TidalScale System Performance Analysis for R

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
0.2	2015-01-20	version 0.2	CP
0.5	2015-04-26	version 0.5	CP

Contents

Chuck Piercey <chuck.piercey@tidalscale.com>

1 Executive Summary

This white paper explores the performance of the R analytics of a large data set on a TidalScale system.

The TidalScale Hyperkernel provides the ability to scale-up the execution of applications beyond the limits of a single server. TidalScale's approach delivers better performance for the application while avoiding the need for re-architecting the application or re-engineering for scale-out to achieve good performance.

R is an analytic and statistical programming language whose use is rapidly spreading as organizations sharpen their ability to understand and learn from data they amass. Many operations in R are memory intensive, and analysts and data scientists often struggle to keep their working data sets within the limitations of a single computer. The TidalScale Hyperkernel allows multiple physical computers to run as a single system that allows the unmodified application to use the aggregate RAM, CPUs and I/O of the underlying hardware. The R applications therefore "see" a single computer system while the TidalScale Hyperkernels enable growth and performance by scaling-out on commodity hardware.

In this report, we measure the performance and capacity of R applications with very large datasets, running on "bare metal" servers versus a server instance on TidalScale. The performance results described below illustrate important aspects of TidalScale performance as of version 1.0. We demonstrate scalable performance of four different R operations which, more generally, are representative of the kinds of analytic workloads that benefit from running on large, coherent systems.

We explore two aspects of scalable performance:

- **Conquering the Memory Cliff** that applications typically encounter when they exceed the size of reasonably priced currently available systems, and
- **Linear hardware costs** & **linear resource increase** achieved by combining commodity components: scaling the system size by adding components produces a bigger system without slowing the system down, and can be done dynamically as customer requirements dictate.

This comparison demonstrates that removing the memory barrier and constraints of a single piece of hardware, R applications can continue to scale with better performance to very large Terabyte-scale workloads.

2 Test Hardware

All of the tests were run on standard server systems built with the following components:

- **Motherboard**: Colfax CX1260i-X6
- **Processor**: Single Xeon E5-26032 v3 6C/6T 1.6 GHz 6.4GT/s 15mb cache 85w (only one of 2 sockets populated)
- **Chipset**: Intel C612
- **Memory**: 8 slots filled (out of 24 slots) with 16GB DDR4 2133MHz DIMMs for a total of 128GB RAM
- **Storage**: two Samsung 512GB SSDs and two 2TB hard drives (need brand and model#'s)

The tests are run across two system configurations:

- **Linux directly on Hardware**: Our base of comparison is a single 128 GB Colfax CX1260i as described above.
- **Linux on Multi-node TidalScale TidalPods**: We use multiple standard 128GB servers in a 5 node TidalScale configuration [1], called a TidalPod. The TidalPod uses a 10G network as its "system resource bus" that is invisible to the guest CentOS 6.5 OS running on the TidalPod.

[1] For the purposes of this test, we do not count the memory footprint of the TidalScale hypervisor itself, which is about 5% overhead, or the TidalScale replication cache (whose size can be controlled by the user).

3 Test Software

The testing environment included the following software:

- CentOS 6.5
- Revolution R Enterprise 3.1

4 R Test Data

To performance test R we use the Centers for Medicare and Medicaid Service's (CMS) 2008-2010 Data Entrepreneurs' Synthetic Public Use Data Set (DE-SynPUF). Through Medicare, Medicaid, the Children's Health Insurance Program, and the Health Insurance Marketplace, CMS provides medical coverage to 100 million people and has a plethora of data to offer.

The **CMS DE-SynPUF** data set was created with the goal of providing a realistic set of claims data in the public domain while providing the very highest degree of protection to the Medicare beneficiaries' protected health information. The purposes of the DE-SynPUF are to:

1. Allow data entrepreneurs to develop and create software and applications that may eventually be applied to actual CMS claims data;
2. Train researchers on the use and complexity of conducting analyses with CMS claims data prior to initiating the process to obtain access to actual CMS data; and,
3. Support safe data mining innovations that may reveal unanticipated knowledge gains while preserving beneficiary privacy.

The DE-SynPUF contains five types of data for 2008, 2009 and 2010 – Beneficiary Summary, Inpatient Claims, Outpatient Claims, Carrier Claims, and Prescription Drug Events.

Table 1: CMS DE-SynPUF Data Set Overview

DE-SynPUF	**Unit of record**	**Number of Records 2008**	**Number of Records 2009**	**Number of Records 2010**
Beneficiary Summary	Beneficiary	2,326,856	2,291,320	2,255,098
Inpatient Claims	claim	547,800	504,941	280,081
Outpatient Claims	claim	5,673,808	6,519,340	3,633,839
Carrier Claims	claim	34,276,324	37,304,993	23,282,135
Prescription Drug Events (PDE)	event	39,927,827	43,379,293	27,778,849

Although the CMS DE-SynPUF has very limited inferential research value to draw conclusions about Medicare beneficiaries due to the synthetic processes used to create the data, it provides a realistic insurance claims data set for **R** testing purposes. For more details about the data structures in this data set and coding guidelines see the SynPUFs web site.

The CMS data set is chunked into 20 "samples." Each sample consists of 8 raw .csv files:

1. DE1_0_2008_Beneficiary_Summary_File_Sample_1.csv"
2. DE1_0_2009_Beneficiary_Summary_File_Sample_1.csv"
3. DE1_0_2010_Beneficiary_Summary_File_Sample_1.csv"
4. DE1_0_2008_to_2010_Inpatient_Claims_Sample_1.csv"
5. DE1_0_2008_to_2010_Outpatient_Claims_Sample_1.csv"

6. DE1_0_2008_to_2010_Carrier_Claims_Sample_1A.csv"
7. DE1_0_2008_to_2010_Carrier_Claims_Sample_1B.csv"
8. DE1_0_2008_to_2010_Prescription_Drug_Events_Sample_1.csv"

Loaded as separate data structures in R, the 8 files of a single sample occupy 6.0GB on disk. Loading all 20 samples from all 160 csv files (8 files/sample * 20 samples) occupies ~120GB on disk.

5 R Test Program

The R test operations included in this benchmark are:

1. Read .csv files of a specified sample range and load the matching data structures in a R data frame,
2. Add some required metadata columns for the regressions and join the data structures by patient ID,
3. Perform a **Generalized Additive Model** linear regression, and
4. Perform a **Generalized Linear Model** linear regression.

For more details on GLM and GAM see this CMU Lecture.

Each test operation step requires additional memory. For instance, for a single sample the sequence of memory footprint progresses as follows:

- Load: 6.0GB
- Join: 9.4GB
- GAM & GLM: 16.9GB

Extended across all 20 samples, these four operations require 336GB of system memory in total.

The testing methodology involves running these four operations at various in-memory sizes as shown in the table below:

Table 2: Test Data Points

Number of Samples	Starting Sample	Ending Sample	Starting In-Memory Size	Ending In-Memory Size
1	1	1	6.0GB	16.9GB
3	1	3	18.1GB	50.5GB
5	1	5	30.2GB	84.1GB
8	1	8	48.3GB	134.8GB
10	1	10	60.1GB	168.4GB
13	1	13	78.43GB	218.4GB
20	1	20	120.7GB	336.0GB

In other words, the ending in-memory size is driven by the number of samples loaded combined with the types of analysis performed on the data.

The key code of each R operation in the test is detailed below:

R Operation 1: Load Data

```
# Get patient data, also known as beneficiary summary
```

```r
patient_file_name <- "DE1_0_%s_Beneficiary_Summary_File_Sample_%s.csv"
load_time <- load_time + system.time(patient <- get_data_read_csv_patients(patient_file_name, ↩
    sample_id, years))
patient <- data_pre_process(patient)
# Get Inpatient claims data
inpatient_file_name <- "DE1_0_2008_to_2010_Inpatient_Claims_Sample_%s.csv"
load_time <- load_time + system.time(inpatient <- get_data_read_csv(inpatient_file_name, ↩
    sample_id, ...))
# Get out patient claims data
outpatient_file_name <- "DE1_0_2008_to_2010_Outpatient_Claims_Sample_%s.csv"
load_time <- load_time + system.time(outpatient <- get_data_read_csv(outpatient_file_name, ↩
    sample_id, ...))
# Get carrier claims data
claims_file_name <- "DE1_0_2008_to_2010_Carrier_Claims_Sample_%s.csv"
load_time <- load_time + system.time(carrier_claims <- get_data_read_csv(claims_file_name, ↩
    claims_sample, ...))
# Get patient drugs data
drug_file_name <- "DE1_0_2008_to_2010_Prescription_Drug_Events_Sample_%s.csv"
load_time <- load_time + system.time(drug <- get_data_read_csv(drug_file_name, sample_id, ...) ↩
    )
```

R Operation 2: Join Data

```r
# Create a new column for year, using existing date column
inpatient <- add_year(inpatient, "clm_from_dt")
join_time <- join_time + system.time(cms_inpatient <- left_join(patient, inpatient, by = c(" ↩
    desynpuf_id", "year")))
# Create a new column for year, using existing date column
outpatient <- add_year(outpatient, "clm_from_dt")
join_time <- join_time + system.time(cms_outpatient <- left_join(patient, outpatient, by = c(" ↩
    desynpuf_id", "year")))
carrier_claims <- add_year(carrier_claims, "clm_from_dt")
join_time <- join_time + system.time(cms_claims_patient <- left_join(patient, carrier_claims, ↩
    by = c("desynpuf_id", "year")))
drug <- add_year(drug, "srvc_dt")
join_time <- join_time + system.time(cms_drug_patient <- left_join(patient, drug, by = c(" ↩
    desynpuf_id", "year")))
```

R Operation 3: Generalized Additive Model 2 Linear Regression

```r
# Predict total inpatient expenses using Generalized Additive Model 2.
# Includes patient demographic, chronic condition, and prescription drug covariates.
# Patient demographics varibales includes age, gender, and ethnicity/race.
formula_gam1 <- (medreimb_ip + benres_ip + pppymt_ip) ~ s(age) + bene_sex_ident_cd + ↩
    race_white + race_black + race_others + race_hispanic + sp_alzhdmta + sp_chf + sp_chrnkidn ↩
     + sp_cncr + sp_copd + sp_depressn + sp_diabetes + sp_ischmcht + sp_osteoprs + sp_ra_oa + ↩
    sp_strketia + s(qty_dspnsd_num) + s(days_suply_num) + s(ptnt_pay_amt) + s(tot_rx_cst_amt)
fit_gam1_time <- system.time(fit_gam1 <- gam(formula_gam1, family=gaussian(link=identity), ↩
    data=cms_drug_patient, subset = (medreimb_ip > 1000)))
```

R Operation 4: Generalized Linear Model Linear Regression

```r
# Predict total inpatient expenses using Generalized Linear Model.
# Includes patient demographic, chronic condition, and prescription drug covariates.
# Patient demographics varibales includes age, gender, and ethnicity/race.
formula_glm1 <- (medreimb_ip + benres_ip + pppymt_ip) ~ age + bene_sex_ident_cd + race_white + ↩
     race_black + race_others + race_hispanic + sp_alzhdmta + sp_chf + sp_chrnkidn + sp_cncr + ↩
     sp_copd + sp_depressn + sp_diabetes + sp_ischmcht + sp_osteoprs + sp_ra_oa + sp_strketia ↩
    + qty_dspnsd_num + days_suply_num + ptnt_pay_amt + tot_rx_cst_amt
fit_glm1_time <- system.time(fit_glm1 <- glm(formula_glm1, family=gaussian, data= ↩
    cms_drug_patient, subset = (medreimb_ip > 1000)))
```

The complete test program and the dataset used are available on git for external use at http://www.github.com/tidalscale/R_benchmark_test.

6 Bare Metal Performance Results

R will take advantage of as much system memory as it needs to perform its work. During and R operation such as GLM or GAM, R may demand memory several multiples of the original data set size to complete its work. Is system memory is insufficient to complete the operation, R will begin paging to storage (i.e. swap to disk) for some not-infinite amount after which time it fails. We can describe this as a "Memory Cliff." The chart below is an example of the Memory Cliff: on 128GB bare metal nodes our test R workload functions up to 108GB of available memory after which point the job fails.

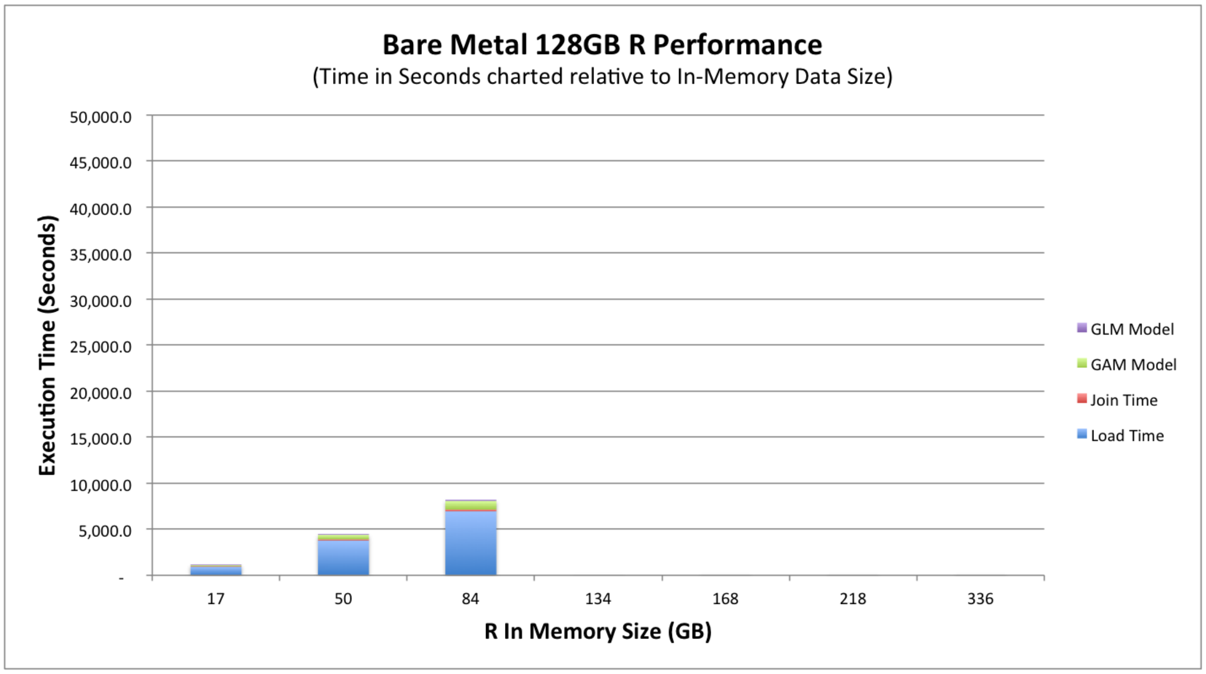

Figure 1: Bare Metal - R Execution Job Times by In-Memory Job Size

One of two failures occur past the Memory Cliff:

1. R fails to obtain a block of memory, or
2. CentOS kills the R process with a `kill -9` error (this means that the process is using too much memory for the system).

Beyond 108GB of system memory, using bare metal systems requires rewriting the R application and incorporating new libraries to divide the work across multiple servers. This introduces risk of result variations and inconsistencies in the analytic results.

7 TidalScale Performance Results

We booted a TidalScale system with 655GB of RAM and running an unmodified CentOS 6.5 guest OS and ran the same test sequence as in the bare metal case. TidalScale allows the guest OS and applications to continue running in-memory beyond the limits of a single physical system. We found that R is able to run and perform well when it has access to very large amounts of RAM. The results show that TidalScale enables R and CentOS to seamlessly to handle growing data analysis requirements by binding multiple smaller hardware systems into a single, coherent virtual system.

TidalScale's architecture virtualizes and mobilizes all system resources. The guest operating system's view of what hardware it manages and how it is connected does not correspond to the real physical system's structure. Instead, the guest system runs on a virtual machine. The TidalScale HyperKernel presents the GuestOS with virtual processors, virtual physical memory blocks and virtualized I/O devices. TidalScale assigns virtual processors, memory, and I/O to the physical processors, memory blocks and I/O devices so that the guest system will execute as efficiently as possible. The TidalScale HyperKernel reacts to Guest system activity, learns and reconfigures physical resources so that processors that access the same memory blocks will be on the same node and share the same caches as much as possible for as long as they continue to interact. The physical location of memory, processors, and I/O devices is rearranged on the fly based on the observed working set of the current workload.

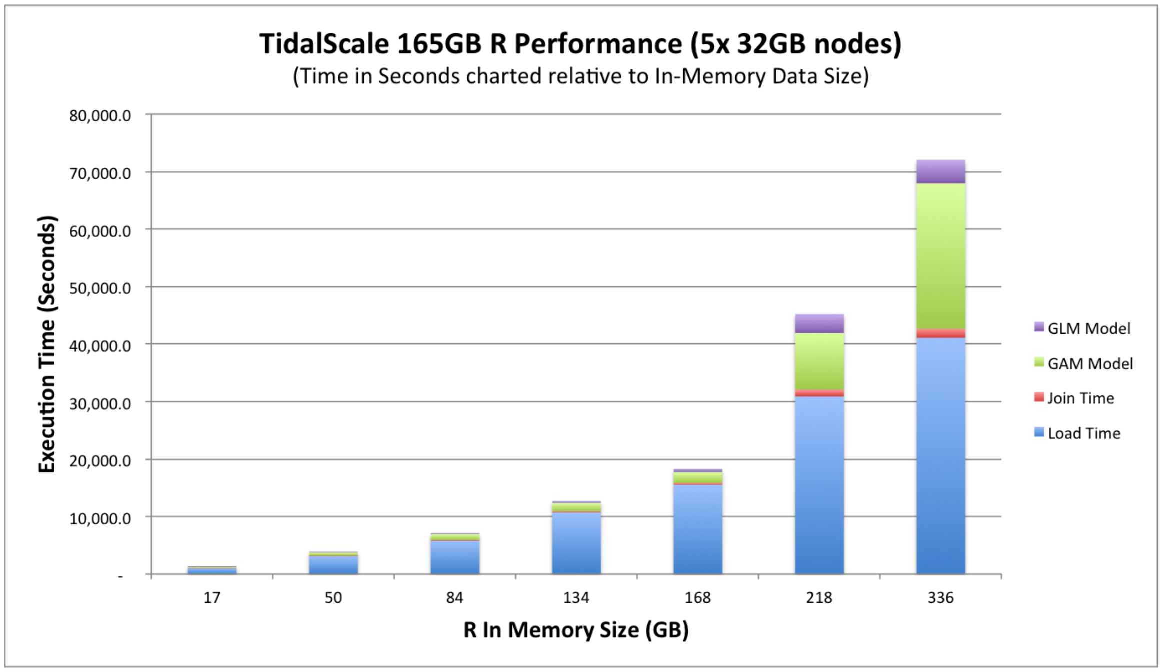

Figure 2: TidalScale (5x128GB nodes) - R Execution Job Times by In-Memory Job Size

This result shows that TidalScale is effectively performing with in-memory R data loads that are greater than any single 128GB node in the 5 node TidalPod. This is seen more clearly in a side-by-side comparison of the two sets of test results.

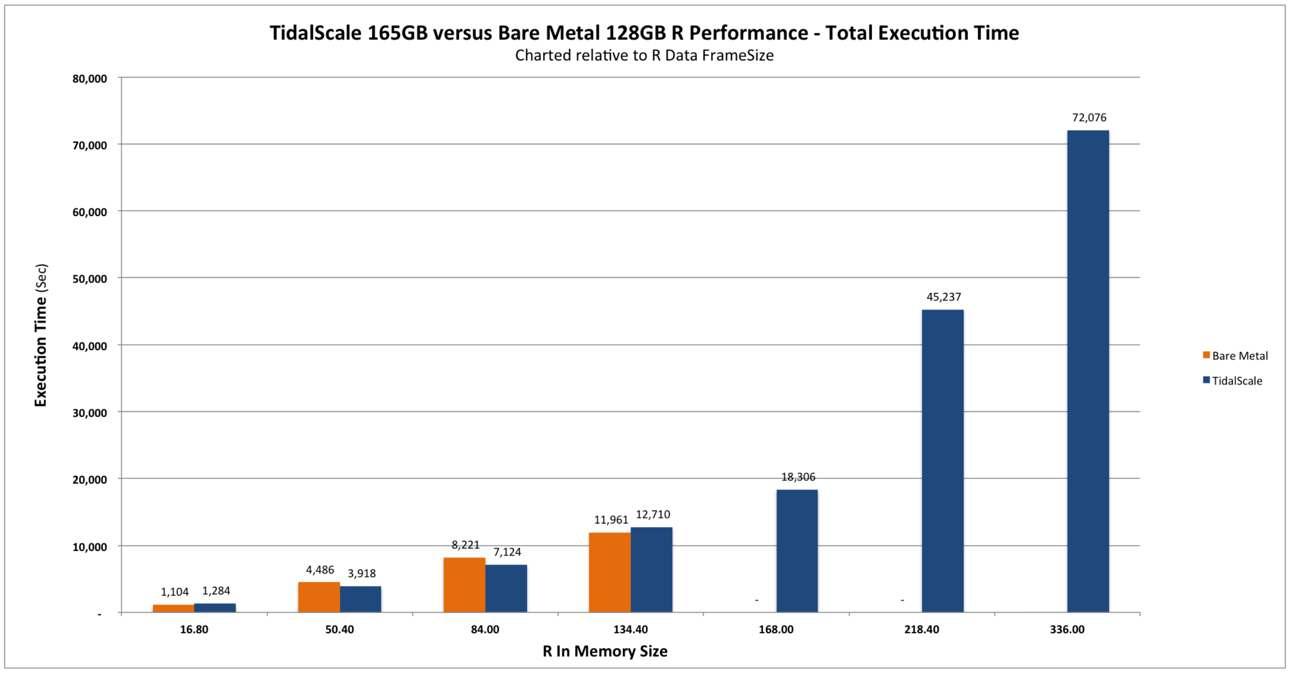

Figure 3: TidalScale (5x128GB nodes) - R Execution Job Times by In-Memory Job Size

For the more mathematically inclined, the relationship between data size and execution time is more discernable visually on a log scale chart of total execution time scatter-plotted against in-memory data size.

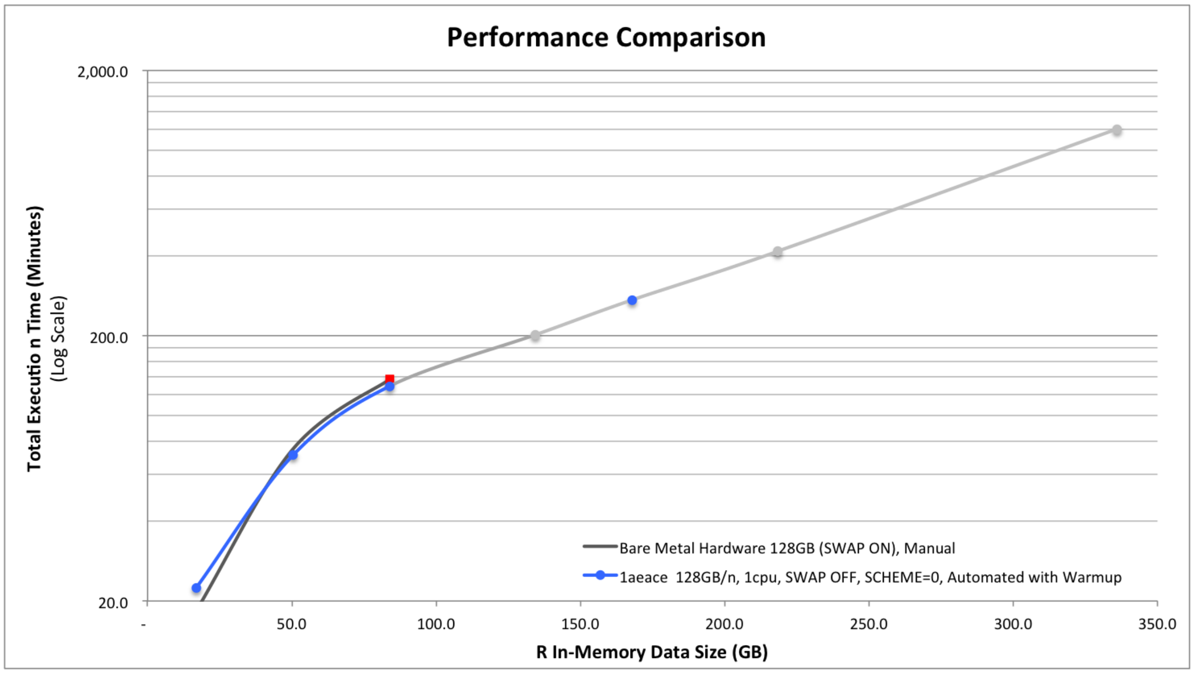

Figure 4: TidalScale Performance versus Bare Metal

In short, the TidalScale HyperKernel's software defined scalability effectively removes the Memory Cliff and delivers a scalable and high performance platform for running analytic loads against datasets that exceed the capacity of traditional single servers.

8 Linear Hardware Cost Increase Using Modular Components

As the dataset grows larger, at some point it will exceed the capacity of the current TidalScale Pod. We simply add another node containing cpus and memory so that the data can be held in memory, giving those extra resources to Linux and to the application. Since each node is identical, the cost of the pod grows linearly with the size of memory, while gaining additional processors that can process the data faster. The following chart shows this linear cost growth, compared with other approaches to scaling system size, showing the combined hardware/software cost growth for running Linux.

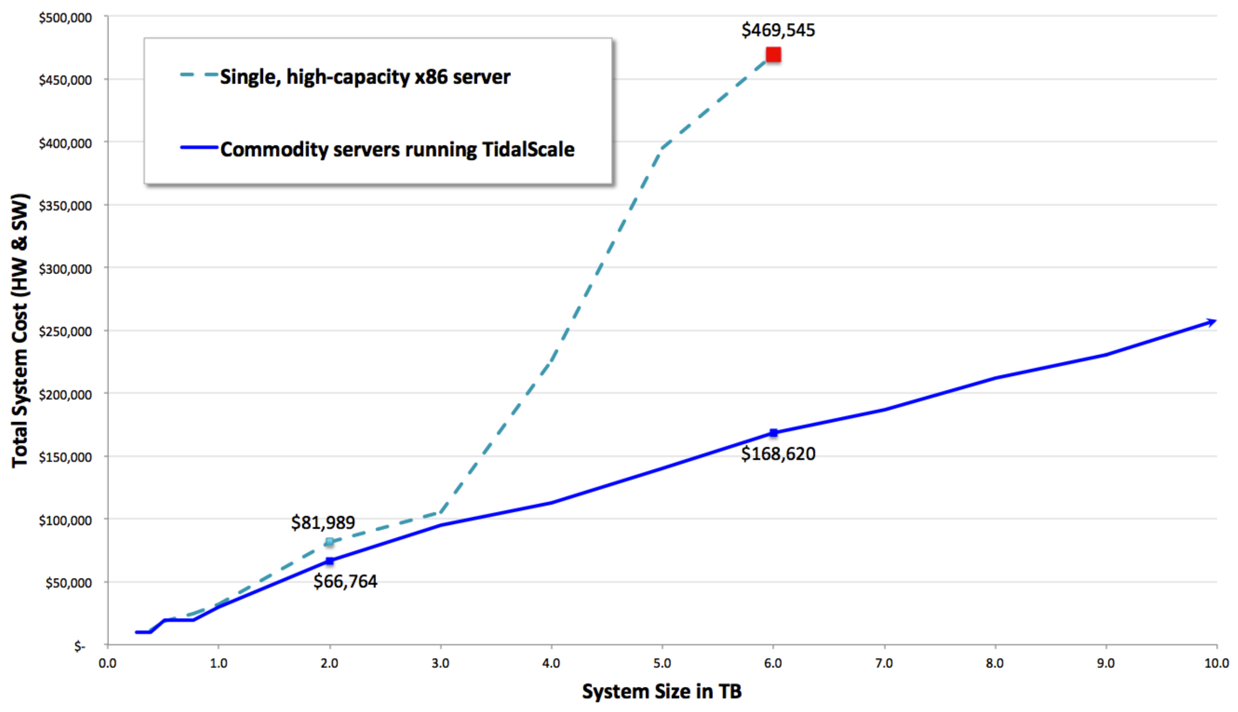

Figure 5: TidalScale Delivers Linear Hardware Costs compared to Traditional Systems

A key observation is that for non-modular systems, cost growth is far from linear, and that commodity-based designs have significantly lower costs for the same capacity.

9 Linear Performance Increase Using Modular Components

Gustafson's Law states that computations involving arbitrarily large data sets in fact can be efficiently parallelized. Gustafson's Law proposes that programmers tend to set the size of problems to use the available equipment to solve problems within a practical fixed time. Therefore, if faster (more parallel) equipment is available, larger problems can be solved in the same time.

It turns out that most people scale systems up not to make a particular problem run faster, but to handle larger and larger problems of the same sort. TidalScale is the first technology to fully implement Gustafson's law in the way it delivers scaled speedup for computations on larger and larger datasets at low incremental cost.

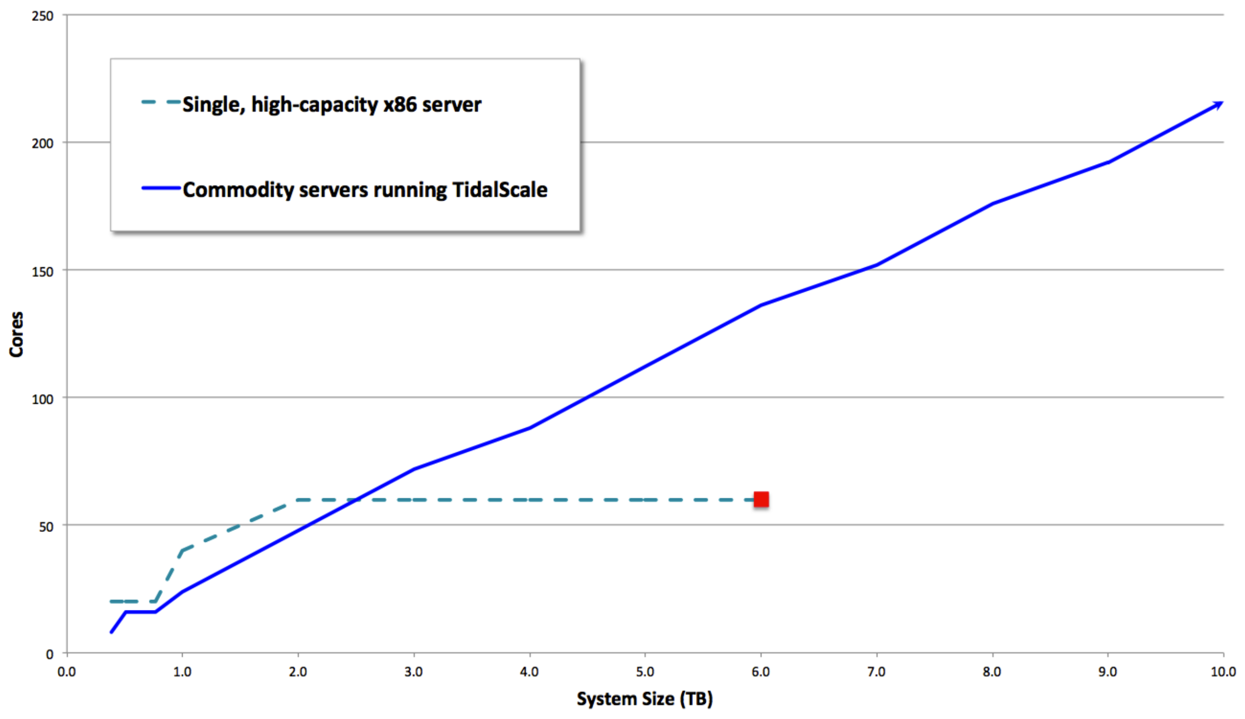

Figure 6: TidalScale Delivers Linear Performance Scaling compared to Traditional Systems

The TidalScale Hyper Kernel shows that when you scale up a problem you **can** use more and more processors.

The combination of linear hardware costs, learning-based optimization and a standards-based deployment process make TidalScale ideal for deployment in enterprise data centers.

10 Anticipated Further Work

All of the aspects of the TidalScale technology are functional today, but we have only begun measuring the various benefits of the TidalScale approach. Current work and expected results include the following:

- Measuring how performance of applications improves as we add nodes to the system to support larger problems. As many applications grow, they are able to use more and more virtual processors to run sub-problems in parallel. [2] We expect that DBMS's and large scale analytics that hold their data in memory will run larger problems without increasing the time to run in some cases. (We call this "avoiding the processor cliff").
- Interactive analytics, where analytic models explore the same model with small variations in filters and parameters, ought to benefit from "learning" the patterns of access by reconfiguring memory and processing based on observed usage patterns should allow much quicker analyst "cycle time" as they explore. We expect to measure this in the context of data mining and graph analytics (using platforms like Neo4J).
- Large-scale in-memory computations that are under pressure to scale to larger datasets or intermediate results, such as EDA, certain kinds of bioinformatics and health, and so forth, are likely to scale well as the problem size scales on TidalScale. For example, semiconductor floor planning problems grow larger with every increase in silicon density. We expect to measure this in the context of specific popular software packages.

[2] As suggested by Gustafson's Law, since the problem size is growing, there is no need for finer-grained parallelism, there is simply more work to do in the same size chunks, so more processors can be productively employed.

- Finally, there are some standard benchmarks that focus on the ability to scale processing. We plan to select application oriented benchmarks where the ability to scale problem size is a key factor. There aren't that many of these benchmarks, but we expect to show linear compute and memory capacity scaling as the problem size increases. We prioritize benchmarks lower than actual user applications, because we focus on immediate practical benefits, rather than artificial test cases.

11 Conclusion

The results of these performance tests demonstrate the three key benefits of the TidalScale architecture:

- **Conquering the Memory Cliff** that applications typically encounter when they exceed the size of reasonably priced currently available systems,
- **Linear hardware costs** & **linear capacity increase** achieved by combining commodity components: scaling the system size by adding components produces a bigger system without slowing the system down, and can be done dynamically as customer requirements dictate.

R is a pretty typical application that shows that, as an application scales in its memory requirements, TidalScale can provide a larger platform to run the application, without modification, with in-memory performance levels.