# Evaluation of MapReduce-Style Computation on a Cluster of Arduinos

## Abstract

Not really. This work involved hands-on implementation. See Figure 2.

## 1. Introduction

With the advancement of small, low-power sensor and processing technology, tiny sensor motes with its own computation and sensor array are being constructed to collect and process data from various sources spanning from human to natural activities. Such motes form the basis of the so-called Internet-of-Things (IoT), which collect and process enormous information from various 'things' around us.

The Arduino [1] is a standard architecture for rapid development of such devices. It consists of an 8-bit AVR microcontroller with a Arduino bootloader. The programmer can write C code on top of the bootloader and Arduino's set of standard libraries. It also defines a standardized set of general purpose I/O pins including analog and pulse width modulation (PWM) enabled digital pins. This enables easy development and integration of expansion cards, or "shields" in the Arduino lingo, using standard interfaces such as I2C or SPI. There are hundreds of such shields developed and in production, spanning from various sensors to SD card, Ethernet, or Bluetooth interfaces.

On the other end of the computation spectrum, the vast amount of data that is collected using such devices are generally collected at a cluster of servers in a datacenter to be processed and analyzed for meaningful information. Since many datasets of interest exceed the size that can be managed by a single machine, a distributed processing platform is usually used to ease the development overhead of such analytics software. One of the most popular of such platforms is Hadoop [4], based on the MapReduce processing paradigm.

Because the amount of raw data collected at each sensor nodes is immense, it is valuable to preprocess it and reduce the total size of data that needs to be transferred to the datacenter. However, the computation capacity of each sensor node is extremely lacking, both in processor performance and the on-chip DRAM capacity each sensor node has. Under such restrictions, a MapReduce-like distributed processing platform between a cluster of sensor nodes may be of value.

This paper explores the usefulness of a MapReduce style method of distributed computation on a cluster of Arduinos, using a simple word count workload as the benchmark. We have discovered that the performance of MapReduce style computation incurs some amount of overhead, but is scalable on the small number of nodes we have experimented with, under certain situations. In order to make MapReduce on Arduinos feasible, there needs to be some improvements to the Arduino platform, including network DMA and faster storage access.

The rest of the paper is organized as follows: Section 2 introduces some detailed background information on Arduinos and MapReduce. Section 3 describes the architecture and implementation of the evaluation platform we constructed. Section 4 presents the results and analysis we obtained from the evaluation system.

## 2. Background

Arduinos and AVR

Arduino devices come in multiple form factors, with different AVR chips at their heart. The most basic model, the Arduino Uno, comes with a ATmega328P running at 16MHz, which includes 2KB of on-chip SRAM for processing. The Arduino Mega, which comes with a ATmega1280 running at 16MHz, includes 8KB of on-chip SRAM, also with a much larger number of general purpose pins.

The arduino platform provides multiple ready-to-use interfaces for communicating with various peripherals, and each other. Many peripheral devices can be connected to an Arduino device via the SPI (Serial Peripheral Interface) interface. Arduinos have pins dedicated to SPI, as well as a hardware controller that removes the burden of SPI management from the AVR microcontroller. Arduino also provides dedicated pins for the I2C (Inter-Integrated Circuit) bus interface. Not only can multiple peripherals be wired onto the same bus, other Arduinos can be wired into the I2C bus as slaves for inter-arduino communication.

Data that is collected via various sensors need to be written temporarily to a non-volatile storage. The most popular medium for this is using a micro-SD card plugged into an SD card expansion board. This way, Arduinos can have gigabytes of cheap storage running on a small power budget. However, the software SD card controller library that comes with the Arduino does not have very high performance, especially write performance.

MapReduce [2] is a programming model for distributed, or parallel processing of data, inspired by the Map and Reduce functionalities often used in functional languages. In order to process data using MapReduce, the programmer provides two programs, *Map* and *Reduce*. The MapReduce implementation deploys multiple instances of these programs to its compute nodes and manages marshalling of data between them. Most major distributed implementations of MapReduce also handles issues like fault tolerance. The MapReduce abstraction allows simple development of high-

DOI: http://dx.doi.org/10.1145/(to come)

performance distributed programs. Various MapReduce platforms exist, from massively distributed ones [4] to shared multicore systems [3].

User programs in MapReduce are developed in two parts, Map and Reduce. The Map function is called repeatedly with a key-value pair as argument, and emits a list of key-value pairs in a possibly different domain. The resulting stream of key-value pairs are partitioned across reducer instances, and sorted by the MapReduce platform to group pairs by key values. The programmer often has to provide a comparator function in order to sort custom key values. The Reduce function is called repeatedly with two arguments: a key and a list of values associated with that key. One of the major functions that a MapReduce platform must perform is the sorting of key-value pairs that must happen between the mapping and reducing phase. Because the amount of intermediate data between the mapping and reducing phases is quite large, they are usually logged to secondary storage to be processed.

There are multiple system component at play during the process of MapReduce execution, and many of them may end up being the bottleneck depending on the architectural characteristic of the system and the characteristic of the workload. There has been many attempts to analyze these bottlenecks, and to improve performance my improving the network, storage, or computation components.

## 3. Architecture and Implementation

### 3.1 Architectural Components

There are multiple system components involved in performing MapReduce-like computation on a cluster of Arduinos. The most prominent ones effecting performance are: Computation, Memory, Storage and Network, which almost make up the entirety of the cluster system. The computation performance of the AVR chip running at 16MHz is not very high, but that's why we're looking into clustering. The on-chip memory is also very small, in the range of 2 8KB, also why we're looking into clustering. Larger system memory would be helpful, but the Arduino doesn't really expose an interface that can support low-latency memory access at 16MHz.

Storage is usually added in the form of an SD-card reader connected to SPI. This is very useful because gigabytes of data can be logged to an off-the-shelf SD-card with a familiar FAT organization. However, the Arduino interface to the SD card is not particularly high performance. While reads achieved a bit more than 30KB/s, writes are much slower, achieving around 0.1KB/s. Such low performance effectively rules out using secondary storage to manage intermediate data between map and reduce phases.

We used I2C between Arduino nodes to network them together. Using the default library, I2C between nodes perform at aboutn 100KB/s. The library can be easily modified to perform at 400KB/s. Compared to the rest of the system, network performance is actually not bad. However, because copying data via I2C needs to be done byte-by-byte by the CPU instead of, say, a DMA controller, the system cannot do useful work between the network is busy.

Table 1 shows the performance of various components.

| Component | Performance |
|---|---|
| Storage Read | 30KB/s |
| Storage Write | 0.1KB/s |
| I2C | 100KB/s |

**Table 1.** System Component Performance

### 3.2 Architecture

Due to the practical unavailability of secondary storage, the cluster can only deal with the size of problems that can fit on the collective main memory of the cluster. More precisely, the intermediate data output of the mappers need to fit inside the collective memory of the cluster. Once the intermediate data is generated, it can be sorted in-memory before fed to the reducers. The resulting value from the reducer can be directly output regardless of size.

The limitation of problem size practically renders this system useless, but we'll keep going just in case.

Figure 1 describes the architecture of the experimental setup. Three Arduinos are involved because that's how many we had. The SD card is connected via SPI, and all Arduinos are networked via I2C.

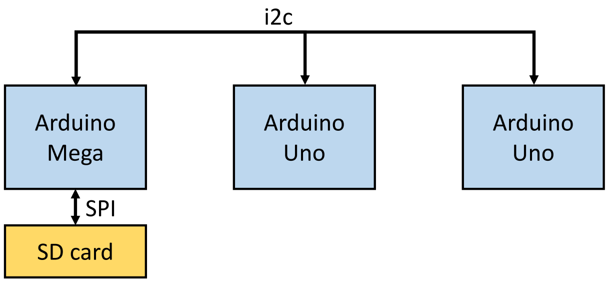

**Figure 1.** Example architecture of an Arduino MapReduce cluster

Figure 2 shows the actual implementation. The penny is for scale.

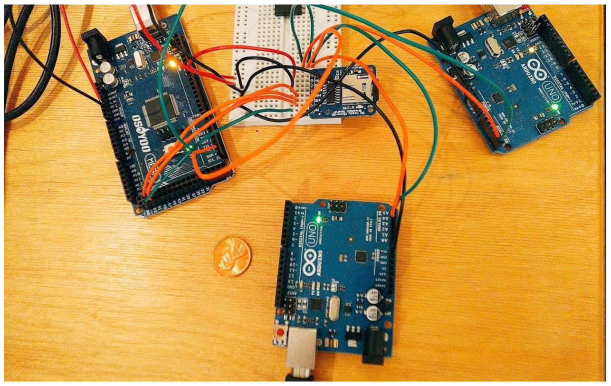

**Figure 2.** Physical appearance of the implementation

## 4. Evaluation

We used a word-counting example, which is one of the canonical applications for MapReduce platform evaluation. The application reads in a file from the SD card and feeds it into the mapper until the intermediate values fill up all allocated buffers, or until the file is completely read. Once the intermediate key-value list is generated, it is sorted in-memory and then fed to the reducers. Because the reducers in wordcount is extremely simple, there was no performance advantage in shuffling the intermediate data. Instead, all data is collected to the main node where they are merged and fed a single reducer instance.

Figure 3 shows the computation flow of word-count. In the diagram, the "Mapper" includes the in-memory sorting operation.

Figure 4 show the performance difference of using various clusters of Arduinos. Table 2 shows the configurations of the various clusters. Note that in the case of Uno+Uno, the Mega is still managing data transfer between the storage and two Unos. The reducer is also still running on the Mega. The amount of data processed was small enough to fit in all configurations of the system.

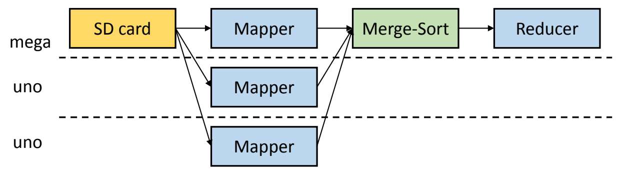

**Figure 3.** Computation flow of MapReduce

| Name | Configuration |
|---|---|
| Mega | 1 Mega (No MapReduce) |
| Mega+1 | 1 Mega + 1 Uno |
| Mega+2 | 1 Mega + 2 Uno |
| Uno+Uno | 2 Uno |

**Table 2.** Cluster Configuration

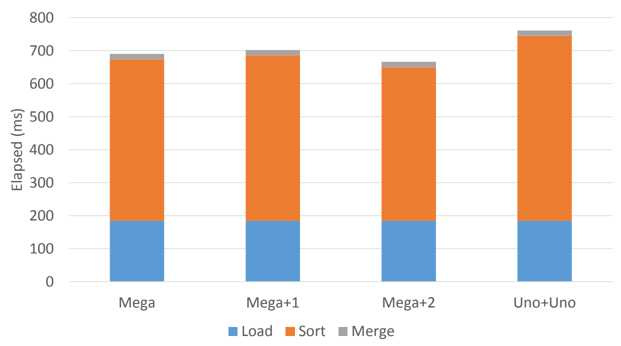

**Figure 4.** Evaluation results

Because additional code has to be executed to route the data between computing nodes, using a MapReduce style distribution was not worthwhile with only one additional node. However, the performance increases with two additional nodes. When using two remote nodes, the performance was worse, because it suffers from the added network overhead of two remote nodes. It is good proof that the network overhead is actually rather high.

I can claim that This performance ”scales”, because three nodes are faster than two. It would be interesting to see if four nodes did any better, which I assume it will, but we only had three Arduinos.

## 5. Conclusion and Future Work

As it stands, running MapReduce on Arduinos are not very fruitful. But the solution appears to be simple. The performance overhead of the network seems to be damaging, not because the network bandwidth is slow, but because it eats into available computation overhead. The solution would be to add a DMA controller for the network.

Another possible improvement would be a faster SD card interface. Even if a separate peripheral was able to provide I/O into the SD card at I2C or SPI’s maximum bandwidth, the storage can be used as an intermediate storage, vastly improving the size of the problems it can deal with.

## References

[1] *Arduino*, Accessed Feb 8, 2017. URL https://www.arduino.cc.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[3] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 9–16. ACM, 2011.

[4] T. White. *Hadoop: The definitive guide*. ” O’Reilly Media, Inc.”, 2012.