BCoT-Based Smart Manufacturing: An Enhanced Precise Measurement Management System

Mst. Surma Khatun[1], Aofan Liu[2(✉)], and Mahdi H. Miraz[2]

[1] ACCA global, Moscow, Russia
[2] Xiamen University Malaysia, Sepang, Selangor, Malaysia
SWE2009510@xmu.edu.my

Abstract. Blockchain with its transparent, decentralised and secured characteristics have surfaced it as a futuristic technology for a surplus of advanced industrial applications. Blockchain of Things (BCoT) is the fusion of blockchain with the internet of things (IoT) technologies. The developments in multi-virtual Sensor IoT, homogeneous and heterogeneous multi-system information fusion for BCoT, industrial applications of BCoT has transformed the way digital world work. Ever since the smart devices were introduced, the world has evolved and progressed by making the entire world more dynamic by bringing technology, machines and people together. Smart manufacturing is another emergent sector where BCoT can significantly contribute. Smart manufacturing is basically formed by integrating multifaceted technologies, such as IoT-enabled technologies, service-oriented and cloud manufacturing, blockchain, etc., in various industrial applications. In fact, the IoT, particularly the industrial internet of things (IIoT) are enhancing sensor usage and remarkably contributing to smart manufacturing. This article contributes to the existing knowledge domain by exploring and studying various sensors and proximity sensors in details, customer and product movement, reaction of customers towards sensors and smart technologies based on BCoT, usages and advantages of sorting techniques with regards to the memories of BCoT devices and sensors, a precise measurement system using quick sort for smart manufacturing and future challenges and possibilities.

Keywords: BCoT · Measurement management system · Proximity sensor · Smart manufacturing and sensors

1 Introduction

Blockchain [1, 2] integrated with the internet of things (IoT) [3], i.e. the blockchain of things (BCoT) [4], is revolutionising the Industry 4.0 [5] as well as the emerging Industry 5.0. BCoT has been being employed to multifaceted manufacturing and/or industrial applications, since it provides the necessary supports to create a smart manufacturing system which is highly automated and decentralised with high level of productivity and efficiency.

M. H. Miraz et al. (Eds.): iCETiC 2022, LNICST 463, pp. 29–53, 2023.
https://doi.org/10.1007/978-3-031-25161-0_3

Sensors, a fundamental component of the Internet of Things (IoT), are used in our homes and workplaces being embedded in various smart devices. They are used in different formal and informal infrastructures where digital technologies are used for security purposes or to detect or collect data from any harbingers, for example in industries, shopping centres, hospitals, etc.

Sensors can 'sense' and respond to fluctuations in various aspects of the surrounding environment, such as pressure, light, motion, temperature, etc. These connected devices are capable to output information on the basis of what they detect by sharing data with other connected devices as well as management systems. Since IoT sensors are now widely used there is plenty of room for improvements, which will result in enhanced operational efficiency, reduced cost, enhanced safety measures, across the board automation, etc.

Proximity sensors are mostly applicable in industrial sectors. In this paper, we focused on the improvement of technology which would be more relevant in large clothing stores, hypermarkets, furniture stores or in other centres where there will be a juggernaut of connectivity of prices, commodities and customers. That being said, the proposed measurement management system can be used in many other aspects too.

The smart manufacturing sector is drastically expanding, embracing the emerging technologies. However, utilisation of third-party-based authorisation and centralised industrial networks lead to lower efficiency, lesser flexibility, lower scalability and weaker security. To address these issues, BCoT can play a vital role. In this context, this article introduces an enhanced precise measurement management system utilising BCoT-based smart manufacturing. The proposed measurement system utilises quick sort to easily detect a value within a very short time. It can furnish intuitive solutions for addressing the issues and challenges with regards to the limitations of BCoT-based systems.

2 Smart Manufacturing and Sensors

Sensors have been used in the industries and organisations for a long time. In fact, thermostat was first introduced in the late 1880s while the infrared sensors in the late 1940s. The ability to detect was a need from long ago. Sensors are smaller, slimmer and stretchable, stronger and smarter, more solid, more capable and cheaper.

The invention of the internet of things (IoT) contributed in revolution of multifaceted sensors. Its functionality and delivery of different types of intelligence as well as data by using different types of sensors in the whole network of connected devices is making sensors more efficient and smarter. It offers autonomous functionalities by integrating devices, sensors and communication network together to exchange data and information with each other and boost the efficiency of the whole system. These resulted sensors to be an essential part in smart manufacturing.

Therefore, a novel paradigm of measurement system has been proposed in this article, utilising quick sort, which can be viewed as a potential enabler of wide-ranging use cases scenarios as well as applications.

The following sub-sections gives an account of various sensors commonly used in smart manufacturing [6]:

Temperature Sensor - measures the quantity of the heat energy which lets a physical transformation in temperature to be detected from a specific source and translates the captured data to a user or device. Temperature sensors are widely utilised in freezers, A/C control and similar environmental control systems, agriculture, manufacturing processes, health industry, etc., to maintain the manufacturing process always optimal and to maximise output or production. Sub-categories of temperature sensors include: IC (Semiconductor), resistor temperature detectors (RTD), thermocouples, thermistors and infrared sensors, etc.

Proximity sensors - recognises the presence or absence of any product within range, or the qualities of that product and transform it into signals which can be understood by customers or electronic devices without coming into touch. Proximity sensors are utilised in cars, in the retail business and parking lots such as malls, stadiums airports, etc. Sub-categories of proximity sensors include: inductive, photoelectric, ultrasonic and capacitive sensors are examples of proximity sensors.

Pressure sensors - detect pressure fluctuations and decreases in pressure, which are then translated to an electronic signal. The amount here is determined by the amount of pressure used. Pressure sensors are used in the manufacturing industry, in the upkeeping of indoor water as well as heating systems, etc.

Water quality sensors - detect the water quality. These sensors are utilised in manufacturing, for maintenance of indoor water as well as heating systems, for outdoor water monitoring, etc. Examples of water quality sensors include: total organic carbon sensor, conductivity sensor, chlorine residual sensor, turbidity sensor, pH sensor, oxygen-reduction potential sensor, etc.

Chemical sensors - indicate changes in liquid, find out air chemical changes. Chemical sensors are employed in industrial environmental process control as well as monitoring, intentionally or unintentionally released toxic chemical detection, radioactive and explosive detection, recycling operations at the space stations, pharmaceutical companies and laboratories, etc. Examples of chemical sensors include: Chemical field-effect transistor, Hydrogen sulphide sensor, non-dispersive infrared sensor, electrochemical gas sensor, Zinc oxide nanorod sensor, chemi-resistor, fluorescent chloride sensor, potentiometric sensor, pH glass electrode, etc.

Gas sensors - monitor any changes in quality of the surrounding air and sense the presence of different gases. Gas sensors are utilised for monitoring air quality, oil and gas sectors, toxic or flammable gas detection, monitoring of hazardous gas in coal mines, research in chemical laboratories, manufacturing – pharmaceutical, rubber, plastics, paints and petrochemical, etc. Examples of gas sensors include: carbon dioxide sensor, hydrogen sensor, breathalyzer, nitrogen oxide sensor, ozone monitor, air pollution sensor, electro-chemical gas sensor, carbon monoxide detector, catalytic bead sensor, oxygen sensor, gas detector, hygrometer, etc.

Smoke sensors - detect smoke, i.e. airborne particulates and gases, and its level. Smoke sensors are utilised in HVAC, manufacturing industry, buildings and accommodation safety measures - detect and alarm fire and/or gas incidences. Examples of smoke sensors include: ionisation smoke sensor, optical smoke sensor (photoelectric).

Infrared sensors - detect certain features or characteristics of the surrounding environment by either detecting or emitting infrared radiation. Infrared radiation can be measured by the heat being emitted by the objects. Infrared sensors are used in healthcare, multifaceted IoT projects, smart watches, remote control, smartphones, breath analysis, home appliances, optical communication, measurement of temperature without any physical contact, wearable electronics, infrared vision, etc. Examples of infrared vision include visualising heat leaks in electronics, monitoring blood flow, seeing under layers of paint by art historians, etc.

Level sensors - work out the amount or level of liquids, fluids, or any other substances which flow in a closed or open system. Level sensors are utilised in businesses involving liquid materials, e.g. the recycling industry, the alcohol and juice industry, etc. Level sensors can be used for determination of liquid levels and fuel gauging in closed or open containers, water reservoirs, monitoring of sea level as well as tsunami warning, machine tools, compressors, hydraulic reservoirs, medical equipment, pharmaceutical and beverage processing, etc. Examples of such sensors include: continuous level sensor, point level sensors, etc.

Image sensors - are employed to transform optical images into electronic signals to electronically store or display them. Image sensors are used in digital camera and modules, sonar, thermal imaging devices, radar, medical imaging as well as equipment designed for night vision, biometric devices, media house, etc. In integrated circuits such as charge-coupled device (CCD), in the car industry, CMOS (complementary metal-oxide semiconductor) imagers, in the retail industry, in improved security systems, in IoT industry, these sensors are deployed for collecting data about customers - aiding businesses in getting better insights of the store visitors by identifying the race, gender, age, etc.

Motion detectors - identifies any motion or physical movement in the surrounding area and it converts the motion into electric signals. Motion detectors are highly used in the security industry, particularly for intrusion detection systems and smart cameras with motion-based capture/video recording., Other usages include but not limited to hand dryers, automated sinks/toilet flushers, boom barriers, automatic door control, toll plazas, automatic parking systems, energy management systems (e. g. automated lighting, fan, AC, etc.), and so forth. Examples of motion detectors include: ultrasonic, passive infrared (PIR), microwave, etc.

Accelerometers - gauge the physical acceleration, i.e. the rate of change of velocity per unit time, of an object caused by inertial forces. Accelerometers are used in cellular and media devices, free fall detection, automotive control aa well as detection, tilting, aircraft and aviation industries, vibration measurement, consumer electronics, sports academy (e.g. athletes' behaviour monitoring), industrial and/or construction sites, monitoring driving fleet and so forth. Examples of accelerometers include: hall-effect accelerometers, piezoelectric accelerometers, capacitive accelerometers, etc.

Gyro sensors - measure the angular velocity or rate i.e. speed of rotation around an axis. Gyro sensors monitors the orientation of an object. Gyro sensors are used for the automation of some production processes in automotive navigation systems, cellular and camera devices, robotics control, consumer electronics, game controllers, drone and radio-controlled (RC) helicopter or unmanned aerial vehicle (UAV), vehicle control or

advanced driver assistance systems (ADAS), etc. Examples of gyro sensors include: rotary or classical gyroscopes, optical gyroscopes, micro-electro-mechanical systems (MEMS) gyroscopes, vibrating structure gyroscope, etc.

Humidity sensors - similar to the temperature sensors, detect the change in humidity, almost instantaneously. Humidity sensors are used for controlling ventilation, heating and AC systems in the industrial as well as residential domains, such as automobiles, museums, greenhouses, industrial spaces, paint and coatings industries, meteorological stations, hospitals and pharmaceutical companies to safeguard pharmaceuticals, etc.

Optical sensors - measure the physical quantity of light rays and convert it into electrical signal which can be easily read by user or an electronic instrument/device. Optical sensors are used in healthcare, environmental monitoring, energy, pharmaceuticals mining, aerospace, optical fibre communications, digital optical switches, ambient light detection, assembly line part counters, high speed network systems, civil and transportation fields, oil and gas applications, elevator door control and safety systems, etc. Examples of optical sensors include: photo detector, proximity, fibre optics, pyrometer & infrared.

3 Proximity Sensors

The proximity sensor can sense the proximity of an object (e.g. in an automated production line) without touching it. The existence as well as movement information of the objects are captured and then transformed into electrical signals.

Proximity sensors are used in parking lots to indicate parking spaces, consumer electronics, assembly lines (particularly in chemical industry), food industry, etc. On consumer electronic devices, proximity sensors act as capacitive touch switches; for instance, it can be used for detecting whether a phone user is holding it near his/her face. Proximity sensors can also be employed as diffuse sensors such as in washrooms, or even as collision detection sensors for robots.

Proximity sensors are highly utilised in multifaceted industrial and manufacturing applications, such as for detecting parts in conveyor systems, for safety measures, inventory management for object positioning, detection, inspection and counting, etc.

Proximity sensors typically generate electromagnetic fields or emit radiation beams, e.g. infrared rays. In retail industry, such as in a supermarket, proximity sensors can be employed for detecting the movement between any customer and the product the customer is interested in. Notifications of ongoing promotions and offers can then be sent to that particular customer for the products near the respective sensors. On the contrary, unlike the traditional proximity sensors, capacitive proximity sensors are not limited to only metallic targets. They can, in fact, be utilised to detect 'anything' carrying electrical charges.

The following are some features of proximity with regards to increasing lucidity:

- Contactless (ensuring object stays well-conditioned, detecting both versatile metallic as well as non-metallic objects, including powders, liquid and granular)
- Natural by the surface colours of the objects (usually distinguishes any physical changes)

- Usable in humid conditions and wide temperature range usage, contrasting traditional optical detection.
- Cheaper price but longer service life in comparison with other sensors.
- Higher response rates.
- Emit a light beam using high-end photoelectric technology having the ability to detect any sorts of objects (photoelectric).

The selection of the appropriate sensor depends on various factors. For example, for measuring range/distance, the distance (long or short) between the object placed and the sensor, amount of light, possible obstacles, etc. are to be considered.

Proximity sensors can be further categorised in capacitive sensors, inductive sensors, ultrasonic sensors, photoelectric sensors, etc. Inductive sensors are used in short distance applications and suitable for only metals. They are a good choice for harsh environmental conditions; commonly used in industry machineries and automations. Capacitive sensors are also used in short distance applications, primarily for non-metallic as well as metallic objects including powders granular, and liquid. They are used in industry, automations, liquid and moisture, machineries, touch sensing, etc. Capacitive sensors are highly suitable for using in harsh environmental conditions. Photoelectric sensors, on the contrary are for long distance applications, for object with simple surfaces. They are used in distance measurement anemometers for detecting wind speed as well as direction, automation production processes, fluid detection, unmanned aerial vehicles (UAVs) for object monitoring robotics. They are a good fit for harsh environmental conditions except vacuum. Ultrasonic sensors are also for long distance applications, particularly for object with simple/complicated surfaces. They are used in security systems such as surveillance and burglar alarms, item counter, monitoring as well control applications. However, unlike the aforementioned ones, ultrasonic sensors are not a good fit for harsh environmental conditions [7].

Amongst various types of proximity sensors, we opted for capacitive sensors for implementing the proposed measurement application.

Capacitive proximity sensor generates electrostatic fields when any object (conductive/non-conductive - including glass, wood, metals, plastic, water, etc.) reaches the target area, the capacitances of both the plates increase, causing oscillator amplitude gain which generates sensor output switch. Capacitive proximity sensors are highly suited for industrial applications, e.g. production automation machines which count products, pipelines, filling processes, inks, product transfers, etc., composition and pressure, fluid level, non-invasive content detection, moisture control, touch applications and so forth.

4 Customers' Reaction Towards Sensors and Smart Technologies

An IoT device is a device embedded with at least one sensor that connects to and shares data from the sensor(s) with other devices directly or indirectly via a wireless communication protocol, such as through an IoT hub or a smart device including smartphone.

Sensor can be a component of an IoT device, or a stand-alone object (a radio-frequency identification (RFID) which can be read by an RFID reader) which detects

changes. Let us consider the case of a smartphone - it uses a number of sensors, such as a timer, accelerometer, global positioning system (GPS) sensor, etc.

Consumer/customer can be indicated as an entity who purchases a product or interacts with a product for the purpose of purchasing it or does both or equivalents. Normally, a consumer/customer makes purchase decisions of a product on the basis of the information available online or at a brick-and-mortar store.

Consumer device or customer device is owned by a consumer/customer which is capable of communicating with IoT devices enhanced with sensors or without sensors.

Retailers provide products for consumers/customers for purchasing regardless of their physical or web presence. A retailer can play a role as a manufacturer of a product or any other person within the distribution chain of the product.

Marketer provides information (coupon or alert) on products to customers. The marketer or the retailer can collect data on consumers and use that data for marketing purposes.

When customers visit brick-and-mortar stores to purchase anything they may seek assistance from anyone with regards to making purchasing decisions of the products. Customers examine the product by picking up the product, reading its label or viewing a review of the product on a website before making a decision to buy (online or offline). IoT devices are located in the supermarkets as well as in the customer devices (such as smartphone, smart glasses, smart watch, etc.) which collects data from customer devices for the retailer or the marketer. On the other hand, the customer's smartphone can determine the location of the person near a product based on a proximity sensor on the product to determine the possibility of interacting with the product.

The retailer or marketer analyses the sensor data immediately after receiving it in order to determine the interactions between the consumer and an IoT device. To map some interactions, specific sensor data may be required to determine that the consumer has performed that specific communication. The received sensor data is checked to see if it matches the specific sensor data mapped to a specific interaction. If it matches, then we know that the interaction has been identified.

In customer navigation tech (e.g. bluetooth low energy (BLE)) or customer devices, machine-learned models use historical information regarding interaction. Each customer device specifies specific data. For any specific customer, input, output and lead scores are counted.

A standard conversion rate can be calculated based on any type of interaction between a product and a customer, such as when a customer purchased a specific product after examining the details, as opposed to other customers who came across the details.

Finding customers' choices, finding best prices for the product by customer, costing determination by retailer for example garments manager or financial advisor, finding average cost or average values of complicated rates or functions, calculating the most appropriate and relevant scores or costs or values after updating the data or getting additional new data with any specific interval of time, finding the probability of the best choice i.e. proximity of the consumer to the product on the basis of previous calculations for predetermination of cost or choice can be performed by any invention represented as machine-useable instructions or computer codes, comprising computer-executable instructions, e.g. program components run by any computer or other form of computing machines, such as bluetooth low energy (BLE), personal data assistants (PDA) or any other handheld devices. Program components, including programs, routines, components, data structures, objects or code accomplish particular tasks or administer particular data types. Multifaceted factors, such as performance expectancy, social influence, effect expectancy, facilitating conditions, price value, habit, motivation, etc., influence the customers to install a retailer's application (app) on their IoT devices enhanced with sensor, to utilise the respective mobile app in a retail store, allowing their location to be utilised for personalised/customised/targeted service rendering and increase demand and facility towards customer.

We can implement the proposed algorithm (i.e. using quicksort in approximate or precise memory of BCoT devices or sensors) in existing different aspects of the inventions which may be applied in a wide range of system configurations, including customer devices, handheld devices, BCoT- based electronics, computers, computing devices, etc.

For example, Qualcomm introduced proximity beacons while Apple released iBeacon (relays three values: minor ID, major ID and unique ID). Both of these products are compatible with the BLE technology stack as well as can be utilised for tracking indoor location.

The program can be used in distributed computing environments. The advantage of this will be performing any work remotely linked through any communications network. Customers will be benefited by using the system as well as retailers or marketer will get smoother sensors or devices.

One of the most important factors in improving the performance of some real-time BCoT applications is transaction time for data collection and sorting. Transaction time is, however, constrained by the computational and communication capabilities of processing computers. To overcome this limitation, we propose an efficient method for sorting massive amounts of data that employs a progressive quality improvement approach.

The use of industrial internet of things (IIoT) networks within the industrial settings, such as smart manufacturing factories, SCADA and ICS, can lead to a disaster or even financial loss if they fail to initiate or perform their function at the proper time. Which shows, we need to use such a program which takes less time within the performance as well as updating blocks. Smoother function or coding is always helpful for any digital devices.

5 Sorting, Usage and Advantages of Utilising Quicksort in the Memories of BCoT Devices and Sensors

Sorting is a widely explored topic in the domain of algorithms. In point of fact, optimised deployment and executions of some algorithms, including quicksort, has been widely adopted. Examples include optimised implementations in Microsoft Visual C++ 6.0 as well as Intel C++ compilers. In Intel compiler, the implementation of Quicksort has been optimised using Hyper-Threaded technology. In many literatures, various fast algorithms have been designed and implemented for processing transactions as well as disk-to-disk sorting. But if we focus on the sorting algorithms' performance on the conventional CPUs, we will see it is administrated by cache slips and instruction enslavements.

Sorting is a relatively common computer operation that converts a set of "unordered" records into an "ordered" sequence of records. The record here can be any type of elements in computer.

Quicksort is one of the most efficient, fastest (as it has the upper hand in the average cases for most inputs), most used, in-place and comparison-based sorting algorithms which is better suited for large data sets. But this sorting algorithm does not demand extra space. Thus, it performs better on Arrays compared to any other sorting.

In this article we have presented an external sorting algorithm grounded on the quicksort approach. The file that needs to be sorted is stored on a disk; only the blocks which are currently needed are fetched into the main memory.

The time complexity of quicksort is O (n log n) in the best-case scenario, O (n log n) in the average-case scenario and O (n^2) in the worst-case scenario. It's running time is actually O (nBlog(nB)), where B is the block size.

The sorting algorithm is used for information searching. Due to the advantages Quicksort offers, it is widely used for this purpose, particularly when a stable sort is not required.

Quicksort works by splitting a large array of data into smaller sub-arrays. This implies that each iteration works by splitting the input into two components, sorting them, and then recombining them. The whole process can be summarised in three steps: 1) pick, 2) divide and 3) repeat and combine.

When the data to be sorted comprises of many duplicate values, the quicksort can be improved by grouping together all the values which are equal to the pivot to the middle. The quicksort algorithm then needs to be run recursively to sort the values on the left as well as the values on the right.

The principle of divide-and-conquer is the main base on this sorting algorithm. For the advantages it offers, it is used widely for many purposes. Major advantages of it includes: it uses only a small auxiliary stack thus consumes relatively fewer resources during execution, requires only n (log n) time sorting n items, possess a very short inner loop, etc. Quicksort has undergone through comprehensive mathematical analysis – very precise statement regarding performance issues can be made.

To broadly widen scope of the traditional approximate computing, the approximate memory can be leveraged for improving sorting algorithms' performance, but still producing precise results.

There are three classic and popular sorting algorithms that use approximate main memory: mergesort, quicksort as well as radixsort. In fact, the first two algorithms are comparison-based, however, the last one is not. Without the need for precise outputs, simulation results show that quicksort and radixsort can produce a nearly sorted sequence while dropping write latencies on approximate memory by 30% to 40% [8].

Apart from focusing on approximate computing using approximate hardware, a fast sorting algorithm, on the memory system with both approximate as well as precise memory for ensuring precise results, has rather been proposed. A novel algorithmic level execution mechanism on hybrid approximate/precise memory has been implemented, to generate precise results. Specifically, we propose program mechanism in which the approximate memory acts as an accelerator. If the input data is copied to the approximate memory from the precise memory, and then an existing sorting algorithm is performed on the approximate memory, the approximate results are possible to be precise in the precise memory. If the sorting algorithm is able to produce a nearly sorted result on approximate memory, only a lightweight IoT device is required afterwards. As a result, the cost of devices and data copies between approximate memory and precise memory can be compensated through the gain of uploading the sorting algorithm to the approximate hardware. To enhance the performance of precise computing, approximate hardware can additionally be utilised, which widens the application scope of approximate hardware.

Once upon a time, approximate hardware was only used for approximate computing. As a result, we develop and test commonly used sorting algorithms on hybrid storage systems with approximate and precise storage, as well as show system and architectural insights for enabling precise computation on approximate hardware.

Although the system interfaces are to be sensibly redesigned for supporting hybrid approximate/precise main memory, the required hardware modification remains lightweight as well as easily to implementable.

This algorithm is generated for database and data mining applications. We use the texture mapping and mixing capabilities of GPUs which can be implemented as an efficient Bitonic sorting network.

Meanwhile, in order to improve overall sorting performance, we describe an efficient instruction dispatch mechanism and an efficient memory data access pattern in our novel algorithm. Our sorting algorithm has been used to speed up stream mining algorithms as well as join-based queries.

The results show a significant improvement over previous CPU as well as GPU-based sorting algorithms.

6 Using Quicksort in Approximate or Precise Memory of BCoT Devices or Sensors

6.1 Quicksort: Theory and Experiments

6.1.1 Pseudocode

Pseudocode is an effective way to abstracts away the syntax to let us focus on solving the problem in front of us instead of getting bogged down in the exact syntax language. Moreover, it allows us to work on pure programming logic which provides us a chance to simply write in plain English. Therefore, we start with pseudo code.

QuickSort 1 Sort the orders using Quicksort according to the parcel ids.

Input: User will input the number of parcel ids, n, to generate and sort. Users can input the value of variant, *choice*, to choose option 1 - 4 if they wish to see the details of the parcel id and the delivery cost. 1 for just sorting, 2 for printing the unsorted parcels then sorting, 3 for sorting then printing the sorted parcels, 4 for sorting and printing both sorted and unsorted parcels

Output: The algorithm should print out the time taken to perform the sorting. Moreover, it shall print out the sorted or unsorted parcel ids and the corresponding delivery cost that was generated on demand.

```
1.  Declare hash map uset
2.  Declare struct{id, cost} parcel
3.
4.  function PRINT(p[], n)
5.      for i = 0 → n − 1 do
6.          print out serial number i + 1, parcel id and the corresponding cost
7.      end for
8.  end function
9.
10. function PARTITION(p[], left, right)
11.     mid ← (left + right)/2
12.     swap((median of left, mid and right), left)
13.     pivot ← p[left].id
14.     i ← left
15.     j ← right
16.     while i < j  do
17.         while p[j].id greater or equal to pivot and i less than j do
18.             j−
19.         end while
20.         while p[i].id less than or equal to pivot and i less than j do
```

```
            i++
        end while
        if i < j then
            swap(p[i],p[j])
        end if
    end while
    swap(p[left],p[i])
    return i
end function

function QuickSort(p[], left, right)
    if left < right then
        pivotIndex ← partition(p, left, right)
        QuickSort(p, left, pivotIndex − 1)
        QuickSort(p, pivotIndex + 1, middle)
    end if
    return
end function

function CountSortTime(p[], int n)
    freq ← system clock cycle frequency
    startTime ← current clock cycle counted
    QuickSort(p, left, right)
    endTime ← current clock cycle counted
    time ← (startTime − endTime) * 1000000/freq
    return time
end function

function Main
    n ← user input number of parcels
    declare parcel p[n]

    initialize seed for rand()
    initialize seed rd for mersenne_twister_engine
    standardize mersenne_twister_engine with rd()
    encapsulate engine to function randomLarge()

    for i = 0 → n − 1 do
        value ← large random integer
        while 1 do
            it ← traverse uset using an iterator to find value
            if it is the end of uset then
                p[i].id ← value
                break
            else
                value ← new large random integer
                it ← traverse uset using an iterator to find value
            end if
        end while
        p[i].id ← value
        p[i].cost ← random double value
    end for

    display user interface to show options
```

```
    choice ← user choose one from options
    if choice == 1 then
        print out the return value of CountSortTime(p, n)
    end if
    if choice == 2 then
        Print(p,n)
        print out the return value of CountSortTime(p, n)
    end if
    if choice == 3 then
        print out the return value of CountSortTime(p, n)
        Print(p,n)
    end if
    if choice == 4 then
        Print(p,n)
        time ← CountSortTime(p, n)
        Print(p,n)
        print out the value of time
    end if
    empty p
    return
end function
```

6.1.2 Growth Rate

Normally, growth rate refers to how the scale of the algorithm's time complexity and space complexity increase as the scale grows. In addition to predicting the performance of the algorithm, analysing the growth rate helps to classify problems as well as algorithms by difficulty. This is very useful when we compare different algorithms accordingly.

Under normal circumstances, we can analyse the growth rate of the algorithm through two methods: empirical analysis and theoretical analysis [6]. Here we apply the empirical analysis to find the growth rate of the algorithm.

6.1.3 Experimental Data

To test the growth rate of the program using empirical analysis method, we run the program with the following n values: 1000, 3000, 5000, 8000, 100000, 15000, 25000, 35000, 51200, 66000, 86400, 100000, 125000, 150000, 180000, 200000, 250000, 300000, 4000000, 500000, 600000, 700000, 800000, 900000, 1000000.

In order to ensure the accuracy of the data, we adopt the method of taking the average of multiple measurements. For each of them, we will run for ten times and take the average value as the final value.

Tables 1, 2, 3 and 4 below list the measurement particulars for the series of aforementioned experiments:

Table 1. Experiment result from 1000–25000

	1000	3000	5000	8000	10000	15000	25000
Experiment 1	93.5	351.1	598.2	896.3	995.3	1505.0	2989.5
Experiment 2	85.5	349.2	537.6	777.5	1128.6	1496.6	2987.6
Experiment 3	88.2	288.3	541.1	798.1	995.7	1627.4	3550.2
Experiment 4	96.1	312.8	568.3	911.5	1207.3	1396.9	3306.7
Experiment 5	99.6	311.9	538.1	882.6	1134.9	1456.2	3107.6
Experiment 6	95.7	271.7	628.3	788.8	1085.2	1640.7	2860.8
Experiment 7	87.2	289.5	541.4	852.4	1198.3	1505.3	3300.4
Experiment 8	89.1	309.9	562.4	889.1	1067.5	1550.4	3421.5
Experiment 9	84.5	301.2	589.7	873.6	952.2	1589.0	3523.4
Experiment 10	86.2	303.6	601.2	885.4	987.4	998.6	3601.9
Average	90.6	308.9	570.6	855.5	1075.2	1476.6	3265.0

Table 2. Experiment result from 35000–150000

	35000	51200	66000	86400	100000	125000	150000
Experiment 1	4305.9	5592.3	7487.2	10028.9	13214.8	16932.8	19908.0
Experiment 2	4290.6	5166.1	6689.7	9730.2	12551.0	16301.4	18376.2
Experiment 3	4019.6	4984.9	6624.4	10784.8	12790.9	15925.9	20291.9
Experiment 4	4400.7	5380.9	6664.9	10069.0	12761.1	16476.8	19234.9
Experiment 5	4002.5	4810.7	6655.9	9645.5	13457.1	16049.6	20474.8
Experiment 6	4104.4	5243.0	6677.8	10882.4	12999.3	16861.1	19275.0
Experiment 7	4003.3	5067.7	6638.6	9966.0	12852.6	15140.6	20116.7
Experiment 8	4307.1	5101.7	6674.1	10662.9	12633.1	15678.9	19507.6
Experiment 9	4461.1	5376.7	6622.7	10962.3	12845.8	15154.2	18954.7
Experiment 10	4040.4	4976.7	6611.8	10808.6	12562.0	15003.0	19960.8
Average	4193.6	5170.1	6734.7	10354.1	12866.8	15952.4	19610.1

Table 3. Experiment result from 180000–600000

	180000	200000	250000	300000	4000000	500000	600000
Experiment 1	23917.2	26060.3	32352.6	38931.7	51491.6	66087.5	80563.8
Experiment 2	21946.7	27172.8	29758.0	37532.6	48837.0	66646.8	78239.4
Experiment 3	21186.3	25759.9	30055.8	36384.8	50522.0	65820.0	80201.9
Experiment 4	21580.1	26417.3	30379.3	36802.6	50023.3	63413.9	74872.5
Experiment 5	23456.6	26765.9	29505.6	36721.7	52330.6	65212.4	78974.1
Experiment 6	23561.8	27482.3	29859.6	39207.1	48657.6	72475.8	77629.2
Experiment 7	23311.9	26960.4	30555.2	35618.3	53729.0	68441.6	77005.6
Experiment 8	21331.6	26020.5	31193.8	36113.0	50215.1	63275.0	81014.4
Experiment 9	22532.6	27263.7	30975.6	38039.2	49329.2	72554.4	74982.7
Experiment 10	23383.9	25801.6	31620.9	36535.0	49896.3	64280.8	82908.5
Average	22620.9	26570.5	30625.6	37188.6	50503.2	66820.8	78639.2

Table 4. Experiment result from 700000–1000000

	700000	800000	900000	1000000
Experiment 1	92960.4	106881.0	118951.0	135267.0
Experiment 2	95680.8	92260.1	111529.1	127008.5
Experiment 3	94077.4	92694.6	112814.4	126985.8
Experiment 4	85720.6	105417.3	112653.3	129414.8
Experiment 5	92945.2	96946.4	110922.3	133788.7
Experiment 6	87461.9	100097.9	116653.5	138450.3
Experiment 7	93838.2	91980.6	117413.5	127723.6
Experiment 8	89178.0	95847.4	115637.9	133766.2
Experiment 9	90659.1	96326.0	113625.5	140870.8
Experiment 10	87643.1	105245.6	118134.6	126903.4
Average	91016.5	98369.7	114833.5	132017.9

6.1.4 Summary and Graph

Although the rate of growth varies with the configuration of the machine, the trend is generally the same. To better illustrate it, we plot a graph, refer to Fig. 1 below, to show the growth rate:

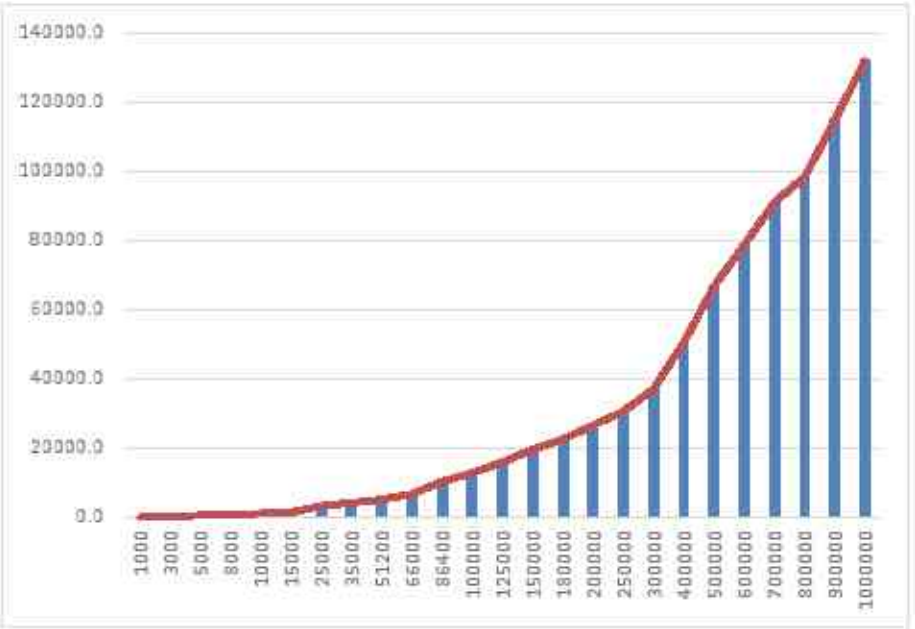

Fig. 1. Relationship between running time and array size.

We already know that the best case of the quicksort is $O(nlogn)$, while the worst case is $O(n^2)$. So, the range where we fit this function is probably from $O(nlogn)$ t $O(nlogn)$.

To find the trend of the average cost, we fit the above function with the following polynomial:

$$y = 362.26x^2 - 4533.2x + 12141$$

Obviously, $4533.2x$ will play an important role in our current data range, so we may first start with growth rate $O(nlogn)$. We try to draw the graphs of $10nlogn$, $nlogn$, $0.1nlogn$ respectively, but they are all larger than the current data. After several attempts, we found the most suitable fitting function which is above our average cost in all values, as shown in Fig. 2 below:

$$y = 0.04nlogn$$

Similarly, we also find another function which is below our average cost in all values, refer to Fig. 3 below.

$$y = 0.01nlogn$$

By definition, we know that the growth rate of this algorithm is $\theta(nlogn)$.

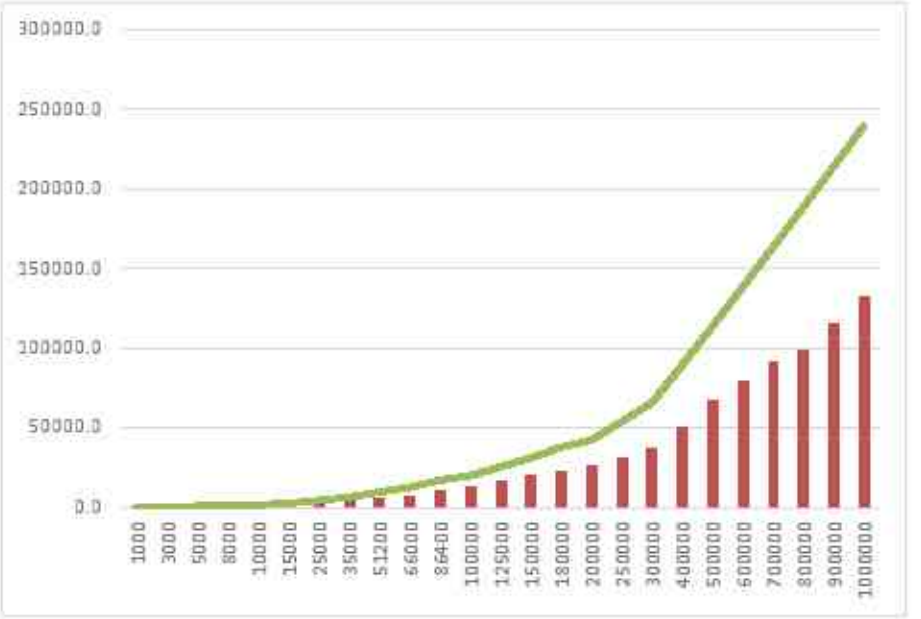

Fig. 2. Average cost and 0.04nlogn.

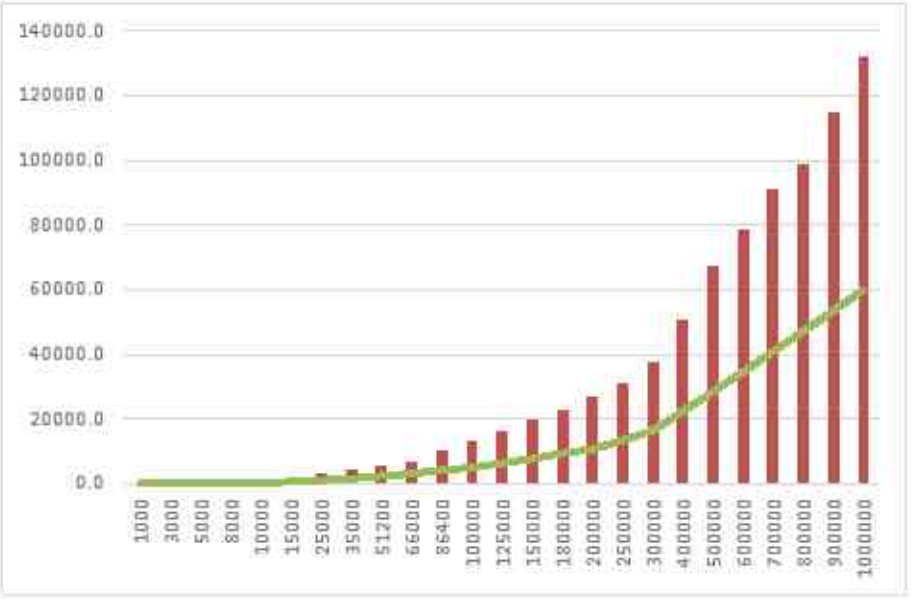

Fig. 3. Average cost and 0.01nlogn.

7 Algorithm Performance

As an intensively studied problem in the field of computer science, the sorting problem has attracted a large number of researchers to focus on it. Moreover, sorting algorithms have a significant impact on performance on complex computing tasks. Even 5% performance optimisation will bring a significant improvement. Therefore, it is worthwhile to take the effort to solve this problem. In the following subsections, we discuss how Quicksort has been optimised in this research.

7.1 Pivot Selection

Adopting the idea of divide and conquer, quicksort splits the big into small ones and splits the small into smaller ones. In simple terms, the principle of quicksort is to select a pivot, divide the original array into two parts by comparing each element in the original array with pivot, and repeat this process continuously. After sorting all the small arrays, the final result is already sorted.

Therefore, the choose of pivot is very important for the efficiency of quicksort algorithm. The pivot strategy we used in the research is the median of three, here we will illustrate why we have choosen this method by giving a detailed explanation to several commonly used pivot selection methods.

However, if we use different pivots, it may enhance the performance of the algorithm linearly but not exponentially. The upper bound is O(nlogn).

7.1.1 Choose First

This method is the simplest - just need to return the subscript of the first element of the sub-array, which is implemented below.

```
int partition(int* arr, int left, int right)
  int pivot = arr[left];
```

The only reason for choosing the first as pivot is because this method is relatively simple and easy to implement. However, it shall be noted that it is very likely to deteriorate to the worst case **[O(n] ^2)** while using this pivot.

7.1.2 Choose Last

This method is the same as the previous method, but we selected the last element.

```
int partition(int* arr, int left, int right)
  int pivot = arr[right];
```

7.1.3 Randomised Quicksort

Here, we use a random function to randomly generate a number in the array, the variable *left* is the left boundary of the current array, and the variable *right* is the right interface. We use rand()%(right − left) + left) to randomly select.

```
int partition(int* arr, int left, int right)
  swap(arr[rand()%(right − left) + left], arr[left]);
  int pivot = arr[left];
```

The pivot generated by randomisation may help to solve this problem, but at the same time, it should be noted that the random numbers generated by C++ are pseudo-random numbers generated according to algorithms and random seeds, and most of them have some drawbacks.

For example, the random numbers generated by the rand() function in a very short time are the same. In addition, in order to ensure the randomness of random numbers, these algorithms often require a lot of calculations, which will consume a lot of resources and affect our sorting speed.

7.1.4 Median of Three Partitioning

Select the median of the first, last as well as middle element

```
int partition(int* arr, int left, int right)
  int mid = (left + right)/2
  int pivot =
  min(min(max(left, mid), max(mid, right)), max(right, left))
```

Choosing the first or last one may be due to the fact that the array is close to being sorted and deteriorated to an [O(n] ^2) algorithm. Randomisation generation can also take a lot of time, and the speed of the algorithm is uncertain. Taking the median of the three numbers, it is likely to be able to reduce this situation. We consider this selection approach to be the best choice, amongst the available ones. Therefore, we choose this method for our research.

In fact, there are other improved versions of median of three, such as median of five and median of seven, however, they are essentially the same.

7.2 Further Optimisation

Although the current growth rate of our algorithm is almost satisfactory, we need to note that this is not the best strategy when the range of the array is too large or too small. One way of better implementation may be the STL sort function [9], which is included in the C++ header file <algorithm.h>.

The STL sort function is not just ordinary quicksort. In addition to optimising ordinary quicksort, it also combines insertion sort and heap sort. According to different

quantity levels and different situations, the appropriate sorting method can be automatically selected. When the amount of data is large, it will try to use the method of quicksort partition and recursion at first. Once the amount of data after partition is less than a certain threshold, to avoid excessive extra load caused by recursive calls, insertion sort will be used instead [7]. If the recursion level is too deep, there is a tendency for the worst case to occur, and heap sort will be used instead.

So here are other better sorting techniques in different situation.

7.2.1 Switch to Insertion Sort in Small Array

Fast sorting requires constant recursion. When the array is very small, we can use insertion sorting. Insertion sorting is to insert one element at a time on the basis of an already ordered small sequence. When the length of the sequence to be sorted is between 5 and 20, the use of insertion sort at this time can avoid some harmful degeneration situations [8]. It might be faster to use insertion sort in this case.

7.2.2 Switch to Heap Sort When the Recursion is Too Deep

Heap sorting mainly uses a data structure called a heap, also known as a binary heap. This is a sorting strategy based on comparison. The process of putting elements into the heap data structure is called heapify. Through the adjustment of the heap, we complete the final sorting process. It is often used when the amount of data is extremely large [10].

In addition, we also have other ways to optimise. Here is what we can do to further improve the efficiency of the algorithm.

7.2.3 Multiple-Pivot Quicksort

Since 70s of the last century, some researchers have been committed to implementing double-pivot and three-pivot quicksort algorithms. The paper published by Kushagra *et al.* [11] also talk about the probability of multi pivot quicksort. This quicksort algorithm uses n pivots to divide the original array into $n + 1$ arrays, which is a further divide-and-conquer algorithm. However, according to the results of Budiman *et al.* [12], the performance of the multi pivots algorithm have a great relationship with cache performance and it works best when the number of pivots is three in most cases.

7.2.4 Gather Elements with the Same Value

After a partition is over, the elements equal to pivot can be grouped together, and when the next division continues, there is no need to divide the elements equal to pivot again. In this way, by clustering the elements equal to pivot, the number of iterations can be reduced, and the efficiency will be significantly improved [13].

7.2.5 Find Better Pivot Strategy

As we discussed above, a better pivot selection strategy has a decisive influence on the performance of quicksort. It is necessary to choose a better pivot selection scheme to improve the quicksort algorithm.

Apart from these commonly used pivot selection methods mentioned, there are some other pivot selection methods that might be better. In some special cases, for example, when the most elements of the array are sorted except some of them, we can make targeted improvements to pivot selection strategy. Especially, in the field of engineering calculation or graphics calculation, a lot of repetitive work is often required.

7.2.6 Tail Recursion

If a recursive function calls itself at the end of the function, at this time, we can overwrite the current record instead of creating a new one, thereby improving efficiency. For example, the capacity of our code stack is limited. The conventional method can only sort an array of about 50,000. When we use tail recursion, we can effectively increase the maximum sort, double or even triple.

7.2.7 Multithreading and Multiprocessing

Hardware such as memory, CPU, etc. determine the processing speed. The unit that we allocate resources is the thread. Therefore, if we open multiple processes, more resources will be allocated and tasks will be completed faster. The main effect of multithreading is to increase the number of concurrencies.

In addition, we also use multi-threading to improve resource efficiency. Multiple tasks take turns using the CPU.

7.2.8 Efficiency of Quicksort

The complexity of our quicksort is approximately O(0.04nlogn), which has a certain relationship with the implementation of the algorithm and the configuration of the machine [14].

However, no matter what host machine it is running on, the growth rate would always be the same. In this part, we will analysis the complexity of quicksort in best average and worst case to show why our algorithm appears in that way.

For best case, each time partition can separate it to half, so from n to 1 we need do logn times. But for each level, we shall traverse all the elements.

Best: O(Nlogn)

For average case, we assume that partition can happen in n position each with probability 1/n.

$$\begin{cases} C\_avg(0)=0, \\ C\_avg(1)=0 \\ C\_avg=\sum_{s=0}^{n-1}\left[\frac{1}{n}\left(C\_avg(s)+C\_avg(n-1-s)\right)\right]+n-1 \end{cases} =2n\int_1^n \left[\frac{1}{x}dx\right] =2n\log n$$

Therefore, the complexity of average case is O(nlogn).

Worst: O(N^2)

In the worst case, the array is sorted or the elements in it are all equal, quicksort degenerates into bubble sort. The time complexity of bubble sort is O(n^2) because each element in the array will compare with other elements.

No O(N) Comparison-Based Sorting Algorithm

Suppose we have an array to be sorted which has the size of n. If we want to order it, we need to access each element at least once to know all the information, but we can't use the additional resource on the constraints of comparison-based sorting algorithm [15]. So, we need more actions to achieve the goal of sorting the array.

We carry out the comparison of sorting algorithms in pairs. We can abstract it into a decision tree. we compare the left child node and the right child node in each node. For an array of length n, there are n! combinations of elements. The result of sorted results is one of them. So, we have n! leaves for the decision tree.

Every time we do a sort, we eliminate at most half of the possibilities. We divide all cases where the left subtree is larger than the right subtree into one pile, and divide all cases where the right subtree is larger than the left subtree into another pile. At the beginning we have n! possibilities, and at the end we only have one possibility, which is the sorted array that we want.

Assume we get the final result after k comparisons:

$$\lfloor 1/2 \rfloor \hat{}k\, n! \leq 1$$

Therefore,

$$k \geq \lceil \log \rceil\_2\,(n!)$$

Because:

$$n! = n*(n-1)*(n-2)*.....*1$$

For $\lceil \log \rceil\_2\,(n!)$:

$$O(\lceil \log \rceil\_2\,(n!)) \leq O(\lceil \log \rceil\_2\,(n\hat{}n)) = O(n \lceil \log \rceil\_2\, n)$$

Beyond Comparison-Based Sorting Algorithm

For comparison-based sorting algorithm, it is possible to slightly improve it. For example, we might improve its time complexity from 0.05nlogn to 0.03nlogn. However, it is impossible to improve it from O(nlogn) to O(n) unless we use additional resources.

It is for sure that we cannot create an O(n) comparison-based sorting algorithm, but if we use additional spaces, we do have better techniques. We have briefly introduced them below:

7.2.9 Radix Sort

We divide the maximum value in the array according to the number of digits, and then sort it by units, then by 10, compare each bit, and so on to make it sorted. The data are fetched according to the queue's rule: first in, first out. The time complexity would be O(kn).

7.2.10 Count Sort

The first step of count sort is that we create an auxiliary array to store elements in it. We then traverse the elements in the original array, use the elements in the original array as the index of the count array, and use the number of occurrences of the elements in the original array as the count array element value.

7.2.11 Bucket Sort

As an extended version of counting and sorting, we first group the sorted numbers into several different buckets. Then we use the mapping function to calculate the corresponding mapping value of the elements in the array. When needed, we directly access it through the subscript. It needs O(n) times operation to do the mapping. However, once the mapping is done in advance, the speed can reach O(1) when searching [16, 17].

8 Concluding Discussions

General Beacon devices or bluetooth low energy (BLE) [18] or any IoT sensors [19] uses improvable programs which can be more effective after improving. Retailers, marketers or industrialists can use data or can easily perform any operation using improved programs.

Limitations of the customer devices or IoT sensors are mainly due to software and hardware constraints. These resource constraint devices sometimes do not have enough space or processing power to run various programmes, such as traditional encryption mechanisms, blockchain, etc. Lightweight cryptography, lightweight blockchain of things (BCoT) have thus became emerging research and development fields. However, the level of security compromised for making them light weight remains a concern which needs to be studied further.

Obviously without these customer devices, blockchain, IoT and sensors, we cannot think about dynamic advancement in industrial, agricultural or in any other sectors or significant revolution in the technological world. When all these are combined, including the fusion and blockchain and IoT i.e. BCoT, in smart manufacturing to manufacture incredible inventions, software is also needed to compete with the journey of the combinations of all of the hardware.

This research enhances utilisation of memory in efficient way. More detailed research needs to be conducted in the future to eliminate limitations by focusing on several factors, e.g. better pivot selection, Hoare's partitioning scheme, handling repeated elements, using tail recursion, hybrid with insertion sort, etc.

In this article, a cache-efficient sorting algorithm has been presented which maps to the GPUs. Techniques to further enhance the computational performance have also been put forward. The proposed novel sorting algorithm not only makes comparatively fewer memory accesses but also demonstrates better locality in the patterns of data access. Taking into account the input sequence's sorted nature, if significantly enhances the overall performance.

Our future plan includes to apply the proposed algorithm for other data mining applications particularly, in the domain of smart industry. In addition, we also aim to

develop cache-friendly and efficiently algorithms for other types of computations which will eliminate limitations of memory of BCoT devices, sensors and others.

References

1. Al Hussain, A., Emon, M.A., Tanna, T.A., Emon, R.I., Onik, M.M.H.: A systematic literature review of blockchain technology adoption in Bangladesh. Ann. Emerg. Technol. Comput. (AETiC) **6**(1), pp. 1–30 (2022). Print ISSN: 2516-0281, Online ISSN: 2516-029X. https://doi.org/10.33166/AETiC.2022.01.001, http://aetic.theiaer.org/archive/v6/v6n1/p1.html
2. Emon, R.I., et al.: Privacy-preserved secure medical data sharing using hierarchical blockchain at the edge. Ann. Emerg. Technol. Comput. (AETiC) **6**(4), 38–48 (2022). Print ISSN: 2516-0281, Online ISSN: 2516-029X. https://doi.org/10.33166/AETiC.2022.04.005, http://aetic.theiaer.org/archive/v6/v6n4/p5.html
3. Malik, A.D., Jamil, A., Omar, K.A., Abd Wahab, M.H.: Implementation of faulty sensor detection mechanism using data correlation of multivariate sensor readings in smart agriculture. Ann. Emerg. Technol. Comput. (AETiC) **5**(5), 1–9 (2021). Print ISSN: 2516-0281, Online ISSN: 2516-029X. https://doi.org/10.33166/AETiC.2021.05.001, http://aetic.theiaer.org/archive/v5/v5n5/p1.html
4. Liu, A., Khatun, M.S., Liu, H., Miraz, M.H.: Lightweight blockchain of things (BCoT) architecture for enhanced security: a literature review. In: 2021 International Conference on Computing, Networking, Telecommunications & Engineering Sciences Applications (CoNTESA), pp. 25–30. IEEE (2021)
5. Onik, M.M.H., Miraz, M.H., Kim, C.S.: A recruitment and human resource management technique using blockchain technology for industry 4.0. In: Smart Cities Symposium 2018, pp. 1–6. IET (2018)
6. Vaz, R., Shah, V., Sawhney, A., Deolekar, R.: Automated big-O analysis of algorithms. In: 2017 International Conference on Nascent Technologies In Engineering (ICNTE) (2017). https://doi.org/10.1109/icnte.2017.7947882
7. Song, H., Fu, Y., Zhang, L., Peng, H., Liang, H.: Multi-thread quicksort algorithm based on partitioning. J. Comput. Appl. **30**(9), 2374–2378 (2010). https://doi.org/10.3724/sp.j.1087.2010.02374
8. Shaffer, C.: Data Structures and Algorithm Analysis in C++. Dover Publications (2012)
9. Bahig, H.M.: Complexity analysis and performance of double hashing sort algorithm. J. Egypt. Math. Soc. **27**(1), 1–12 (2019). https://doi.org/10.1186/s42787-019-0004-2
10. Li, H., Chen, P., Wang, Y.: Heap sorting based on array sorting. J. Comput. Commun. **05**(12), 57–62 (2017). https://doi.org/10.4236/jcc.2017.512006
11. Kushagra, S., López-Ortiz, A., Qiao, A., Munro, J.: Multi-pivot quicksort: theory and experiments. In: 2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 47–60 (2013). https://doi.org/10.1137/1.9781611973198.6
12. Budiman, M., Zamzami, E., Rachmawati, D.: Multi-pivot quicksort: an experiment with single, dual, triple, quad, and penta-pivot quicksort algorithms in python. In: IOP Conference Series: Materials Science and Engineering, vol. 180, p. 012051 (2017). https://doi.org/10.1088/1757-899x/180/1/012051
13. Wild, S.: Dual-pivot and beyond: the potential of multiway partitioning in quicksort. IT – Inf. Technol. **60**(3), 173–177 (2018). https://doi.org/10.1515/itit-2018-0012
14. Jadoon: Design and analysis of optimized stooge sort algorithm. Int. J. Innov. Technol. Explor. Eng. **8**(12), 1669–1673 (2019). https://doi.org/10.35940/ijitee.l3167.1081219
15. Bustos, B., Pedreira, O., Brisaboa, N.: A dynamic pivot selection technique for similarity search. In: First International Workshop on Similarity Search and Applications (Sisap 2008) (2008). https://doi.org/10.1109/sisap.2008.12

16. Faujdar, N., Saraswat, S.: The detailed experimental analysis of bucket sort. In: 2017 7Th International Conference on Cloud Computing, Data Science & Engineering - Confluence (2017). https://doi.org/10.1109/confluence.2017.7943114
17. Faujdar, N., Ghrera, S.: Performance evaluation of parallel count sort using GPU computing with CUDA. Indian J. Sci. Technol. **9**(15) (2016). https://doi.org/10.17485/ijst/2016/v9i15/80080
18. Cordiglia, M., Van Belle, J.-P.: Consumer attitudes towards proximity sensors in the South African retail market. In: Proceedings of the 2017 Conference on Information Communication Technology and Society (ICTAS), pp. 1–6 (2017). https://doi.org/10.1109/ICTAS.2017.7920651
19. Miraz, M.H., Ali, M.: Integration of blockchain and IoT: an enhanced security perspective. Ann. Emerg. Technol. Comput. (AETiC) **4**(4), 52–63 (2020). Print ISSN: 2516-0281, Online ISSN: 2516-029X. https://doi.org/10.33166/AETiC.2020.04.006, http://aetic.theiaer.org/archive/v4/v4n4/p6.html