A COMPARISON OF SERIAL VERSUS PARALLEL ALGORITHMS FOR ENERGY CONSUMPTION IN WIRELESS SENSOR NETWORKS

By

Gregg Reavis

A thesis submitted in partial fulfillment

of the requirements for the degree of

MASTER OF SCIENCE In Computer Science

Middle Tennessee State University

May 2016

Thesis Committee:

Dr. Dr. Yi Gu, Chair

Dr. Chrisila Pettey

Dr. Sung Yoo

ACKNOWLEDGEMENTS

Dr. Yi Gu has kindly guided me through this work and first introduced me to parallel processing. Dr. Chrisila Pettey and Dr. Sung Yoo have been kind enough to serve as my committee. I am also grateful to all my committee members for their valuable comments and suggestions.

ABSTRACT

The majority of low-end sensors in wireless sensor networks (WSNs) operate on batteries, which either cannot be replaced or are not practical to replace. Therefore, it is important to measure the total energy consumption in WSNs, in order to minimize power consumption and maximize network lifespan. Many researchers have been devoting their efforts into this area, which shows that a heterogeneous network produces a better solution to prolonging the network lifespan. So far as we know, the algorithms for minimizing the energy consumption have all been implemented in serial algorithms. In this work, we propose a parallel programming approach for optimizing the minimum energy consumption and maximizing the lifespan of WSNs. The results from an extensive set of experiments on a large number of simulated sensor networks illustrate the performance superiority of the proposed parallel approach over an existing serial algorithm and confirms a parallel solution will provide faster results.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

BS – Base Station

CH – Cluster Head. A sensor that gathers data from LN

DCC – Distance Distance-based Crowdedness Clustering

LN – Leaf Node, the low-end sensors within a cluster, usually battery-supported

RFID – Radio Frequency Identification

TEC – Total Energy Cost

WSN – Wireless Sensor Network

CHAPTER I

INTRODUCTION

A wireless sensor network (WSN), is a collection of low-cost, low-power, multifunctional sensor nodes that are small in size and designed to communicate for short distances over low-power radio transmitters. Each sensor consists of at least some type of radio transceiver and microcontroller, which includes a radio frequency identification (RFID) type of sensor. Sensors range in size with some simple ones so small they have been attached to worker bees to track their flights [1].

However, the sensors typically include a battery to power the radio, microcontroller, etc. These types of sensors' batteries are usually difficult, if not impossible, to replace. In the case of a device for monitoring a nuclear facility[2] due to the exposure to radiation, it would never be touched after the initial placement. With the batteries unchangeable, it is necessary to employ some techniques to optimize the battery life of each sensor by minimizing the amount of energy consumed, such as sharing the radio transmission load.

These distributed sensor networks are used in diverse areas such as agriculture, industry, civil and the military, because they enable reliable and consistent monitoring of the environment. For example, in the agricultural environment, the sensors are used to relay soil data to decide when to use irrigation and fertilizers. Industry uses sensors to monitor items as diverse as machinery and nuclear facilities. Civil uses include forest fire detection and water quality. Military uses include troop movement and perimeter policing.

Another example is a WSN network is set to monitor a volcano[3]. In many cases, the batteries in the wireless sensors either cannot be replaced, because they are soldered in place, or it is not practical to replace them because of the large quantity and random locations. For monitoring a volcano, even if the batteries could be replaced, once it begins to erupt there is a likelihood of poisonous gases, extreme heat, and rivers of magma. Therefore, the longer the lifespan of the network the longer it will produce useful data.

As far as we know, the algorithms for minimizing energy consumption have all been implemented as serial algorithms. In this work, we will use parallel programming techniques to optimize energy consumption and maximize the lifespan of the wireless sensors and will achieve the results faster than a serial algorithm.

CHAPTER II

BACKGROUND

Wireless sensor networks are comprised of a large quantity of inexpensive wireless sensors, and there are different ways to organize the sensors. Some networks may also deploy a small quantity of expensive but more powerful sensors to partition the entire WSN into several clusters and form a hierarchical structure [4]. The three common approaches considered and proposed are (a) having all low-level sensors in the network and forming several clusters with fixed cluster heads [5], (b) having all low-level sensors in the network and forming several clusters with dynamically rotated cluster heads [4, 6], and (c) having a mixture of high-level and low-level sensors in the network with high-level sensors being cluster heads [4, 7].

In paper [4] with a homogeneous network, where all sensors are the same, it is necessary and important to optimize the energy consumption in the network to prolong the battery life. The LEACH (Low-Energy Adaptive Clustering Hierarchy) algorithm is proposed to let the sensors take turns being the cluster heads to spread out and balance the energy cost of transmitting the data throughout the network. It further reduces the energy required by aggregating the data and compressing it before transmission. With this scheme, there is a factor of 8 energy reduction as compared to having the sensors directly transmit to the base station.

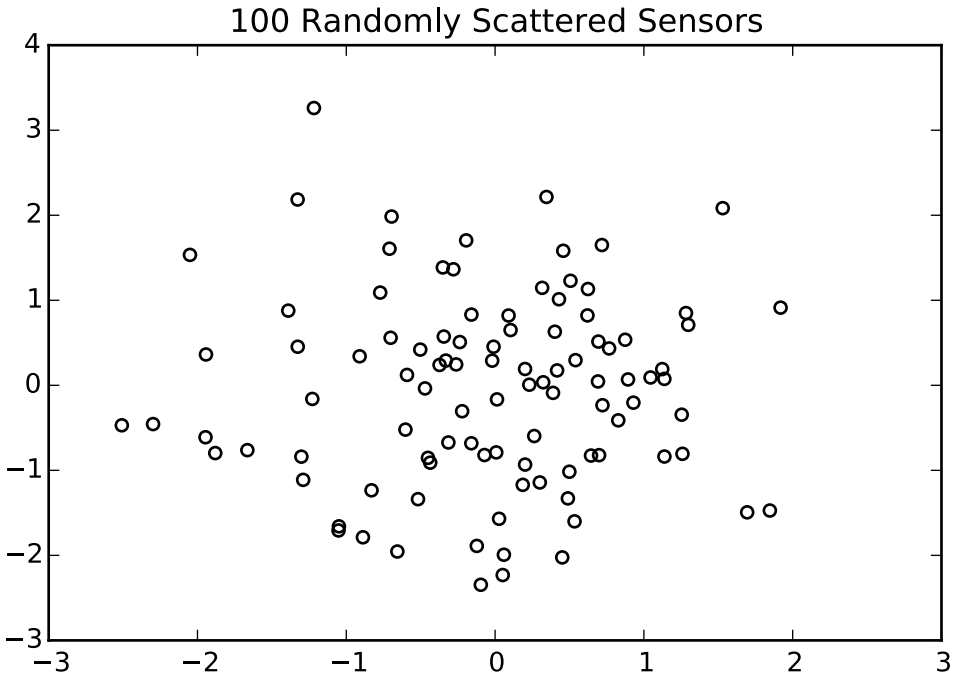

Figure 1: 100 Randomly Scattered Sensors

As Figure 1 shows, there are 100 randomly scattered sensors within that square area. With the LEACH algorithm, all of these sensors are the same and take turns collecting the data from the other sensors. This strategy improves the lifespan of the entire network but does have some disadvantages as shown in Figure 2.

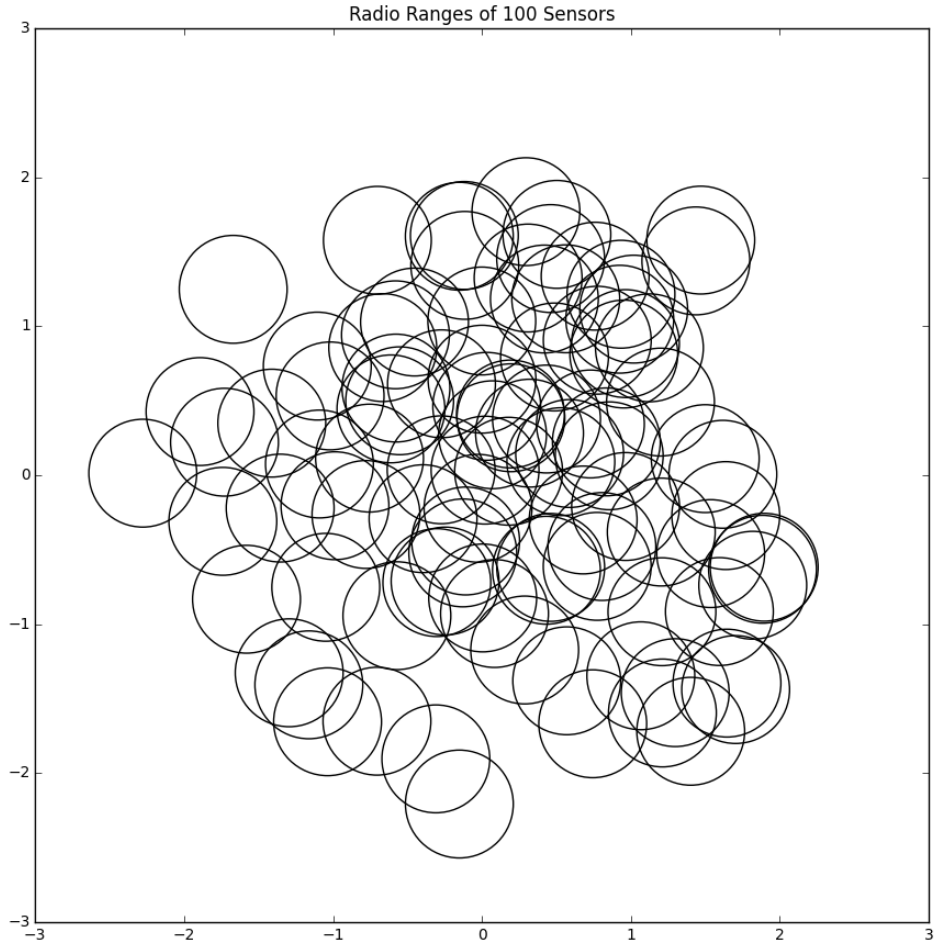

Figure 2: The radio ranges of 100 sensors

In figure 2, the circles represent the radio ranges of the100 sensors. While most of the ranges overlap, and thus have good coverage, there are two problem areas. In the upper left, one sensor's radio range does not overlap with others, so it will always have to transmit its data to the base station by itself and thus will not have the benefit of sharing the load. At the bottom of the sensor field, there is one sensor that only overlaps with one

other sensor. In this case, it will not be practical to have it share the load so the only other sensor it overlaps with will always have to gather data and transmit it. These types of situations will cause the sensor network to have a less than optimal lifespan.

For this paper, the one-hop sensor network will be similar to the one in Figure 3.

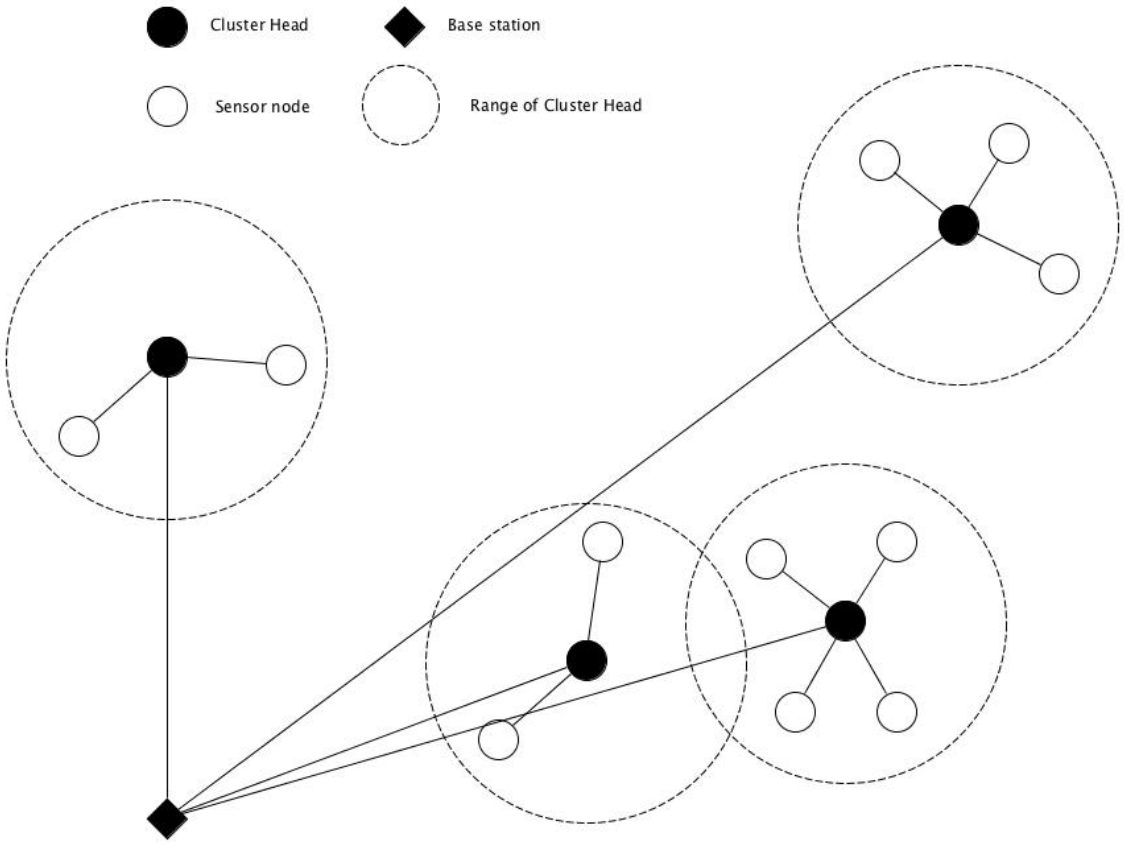

Figure 3: A one-hop wireless sensor network

In figure 3, the sensors are either leaf nodes or cluster heads. The sensors designated cluster heads collect data for themselves, receive the data from the LNs, aggregate the data, and then compress it before sending it on. This extends the battery life of the LNs because they only need to transmit a short distance. Since the LNs have a very

limited radius range, it is very important to determine the optimal placement of the cluster heads to cover the entire WSN.

Many researchers have been looking into the advantage of using parallel processing in improving the performance of WSNs in various ways. For instance, parallel processing is implemented to ascertain the location of nodes within a WSN to minimize the need for GPS [8]. It is used within a WSN for pattern recognition [9] as well as data detection [10]. Parallel processing can also be incorporated into kNN queries in a simulator to determine energy efficiency [11] and into a GPU architecture to simulate energy consumption wireless networks [12]. However, so far as we know, parallel processing has not been applied to minimize total energy consumption in WSNs yet.

CHAPTER III

METHODS

Current approaches to optimizing energy consumption in wireless sensor networks use a serial approach[4], but we feel a parallel approach would be more scalable and lead to better optimization. The first step is to take an existing serial algorithm[13] and apply parallel techniques to it. Thus this preliminary research seeks to confirm that a parallel solution will produce the same result as the serial solution, in terms of the energy consumption, but in less time.

Our WSN will be set up with all the sensors randomly placed, within a 200m x 200m area (LxL) with full battery power and the base station is in place with no restrictions on energy consumption. Neither the sensors nor the base station will change locations. With those prerequisites[13], we will seek to determine which nodes will be designated as CH to minimize total energy consumption.

Our serial algorithm[13] is a heuristic based on the distance from the LN to the CH and the number of sensor neighbors. Each sensor is examined for its nearest neighbors with the possibility of being a CH and the energy consumption is calculated for each cluster. Once all possible CH combinations have been calculated, the minimal energy consumption is determined.

This Distance-based Crowdedness Clustering (DCC) algorithm is shown in figure 4. This serial algorithm is our starting point for a parallel optimization method. For this

```
Algorithm 1. Distance-based Crowdedness Clustering

Input: a sensor network G = (V,E) with n LNs randomly deployed in a
L × L (m^2) square region and one BS deployed inside or outside the
region.
Output: the optimal number k and location of CHs with minimum TEC.

0:  Initialize adjacency matrices
1:  Calculate all-pair distances d_{i,j}, for v_i,v_j ∈ V, in an array A_d;
2:  Initialize minimum TEC TEC_min = +∞;
3:  for all distances d_{i,j} ∈ A_d do
4:  Set cut-off distance d_cut = d_{i,j} ;
5:  Set v_m as a neighbor of v_n if d_{m,n} ≤ d_cut  for all m,n ∈ V;
6:  Sort all v ∈ V according to the number of neighbors in a
decreasing order and place them in an array A_v;
7:    Insert all v ∈ V in an unclustered sensor queue Q_u;
8:    Initialize a clustered sensor queue Q_c = 0;
9:    Initialize the number of clusters n_clusters = 0;
10:   while Q_u  != 0 do
11:         Retrieve v_k ∈ Q_u from Av and designate it as a CH;
12:         Form a cluster C_k of v_k and its neighbors v_l ∈ Q_u;
13:         Insert all v ∈ C_k in Q_c;
14:         Remove all v ∈ C_k from Q_u;
15:         n_clusters ++;
16:   end while
17:   Calculate the TEC;
18:   if TEC_min > TEC then
19:         TEC_min =TEC;
20:         k = n_clusters;
21:   endif
22:  endfor
23:  return k and location.
```

Figure 4: DCC, the serial algorithm for parallelization[13]

application, we chose to use MPI to implement parallel techniques. To begin with, we will initialize our data arrays using parallel techniques, which occurs in line zero of the

listing in figure 4. In line 6, there is a sorting technique employed, which is an odd-even merge sort in parallel. Lastly, in lines 17 - 21, for the total energy consumption the outer loop is performed in parallel.

For the parallel initialization of the adjacency matrix, the rows are divided up amongst the available processes. Each row is initialized and then the rows are gathered to all the processes. Since the number of processes and number of rows are unknown, it is necessary to use a variable gather routine to distribute the adjacency matrix to all processes. See Figure 5.

```
For each process:
numRowsPerProcess = integer(numberOfRows / number of processes);
extraRows = remainder(numberOfRows / number of processes);
if LastProcess
       numRowsPerProcess =  numRowsPerProcess + extraRows;
initAdjacencyMatrix(numRowsPerProcess);
MPI_Allgatherv(gather all rows to the root);
Root process then determines what nodes are connected.
```

Figure 5: Parallel initialization

The sorting routine for the serial algorithm is a heap sort, while an odd-even merge sort is used for the parallel implementation. The array of distances, a two dimensional adjacency array ranging in size from 10 x 10 to 900 x 900, see Table 1, is scattered to the processes, sorted and then gathered back.

Figures 6a-d show the odd-even merge sort broken down into functions. The function names are in bold face.

```
pseudocode for odd-even merge sort

// sort the distances in increasing order
// this will run on each process

oddEvenMergeSort
   MPI_Scatter(arrayOfDistances,
       numberOfArrayElements/numberOfProcessors, localArrayOfDistances)
   merge_sort(localArrayOfDistances,
        numberOfArrayElements/numberOfProcessors )

   for 1 to numberOfProcessors Do
        if processorRank is even
            pairExchange (numberOfArrayElements/numberOfProcessors,
            localArrayOfDistances, processorRank, processorRank + 1)
        else
            pairExchange(numberOfArrayElements/numberOfProcessors,
            localArrayOfDistances, processorRank - 1, processorRank )

    MPI_Gather(localArrayOfDistances,
    numberOfArrayElements/numberOfProcessors, arrayOfDistances )
```

Figure 6a: Odd-even merge

```
pairExchange
    if processorRank == sendingRank
        MPI_Send(localArrayOfDistances,
           numberOfArrayElements/numberOfProcessors, receiveRank,
            mergeTag)
        MPI_Recv(localArrayOfDistances,
           numberOfArrayElements/numberOfProcessors, receiveRank,
            sortTag)
    else
        MPI_Recv(recvArrayOfDistances,numberOfElementsRecvArray, sendRank,
           mergeTag)
        merge(localArrayOfDistances, numberOfElementsLocalArray,
           recvArrayOfDistances, numberOfElementsRecvArray, mergedArray)
        if sendingRank > processorRank
            theOtherPairStart = numberOfElementsRecvArray
            thisPairStart = 0
        else
            theOtherPairStart = 0
            thisPairStart = numberOfElementsRecvArray
        MPI_Send(mergedArray[theOtherPairStart], numberOfElementsRecvArray,
           sendingRank, sortTag)
        for i = thisPairStart to (thisPairStart +
           numberOfElementsLocalArray)
            localArrayOfDistances[i - thisPairStart] = mergedArray[i]
```

Figure 6b: Pair exchange

```
split_mergeSort
    if (end - start) <= 1
        return
    midPoint = (end - start) / 2
    split_mergeSort(localArray, start, midPoint, workingArray)
    split_mergeSort(localArray, midPoint,   end, workingArray)
    merge( &(localArray[start]), midPoint-start, &(localArray[midPoint]),
      end-midPoint, &(workingArray[start]))
    copy workingArray into localArray
```

Figure 6c: Split merge sort function

```
merge
    for i from 0 to lenA
        while( (arrayB[j] < arrayA[i]) and (j < lenB))
            mergedArray[k++] = arrayB[j++]
        mergedArray[k++] = arrayA[i]
    while(j < lenB)
        mergedArray[k++] = arrayA[j++]
```

Figure 6d: Merge function

Last we used a parallel technique to calculate the total energy cost. The outer loop is used to distribute the calculation across all the processes. Then each process will calculate a minimum total energy consumption. The results are gathered and the smallest amount of energy consumed is considered the minimum. See Figure 7.

```
If Root Process Then
    If  numberOfDistancePairsBetweenSensors > numberOfProcessors
        numDistPerProcess = integer(numberOfDistancePairsBetweenSensors /
           numberOfProcessors)
        numExtraDistForLastProcess =
           remainder(numberOfDistancePairsBetweenSensors / numberOfProcessors)

        For each process
            if theLastProcess
               numDistPerProcess += numExtraDistForLastProcess
            firstDist = numDistPerProcess * processNumber
            arrayNumDist[process][0] = firstDist
            lastDist  = firstDist + numDistPerProcess
            arrayNumDist[process][1] = numDistPerProcess * processNumber

    else
        // there are more processes than distance pairs
        // each process gets one pair
        For each process
            if processNumber < numberOfDistancePairsBetweenSensors
               arrayNumDist[process][0] = processNumber
               arrayNumDist[process][1] = processNumber + 1
            else
               // the extra processes get assigned 0 and max values
               arrayNumDist[process][0] = 0
               arrayNumDist[process][1] = 0
               arrayOfdMinTotalEC[process]  = MAXDOUBLE
               arrayOfdOptDist[process]     = MAXDOUBLE
               arrayOfiOptNumCHs[process]   = MAXINT

MPI_Bcast(arrayNumDist)

For arrayNumDist[process] != 0
        calculate: dMinTotalEC, dOptDist, iOptNumCHs

MPI_Gather (arrayOfdMinTotalEC)
MPI_Gather (arrayOfdOptDist)
MPI_Gather (arrayOfiOptNumCHs)

If Root Process Then
        find minimum value and index in arrayOfdMinTotalEC
        dMinTotalEC = minTotEC
        minimumIndex = index
        dOptDist = arrayOfdOptDist[minimumIndex]
        iOptNumCHs = arrayOfiOptNumCHs[minimumIndex]

        group sensors into clusters based on current cut off distance
```

Figure 7: Total energy cost

Since we are using a known algorithm with known results, we expect applying parallel processing techniques will yield the same energy consumption results but run faster.

CHAPTER IV

RESULTS

We have conducted an extensive set of experiments using various simulation datasets on the MTSU Beowulf cluster, which is comprised of 16 Dell PowerEdge R210 with 8 cores and 32GB of RAM for a total 128 cores and 512 GB of RAM.

The performance comparison of execution times, in seconds, for serial and parallel processes with ten different problem cases are shown in Figure 8. As expected, with a small dataset there is no advantage to using parallel processing. However, as the datasets grew larger, the execution time of parallel processing is noticeably improved.

Table 1: Number of sensors per dataset

Dataset	**Number of Sensors**
1	10
2	100
3	200
4	300
5	400
6	500
7	600
8	700
9	800
10	900

The datasets are sets of sensors ranging from 10, in the first case, to 900 sensors, in the tenth case, see Table 1. The curves represent serial execution, parallel with 4 nodes, and parallel with 8 nodes.

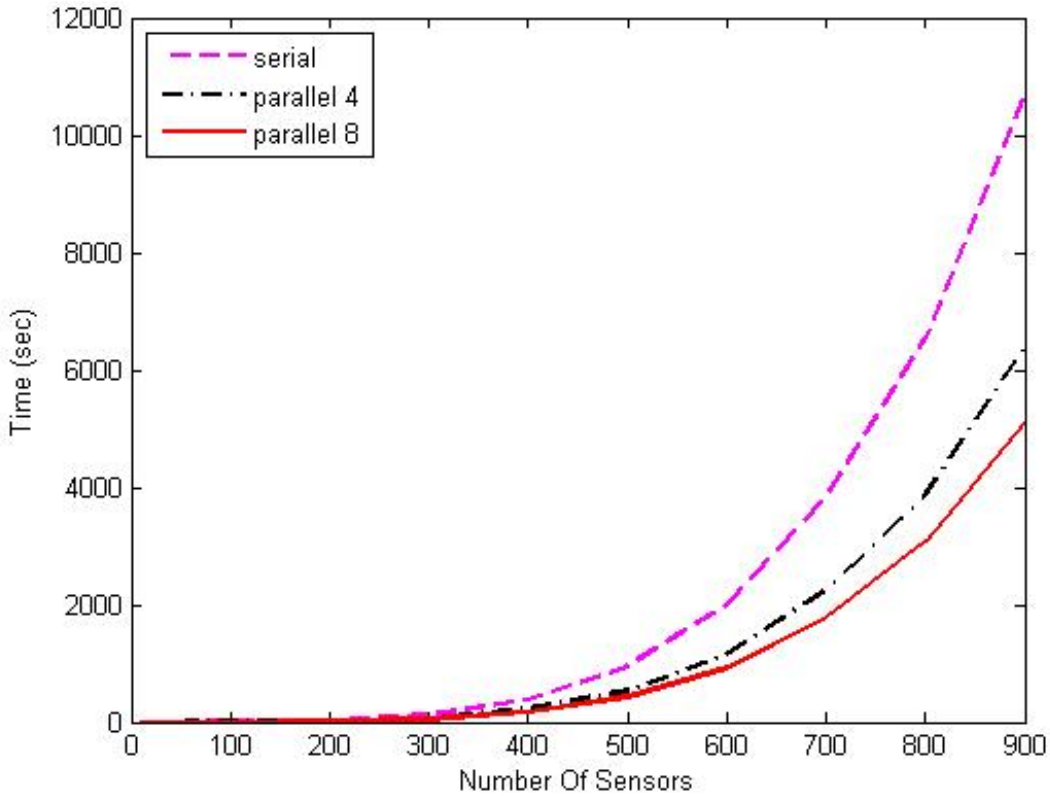

Figure 8: Serial versus parallel execution time

In Figure 9, we further plot the performance comparison of total energy consumption, execution time and problem cases in a 3D figure to illustrate the time superiority of parallel implementation over the serial version while both of them are producing the same optimization results.

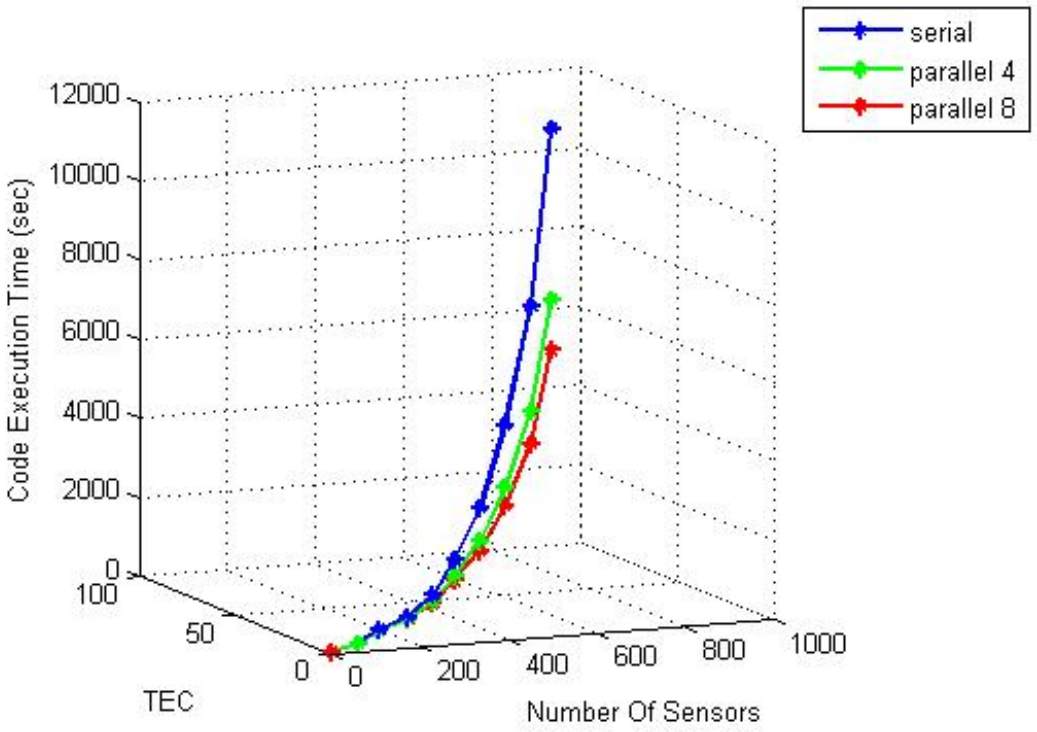

Figure 9: Total energy consumption, code execution time, number of sensors

In Figure 10, the comparison of the total energy cost versus the case number is shown. The serial and parallel graphs overlap each other on the total energy consumption as they produce exactly the same results, which further verifies the correctness and accuracy of the proposed parallel approach.

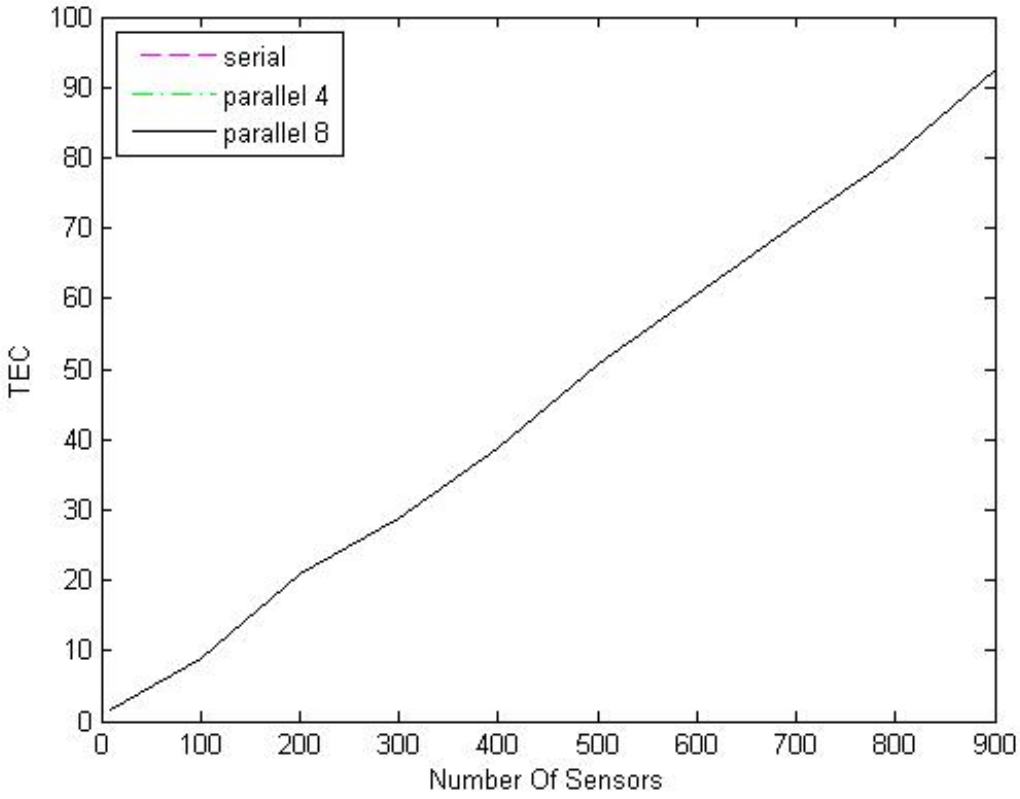

Figure 10: Total energy consumption of networks proposed by serial and parallel programs

CHAPTER V

DISCUSSION AND FUTURE WORK

A wireless sensor network is extremely useful for monitoring a remote or harsh environment. Sensors can be randomly scattered over an area or deliberately placed. Once set up, sensors will collect environment data and forward it to the base station, reporting on the current status of its environment.

Optimizing total energy consumption in WSNs greatly increases its lifespan. If all the nodes in the network are the same, i.e. a homogenous network, it can be optimized by sharing the load of data gathering and data transmission. This will prolong the lifespan of a homogeneous network.

The first step in adapting parallel processing to optimize a wireless sensor network, is to incorporate parallel programming techniques to an existing serial approach. As expected, the same results are produced in significantly less time using parallel techniques in MPI.

The future work in this area is to use this technique to further develop optimization algorithms. Since there is some shared memory in multi-core systems, it would be good to compare the speed of this MPI version with a version using pthreads or OpenMP. Another interest could be to explore an embedded parallel optimization solution.

BIBLIOGRAPHY

[1] Z. Kleinman, "Bee behaviour mapped by tiny trackers," *http://www.bbc.com/news/technology-32033766*.

[2] Libelium, "Wireless sensor networks to control radiation levels," *http://www.libelium.com/wireless_sensor_networks_to_control_radiation_levels_geiger_counters/*.

[3] H. S. N. Lab, "Volcano monitoring," *http://www.cs.harvard.edu/~mdw/talks/fiji-buffalo-oct07.pdf*

[4] A. A. Abbasi and M. Younis, "A survey on clustering algorithms for wireless sensor networks," *Comput. Commun.*, vol. 30, pp. 2826–2841, 2007.

[5] S. Soro and W. B. Heinzelman, "Prolonging the lifetime of wireless sensor networks via prolonging the lifetime of wireless sensor networks via," *19th IEEE Int. Parallel Distrib. Process. Symp.*, 2005.

[6] W. B. Heinzelman, a. P. Chandrakasan, and H. Balakrishnan, "An application-specific protocol architecture for wireless microsensor networks," *IEEE Trans. Wirel. Commun.*, vol. 1, no. 4, pp. 660–670, 2002.

[7] M. Kowshalya, "Clustering algorithms for heterogeneous wireless sensor networks - a brief survey," *Int. J. Ad hoc, Sens. Ubiquitous Comput.*, vol. 2, no. 3, pp. 57–69, 2011.

[8] V.-O. Sai, C.-S. Shieh, T.-T. Nguyen, Y.-C. Lin, M.-F. Horng, and Q.-D. Le, "Parallel firefly algorithm for localization algorithm in wireless sensor network," *2015 Third Int. Conf. Robot. Vis. Signal Process.*, pp. 300–305, 2015.

[9] A. I. Khan, M. Isreb, and R. S. Spindler, "A parallel distributed application of the wireless sensor network," *2004. Proceedings. Seventh Int. Conf. High Perform. Comput. Grid Asia Pacific Reg.*, pp. 81–88, 2004.

[10] I. Bahceci, G. Al-Regib, and Y. Altunbasak, "Parallel distributed detection for wireless sensor networks: performance analysis and design," *GLOBECOM 05 IEEE Glob. Telecommun. Conf.* 2005, vol. 4, p. 5 pp.–2424, 2005.

[11] J. Chempavathy and V. Vijayaraja, "Optimizing parallel concentric circle itinerary based KNN query processing in wireless sensor networks," *Proc. 2nd Int. Conf. Trendz Inf. Sci. Comput. TISC-2010*, pp. 226–229, 2010.

[12] M. Lounis, A. Laga, and B. Pottier, “GPU-based parallel computing of energy consumption in wireless sensor networks,” *Eur. Conf. Networks Commun.*, pp. 295–300, 2015.

[13] Y. Gu and Q. Wu, “Optimization of cluster heads for energy efficiency in large-scale wireless sensor networks,” *Lect. Notes Inst. Comput. Sci. Soc. Telecommun. Eng.*, vol. 28 LNICST, pp. 33–48, 2010.

[14] Y. Gu, Q. Wu, X. Cai, and J. Bond, “On efficient deployment of high-end sensors in large-scale heterogeneous WSNs,” *2009 IEEE 6th Int. Conf. Mob. Adhoc Sens. Syst. MASS ’09*, pp. 912–917, 2009.