VisHive: Creating Ad-hoc Computational Clusters using Mobile Devices in Web-based Visualization

Shivalik Sen*
Birla Institute of Technology and Science
Goa, India

Sriram Karthik Badam†
University of Maryland
College Park, MD, USA

Niklas Elmqvist‡
University of Maryland
College Park, MD, USA

Abstract—Current web-based visualizations are designed for single computers and cannot make use of multiple devices, even when a user has access to not only a laptop, but also a tablet and a smartphone. We present VisHive, a JavaScript toolkit for constructing web-based visualization applications that can transparently connect multiple devices—called cells—into an ad-hoc cluster—called a hive—for local computation. Cells are organized into a master-slave architecture, where the master provides the visual interface to the user and controls the slaves, and the slaves mainly perform computation. VisHive is built entirely using current web technologies, runs in the native browser of each cell device, and requires no specific download and install on the involved devices. We demonstrate VisHive using a time-series prediction visual analytics tool that utilizes connected slave cells to continuously perform additional predictions while the master awaits user input.

Index Terms—Parallel computing, web-based visualization, junkyard cluster, ad-hoc clusters, JavaScript toolkit.

1 INTRODUCTION

Modern browsers have a lot to offer visualization developers, including advanced accelerated graphics; support for multi-touch, gesture-based, and pen-based computing; and seamless integration with the entire web ecosystem, including remote databases, sophisticated web services, and online geographical map systems, to name just a few examples. Most importantly, the web browser is now ubiquitous on all devices, from laptop to smartphone, tablet to smartwatch, and require no specific download or install to run sophisticated applications. For these reasons, it is not surprising that the web is quickly becoming one of the most popular target platforms for visualization. Accordingly, a host of toolkits, frameworks, and middlewares have recently become available for visualization development, such as D3 [6] for creating visual representations of data using declarative syntax, PolyChrome [2] for duplicating visualizations across multiple devices, and VisDock [11] for providing cross-cutting interaction techniques to visualizations in a standardized tooldock similar to Adobe Photoshop.

However, the browser is still not an ideal computational environment for executing the kind of heavy algorithms—such as cluster analysis, graph layout, or probabilistic topic modeling—that many visualization and, in particular, visual analytics (VA) applications may require. One reason for this lies with JavaScript (the programming language of the web browser) itself: it is an interpreted and weakly/dynamically typed scripting language that was never intended for high-performance computation. While it is possible to write high-performance code using JavaScript, this requires special care and in-depth knowledge to avoid the many pitfalls inherent in common
JavaScript coding idioms. JavaScript also long had no support for mutithreading, which is critical for concurrent implementations of many popular algorithms. As a result, JavaScript libraries for scientific computing and other heavy computational domains were basically inconceivable. Fortunately, the HTML5 standard brings the Web Worker API\(^1\), which allows for splitting computation across multiple concurrent threads of execution. This API, as well as the continuous improvement of browser JavaScript interpreters such as Google’s V8 engine (which is used in Google Chrome), has belatedly led to JavaScript and the browser now also becoming a platform for scientific computing.

One major gap in web-based visualization development remains: parallelization of computation across multiple devices. A common use-case of a web-based visualizations involves a user that has access to multiple devices in their immediate physical surroundings. For example, if the user is accessing the visualization using a laptop, they may also have a smartphone in their pocket, a tablet in their backpack, and a desktop computer under their office desk. While offloading computation to a server-side or cloud-based component is certainly possible, it would make a lot of sense if the user was also able to fire up the additional devices and use them for serendipitously offloading any heavy computation required by the visualization tool, such as a sentiment analysis, a complex graph layout, or a dimensionality reduction algorithm. Compared to the server-based or cloud-based solution, this “local cloud” of co-located physical devices brings benefits to both end-users and developers. More specifically, the end-user can avoid any mobile network fees while minimizing latency by confining the communication to the local Wi-Fi network, whereas the visualization developer can implement concurrent computation using JavaScript in the web-based visualization application itself and without having to worry about creating separate server-side software for this purpose.

In this paper, we propose VisHive, a JavaScript toolkit for creating this type of ad-hoc, opportunistic clusters consisting of local, networked devices that is directly integrated in a web application, thus requiring no server-side components or client-side download and install. Using the VisHive toolkit is simple for developers: they simply create a single web application using the library that also includes the computational code. Multiple devices—each called a cell—join a session through a simple discovery mechanism to form a cluster—called a hive—that can be dynamically reconfigured as cells join and leave. The user manually designates one cell as a master—typically the device that the user launches the visualization on first, though this can change over the duration of a session—and the others become slaves. The master, in addition to also participating in the computation, is responsible for partitioning the problem (or dataset) into smaller chunks, assigning the chunks to cells as they become available (or recovering a lost chunk if the cell is lost from the hive), and finally recombinating the results and presenting them to the user in the visualization on the master device.

The VisHive toolkit is in no way a replacement to existing software architectures, frameworks, or middleware for scientific computing, machine learning, or other computational engineering applications. In particular, it does not replace server-based, cloud-based, or cluster-based computational frameworks that are already being used to power web applications and visualizations today. Rather, VisHive provides an entirely new opportunity to cheaply, conveniently, and easily build visualizations that also integrate opportunistic offloading of computation to networked devices that also happen to be available in the same physical environment of the device being used to display a visualization. To facilitate this mission, our implementation of VisHive is built in JavaScript and uses standard web technologies, such as WebRTC, WebSockets, and JQuery, thus requiring no separate download or installation for each participating cell. In other words, the main objective of the VisHive toolkit is to provide a “poor man’s cluster” solution inside the very same web framework that developers routinely use when building web-based visualizations, while at the same time eliminating the need for server-side components.

To demonstrate the utility of the VisHive toolkit, we present StockHive, a time-series prediction visual analytics tool for stock market data. StockHive supports a mixed-initiative interaction model where the tool provides predictions of future changes for different stocks, and the analyst can choose to either accept a specific prediction or make a new one. The tool then responds by computing new predictions based on the selected one. Temporal as well as inter-stock predictions generate a prediction path into the future for each stock, which is a computationally expensive operation performed using neural networks. Since user choice will cause predictions to become a branching decision tree of alternative possibilities, StockHive offloads the predictions in a breadth-first way to available VisHive cells. While waiting for user input, the tool continuously adds new predictions as the cells finish their computation and return with a result. When the user makes a choice, the prediction tree is invalidated and a new one is created and iteratively refined. We also show performance measurements for VisHive for various combinations of devices, such as a laptop, a tablet, and a smartphone device. In summary, our performance results show a significant completion time improvement using the VisHive toolkit compared to single-thread computation.

The remainder of this paper is structured as follows: We first review the relevant background on distributed computing and web-based visualization, as well as the application to time-series prediction. We then discuss our design framework for ad-hoc clusters in web-based visualization. We present the VisHive toolkit, and then exemplify its use with the StockHive visual analytics tool. This is followed by a performance evaluation. We close the paper with a discussion, our conclusions, and our plans for future work.

2 BACKGROUND

In this section, we discuss existing literature on distributed computing, especially on mobile devices, web-based visualization methods, and visual analytics for big data. We focus on the specific components relevant to the design and construction of VisHive.

2.1 Distributed Computing On Mobile Devices

Lin et al. [26] first proposed a mobile network where nodes would be organized into non-overlapping clusters that are independently controlled and dynamically loaded. The proposed cluster algorithm was robust in the face of node failure or insertion/deletion. Wang et al. [33] worked on a bandwidth adaptive clustering approach for mobile ad-hoc networks that maintains clusters using local topology information only. In their approach, the member nodes forward only the maintenance messages probabilistically based on available bandwidth. This ensured adaptability to network conditions and reduces message overhead. This yielded the idea to construct our system to forward chunks of data for calculation to the member nodes in a probabilistic manner depending on the topology information. That is, the member nodes with more computational power and faster response time get more preference automatically. Lee et al. [25] discussed the challenges and advantages of utilizing mobile devices for distributed analytics based on an implementation of the Hadoop analytic framework. After a performance analysis of their implementation, they concluded that current mobile devices face significant limitations on transmitting and receiving reliable TCP data streams, which is required to avoid interruptions while performing distributed analytics.

A number of computational offloading frameworks have been proposed for computationally intensive mobile applications, such as by Hung et al. [22], Shiraz et al. [30], and Goyal et al. [19]. Such applications are said to be elastic in nature, and each approach partition problems at different levels of granularity at runtime. In most cases, the distributed application processing platform is composed of a mobile device that runs a local application, a wireless network medium, and a remote cloud server node. In cases where there are insufficient resources on the mobile device, an elastic mobile application can be partitioned such that any computationally intensive components of the application can be offloaded dynamically during runtime. Hassan et al. [20] showed in their study of computing-intensive mobile applications that outsourcing these computations to nearby residential computers or devices may be more advantageous than public clouds due to

\(^1\)http://www.w3.org/TR/workers/
to network impact. Cuckoo, a computation offloading framework for smartphones developed by Kemp et al. [23], allows computation offloading for Android phones to a remote server. Shiraz et al. [30] showed that current mobile computational offloading frameworks implement resource-intensive procedures for offloading. This involves the overhead of transmitting application binary code as well as deploying distributed platforms at runtime. However, runtime computational offloading is useful in decentralized distributed platforms, such as mobile ad-hoc networks. On the other hand, as Shiraz et al. mention in their conclusion, remote server nodes are unpredictable and computational offloading should therefore be performed on ad-hoc basis, at runtime. This motivated us to design our framework as an ad-hoc network of mobile devices that perform computations as needed.

### 2.2 Visualization on the Web

In the early 1990s, information visualization was still considered an emerging discipline. Towards the turn of the century, however, the pervasiveness of the World Wide Web had led to many changes, including one important application: visualization on the web. Rohrer et al. [29] notes that the Web had progressed as a source of information as well as the underlying delivery mechanism for interactive information visualization. They also note that the web, as a fundamentally new medium for visualization, was changing the way visualization applications were developed and used.

Most recent work includes Protovis [5], the JavaScript InfoVis toolkit [4], and additional toolkits for web visualization such as Processing.js, Raphael, and Paper.js. Protovis aimed at making visualizations more accessible to the web and interaction designers, using ‘marks’ to define visualization primitives such as bars (for bar charts), lines (for line charts), and labels. It provides the ability to assign data and visual properties such as position, color, and opacity both statically and dynamically to each mark, making for a succinct way for expressing visualizations over the web. In contrast, the JavaScript InfoVis (TheJIT) toolkit provides chart-level abstractions instead of shapes. However, of most interest to us is D3. Bostock et al. [6] developed the D3 visualization toolkit to enable a direct binding between the input data and the document object model. This is ideal for browser-based visualizations because browsers do not provide the same optimization opportunities as compiled programming languages. It is also twice as fast as Protovis. For these reasons, we use D3 as the target platform for web-based visualization in VisHive.

### 2.3 Big Data Analytics

When it comes to big data and its visualization techniques, one set of rules might not apply across all datasets. It is usually the case that researchers develop or use a few specialized techniques to tackle different types of datasets. For example, Fisher et al. [16] show techniques for tackling business intelligence, Wong et al. [34] discuss challenges facing extreme-scale visual analytics, and Steed et al. [31] developed a visual analytics system for specific application to the analysis of complex earth system simulation datasets.

Nevertheless, some common problems exist when it comes to visual analysis of big data. For example, traditional data analysis tools for big data rarely allow real-time visualization, whereas most real-time visualization tools perform poorly with big data [21]. Hence, tackling big data visualization involves two main challenges: perceptual and interactive scalability. Support for perceptual scalability has been discussed by Ahlberg and Shneiderman [1] using filtering, by Das Sarma et al. [13] using spatial sampling, and by Carr et al. [9] using aggregation for scatterplots. For interactive scalability, Liu et al. [27] developed a VA system called mMens, which uses WebGL for data processing and is based on the principle that interactive scalability should be limited by the chosen resolution of the visualized data and not the total number of records. The computational requirement is another stumbling block that hinders real-time interactive visualization of big data. Choo and Park [12] propose methods such as data scale confinement, classification of pre-clustered data, and linear transforms of higher dimensions to deal with computational complexity.

### 3 Design Framework: Ad-hoc Computational Clusters for Web-based Visualization

The web is becoming a ubiquitous medium for sensemaking through visualizations [6], sharing visual insights from data, and harnessing collective intelligence [32]. The goal of the VisHive framework is to facilitate the creation of ad-hoc and opportunistic device clusters over the web that forms a distributed system for handling computationally intensive algorithms driven by visual analytics applications. The driving scenario behind VisHive is the fact that people tend to carry more than a single device with them at all times. Leveraging these devices together can help scale our analytics applications to the challenges of big data. Many open and public distributed platforms exist for similar purposes; however, these are not built for visual sensemaking and data science, which yields its own unique requirements. Visualizations follow a transformative pipeline that turn data into interactive graphical representations through multiple stages [10].

To target visual analytics of big data, we need distributed frameworks capable of coupling with the visualization pipeline and aiding each step in the pipeline using connected local devices to generate a visual representation and handle user interaction. VisHive is our attempt at building such a framework using modern web technologies. For this purpose, we list seven design guidelines categorized into three groups for developing the VisHive framework.

#### 3.1 Networked Devices

The fundamental requirement for creating a distributed system is a network of connected devices. To utilize their devices including personal computers and mobile phones for distributed computation, the VisHive framework should be capable of seamlessly connecting multiple devices into a distributed system.

**D1 Cross-platform support:** The devices used by analysts for personal computing and sensemaking can be diverse, ranging from personal and public computers to mobile devices. Therefore, the framework should work independent of the underlying platforms of these devices. A distributed computational cluster should be developed using the available devices irrespective of their platform, modality, and physicality.

**D2 Ad-hoc connectivity:** A user should be capable of opportunistically creating a cluster from available devices. This includes adding to or removing devices from the clusters at any point.

Peer-to-peer networks are ideal for this purpose [3, 17] as they do not set a hierarchy among the devices, and they do not require additional infrastructure in the form of a server to create clusters.

#### 3.2 Responsive Distribution

Once the devices are connected into a distributed system, supporting visual sensemaking on the device cluster would require an intelligent management of the connected devices. The challenge in this case is to ensure that opportunistic maintenance of the clusters—adding or removing devices at any point—does not interfere with user activity within the visual analytics system.

**D3 Responsive computation management:** The devices that are not actively used are free to engage in potentially hard computational activities compared to the active ones. Computation jobs assigned to the devices within a cluster should not only be based on the processing power and available memory on the device, but also based on their current use. Therefore, the framework should identify activity on these devices and manage distribution accordingly.

**D4 Fault-tolerance:** Devices entering or leaving the cluster should be able to take up new jobs or pass on existing jobs. The framework should be fault tolerant: adaptive to the event of failure at some devices and new devices in the cluster.
3.3 Supporting Visualization and Interaction

Visual analytics systems often utilize computationally complex algorithms. For example, to understand the web traversal history of users, spanning trees from browsing histories were generated and visualized [15]. Machine learning and data mining models are also used to identify specific features, visualize interesting patterns, and prompt user exploration [18, 28]. While some of these models are inherently parallelizable, it should also be possible to configure how the underlying algorithm can be spread across the clusters.

D5 Parallelization: The algorithms driving visual analytics should be parallelized to take full advantage of the computational cluster. The framework should support defining a parallel version of an algorithm at each stage of the visualization pipeline with features to adapt the algorithm to the cluster's size and elements.

D6 Data-driven distribution: Data can come from multiple locations over time (spatiotemporal), multiple data sources (multi-modal), and can comprise several attributes per data point (multi-variate). Computations in the visualization pipeline involve transforming data of one form (input) to another (output) at each step. It should be possible to create computational job chunks for devices in the cluster based on a split at any dimension of the data. For example in spatiotemporal data, jobs can be created either by splitting data based on time or space.

D7 Handling user interaction: Interaction is an integral part of visualization and visual analytics. Users often perform continuous interaction to explore different segments of data. These interactions generate computations for the cluster to handle, and new interactions can render the previous computations obsolete. Therefore, the framework should be able to preempt and restart computations at each device in the cluster upon user interaction.

4 The VisHive Toolkit

The VisHive toolkit was developed for building ad-hoc and opportunistic clusters of computing devices for web-based visualization. It is implemented completely in JavaScript targeting the web platform and thus providing cross-platform support (D1). It uses the WebRTC API by W3C\(^2\) for establishing peer-to-peer connections across web browsers. Since the web is the target platform, the devices—called cells—are connected into a device cluster—known as a hive—as soon as they open a VisHive application webpage on the web browser (D2). The toolkit provides modules for structural definitions of distributed algorithms based upon the attributes of the hive (D3), and handles entering/leaving cells in the hive (D4). The toolkit integrates closely with the visualization pipeline, allowing developers to handle the stages in the pipeline in parallel using the connected devices (D5, D6, D7). Figure 2 shows the network architecture of an VisHive toolkit example.

4.1 System/Network Architecture

The VisHive toolkit consists of four components based on the design guidelines D1 through D7:

**C1 Job partition layer** that divides a high-level computation operation into computation jobs (known as chunks);

**C2 Communication layer** to share the chunks across the cells;

**C3 Integration layer** that combine the results from all the cells and pass them to the web visualization; and

**C4 Job control layer** that handles fault-tolerance i.e., cells entering and leaving the hive.

Figure 3 depicts the VisHive architecture covering these components. The VisHive network architecture is a peer-to-peer (P2P) connection established across the browsers of the cells using WebRTC technology, popularly used for real-time video calls over the web browser\(^3\). Our implementation uses the open source PeerJS framework\(^4\) for establishing the P2P connections across the cells. The P2P connection creates the communication layer (C2) for transferring chunks to the cells within the hive.

The control of the distributed system is inherently with the instance that the user actively interacts with. Currently, we support just one active user interacting with VisHive on a device—the framework is not intended for collaborative visualization. That way, the controlling instance, or master, takes the help of other idle devices, or slaves, and distributes the computation jobs amongst them. The slaves are our cells, and, along with the master instance, make up the hive. The cells use their browsers and join the VisHive server by navigating to a VisHive application URL. As these cells join the hive, they share details on their free CPU power and resource allocation, which the master then uses to allocate the jobs amongst them. Since the VisHive toolkit targets ad hoc and opportunistic device clusters (for example, between an analyst’s smartwatch, smart phone, and laptop), this registration process ensures that distribution happens in an environment-aware fashion.

After the cell registration process, the individual cells are capable of distributing the computations involved in each stage of the pipeline for generating a visualization (C1). The VisHive toolkit also maintains common data structures across cells. For example, this can contain the computational models used for each step of the pipeline. When a master shares computation jobs with the slaves in the hive, the cells accept the jobs and look up the input data from the job definition. All the cells including the master will then perform the required computations on the input using the shared computational models and send the output back to the master. The master then handles the integration process by recombining the results from the cells using the integration modules (C3). By default, the integration modules combine the results from the cells into a list, however, this logic can be overridden by the developer based on the context of use.

4.2 Job Allocation and Control

Job allocation and control within the VisHive distributed system is handled by the C1 and C4 components of the toolkit. Each computation job (chunk) is treated as a mapping from input to output, generated by shared computation logic. This is similar to the MapReduce model [14] for processing big data on parallel and distributed systems. The default configuration for job partitioning involves splitting the input data for a high-level computation into jobs that work on parts of the data. The job allocation module creates the chunks based on the available resources on each cell and the number of cells in the hive (including the master and the slave cells). Furthermore, explicit application logic created by the VisHive application developer (end-user developer) for splitting a computation (and input) into chunks is also supported.

\(^2\)http://www.w3.org/TR/webrtc/

\(^3\)http://apprtc.appspot.com/

\(^4\)http://peerjs.com/
The VisHive toolkit maintains a shared memory containing the common data structures for performing computation jobs on each cell. For example, a pattern recognition model that translates multi-dimensional input vectors part of a large dataset into features, can be shared with other cells to perform individual input to feature mapping. Once the computation jobs are finished the results are sent back to the origin, where the results are compiled and transferred to the visualization pipeline.

The recovery modules (C4) are responsible for handling the failure scenarios in the VisHive distributed systems. This automatically handles the failure scenarios when existing cells leave or new cells enter the hive. For these scenarios, a cell that leaves a hive passes on its assigned computational jobs uniformly to the other cells in the hive, and a cell that enters a hive during a current computation process has to wait till the next batch of jobs.

Distributed systems also face issues with conflicts and concurrency. In our case, the VisHive toolkit inherently splits the input data based on the application logic and passes it to the cells in the hive. This means each cell handles a specific job and, therefore, cannot by design conflict with one another. When simultaneous responses are sent back to the origin, the event-driven and asynchronous I/O nature of JavaScript handles the incoming results in a queue. This provides the toolkit with the capability of concurrency managed distribution. Apart from this, we assume that any conflicts occurring during the integration process that are modified into chunks and distributed across the hive, for Google can help the user make a prediction for Yahoo.

When a user interacts with the system by making a prediction based on the past 2 years of stock data provides multiple predictions for the future given the stock prices within the last week (temporal predictions). This model generates predictions of different confidences. Furthermore, predictions from the neural network can be chained together to create a temporal prediction path in the future, which resembles a tree (Fig. 4).

When a user interacts with the system by making a prediction based on the past 2 years of stock data provides multiple predictions for the future given the stock prices within the last week (temporal predictions). This model generates predictions of different confidences. Furthermore, predictions from the neural network can be chained together to create a temporal prediction path in the future, which resembles a tree (Fig. 4).

When a user interacts with the system by making a prediction based on the past 2 years of stock data provides multiple predictions for the future given the stock prices within the last week (temporal predictions). This model generates predictions of different confidences. Furthermore, predictions from the neural network can be chained together to create a temporal prediction path in the future, which resembles a tree (Fig. 4).
gets a list of stocks it needs to predict; and (2) splitting the process based on time, where each cell in the cluster generates some prediction paths (branches) in the tree.

O2 Prediction fitting: Once a prediction tree is generated, when user interacts by making a prediction for a stock, a path in the tree which fits this prediction should be found. For example, if the user says that Apple will increase by 10% in the next 10 days, a path in the prediction tree that best fits this estimate should be found. This process involves finding the closest and highest confidence paths from a list of $N^2$ paths. The paths object maintained on the master device is split into chunks and passed to all other devices for filtering the best possible fit on each cell, which are further verified to find the best prediction fit by the master.

O3 Correlation predictions: After the best prediction fit is calculated, the prediction tree for the rest of the stocks are modified to reflect the correlation between stocks. This process involves recreating the prediction tree using the correlation predictions at each time step, in contrast to the temporal prediction tree generation. However, similar to previous operation, this process is parallelized by splitting the tree generation computation into chunks based on the rest of the stocks or prediction paths/branches that are handled by each cell.

While O1 represents an operation happening when the user visualizes stock market data for any time period by adding a stock to his portfolio, the remaining operations (O2 and O3) happen when the user interacts with specific stocks of interest. The operations performed in the visualization pipeline to handle the stock data are inherently parallelizable. Therefore, we use VisHive to split the data belonging to the stocks that are in the user’s portfolio based on the strategies described above and generate the chunks that need to be handled by each cell.

For tree generation (O1), VisHive defines chunks as the past seven values that are used for prediction and along the strategy that needs to be followed to handle this data. On the other hand for handling user interaction (O2, O3), the chunks contain the predictions made by the user and links to the computational models to perform the corresponding operation. Once the chunks are received by the cells, they process the chunks using the computational models (trained neural networks) that are maintained on the shared memory of each cell. StockHive is able to seamlessly handle generation of these complex prediction paths with VisHive and utilize the computational power of the opportunistic cluster. We tested the performance of this application using multiple device combinations: laptop, smartphones, and tablets, and results of this study are described in the next section.

6 Performance Evaluation

We executed StockHive on an ad hoc network hosted by a laptop running Google Chrome that served as the master instance. Worker cells are mobile devices that joined the ad hoc network and were added to the StockHive cluster. A preliminary performance evaluation was done to see how the number of cells in the hive, apart from the master, impacted the performance for a computation that can be split into chunks just by splitting the data. For the performance evaluation, we chose to create the chunks by letting each cells handle a subset of the stocks of interest. The performance of VisHive was measured as the total time taken to finish all the predictions for the stocks values of the companies chosen by the user.

During the performance evaluation, the system was started and the user chose 2, 4, or 8 stocks as their portfolio. After the stocks have been chosen, their future values have to be calculated by StockHive and the prediction tree should be drawn for each. We predict 20 days into the future, using the trained neural network on the past values. This was repeated for the scenario of one master instance, one master and a slave (a tablet) and one master and two slaves (a tablet and a phone). Results are reported in Fig. 5.

When the timing results were analyzed, we found that the time taken for all the predictions to finish reduced (almost) linearly with the number of cells added, with few overheads that need to be further investigated. Although this is expected in embarrassingly parallel scenarios, it validates our system as the performance of the hive improves with added cells and more computation power.

On closer analysis, we found that the time taken for the communication between the cells and the master or vice versa was in fact negligible (order of 10ms) since the chunks are fairly small in size (order of Kb). This means that there was a small overhead due to the communication layer itself. However it is to be noted that since this was a locally hosted ad hoc network, the communication was much faster and more stable than other wireless connections (for instance, public Wifi).

The processing power of the devices plays a big role in the performance, as the phone outperformed the tablet due to its superior processor. However mobile devices are inherently throttled to avoid overheating under continued stress and this could be a major reason for the delays on both devices, along with their limited computing power.

7 Discussion

One contentious topic that we have not yet covered in this paper is the existence of the IPython development environment, which through the guise of IPython Notebooks—a web-based interactive shell for

http://www.ipython.org/
parallel computing, distributed systems, and high-performance computing that can begin to guide the design of suitable algorithms that can be run on top of VisHive.

8 CONCLUSION AND FUTURE WORK

We have presented VisHive, a JavaScript toolkit that allows for connecting multiple devices into an ad-hoc cluster using just the web browser as the computational platform. Devices become cells in a hive where a master allocates and recombines jobs to slaves that perform the actual calculation. The communication between the cells is performed using direct browser-to-browser connections in a peer-to-peer architecture, thus requiring no central server or connection to the internet. To showcase the utility of the technique, we presented StockHive, a visual analytics tool for stock market prediction that employs mixed-initiative interaction. Instead of performing a set number of predictions and then waiting for user input, StockHive splits the predictions across the available devices and keeps performing them until interrupted by user input. Our performance evaluation using the StockHive application show a significant speedup basically linear with the number of connected cells.

We see many potential refinements and improvements of the VisHive toolkit in the future. For example, we envision continuing our work on parallelizing all steps of the visualization pipeline, including not just data transformations and the visual encoding, but also the view transformations and input management. Furthermore, we are interested in investigating how to use the slave cells not just as headless computational units, but also for collaboration (for multiple users) or for supporting the main device with additional views and input surfaces (for a single user with multiple devices). Finally, we would like to study the usability aspects of firing up multiple devices to offload a main device, and how this discovery process can be streamlined.

ACKNOWLEDGMENTS

This work was partially supported by U.S. National Science Foundation award IIS-1253863. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the funding agency.

REFERENCES


