QUALITATIVE SOFTWARE ENGINEERING AND PARALLEL SORTING ALGORITHM FOR REAL NUMBERS

A THESIS IN
Computer Science

Presented to the Faculty of the University of
Missouri-Kansas City in partial fulfillment of
the requirements for the degree

MASTER OF SCIENCE

By
MD USMAN GANI SYED

B. Tech(Hons), National Institute of Technology-Jamshedpur, India, 2015

Kansas City, Missouri
2018

QUALITATIVE SOFTWARE ENGINEERING AND PARALLEL SORTING ALGORITHM FOR REAL NUMBERS

Md Usman Gani Syed, Candidate for the Master of Science Degree
University of Missouri – Kansas City, 2018

ABSTRACT

The research work consists of two parts. Part one is about qualitative software engineering and Event-B modelling for class and Use case diagrams. Now a days distributed and parallel applications are most popular and are used in applications like telecommunications and aircraft systems with complex computations. It is very important to define the exact properties and features of these systems along with the workflow. UML provides a great opportunity of modelling complex applications but lacks in providing the detailed semantics. In this work, we have provided the importance of implementation of specifications using formal methods like event-B through a simple example and verify its results using ProB. Later, we have defined the UML diagrams like use case and class diagrams in various scenarios and have performed the Event B modeling for these examples. The part one report had been published as a research paper to "The 2018 International Conference on Computational Science and Computational Intelligence 2018, Las Vegas, USA". The paper was accepted to the conference with Paper Id "CSCI6051".

Part two is on parallel Sorting algorithm on real numbers. There are various best algorithms for sorting integers. The current research work applies the recent important results of serial sorting of real numbers in $O(n\sqrt{logn})$ time to the design of a parallel algorithm for sorting real numbers in O($\log^{1+\varepsilon}n$) time and $O(\frac{nlogn}{\sqrt{loglogn}})$ operations. This is the first NC algorithm known to take o(nlogn) operations for sorting real numbers.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of School of Computing and Engineering, have examined a thesis titled "Qualitative Software Engineering and Parallel Sorting Algorithm for Real Numbers" presented by Md Usman Gani Syed, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yijie Han, Ph.D., Chair
School of Computing and Engineering

Baek-Young Choi, Ph.D.
Department of Computer Science Electrical Engineering

Xiaojun Shen, Ph.D.
Department of Computer Science Electrical Engineering

CONTENTS

ILLUSTRATIONS

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank following people who have directly or indirectly helped me in academic achievements. Firstly, I would like to thank Dr. Yijie Han for the constant and endearing support which has helped me in fulfilling my thesis. He has provided me with an opportunity to realize my potential in the field of my thesis. His encouragement and inputs were elements of vital guidance in my thesis. He has been a constant source of motivation and challenged me with algorithms, deadlines, that have contributed to me acquiring inspiration and ideology. His expertise and innovative insights have been phenomenal in completing my thesis.

I sincerely thank Dr. Baek-Young Choi and Dr. Xiaojun Shen and for accepting to be a part of my thesis committee and making time for me from their busy schedule. I would like to thank the University of Missouri- Kansas City for providing me with an opportunity to continue my research and supporting me in this regard.

I would like to dedicate my thesis to my parents who constantly inspired me to pursue higher studies. I would like to thank my family who stood behind me all these years during my degree. Finally, I would like to thank all my teachers, educational administrators, present, and past and all who helped me achieve this academic goal.

CHAPTER 1

INTRODUCTION

To develop a software, we gather requirements, specify the design and then implement. When we consider design specifications, there are various models for representing it and the most popular one which industry uses widely is UML. They give good understanding of the design through graphical representation and help developers, project management and customers to understand the basic structure and requirements of the system. However, UML lacks in providing the precise semantics which result in wrong implementation of their specifications. These flaws rise as bugs during the production cycle. There is a cost factor for every bug. If it's a small bug, it can be debugged, fixed and closed. But when it comes to safety critical complex systems like aircraft management, automatic train operation system etc, bugs are not compromised at any cost. A bug in their operation systems means, we are putting human life in danger. So we need more quality way of specifications before actual implementation. This can be achieved through formal methods. They provide mathematical foundation for a software. The specifications in formal methods are done at low level and are verified before the actual implementation cycle starts. So by following formal methods quality softwares can be generated. There are various formal methods and tools to implement. In first part of this work, we try to focus the implementation of formal methods using event-B and Rodin tool. we will see how to debug and verify our system specifications using tool ProB and then we have defined the UML diagrams like use case and class diagrams in various scenarios and have performed the Event B modeling for these examples.

The second part of this work is about parallel sorting algorithm of real numbers. In computer science, algorithms plays an important role. The performance of the developed system is determined by the efficient algorithm used by it. There are various algorithms available for various purposes. The second part of work is done on the parallel sorting algorithm for real numbers which produces best results in terms of number of operations. To achieve this I have used the recent best results of sequential sorting of real numbers[14] by Dr. Yijie Han.

CHAPTER 2

QUALITATIVE SOFTWARE ENGINEERING USING EVENT-B

This chapter explains the importance of qualitative software engineering and uses Event B to specify precise semantics of the requirements. I(Md Usman Gani Syed) have completed this research for the course CS552(Formal Software Specifications) under the guidance of Dr. Yijie Han along with other two more members of this course, namely, Sravanthi Gogadi and Bhargavi Nadendla. My self acted as a primary author and point of contact we as a team of four published our work of this course as a research paper with title “Qualitative Software Engineering and Event-B Modeling for Communication and Use Case Diagrams” to “The 2018 International Conference on Computational Science and Computational Intelligence, Las Vegas, USA”. The paper was accepted to the conference with Paper Id “CSCI6051”. In next sections lets understand our work on this topic.

Now understand how do define the requirements precisely using Event-B notation. To do so, we should understand the elements of Event-B first.

2.1 Elements of Event-B:

The overall definition of a software development is achieved using “model”[1] in Event-B. A model in Event-B[2] consists of various elements called “variables”, “events”, “invariants”, “assertions” and “initializations” where variables are the features of a model and these represents the states. Events are the operations performed in the model and these represent the transitions between the states(variables) in the model. Invariants are the properties that a variable hold and these are the first order logic expressions which has to be true while executing a program. Assertions are to be derived and proved from the first order logic expressions(invariants). Assertions are to be proved for the most vulnerable invariants and need not to be proved for all the properties of variables. Initializations help to give the initial values to the variables. These set the default states of the system.

We will go through a simple example of a traffic signal system which was given in the documentation of Rodin tool [21]. We will implement it using event-B on Rodin and verify it with the ProB. We assume that our traffic signal system consists of two states, green(TRUE) and red(FALSE) and two types of users, Pedestrians and Vehicles (or simply cars). The model is shown
in figure 1. Initially we create a MACHINE named traffic system (1) and then we create two variables cars_go(2) and peds_go(3). If the status of peds_go is TRUE, then pedestrians can go. If FALSE, then they cannot go. Similarly for cars. We have not yet declared variable types till now. So we create two invariants inv1(4) and inv2(5) to define the variable types. Now we initialize these variables cars_go and peds_go with FALSE at (7) and (8) respectively. Now that we have created variables and assigned with initial values. The state of these variables can be changed by using events. Lets create two events set_peds_go(9) and set_peds_stop(11). The event set_peds_go will have an action act1 which sets peds_go to TRUE(10) on entering this event. Similarly set_peds_stop will have an action that sets peds_go to FALSE(12) upon entering this event. For setting the traffic light of cars, instead of creating two events, lets handle it by creating one event and its status will be set by using the value of the parameter with which it is passed. To do so, we define parameter new_value using ANY and create an event-B guard using WHERE to define the domain of the parameter. On entering this event, we set the status of the variable

cars_go to the value of the parameter.

Now we have completed the domain of our model. Now lets focus on the requirement of our problem. The requirement is both traffic lights must not be true at the same time. We can model this requirement by using the invariant "$\neg((\text{cars\_go} = \text{TRUE}) \land (\text{peds\_go} = \text{TRUE}))$" at (6). Now that we have built our model and lets work on the proof obligations (Consistency of system). Obviously, our current model has a bug and Rodin will tell us that our system is inconsistent in its Proof Obligations column. We debug our model using ProB tool and find where our invariant (6) has been violated.

```
MACHINE
   traffic_system ›                                (1)
 VARIABLES
  ○ cars_go  ›                                     (2)
  ○ peds_go  ›                                     (3)
 INVARIANTS
  ○ inv1:  cars_go ∈ BOOL  ›                       (4)
  ○ inv2:   peds_go ∈ BOOL  ›                      (5)
  ○ inv3:  ¬((cars_go = TRUE)
              ∧ (peds_go = TRUE))  ›               (6)
 EVENTS
 ○ INITIALISATION:  ›
   THEN
    ○ act1:  cars_go ≔ FALSE ›                     (7)
    ○ act2:  peds_go ≔ FALSE ›                     (8)
    END
  ○ set_peds_go:  ›                                (9)
    THEN
   ○ act1:  peds_go ≔ TRUE ›                       (10)
    END
  ○ set_peds_stop:  ›                              (11)
    THEN
    ○ act1:  peds_go ≔ FALSE ›                     (12)
     END
  ○ set_cars:  ›                                   (13)
    ANY
   ○ new_value  ›                                  (14)
    WHERE
    ○ grd1:  new_value ∈ BOOL  ›                   (15)
     THEN
     ○ act1:  cars_go ≔ new_value ›                (16)
     END
 END
```

Figure 1: Event-B model

```
MACHINE
   traffic_system ›                                (1)
 VARIABLES
  ○ cars_go  ›                                     (2)
  ○ peds_go  ›                                     (3)
 INVARIANTS
  ○ inv1:  cars_go ∈ BOOL  ›                       (4)
  ○ inv2:   peds_go ∈ BOOL  ›                      (5)
  ○ inv3:  ¬((cars_go = TRUE)
              ∧ (peds_go = TRUE))  ›               (6)
 EVENTS
 ○ INITIALISATION:  ›
   THEN
    ○ act1:  cars_go ≔ FALSE ›                     (7)
    ○ act2:  peds_go ≔ FALSE ›   (8)
    END
  ○ set_peds_go:  ›                                (9)
    WHEN
    ○ grd1:  cars_go ≔ FALSE ›                     (17)
    THEN
    ○ act1:  peds_go ≔ TRUE ›                      (10)
    END
  ○ set_peds_stop:  ›                              (11)
    THEN
    ○ act1:  peds_go ≔ FALSE ›   (12)
     END
  ○ set_cars:  ›                                   (13)
    ANY
   ○ new_value  ›                                  (14)
    WHERE
    ○ grd1:  new_value ∈ BOOL  ›                   (15)
    ○ grd2:  new_value = TRUE
                 ⇒peds_go = FALSE ›                (18)
     THEN
     ○ act1:  cars_go ≔ new_value ›                (16)
     END
 END
```

Figure 2: Event-B model after system correction

2.2 Debugging:

Total we have 3 events, namely set_peds_go, set_peds_stop and set_cars which will be altering the states of the two variables peds_go and cars_go

1. set_peds_go will set the status of peds_go to TRUE.

2. set_peds_stop will set the status of peds_go to FALSE.

3. set_cars will set the status of cars_go to whatever the status value it receives in parameter.

In ProB follow these below steps to debug.

Step1: Click the event set_peds_stop and pass FALSE as parameter to set_cars. That means peds_go =FALSE and cars_go = FALSE.Invariant(6) = TRUE. So it did not violate our invariant(6).

Step2: Similarly, click set_peds_go and pass FALSE as parameter to set_cars. That means peds_go = TRUE and cars_go = FALSE. It implies pedestrians can go but not cars. Invariant(6) = TRUE. So it did not violate our invariant(6).

Step3: Now, click set_peds_stop and pass TRUE as parameter to set_cars. That means peds_go = FALSE and cars_go = TRUE. It implies cars can go but not pedestrians. Invariant(6) = TRUE. So it did not violate our invariant(6).

Step4: Similarly, click set_peds_go and pass TRUE as parameter to set_cars. That means peds_go = TRUE and cars_go = TRUE. It implies both pedestrians and cars can go. Invariant (6) = FALSE. It clearly violated our invariant (6).

The above process of checking the system with all possible combinations can also be done automatically. Now, how to fix our model is by setting peds_go to TRUE when cars_go is false. Similarly setting peds_go to FALSE when cars_go is TRUE. This can be achieved by adding extra guards to the events. Add guard cars_go=FALSE (17) under the event set_peds_go. This means peds_go will be set to TRUE only when cars_go is FALSE. Similarly, add peds_go to FALSE (18) when the parameter new_value is set to

TRUE under the event set_cars. Now our system is consistent and no errors. The ProB can also be used for deadlock checking, test-cases generation etc. But we limit to here in this paper. The modified model is shown in fig2.

2.3. UML Communication Diagrams:

UML is a general purpose semi formal language defined to visualize the design of a system[18]. There are a various set of diagrams described to visualize the workflow of a UML model and these set of diagrams need not describe the entire UML model and addition of deletion of a diagram doesn't affect the model much. Communication diagram is one such diagram which explains the workflow between the objects[8] i.e., about the interactions between various objects in a model. Communication diagram is a good source of visualizing a scenario than that of the sequence diagram as they define all the effects that would exists between objects and therefore can serve better in the procedural development of the system. Let us illustrate this with an example of multiple clients requesting resources from a server. Assume that, a server can serve a fixed number of clients at a time by doing multithreading[17] and the following diagram specifies the interaction between various objects,

1. Browsers send multiple requests to the server for resources at an instance
2. Servers perform multithreading by serving the threads scheduled by a scheduler
3. Servers serve the browsers with the requested resources

The translation of the below communication diagram to Event-B in Rodin can be seen as follows[20] (figure 3, 4).

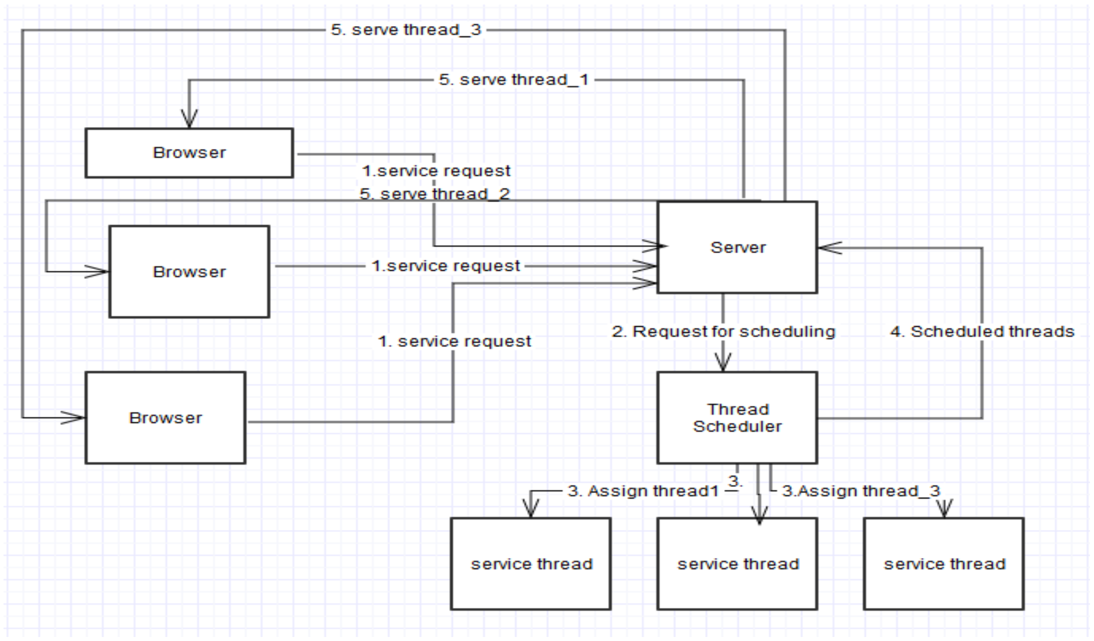

Figure 3: UML Communication diagram for multi cast client-server communication

CONTEXT
Client Server.. C ›
SETS
- Objects ›
  - Interactions ›

CONSTANTS
  - Browsers›
  - Server Scheduler ›
  - Server Manager ›
  - Web request ›
  - Service Request ›
  - Scheduler Request ›

MACHINE
Client Server.. C ›
SEES
- Request Object
  - Server Object
- Request Message
  - Sequence

EVENTS
- Initialization: not extended
  ordinary
  END
  - Serve Request 1: not extended
    ordinary
  END
- Server Request 2: not extended
  ordinary
  END

Figure 4:Transformed Event-B for communication diagram client-server communication

2.4 Use Case Diagram

A Use case diagram[20] is used to represent the correlation or a list of actions between the system elements and the actor. It is used to show the relationships and dependencies in the system.An actor can be a human or a system feature, used to specify a role played by the user or any other system that interacts with the subject. UML[5] 2 does not permit associations between the actors It is used to identify different users of the system and their use cases. These are commonly used by the stakeholders to provide a high level view of the system and specifically define the requirements of different users.

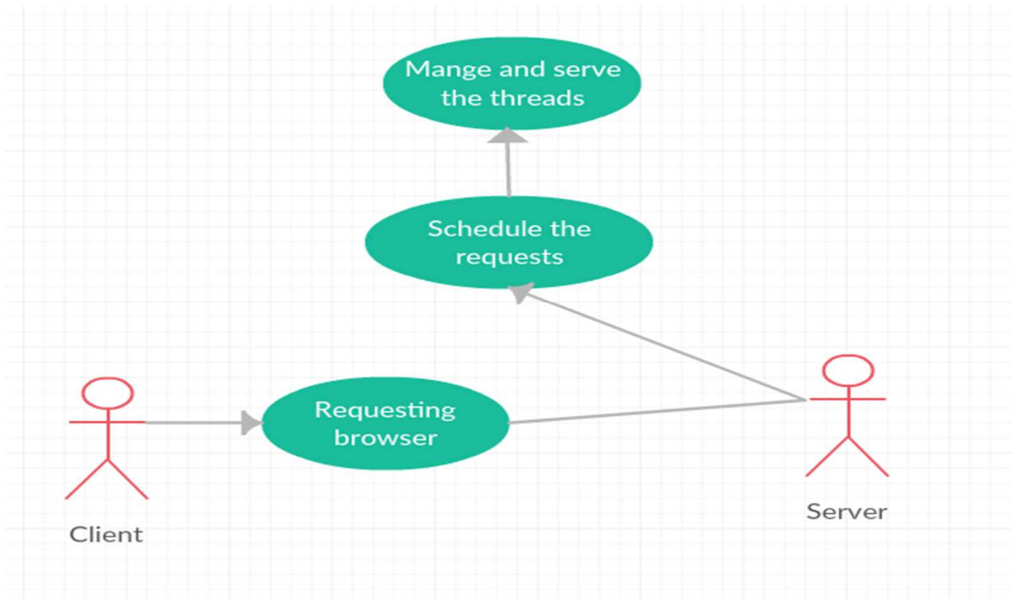

Figure 5: Usecase diagram for client-server communication

```
CONTEXT
        Client Server.. C ›
 SETS
o       Objects         ›
        o       Interactions        ›
CONSTANTS
        o       Browsers›
        o       Server Scheduler ›
        o       Server Manager ›
        o       Web request ›
        o       Service Request ›
        o       Scheduler Request ›
```

```
MACHINE
        Client Server.. C ›
 SEES
o       Request Object
        o       Server Object
o       Request Message
        o       Sequence
 EVENTS
o       Initialization:  not extended
                              ordinary
             END
        o       Serve Request 1: not extended
                                ordinary
             END
o       Server Request 2: not extended
                               ordinary
             END
```

Figure 6: Transformed Event-B for use case diagram of client-server communication

CHAPTER 3

A PARALLEL SORTING ALGORITHM FOR REAL NUMBERS

3.1 Introduction

In this chapter we use the results of sequential sorting algorithm for real numbers. Using those results we explain our approach for parallel sorting of real numbers in O($\log^{1+\varepsilon}$n) time and $O(\frac{nlogn}{\sqrt{loglogn}})$ operations. I have completed this work under Dr. Yijie Han along with the co-author Sneha Mishra.

It is known widely that serial comparison sorting takes θ(nlogn) time [10]. Although integer sorting can outperform the Ω(nlogn) lower bound for sorting integers [7,12,14,15,16,17], these algorithms generally do not apply to the problem of sorting real numbers. It has been known that integers can be sorted in O(nloglogn) time and linear space [14,15], the O(nlogn) time bound remains for sorting real numbers ever since. Recent results shown by Dr. Han which we explained above prove that real numbers can be converted to integers for the sorting purpose in $O(n\sqrt{logn})$ time [13], thus enabling the serial sorting of real numbers in $O(n\sqrt{logn})$ time.

Parallel sorting algorithms for sorting real numbers run on the PRAM (Parallel Random Access Machine) model are known [4,9]. The AKS sorting network [4] can be transformed into an EREW (Exclusive Read Exclusive Write) PRAM algorithm with O(logn) time and O(nlogn) operations. Cole's parallel merge sort [9] sorts n numbers in O(logn) time using n processors on the EREW PRAM. On the CRCW (Concurrent Read Concurrent Write) PRAM Cole showed [9] that his parallel merge sort can run in O(logn/loglog(2p/n)) time using p processors. Also see[23].

There are also parallel algorithms for integer sorting [7,12,14,15,16,17]. In the case of integer sorting the operation bound can be improved to below O(nlogn). In particular, [16] presents a CRCW PRAM integer sorting algorithm with O(logn) time O(nloglogn) operations and [15] presents an EREW PRAM integer sorting algorithm with O(logn) time and $O(n\sqrt{logn})$ operations.

For sorting real numbers the previous best serial algorithm sorts in O(nlogn) time. It was also known that for comparison sorting Ω(nlogn) is the tight lower bound. Thus if we use comparison sorting to sort real numbers then in serial algorithms we cannot avoid the Ω(nlogn) time bound and in parallel algorithms we cannot avoid the Ω(nlogn) operation bound. In the past no other sorting methods are known to sort real number in less than O(nlogn) time and comparison sorting remained the norm for sorting real numbers.

However the situation is recently changed completely as we found a way to convert real numbers to integers for sorting purpose and therefore we can sort real numbers in $O(n\sqrt{logn})$ time [13]. This result enables us to move further to improve the operation bound of parallel algorithms for sorting real numbers to below O(nlogn), as in the past all parallel algorithms for sorting real numbers has an operation bound at least O(nlogn).

In this work we will apply the $O(n\sqrt{logn})$ time serial real number sorting algorithm to the design of an NC algorithm with $O(\log^{1+\varepsilon} n)$ time and $O(\frac{nlogn}{\sqrt{loglogn}})$ operations on the CREW (Concurrent Read Exclusive Write) PRAM. NC algorithms are parallel algorithms with polylog time and polynomial operations. Algorithm in [13] is an inherently serial algorithm and without much parallelism within it. Here we use it in the design of an NC algorithm with $O(\log^{1+\varepsilon} n)$ time and $O(\frac{nlogn}{\sqrt{loglogn}})$ operations.

The computation model used for designing our algorithm is the CREW PRAM. On this model in one step any processor can read/write any memory cell. Concurrent read of one memory cell by multiple processors in one step is allowed and concurrent write of one memory cell by multiple processors in one step is prohibited. Parallel algorithms can be measured with their time complexity and the number of processors used. They can also be measured with time complexity and operation complexity which is the time processor product. The operation complexity ($T_p p$, with T_p time using p processors) of a parallel algorithm is often compared with the time T_1 of the best serial algorithm. In general $T_p p \geq T_1$. When $T_p p = T_1$ the parallel algorithm is said to be an operation optimal algorithm.

3.2 The Algorithm

Consider an algorithm for sorting n real numbers. Suppose each of the n/m lists with m real numbers in each list have already been sorted. We are to merge these n/m lists into one sorted list. We will do a-way merge in each pass to merge every a lists into 1 sorted list and there are log(n/m)/loga passes to have all n/m lists merged into 1 sorted list.

For simplicity, let us break down n elements into lists with m elements in each list. We can do parallel sort on the individual list of m elements recursively. Now we pick every a lists and have them merged together. This a-way merging is done in parallel to increase the computation time for the entire sorting process.

The a-way merging of sorted lists L_0, L_1, …, L_{a-1} is done as follows. For each sorted list of m real numbers we pick every a^2-th real number, i.e. we pick the 0th real number, the a^2-th real number, the $2a^2$-th real number, the $3a^2$-th real number, …, and so on. Thus from each list L_i we picked m/a^2 real numbers and these m/a^2 real numbers forms a sorted list L_i'. and from these a lists we picked m/a real numbers they form sorted lists L_0', L_1', .., L_{a-1}'. We merge L_0', L_1', ..,L_{a-1}' into one sorted list L' using Valiant's merging algorithm [24] (its improved version is given by Kruskal in [22] with time complexity of O(loglogm) and linear operations for merging two sorted lists of m elements each) in loga passes and O(loglogm) time and O(m/a^2) operations in each pass. Thus the total time for merging L_0', L_1', .., L_{a-1}' is O(logaloglogm) and the total operation is O(mloga/a^2). Now for each real number r in L_i' and for any L_j' r knows the largest real number s in L_j' that is smaller than r and smallest real number l in L_j' that is larger than r. s and l are actually neighbors in L_j'. There are a^2 elements between s and l in L_j. r then uses binary search in O(loga) time to find the largest real number among these a^2 real numbers that is smaller than r and the smallest real number that is larger than r. That is, r finds the exact insertion point of r in L_j. Because there a lists and there are m/a real numbers in L' thus the time for this binary search is O(loga) and the operation is O(mloga). The operation for all lists is O(nloga/a) because there are n/m lists and every a lists are merged in the a-way merge, we picked n/a^2 real numbers and every one of them has to use a processors to check a lists in the a-way merging. Because r is arbitrary picked and thus we know that every real number in L' knows its insertion point in every L_j. Let the real numbers in sorted order in L' be r_0, r_1, …, $r_{m/a-1}$. To merge L_0, L_1, …, L_{a-1} we need now to merge or sort all real numbers between the insertion points of r_i and r_{i+1} in L_0, L_1, …, L_{a-1}. There are no more than a^2 real numbers in L_j between the insertion points of r_i and r_{i+1} and therefore the total number R(i, i+1) of real numbers (call them a block) in L_0, L_1, …, L_{a-1} between the insertion points of r_i and r_{i+1} is no more than a^3 (i.e. R(i, i+1) $\leq$ a^3). When R(i, i+1) < a^3 we will combine multiple blocks together to reach a^3 real numbers. We use the $O\left(n\sqrt{logn}\right)$ serial sorting algorithm to sort them in $O(a^3\sqrt{loga})$ time. This represents $O(a^3\sqrt{loga})$ time and $O(n\sqrt{loga})$ operations in our parallel algorithm.

Thus the time for each stage is $O(a^3\sqrt{loga})$ and the operation for each stage is $O(n\sqrt{loga})$. When we start with m as a constant then there are logn/loga stages and therefore the time of our algorithm is $O(\frac{a^3logn}{\sqrt{loga}})$ and the operation is $O(\frac{nlogn}{\sqrt{loga}})$. Pick a=$\log^{\varepsilon}$n, we get O($\log^{1+\varepsilon}$n) time and $O(\frac{nlogn}{\sqrt{loglogn}})$ operations.

3.3 Procedure

Step 1: Lets say we have ‘m’ sorted elements in each list, and we have a total of ’n’ elements to sort. This implies that we have ’n/m’ lists to sort. To sort these blocks, we will apply a-way merging.

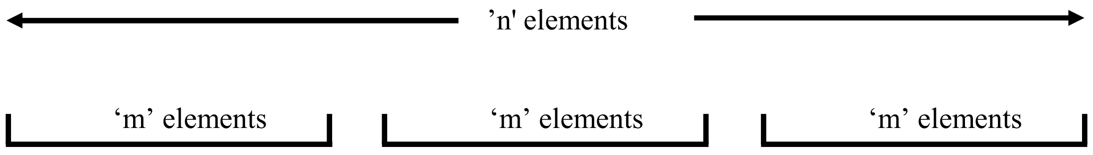

Figure 1

Step 2: Each stage of a-way merging is to merge every a sorted lists into 1 sorted list. This is repeatedly until all n/m lists are merged into one list.

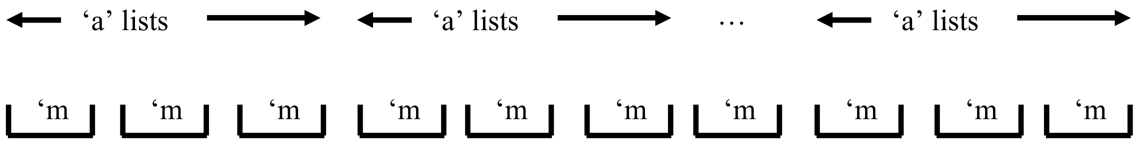

Figure 2

Step 3: To merge a lists into 1 list, we need to pick the '0-th', 'a^2-th', '$2a^2$-th', '$3a^2$-th' real numbers in each list L_i to form a new list L_i' of m/a^2 elements. This is shown as the figure below.

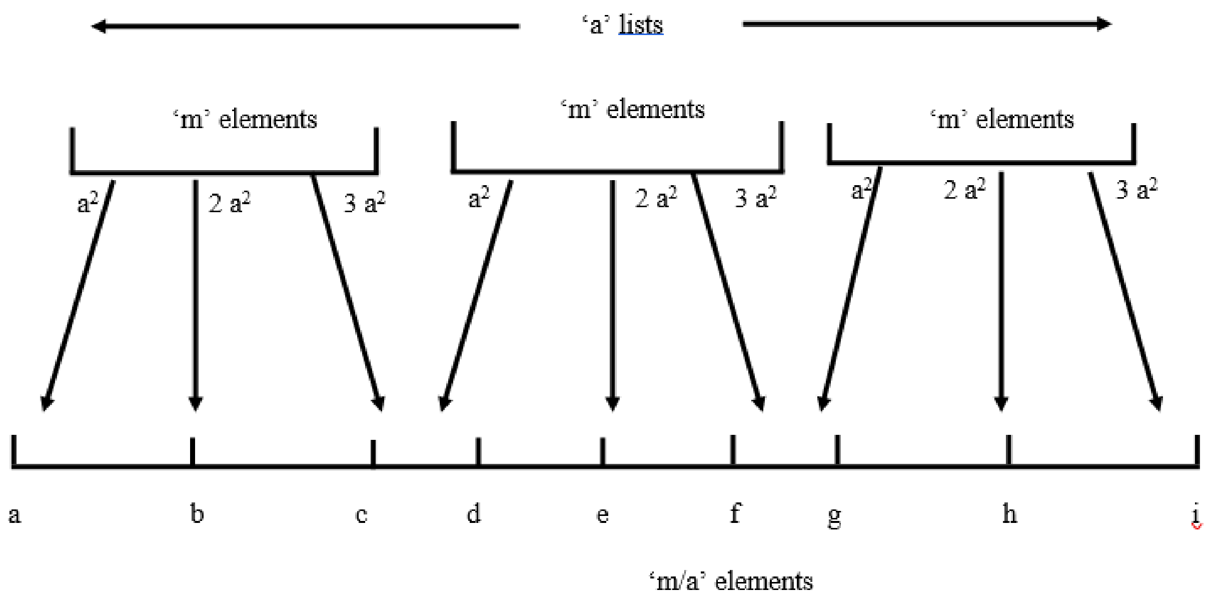

Figure 3

Step 4: We merge L_0', L_1', .., L_{a-1}' into one sorted list L'. The elements of this new formed list L' then use binary search to find their exact insertion point in L_0, L_1, L_{a-1}. These insertion points then partition L_0, L_1, …, L_{a-1} into m/a blocks with each block containing no more than a^3 real numbers.

Step 5: When every one of these m/a blocks are sorted we effectively merged L_0, L_1, …, L_{a-1} into one sorted list L.

Main Theorem: n real numbers can be sorted in $O(\log^{1+\varepsilon} n)$ time and $O(\frac{nlogn}{\sqrt{loglogn}})$ operations on the EREW PRAM.

CHAPTER 4

IMPROVED PARALLEL SORT ALGORITHM

In this chapter we will optimize the previous algorithm and reduce the time complexity. Let say we have ‘n’ real numbers and divide these numbers into n/m groups with each group consisting of ‘m’ numbers.

4.1 Algorithm

Step1:

Assign one processor to each of these n/m groups. Sort each group of ‘m’ numbers parallelly using the sequential sort[13] algorithm for real numbers. The time taken for this is $O(m\sqrt{\log m})$ and the number of operations are $O(n\sqrt{\log m})$.

Step2:

In this step lets do another level of grouping. Lets call each group as a super group. Each of this super group will have ‘a’ number of lists(where each list is a group of m sorted numbers which we got in step 1). Lets call these lists as L_0, L_1, …, $L_{a-1.}$ So, each of these super group will have ‘ma’ numbers and total there are n/ma super groups. For each sorted list of m real numbers we pick every a^3-th real number, i.e. we pick the 0^{th} real number, the a^3-th real number, the $2a^3$-th real number, the $3a^3$-th real number, …, and so on. Thus from each list L_i we picked m/a^3 real numbers and these m/a^3 real numbers forms a sorted list L_i’. The sortest lists being L_0’, L_1’, .., L_{a-1}’. Now we do pair wise merging on these lists L_0’, L_1’, .., L_{a-1}’.

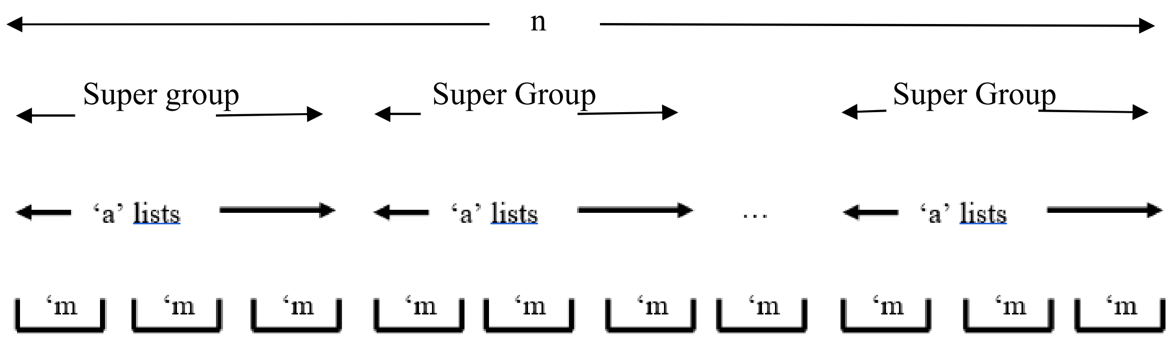

Fig. 4

We will do pair wise merging now. Each of the sorted lists (L_0', L_1', .., L_{a-1}') will be merged[22] with each of other (a-1) lists in parallel. i.e. L_0' will be merged with L_1', .., L_{a-1}' in parallel. L_1' will be merged with L_0', L_2', .., L_{a-1}'. So in this way each of L_2', L_3'… L_{a-1}', will also be merged with L_0', L_1', .., L_{a-1}' lists. So in total we are doing a^2 merges in parallel. Each merge sort takes O(loglogm) time and takes O(m/(a^3loglogm)) processors. The total operations in each merge is O(m/ a^3). Total we have a^2 merges. So we use O(m/ (aloglogm)) processors for parallel merging. This gives the total operations of O(m/a). The total operations for these n/ma super groups is O(n/a^2). Since all the merge sorts are in parallel the total time for these operations is still O(loglogm).

Step 3:

The goal in this step is to form the sorted list L'((consisting of m/a^2 elements) from L_0', L_1', .., L_{a-1}' by using the results of pair wise merging. After pair wise merging, each list will have 2m/a^3 elements. Lets say when L_0' is merged with each of L_1', .., L_{a-1}' we get new sorted lists and call them as $L_{0,1}$', $L_{0,2}$', .., $L_{0,a-1}$'. When L_1' is merged with each of L_0', .., L_{a-1}' we get new sorted lists as $L_{1,0}$', $L_{1,2}$', .., $L_{1,a-1}$'. So the new merged lists are of form $L_{i,j}$' where i, j lies in between 0 to a-1 and i not equal to j. When L_0' is merged with each of L_1', .., L_{a-1}', we get to know the rank of each of elements of L_0' in $L_{0,1}$', $L_{0,2}$', .., $L_{0,a-1}$'. Rank of each of elements of L_0' in L' is found by adding rank of that element in all the merged lists $L_{0,1}$', $L_{0,2}$', .., $L_{0,a-1}$'. The summation is done in O(loga) time by assigning 'a' processors to each of numbers in L_0'. Similarly ranks of all elements are found in each of elements in L_1', L_2', .., L_{a-1}'. Now we know all the ranks of elements in L_0', L_1', .., L_{a-1}' and so our new sorted list L' can be formed. This summation process is done in parallel. The total number of processors we use are m/a. So the total operations required to do these summation is O(mloga/a). The total operations required in all the n/ma super groups is O(nloga/a^2) and time is O(loga).

Step 4:

From now, the algorithm is same as discussed in previous chapter. In this step we assign 'a' processors to each of m/a^2 numbers present in L_1'. We assigned 'a' processors to each number as we have 'a' lists. Lets say for any real number 'p' in L', there are 'a' processors assigned to it. Each of those processors will find the exact insertion point of 'p' in the lists L_0, L_1, …, L_{a-1}. i.e 1st processor will find the exact insertion point of 'p' in L_0, 2nd processor will find the exact insertion point of 'p' in L_1, 3rd processor will find the exact insertion point of 'p' in L_2 and so on. Each processor uses binary search to find the exact insertion point and the time taken in O(logm) and the total number of operations are O(mlogm/a). Thus total operations for n/ma lists is O(nlogm/a^2).

Step 5:

For any two neighbor elements l and s of L' there exists at max a^3 elements between l and s in L_i. Let $r_0, r_1, \ldots, r_{m/a^2-1}$ be the real numbers in L'. we know that every real number in L' knows its insertion point in every L_i. To merge $L_0, L_1, \ldots, L_{a-1}$ we need now to sort all real numbers between the insertion points of r_i and r_{i+1} in $L_0, L_1, \ldots, L_{a-1}$. There are no more than a^3 real numbers in L_i between the insertion points of r_i and r_{i+1} and therefore the total number R(i, i+1) of real numbers (call them a block) in $L_0, L_1, \ldots, L_{a-1}$ between the insertion points of r_i and r_{i+1} is no more than a^4 (i.e. $R(i, i+1) \leq a^4$).

When $R(i, i+1) < a^4/2$ we will combine multiple blocks together to reach the block size in the range of $a^4/2$ to a^4 (i.e. $a^4/2 <$ block size $\leq a^4$) real numbers. We use the $O(n\sqrt{logn})$

serial sorting algorithm to sort them in $O(a^4\sqrt{loga})$ time. This represents $O(a^4\sqrt{loga})$ time and $O(ma\sqrt{loga})$ operations in our parallel algorithm. For all n/ma groups, theo total operations are $O(n\sqrt{loga})$

Thus the time for each stage is $O(a^3\sqrt{loga})$ and the operation for each stage is $O(n\sqrt{loga})$. When we start with m as a constant then there are logn/loga stages and therefore the time of our algorithm is $O(\frac{a^4 logn}{\sqrt{loga}})$ and the operation is $O(\frac{nlogn}{\sqrt{loga}})$. Pick $a=\log^{\varepsilon}n$, we get $O(\log^{1+\varepsilon}n)$ time and $O(\frac{nlogn}{\sqrt{loglogn}})$ operations.

4.2 Comparing Two Algorithms

In 2nd, 3rd and 4th steps, the previous algorithm takes $O(a^4\sqrt{loga} + logaloglogm)$ time and repeated for logn/loga steps. But in case of expanded algorithm, it takes

O($a^4\sqrt{loga}$+ loglogm+loga) time. If $a^4\sqrt{loga}$ <=logaloglogm, then optimized algortithm takes O(loglogm+loga) time through logn/loga steps and previous algorithm takes O(logaloglogm) time through logn/loga steps. Hence optimized algorithm is faster than previous algorithm.

CONCLUSION

In this first part of work, we have shown the implementation of formal methods using event-B through an example and verified it using ProB tool[19]. We proved that following formal methods, we can develop quality products. Later we have proposed Event B modelling for use case and communication diagrams for various real time applications. In future we would like to apply B models for the other UML diagrams along with their refinement.

In second part we have shown that real numbers can be sorted in $O(\log^{1+\varepsilon} n)$ time and $O(\frac{nlogn}{\sqrt{loglogn}})$ operations on the EREW PRAM. Later we have optimized it further more and reduced the time complexity. This is the best algorithm till now based on parallel sorting on real numbers.

REFERENCES

[1] Jean-Raymond Abrial. 2010. *Modeling in Event-B:System and Software Engineering.* Cambridge University Press.

[2] J.R. Abrial. 1996. *The B Book. Assigning Programs to Meanings*. Cambridge University Press.

[3] J.R. Abrial Chapter 2 of the forthcoming book: Modeling In Event-B: System and Software Engineering Forthcoming book. http://www.event-b.org/A ch2.pdf.

[4] A. Ajtai, J. Komlós, and E. Szemerédi. 1983. An O(n log n) sorting network. In *Proceedings of the 15th Annual ACM Symposium on Theory of Computing* (STOC'83), 1-9. DOI:10.1145/800061.808726

[5] Eman Alkhammash, Michael Butler, Asieh Salehi Fathabadi, and Corina Cîrstea. 2015. Building traceable Event-B models from requirements. *Science of Computer Programming* 111, 318-338.

[6] James Rumbaugh, Ivar Jacobson, and Grady Booch. 1998. *The Unified Modeling Language reference Manual*. Addison- Wesley.

[7] P.C.P Bhatt, K. Diks, T. Hagerup, V.C. Prasad, T. Radzik, and S. Saxena. 1991. Improved deterministic parallel integer sorting. *Information and Computation*, 94, 1, 29-47.

[8] G.Booch, I.Jacobson and R.Johason. 1998. The Unified Modelling Language reference Manual. Addison-Wesley.

[9] R. Cole. 1988. Parallel merge sort. *SIAM Journal on Computing*, 17, 4, 770-785. Correction: Parallel merge sort. *SIAM Journal on Computing,* 22, 6, 1349 (1993).

[10] T.H. Corman, C.E. Leiserson, R.L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd. ed.) The MIT Press.

[11] Jim Conallen. 1999. Modeling web application architectures with UML. . Communications of the ACM 42, 10 (October 1999), 63-70. DOI:=http://dx.doi.org/10.1145/317665.317677.

[12] T. Hagerup. 1987. Towards optimal parallel bucket sorting. *Information and Computation*, 73, 39-51.

[13] Y. Han. 2017. Sort real numbers in O(n√logn) time and linear space. In arXiv.org

with paper id 1801.00776

[14] Y. Han. 2004. Deterministic sorting in O(nloglog n) time and linear space. J*ournal of Algorithms*, 50, 96-105.

[15] Y. Han. 2015. A linear time algorithm for ordered partition. In *Proceedings of the 2015 International Frontiers in Algorithmics Workshop* (FAW'15), LNCS 9130, 89-103.

[16] Y. Han, and X. Shen. 1995. Conservative algorithms for parallel and sequential integer sorting. In *Proceedings of 1995 International Computing and Combinatorics Conference*, Lecture Notes in Computer Science 959, 324-333.

[17] Y. Han, and X. Shen. 2002. Parallel integer sorting is more efficient than parallel comparison sorting on exclusive write PRAMs. In *Proceedings of Tenth Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'99), Baltimore, Maryland, 419-428(January 1999). Also in *SIAM Journal on Computing.* 31, 6, 1852-1878.

[18] https://en.wikipedia.org/wiki/Communication_diagram

[19] https://en.wikipedia.org/wiki/UML_tool

[20] https://en.wikipedia.org/wiki/Use_case_diagram

[21] https://www3.hhu.de/stups/handbook/rodin/current/html/index.html, 2.4 The First Machine – Traffic Light Controller

[22] C. P. Kruskal. 1983. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, C-32, 942-946.

[23] S. Saxena, P. Chandra, P. Bhatt, V.C. Prasad, 1994. On parallel prefix computation. *Parallel Processing Letters* 4, 429-436.

[24] L. G. Valiant. 1975. Parallelism in comparison problems. *SIAM Journal on Computing*, 4, 348-355.

VITA

Md Usman Gani Syed was born on May 04, 1994, in Andhra Pradesh, India. He completed his Bachelor's degree in Electronics and Communication Engineering from National Institute of Technology, Jamshedpur, India.

After completing his under-graduation, he worked as Assistant Systems Engineer in TCS, Hyderabad for 6 months and later worked as a Software Engineer in Mahindra Comviva, Bangalore for 2 years. To enhance his career further he started his masters in Computer Science at the University of Missouri-Kansas City (UMKC) in Spring 18, specializing in Software Engineering. Upon completion of his requirements for the Master's Program, Md Usman Gani Syed plans to work as a Software Developer.