A power optimised and reprogrammable system for smart wireless vibration monitoring

James Long | Oral Büyükoztürk

Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, 02139, USA

Correspondence
Oral Büyükoztürk, Department of Civil and Environmental Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, 02139, USA. Email: obuyuk@mit.edu.com

Funding information
Shell Exploration and Production Company

Summary
Structural health monitoring (SHM) applications generally utilise high sampling rates, which low-power wireless protocols are not well equipped to handle. Smart sensing approaches can overcome this, by using the processing capability of the sensor nodes to reduce the volume of data prior to communication. Most smart sensing approaches are preprogrammed and static. This causes two issues: First, the data processing logic cannot be easily modified, making it difficult to update and improve algorithms once deployed. Secondly, there is limited ability to adapt to changes in the environment or degradation of hardware. To address these problems, we have developed a system that allows users to remotely specify their computational logic on the fly in a MapReduce style syntax. We model these user-specified tasks as a directed acyclic graph, and combine this model with statistics of the performance of each node in the network to formulate an optimisation problem. Solving this problem optimally allocates data processing operations to nodes in the network, such that the total time spent is minimised. We demonstrate a field deployment of this system, which illustrates the advantages of the proposed approach for a typical SHM application, and examines robustness of the system under environmental variations.

KEYWORDS
distributed computing, optimisation, smart sensing, structural health monitoring, wireless sensor networks

1 | INTRODUCTION

Wireless sensor networks alleviate the need to install cables to supply sensors with power and communication capability. This dramatically reduces the cost and time required to set up the network, as well as enabling the instrumentation of structures and facilities where the installation of cables is not possible. However, without a constant supply of power and a reliable, fast communication link, the management of sensor resources, for example, battery life, and available bandwidth, become major issues. Wireless vibration monitoring in particular, poses some difficult challenges. Sampling rates of accelerometers for structural vibration monitoring are typically in the 100s of Hz, and it is often required to simultaneously collect data from many different sensors in the network. Mesh network protocols, desirable for their low power, long range, and self-forming and self-healing properties, are not well suited to the inundation of data created by this type of task.
One example is given in Pakzad et al., where it is reported that 1,600 s of data at a 50 Hz sampling rate was recorded from 64 nodes installed on the Golden Gate Bridge. To transmit this data from all nodes to a central data sink required a total of 9 h. To combat this problem, rather than viewing the sensor network as a collection of data sources that push their data to a centralized sink, it can be viewed as a distributed computing system, where each node in the network is capable of processing data before transmitting. This is a paradigm generally referred to as smart sensing. In many real world applications, the desired end result of data acquisition is not the raw time series, but rather some much lower dimensional metadata, and so carrying out some or all of the processing within the network can reduce the volume of data transmitted and alleviate the bandwidth saturation.

Many approaches to smart sensing in the structural health monitoring (SHM) literature are preprogrammed and static. This results in two major drawbacks. First, the embedded application logic cannot be easily adapted without firmware updates. This means iterative approaches to algorithm development are difficult, creating a high barrier to the installation of new systems. Similarly, it is quite difficult to accommodate multiple users of the same sensor network who wish to deploy different application logic. Flouri et al. is one notable instance of a reconfigurable system, which allows users to update and adapt the smart sensing application logic over the air.

Smart sensing systems proposed for SHM purposes can also be characterized by the level of node-to-node coordination they enable, as discussed by Nagayama et al. In the simplest approaches, nodes operate independently of each other, acquiring and processing data, before returning results to a centralized server. In these independent approaches, no sharing of data between nodes is possible, and so, computation that requires data from multiple nodes is only possible at the server. In vibration monitoring and structural damage detection applications this can be limiting, as system-wide properties such as mode shapes or other internode spatial dependencies are often of interest.

To manage the complexity arising from node-to-node coordination, a specific computational pattern is often enforced. The adoption of a standardized pattern for parallel processing ensures that the flow of data between computational operations is strictly defined, even if the actual logic associated with these operations may change. For, example Gao et al. propose a hierarchical tree model of parallel computation, where neighboring sensor nodes form communities, with each containing one cluster node. The cluster node aggregates data from the other nodes in its community, and can exchange data with cluster nodes from other communities, as well as forward data to the base station. In and Rice et al. implement the distributed computing strategy proposed in as part of an autonomous smart sensing system, and use the tree-type computational model as a template for implementing system identification algorithms, such as frequency domain decomposition. Jo et al., adopt a similar computational topology, and use high sensitivity sensors as reference sensors in a decentralized implementation of the Natural Excitation Technique approach. Sim et al., provide another good example of a coordinated smart sensing approach, where the random decrement technique is implemented using the tree-based pattern and demonstrated on an experimental steel truss. These approaches illustrate the benefit of coordinated approaches to smart sensing that adopt a specific computational model: Once the computational model is defined, low-level networking and control flow logic can be standardized and incorporated into the smart sensing system, simplifying the implementation of new data processing algorithms.

In it is noted that "to facilitate communication, the manager sensor should be centrally located to the other sensors in the community...", and in it is recommended that "the network topology should be tailored to the modal analysis method considered...". Incorporating the decision-making capability to autonomously solve these operational problems within the smart sensing system is possible with sufficient information on the network characteristics and data processing methodology. In real deployments of wireless sensor networks, communication link strength may vary greatly over time, whereas nodes' effective processing capabilities can be heterogenous across the network and may vary according to available memory. Enabling the smart sensing system to adapt to both the network characteristics, and the computational characteristics of the deployed processing methodology, could significantly improve performance.

For example, Zimmerman and Lynch, develop a smart sensing system for the distributed computation of mode shapes within a sensor network, which uses a similar tree structure to and but incorporates a market-based approach for resource allocation. This approach allows the topology of the computational tree to be decided through an auction, which maximizes the total utility of the system, where utility is defined as a combination of improved modal assurance criteria and communication reliability and decreased computational and storage burden. This illustrates how the addition of more sophisticated decision-making capabilities within the sensor network can augment coordinated smart sensing approaches and improve their performance.
In this work we propose a novel smart sensing system, for SHM applications which is dynamic, coordinated, and can adapt to differing processing methodologies and network characteristics. This system allows users to specify their application logic on-the-fly in the form of a MapReduce style distributed computing job, implemented in Python and submitted remotely via a web connection. This framework is flexible enough to enable implementation of many different analyses, while also being structured in a way that is suitable for optimisation.

Taking advantage of this structure, we model each MapReduce job as a directed acyclic graph (DAG), whose vertices are tasks, each with an estimated computational burden, and whose edges represent the flow of data between tasks, again each with an estimate of the volume of data. Combining this model with network statistics describing the processing speed and connectivity of each node in the network, we formulate an integer programming-based optimisation problem, which seeks to allocate each task in the DAG to a node in the sensor network.

The system proposed in this paper allows users to specify their application logic on the fly, enabling multiple users to share the same sensor network, and to easily update methodologies without reprogramming devices. It also allows the generation of suitable execution plans for any user-defined job. Some applications benefit greatly from distributing processing tasks to low-power nodes in the network, whereas for other applications it is more suitable to simply push the data back to the powered sink node. Our approach aims to generate the optimal execution plan for each application, based on both the nature of the application logic, and the current state of the network. By incorporating almost real-time tracking of network statistics as an input to our optimization algorithm, our execution plan adapts dynamically to changes in the environment and operating conditions. This enables us to effectively adapt to changes in node’s effective processing capability or connectivity.

2 | SYSTEM OVERVIEW

The system described in this paper is comprised of three separate components: (a) an interactive browser application that displays information about the sensor network to the users, and allows the user to submit their computational tasks and view the results of these tasks; (b) a gateway server that connects the internet (and therefore the user) to the local sensor network. This gateway server also tracks statistics on the performance of the network, and optimizes the execution plans of submitted jobs; and (c) The mesh network itself that is comprised of many individual nodes, each containing a battery, a processor capable of running the MicroPython interpreter, a XBee Digimesh radio, and a microelectromechanical systems (MEMS) accelerometer. These individual nodes run a lightweight multitasking system developed specifically to execute the computational framework proposed in this work, which we will describe in more detail in the following section. An overview schematic of the system is shown in Figure 1.

* Coordinator node with a serial connection to the server

FIGURE 1   Schematic of system architecture
3 | DISTRIBUTED COMPUTING FRAMEWORK

To execute a monitoring job, users specify their application logic in the form of a MapReduce-style computation, and submit it to the gateway server via the interactive browser application or a simple web socket connection.

MapReduce is a popular programming model for specifying and executing parallel, distributed algorithms on clusters. In the canonical MapReduce model, distributed algorithms are composed of: (a) a map method, which operates in a fully parallel fashion on chunks of data commonly stored on a distributed file system, producing one or more key-value pairs for each chunk of data and (b) a reduce method, which operates on groups of values that share the same key, producing a result for each key and associated group of values. As a simple example, in a text-processing application, a map method may operate on a list of text files, counting the frequency of occurrence of words for each file, and emitting key value pairs where the key is the word, and the value is the number of occurrences. Clearly, the processing of each text file can be done completely in parallel. The reduce step can then sum the counts for each unique word from each text file, producing an aggregate count of word occurrences over the entire list of files. The assignment of cluster resources, error handling, data communication, and so forth is orchestrated internally by a framework, for example, Hadoop, and shielded from the end user of the framework. This greatly simplifies the process of developing and executing distributed computing algorithms.

In this work, we develop an analogous framework that allows users to specify their smart-sensing applications in the MapReduce style, for execution on a network of wireless sensors. The MapReduce style is adapted slightly for the wireless sensing environment: In addition to a map and reduce method, users must also provide a sampling method, that specifies the sensor interface from which data is to be acquired and details of the desired sampling frequency and duration, as well as a list of nodes at which data is to be sampled. The result of each sampling method is then directly processed by the map methods.

By utilising the system developed in this work, the end user does not have to specify how data is routed through the network, how to interface with hardware, how to handle errors, long-running tasks, communication failures, or other exceptions. This will allow engineers to instead focus on the development of useful SHM methodologies, while still availing of the considerable advantages of in-network, distributed processing. The MapReduce style of distributed computing, although not universally applicable, has been demonstrated in cluster computing to be adaptable to many algorithms. We will demonstrate in this paper how some common SHM applications can be expressed easily in this style. Note that although simple fully decentralised algorithms can be straightforwardly implemented, the reduce step allows for the implementation of more complicated algorithms that require coordination of data from multiple sensor nodes.

Conceptually, the MapReduce based framework as implemented in this work can be described approximately by the procedure outlined below in Algorithm 1. For a more detailed examination of MapReduce based distributed computing algorithms see.

As described in Algorithm 1, to submit and execute a new monitoring job, a user of the system must specify the following: (a) a sampler function, which samples data from a sensor interface on the node and returns a stream of this data, (b) a mapper function, which takes the output of the sampler function as an argument, performs a transformation, and yields a result in the form of one or more key-value pairs, (c) a reducer function, which takes a key and a list of values as arguments, and returns a final value to be returned to the user, (d) a list of nodes at which to sample data using the sampler function, and (e) optionally, a list of candidate nodes at which to execute map functions and reduce functions.

Each sampler function executes in parallel, at its specified node in the network. The mapper functions then operate in parallel on the output of the sampling functions. Collectively, the list containing the output of all of the sampling functions is denoted \( \text{streams} \). The higher order map function then applies the user-defined function mapper to each element of \( \text{streams} \), producing a list of key value pairs, \( \text{kvs} \). In practice, this map step is executed in parallel in the network, and the list \( \text{streams} \) does not need to be physically stored in any one location. The resulting key-value pairs are each routed to a reduce node based on their key, such that all key-value pairs sharing the same key are sent to the same reduce node. Conceptually, this routing implements the \( \text{groupbykey} \) operation on the list of key-value pairs outputted by the map step and results in a new distributed list \( \text{shuffled} \), where each element is comprised of a unique key, and an associated list of values. Finally, the reduce function is executed in parallel on each key and its associated list of values, and the results of the reduce functions are returned to the user.
The intermediate shuffling step which occurs between the map step and the reduce step groups key-value pairs resulting from the map step by their key, sending all pairs with matching keys to the same reducer. This grouping is achieved by means of a partitioning function. The simplest choice of partitioner, which is utilized in our implementation, is to compute the hash of the key, modulo the number of reduce nodes, thereby providing the index of the destination reduce node. Other partition functions are possible, but functions that result in an uneven distribution of keys to reduce nodes may impact the overall performance of the system.

3.1 Example problem

To demonstrate how users of our system can implement monitoring jobs, we will describe a simple example in this section, and provide working Python code that runs this example job if submitted to the system.

In this example, we sample triaxial acceleration data at three locations in the network, and identify, for each axis, which location of the three recorded the peak acceleration value. This job can be expressed with the following python code.

```python
class SenseReduce:
    def __init__(self):
        # sample data at node 1,2 and 3
        self.samplenodes = [[1],[2],[3]]
        # execute the map step at node 1,2 and 3
        self.mapnodes = [[1],[2],[3]]
        # execute the reduce step at both node 0 (the server), and node 5
        self.reducenodes = [[5],[0]]
        self.every = 3600  # execute every hour (3600sec = 1 hour)
        self.repeat = -12 # execute 12 times in total

def sampler(self,node):
    # sample from node.accel (triaxial accelerometer)
    # at 100 Hz for 2 seconds total
    acc = yield from node.accel(f=100,t=2)
    # return a tuple of the node ID (e.g. 1,2 or 3)
    # and the sampled triaxial accelerometer which
    # is a dictionary with the following structure
    # {'x':[(a0)...(atn)],
    #  'y':[(a0)...(atn)],
    #  'z':[(a0)...(atn)]}
    return (node.ID, acc)

def mapper(self,node,data):
    # unpack the tuple
    nodeid, acc = data
    # absmax is a function which returns the element
    # with the largest absolute value in the list
    absmax = lambda x: max(min(x), max(x), key=abs)
    # peak_by_axis describes the largest
    # absolute acceleration recorded for each axis
    peak_by_axis = {ax:absmax(meas) for ax, meas in acc.items()}
    # iterate through peak_by_axis yielding a key-value pair for each axis, where the axis letter, e.g. 'x' is the key
    # and the value is a tuple of the node ID where the data was recorded
    # and the absolute largest recorded acceleration
    for ax, peak in peak_by_axis:
        yield (ax, (nodeid, peak))

def reducer(self,node,k,vs):
    # peak by location operates on a list of (nodeid, abs_accel) tuples
    # and returns the nodeid with the largest abs_accel value
    peaklocation = max(vs, key=lambda x: x[1])
    # finally return a tuple of the key (i.e. axis letter)
    # and the peaklocation, (i.e. node ID where largest absolute acceleration was recorded)
    yield(k, peaklocation)
```
The sampler function in this job samples 2 s of data at 100 Hz, and returns a tuple of its node ID, and the data, which is in the form of a dictionary, with x, y, and z fields, associated with arrays of time-series measurements. The mapper function finds the value of the peak acceleration for each axis and yields three key-value pairs, with the axis letter as the key. The values are a tuple of the sampling location id, and the peak value observed. The reduce function then operates on unique keys and their associated list of values. In this example, acceleration axis is the key, and is associated with a list of tuples of sampling location id and peak acceleration, which corresponds to this axis. The reduce function computes the tuple with the largest peak value and returns it to the user, with a key denoting the axis. All three reduce functions (one for each axis), will then return their axis letter, and the sampling location that recorded the peak acceleration for this axis.

Note in this example that each task is assigned exactly one node in the network for execution. This is specified in the "__init__" method of the submitted job, where the user defines the number and location of the "samplenodes", as well as optionally the "mapnodes" and "reducenodes". In this example, the samplenodes are specified to be Nodes 1, 2, and 3, meaning that the sampler function will be executed at these nodes. The mapnodes are identical to the samplenodes, meaning that the mapper function will be executed at the location the data is sampled at, whereas the reducenodes are defined to be Node 0 (the server) and Node 5. As will be described later, it is also possible to specify a set of candidate nodes for each task, from which the optimal node is chosen. Note also that there are two parameters, "repeat" and "every", which allow this job to be scheduled on a recurring basis. If specified, the job will run repeat number of times, with an interval of every seconds between runs.

### 3.2 Job profiling

Monitoring jobs specified in our computational framework are modeled as DAGs, where each computational task corresponds with a vertex in the graph, and the flow of data between these computational tasks corresponds with the edges of this graph. An illustration of the graph associated with the peak acceleration example is shown in Figure 2. When a job is submitted, it is profiled by gateway server, to obtain an estimate of the computational burden of each node in the DAG, and of the volume of data to be transmitted on each edge. This is similar to the approach of Newton et al. This information is then used as an input to the optimization engine that assigns computational tasks to nodes in the network.
3.3 | Summary of key features

The distributed computing framework developed in this paper allows for in-network, parallelized execution of user-specified jobs, written in Python, in the established MapReduce style. These jobs can be submitted to the network via the internet, enabling users to reprogram the behavior of the network almost instantaneously over the air. For convenience, users can simply schedule these jobs to execute multiple times at a given interval.

The parallel execution of user-specified monitoring jobs is optimized by the system: Each job consists of multiple subtasks, each of which is assigned to a given node in the network for execution. The system profiles submitted jobs to analyze each subtask, and combines this with information on node processing speeds and communication strengths to optimally assign each subtask in the job to the best node in the network. This enables the system to adapt its behavior to varying application logic and environmental conditions.

4 | IMPLEMENTATION DETAILS

4.1 | Node-level multitasking

Wireless smart sensing applications require sensor node platforms, for example the iMote2,16 and Narada,17 which are capable of sampling, processing and transmitting sensor data, coordinating with other nodes, managing power consumption, and many other tasks. To coordinate and schedule these tasks, a (typically lightweight) operating system is required.

One common choice for SHM applications is TinyOS.18 TinyOS is an event-driven, cooperative task scheduler. Two modes of execution are available in TinyOS: tasks and interrupts. Tasks are maintained in a first in first out queue, where they are then executed by the operating system in order. Interrupts can preempt tasks and have an associated interrupt service routine or handler. Because tasks are expected to be nonblocking in TinyOS to maintain concurrency, implementing complex or long-running logic is usually achieved by combining event handlers. Among the drawbacks of
TinyOS, which are mentioned as challenges or motivation in the literature are (a) that this concurrency model and use of NesC programming language creates a significant intellectual burden for the application programmer, (b) that the lack of any concept of priority in the task queue can lead to important tasks being delayed by less important tasks, (c) That the static nature of compiled TinyOS applications limits the flexibility of wireless sensor networks compared with wired, and (d) that the cooperative nature of TinyOS makes it challenging to achieve real-time performance.

In, Wang et al. describe a two-threaded concurrency model which allows the use of timer interrupts to pause and then resume an executing task, and demonstrate its efficacy in achieving real-time continuous data collection in a sensor network deployed on the Geumgang Bridge, South Korea. The sensor node design proposed in this work is comprised of a central computational core, with serial interfaces to both the wireless transceiver, and a dedicated analog-to-digital converter which itself interfaces with analog sensors. The multithreaded concurrency system allows each node to easily switch between collecting data from the sensor interface, processing data and transmitting data.

Linderman et al., developed a real-time data acquisition framework on top of TinyOS. This is not achieved using a real-time operating system (which TinyOS is not), but rather by analyzing required processing and send times, and using timer interrupts to precisely schedule communication slots in a time-division multiple access scheme. In contrast, Fu et al., propose the use of FreeRTOS, as the underlying sensor node operating system. FreeRTOS is a preemptively scheduled, real time operating system, commonly used in industrial process control systems, and other applications requiring hard real time guarantees. In addition to providing well defined time constraints on tasks, the use of a RTOS in, also provides the convenience of over-the-air reprogramming of sensor nodes, which the authors state "is a significant benefit for SHM application".

Flouri et al. consider the problem of sensor node reconfigurability. The STONE system presented in, which is built on top of TinyOS, allows both an execution mode and a configuration mode. In configuration mode, the parameters of monitoring tasks can be specified by the user. These parameters include sampling rates, which processing steps from the available library to apply to sampled data, and task on and off times. Once in execution mode, these monitoring tasks are executed by the node scheduler. The system can then reenter configuration mode by waiting for any executing tasks to finish, and removing any future tasks from the queue.

In this work, we implement a cooperatively scheduled coroutine-based concurrency model using MicroPython, and the asyncio library, which our distributed computing framework for SHM is then built on top of. MicroPython is a free and open source Python compiler and run-time that runs on microcontrollers with as little as 16 kB of RAM. Choosing MicroPython alleviates the need for complicated reprogramming or reconfiguration steps: Our framework simply receives jobs specified in Python code over the radio, and executes these jobs in the framework on-the-fly, without stopping the scheduler or modifying the underlying program in any way.

The coroutine-based concurrency model allows for tasks to be easily paused and resumed, and so eliminates the need for many callbacks that is typical of simple event-based cooperative schedulers. Communication between coroutine tasks is achieved using queues, and a producer–consumer pattern. Although callbacks are not typical in this model, reader and writer callbacks for serial interfaces can be registered on the event loop, which then efficiently polls the interface for read or write availability. A priority queue is used for task scheduling, rather than a first in first out queue, which allows for more control over execution order. Important coroutine tasks in our framework include: Reading messages from radio, placing jobs received over radio on the queue, executing queued jobs, triggering sensor collects, reading available sensor data, sending data over radio, awaiting send acknowledgments, and managing intermittent sleep cycles. Figure 3 shows a simplified model of how coroutine tasks are concurrently executed with subtasks depicted on the left side of the figure. On the right side of this figure, a typical progression of subtask execution is shown, with each time slice color coded according to the subtask being executed at that time. In this progression, the processor is initially in low power sleep mode, regularly returning control to the event loop. A message is received from the radio containing a job, which is then placed on the job queue for later execution. Once the execution time is surpassed, the function is executed, and the result of this execution is transmitted over radio.

Because this concurrency model is cooperative in nature, subtasks can take an arbitrary amount of time to complete before yielding control to the event loop, and so it is not currently suitable for applications which have hard real time requirements, such as control and actuation.

4.2 Scheduling and synchronization

The architecture of our sensor node is similar to that described in, where analog to digital conversion, signal conditioning, and time stamping are carried out independently by a separate hardware module. The central node processor then provides a serial interface over which digitized data is received.
To ensure that the execution of network wide jobs occurs synchronously, the gateway server attaches an "earliest execution time" to the job details when disseminating them to individual nodes. Data collection is then triggered at each of the sampling nodes no earlier than this earliest execution time, according to each node's real time clock (RTC). This strategy is not guaranteed to succeed: Although nodes will not begin earlier than the specified execution time, initialization may be delayed if engaged in a different task. If the difference between the specified earliest execution time and the actual start time is larger than a defined threshold, an error is thrown. The maximum acceptable discrepancy can be adjusted remotely, which may be desirable in particularly time sensitive applications.

To compensate for long-term drift of the RTCs, each node intermittently updates its RTC either by referencing GPS time, if a GPS module is present, or using the network time protocol, if the node is currently connected to a local area network. In the experimental deployment described in this paper, network time protocol was used to update the node's RTCs.

### 4.3 Computational library

To aid implementation of sophisticated monitoring methodologies in our framework, we have developed a library of useful signal processing and scientific computing algorithms, which can be called from user specific map and reduce functions. In particular, we have implemented these algorithms with two goals in mind: avoiding excessive blocking of the event loop, and minimising the amount of RAM used in computation. It should be noted that in addition to the algorithms outlined below, the underlying micropython interpreter also provides many convenience functions (sum, max, min, etc.) that are useful in specifying monitoring jobs.

- Matrix-vector multiplication
- Vector L1 norm computation
- Vector L2 norm computation
- Vector normalization
- Matrix multiplication
- Matrix transpose
- Eigenvalue decomposition
- Matrix inversion, using eigenvalue decomposition
- Fast fourier transformation, using radix2 algorithm
- Lasso, using coordinate descent algorithm

### 4.4 Message passing

Node to node communication in our system is implemented with at-least-once semantics. The message sending task in our system consumes messages from a message queue and sends them over serial port to the radio. A separate task caches sent messages until a failure, success, or timeout occurs. Messages which fail or timeout are then requeued for sending. To accommodate this message passing scheme, we adopt the principle of idempotency: Any subsequent receptions of the same message have no additional effect on the system.
4.5 Fault tolerance

Although the at-least-once message passing scheme does guarantee eventual delivery of messages if possible, occasionally, nodes lose communication with the network, which may cause monitoring jobs to hang until communication is restored. Rather than let this occur, consuming valuable energy resources, we instead implement a timeout on each subtask in the job. Tasks receiving data from parents in the dataflow graph wait for a specific timeout period for data from each parent, before throwing a timeout error. A key value pair, which contains an error code as the key, and the text of the timeout error, along with the id of the associated parent task, as the value then acts as the received data.

Thus, any task in the graph can either pass a successful result, or a failure message to its child task. Errors thrown during execution of tasks, for example memory allocation failures, or errors thrown as a result of bugs in user-specified code, are handled in the same way. If a parent task’s computation is successful and is sent within the timeout period, the result is received by the child task. If either the computation fails, or the communication times out, the child task receives an error message.

Any received error messages are simply propagated through the dataflow graph and returned to the user along with valid results. Error results take the form of a key-value pair, with an error code as the key. Errors arising from the sampling layer are passed through the map layer unaltered. Thus, any errors arriving at the reduce layer are grouped by the error code key, and sent to their own reduce task, separate from nonerror key value pairs. This allows for successful execution of as much of a given job as possible, with transparent handling of any errors.

A detailed illustration of the designed fault tolerance strategy is shown in Figure 4, with several examples of how potential errors are propagated. In designing this fault tolerance strategy, we have deliberately chosen an approach that quickly reports errors to the user, rather than one which attempts reconfiguration or recovery. Consider for example a sampling task that encounters a transmission failure when attempting to communicate its results to the downstream mapping task. Using our strategy, the mapping task will encounter a timeout error that will then be propagated through the task graph to the user, indicating that the communication link between the sampler and mapper node in question was faulty. In an alternative approach, the sampler task, upon failure to transmit, could attempt to reconfigure the task graph and assign a new node to execute the downstream mapping task. Whereas such an approach would in some instances result in successful completion of the job, it would also necessitate significant communication overhead, in reassigning tasks and communicating this reassignment to all downstream nodes. In general, our chosen error handling strategy may be overly pessimistic in some instances but avoids unpredictable long-running jobs arising from repeated reconfiguration attempts and requires minimal implementation overhead.

5 TRACKING NETWORK PERFORMANCE

The continuous collection of relevant network characteristics is a critical aspect of the system described in this paper. We collect near real-time information on the processing speeds of nodes in the network, and the communication speed of links between these nodes. This allows the system to generate an execution plan that adapts to the characteristics of the network. For example, if processing speeds are very low, but communication speed is very high, it is relatively less advantageous to carry out computation on the edges of the network. Conversely, if processing speeds are relatively high, but communication is very slow (as is often the case), it may be more advantageous to utilize the computing capability of the devices in the network.

Node processing speeds are updated every time a computational job is executed on a node. The server node, which is typically the most computationally powerful, is assigned a speed of 1. Every processing task is executed by the server, and the time taken is recorded. When a node executes this same processing task, the execution time is logged and reported to the server. This allows comparison of the speed of the nodes relative to the speed of the server. For simplicity, we use the most recently recorded speed for each sensor in the network. Many other estimation methods could be applied to improve the accuracy of this measurement, and this is an interesting area for future work.

The network is also periodically polled for communication strength statistics. The result of this polling is a graph representation of the network, with edge weights given by the received strength signal indicator (RSSI) in decibels. Typical values of RSSI range from $-40\,\text{dB}$ to $-100\,\text{dB}$. We first convert this RSSI value to a probability of individual message success using a sigmoid function, similarly to Zimmerman and Lynch\textsuperscript{10}:

$$p_s = \frac{1 - P_f}{1 + e^{-k(RSSI-x_0)}}, \quad (1)$$
where \( p_{fp} \) is the probability of an individual message failing even when signal strength is near perfect, \( k \) is a parameter controlling the steepness of the curve, and \( x_0 \) is the \( x \) value of the function corresponding with the \( y \) value of 0.5. Here we choose \( x_0 \) to be −70, as this corresponds with the midpoint of the typical range of RSSI values, and we choose \( k \) to be 0.2. This model assumes that message failure is uncorrelated temporally, which is likely inaccurate, but greatly simplifying.

---

**FIGURE 4** Illustration of error handling and communication timeouts in dataflow graph
Using Equation (1), we can calculate the expected number of required attempts to successfully transmit one message:

\[ n = \frac{1}{p_s}. \quad (2) \]

Assuming a number of bytes per message, \( bpm \), which is 100 in our implementation, and a time per message \( tpm \), which is approximately 200 ms in our implementation, we can calculate the number of bytes per second:

\[ bps = \frac{bpm}{n \times tpm}. \quad (3) \]

Applying Equation (3), we can calculate the bytes/second for each given link in the network graph. However, the reported graph contains RSSI values only for pairs of nodes which are immediate neighbors, so to estimate the communication speed for multihop routes, we first invert our values to obtain seconds/byte and then apply a shortest path algorithm. The result is a pseudoconnected graph where every pair of nodes in the network is attributed an estimate of the number of seconds taken per byte to be sent.

6 | OPTIMAL TASK ALLOCATION

The framework outlined in this paper allows users to specify distributed monitoring jobs for execution within a wireless sensor network. Dividing subtasks in this job between low power leaf nodes in the network, and the powered server node, is referred to in the literature as partitioning. Execution location choice can have a significant impact on the overall efficiency of smart-sensing schemes. Although it is often the case that executing tasks on leaf nodes is advantageous,\(^{22}\) this is not universally true, and depends on the nature of the task and the properties of the network.

In\(^ {23}\) and\(^ {2}\) partitioning of jobs is manually controlled by the user and is reconfigurable. Zimmerman et al.\(^ {10}\) propose an adaptive, market-based algorithm that allows nodes in the network to opt in or out of computational tasks based on their expected utility. This approach allows the solution to vary depending on the network topology and node characteristics. This approach is demonstrated using a decentralised frequency domain decomposition scheme in\(^ {10}\).

Newton et al.\(^ {15}\) propose an automatic partitioning scheme that solves an integer program to minimize a weighted sum of computational and communication costs, subject to a budget on total leaf node CPU. To formulate this problem, operators in a streaming dataflow graph are profiled, to understand their computational and communication costs on typical dummy data. The final result is an assignment for each operator to either server or node.

Similarly to\(^ {15}\), we model jobs as a DAG and profile these jobs to estimate the computational load of each vertex in the graph, and the communication volume of each edge. We additionally collect network statistics that describe the computational capability and communication capacity of each node in the wireless sensor network. Combining these, we solve an integer program that explicitly minimises the total time cost of the job, and assigns each task in the DAG to a specific node in the network, or to the server sink node. In comparison to\(^ {15}\), our optimization solves explicitly for the time cost of the job, is performed automatically for every job submitted to the network, and solves the more general problem of assigning each task to any one of the nodes in the network.

Our centralized approach to execution optimisation is contrasted with the decentralized market-based optimisation proposed in\(^ {10}\). The auction procedure described in\(^ {10}\), is fully decentralized, with nodes passing messages to each other to formulate the execution plan. In our approach, individual nodes pass information about their state to the centralized server, which performs the computation involved in the optimization step. Unlike Zimmerman and Lynch\(^ {10}\), this does require the presence of a centralised server, and cannot be executed in a decentralised way.

6.1 | Integer program formulation

Let \( G \) be the fully connected graph of the sensor network where \( S \) is the set of nodes in the network, and \( E \) is the set of edges. Elements of the set \( E \) are tuples of nodes \((n, m)\) where \( n, m \in S \). We will also refer to elements of the set as \( e_{nm} \) interchangeably. The communication strength between node \( n \) and node \( m \), \( bw(e_{nm}) \) is given by the function:

\[ bw(e_{nm}) \text{ where } bw : E \rightarrow \mathbb{R}. \quad (4) \]
The processing speed of node $n$, relative to the server, is denoted as $sp(n)$.

$$sp(n) \text{ where } sp : S \rightarrow \mathbb{R}. \quad (5)$$

By profiling submitted jobs, we can estimate the computational load of each task in the dataflow graph representation of the job and the volume of data to be transmitted between tasks. We denote this DAG as $M$, with $T$ being the set of vertices, corresponding with tasks in the job, and $D$ the set of edges between tasks. Each task vertex $t_i \in T$ has an associated computational weight given by:

$$w(t_i) \text{ where } w : T \rightarrow \mathbb{R}. \quad (6)$$

Also, let the function defining the volume of data to be transmitted along any edge $d_{ij} \in D$ be given by:

$$v(d_{ij}) \text{ where } v : D \rightarrow \mathbb{R}. \quad (7)$$

Additionally, each task $t_i \in T$ occurs at a given layer in the graph $M$, where the layer number, or the depth, $l$, is denoted:

$$l(t_i) \text{ where } l : D \rightarrow \{0, 1, 2, 3\}. \quad (8)$$

If we allocate a sensor node $s_n \in S$ to each task $t_i \in T$, we can calculate the total execution time for the DAG, by scaling each task’s computational load by the processor speed of the assigned sensor node, and each edge’s communication load by the link strength between the sensor nodes assigned to the vertices at either end of the edge. This allocation of sensor nodes to tasks in the DAG is termed an execution plan. We aim to find the optimal execution plan matching our requirements: The allocation of sensor nodes to tasks in the DAG that has the lowest total execution seconds, summed across all assigned sensor nodes. We formulate this problem as an integer linear programming problem, as the brute force solution quickly becomes intractable as the set of candidate nodes and the network size grows.

For each task in the DAG, we specify a set of candidate nodes, which are eligible to be assigned to this task. Typically, the sampling tasks in the layer zero of the DAG, will each have only a singleton set of eligible nodes; sampling is required to be carried out at a specific sensor node in the network. This also allows the user to manually reduce the size of the integer linear program by specifying logical candidate nodes for each task.

$$\forall t_i \in T \text{ let } C_{t_i} \subset S \quad (9)$$

In our integer-programming formulation, each edge $d_{ij} \in D$ has a set of associated dummy variables $A_{ij}$, each of which represent the assignment (or not) of the parent vertex and child vertex of this edge to a pair of sensor nodes:

$$\forall d_{ij} \in D \text{ let } A_{ij} = \{(d_{ij}, sp, sc) \in d_{ij} \times C_{t_i} \times C_{t_j} \} \quad (10)$$

where $sp \in C_{t_i}$, $sp \in C_{t_j}$,

Given this set of dummy variables for each edge, we now define a function giving the processing cost for each dummy variable in the set:

$$\gamma_p(d_{ij}, sp, sc) = \frac{w(t_i)}{sp} \quad (11)$$

where $w(t_i)$ is the computational load of task $t_i$, and $sp$ is the computing speed of sensor $sp$. In addition to transmission cost, we define the processing cost for each dummy variable:

$$\gamma_p(d_{ij}, sp, sc) = \frac{v(d_{ij})}{bw(sp, sc)} \quad (12)$$

where $v(d_{ij})$ is the volume of data to be transmitted from task $t_i$ to task $t_j$, and $bw(sp, sc)$ is the estimated bandwidth between sensor node $sp$ and sensor node $sc$. Combining Equations (11) and (12), we can calculate the total cost for each dummy variable as:

$$\gamma(d_{ij}, sp, sc) = \gamma_t(d_{ij}, sp, sc) + \gamma_p(d_{ij}, sp, sc). \quad (13)$$
The total cost of all dummy variables in the in the set $A_{ij}$, corresponding with a given edge $d_{ij}$, can now be expressed by combining Equations (10) and (13):

$$\text{Cost}(d_{ij}) = \sum_{\alpha \in A_{ij}} \alpha \cdot \gamma(\alpha).$$

(14)

To ensure that our integer program logically corresponds to the task allocation problem we need to add constraints to satisfy two criteria: Edge uniqueness and vertex consistency.

**Edge uniqueness**

For each edge in the graph to be assigned to only one parent node and one child node, we need to ensure that for each edge $d_{ij} \in D$ only one corresponding dummy variable in the set $A_{ij}$ is given a value of one. We can express this as a simple linear constraint that requires that for each edge, the sum of associated dummy variables equals exactly one.

$$\forall d_{ij} \in D \sum_{\alpha \in A_{ij}} \alpha = 1$$

(15)

**Vertex consistency**

Vertex consistency requires that for each pair of edges that share a common vertex $t_i \in T$, the assigned child sensor node of the incoming edge is the same as the assigned parent node of the outgoing edge. Let $A_g$ be the union of all dummy variables sets:

$$A_g = \cup \{ A_{ij} | i,j \in D \}.$$

(16)

For every task $t_i \in T$, find the set $in_{t_i}$ of dummy variables corresponding with the incoming edge, and the set $out_{t_i}$ of dummy variables corresponding with the outgoing edge:

$$\forall t_i \in T$$

$$in_{t_i} = \{ \alpha \in A_g | \phi_{in}(\alpha, t_i) \}$$

where $\phi_{in}((d_{ij}, s_p, s_c), t_i) \rightarrow i == t$

$$out_{t_i} = \{ \alpha \in A_g | \phi_{ou}(\alpha, t_i) \}$$

where $\phi_{ou}((d_{ij}, s_p, s_c), t_i) \rightarrow j == t.$

(17)

For a given task $t_i$, we calculate the sets of dummy variables corresponding with the incoming and outgoing edges respectively using Equation 16. We can now find the set of pairs of dummy variables that are inconsistent: The assigned child of the incoming edge is not the same as the assigned parent of the outgoing edge. For each of these inconsistent pairs of dummy variables we add a constraint to ensure that the pair are mutually exclusive.

$$\forall t_i \in T \text{ let } K_{t_i} = \{ (d_{ij}, s_p, s_m), (d_{jk}, s_q, s_d) \in in_{t_i} \times out_{t_i} | s_c \neq s_q \}$$

(18)

**Bottleneck penalization**

Recall from Equation (8) that $l(t_i)$ describes which layer of the computational graph a task is in. If two or more tasks in the same layer are assigned to the same sensor node, this creates a bottleneck. Although our sensor-node firmware enables concurrent execution of tasks, there is still a performance penalty. If two parent nodes attempt to transmit data to the same child node, the time required for successful transmission will be approximately doubled, compared with transmission to two distinct child nodes with equivalent wireless connectivity. This bottleneck occurs, despite the sensor nodes’ ability for concurrency, primarily because the wireless radios operate at a constant baud rate of 9,600, although there are other factors, such as increased collision rate.

To account for this, we introduce a bottleneck penalty that discourages solutions that assign tasks in the same layer to the same nodes. Recall that each dummy variable $\alpha$ is associated with a triple $(d_{ij}, s_p, s_c)$, where $d_{ij}$ is an edge in $M$ and the computational DAG, $s_p$ and $s_c$ are the nodes assigned to the parent and child tasks of $d_{ij}$ respectively. To add
this bottleneck penalty, we create bottleneck dummy variables \( b_{a_1,a_2} \in B \), each one corresponding with a pair of dummy variables \((a_1, a_2) \in A_g \times A_g\) for which the layer of the child tasks is equal, and the child node assignments are equal:

\[
\forall (a_1, a_2) \in \{(d_{ij}, s_{p1}, s_{c1}), (d_{nm}, s_{p2}, s_{c2}) \in A_g \times A_g\}
\[

l(t_j) = l(t_m) \wedge s_{c1} = s_{c1} \}
\exists b_{a_1,a_2}.
\]

Each of these bottleneck dummy variables can be constrained to be a logical and of its associated pair of assignment dummy variables, by adding the following three constraints to our linear program for each bottleneck dummy variable:

\[
\forall b_{a_1,a_2} \in B
\[

b \geq a_1 + a_2
\]

\[

b \leq a_1
\]

\[

b \leq a_2
\]

By imposing these three constraints, the binary bottleneck dummy variables are required to take a value of 1 when both of the associated assignment variables are 1 but a value of 0 otherwise. Finally, for each bottleneck dummy variable the associated cost is given by:

\[
\psi(b_{a_1,a_2}) = \min(\gamma_l(a_1), \gamma_l(a_2)),
\]

where \( \gamma_l \) is a function giving the transmission cost for an assignment dummy variable. Here, the penalty cost for a bottleneck between two assignment variables that share a common child node at the same level is the lesser of the two communication costs. Consider a bottleneck occurring between two edges, one with a large transmission cost and one with a much smaller transmission cost. It follows logically that the bottleneck penalty in this situation should be the lesser of these costs; once the edge with the smaller communication cost has successfully transmitted data, there will be no slowdown of the other tasks transmission.

Combining our cost function definition from Equation (13), with vertex consistency constraints from Equation (18), edge uniqueness constraints and bottleneck penalties, we define our integer linear program as:

\[
\alpha = \text{argmin} \sum_{d_{ij} \in D} \sum_{a \in A_{ij}} a \cdot \gamma(a) + \sum_{b_{a_1,a_2}} b \cdot \psi(b)
\]

subject to \( \forall d_{ij} \in D \)

\[
\sum_{a \in A_{ij}} a = 1
\]

\[
and \ \forall t_i \in T
\]

\[
\forall (x, y) \in K_{t_i}
\]

\[
x + y \leq 1
\]

\[
and \ \forall b_{a_1,a_2}
\]

\[
b \geq a_1 + a_2
\]

\[
b \leq a_1
\]

\[
b \leq a_2.
\]

This approach optimizes for the summed time cost of executing a job, as a proxy for the energy cost. Depending on the desired performance characteristics of the network, the proposed optimization formulation may be modified. For example, this approach can be straightforwardly extended to explicitly account for the energy cost, by incorporating each node’s active energy draw into the cost functions in Equations (11) and (12).
In the implementation described in this paper, each node has identical energy consumption characteristics, and so the summed time cost formulation is equivalent. However, in networks with heterogeneous node hardware, or where processors allow for programmable changes in clock speed which reduce the energy draw, incorporating node energy consumption into the cost function would be recommended. An interesting extension of the optimization formulation would be to additionally optimize not just the chosen node for each task, but also its operating clock speed.

7 | RESULTS

Evaluation of the system is carried out on an experimental steel tower structure located in Woburn, Massachusetts. A schematic of this braced frame steel tower is shown in Figure 5. The tower is equipped with 24 wireless sensor node prototypes, as well as two electronically controlled shakers, capable of exciting the structure programatically. The sensor node prototypes on this structure each contain a Toradex Colibri iMX6 processing core, a XBee series 1 radio, and a STMicro LIS344ALH triaxial MEMS accelerometer, with a sampling rate of 2,000 Hz. These prototype nodes can be powered either via a power over ethernet connection, or a 5V battery power source. A laptop computer is used as a portable sink node, running the gateway server software, including the optimization engine.

**FIGURE 5** Schematic of experimental steel tower showing location of sensor nodes and accelerometers

**FIGURE 6** Comparison of predicted and observed total execution times for the node strategy and the server strategy for increasing values of node processing speed relative to server processing speeds
7.1 | Single node

To demonstrate the adaptability of our proposed optimization scheme to changing node characteristics, we first conduct an experiment using a single node in the network. In this experiment, the monitoring job specification requires the sampling of uniaxial acceleration data at 2000 Hz, followed by the computation of the single-sided FFT, rounded to four decimal places in the mapping step, and a simple no operation in the reducing step.

The processing speed of the node is varied throughout this experiment in steps. The optimization algorithm is run at each step to decide whether to execute the mapping step on the node or on the server. In this experiment, we denote the strategy of computing the mapping step on board the node as the node strategy and the strategy of sampling at the node and computing the mapping step on the server as the server strategy. Figure 6, shows both experimentally observed total execution times, omitting the time taken to sample the data, as this is a constant, and predictions outputted from the integer programming solver, for varying speeds of the node processor. The slowdown ratio describes the speed of the node relative to its maximal capability of 1.0, which is still considerably slower than the server node (approximately 0.05 the speed).

At very low node processing speeds, the cost of computing the FFT onboard the node, and thereby shrinking the volume of data required to be transmitted, is too high to make the node strategy worthwhile. A 0.01 slowdown ratio is outside the typical variation of individual microprocessors. However, the default clock speed of the node hardware used in this experiment operates at a relatively high clock frequency of 396 MHz, and so this slowdown ratio approximately represents a sensor node processor operating 4 MHz. As the speed of the node processor increases, and the cost of computing the FFT decreases, the advantage gained in reducing communication volume is realized. At the maximal node processing speed, the node strategy is approximately 25% more efficient than the server strategy. Good agreement between experimentally observed execution times, and predictions, is shown at each step. Despite no change in application logic, the optimization algorithm adapts to changing node properties, and succeeds in recommending the optimal strategy at each step.

![Figure 7](image-url)
Whereas the application logic specified in this example is quite simple, and offers only limited data reduction, this example demonstrates how the tracking of node properties allows the optimisation algorithm to adapt and improve performance under changing environmental or operating conditions.

7.2 | Eight-node network

In this experiment, a more realistic SHM job is considered. Here we analyze a subnetwork of eight nodes on the steel tower. The job specification requires uniaxial acceleration data to be sampled at 2000 Hz for 2 s in the sampling step. In the mapping step, the FFT of this time series is computed, and a key-value pair consisting of a zero key, and a value tuple of the node ID and the complex magnitude of the FFT at a specific frequency is returned. Finally, at the reducing step, a spectral matrix is formed from each nodes frequency response value, and is decomposed to find the modeshape of the eight-node system. This is an implementation of the frequency domain decomposition algorithm,\(^7\) in our MapReduce style framework.

In this experiment, we consider how the node at which the reduce step is executed affects the performance of this algorithm. Surprisingly, as is shown in Figure 7a, the optimal node assignment for the reduce task is not the server node, but rather a leaf node (Node 21), which provides for roughly 50% more efficient total performance. As in the previous

![Figure 8](image)

**FIGURE 8** Comparison of predicted and observed probability of message failure for varying values of received signal strength indicator (RSSI)

![Figure 9](image)

**FIGURE 9** Implementation of the decentralised frequency domain decomposition algorithm, illustrating how the network is divided into overlapping neighborhoods, allowing parallel estimation of modeshapes for each neighborhood which are ultimately combined to return a noisy version of the full network modeshape.
experiment, the optimal strategy outputted by the integer program outperforms all others, and predicted and experimentally observed execution times match well. Note here that the total execution time is the sum of the time taken at each node, not including the time taken to sample the data, as this is a constant. Note also that not any leaf node will provide a performance improvement: Choosing poorly (for example Node 41), results in major performance penalties.

The sampling rate of the MEMS accelerometer used in this study is higher than typical for modal analysis applications. However, the results presented here are still instructive when using a lower sampling rate. We compare the performance of computing the map step and reduce step within the network, versus computing the map step on board the sensor nodes, and the reduce step at the central server. As a result, for a lower sampling rate application, there would be only a small change in the execution time: The communication cost would not change, and the computational cost would slightly decrease.

In Figure 7b, a force-directed graph of the network connectivity is shown. Here, distance between nodes indicates poor wireless connectivity. Select values of RSSI are shown. Examining this graph, we see that the poor connectivity of the server node may explain why executing the reduce step of the frequency domain decomposition (FDD) algorithm on a leaf node offers better performance. The low degree and strength of connectivity of Node 41 also may explain why it is a particularly poor choice in comparison to the optimal, and well-connected choice, of Node 21.

Figure 8 shows experimental message failure rates at various different observed RSSI values throughout this experiment. Plotted alongside these experimental observations is the model described in Equation (1).

This example demonstrates how the computational framework proposed in this paper can be used to easily specify common SHM methodologies. We also show how optimal assignment of tasks to nodes can yield non-obvious results which significantly improve performance.
7.3 | 24-node network

In this experiment, we expand the previous eight-node frequency domain decomposition to demonstrate the performance benefits of our system with increasing scale. Here, we again wish to calculate modeshapes using frequency domain decomposition, but in this experiment, we consider all three axes of acceleration, for network sizes of 8, 16, and 24.

FIGURE 12 Calculated modeshapes for the 15.1 Hz vibration mode, with sensor locations illustrated. (a) shows the modeshape calculated with the centralised FDD algorithm. (b) shows the modeshape calculated with the decentralised algorithm. (c) shows the static configuration of the tower.
As the network size increases, the size of the spectral matrix in the FDD calculation becomes too large to fit in the limited memory of our sensor nodes. As in,\textsuperscript{10} we adopt a decentralized frequency domain decomposition (DFDD) approach. Each node in the network is assigned a neighborhood number from 1 to 4 (updated over the air). There is an overlap of exactly one node between neighborhoods, that is, three nodes are assigned to two distinct neighborhoods each. The DFDD algorithm can then easily be derived from the FDD algorithm described in the previous section. All that is required is to modify the mapping function to emit key-value pairs where the key is the neighborhood, and the value is, as before, a tuple of the node ID and the FFT magnitude. Each reducer then operates in parallel to compute the spectral decomposition for an individual neighborhood. The final result can be obtained by summation of the individual neighborhood modeshape estimates, noting that each modeshape overlaps with another at exactly one node. If the overlap is greater than one node, a least squares approach can be adopted to reconcile the individual neighborhood modeshape estimates. A schematic representation of the decentralized frequency domain decomposition algorithm is shown in Figure 9.

For each network size, the DFDD algorithm is compared to the FDD algorithm. For the DFDD algorithm, the location of the four reducers, one for each neighborhood, is decided optimally by the integer program solution. For the FDD algorithm, the reduce location is fixed to be the server node. Figure 10 shows the topology of the 24-node network, with the locations of the four reducer nodes chosen by the optimization algorithm shown in red.

Figure 11 shows both predictions outputted by the optimization algorithm, and 95% confidence intervals for experimentally observed data. The DFDD algorithm outperforms the FDD algorithm as the size of the network grows. This is primarily due to the bottleneck phenomena occurring in the FDD algorithm. In addition to this, as in the previous section, the server node suffers from relatively poor connectivity, which exacerbates the bottleneck. In the DFDD implementation, the reducers are both better connected, and receive data from fewer mappers simultaneously, leading to improved performance over the FDD implementation.

Although the DFDD approach improves performance, by dividing the modeshape calculation into substructures, this approach may lead to inaccuracies in the final calculated modeshape. In Figure 12a, the modeshape calculated at 15.1 Hz using the FDD approach is shown. Figure 12b shows the same modeshape calculated using the DFDD approach. The modal assurance criterion value for these two mode shapes is 0.96.

This experiment shows how the computational framework developed in this paper can be easily adapted to implement highly distributed algorithms such as decentralised frequency domain decomposition. In this case, for a 24-node network, the parallelization of the modeshape calculation leads to lower memory consumption, allowing better positioned leaf nodes to participate, and thereby improving efficiency compared to the centralized algorithm.

Experimentally calculated modeshapes are shown for the dominant 15.1 Hz mode of the steel braced frame tower. Good agreement is shown between modeshapes calculated using FDD and those calculated using decentralized FDD.

8 | SUMMARY AND CONCLUSIONS

In this study, a novel computational framework for wireless smart sensing is developed and presented. This framework allows users of a wireless sensor network to specify monitoring application logic in a high-level MapReduce style to be executed in a distributed fashion in the network. This framework enables users to quickly develop distributed sensing applications which utilize the computational resources of the network. Users can conveniently specify their jobs in Python code and submit them remotely via the internet. Once submitted, the network is reprogrammed on the fly by the framework, and results are returned to the user via the internet once available.

Careful consideration of fault tolerance is given in the development of the computational framework. Exceptions arising from errors in user-defined jobs, from communication timeouts and other sources are propagated transparently back to the user. To facilitate the implementation of this computational framework, a novel node level scheduling system is developed. This system, which is built on the MicroPython compiler and environment, uses a coroutine-based cooperative scheduling model to provide the necessary multitasking to implement the framework.

Taking advantage of the structured nature of the MapReduce style computational model, an optimal task assignment algorithm is developed to minimize the total time taken, and therefore the total power consumed, by submitted jobs. Each job is modeled as a DAG. By incorporating real-time network performance statistics, we formulate and solve an integer linear program. This algorithm optimally assigns each task in DAGs to a node in the network.

Finally, a real deployment of this system on a braced frame steel tower is presented. We experimentally demonstrate the use of the computational framework in specifying typical SHM applications.
The accuracy and utility of the optimal task allocation algorithm is demonstrated for three separate experiments. First, we show how the algorithm adapts to varying node characteristics, for a single node example. Second, we present an implementation of the frequency domain decomposition algorithm for an eight-node network; here, the optimal allocation of the matrix decomposition task significantly improves performance. Lastly, we demonstrate an implementation of the decentralized frequency domain decomposition for varying network sizes. This is compared with a centralized FDD algorithm. By parallelizing the decentralized FDD algorithm and optimally locating the matrix decomposition step for each cluster of nodes in the decentralized FDD scheme, significant performance increases are shown.

Future work will include the implementation of more complex SHM methodologies in the presented computational framework, as well as further development of the optimal task allocation algorithm by considering not only the total time cost, but also the available battery life of each node.

ACKNOWLEDGEMENTS

The authors acknowledge the support provided by Royal Dutch Shell through the Massachusetts Institute of Technology Energy Initiative and thank Dr Dirk Smit for his support, vision, and many fruitful discussions. Thanks also to Dr Lorna Ortiz-Soto and Dr Xander Campman for their support and oversight. Thanks to Justin Chen, Reza Mohammadi, and Hao Sun for their assistance in data collection. Special thanks to Murat Uzun for many helpful discussions, and to Dr Abel Sanchez, Prof Daniele Veneziano, and Prof John Williams for their guidance. Also, thanks to Jay Petrie, Jerry Earl, Lewis Mendall, William Moyne, Chris Boutiette, Michelle Estaphan Owen, and the team at Shell Techworks for their collaboration in the development of the experimental structural model and hardware.

ORCID

James Long  
https://orcid.org/0000-0002-1028-8322

Oral Büyüköztürk  
https://orcid.org/0000-0002-7712-7478

REFERENCES

1. Pakzad SN, Fenves GL, Kim S, Culler DE. Design and implementation of scalable wireless sensor network for structural monitoring. *Journal of infrastructure systems*. 2008;14(1):89-101.

2. Flouri K, Saukh O, Sauter R, et al. A versatile software architecture for civil structure monitoring with wireless sensor networks. *Smart Structures and Systems*. 2012;10(3):209-228.

3. Nagayama T, Sim SH, Miyamori Y, Spencer B Jr. Issues in structural health monitoring employing smart sensors. *Smart Structures and Systems*. 2007;3(3):299-320.

4. Gao Y, Spencer B, Ruiz-Sandoval M. Distributed computing strategy for structural health monitoring. *Structural control and health monitoring*. 2006;13(1):488-507.

5. Rice JA, Mechitov K, Sim SH, Spencer BF Jr, Agha G. Enabling framework for structural health monitoring using smart sensors. *Structural Control and Health Monitoring*. 2011;18(5):574-587.

6. Rice JA, Mechitov K, Sim SH, et al. Flexible smart sensor framework for autonomous structural health monitoring. *Smart Structures and Systems*. 2010;6(5-6):423-438.

7. Brincker R, Zhang L, Andersen P. Modal identification of output-only systems using frequency domain decomposition. *Smart materials and Structures*. 2001;10(3):441.

8. Jo H, Sim SH, Nagayama T, Spencer B Jr. Development and application of high-sensitivity wireless smart sensors for decentralized stochastic modal identification. *Journal of Engineering Mechanics*. 2011;138(6):683-694.

9. Sim SH, Carbonell-Márquez JF, Spencer B Jr, Jo H. Decentralized random decrement technique for efficient data aggregation and system identification in wireless smart sensor networks. *Probabilistic Engineering Mechanics*. 2011;26(1):81-91.

10. Zimmerman AT, Lynch JP. Market-based frequency domain decomposition for automated mode shape estimation in wireless sensor networks. *Structural Control and Health Monitoring*. 2010;17(7):808-824.

11. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM*. 2008;51(1):107-113.

12. George DP. Micropython. GitHub; 2018. https://github.com/micropython.

13. White T. Hadoop: The definitive guide. "O’Reilly Media, Inc."; 2012.

14. Lämmel R. Google’s MapReduce programming model–Revisited. *Science of computer programming*. 2008;70(1):1-30.

15. Newton R, Toledo S, Girds L, Balakrishnan H, Madden S. Wishbone: Profile-based Partitioning for Sensornt Applications. In: *NSDI*. 2009;9:395-408.

16. Rice JA, Spencer B. Structural health monitoring sensor development for the Imote2 platform. In: Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2008, vol. 6932 International Society for Optics and Photonics; 2008. p. 693234.

17. Kim J, Swartz A, Lynch JP, Lee JJ, Lee CG. Rapid-to-deploy reconfigurable wireless structural monitoring systems using extended-range wireless sensors. *Smart Structures and Systems*. 2010;6(5-6):505-524.
18. Levis P, Madden S, Polastre J, Szewczyk R, Whitehouse K, Woo A, Gay, D., Hill, J., Welsh, M., Brewer, E. and Culler, D. TinyOS: An operating system for sensor networks. In: Ambient intelligence Springer; 2005.p. 115–148.
19. Wang Y, Lynch JP, Law KH. A wireless structural health monitoring system with multithreaded sensing devices: design and validation. Structure and Infrastructure Engineering. 2007;3(2):103-120.
20. Linderman LE, Mechetov KA, Spencer BF. TinyOS-based real-time wireless data acquisition framework for structural health monitoring and control. Structural Control and Health Monitoring. 2013;20(6):1007-1020.
21. Fu Y, Mechetov K, Hoskere V, Spencer Jr B. Development of RTOS-based wireless SHM system: Benefits in applications. In: Proceedings of the International Conference on Smart Infrastructure and Construction, Cambridge, UK; 2016. p. 27–29.
22. Lynch JP, Sundararajan A, Law KH, Kiremidjian AS, Carryer E. Embedding damage detection algorithms in a wireless sensing unit for operational power efficiency. Smart Materials and Structures. 2004;13(4):800.
23. Whitehouse K, Tolle G, Taneja J, Sharp C, Kim S, Jeong J, Hui, J., Dutta, P. and Culler, D. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In: Proceedings of the 5th international conference on Information processing in sensor networks ACM; 2006. p. 416–423.

How to cite this article: Long J, Büyükoztürk O. A power optimised and reprogrammable system for smart wireless vibration monitoring. Struct Control Health Monit. 2020;27:e2468. https://doi.org/10.1002/stc.2468