TritonSort 2011

Alexander Rasmussen
University of California San Diego

Michael Conley
University of California San Diego

George Porter
University of California San Diego

Amin Vahdat
University of California San Diego

{arasmuss,mconley,gmporter,vahdat}@cs.ucsd.edu
http://tritonsort.eng.ucsd.edu/

Abstract

We present TritonSort, a sorting system designed to maximize system resource utilization. We present the results for: 1) Indy GraySort and Daytona GraySort, 2) Indy MinuteSort, and 4) Indy and Daytona 10^{12} JouleSort.

1 Architecture

For a detailed presentation of the TritonSort architecture, please consult [2].

TritonSort is composed of a series of *tasks* that typically consist of some simple processing on a small piece of data. We call the small pieces of data that tasks process *work units*. Tasks are composed together into a directed graph so that workers from one task produce work units for workers in a subsequent task. In the Indy GraySort version of TritonSort, tasks are further subdivided into two *phases*, with a distributed barrier between the first and second phase. In the Daytona GraySort version of TritonSort, tasks are divided into three phases (the same two as with Indy, plus a distributed sampling phase to determine the empirical key distribution of the input data).

A fixed collection of *workers*, each of which is a thread, perform a given task. A single *worker tracker* for each task coordinates the passing of work units to and from the workers performing that task. For brevity, we will name workers based on the task that those workers perform; for example, workers performing the "read" task are simply called "readers".

When a worker is ready to send a work unit to the next stage, it can direct the next stage's tracker either to give the work unit to a particular worker or to give the work unit to the next worker that runs out of work to do.

When a task runs out of work to do, the task's workers shut down. When a task tracker is notified that an upstream task tracker has shut down, it waits until all outstanding work units have been processed and then begins shutting down itself. In this way, the entire graph is shut down at the end of the phase.

Tasks that are sources in the task graph do not receive work units from other tasks, but rather generate work units themselves. In TritonSort, these are tasks that receive from the network or read from a disk. Tasks that are sinks in the task graph do not produce work units for other tasks; these are tasks that write to a disk or transfer data over the network.

1.1 Disks

We subdivide the disks on each machine into equal numbers of input and output disks. In the first phase, input data is read sequentially from the input disks and intermediate data is written to the output disks. In phase two the disks' roles are reversed; intermediate data is read from the output disks and the final output is written to the input disks.

Each disk in the cluster stores tuples whose keys are in a given range. Physical disks are further subdivided into a number of *logical disks*, each of which is a file. Each logical disk on a given physical disk is responsible for its own disjoint sub-range of that physical disk's range. In this way, every tuple can be mapped to a destination logical disk according to that tuple's key.

An operator specifies the number of logical disks for each physical disk before running TritonSort. The number of logical disks per physical disk is determined differently depending on the architecture. For GraySort, the number of logical disks is chosen such that three logical disks for each physical disk can be resident in memory at the same time during phase two; in this way, each worker in phase two will always have a logical disk to process at any given time. For MinuteSort, the number of logical disks is chosen such that all logical disks for a given node can be resident in that node's memory simultaneously; this is necessary because the logical disks are not written to disk before they are sorted.

Benchmark Category	Variant	Data Set Size	# Nodes	Quantity of Interest
GraySort	Indy	100TB	52	6395 seconds (0.938 TB per min.)
GraySort	Daytona	100TB	52	8274 seconds (0.725 TB per min.)
MinuteSort	Indy	1353GB	66	59.2 seconds median
JouleSort	Indy	100TB	52	9704 records/Joule
JouleSort	Daytona	100TB	52	7595 records/Joule

Table 1: Submitted benchmark results

Benchmark Category	Variant	Checksum
GraySort	Indy	`746a51007040ea07ed`
GraySort	Daytona	`746a51007040ea07ed`
MinuteSort	Indy	`193395a80fb0129db`
JouleSort	Indy	`746a51007040ea07ed`
JouleSort	Daytona	`746a51007040ea07ed`

Table 2: Checksums of data sets for each benchmark

2 GraySort Architecture

TritonSort's architecture for the GraySort benchmark aims to sort large datasets by reading and writing each tuple exactly twice, which is the theoretical minimum I/O when the amount of memory in the system is less than the amount of data to be sorted [1]. The task pipeline is designed so that (ideally) all workers constantly have work units to process, thus maximizing utilization of disk I/O, network bandwidth, and CPU processing power.

2.1 Phase Zero: Sampling

For the 'Daytona' variant of GraySort, the input data does not necessarily follow a uniform key distribution. To prevent our system from becoming unbalanced, we need to construct a hash function that will ensure that tuples read from the input are spread across the nodes in our system evenly. Thus before we can begin sorting, we have to sample the input data to construct an empirical hash function based on that input data. The stages that make up phase zero are interconnected as shown in Figure 1.

We chose to use the well known approach of reading a subset of the input data (sampled evenly throughout the entire input) to determine this distribution. This process works as follows. The input data is spread across N nodes. At the start of phase zero, each node opens its input file and reads some number of 80MB buffers' worth of data from each file. The number of buffers used depends on the amount of data sampled from each disk; for our experiments, we chose to sample at least 1 GB of data from each node, which means that we read two buffers from each disk. The keys of the tuples in these buffers are then summarized by recording their values in a fixed-depth, fixed-fanout full partition trie.

We choose a partition trie with a depth of three and a fanout of 256. Every path from the root to a non-root node in the partition trie represents a possible key value; for example, the key whose first three bytes are 234, 119, and 6 would correspond to the node that is the 6^{th} child of the 119^{th} child of the 234^{th} child of the root. Every node in the trie maintains a *sample count* indicating how many tuples were recorded with keys equal to that node's key. Keys that are less than three bytes long will be recorded as samples in interior nodes of the trie; keys that are three bytes long or longer will be recorded at the trie's leaves.

Each reader records its sample values in a separate partition trie. Partition tries from multiple readers are merged together into a single trie. Tries are merged together simply by adding their sample counts at each node.

Once a node's partition tries have been merged into a single trie, that trie is sent to a single designated node, called the coordinator. The coordinator merges the partition tries from each node together into a single partition trie, and then uses this combined partition trie (which contains a summary of sampling information across all nodes) to figure out how to split the key space across partitions such that each node receives a roughly equal division of the input data set.

To do this, the coordinator determines a target partition size, which it calculates as the total number of samples divided by the total number of partitions. The total number of partitions is equal to the number of logical disks per physical disk multiplied by the number of physical disks in the cluster.

Once it has computed the target partition size, the coordinator does a pre-order traversal of the trie. As it does this traversal, it keeps track of the current partition and the number of samples allocated to that partition so far.

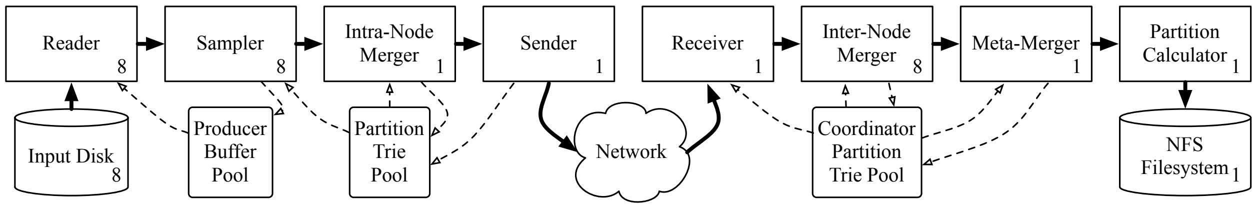


Figure 1: Architecture pipeline for phase Zero

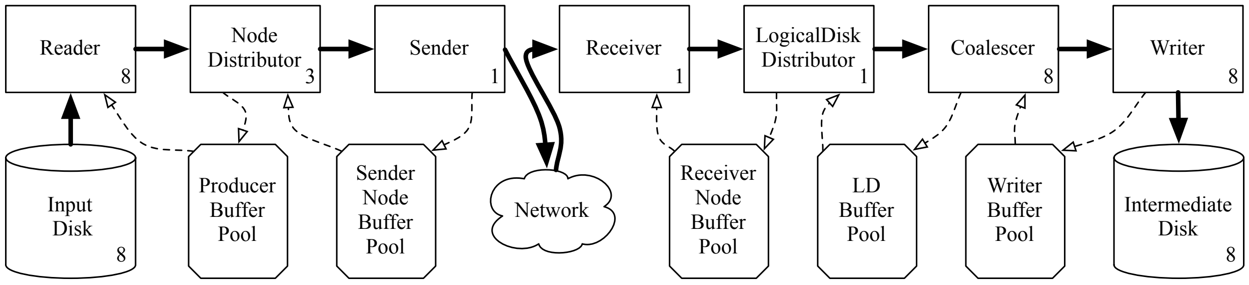


Figure 2: Architecture pipeline for phase one

At each node, it sets that node's partition to the current partition and adds that node's sample count to the total number of samples seen so far. If the number of samples seen so far meets or exceeds the target partition size, the current partition is incremented and the number of samples seen is reset.

We found in practice that this greedy allocation of nodes to partitions could potentially starve later partitions of samples if many previous partitions' sample counts slightly exceeded the target sample count, hence taking more than their fair share of samples. To mitigate this problem, we slightly adjusted the above algorithm to re-adjust the target partition size based on how much "slack" the previous partition had. For example, if the target partition size was 10 and the number of samples greedily allocated to it was 12, the target size of the next partition is set to 8. While this introduces minor imbalances in sample allocation, we found that this produces extremely uniform partitions in practice without starving partitions of tuples.

Once the coordinator has computed the partition assignments for each node in the trie, it writes the trie as a file on an NFS filesystem shared by all the nodes. At the start of phase one (described below), each node will read this trie from NFS and use it to drive its hash function. The trie is used by the hash function by simply traversing the trie based on the first three bytes of the key and returning the partition number at the appropriate node.

In practice, phase zero takes between 15 and 30 seconds to execute at scale.

2.2 Phase One: Distribute

The goal of TritonSort's first phase is to read all tuples from the input disks and transfer each tuple to the appropriate intermediate logical disk. The stages that make up phase one are logically connected to each other as shown in Figure 2. Each stage is responsible for a subset of the overall task as follows:

Reader: reads from an input file into a collection of 80 MB in-memory *producer buffers*. When a producer buffer becomes full, pass it to the next stage. There is one reader for each input disk.

Node Distributor: scans through a producer buffer, hashing each of its tuples to determine that tuple's destination logical disk. For Indy sort, we use a uniform hash function, and for Daytona sort, we the empirical hash function determined in phase zero. Each tuple is copied into an in-memory *node buffer* appropriate to its destination (one per destination host). When a node buffer becomes full, it is passed to the sender phase to be transmitted over the network.

Sender: transmit a node buffer across the network to the appropriate destination. There is a single Send stage that operates in single-threaded mode. Internal to the Send stage is a set of full or partially full node buffers. The Send stage opens a connection to each destination host at the beginning of phase one, thus there are $N - 1$ sockets per sender for a system with N nodes (a socket is not opened to localhost, rather buffers are simply forwarded to the downstream stages of phase one directly). The Sender stage proceeds in a loop, visiting each node buffer and sending as much data as possible with a non-blocking send operation. Note that since data is streaming to all of the downstream nodes, the send stage does not rely on `select()` or `epoll()`, since it is expected that during normal operation all sockets are active.

Connector: establishes connections with remote senders. This stage initializes each of the sockets and is called at the start of phase one.

Receiver: receives node buffers from each source host. There are a set of partially full node buffers, one per source. The receiver loops across those buffers, calling a non-blocking `recv()` call. When a node buffer becomes full, it is passed to the next stage.

LD Distributor: distributes the tuples in a node buffer to the appropriate logical disk. It starts by receiving a node buffer from the receiver stage. It scans this node buffer to extract tuples, and then it hashes the keys in those tuples to assign each tuple to one of several logical disk buffers, or LDBuffers. In our system each physical disk has approximately 300 logical disks, and so a tuple can be hashed to one of approximately 2400 logical disk buffers (the exact number is a configuration parameter).

When a particular LDBuffer gets full, it is added to the back of a linked list, or chain, of LDBuffers. There is one chain for each logical disk on a particular node. Periodically, the LD distributor examines each chain and attempts to send the longest chain it can to the next stage. Details on this stage's construction and its interplay with the writer are discussed in [2].

Coalescer: concatenates the tuples from a chain of LD buffers into a single in-memory buffer. The size of this buffer is configurable and determines the maximum size of the chain that can be emitted by the LD distributor.

Writer: repeatedly write data from the most full circular buffer to its corresponding logical disk. There is one writer per physical intermediate disk.

2.2.1 Phase One Daytona

We achieve generality in our 'Daytona' variant of TritonSort by implementing MapReduce on top of the core TritonSort architecture. Thus Daytona phase one and phase two are slightly different from their Indy counterparts. In particular, phase one encompasses both tuple distribution and the `map()` function. We have one new stage.

Mapper: transforms tuples by applying a user-written `map()` function. The mapper takes the place of the node distributor in Daytona TritonSort. It copies transformed tuples into sender buffers and uses the empirical hash function to determine which node will receive the transformed tuples. GraySort uses an identity `map()` function that leaves tuples unchanged.

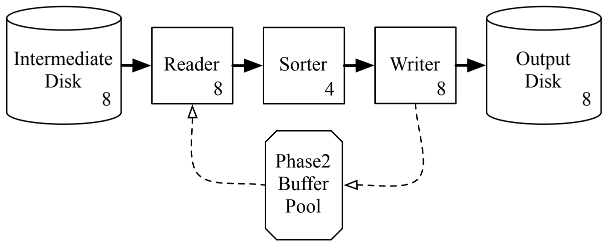


Figure 3: Architecture pipeline for phase two

2.3 Phase Two: Sort

Once each tuple has been transferred to its appropriate logical disk, each logical disk must be sorted. The sorting of logical disks is done with three tasks:

Phase Two Reader: reads an entire logical disk into an 850MB in-memory buffer. There is one phase two reader per intermediate disk.

Sorter: sorts the tuples in a buffer using a variant of radix sort. The number of sorters is variable, and is currently set to half the number of phase two readers.

Phase Two Writer: writes a buffer to a file on the appropriate input disk. There is one phase two writer per input disk.

2.3.1 Phase Two Daytona

Phase two of our Daytona variant implements both sorting and the `reduce()` function. This version of phase two has a larger memory requirement than the Indy version, and so we can only afford about 700MB per logical disk. Consequently, we are forced to use about 100 more logical disks per physical disk than in Indy TritonSort. We have one new stage.

Reducer: combines all values for a given key by applying a user-written `reduce()` function. The reducer operates between the sorter and the writer, so its input buffer is guaranteed to be sorted by key. All values for a given key can be read sequentially from the in-memory buffer. GraySort uses an identity `reduce()` function that emits all values unchanged for any given key.

2.4 Daytona Tuples

The Daytona specification mandates that we must be able to handle tuples of variable size. We follow this guideline by tagging each tuple with metadata specifying its key and value lengths. Any operation on individual tuples must take this meta data into consideration, and this can cause subtle changes to our processing pipeline. For example, our Daytona implementation of radix sort copies sorted tuples to a spare output buffer. We cannot simply swap out-of-order tuples if we cannot assume two tuples are the same length. These spare buffers account for most of the memory differences between the Indy and Daytona variants of phase two.

3 MinuteSort Architecture

For the MinuteSort benchmark, we modify our architecture as follows. In the first phase, as before, we read the input data and spray tuples across machines based on the logical disk to which the tuple maps. However, logical disks are maintained in memory instead of being written to disk immediately.

In phase two (once all input tuples have been transferred to their appropriate logical disks), the in-memory logical disks are directly passed to workers that sort them. These sorters in turn pass sorted logical disks to writers to be written to disk. Hence, logical disks are still written to disk but are not written until after they have been sorted.

4 Environment

Our testbed consisted of 52 HP ProLiant DL380 G6 servers, although we use different numbers of servers for different benchmarks. Each server has two quad-core Intel Xeon E5520 processors, clocked at 2.27 GHz, and 24 GB of RAM. Each server also hosts 16 2.5-inch 500 GB, 7200 RPM SATA hard drives. 52 of the machines use HP Seagate MM0500EANCR drives that are enterprise-grade and therefore have a much higher reliability. The remaining machines use Seagate Momentus 7200.4 drives, which are consumer-grade.

Each machine is equipped with a 10G-PCIE2-8B2-2S NIC Myricom 10Gbps network card. All the machines in our testbed are inter-connected via a Cisco Nexus 5096 switch, which provides 10 Gbps connectivity between all pairs.

All machines are running version 2.6.35.1 of the Linux operating system. Each hard drive is configured with a single XFS partition. Each XFS partition is configured with a single allocation group to prevent file fragmentation across allocation groups, and is mounted with the `noatime`, `attr2`, `nobarrier`, and `noquota` flags set. Each server has two HP P410 drive controllers with 512MB on-board cache. The servers run Linux 2.6.35.1, and our implementation of TritonSort is written in C++.

5 Experiment Setup

TritonSort is bootstrapped by a pair of shell scripts. The user executes a central shell script (called the coordinator) that is responsible for starting TritonSort on all cluster nodes. This script is assumed to run on the same subnet as the cluster nodes. The coordinator starts a script called a monitor as a daemon on each cluster node. After a monitor finishes initializing its internal state, it opens a TCP pipe to the coordinator and announces that the node it is monitoring is ready to run. The monitor then waits for a UDP broadcast packet. Once the coordinator has established connections to all monitors and all monitors are ready, the coordinator sends a UDP broadcast indicating that the experiment is ready to begin. Once a monitor receives the UDP broadcast, it starts the TritonSort instance for its node immediately.

Each monitor checks the status of its TritonSort instance every tenth of a second. While its TritonSort instance is still running, the monitor transmits a keep-alive message to the coordinator via its established TCP pipe every second. Once the TritonSort instance has finished, the monitor notifies the coordinator that it is done. The coordinator exits when it has received notifications from each monitor in this way.

The coordinator and monitor scripts are not strictly part of TritonSort, and were built primarily for experimental convenience. For our MinuteSort runs, where strict timing is essential, we measure the elapsed time of the sort as the time between when the coordinator sends the UDP broadcast to start the experiment and when the last monitor reports to the coordinator that it is finished. This ensures that the measured time encompasses the starting up and shutting down of all TritonSort instances (since the monitor starts TritonSort after receiving the UDP broadcast and TritonSort stops before the monitor notifies the coordinator that it is finished).

For GraySort, we choose to underestimate our performance by a few seconds to make logging simple and measure the time between when the coordinator starts and when the coordinator exits. This ensures that the total elapsed time begins slightly before the first TritonSort process starts and ends slightly after the last TritonSort process exits. In practice, this overestimate is less than 30 seconds.

6 JouleSort Measurement Methodology

For the 100TB Daytona JouleSort benchmark, we used 52 nodes and one experiment head node, all of which are HP DL380G6s.

When measuring the energy consumed by the testbed during the run, we measure the combined energy used by the experiment nodes, the experiment head node, and the 10Gbps switch that connects the machines together.

6.1 Measuring the switch

To measure the energy used by our Cisco 5596UP datacenter switch, we plugged the switch into an Avocent PM

3000V PDU during our sorting runs. The PDU tracks maximum, minimum and "present" power draw on a per-port basis. To determine the total amount of energy used by the switch throughout the run, we multiplied the maximum power draw from the switch (in watts) as measured by the PDU by the duration of the run in seconds. This overestimates the energy used by the switch, but makes our calculations easier. The power drawn by the switch measured in this way is 566 watts.

6.2 Measuring the nodes

We measured the power consumed by the cluster machines (both experimental nodes and head node) using two different power meters. The first meter is available on each machine, but does not meet the accuracy standards required by the benchmark's guidelines. The second can only be attached to one machine at a time, but meets the required accuracy standards. As we will show in later sections, the two meters' power measurements are very similar. We describe each meter and the methodology for measuring power from it below.

One danger when measuring power on many machines is that the clocks on those machines may become out of sync and cause the aggregate power measurements from multiple nodes (that should be correlated by time but aren't) to be inaccurate. To prevent this from being a problem, we issue all power meter queries from a single machine and timestamp the power meter measurements when they are received. Further, we issue the measurements from the experiment head node so that the timestamps recorded when the sort starts and stops (see above) are taken from the same clock as the timestamps for the power measurements.

All power measurements are performed by simple Python scripts. The content of the script varies depending on the power monitoring system being queried. In cases where multiple machines are to be monitored at once, the script spawns a thread per monitored machine and each thread runs independently. We start the power monitor scripts manually several minutes before starting the sort run to allow them to "warm up" and make sure everything is working properly, and stop them manually several minutes after the sort run ends. The scripts dump power measurements to a file as they run, and these files are analyzed after the run to determine total energy usage.

6.2.1 HP ILO Power Meters

The first meter we used was the power measurement subsystem of HP's Integrated Lights-Out (ILO) management tool. Each of our DL380G6 machines comes equipped with an on-board service processor running version 1.82 of ILO2.

We query the ILO power meter using the Remote Board Insight Command Language (RIBCL). RIBCL allows operators to issue commands to ILO by sending an XML document to the ILO system over an SSL-encrypted TCP session and receive an XML response. The power monitoring script repeatedly opens an SSL connection to the ILO system, issues a power monitoring command, retrieves a response, and closes the connection.

RIBCL's power measurement reports four numbers: maximum, minimum and average power over the past 24 hours, and "present" power, which measures the number of watts for the most recent 0.5 second sample. We use present power as our power measurement for each sample.

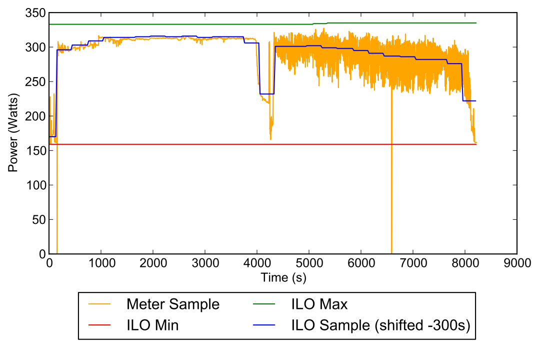
Unfortunately, RIBCL requires that only a single XML document "command" be sent per connection. We found in practice that we could not reliably issue RIBCL commands to the ILO system more than once every 15 seconds because the on-board service processor is quite slow and the high overhead of establishing an SSL session must be incurred once per measurement.

6.2.2 WattsUp? Power Meter

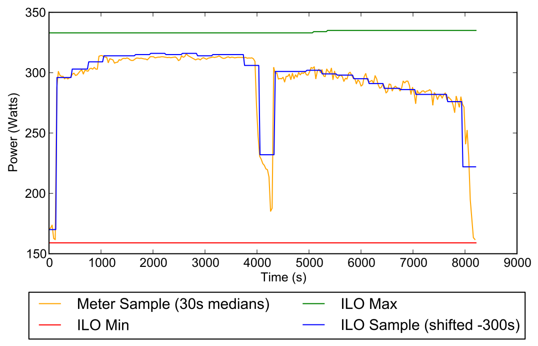
To provide once-per-second measurements of our machines' power consumption, we attached a power meter that could provide once-per-second power measurements to a representative node in the cluster. The particular meter that we used was the IEC 320 universal outlet (UO) version of the WattsUp? Pro/ES/.Net power monitor. We refer to this meter as the WattsUp meter for brevity for the remainder of the text. We chose this meter because of its ready availability and known reliability; several other research projects at UCSD have used this meter to measure server power successfully.

The WattsUp meter has a simple serial-over-USB interface. The client opens a TCP connection to the meter and sends the meter a request for data and a data collection interval. The meter responds by sending the requested data once per interval until it's told to stop or the client closes the TCP connection. Our power measurement script sends the meter a request for power information at an interval of one second. The script then receives and parses the response (by issuing a blocking read call to the socket, which consistently unblocks with a new response once per second) and appends the parsed response to a file.

During the first four runs of each benchmark type, we used the WattsUp meter to measure the power on a random experiment node. On the fifth run, we used it to measure the experimental head node. Since the head node is not doing anything particularly intensive (mon-



(a) Raw WattsUp meter data



(b) Median of each 30 seconds of WattsUp data

Figure 4: Power consumed by a representative node during a Daytona JouleSort run

itoring power on each machine and recording experiment time), we found that its power consumption was relatively low. Through measurements on both types of power meters, we found that the average draw for the head node was 134 watts with a deviation of about 2 watts. Because of this, we assume that the experiment head node's power draw is a constant 134 watts for the duration of the sort run.

6.2.3 Resolving Discrepancies Between Meters

We found that the power measured by the ILO system lags that measured by the WattsUp meter by exactly five minutes. Figure 4 shows both the maximum, minimum and "present" power reported by the ILO and the power reported by the WattsUp meter during a Daytona JouleSort run, with the ILO measurements appropriately time-shifted. When calculating power with the ILO meters, we time-shifted all samples by 5 minutes to compensate for this observation and allowed the power collection scripts to run for several minutes after the sort run finished to collect sufficient additional samples to cover the entirety of the sort run.

In these runs you can see that there is a sharp reduction in power usage about halfway through the sort run. This is a result of the barrier between phases one and two. Due to the natural variation in node performance, some nodes finish phase one earlier than others, and so their power usage is reduced. However, none of the nodes can start phase two until all nodes are done with phase one, which results in the gap visible in Figure 4.

The WattsUp meter's data is more variable during phase two; we suspect that this is due to the fact that the CPU is far more active during phase two than it is during the other phases. However, if we look at the median power reported by the WattsUp meter during each 30 second interval throughout the run, we notice that the WattsUp meter's measurements track the ILO's measurements quite closely.

6.3 Calculating Energy

To estimate energy used by the experiment head node and the switch, we multiply their estimated instantaneous power draws (134 watts and 566 watts, respectively) by the duration of the sort run in seconds. Call the energy used by the head node and the switch E_{head} and E_{switch} respectively.

All power measurements reported by our meters are reported in watts. To obtain an energy measurement from this collection of instantaneous power measurements, we start by filtering the set of measurements so that we only consider those measurements that were taken on or after the sort run's start timestamp and on or before the sort run's end timestamp. The subsequent power calculation varies depending on the meter being used. We refer to the energy used by the experiment nodes as E_{nodes}.

When calculating power based on measurements gathered from the ILO meters, we start by sorting measurements in ascending order by timestamp. We then compute the total energy for a node in the following way. For each measurement (W_i, T_i), we consider the previous measurement (W_{i-1}, T_{i-1}) and add $W_i * (T_i - T_{i-1})$ to the total energy. In cases where the measurement abuts the start or end of the run, we use the start and end timestamps of the sort run as T_i and T_{i-1} as appropriate to "fill in the gaps" at the beginning and end of the run. We compute the total energy for each node in this way and sum the energy from each node to compute E_{nodes}.

When using the WattsUp meter, we have once-per-second measurements and can produce power estimates in line with the sort benchmark guidelines. To do this, we compute that average (mean) power for the representative node. For the first four trials (where an experimental node is being measured), this data is derived by taking the average of all measurements taken from the WattsUp meter during the sort run. For the fifth trial (where the

Benchmark	Trial	Avg. Server Power
Indy	1	287 Watts
Indy	2	285 Watts
Indy	3	309 Watts
Indy	4	297 Watts
Daytona	1	290 Watts
Daytona	2	285 Watts
Daytona	3	293 Watts
Daytona	4	306 Watts

Table 3: Average power consumed by a node throughout a sort benchmark run for the first four trials of our JouleSort experiments

head node is being measured), the average power is assumed to be the average of the previous four trials.

We believe that this assumption is reasonable because the average power consumed by a node does not vary much; we provide the average power drawn by a node in the first four trials in Table 3. The standard deviation for the first four measurements is 11 watts for Indy and 9 watts for Daytona, 3.7% and 3.0% of the mean, respectively.

Once we have computed the average power for a representative node, we multiply that average power by the length of the run in seconds to yield the total energy consumption for a node, and multiply that number by the number of nodes (52 in our experiments) to yield E_{nodes}.

Once we have computed E_{nodes}, E_{head} and E_{switch}, we compute total energy E_{tot} as $E_{nodes} + E_{head} + E_{switch}$.

We present the results of the measurements obtained in this way in Section 7.1.

7 Evaluation

For all reported results, we calculated input and output checksums and verified that the input and the output checksums matched. We never encountered any duplicates in any of our data sets.

Our results are summarized in Table 1.

7.1 JouleSort

We ran five trials each of 100TB Indy and Daytona JouleSort. The raw energy measurements for these trials are given in Table 4 and metadata about the total energy measurements is given in Table 5.

7.1.1 Deriving Standard Deviation and Standard Error

When calculating standard deviation and standard error in Table 5, we assume that the WattsUp meters are accurate to ±2% and the ILO meters are accurate to ±10%. We were unable to obtain any data about the accuracy of Avocent's PDUs, and so we assume somewhat pessimistically that the PDUs are accurate to ±5%.

We obtained standard deviation in the following way. First, we derived the mean power drawn by each server, the head node, and the switch. When using the WattsUp meter's measurements, we simply used the mean of all power measurements logged by the meter. When using the ILO meter's measurements, we derived the mean power draw for a node by dividing the total amount of energy consumed by the node by the trial's runtime. Call the mean power produced by server X P_{N_X}, the mean power produced by the head node P_H and the mean power produced by the switch P_S.

We then multiplied each mean power value by its respective meter's accuracy to yield the uncertainty $U(P_{N_X})$, $U(P_H)$ and $U(P_S)$ of each power measurement. We then multiplied these uncertainties by the trial runtime to yield the uncertainty $U(E_{N_X})$, $U(E_H)$ and $U(E_S)$ of each energy measurement. Once these values were derived, we used error propagation to yield the total energy measurement uncertainty for the trial using the following formula:

$$U(E_{trial}) = \sqrt{(\sum_{X=0}^{51} U(E_{N_X})^2) + U(E_H)^2 + U(E_S)^2}$$

Once $U(E_{trial})$ was calculated for each trial, we performed a further round of error propagation across trials to yield total uncertainty, i.e. standard deviation.

$$U(E_{total}) = \sqrt{\sum_{T=1}^{5} U(E_{trialT})^2}$$

Standard error is derived by dividing standard deviation by $\sqrt{5}$.

Since the measurements derived by the WattsUp meter comply with the guidelines for the sort benchmark, we report those numbers. Indy TritonSort sorted an average of 9,704 records per Joule. Daytona TritonSort sorted an average of 7,595 records per Joule.

7.2 Minute Sort

We ran TritonSort in its MinuteSort configuration on 66 nodes with 20.5 GB per node for a total of 1353 GB of data. We performed 15 consecutive trials. For these trials, TritonSort's median elapsed time was 59.2 seconds, with a maximum time of 61.7 seconds, a minimum time of 57.7 seconds, and an average time of 59.2 seconds. All times were rounded to the nearest tenth of a second. Only 3 of the 15 consecutive trials had completion times longer than 60 seconds.

		Energy (Joules)		Records per Joule	
Benchmark	**Trial**	WattsUp	ILO	WattsUp	ILO
Indy	1	103,180,896	108,054,648	9,692	9,255
Indy	2	99,312,480	105,333,939	10,069	9,494
Indy	3	109,495,040	105,425,639	9,133	9,485
Indy	4	102,724,272	105,523,992	9,735	9,477
Indy	5	101,108,112	103,656,684	9,890	9,647
Daytona	1	129,774,720	136,098,976	7,706	7,348
Daytona	2	127,434,720	135,998,793	7,847	7,353
Daytona	3	132,077,568	136,851,456	7,571	7,307
Daytona	4	137,082,224	136,259,721	7,295	7,339
Daytona	5	132,380,352	135,332,800	7,554	7,389

Table 4: Total energy measured for each 100TB trial by both WattsUp and ILO meters

		Energy (Joules)			
Benchmark	**Meter**	Median	Mean (Average)	Std. Dev.	Std. Err
Indy	ILO	105,425,639	105,598,980	3,170,018	1,417,675
Indy	WattsUp	102,724,272	103,164,160	736,610	329,422
Daytona	ILO	136,098,976	136,108,349	4,086,316	1,827,456
Daytona	WattsUp	132,077,568	131,749,917	941,356	420,987

Table 5: Statistics for the energy measurements presented in Table 4

8 Acknowledgements

We'd like to thank Harsha V. Madhyastha, Radhika Niranjan Mysore, and Alexander Pucher for their contributions on the original TritonSort codebase and entry into the 2010 sortbenchmark.org competition.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 1988.

[2] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A Balanced, Large-Scale Sorting System. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2011.