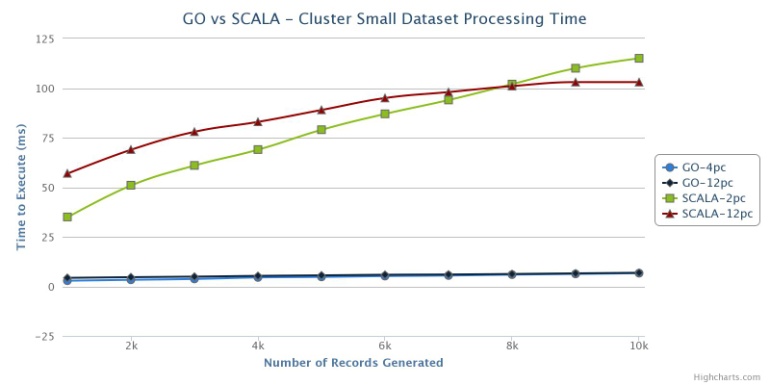


Figure Go vs Scala Cluster Small Dataset Processing Time

As can be seen in Figure 33, Go outperforms Scala significantly when sorting small datasets. As appendix B1.2 explains, for small pay-loads, Go outperforms Scala by a significant margin when creating variables and calling functions. For small payloads, this gives Go a distinct advantage over Scala.

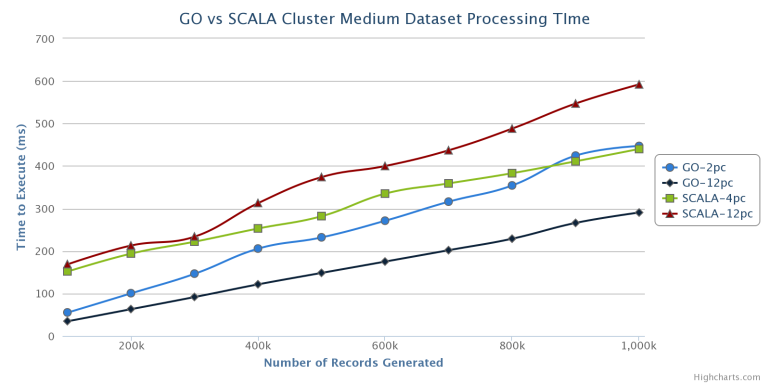


Figure Go vs Scala Cluster Medium Dataset Processing Time

As can be seen by figure 32, Go outperforms Scala when processing medium-sized datasets in a cluster configuration, although its performance lead is not as stark as it was when processing small datasets.

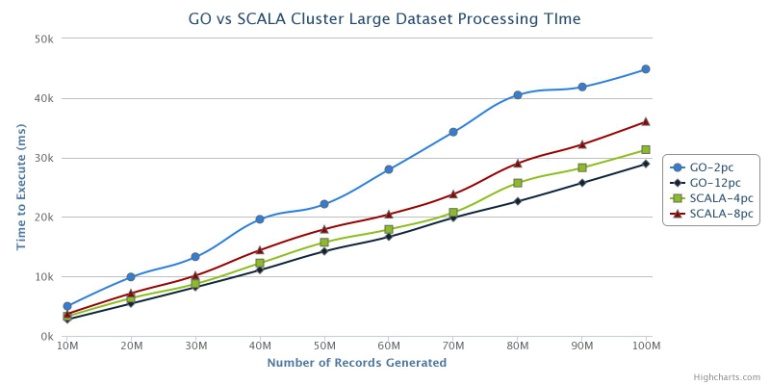


Figure Go vs Scala Cluster Large Dataset Processing Time

For large datasets, Go’s 12-PC configuration barely outperforms Scala’s 4-PC configuration, generally by less than 10%. Of note is that Scala’s low-resource applications outperform Go’s low-resource applications.

# Conclusion

## Go

When dealing with datasets larger than 1000 records, Go’s stock sorting algorithm doesn’t perform as well as a custom method that leveraged multiple CPUs; it only marginally outperforms a multithreading approach using only 1 CPU. Go’s stock application outperformed a 2-CPU cluster application but did not outperform a 4-CPU cluster application. It would seem that, given the current trend in CPU architecture, it would behoove a Go developer to leverage local multithreading instead of using cluster-level computing for trivial tasks such as dataset iteration and sorting. It remains to be seen how Go multithreading handles other tasks.

## Scala

Because of the relative slowdowns Scala experiences when making function calls outlined in appendix B1.2, it would behoove a developer to create a Scala application that was able to do everything that they needed to do in one function.

If a developer is interested in sorting data, Scala’s stock sorting algorithm performs admirably, besting all other configurations tested.

If a developer is interested in creating their own sorting application in Scala, local multithreading performs faster than its equivalent cluster configuration.

## Go versus Scala

Go’s multithreaded applications outperformed Scala’s for all instances, often by many orders of magnitude. Only for very large datasets (10m-100), small-resource (2-4 CPU) Scala applications were able to operate on-par or better than their Go counterparts.

Based on the evidence above, if an application is needed that will sort large amounts of data leveraging a computer with many CPUs, Go should be recommended over Scala.

## Caveats

* Because of Scala’s poor relative performance when doing –anything- in small increments, comparing the two languages on small dataset processing may not be an accurate assessment.
* Scala is a functional programming language. Its compiler is designed around this fact and optimizes code written in a functional manner much better than it does with code written in a imperative manner. The researcher’s inexperience with functional languages almost definitely negatively affected the performance of the Scala applications.
* Once each dataset was split into n smaller datasets, sorting was executed sequentially. Sorting may not be the most representative of how each language’s multithreading capabilities.
* Since the beginning of the research project, Scala’s Actor framework has been replaced with the Akka framework.
* As stated multiple times, Scala is a functional programming language. As such, it operates under the assumption that Scala developers are well-versed in functional programming paradigms. A developer can experience significant performance penalties when using imperative programming where functional programming is better suited.

# Further Research

* As discussed previously, sorting data is not a truly exhaustive representation of how a programming’s multiprocessing performance. Other tests, such as leveraging each language’s RPC framework to develop a multithreaded server application, could be used to test how each language deals with routed threading.
* As stated above, Scala’s Actor framework has been replaced with the Akka framework. It would be interesting to see if this improves Scala’s multithreading performance.

# Works Cited

Google, Inc. (2013). *sort.* Retrieved from The Go Programming Language: http://golang.org/pkg/sort/#Sort

Haller, P. (2010). *The Scala Actors API.* Retrieved from http://docs.scala-lang.org/overviews/core/actors.html

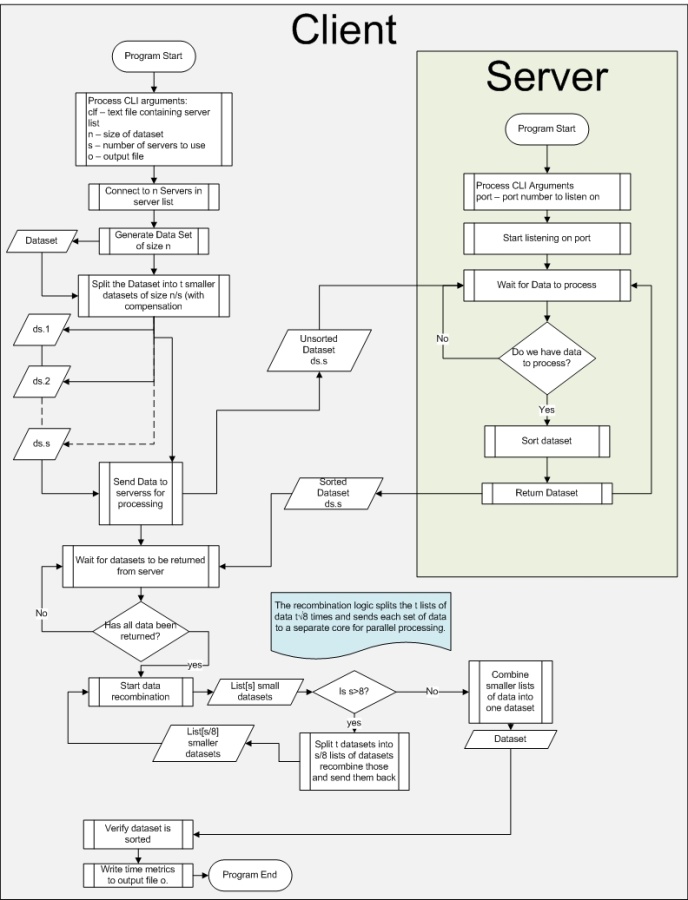
Hoare, T. (2013). *Quicksort.* Retrieved from Tony Hoare >> Contributions: http://www-cs-faculty.stanford.edu/~eroberts/courses/soco/projects/2008-09/tony-hoare/quicksort.html

Pike, R. (2011, June). *GoRoutines.* Retrieved from http://go.googlecode.com/hg-history/release.r60/doc/GoCourseDay3.pdf

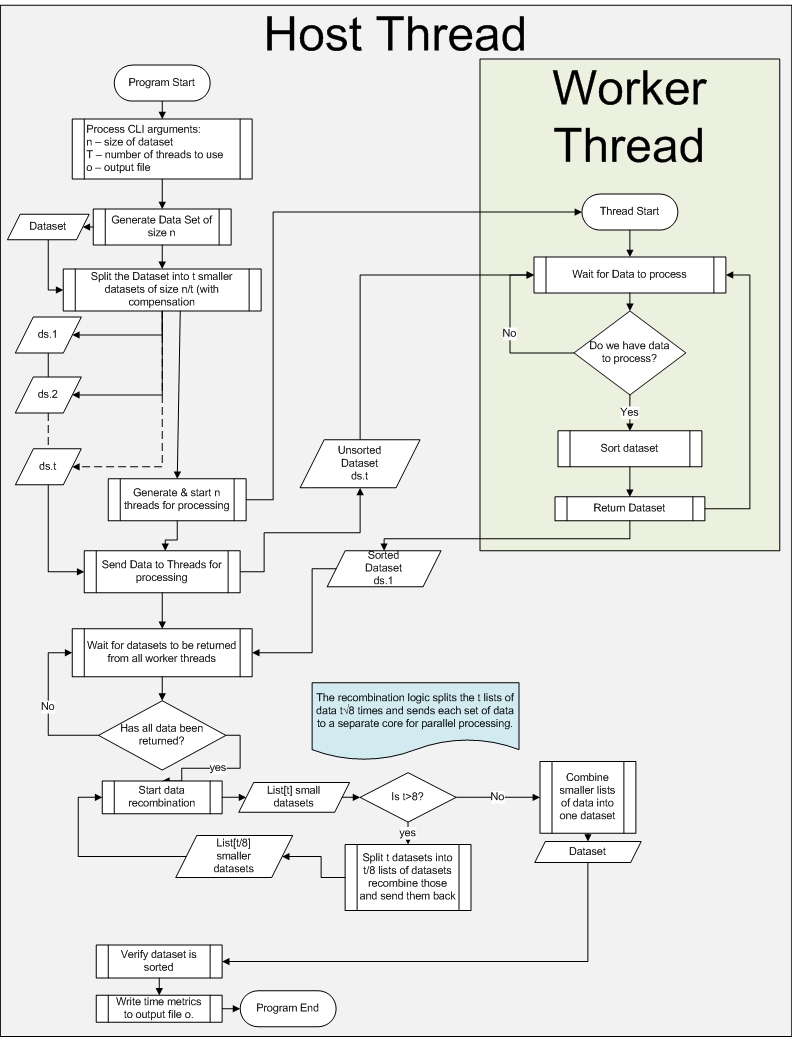
Appendix

# Appendix A: Detailed Program Flow

## Program Flow – Cluster



## Program Flow – Multicore



# Appendix B: Ancillary Testing

The below tests were conducted as reference material for the experiment. Unless otherwise noted, all tests were executed on the UNF ATLAS SMP machine. As these tests generally include much fewer data points than those used in general testing, they are to be used as anecdotal evidence only.

## B1: General Language Performance

### B1.1: Number Generation

As discussed in The Hypothesis Section, it was believed that since Scala uses a Class-based object model where all variables inherit from the System.Object class and Go does not, Go would generate integers faster than Scala. In the interest of science, tests were ran to prove this point. For the test, a dataset of integers with size N was initialized and populated. Below is a chart representing the results of those tests..

Table : Go vs Scala - Generating N integers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | Go-Time (ns) | Scala-Time (ns) | % Difference | Sampe Size |
| 1000 | 8462 | 716436 | 8367% | 100 |
| 10000 | 40455 | 1450618 | 3486% | 100 |
| 100000 | 554510 | 4649636 | 739% | 100 |
| 1000000 | 4372779 | 5880964 | 34% | 100 |
| 10000000 | 46732601 | 19179855 | -59% | 100 |

As the results above indicate, the GO Runtime almost always generates numbers faster than SCALA. Of note is that, as the dataset size increases, the gap in performance closes dramatically. Whereas Go’s processing time appears to increase linearly relative to the number of integers generated, Scala’s processing time increases logarithmically. This may indicate that Scala uses some unknown functionality to streamline generating large numbers of data.

### B1.2: Function Calls

In Scala, every type used is an object, including functions. In Go, functions are freely-accessible memory addresses; because Go is an imperative language and lacks the trappings of an Object-oriented language such as Scala. Because of this difference, it was hypothesized that function calls in Scala would be executed slower than those in GO. A test was devised to call a function that did nothing with no parameters N times to determine if there was a performance difference between the two languages simply by calling functions and, if so, what difference there was. Table 2 below shows the result of that test:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Average Time per function call (ns) | |  |  |
| N | Go | Scala | % Difference | Sample Size |
| 1000 | 0.908 | 419.31 | 46080% | 10 |
| 10000 | 0.882 | 110.23 | 12398% | 10 |
| 100000 | 0.854 | 58.36 | 6734% | 10 |
| 1000000 | 0.869 | 7.16 | 724% | 10 |
| 10000000 | 0.861 | 1.56 | 81% | 10 |

Like the results found in B1.1, GO is able to call functions significantly faster than Scala is but, as the number of function calls increases, the performance gap decreases significantly. Go’s function-call time linearly relative to N. Scala’s function-call performance increases logarithmically as N increase. This may actually explain the performance efficiency increases seen in B1.1. If function-calls are called faster the more times that they’re called then the Instantiate() function used to create Scala objects would, by correlation, run more efficiently the more times that it’s called. More testing should be conducted to determine if Scala’s function-call performance surpasses that of Go’s.

As discussed earlier, Scala only allows Call-By-Value function calls, which duplicates any parameters passed to a function when it is called. Conversely, GO allows the user to use Call-By-Reference function calls which only passes the memory address of each parameter, removing the need to duplicate them. It was hypothesized that, because of this, function calls that passed the large data sets that were used during testing within GO would inherently be faster than the same calls made in Scala. A test was devised that passed a dataset of size N to a function N times and do nothing with it. It is expected that, because GO passes the dataset by-reference that its performance, as seen above, will increase linearly as the number of function calls (N) increases. Table 3 shows the results of this test.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Average Time per function call (ns) | | | |  |
|  | Go |  | Scala |  |  |
| N | No Param | with Params | No Param | With Param | Sample Size |
| 1000 | 0.908 | 0.911 | 419.31 | 456.51 | 100 |
| 10000 | 0.882 | 0.842 | 110.23 | 128.54 | 100 |
| 100000 | 0.854 | 0.861 | 58.36 | 26.08 | 100 |
| 1000000 | 0.869 | 0.857 | 7.16 | 6.80 | 100 |
| 10000000 | 0.861 | 0.863 | 1.56 | 5.08 | 100 |

Go’s processing time, when passing dataset as function parameters, remained constant and was considerably lower than Scala’s processing time. Go function calls with parameters executed as swiftly as Go function calls made without passing parameters. As expected, Go’s function call processing time does not seem to be affected by the size of the dataset being passed to it. Compared to Go’s function call performance, Scala is glacial. When processing lower payloads, It is often orders of magnitude slower than function calls in Go. What is interesting is that Scala’s function-call performance when processing parameters is not always slower than its non-parameterized function call performance. The only thing that can be gleaned from these results is that Scala is significantly slower than Go when making function calls.

# Appendix C: Figure-Accompanying Data Charts

This section contains the datasets associated with each chart used above. Each chart will be labeled with the Figure associated with it.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Chart 3 – Go SMP Performance – Small Dataset Processing Time (ns) | | | | | | |
| N | GO-2pc | GO-4pc | GO-8pc | GO-16pc | GO-32pc | GO-64pc |
| 1000 | 0.341821 | 0.305898 | 0.499028 | 0.520529 | 2.429801 | 2.717078 |
| 2000 | 0.450384 | 0.510057 | 2.356209 | 2.350747 | 3.245504 | 4.582575 |
| 3000 | 0.715222 | 0.675122 | 2.730313 | 2.802889 | 3.285328 | 4.368909 |
| 4000 | 0.871739 | 0.852094 | 2.950223 | 2.810074 | 4.12868 | 4.826279 |
| 5000 | 1.055594 | 1.009611 | 2.418785 | 3.392522 | 4.013066 | 4.679578 |
| 6000 | 1.516485 | 2.18502 | 2.683926 | 2.937724 | 4.232106 | 3.194481 |
| 7000 | 1.67289 | 2.84298 | 2.22492 | 2.806122 | 4.146613 | 2.759137 |
| 8000 | 1.870501 | 2.065869 | 2.194479 | 2.570237 | 3.110007 | 2.866542 |
| 9000 | 1.806131 | 1.967905 | 2.325479 | 2.414834 | 3.933597 | 4.028512 |
| 10000 | 2.183865 | 2.144469 | 2.295617 | 2.413813 | 4.239847 | 3.945569 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Chart 4 – Go SMP Performance – Smallish Dataset Processing Time (ns) | | | | | | |
| N | GO-2pc | GO-4pc | GO-8pc | GO-16pc | GO-32pc | GO-64pc |
| 10000 | 2.183865 | 2.144469 | 2.295617 | 2.413813 | 4.239847 | 3.945569 |
| 20000 | 3.813149 | 3.687198 | 5.135305 | 5.181977 | 5.432959 | 5.576919 |
| 30000 | 6.127384 | 7.838139 | 7.148023 | 7.135375 | 8.848912 | 9.839114 |
| 40000 | 8.128153 | 9.649768 | 8.327324 | 9.167372 | 9.983879 | 10.75452 |
| 50000 | 10.30178 | 11.7336 | 10.19526 | 10.65238 | 11.15365 | 11.75554 |
| 60000 | 12.36542 | 13.60017 | 13.49392 | 11.89129 | 12.54352 | 13.07501 |
| 70000 | 14.40023 | 16.23842 | 15.51722 | 13.80656 | 13.10956 | 13.98079 |
| 80000 | 16.53601 | 12.20363 | 16.95312 | 14.30291 | 14.06932 | 14.90019 |
| 90000 | 19.24575 | 13.65925 | 18.8228 | 16.37153 | 14.59836 | 15.75308 |
| 100000 | 21.70893 | 15.23357 | 20.52102 | 17.63849 | 15.74207 | 16.3641 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Chart 5 – Go SMP Performance – Medium Dataset Processing Time (ns) | | | | | | |
| N | GO-2pc | GO-4pc | GO-8pc | GO-16pc | GO-32pc | GO-64pc |
| 100000 | 21.70893 | 15.23357 | 20.52102 | 17.63849 | 15.74207 | 16.3641 |
| 200000 | 44.61688 | 30.49798 | 31.57017 | 31.55586 | 27.04996 | 24.72925 |
| 300000 | 68.22403 | 47.04854 | 46.66829 | 40.55717 | 38.11541 | 34.85883 |
| 400000 | 92.3298 | 63.28933 | 63.04597 | 51.64033 | 48.00094 | 44.32125 |
| 500000 | 117.1444 | 78.83959 | 78.47026 | 63.72302 | 54.16419 | 53.33485 |
| 600000 | 142.0265 | 96.96908 | 90.40751 | 76.62048 | 66.94876 | 62.2873 |
| 700000 | 167.3206 | 112.2479 | 106.2739 | 86.23569 | 70.33562 | 70.99566 |
| 800000 | 192.9117 | 130.2081 | 97.75835 | 96.68141 | 78.71934 | 81.97651 |
| 900000 | 218.7007 | 149.0042 | 108.3572 | 108.3784 | 87.12501 | 90.59721 |
| 1000000 | 244.843 | 164.1235 | 119.2737 | 117.2037 | 95.90639 | 99.10668 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Chart 6 – Go SMP Performance – Large Dataset Processing Time (ns) | | | | | | |
| N | GO-2pc | GO-4pc | GO-8pc | GO-16pc | GO-32pc | GO-64pc |
| 10000000 | 2833.959 | 1869.633 | 1244.189 | 1039.06 | 865.9947 | 880.2212 |
| 20000000 | 5899.482 | 3866.782 | 2555.31 | 2102.498 | 1729.558 | 1724.47 |
| 30000000 | 9048.509 | 5845.689 | 3895.531 | 3175.643 | 2608.402 | 2599.615 |
| 40000000 | 12268.85 | 7673.237 | 5257.513 | 4261.022 | 3474.045 | 3464.502 |
| 50000000 | 15746.33 | 9689.257 | 6585.117 | 5346.7 | 4358.613 | 4373.865 |
| 60000000 | 19013.37 | 11703.76 | 7692.544 | 6443.507 | 5236.094 | 5224.329 |
| 70000000 | 22521.42 | 14433.14 | 9384.862 | 7529.4 | 6127.716 | 6082.266 |
| 80000000 | 26076.1 | 16274.26 | 10770.11 | 8628.557 | 7004.446 | 6944.806 |
| 90000000 | 28992.47 | 18057.47 | 12145.06 | 9726.608 | 7896.664 | 7848.222 |
| 100000000 | 32459.84 | 20248.35 | 13554.17 | 10834.88 | 8771.858 | 8700.837 |

|  |  |  |  |
| --- | --- | --- | --- |
| Chart 7 – Go Stock vs Multiprocess Large Dataset Processing Time (ns) | | | |
| N | GO-1pc | GO-2pc | GO-STOCK |
| 10000000 | 5647.395 | 2833.959 | 5288.434 |
| 20000000 | 11657.26 | 5899.482 | 11101.21 |
| 30000000 | 17973.33 | 9048.509 | 17031.07 |
| 40000000 | 24189.52 | 12268.85 | 23013.96 |
| 50000000 | 30578.21 | 15746.33 | 29148.62 |
| 60000000 | 37787.5 | 19013.37 | 35311.71 |
| 70000000 | 44531.46 | 22521.42 | 41701.94 |
| 80000000 | 52449.23 | 26076.1 | 48051.35 |
| 90000000 | 58470.6 | 28992.47 | 54263.63 |
| 100000000 | 64658.87 | 32459.84 | 60782.48 |

|  |  |  |  |
| --- | --- | --- | --- |
| Chart 8 – GO SMP HIGH-RESOURCE PROCESSING TIME (ns) | | | |
| N | GO-16pc | GO-32pc | GO-64pc |
| 1000000 | 117.2037 | 95.90639 | 99.10668 |
| 2000000 | 206.8269 | 185.1781 | 177.0495 |
| 3000000 | 309.7443 | 270.9255 | 259.3509 |
| 4000000 | 412.1757 | 354.751 | 340.9763 |
| 5000000 | 516.1452 | 443.1557 | 423.1303 |
| 6000000 | 621.3584 | 530.063 | 506.4416 |
| 7000000 | 724.4641 | 608.5495 | 591.2593 |
| 8000000 | 830.039 | 694.1222 | 675.3853 |
| 9000000 | 932.0708 | 779.1281 | 760.3669 |
| 10000000 | 1039.06 | 865.9947 | 846.1967 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Chart 9 - GO - CLUSTER SMALL DATASETS PROCESSING TIME (ns) | | | | | | |
| N | GO-2pc | GO-4pc | GO-6pc | GO-8pc | GO-10pc | GO-12pc |
| 1000 | 2.716377 | 2.844766 | 3.035249 | 3.782478 | 3.984496 | 4.294327 |
| 2000 | 3.282744 | 3.300142 | 3.847531 | 4.112187 | 4.295735 | 4.631279 |
| 3000 | 3.804866 | 3.734187 | 4.170428 | 4.412398 | 4.679052 | 4.911688 |
| 4000 | 4.344513 | 4.49898 | 4.505006 | 4.788921 | 4.901761 | 5.288936 |
| 5000 | 5.360306 | 4.769398 | 4.74421 | 4.936517 | 5.237517 | 5.556344 |
| 6000 | 6.042745 | 5.177391 | 5.041939 | 5.102622 | 5.43479 | 5.849976 |
| 7000 | 6.681948 | 5.461675 | 5.3834 | 5.503045 | 5.718774 | 5.995131 |
| 8000 | 7.21275 | 5.916089 | 5.637955 | 5.875308 | 6.018574 | 6.243237 |
| 9000 | 7.743206 | 6.267872 | 5.960255 | 5.94047 | 6.240338 | 6.555359 |
| 10000 | 8.396469 | 6.641059 | 6.16061 | 6.282773 | 6.528607 | 6.851198 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Chart 10 – Go Cluster Medium Dataset Processing Time (ns) | | | | | | |
| N | GO-2pc | GO-4pc | GO-6pc | GO-8pc | GO-10pc | GO-12pc |
| 100000 | 55.2792 | 39.32091 | 35.84198 | 34.6717 | 34.00735 | 34.78201 |
| 200000 | 100.5845 | 70.03474 | 65.85002 | 65.09527 | 63.90027 | 63.18231 |
| 300000 | 146.5857 | 104.6142 | 93.90604 | 92.79769 | 90.739 | 91.71839 |
| 400000 | 205.6153 | 133.4339 | 120.5399 | 122.9167 | 118.8818 | 121.4968 |
| 500000 | 232.0566 | 156.4521 | 147.1355 | 147.0014 | 147.8316 | 148.432 |
| 600000 | 271.1779 | 191.7021 | 173.8586 | 180.444 | 171.6039 | 175.0178 |
| 700000 | 315.9848 | 225.8805 | 193.7167 | 209.0977 | 199.996 | 201.7918 |
| 800000 | 354.1813 | 257.3021 | 218.6796 | 236.8381 | 231.7883 | 228.832 |
| 900000 | 424.3785 | 289.5853 | 247.3628 | 267.9386 | 260.0138 | 265.977 |
| 1000000 | 447.0065 | 303.3069 | 277.9836 | 285.8687 | 287.383 | 290.4311 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Chart 11 – Go Cluster Large Dataset Processing Time (ns) | | | | | | |
| N | GO-2pc | GO-4pc | GO-6pc | GO-8pc | GO-10pc | GO-12pc |
| 10000000 | 5000.027 | 2994.47 | 2621.894 | 2706.317 | 2689.831 | 2773.582 |
| 20000000 | 9906.648 | 5956.139 | 5227.277 | 5434.868 | 5368.236 | 5462.548 |
| 30000000 | 13307.96 | 9006.569 | 7832.907 | 8146.339 | 8111.046 | 8208.193 |
| 40000000 | 19605.9 | 12077.19 | 10531.49 | 10661.26 | 10713.29 | 11096.34 |
| 50000000 | 22163.87 | 15074.48 | 13115.35 | 13698.49 | 13242.11 | 14231.48 |
| 60000000 | 27990.6 | 17827.44 | 15626.21 | 15848.82 | 16195.17 | 16678.76 |
| 70000000 | 34276.87 | 20763.35 | 18576.69 | 18745.6 | 18720.52 | 19868.02 |
| 80000000 | 40488.53 | 24420 | 21005.6 | 21021.34 | 21288.02 | 22636.31 |
| 90000000 | 41859.48 | 27556.36 | 23628.3 | 24507.58 | 23912.18 | 25726.98 |
| 100000000 | 44856.83 | 31127.25 | 26066.05 | 27484.36 | 26672.11 | 28934.59 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Chart 12 – Go Cluster vs Stock Processing Time (ns) | | | | |
| N | GO Cluster - 1pc | GO Cluster - 2pc | GO Cluster - 4pc | GO-STOCK |
| 100000 | 88.29345 | 55.2792 | 39.32091 | 30.88623 |
| 200000 | 163.0745 | 100.5845 | 70.03474 | 65.44403 |
| 300000 | 237.9145 | 146.5857 | 104.6142 | 101.3825 |
| 400000 | 318.908 | 205.6153 | 133.4339 | 138.2555 |
| 500000 | 390.0303 | 232.0566 | 156.4521 | 175.8544 |
| 600000 | 461.6972 | 271.1779 | 191.7021 | 214.0647 |
| 700000 | 536.8347 | 315.9848 | 225.8805 | 252.5757 |
| 800000 | 610.4121 | 354.1813 | 257.3021 | 291.5914 |
| 900000 | 690.0015 | 424.3785 | 289.5853 | 330.8612 |
| 1000000 | 831.7519 | 447.0065 | 303.3069 | 370.5718 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Chart 13 – Go SMP vs Cluster Processing Time (ns) | | | | | |
| N | Cluster - 12 PC | SMP - 1 CPU | SMP - 2 CPU | SMP - 4 CPU | SMP - 32 CPU |
| 100000 | 34.78201 | 39.92986 | 21.70893 | 15.23357 | 15.74207 |
| 200000 | 63.18231 | 84.89058 | 44.61688 | 30.49798 | 27.04996 |
| 300000 | 91.71839 | 131.8151 | 68.22403 | 47.04854 | 38.11541 |
| 400000 | 121.4968 | 181.7361 | 92.3298 | 63.28933 | 48.00094 |
| 500000 | 148.432 | 235.1785 | 117.1444 | 78.83959 | 54.16419 |
| 600000 | 175.0178 | 279.0172 | 142.0265 | 96.96908 | 66.94876 |
| 700000 | 201.7918 | 335.2515 | 167.3206 | 112.2479 | 70.33562 |
| 800000 | 228.832 | 386.1487 | 192.9117 | 130.2081 | 78.71934 |
| 900000 | 265.977 | 439.6134 | 218.7007 | 149.0042 | 87.12501 |
| 1000000 | 290.4311 | 491.6572 | 244.843 | 164.1235 | 95.90639 |