PARALLEL ALGORITHMS FOR PUBLIC KEY INFRASTRUCTURE BASED SECURITY TECHNIQUES

THESIS

Submitted

in fulfilment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

By

Sapna Saxena

University Regd. No: PHDENG10022

Supervised by

Dr. Bhanu Kapoor,

Professor, Chitkara University, Himachal Pradesh

December, 2014

Department of Computer Science & Engineering

CHITKARA UNIVERSITY, HIMUDA EDUCATIONAL HUB, SOLAN, HIMACHAL PRADESH-174103
CHITKARA UNIVERSITY, HIMACHAL PRADESH

DECLARATION BY STUDENT

I hereby certify that the work which is being presented in this thesis entitled “Parallel Algorithms for Public Key infrastructure Based Security Techniques” is for fulfillment of the requirement for the award of Degree of Doctor of Philosophy submitted in the Department of Computer Science and Engineering, Chitkara University, Barotiwala, Solan, Himachal Pradesh is an authentic record of my own work carried out under the supervision of Dr. Bhanu Kapoor.

The work has not formed the basis for the award of any other degree or diploma, in this or any other Institution or University. In keeping with the ethical practice in reporting scientific information, due acknowledgements have been made wherever the findings of others have been cited.

(Signature)
(Sapna Saxena)
This is to certify that the thesis entitled “Parallel Algorithms for Public Key Infrastructure Based Security Techniques” by Ms Sapna Saxena (PHDENG10022) to the Chitkara University, Barotiwala, Solan, Himachal Pradesh in fulfillment for the award of the degree of Doctor of Philosophy is a bona fide record of research work carried out by her under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institution or University for the award of any degree or diploma.

(Signature)

Dr. Bhanu Kapoor
Professor
Chitkara University
Himachal Pradesh, India
ACKNOWLEDGEMENT

It is not possible to thank enough everyone who has been helped me during this PhD research, and I can make only a rough attempt to express my gratitude.

First of all, I would like to thanks my supervisor Dr Bhanu Kapoor for his valuable guidance throughout this PhD thesis. His continued interest, support and encouragement were helpful during the thesis.

I would also like to take this opportunity to thank Dr. Ashok K. Chitkara, Chancellor, Chitkara University, Dr. Madhu Chitkara, Pro Chancellor, Chitkara University, Brig. (Dr.) R. S. Grewal Vice Chancellor, Chitkara University, Dr. Varinder Kanwar, Registrar, Chitkara University, Dr. Rajnish Sharma, Dean Academics, Chitkara University and Dr. Sudhir Mahajan, Dean R&D, Chitkara University who motivated me by always being interested in the progress of my research.

I extend my sincere thanks to one and all of internal and external examiners who gave their valuable feedback and helped me to overcome my shortcomings throughout this research.

My deepest appreciation goes to my research colleagues Ms Tanu Sharma, Ms Neha Kishore and Ms Disha Handa who supported me at every point of time whenever I needed them.

Besides my colleagues, I would like to thank my family and in particular my parents and my brothers for being there for me.

I would also like to express my gratitude to my little nieces who remained source of inspiration for me throughout this PhD thesis.

Finally, I would like to thanks Almighty to provide me strength to achieve this goal.

Sapna Saxena
LIST OF PUBLICATIONS

Published / Presented


- KISHORE, N., KAPOOR, B., SAXENA, S. & HANADA, D. Parallel implementation of redesigned SHA-1 on GPUs. *nVidia GPU Technology Conference 2013*, San Jose, California, US.

Accepted


**Communicated**

## ABBREVIATIONS

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advanced Encryption Standard</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>DNA</td>
<td>Deoxyribonucleic Acid</td>
</tr>
<tr>
<td>DSA</td>
<td>Digital Signature Algorithm</td>
</tr>
<tr>
<td>DSS</td>
<td>Digital Signature Standard</td>
</tr>
<tr>
<td>DVFS</td>
<td>Dynamic Voltage and Frequency Scaling</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-Programmable Gate Array</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU's Compiler Collection</td>
</tr>
<tr>
<td>GNU MP</td>
<td>GNU's Multi precision Library</td>
</tr>
<tr>
<td>GPGPU</td>
<td>General Purpose Graphics Processing Unit</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuit</td>
</tr>
<tr>
<td>MD5</td>
<td>Message Digest 5</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction Stream and Multiple Data Stream</td>
</tr>
<tr>
<td>MISD</td>
<td>Multiple Instruction Stream Single Data Stream</td>
</tr>
<tr>
<td>MIT</td>
<td>Massachusetts Institute of Technology</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>OpenMP</td>
<td>Open Multiprocessing</td>
</tr>
<tr>
<td>PKC</td>
<td>Public Key Cryptography</td>
</tr>
<tr>
<td>PKCS</td>
<td>Public Key Cryptography Standard</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>RSA</td>
<td>Rivest Shamir Adleman</td>
</tr>
<tr>
<td>SHA-1</td>
<td>Secure Hash Algorithm - 1</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Stream Multiple Data Stream</td>
</tr>
<tr>
<td>SISD</td>
<td>Single Instruction Stream Single Data Stream</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multiprocessor Architecture</td>
</tr>
<tr>
<td>SSL</td>
<td>Secure Socket Layer</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
</tbody>
</table>
## NOTATIONS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E_{EK}$</td>
<td>Encryption Algorithm using Encryption Key EK</td>
</tr>
<tr>
<td>$E_{DK}$</td>
<td>Decryption Algorithm using Decryption Key DK</td>
</tr>
<tr>
<td>M</td>
<td>Plain Text</td>
</tr>
<tr>
<td>C</td>
<td>Cipher Text</td>
</tr>
<tr>
<td>$H(m)$</td>
<td>hash value of the message m</td>
</tr>
<tr>
<td>S</td>
<td>signature generation with the private key a</td>
</tr>
<tr>
<td>V</td>
<td>signature verification with the public key A</td>
</tr>
<tr>
<td>p</td>
<td>prime number</td>
</tr>
<tr>
<td>q</td>
<td>prime number</td>
</tr>
<tr>
<td>n</td>
<td>$p \cdot q$</td>
</tr>
<tr>
<td>$T_s$</td>
<td>Serial Time</td>
</tr>
<tr>
<td>$T_p$</td>
<td>Parallel Time</td>
</tr>
<tr>
<td>$T_o$</td>
<td>Total Overhead</td>
</tr>
<tr>
<td>E</td>
<td>Efficiency</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table No.</th>
<th>Title</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1</td>
<td>Comparison of proposed parallel approaches for RSA</td>
<td>21</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Calculation based on Naïve Method</td>
<td>54</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Example of Calculations in Right-to-left binary method</td>
<td>62</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Configuration of the Computer System used for Experiments</td>
<td>63</td>
</tr>
<tr>
<td>Table 3.4</td>
<td>Comparative results of three forms of RSA</td>
<td>64</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Configuration of Computer used for Experiments</td>
<td>76</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Execution Time of Encryption Test case set 1</td>
<td>77</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Execution Time of Decryption Test case set 1</td>
<td>78</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>Execution Time of Encryption Test case set 2</td>
<td>80</td>
</tr>
<tr>
<td>Table 4.5</td>
<td>Execution Time of Decryption Test case set 2</td>
<td>81</td>
</tr>
<tr>
<td>Table 4.6</td>
<td>Speedup obtained for Test Case Set 1</td>
<td>83</td>
</tr>
<tr>
<td>Table 4.7</td>
<td>Speedup obtained for Test Case Set 2</td>
<td>84</td>
</tr>
<tr>
<td>Table 4.8</td>
<td>Efficiency of Encryption for Test Case Set 1</td>
<td>86</td>
</tr>
<tr>
<td>Table 4.9</td>
<td>Efficiency of Decryption for Test Case Set 1</td>
<td>86</td>
</tr>
<tr>
<td>Table 4.10</td>
<td>Efficiency of Encryption for Test Case Set 2</td>
<td>87</td>
</tr>
<tr>
<td>Table 4.11</td>
<td>Efficiency of Decryption for Test Case Set 2</td>
<td>87</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Configuration of Computer used for Experiments</td>
<td>104</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Encryption Time for Test Case Set 1</td>
<td>105</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>Decryption Time for Test Case Set 1</td>
<td>106</td>
</tr>
</tbody>
</table>
Table 5.4 Encryption Time for Test Case Set 2..........................108
Table 5.5 Decryption Time for Test Case Set 2..........................109
Table 5.6 Encryption Time for Test case Set 3..........................110
Table 5.7 Decryption Time for Test Case Set 3..........................111
Table 5.8 Speedup obtained for Test Case Set 1..........................114
Table 5.9 Speedup obtained for Test Case Set 2..........................114
Table 5.10 Efficiency of Encryption for Test Case Set 1...............116
Table 5.11 Efficiency of Decryption for Test Case Set 1...............116
Table 5.12 Efficiency of Encryption for Test Case Set 2...............117
Table 5.13 Efficiency of Decryption for Test Case Set 2...............117
Table 5.14 Efficiency of Encryption for Test Case Set 3...............117
Table 5.15 Efficiency of Decryption for Test Case Set 3...............118
Table 5.16 Comparison of Encryption Execution time between PRSA1 and PRSA2..........................................................121
Table 5.17 Different versions of Parallel RSA proposed worldwide...122
Table 5.18 Comparison between PRSA2 and BEA1RSA by Q.Liu et al..123
Table 5.17 Comparison between PRSA2 and Belecki and Burak RSA...124
Table 6.1 Configuration of Computer used for Experiments.............140
Table 6.2 Execution time for the Digital Signature Generation...........141
Table 6.3 Execution time for the Digital Signature Verification ...........142
Table 6.4 Speedup obtained of Signature Generation on 8 core machine..143
Table 6.5 Efficiency of Signature Generation in PRSADSA.............145
Table 6.6 Efficiency of Signature Verification in PRSADSA…………………145
Table 6.7 Computations using Coarse Grained Granularity for PRSADSA…..147
Table 7.1 Configuration of Laptop used in Experiments………………………158
Table 7.2 Energy Consumption recorded on Laptop before Calibration………158
Table 7.3 Energy Consumption recorded on Laptop after Calibration……….159
Table 7.4 Configuration of Desktop used in Experiments……………………160
Table 7.2 Energy Consumption recorded on Desktop before Calibration……..161
Table 7.3 Energy Consumption recorded on Desktop after Calibration……….161
<table>
<thead>
<tr>
<th>Figure No.</th>
<th>Title</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 1.1</td>
<td>Encryption and Decryption</td>
<td>3</td>
</tr>
<tr>
<td>Fig. 1.2</td>
<td>Symmetric Cryptography</td>
<td>6</td>
</tr>
<tr>
<td>Fig. 1.3</td>
<td>Public Key Cryptography</td>
<td>6</td>
</tr>
<tr>
<td>Fig. 1.4</td>
<td>Hashing Mechanism</td>
<td>7</td>
</tr>
<tr>
<td>Fig. 1.5</td>
<td>PKC Mechanism</td>
<td>10</td>
</tr>
<tr>
<td>Fig. 1.6</td>
<td>Parallel Programming</td>
<td>16</td>
</tr>
<tr>
<td>Fig. 1.7</td>
<td>GPGPU</td>
<td>18</td>
</tr>
<tr>
<td>Fig. 2.1</td>
<td>Task Parallelism</td>
<td>27</td>
</tr>
<tr>
<td>Fig. 2.2</td>
<td>Data Parallelism</td>
<td>28</td>
</tr>
<tr>
<td>Fig. 2.3</td>
<td>SISD Architecture</td>
<td>30</td>
</tr>
<tr>
<td>Fig. 2.4</td>
<td>SIMD Architecture</td>
<td>30</td>
</tr>
<tr>
<td>Fig. 2.5</td>
<td>MISD Architecture</td>
<td>31</td>
</tr>
<tr>
<td>Fig. 2.6</td>
<td>MIMD Architecture</td>
<td>31</td>
</tr>
<tr>
<td>Fig. 2.7</td>
<td>Threads Model</td>
<td>32</td>
</tr>
<tr>
<td>Fig. 2.8</td>
<td>Distributed Memory Model</td>
<td>33</td>
</tr>
<tr>
<td>Fig. 2.9</td>
<td>Data Parallel Model</td>
<td>33</td>
</tr>
<tr>
<td>Fig. 2.10</td>
<td>Fork-Join Model of OpenMP</td>
<td>42</td>
</tr>
<tr>
<td>Fig. 3.1</td>
<td>RSA Encryption</td>
<td>50</td>
</tr>
<tr>
<td>Fig. 3.2</td>
<td>RSA Decryption</td>
<td>51</td>
</tr>
</tbody>
</table>
Fig. 3.3 Recursive definition of modular exponentiation by squaring…55
Fig. 3.4 Graphical Representation of Time of various Implementations…64
Fig. 4.1 Modular Exponentiation for Even Keys…71
Fig. 4.2 Modular Exponentiation for Odd Keys…71
Fig. 4.3 PRSA1 Methodology…73
Fig. 4.4 Execution time of Encryption on 2, 4, 6 and 8 Cores (Test case set 1)…78
Fig. 4.5 Execution time of Decryption on 2, 4, 6 and 8 Cores (Test case set 1)…79
Fig. 4.6 Execution time of Encryption on 2, 4, 6 and 8 Cores (Test case set 2)…80
Fig. 4.7 Execution time of Decryption on 2, 4, 6 and 8 Cores (Test case set 2)…81
Fig. 4.8 Efficiencies of PRSA1 on 8 Cores for increasing problem size…88
Fig. 4.9 Computations using Coarse Grained Granularity for PRSA1…90
Fig. 5.1 Mechanism of PRSA2…95
Fig. 5.2 Encryption Execution time on 2, 4, 6 and 8 cores for Test case set 1…106
Fig. 5.3 Decryption Execution time on 2, 4, 6 and 8 cores for Test case set 1…107
Fig. 5.4 Encryption Execution time on 2, 4, 6 and 8 cores for Test case set 2…108
Fig. 5.5 Decryption Execution time on 2, 4, 6 and 8 cores for Test case set 2…109
Fig. 5.6 Encryption Execution time on 2, 4, 6 and 8 cores for Test case set 3…110
Fig. 5.7 Decryption Execution time on 2, 4, 6 and 8 cores for Test case set 3…111
Fig. 6.1 Digital Signing Mechanism…132
Fig. 6.2 Digital signature Verification Mechanism…132
Fig. 6.3 Parallel Digital Signing Process…134
Fig. 6.4 Parallel Digital Signature Verification Process…137
Fig. 6.5 Execution time of Signature Generation on 2, 4, 6 and 8 cores……141

Fig. 6.6 Execution time of Signature Verification on 2, 4, 6 and 8 cores……142

Fig. 7.1 Joulemeter Window…………………………………………………………155

Fig. 7.2 Calibration in Joulemeter…………………………………………………..156

Fig. 7.3 Example of Excel sheet created by Joulemeter………………….157
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm No.</th>
<th>Title</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 3.1</td>
<td>Right-to-left binary modular exponentiation</td>
<td>56</td>
</tr>
<tr>
<td>Algorithm 3.2</td>
<td>Naïve modular reduction algorithm</td>
<td>56</td>
</tr>
<tr>
<td>Algorithm 3.3</td>
<td>Parallel RSA based on Repeated Square-and-multiply method</td>
<td>60</td>
</tr>
<tr>
<td>Algorithm 3.4</td>
<td>Right-to-left binary modular exponentiation algorithm</td>
<td>61</td>
</tr>
<tr>
<td>Algorithm 4.1</td>
<td>Proposed PRSA1 algorithm</td>
<td>72</td>
</tr>
<tr>
<td>Algorithm 5.1</td>
<td>PRSA2 encryption algorithm</td>
<td>97</td>
</tr>
<tr>
<td>Algorithm 5.2</td>
<td>PRSA2 decryption algorithm</td>
<td>100</td>
</tr>
<tr>
<td>Algorithm 6.1</td>
<td>PRSADSA-Digital Signature generation algorithm</td>
<td>135</td>
</tr>
<tr>
<td>Algorithm 6.2</td>
<td>PRSADSA-Signature verification algorithm</td>
<td>138</td>
</tr>
</tbody>
</table>
# Table of Contents

List of Publications ............................................................................................... v
Abbreviations ........................................................................................................ vii
Notations ................................................................................................................ ix
List of Tables .......................................................................................................... x
List of Figures ......................................................................................................... xiii
List of Algorithms .................................................................................................. xvi
Abstract .................................................................................................................. xxiii

Chapter 1 INTRODUCTION ................................................................................... 2
1.1 Cryptography ..................................................................................................... 2
   1.1.1 Encryption and Decryption ......................................................................... 3
   1.1.2 Conventional Cryptography ...................................................................... 4
   1.1.3 Mechanism of Cryptography .................................................................... 4
   1.1.4 Secret key or Symmetric Cryptography .................................................... 5
   1.1.5 Public-Key or Asymmetric Cryptography .................................................. 6
   1.1.6 Hashing or Hash Functions ....................................................................... 7
1.2 Public Key Cryptography ................................................................................... 8
   1.2.1 PKC Mechanism ....................................................................................... 10
   1.2.2 Key Length and PKC Strength ................................................................ 11
   1.2.3 PKC Algorithms ....................................................................................... 12
1.3 Issues with PKC Algorithms ............................................................................. 15
1.4 Possible Solutions to the Issues Related to PKC Methods ......................... 15
   1.4.1 Memory Efficiency Methods ................................................................... 15
   1.4.2 Parallel Programming .............................................................................. 16
   1.4.3 FPGA Implementation ............................................................................ 17
   1.4.4 VLSI Implementation ............................................................................. 17
   1.4.5 GPGPU Implementation ......................................................................... 17
1.5 Motivation ......................................................................................................... 18
1.6 Contribution ..................................................................................................... 19
1.7 Literature Review ............................................................................................ 20
1.8 Organization of the Thesis .............................................................................. 23
Chapter 2 PARALLEL PROGRAMMING TECHNIQUES AND TOOLS USED

2.1 Introduction ........................................................................................................... 26

2.2 Parallel Programming - Decomposition ............................................................... 27
   2.2.1 Task Parallelism ............................................................................................... 27
   2.2.2 Data Parallelism ............................................................................................... 28

2.3 Flynn’s Taxonomy .................................................................................................... 29
   2.3.1 SISD Architecture ............................................................................................ 29
   2.3.2 SIMD Architecture ......................................................................................... 29
   2.3.3 MISD Architecture ......................................................................................... 31
   2.3.4 MIMD Architecture ......................................................................................... 32

2.4 Parallel Programming Models .............................................................................. 32
   2.4.1 Threads Model .................................................................................................. 32
   2.4.2 Distributed Memory Model ............................................................................. 33
   2.4.3 Data Parallel Model ......................................................................................... 34
   2.4.4 Hybrid Model .................................................................................................. 34

2.5 Decomposition Techniques ................................................................................. 34
   2.5.1 Recursive Decomposition ............................................................................... 34
   2.5.2 Data Decomposition ....................................................................................... 34
   2.5.3 Exploratory Decomposition ............................................................................ 35
   2.5.4 Speculative Decomposition ......................................................................... 35

2.6 Tasks ..................................................................................................................... 35

2.7 Granularity ............................................................................................................ 36

2.8 Mapping Techniques ........................................................................................... 36
   2.8.1 Static Mapping ............................................................................................... 36
   2.8.2 Dynamic Mapping ............................................................................................ 36

2.9 Memory Allocation ............................................................................................... 37

2.10 Scalability ............................................................................................................. 37

2.11 Performance Metrics for the Parallel Program .................................................. 37
   2.11.1 Execution Time .............................................................................................. 38
   2.11.2 Total Parallel Overhead ................................................................................ 38
   2.11.3 Speedup ........................................................................................................ 38
2.1.12 Tools Used for the Parallelization
2.1.12.1 Linux Platform
2.1.12.2 GCC Infrastructure
2.1.12.3 Faster Parallelization Using OpenMP
2.1.12.4 Fast Modular Computations Using GNUs MP Library
2.13 Conclusion

Chapter 3 PRSA1 - PARALLEL RSA USING EFFICIENT MEMORY

TECHNIQUES

3.1 Introduction
3.2 RSA Algorithm
3.2.1 Key Generation Algorithm
3.2.2 RSA Encryption
3.2.3 RSA Decryption
3.2.4 Example of RSA
3.3 Memory Efficiency Methods
3.3.1 Modular exponentiation
3.3.2 Modular reduction
3.4 First Form: Sequential Implementation of RSA Algorithm
3.4.1 Key Generation
3.4.2 RSA Encryption
3.4.3 RSA Decryption
3.5 Second Form: Parallel RSA based on Repeated-Square and Multiply Method
3.5.1 For Even Exponent
3.5.2 For Odd Exponent
3.6 Third form: Parallel RSA based on Right-to-Left Binary Method
3.7 Methodology
3.8 Platform used for Experiments
3.9 Results and Discussion ............................................................................. 64
3.10 Conclusions ............................................................................................. 65

Chapter 4 PRSA 1 – PARALLEL RSA ALGORITHM BASED ON REPEATED
SQUARE-AND-MULTIPLY METHOD ....................................................... 67
4.1 Introduction ............................................................................................... 67
4.2 PRSA 1 – The Parallel RSA Algorithm ................................................... 69
  4.1.1 PRSA1 for Even Key ........................................................................... 70
  4.1.2 PRSA1 for Odd Exponent .................................................................. 71
4.3 Methodology .............................................................................................. 73
4.4 Platform used for Experiments ................................................................. 76
4.5 Experimental Results ............................................................................... 76
  4.5.1 Test case Set 1 ..................................................................................... 76
  4.5.2 Test case Set 2 ..................................................................................... 79
  4.6.4 Performance Analysis ......................................................................... 82
  4.5.3 Speedup ............................................................................................... 82
  4.5.4 Total Overhead ................................................................................... 84
  4.5.5 Efficiency and Scalability ................................................................. 85
  4.6.5 Complexity ......................................................................................... 88
  4.6.6 Cost Optimality .................................................................................. 90
4.7 Conclusion .................................................................................................. 91

Chapter 5 PRSA 2 – PARALLEL RSA ALGORITHM BASED ON
SYMMETRIC MULTIPROCESSOR ARCHITECTURE .................................. 93
5.1 Introduction ............................................................................................... 93
5.2 PRSA 2 - Parallel RSA Algorithm .......................................................... 94
  5.2.1 Key Generation ................................................................................... 96
  5.2.2 Parallel RSA Encryption ................................................................... 96
  5.2.3 Parallel RSA Decryption ................................................................... 99
5.3 Methodology .............................................................................................. 101
5.4 Techniques used for the Parallelization .................................................. 102
  5.4.1 Data Decomposition Technique ....................................................... 102
  5.4.2 Granularity ......................................................................................... 103
  5.4.3 Static Mapping .................................................................................... 103
ABSTRACT

Cryptographic algorithms are used extensively ensuring the security of data when it’s static as well as on the un-trustworthy communication channels. A cryptographic algorithm is the mathematical function used for the encryption and the decryption of data. There are two types of key based data encryption algorithms: symmetric and public Key infrastructure. Symmetric algorithms are those where encryption key can be calculated from decryption key and vice versa. Public key cryptographic algorithms use a pair of two distinct keys – public key and private key that are used for encryption and decryption or for signing and verification. The main property of public key infrastructure based algorithms is that practically it is impossible to calculate the decryption key from a given encryption key.

RSA and RSA-based digital signature algorithms are some of the most popularly used public-key infrastructure cryptographic algorithms with their roots in the modular arithmetic. To strengthen the security of these algorithms, larger key sizes are used typically 1024 bit or more, and that makes these algorithms compute-intensive taking longer time during encryption - decryption and signing - verification modules of secure applications.

Since the public key based cryptographic algorithms are compute-intensive, their sequential implementations take a lot of time and energy to execute. Parallelization of these security algorithms in order to distribute the complex computational part among the various cores available with the processors today, will achieve higher performance and also be more energy efficient.

The objective of this research is to study and analyze various public key infrastructure based cryptographic algorithms and design new parallel algorithms to implement the public-key algorithms such as the RSA and the Digital Signature Algorithm. This research work has been divided into four main parts – (i) to convert memory-efficient algorithms into new parallel algorithms, (ii) to use important parallel programming techniques to obtain the new high
performance parallel variants of these algorithms, (iii) to use OpenMP API to test the algorithms and analyze performance gained by parallelizing the security algorithms through experiments over large number of data sets and, (iv) to use parallelism to demonstrate more energy-efficient algorithms applicable to portable and mobile devices.

During this research work, three parallel algorithms have been developed – PRSA1, PRSA2 and PRSADSA. The PRSA1 and the PRSA2 algorithms are the parallel variants of the RSA algorithm and the PRSADSA is the parallel variant of RSA-based digital signature algorithm. The PRSA1 has shown a speedup of up to 5x approximately, the PRSA2 has shown the speedup of up to 7.5x speedup approximately and the PRSADSA has shown a speedup of up to 5x approximately as compared to their respective sequential counterparts, when executed on 8-core machine. All of the three parallel variants were found to be memory-efficient, time-efficient, scalable and cost-optimal during the experimentation.

Moreover, all of the three parallel variants are energy-efficient also. The energy consumption of all parallel variants has been measured using Joulemeter. It was observed during experimentation that they consume less energy as compared to sequential counterparts at the same performance levels. Hence these parallel variants are more suitable to be used on battery-operated mobile devices.
CHAPTER 1

INTRODUCTION
Chapter 1

INTRODUCTION

The day-to-day activities in the modern era are dominated by the Internet. Plenty of sensitive information is shared among the people every day and all of this information is transmitted over untrusted communication media. Here the communication media is described as untrusted because lots of intruders and attackers are always active on the Internet that can misuse the critical information (William and Stallings, 2006). With the growth of electronic business on the Internet, the situation is becoming even more complex because of the flow of the capital on Internet. In this scenario, security and privacy are becoming the major concerns for the designers of all Internet-based applications. It is the foremost requirement of today’s Internet dominated world to communicate the data securely across open transmission media to avoid the misuse of transmitted data. Also, the integrity of information has to be maintained during transmission so that it is not tampered or modified by an intruder (Bidzos, 1991). Therefore, to ensure the security of sensitive information, there has to be some techniques that can be trusted by the community (Bidzos, 1991). One such technique is cryptography (Paar and Pelzl, 2009) that can be used to provide ample security to the information it is being transmitted (Stinson, 2005).

1.1 Cryptography

Cryptography is the branch of network security where the art of “keeping message secured” is studied. It is the art of disguising the message in such a manner that only intended receiver can read and understand its substance (Menezes et al., 1996). The message is disguised by jumbling up the data or by substituting it with some other text using certain mathematical techniques. Cryptography facilitates the parties involved in communication to store crucial information intact or keeping it secure while transmitting it on insecure networks, like the Internet, so that it
cannot be read or intercepted by any intruder except the intended recipient (Schneier, 2007).

Cryptanalysis is another branch of network security where cryptanalysts work on finding the loopholes in the security of these techniques. It is the science of analyzing the encrypted text and finding the ways to break its security (William and Stallings, 2006). Cryptanalysis involves various techniques simultaneously, for example, using mathematical tools, reasoning, pattern matching and finding, etc. to find the actual information involved in the communication (Biham and Shamir, 1991). Cryptanalysts can also be described as attackers for this reason. (Menezes et al., 1996)

1.1.1 Encryption and Decryption

The data or information that is readable and understandable without applying any special measures to it is called the “plain text” (Menezes et al., 1996). Whereas, when the message disguised using some mathematical technique is called “cipher text” (Menezes et al., 1996).

Cryptography is the combination of two distinct techniques – Encryption and Decryption (Mao, 2003). Encryption is a technique of converting the message or plaintext to cipher text to hide the original content. On the other hand Decryption is the process of converting the cipher text into its original form i.e. plaintext. The process of encryption / decryption is shown in Fig. 1.1.

![Diagram of Encryption and Decryption](image)

Fig. 1.1: Encryption and Decryption
1.1.2 Conventional Cryptography

Conventionally, cryptography is performed using two techniques – substitution and transposition (Mao, 2003). The example of one of the oldest substitution ciphers is Caesar Cipher.

1.1.2.1 Caesar Cipher

Caesar Cipher is the simplest example of substitution cipher (Menezes et al., 1996). A substitution cipher performs encryption by substituting one character of the information by other using some key value (William and Stallings, 2006). The same process is performed for all the characters present in the message.

For example, to encrypt the word “ENCRYPTION” using Caser Cipher with key value 4, then each character of the word is substituted by the 4th character down the alphabet. That is, the first character which is ‘E’, is substituted by the 4th character down the alphabet from ‘E’ which is ‘I’. Thus the word “ENCRYPTION” becomes “IRGVCTMSR” after encrypting with Caesar Cipher.

1.1.3 Mechanism of Cryptography

The encryption and decryption techniques are performed using two distinct mathematical techniques which are called cryptographic algorithms – Encryption Algorithm and Decryption Algorithm (Stinson, 2005). Encryption Algorithm is performed on plaintext in the presence of encryption key whereas the Decryption algorithm is performed on Cipher Text in the presence of decryption key. Usually the keys which are taken for these algorithms have very large size and the strength of encryption / decryption algorithms depends upon the secrecy of these keys. Therefore it can be said that the secrecy and integrity of the message is ensured by keeping the keys secured (Cook et al., 2005).

Plaintext may be any type of text such as a text file, an image, a video file or any other type of data. It is generally denoted by M (Message). Sender
encrypts the message and sends it to the receiver. In mathematical terms it can be denoted by the Eq. 1.1 (Schneier, 2007).

\[ E_{EK}(M) = C \]  

(1.1)

where E is the Encryption Algorithm, M is the message, C is the Cipher text and EK is the encryption key. C is usually the binary data having the size equal to the M or more than that.

In order to decrypt the cipher text receiver performs the reverse process. The receiver decrypts the cipher text with the help of decryption algorithm in the presence of decryption key and obtains plaint text. In mathematical terms it can be denoted by the Eq. 1.2 (Schneier, 2007).

\[ D_{DK}(C) = M \]  

(1.2)

where D is the Decryption Algorithm, C is the Cipher Text, M is the message, and DK is the decryption key. C is usually the binary data having the size equal to or more than the size of M.

To perform encryption and decryption, a variety of cryptographic algorithms are used. These can be classified into two distinct groups – Symmetric or Private Key Algorithms and Asymmetric or Public Key algorithms. Private Key algorithms use a single key to encrypt and decrypt the data which is called private key. In contrast to this Public key algorithms use two separate keys to perform encryption and decryption. The key used for encryption is called public key and the key used for decryption is called private key. For example, RSA is the one of the most popular public cryptographic algorithm which uses two keys.

Generally, there are three types of algorithms that are used to accomplish these goals, which are described in next three sub sections.

1.1.4 Secret key or Symmetric Cryptography

In the private key cryptography single key is used for encryption as well as decryption which is called secret key (Akl and Taylor, 1983).
Because single key is used for encryption as well as decryption it is also called as symmetric key cryptography (Simmons, 1979) as displayed in Fig. 1.2.

The security of the symmetric algorithm depends upon the keeping the secret key private (Simmons, 1979). If it gets revealed due to the public for some reason then the security of the algorithm is lost. The private key algorithms are of two types – Block Ciphers and Stream Ciphers (Handa and Kapoor, 2014). The block ciphers (Courtois and Pieprzyk, 2002) are performed on the fixed blocks of data whereas the stream ciphers are performed on the data - character by character. The examples of some Symmetric Ciphers are Data Encryption Standard (DES), Advanced Encryption Standard (AES) (Daemen and Rijmen, 2002), RC4 (Handa and Kapoor, 2014), Blowfish (Schneier, 1994), etc.

1.1.5 Public-Key or Asymmetric Cryptography

In this technique instead of single key, a pair of keys is used - one for encryption and other for decryption. This is why it is called Asymmetric Key Cryptography(Diffie and Hellman, 1976b, Paillier and Pointcheval, 1999).
The encryption is performed using the public key and decryption is performed using the private key as demonstrated in Fig 1.3. The security of these algorithms depends upon, as the name suggests, keeping the private key secured (Simmons, 1979). The examples of public key algorithms (Cramer and Shoup, 1998) include RSA, Digital Signature (Somani et al., 2010, Johnson et al., 2001) Algorithm, etc.

1.1.6 Hashing or Hash Functions

Hashing is a technique which is used to generate hash codes which are used to ensure the integrity of the messages (Kishore and Kapoor, 2014). The hash codes are first generated and then appended with the messages to ensure the integrity of the message as shown in Fig. 1.4.

Hashing is also performed to keep the message secured. For example, passwords are first hashed and then keep secured in the password database. Some examples of hash (Van Oorschot and Wiener, 1994) functions are SHA-1, MD5, etc (Kishore and Kapoor, 2014).

![Hashing Mechanism Diagram](image-url)

Fig. 1.4: Hashing Mechanism
1.2 Public Key Cryptography

Public-key cryptography (PKC) is one of the most remarkable new developments in cryptography in the last 3-4 centuries. Modern public-key cryptography was first presented in 1976 by the Martin Hellman professor of Stanford University and Whitfield Diffie, a graduate student. In their paper, they described a crypto system where instead on one key, two keys are used and these pair of keys can ensure that two parties can communicate securely over un-trusted communication channel (Diffie and Hellman, 1976b).

PKC (Fischer, 1991) methods are the used at many places, for example to create signed documents, encrypted emails, in Secure Sockets Layer (SSL) communications, etc (ElGamal, 1985). They are used to protect the messages from the mishaps (Boneh, 1999) described in next few sections.

- **Eavesdropping**

  In eavesdropping (Fenton, 2003) the information remains unchanged, but its privacy gets lost somehow. For example, the eavesdropper can obtain the credit card numbers, record a crucial exchange, or intercept classified information.

- **Tampering**

  Tampering (Sjöberg, 2000) is the process of changing or replacing the contents of the message during transit. Thus the receiver gets the incorrect information at the end. For example, some intruder can access order for goods and can change it.

- **Impersonation**

  Impersonation (Heckathorn et al., 2001) is a process of passing wrong person who present himself as a correct recipient. Impersonation can be of two types – Spoofing and Misrepresentation.

    - **Spoofing**

      In spoofing (Xia and Shuwang, 2003) an attacker presents himself as someone else. For example, an attacker can
represent email address which happens to be of some other person, etc.

- **Misrepresentation**
  In misrepresentation (Tenbrunsel, 1998), an attacker can misrepresent himself as some other. For example, a site can presents itself as an online store, receive online payment but never sends the goods.

PKC methods provides security against Internet based attacks (Kocher, 1996) using following security measures –

- **Encryption and decryption** (Abdalla et al., 1998)
  These techniques allow two parties to communicate on un-trusted media by disguising the message they shares between them. The sender disguises (encrypts) the message before sending it to receiver. On the other hand the receiver un-disguises the message after receiving it. Thus, during transmission the message becomes meaningless and insignificant to an intruder.

- **Tamper detection**
  It is the process of identifying and verifying whether the message has been modified during transit or not. If an intruder attempts to do that, these techniques help to detect them.

- **Authentication**
  It is the technique of determining that the received message has been send by the intended sender.

- **Non-repudiation**
  It is the process of preventing the sender from claiming that the information was never sent.

Because the PKC algorithms use two different keys, they are also called Asymmetric Algorithms. Practically it is not possible to derive or extract one key from the other. But, because they are mathematically related, it is possible to design an algorithm. However it is so difficult to calculate one key from the other that it becomes practically impossible.
1.2.1 PKC Mechanism

PKC methods are important because they can be used for transmitting encryption keys or other data securely even when the parties have no opportunity to agree on a secret key in private (Diffie and Hellman, 1979). The encryption key is also called the public key and the decryption key the private key. The security provided by these ciphers is based on keeping the private key secret.

In the pair of the keys described by PKC algorithm, the public key is advertised in the combination with the modulus, another parameter of the algorithm, to the public. The private key is kept secret and not revealed to anyone and only the receiver of the information has it (Wu et al., 2006). The mechanism behind the PKC is as follows –

1. If receiver wants some crucial message from the sender, he generates the pair of keys, private and public key and shares the public key with the sender. He keeps the private key with himself only.
2. The sender encrypts the message with the public key and sends the message to the receiver.
3. The receiver receives the message, decrypts it with the corresponding private key and obtains the original message.

![Fig. 1.5: PKC Mechanism](image-url)
The mechanism of PKC is shown in Fig. 1.5.

Public-key cryptography (Salomaa, 1996) depends upon some one-way mathematical functions that are easy to compute whereas their inverse function is somewhat difficult to compute. These functions and their inverse functions are described in next two sections.

1.2.2 Key Length and PKC Strength

The PKC method can be broken only if the key involved in the decryption process is known. The strength of the PKC methods depends upon the length of the keys used during the encryption / decryption process. The strength of the PKC method can be ensured if the length of the keys taken is sufficiently large (Garfinkel, 1996). For example if the key size is 1024 bits (minimum) then it is assumed that it is impractical to break the security of the PKC method.

On the other hand the strength of PKC method is also ensured by the one-way mathematical functions used in the algorithms (Mohapatra, 2000). In the next two sections, two such functional pairs of mathematical functions are discussed that are incorporated in the algorithms based on PKC. These are multiplication and exponentiation functions which are relatively easy than their reverse functions of factorization (Yeh et al., 2009) and logarithms, which are very difficult to implement. Therefore, the mathematical complexity behind the PKC algorithms’ one way functions is that it is not possible to find loopholes in the security as the reverse functions are very difficult to implement.

The general PKC-based algorithms use two mathematically related keys but it is very difficult to deduce one key from the other one. The one key of the pair is used to encrypt the text and other one is used to decrypt the text in any sequence but it is mandatory to use both the keys.
1.2.2.1 Multiplication and Factorization

Multiplication is a process where two numbers are multiplied to get the product of the given numbers whereas the factorization is the process of finding the integers whose product is the given integer (Hwang, 1979). Multiplication is a very simple process where the product of the numbers can found very quickly. But factorization takes much longer time and simple algorithms for factorization haven’t been developed yet.

1.2.2.2 Exponentiation and Logarithms

Exponentiation is a process, where given the two integers x and y, the value of y raised to the power x is calculated. On the other hand, the logarithms is the process of finding the two integers x and y that can be used to calculate the number z by raising y to the power x. Here again, as in the previous section, exponentiation is a simple process which can be carried out quickly whereas the reverse process of finding the logarithm takes much longer time to execute (Hwang, 1979).

1.2.3 PKC Algorithms

Since 1976 various PKC algorithms have been proposed by various scientists throughout the world. Although some of them are considered to be insecure, some of them are secure but impractical. Only some of them are considered to be both practical and secure. Some of them are listed below –

• RSA Encryption algorithm (PKCS#1)

The RSA algorithm (Rivest et al., 1978) is a public key infrastructure based algorithm and was introduced in the paper “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems” in 1977 published by Ron Rivest, Adi Shamir and Len Adleman. It is named after the initials of their last names. It is an algorithm for public-key cryptography which was one of the great advances in the field of public key cryptography. RSA was proposed as the first algorithm suitable for digital signing as well as for encryption. At present it is
included as the part of Lotus Notes, Quicken and many web browsers such as the Microsoft Internet Explorer and the Google Chrome. It is also extensively used in e-commerce communication protocols because it is believed to be sufficiently secure given the long keys.

RSA algorithm is an asymmetric algorithm because unlike its counterpart symmetric algorithms it uses two keys – a pair of private key and a public key. Public key is made public and shared among the users whereas the private key is kept private and not shared with anybody.

The security of RSA is based upon the factorization problem i.e. the fact that finding the factors of a given integer is difficult. This process takes very long time to execute even in the presence of best algorithms. Due to this fact, strong security can be provided to the data by using long keys. This fact provides the strong security to the data subject to with sufficiently long keys. For example, in RSA encryption, if key length is taken as 1024 bits or more then it becomes nearly impractical to break the security even on the high performance platforms.

- **Digital Signature Algorithm**

The United States Federal Government had proposed DSA (Rivest et al., 1978) in 1991 for the use of Digital Signatures (Kravitz, 1993, Merkle, 1989). It was proposed by the NIST (National Institute of Standards and Technology) in August 1991 for use in their Digital Signature Standard (DSS). The DSA can be used to check the integrity as well as the authenticity of the message by incorporating the digital signatures on the electronic documents. In DSA, the hashing algorithm is used to generate the hash code for the message which is further signed with the private key and associated with the message to ensure the authenticity of the message (Mohanty and Sarangi, 2010). On the other hand, receiver extracts the digital signature from the message and verifies its correctness. If it is found
to be correct that means the message is the authentic one and it is not tampered with during the transit.

Just as there are a number of algorithms for creating hash values, there are also a number of digital signature algorithms. Two of the most commonly used are RSA and DSA.

Digital Signature helps to ensure following security features –

- **Authentication**: The message is the one that was originally sent by the sender.
- **Non-Repudiation**: If the sender has sent a digitally signed message than he cannot deny that he is the original sender of the message.
- **Integrity**: The message is the original one and nobody has tampered or altered it during transmission.

**Diffie–Hellman Key Exchange Protocol**

Diffie–Hellman key exchange protocol (Diffie et al., 1992) is a PKC based method published by Whitfield Diffie and Martin Hellman in 1976. This method is used to exchange the secret key on the untrusted communication media. This method is generally used to share a key of any symmetric key. It allows the parties to share the secret key without having to know each other on the insecure communication channel.

There are some other Public Key Cryptography based algorithms which are relatively less popular. These are –

- ElGamal (Paillier, 1999)
- Paillier cryptosystem (Paillier, 1999)
- Cramer–Shoup cryptosystem (Cramer and Shoup, 2002)
1.3 Issues with PKC Algorithms

Usually, PKC algorithms are implemented using sequential algorithms working in a serial fashion on a single processor. The public key algorithms are based on modular arithmetic (series of modular exponentiations and modular reductions) and these modular computations are performed on very large integers to ensure the security. The sequential implementation of these compute-intensive algorithms – encryption and specially decryption (Zhang et al.) takes a lot of time to execute thus slowing down the process. These computations also reduce the performance of the machine being used, because of the large memory requirement. In addition to this, due to the compute intensive behavior these algorithms become inappropriate for battery operated devices as they require lot of power to execute themselves.

1.4 Possible Solutions to the Issues Related to PKC Methods

There may be many solutions to the above problems (Shand and Vuillemin, 1993). Scientists throughout the world trying to speed up the process of RSA encryption and decryption using various techniques such as including hardware implementations – FPGA (Oksuzoglu and Savas, 2008), VLSI, GPGPU implementations, software techniques – parallel programming (Pearson, 1996, Pieprzyk and Pointcheval, 2003, Pointcheval, 1999), memory efficiency methods etc. Some of these important solutions are described in the next sections.

1.4.1 Memory Efficiency Methods

The RSA encryption is based on modular arithmetic and it is a combination of series of modular exponentiation and modular reduction steps (Bewick, 1994). These computations are performed on very large integers and therefore they take very long time to execute. It is not possible for the single processor memory to accommodate these computations. One solution to these problems is memory efficient methods that are specially designed to accommodate the modular
computations performed on such large numbers in the main memory itself (Nozaki et al., 2001).

1.4.2 Parallel Programming

![Fig. 1.6: Parallel Programming](image)

Parallel programming (Quinn, 2003) is an emerging programming technique, that can be used to improve the efficiency and performance of the multi-core processing machines by the concurrent execution of the instructions provided by parallel program. To design a parallel program, the process is decomposed into multiple sections that can be executed concurrently on the multiple cores available with the parallel architecture. The overall workflow of a typical parallel program is shown in Fig. 1.6.

Therefore to achieve higher performance in the area of security, the security algorithms can be implemented in parallel such that they can be executed on multi core processor to increase the speed and efficiency. By parallelizing these algorithms the power consumption can also be reduced and high performance can be achieved in terms of energy as well.
1.4.3 FPGA Implementation

A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing – thus "field-programmable" (Ciet et al., 2003). The hardware description language (HDL) is used to specify FPGA configuration.

Modern FPGAs (Cilardo et al., 2004) have large resources of logic gates and RAM blocks to implement compute-intensive calculations. FPGAs can be used to implement any logical function that an application-specific integrated circuit (ASIC) could perform.

1.4.4 VLSI Implementation

Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip (Thapliyal and Srinivas, 2005). The technique of VLSI began in the 1970s when complex semiconductor and communication technologies were being developed. The microprocessor is an example of VLSI device. Before the introduction of VLSI technology most ICs had a limited set of functions they could perform. An electronic circuit might consist of a CPU, ROM, RAM and other glue logic. VLSI lets IC makers put together all of these on a single chip.

1.4.5 GPGPU Implementation

GPU CUDA is a parallel programming model presented by the company NVIDIA. It enables the programmers to increase the computation speed to the unbelievably larger extent by using the graphics processing unit (GPU) (Lin et al.).

Millions of software developers, scientists and researchers are using the power of GPU (Fleissner, 2007) with CUDA throughout the world. GPU in combination with CUDA can be used to solve diversified scientific applications. For example to identify hidden plaque in arteries, to analyze...
air traffic flow, to visualize molecules, etc. GPUs are designed as parallel architecture which enables the programmer to achieve high performance by porting the compute-intensive portions from CPU to GPU and after execution, by porting results from GPU to CPU.

![Fig. 1.7: GPGPU](Source: www.hpc.cineca.it)

1.5 Motivation

Security has always been a biggest concern for the computing world in terms of transmitting information and data across the networks. Security algorithms are usually implemented serially (that is, a program consisted of a sequence of instructions, where each instruction executed one after the other and it ran from start to finish on a single processor) which are bit slow as it takes time to perform calculations for encryption as well as decryption. It also requires large amount of memory which is sometimes not possible for a single processor.

Parallel programming (Quinn, 2003) is an emerging area developed as a means of improving performance and efficiency which uses multi core processor for the faster and efficient execution of the instructions. In a parallel program, the processing is broken up into parts, each of which can be executed concurrently. The instructions from each part run simultaneously on different CPUs. These CPUs can exist on a single
machine, or they can be CPUs in a set of computers connected via a network. Not only parallel programs are faster, they can also be used to solve problems on large datasets using non-local resources. Therefore to achieve higher performance in the area of security, the security algorithms can be implemented in parallel in such manner that after dividing them in specific parts they can be executed on multi core processor to increase the speed and efficiency of it.

Nowadays there is growing demand of mobile devices like mobile phones, laptops etc. and the biggest challenge is to provide security to the data which is exchanged during the communication among such devices. To provide security to the data it is obvious that we have to use any of the popular cryptography techniques and as the security algorithms are more compute intensive and are not power efficient they consume lot of power while executing themselves. By parallelizing these algorithms (Chiou, 1993) the power consumption can be reduced and high performance can be achieved.

It has also emerged in mainstream computers and high-end servers as a means to reach high performance targets while also maintaining acceptable power characteristics. So, it can be used for more energy efficient implementation of security algorithm making them more usable in areas like mobile computing, etc.

1.6 Contribution

In PKC based algorithms the greater part of the operation are based on modular arithmetic. In RSA or DSA algorithm the modular arithmetic based operations are the combination of the series of modular exponentiations and modular reductions. Modular exponentiations are usually performed by modular squaring and modular multiplications. In addition to this, these operations are performed on large integers because of the involvement of large keys. Due to the repeated modular calculations on large integers the PKC based algorithms become compute-
intensive and energy-intensive and takes lot of time and energy to execute.

During this research two new variants of the parallel RSA algorithms and one variant of parallel RSA-based digital signature algorithm has been designed and proposed in this thesis. The redesigned parallel algorithms have been designed in such manner that they should be capable of overcoming the problems mentioned in the last paragraph. The proposed parallel algorithms PRSA1, PRSA2 and PRSADSA are time-efficient, memory-efficient and energy-efficient. These are absolutely appropriate to be incorporated into the software developed for battery operated devices like mobile phones or tablets.

Moreover these algorithms has been implemented and tested on the parallel platform and the results have also discussed in the subsequent chapters.

1.7 Literature Review

As described earlier that RSA algorithm is a compute intensive algorithm that takes lot of time and energy to execute. But because of the importance of this algorithm, lot of researchers has proposed their parallel variants of RSA. These variants have been by implemented by using various hardware and software to increase its performance. In this section some important parallel implementations of RSA algorithm are presented based on the literature survey performed during the research. The table 1.1 describes the different variants proposed by the researchers in the area of public key cryptography along with their techniques and prominent results. Thereafter, the brief description of each variant is also explained.

In the paper “A full RNS implementation of RSA” (Bajard and Imbert, 2004) authors presented the hardware implementation of RSA cryptosystem. In their RSA variant they perform faster arithmetic using Residue Number System (RNS). They increased the speed of algorithm
by introducing Montgomery Multiplication Algorithm using RNS and shown the efficiency with two distinct implementations of RSA.

Table 1.1: Comparison of Proposed Parallel Approaches for RSA

<table>
<thead>
<tr>
<th>Author</th>
<th>Technique</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laurent Imbert and Jean-Claude Bajard in 2002</td>
<td>RNS implementation of RSA</td>
<td>High Throughput</td>
</tr>
<tr>
<td>Anshuman Rawat and Shabsi Walfish in 2003</td>
<td>Parallel Signcryption Standard using RSA with PSEP</td>
<td>Signcrypt small as well as long data</td>
</tr>
<tr>
<td>Mathieu Ciet et al in 2003</td>
<td>FPGA Implementation of RSA with Residue Number Systems</td>
<td>Execution of 1024-bit RSA in less than 150 ms</td>
</tr>
<tr>
<td>S.H. Tang et al in 2003</td>
<td>Modular exponentiation on RSA on a Xilinx XC2V3000-6 device</td>
<td>Faster Decryption</td>
</tr>
<tr>
<td>Chia-Long Wu et al in 2004</td>
<td>New algorithm faster than Savas-Tenca-Koc algorithm</td>
<td>Speedup of 1.06 to 2.75</td>
</tr>
<tr>
<td>Qiang Liu et al in 2004</td>
<td>VLSI implementation of the RSA</td>
<td>7% increase in the throughput</td>
</tr>
<tr>
<td>Weng-Long Chang et al in 2005</td>
<td>Three new DNA-based algorithms</td>
<td>Factoring two large prime numbers on DNA-based computer</td>
</tr>
<tr>
<td>Yu-Shiang Lin et al in 2006</td>
<td>Algorithm based on the factoring of RSA-64 integer on GPUs using CUDA</td>
<td>Speedup of 1197.5x as compared to any other CPU based algorithm</td>
</tr>
<tr>
<td>Hong Zhang et al in 2012</td>
<td>Parallel implementation of RSA on GPGPU</td>
<td>Speedup of 45x as compared to multi-core CPU</td>
</tr>
<tr>
<td>Masumeh Damrudi and Norafida Ithnin in 2012</td>
<td>Parallel RSA using Tree Structure</td>
<td>Fast Encryption</td>
</tr>
<tr>
<td>Sonam Mahajan and Maninder Singh in 2014</td>
<td>Parallel RSA on GPU using CUDA Framework</td>
<td>High speed up even when using large prime numbers</td>
</tr>
</tbody>
</table>

In the paper “Parallel Signcryption Standard using RSA with PSEP (Rawat and Walfish, 2003) authors presented a new standard “signcryption” which happens to be the combination of signing as well as
encryption. They implemented the signcryption using the Probabilistic Signature and Encryption Padding (PSEP) scheme which can be employed on small or large message. This implementation can also be used to RSA based key exchange protocols. But the drawback of this approach is that it is improper for the low bandwidth applications.

During the same year, in the paper “Parallel FPGA Implementation of RSA with Residue Number Systems - Can side-channel threats be avoided?” (Ciet, Neve et al. 2003) authors proposed the parallel variant of RSA to perform modular exponentiation at fast rate. They also implemented these computations using Montgomery multiplication algorithm which is based on Residue Number System. They synthesize their algorithm on FPGA and execute RSA1024 in less than 150 ms.

The authors of “Modular Exponentiation using Parallel Multipliers” (Tang et al., 2003), proposed an FPGA based variant for RSA. The authors presented semi-systolic implementation for the modular exponentiation algorithm. They performed 1024-bit RSA decryption in 0.66 ms with the Chinese Remainder Theorem using this architecture.

The Wu et al proposed an efficient parallel variant of RSA and during modular computations and they were able to get the speedup of 1.06 to 2.75 (Wu et al., 2006). They described that their variant is better than Savas-Tenca-Koc algorithm in time complexity.

The paper “A regular RSA processor” (Liu et al., 2004) was published in 2004. The authors proposed VLSI implementation of the RSA algorithm which happens to be the high performance variant based on the systolic array. At the end of their paper they presented the comparison between original modular multiplier architecture with their fanout bottleneck. Their proposed architecture was proved that it could provide 7% increase in the throughput without increase in area.

The most complex part is factoring of two larger primes while implementing the RSA algorithm. To overcome this problem Chang et al proposed three new variants based on DNA - parallel subtractor, parallel
comparator, and parallel modular arithmetic. They happened to be the first using DNA based computer to solve the problem of factoring two large prime numbers (Chang et al., 2005).

An efficient parallel RSA variant GPFA, which happens to be decryption algorithm was presented by Lin et al. It can be executed on GPUs using CUDA. They tested their algorithm for factoring RSA-64 integer and get the speedup of approximately 1197.5x as compared to CPU based algorithm. However the limitation of their algorithm was that they didn’t tested their algorithm for factoring RSA-128 or more.

In the paper “Comparison and Analysis of GPGPU and Parallel Computing on Multi-Core CPU” (Zhang et al.) authors described the fact again that the RSA algorithm has very complex computational part and it cannot be performed on CPU. Therefore they used GPGPU to develop a new variant. They performed the comparative analysis between the GPGPU variant and CPU variant and got the speedup of approximately 45x.

Damurdi and Ithnin applied parallel processing using tree structure to design new RSA variant. They proposed that by parallelizing RSA the speedup and the performance of RSA can be improved (Damrudi and Ithnin, 2013).

The GPU as a co-processor of CPU can be used to implement massive parallelism (Mahajan and Singh, 2014). They designed new parallel RSA variant using CUDA on GPU. Thereafter they tested their parallel variant for larger prime numbers. At the end, they proposed that their parallel RSA variant can be used to avoid the threats to security by using small prime numbers.

1.8 Organization of the Thesis

This thesis is divided into 7 chapters. Following this Introductory Chapter 1 is Chapter 2.

The Chapter 2 – “Parallel Programming Techniques and Tools Used” describes the fundamental parallel programming concepts that have been
used throughout the designing of parallel algorithms during this research. Apart from this, the chapter also describes the tools used during the parallel implementation of the algorithms designed during this research.

The Chapter 3 – “Parallel Implementation of RSA using Memory Efficiency Methods” describes the memory efficiency methods for the betterment of RSA algorithm, description of the parallel implementations of various forms, their methodology and results followed by the Conclusion.

The Chapter 4 – “PRSA1 based on Repeated Square-and-Multiply Method” describes the Design of Parallel RSA algorithm PRSA1, the methodology used in its parallel implementation, the results obtained during the testing of PRSA1 followed by conclusions.

The Chapter 5 – “PRSA2 based on Symmetric Multiprocessor Architecture” describes the Design of Parallel RSA algorithm PRSA2, the methodology of the parallel implementation, results of the various test cases involved in the experimentation. In addition to this the comparative analysis with other approaches proposed by other researchers is also presented in this chapter followed by the conclusions.

The Chapter 6 – “PRSADSA - The Parallel Digital Signature Algorithm” describes the design of the parallel RSA based Digital signature algorithm, the methodology involved in the parallel implementation of the PRSADSA algorithm, its results followed by the conclusions.

The Chapter 7 – “Energy Efficiency of Proposed Algorithms” describes the energy consumptions of proposed algorithms, proofs of energy efficiency of the algorithms by fine tuning the configuration of the Computer System involved in the research and conclusions.

The Chapter 8 – “Summary and Conclusions” describes the summary of the research based on the objectives, conclusion and future scope of the thesis.
CHAPTER 2
PARALLEL PROGRAMMING
TECHNIQUES AND TOOLS USED
Chapter 2

PARALLEL PROGRAMMING TECHNIQUES AND TOOLS USED

This chapter is based on the objective 1 of the proposed research. It describes the major parallel programming concepts in brief which are used during this research including parallel programming paradigms, parallel programming models, etc. It also describes major performance metrics which are used to evaluate the performance of a parallel algorithm. In addition to this the chapter also describes the tools used during this research.

2.1 Introduction

Parallel programming (Quinn, 2003) can be used to improve the efficiency and performance of the applications on multi-core processing machines through concurrent execution of instructions provided by parallel program. For the last 30-40 years, parallel programming has been used in the area of High Performance Computing (HPC). In this area, large complex problems are decomposed into the smaller parts, and then these parts are executed concurrently. Thereafter, the results obtained from all parts are combined to get the final result.

In other words, to design a parallel program (Butler and Lusk, 1994), the process is decomposed into multiple sections that can be executed concurrently on the multiple cores available with the parallel architecture. The overall workflow of a typical parallel program is shown in Fig. 1.6.

Therefore to achieve higher performance in the area of security, the security algorithms can be implemented in parallel (Gilles, 1974) such that they can be executed on multi core processor to increase the speed and efficiency. By parallelizing these algorithms the power consumption can also be reduced and high performance can be achieved in terms of energy as well.
2.2 Parallel Programming - Decomposition

The decision to make a program parallel requires as decision on how to decompose (or divide) it effectively. There are two broad ways to carry out this decomposition: Task and Data Parallelism.

2.2.1 Task Parallelism

In task level parallelism (Memik et al., 2001), the instructions to solve the complex problem are divided into separate tasks. These tasks are then mapped with the distinct processes which in turn are distributed among the various available cores of the processor. Then the cores execute these processes concurrently and the result of each core is recombined to get the final result.

For example, suppose on a dual core computer the user wish to listen the songs while playing the game. Using task level parallelism the CPU can run the media player to play the songs using one core while the other core can run the game. The working of Task Level Parallelism is demonstrated in Fig. 2.1.

Fig. 2.1: Task Parallelism
2.2.2 Data Parallelism

In data level parallelism (Memik et al., 2001, Snyder, 1988)), instead of instructions, the data to be executed is divided into fixed size chunks. Then these data chunks are assigned to the tasks which in turn assigned to the multiple cores of the CPU. In this scenario each core performs the same instructions but on the different set of data (Hillis and Steele Jr, 1986).

For example, suppose to solve some problem, 80 numbers have to be added. To perform this problem using data level parallelism on 8 core machine, the numbers can be divided into the 8 groups of 10 numbers each. Then each group can be assigned to each core to perform the addition concurrently. Thereafter the result of each core can be combined to get the overall sum of the 80 numbers.

Data parallelism also scales much better than task parallelism as the problem size increases. The Data Level Parallelism is demonstrated in Fig. 2.2.

Fig. 2.2: Data Parallelism
2.3 Flynn’s Taxonomy

Flynn’s classical taxonomy (Snyder, 1988) was introduced in 1966 to classify parallel computers. According to Flynn’s Classical Taxonomy the classification of parallel programming models is based on two independent dimensions – Data and Instructions. Each dimension can have a single or multiple state. Based on these dimensions the parallel computer’s architecture can be classified into following categories:

- Single Instruction Stream Single Data Stream (SISD)
- Single Instruction Stream Multiple Data Stream (SIMD)
- Multiple Instruction Stream Single Data Stream (MISD)
- Multiple Instruction Stream and Multiple Data Stream (MIMD)

The types of parallel architectures are described in next four sections.

2.3.1 SISD Architecture

In Single Instruction Stream Single Data Stream (SISD) (Wilkinson and Allen, 1999) model the single data stream is given as an input to the single processing core. Because the single input stream is provided to the single processing core therefore only single instruction stream can be executed by the processing core. The architecture is demonstrated in Fig 2.3. Some examples of SISD architecture are mainframes, minicomputers, single processor PCs.

2.3.2 SIMD Architecture

In this type of architecture all available processing core perform the same instruction stream on the multiple data stream. The architecture is demonstrated in the Fig 2.4. Some examples of the SIMD architecture (Wilkinson and Allen, 1999) are IBM 9000, Hitachi S820 and ETA10.
Fig. 2.3: SISD Architecture

Fig. 2.4: SIMD Architecture
2.3.3 MISD Architecture

In this architecture each available processing unit perform multiple instruction on a single data input. The MISD architecture (Wilkinson and Allen, 1999) is demonstrated in Fig. 2.5.

Fig. 2.5: MISD Architecture

In this architecture each available processing unit perform multiple instruction on a single data input. The MISD architecture (Wilkinson and Allen, 1999) is demonstrated in Fig. 2.5.

Fig. 2.6: MIMD Architecture
2.3.4 MIMD Architecture

In this architecture each available processing core performs different set of instruction on different data input. These instructions may be performed synchronously as well as asynchronously. The MIMD (Wilkinson and Allen, 1999) architecture is demonstrated in Fig. 2.6.

2.4 Parallel Programming Models

The parallel programming model is a technique of writing parallel programs in order to utilize all available processors or processing cores of a target computer system. Each programming problem cannot be designed using the same programming model; therefore, some parallel programming models are already proposed by various scientists. Some common models are described in the next four sections.

2.4.1 Threads Model

This model is also called the Shared Memory Model. In this model, a single heavy weight process is disintegrated into multiple light weight processes, which are called threads. Then these threads are allowed to execute concurrently in the shared memory area. The block diagram of this model is described in Fig. 2.7.

![Fig. 2.7: Threads Model](image-url)
2.4.2 Distributed Memory Model

The distributed memory model (Murdock, 1983) is designed especially for the devices connected through a network. In this a set of tasks are executed in the distributed memory area which can be part of any other device present on the network. In this model tasks communicate with each other by passing messages between themselves. This model is described in Fig. 2.8.

![Distributed Memory Model Diagram](image)

**Fig. 2.8: Distributed Memory Model**

![Data Parallel Model Diagram](image)

**Fig. 2.9: Data Parallel Model**
2.4.3 Data Parallel Model

In data parallel model (Culler et al., 1993) each the tasks are divided according to the basis on which same instruction set operates on different data set. The data set can be different data structure or can be a part of single data structure. This model is described in Fig. 2.9.

2.4.4 Hybrid Model

The hybrid model combines one or more than one programming model in the one model. For example the threads can be generated in the distributed memory architecture to utilize the memory available with other machines.

2.5 Decomposition Techniques

In order to execute any problem into parallel the first step is to decompose the problem into the set of relatively small problems that can be further converted into the tasks for the concurrent execution. These decomposition techniques that can be used to decompose the problem can be classified into four categories – Recursive, Data, Exploratory and Speculative Decomposition which are described in next four sections.

2.5.1 Recursive Decomposition

In recursive decomposition (Mattson et al., 2004) the problem is solved using the divide-and-conquer strategy. For this first the problem is first divided into the set of sub problems. Then these sub problems are solved concurrently and finally the result is combined to get the final result. For example Quick Sort algorithm can be solved using Recursive Decomposition Technique.

2.5.2 Data Decomposition

Data decomposition (Mattson et al., 2004) is the common method used for the decomposition of the parallel problems. This method is usually performed to decompose large data structures for the parallel
programming. This is performed in two steps. In first step the data is partitioned and in the second step the partitioned data is assigned to a task for the concurrent execution. For example this technique can be applied to Matrix Multiplication involving large matrices.

### 2.5.3 Exploratory Decomposition

The Exploratory decomposition (Mattson et al., 2004) technique is applied to those problems where search space has to be created. In the first step the problem is divided into smaller search spaces then the search is applied to these spaces concurrently. For example this technique can be applied to any Artificial Intelligence problem where it is required to create a search space.

### 2.5.4 Speculative Decomposition

The speculative decomposition technique (Mattson et al., 2004) is applied to those problems where the output of the current task is used to select the next appropriate branch of the programming flow from the available decision branches. During this scenario of parallel programming while one task is performing the computation whose result is going to be used in selecting the next computation, the other tasks can be allowed to run concurrently to perform next stage.

### 2.6 Tasks

After decomposing the problem into the set of small sub problems, each of these problems are further converted into tasks (Gropp et al., 1999). These tasks are executed concurrently to exploit the parallelism. Single parallel computation may or may not be consists of multiple tasks. A task encapsulates a sequential program and local memory. A task can perform four basic actions in addition to reading and writing its local memory - send messages, receive messages, generate new tasks, and terminate. Once the tasks are generated these are mapped to the processing units using any of the mapping techniques described in section 1.5.7.
2.7 Granularity

To exploit parallelism first the programming problem is decomposed into multiple independent sub problems which in turn converted into the tasks to be executed concurrently. The size of these tasks is represented as the granularity (Nickolls et al., 2008) of the parallelism. Granularity can be categorized into two categories – coarse grained and fine grained. In coarse grained granularity the problem is decomposed into relatively larger size of work to be assigned to the task. Due to this the task size become larger and will never create overhead for the parallel program. Whereas, when the problem is decomposed into relatively finer size of work to be assigned to the task then it is called fine grained granularity. Due to the small granularity the task size become smaller but it may or may not create the overhead for the parallel program.

2.8 Mapping Techniques

Once the problem is decomposed into the tasks these tasks are required to be allocated to the available processing cores of the target computer system. This process is called the Mapping. There are two mapping techniques available that can be used to map the tasks onto the processing units – Static Mapping and Dynamic Mapping. These techniques are described in next two sections.

2.8.1 Static Mapping

Static mapping (El-Rewini and Lewis, 1990) is often, though not exclusively, used in conjunction with a decomposition based on data partitioning. Static mapping is also used for mapping certain problems that are expressed naturally by a static task-dependency graph.

2.8.2 Dynamic Mapping

Dynamic mapping (El-Rewini and Lewis, 1990) is employed in the situations where static mapping may result in a highly imbalanced
distribution of work among processes. Since the primary reason for using a dynamic mapping is balancing the workload among processes, dynamic mapping is often referred to as dynamic load-balancing.

2.9 Memory Allocation

The memory for the parallel programs can be allocated to the variables using two types – local or global. The threads that require local variables are assigned the local copies of the variables that are usually defined in the cache memory for the faster execution. These variables are never shared among the threads. Whereas the variables that are required to be shared among the threads are defined in the shared memory area and all threads share the same copy of the variable residing in the shared memory.

2.10 Scalability

Scalability (Nickolls et al., 2008) of the parallel program is the ability to handle the growing number of the processing units in the target machine. Whenever the program is sent for the execution it adjusts itself according to the number of cores or processing units available in the target machine and start utilizing the power of all the cores (Chen and Schaumont, 2010).

2.11 Performance Metrics for the Parallel Program

Generally, a sequential algorithm is evaluated as a function of the size of its input which is expressed in terms of execution time. But the parallel algorithm cannot be expressed merely in terms of execution time because there is also one biggest factor involved into it that is number of processing elements involved in the parallel processing of a given algorithm. In addition to the number of processing elements the other factors which are also taken into consideration are the relative computation of the processing elements and the speed of inter process communication. Therefore, to analyze a parallel algorithm efficiently it is evaluated in the combination with parallel architecture.
There are several performance metrics (Quinn, 2003) available which are used to evaluate the strength and efficiency of the parallel algorithm. Some important performance metrics (Kumar et al., 1994) for the parallel algorithms are described in the next sections.

### 2.11.1 Execution Time

The execution time of a serial algorithm is the measurement of an elapsed time between the starting and the end of its execution on single processing element. In contrast to this, the parallel execution time is the measurement of the elapsed time between the starting and the end of the parallel computation of the given parallel algorithm. The execution time of the serial algorithm is denoted by $T_S$ and the parallel execution time is denoted by $T_P$.

### 2.11.2 Total Parallel Overhead

The total overhead of a parallel algorithm is expressed by the difference of the total time taken by all the processing elements collectively to solve the given problem in parallel and the time taken by the best known sequential algorithm to perform that problem sequentially. The total overhead of a parallel algorithm is denoted by $T_O$.

The total runtime of a parallel algorithm is computed by summing the time spent by all the processing elements which is denoted by $pT_P$. Suppose the $T_S$ is the time spent during the execution of the essential part of the work then the $T_O$ can be denoted by the Eq. 2.1.

$$T_o = pT_p - T_s$$  \hspace{1cm} (2.1)

### 2.11.3 Speedup

Speedup is one of the most important performance metrics while evaluating the performance of the parallel program. Speedup is a comparative measure of solving the problem in parallel as compared to sequential. It is calculated by taking the ratio of the execution time taken
by the serial algorithm on a single processing element with respect to the execution time taken by the parallel algorithm on a parallel computing platform having p number of identical processing elements. It is denoted by the symbol S. Speedup is calculated by taking the ratio of parallel runtime with respect to the execution runtime of the best known sequential algorithm.

Suppose in a given parallel program each step is the combination of one multiplication and one communication. Where the multiplication takes the time $T_m$ during execution and the communication of a single word takes time $T_s+T_w$. In such scenario it can be said that the multiplication and communication takes the constant amount of time during each step. Hence it can be expressed as per Eq. 2.2.

$$T_P = \Theta (\log n) \quad (2.2)$$

And since the problem can be solved in the $\Theta(n)$ time on a single processing element then the speedup can be expressed as per Eq. 2.3.

$$S = \Theta \left( \frac{n}{\log n} \right) \quad (2.3)$$

### 2.11.4 Efficiency

Theoretically, it is evident that only ideal parallel system can give the maximum speed up of P on a given parallel platform having p identical processing elements. But practically the ideal speedup is usually not achieved because of the obvious reasons. The 100% time of the processing elements cannot be utilized for the parallel execution because they may remain idle for some time due the inter process communication or some input output operations involved in the process. There is always some overheads are also involved in the execution. The efficiency is the measure of the part of time during which the processing elements were involved performing some vital work. The efficiency can be calculated by taking the ratio of the speedup with respect to the number of processing elements involved during the parallel execution of the algorithm. For an
ideal parallel system the efficiency is equal to the 1 where speedup s equal to P. The efficiency is denoted by E and is given by the Eq. 2.4.

\[ E = \frac{s}{p} \]  

(2.4)

2.11.5 Cost

The cost of the parallel algorithm is the cost of solving it on a parallel platform. It is measured by taking the product of parallel runtime and the number of cores used. Cost is the function of the sum of time taken by each processing element to perform that problem. The efficiency of the algorithm can also be represented by taking the ratio of the serial execution time with the cost of solving it in parallel using p processing elements.

The execution time of best known sequential algorithm is considered as the cost of the algorithm for single processing element. If the cost of solving that problem in parallel has same asymptotic growth as a function of the input size is same as the cost of the sequential algorithm then the parallel algorithm is considered as cost-optimal. Since efficiency is the ratio of sequential cost and parallel cost, the efficiency of the cost-optimal system is \( \Theta (1) \).

2.12 Tools Used for the Parallelization

Following are the tools that can be used for parallelization –

2.12.1 Linux Platform

Linux (Bovet and Cesati, 2005) is an UNIX –like open source operating system that is used throughout the world. It is system software specially designed to access and instructs various hardware devices available with the target machine.

2.12.2 GCC Infrastructure

GCC stands for GNUs Compiler collection (Bovet and Cesati, 2005) which is a compiler system introduced by GNU project. It is a collection of compilers for C, C++, Java, Ada, etc. For the experimentations of this
research the GCC infrastructure is used heavily with the combination of Linux Platform to test the proposed algorithms for worldwide agreement.

2.12.3 Faster Parallelization Using OpenMP

During experiments, the OpenMP (stands for Open Multi-Processing) API (application programming interface) has been used to implement the proposed variants on GCC infrastructure (Chapman et al., 2008). The OpenMP API is chosen for the experiments because it is an excellent portable parallel programming model (Chandra, 2001) that can be implemented on shared memory architecture to achieve high parallel capabilities. OpenMP is an open-source API that used multithreading concepts to perform parallel programming. The OpenMP API supports number of programming languages, such as C, C++ and FORTRAN on diversified parallel architectures. In order to make programs parallel, the OpenMP can be inserted into it in the form of compiler directives. After using OpenMP the programs becomes scalable and adapts themselves easily to be executed on multi-core environment.

OpenMP is a explicit parallel programming model based on shared memory paradigm. It comprises of three main components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

Since the OpenMP is based on multithreading, the parallel programming is performed by handling multiple threads in the shared memory. And because it is an explicit programming model, it provides full control to the programmer to perform parallelization.

For parallelization, OpenMP uses the fork-join model. All OpenMP programs begin with a single process handled by the master thread. This master thread executes serially until the first parallel region construct is reached. The procedure of fork-join is shown in Fig. 1.17.
• FORK

The master thread generates a team of child threads. The instructions which are required to be executed parallel are executed by these child threads concurrently. The process of branching of the master thread into multiple child threads is called fork.

• JOIN:

When the child threads finishes with their execution at the end of the parallel region, they wait for other threads to synchronize and then terminated, leaving only the master thread alone. The process of the termination of child threads is called join.

The part of the parallel program which is meant to be executed in parallel is marked using compiler directives which instructs the system to generate the threads before executing the parallel region. Each thread is identified using thread_ID which can be accessed by a function omp_get_thread_num() of OpenMP API. The integer is used to depict the thread_ID of the threads. The thread_ID of the master thread is 0. After finishing with the parallel region all threads join back into master thread which continues the execution of the program till the end.

2.12.4 Fast Modular Computations Using GNUs MP Library

Any RSA implementation involves intensive modular exponentiation and modular reduction computations on very large numbers which poses lot of
complications while executing. One solution for this problem was to design a whole new library to work with large integers. Another solution was to use some existing third party library (Chandra and Chandra, 2005). But the PKC techniques which are proposed during this research are targeted to implement on diversified architectures present on Internet. Therefore the solution should be capable of handling all the complications arise during the cross-platform computations over large integers. The library provided by GNU is already present which is capable of handling all such complications is GNUs MP Library (Granlund, 1996).

GNU’s Multi Precision Library or GNU MP Library is an excellent portable library provided by GNU to be used with GCC infrastructure. GNU MP Library is comprised of multiple portable functions written in C language to perform arbitrary precision arithmetic on integers, rational numbers, and floating-point numbers. The GNUs MP library was developed to perform the fastest possible computations for those applications that require computations on vary large numbers having higher precision. Because it is written in C language, the basic data types of C support it directly.

GNUs MP Library (Paixao, 2003) is designed in such manner that it can provide good performance both in the case of small precision numbers for example few hundred bits as well as large precision numbers such as thousands of bit. The speed of GNUs MP is attained by using full words as the basic arithmetic type, by using sophisticated algorithms, by including carefully optimized assembly code for the most common inner loops for many different CPUs such as Intel, AMD, etc. and by a general emphasis on speed.

Using GMP library, the large numbers are used in following manner –

```
mpz_t large_num;  // Declaration of variable capable of storing numbers
mpz_init(large_num);  // Initialization of variable capable of storing numbers
mpz_set_str(large_num, “61289746129”, 10);  // Storing large integer in variable
```
The facilities offered by the GMP library are heavily throughout the application. The key generation, encryption and decryption routines all use the integer handling functions offered by this library.

2.13 Conclusion

Parallel Programming is one of the favorite upcoming techniques which is used by various researchers throughout the world to introduce time efficient and energy efficient variant for popular sequential algorithms. The parallel program can be based on instruction level parallelism or data level parallelism as per the requirement or design of the sequential algorithm. There are number of performance metrics that can be used to analyze the performance and quality of the parallel algorithm.
CHAPTER 3
PRSA1
PARALLEL RSA
USING
EFFICIENT MEMORY
TECHNIQUES
Chapter 3  
PRSA1 - PARALLEL RSA USING EFFICIENT MEMORY TECHNIQUES

This chapter describes the RSA algorithm and its sub algorithms in depth along with some practical example using small keys. Then various memory-efficient methods are described that are used during the compute-intensive execution of RSA algorithm. In addition to this the two parallel forms based on distinct memory efficient methods and the sequential forms have been discussed. The comparison of all the three forms in terms of speedup is also presented. Thereafter the reasons for selecting the repeated square-and-multiply method as the appropriate memory-efficient method to be used for Parallel RSA algorithm (in collaboration with parallel programming techniques) has been discussed.

3.1 Introduction

The concept of public key cryptography (PKC) public-key cryptography was first presented in 1976 by the Martin Hellman professor of Stanford University and Whitfield Diffie, a graduate student (Diffie and Hellman, 1976a). In their paper, they described a crypto system where instead on one key, two keys are used and these pair of keys can ensure that two parties can communicate securely over un-trusted communication channel (Diffie and Hellman, 1976b). The concept was independently proposed by Ralph Merkle also (Merkle, 1980). Their contribution to cryptography was that the keys could come in pairs i.e. an encryption key and a decryption key and that the decryption key cannot (practically) be derived from the encryption key. The encryption key is known as the public key and the decryption key is called as the private key. The extent of security offered by these algorithms is based on maintaining the secrecy of private key (Bidzos, 1991).
Public key methods are important because they can also be used for sharing the private key secretly when the parties have no chance to share it securely (Diffie and Hellman, 1979). The most important cryptographic algorithms based on PKC is RSA algorithm (Selby and Mitchell, 1989). The RSA algorithm is a simple algorithm based on modular arithmetic (Bidzos, 1991). The RSA encryption and decryption is based on modular arithmetic (series of modular exponentiation and modular reduction) to be performed on large integers. The RSA algorithm becomes compute-intensive and takes longer runtimes as compared to its counterpart the symmetric cryptographic algorithms (Eldridge and Walter, 1993) because of the usage of large size integers, typically 1024 bits. In addition to this due to modular computations on large integers, the problems related to serious memory consumption also occurs (Du et al., 2005).

Due to all these problems the sequential implementation of RSA takes long runtimes and more energy (Mazzeo et al., 2003). One solution to this problem could be the combination of memory efficiency methods with parallel programming (Fan et al., 2010). To find the appropriate memory efficient method for the parallel implementation of RSA various methods have been tested in collaboration with parallel programming. The performance gained has been analyzed by comparing the sequential version with that of parallel variants of RSA.

For experiments, OpenMP API is used on the GCC infrastructure as for general purpose computing has becoming very popular among the parallel programmers community. Many computational problems have gained a significant performance increase by using the extremely impressive parallel capabilities of the Open MP. GCC infrastructure is a platform which makes these APIs available to the general category of the programmers. The OpenMP API makes the parallelization process very simple.

The memory efficient methods (Nedjah and de Macedo Mourelle, 2002) that have been studied and compared for the parallel implementation
Clark and Gregory, 1986) of the modular exponentiation and modular reduction parts of RSA during this research are –

1. Repeated square-and-multiply method (Brickell et al., 1993)

2. Right-to-left binary method (Joye and Yen, 2000)

The RSA algorithm has been implemented in two distinct parallel forms based on the studied memory efficient methods. For the experiments these forms were implemented on dual core computer along with the sequential version. Small key sizes were used to test the memory efficiency methods in collaboration with Parallel Techniques. The results have been measured in terms of time. Consequently promising results have been obtained which are given in the results section.

3.2 RSA Algorithm

The RSA Algorithm is based on Public Infrastructure Based algorithm and was introduced in the paper “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems” in 1977 published by Ron Rivest, Adi Shamir and Len Adleman (Rivest et al., 1983). It is named after the initials of their last names. It is an algorithm for public-key cryptography which was one of the greatest advances in public key cryptography. RSA was proposed as the first algorithm suitable for signing as well as encryption (Chen et al., 2004). At present it is included as the part of Lotus Notes, Quicken and many web browsers like Microsoft and Netscape (Chandu and Babu) and it is also extensively used in e-commerce protocols because it is believed to be sufficiently secure given sufficiently long keys. RSA algorithm is an asymmetric algorithm as unlike its counterpart Symmetric Algorithms, it uses two keys – a pair of private key and a public key. Public key is made public and shared among the users whereas the private key is kept privately and is not shared with anybody (Jones and O’connell, 2002).

The security of RSA is based on the factoring problem i.e. the fact that finding the factors of a given integer is difficult (Paillier, 1999). Because
of the involvement of very large prime numbers the factorization process becomes very compute intensive and takes lot of time to execute even by using best algorithms. This reality provides almost unbreakable security to the message if sufficiently long keys are used (Dodis et al., 2002). For instance, if the key having size 1024 bits or even more than that is used then it becomes practically impossible to pose the threat in front of RSA security (Jonsson and Kaliski Jr, 2002).

RSA algorithm is a combination of three algorithms which are given below –

- Key Generation Algorithm
- Encryption Algorithm
- Decryption Algorithm

### 3.2.1 Key Generation Algorithm

Key Generation is the first step of the RSA algorithm. The receiver entity generates a pair of keys – a public key and a private key. The receiver generates the keys and shares public key with the sender who is responsible for transmitting the message across the un-trusted network. The receiver keeps the corresponding private key with himself only to decrypt the message later. Thus it is apparent that the encryption process is performed using a public key and the decryption is performed using a private key.

The key generation stage of RSA is a multi-step procedure. The steps are given below-

1. The first step is to select a positive integer e as the public key. Usually 65537 is taken as e.

2. Now, randomly select two distinct odd prime numbers p and q in such a manner that there should not be the common divisors of e and (p-1) and e and (q-1).

3. Then n is calculated by multiplying p and q as given in Eq. 3.1.

\[ n = p \cdot q \]  \hspace{1cm} (3.1)
4. The private key $d$ is taken as the inverse of $e$, such that $de - 1$ is divisible by both $(p-1)$ and $(q-1)$. The length of modulus $n$ in octets is the integer $k$ satisfying the Eq. 3.2.

$$2^{8(1 - 1)} \leq n < 2^{8l} \quad (3.2)$$

5. The length $l$ of the modulus must be at least 12 octets to accommodate the block format as described in PKCS#1.

Now, the $(e, n)$ are taken as public key and $(d, n)$ are taken as private key.

### 3.2.2 RSA Encryption

The encryption process comprises of 4 steps: encryption block formatting, octet string to integer conversion, calculation of cipher text, integer to octet string conversion. The RSA encryption function takes the data $D$, modulus $n$, the public key $e$ as an input. The encryption function outputs an octet string $ED$, the encrypted data. The length of data $D$ should not be more than block size-11 octets, which guarantees that the length of padding string will be at least eight octets. The mechanism is displayed in Fig. 3.1.

![RSA Encryption Diagram](image-url)

**Fig. 3.1: RSA Encryption**
3.2.3 RSA Decryption

The RSA decryption function takes an octet string encrypted data ED, a modulus n and a private key d (an exponent) as the input. The function outputs the data string D. The RSA decryption again comprises of 4 steps – Octet String to Integer Conversion, Calculation of Message, Integer to Octet string Conversion, Encryption Block Parsing. The RSA Decryption mechanism is demonstrated in Fig. 3.2.

![Fig. 3.2: RSA Decryption](image)

3.2.4 Example of RSA

The working example of RSA algorithm including key generation, encryption and decryption is as follows –

**Key Generation**

**Step 1.** Choose two prime numbers as p and q

\[ p = 3 \text{ and } q = 11 \]

**Step 2.** Compute \( n = p \times q \)

\[ n = 3 \times 11 \]

\[ n = 33 \]
**Step 3.** Compute $\varphi(n) = (p - 1) \times (q - 1)$

\[
\varphi(n) = 2 \times 10
\]

\[
\varphi(n) = 20
\]

**Step 4.** Choose $e$ such that $1 < e < \varphi(n)$ and $e$ and $n$ are coprime.

Let $e = 7$

**Step 5.** Compute a value for $d$ such that $(d \times e) \% \varphi(n) = 1$.

\[
d = 3 \ [ (3 \times 7) \% 20 = 1 ]
\]

**Step 6.** Public key is $(e, n) \Rightarrow (7, 33)$

**Step 7.** Private key is $(d, n) \Rightarrow (3, 33)$

**RSA Encryption**

**Step 8.** The encryption of $M = 2$

\[
C = 2^7 \% 33
\]

\[
C = 29
\]

**RSA Decryption**

**Step 9.** The decryption of $C = 29$

\[
M = 29^3 \% 33
\]

\[
M = 2
\]

**3.3 Memory Efficiency Methods**

The RSA computations have their roots in modular arithmetic and they are a combination of series of modular exponentiation and modular reduction. Usually the modular exponentiation is computed by raising one integer to another. The modular exponentiation operation is usually performed by repeated multiplications. Thereafter for modular reduction the result is reduced by taking its mod using modulus. The strength of RSA depends upon the fact that how fast these operations can be performed.
The computation of modular operations such as modular addition, modular subtraction and modular multiplication (Kwon et al., 2001) is performed as per the following mathematical equations –

\[(a + b) \mod m = ((a \mod m) + (b \mod m)) \mod m\]

\[(a - b) \mod m = ((a \mod m) - (b \mod m)) \mod m\]

\[(a * b) \mod m = ((a \mod m) * (b \mod m)) \mod m\]

The following sections describe three of the most important techniques that can be used for the computations of modular exponentiation (Nedjah and de Macedo Mourelle, 2002) and modular reduction.

3.3.1 **Modular exponentiation**

Modular exponentiation (Gura et al., 2004) is a process of applying exponentiation over modulus. It is a process of calculating remainder by applying mod operation on the result of positive integer b raised to the power positive integer a, i.e. ab. Generally modular exponentiation is performed by repeated multiplications (McIvor et al., 2004). Various methods have been proposed by the scientists for the faster computation of these operations. Some popular methods are described in next sections.

3.3.1.1 **Naïve modular exponentiation**

The naïve method is the simplest method among all the modular exponentiation methods. In this method modular multiplication is applied repeatedly for the number of times equal to the exponent (Bellare et al., 1998). For example if base g=4, exponent e=8, and modulus m=267 then \(r \equiv g^e \mod m\) is calculated as described in Table 3.1.

As displayed in the Table 3.1 the final result r for the given example will be 121. In the naïve method the modular multiplication is performed for e-1 times. This method is not efficient because e-1 modular multiplications are required.
### Table 3.1: Calculation based on Naïve Method

<table>
<thead>
<tr>
<th>S. No</th>
<th>Value of e</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>e = 1</td>
<td>r = 4 mod 267 = 4.</td>
</tr>
<tr>
<td>2.</td>
<td>e = 2</td>
<td>r = (4 * 4) mod 267 = 16.</td>
</tr>
<tr>
<td>3.</td>
<td>e = 3</td>
<td>r = (16 * 4) mod 267 = 64 mod 267 = 64.</td>
</tr>
<tr>
<td>4.</td>
<td>e = 4</td>
<td>r = (64 * 4) mod 267 = 256 mod 267 = 256.</td>
</tr>
<tr>
<td>5.</td>
<td>e = 5</td>
<td>r = (256 * 4) mod 267 = 1024 mod 267 = 223.</td>
</tr>
<tr>
<td>6.</td>
<td>e = 6</td>
<td>r = (223 * 4) mod 267 = 892 mod 267 = 91.</td>
</tr>
<tr>
<td>7.</td>
<td>e = 7</td>
<td>r = (91 * 4) mod 267 = 364 mod 267 = 97.</td>
</tr>
<tr>
<td>8.</td>
<td>e = 8</td>
<td>r = (97 * 4) mod 267 = 388 mod 267 = 121.</td>
</tr>
</tbody>
</table>

The performance of any PKC based cryptography technique is determined by how faster the modular exponentiation and modular reduction operations (Bosselaers et al., 1994) can be performed. In the RSA algorithm, to ensure high level of security, the key size is taken very large. These private and public keys are used as exponents in the RSA algorithm. In addition to this the plain text, cipher text and especially partial ciphered text becomes very large because of the large exponents. Due to all these factors it is vital to have lesser number of modular multiplications and also the time involved in the process should be reduced to get an efficient version of RSA. There are some other methods available for modular exponentiation which are relatively faster as compared to the naïve method and can be used to speed up the RSA encryption and decryption process. Some of these methods are described in next section.
3.3.1.2 Repeated square-and-multiply method

The repeated square-and-multiply modular exponentiation (Brickell et al., 1993) algorithm is divided into two steps. Firstly the exponent is divided into two or more parts as required and secondly the final result is obtained by squaring the result as many times. This method is based on the Eq. 3.3 that for an even \( e \),

\[
g^e \mod m = (g^{\frac{e}{2}} \times g^{\frac{e}{2}}) \mod m
\]  

(3.3)

The recursive definition of exponentiation by squaring is illustrated in Fig. 3.3. This method is very helpful in the case of larger \( e \) because it reduces the number of modular multiplications to the great extent. For example if there is \( t \) number of bits present in the exponent then it reduces the number of modular multiplications up to 2\( t \). Therefore this method provides enormous enhancement for a large \( e \).

\[
\begin{align*}
\text{ModExp}(g, e, m) &= \begin{cases} 
1, & \text{if } e = 0 \\
(g \times \text{ModExp}(g, (e-1), m)) \mod m, & \text{if } e \text{ is odd} \\
\text{ModExp}(g, e/2, m)^2 \mod m, & \text{if } e \text{ is even}
\end{cases}
\end{align*}
\]

Fig. 3.3: Recursive definition of modular exponentiation by squaring

3.3.1.3 Right-to-left binary modular exponentiation

The right-to-left binary algorithm gives greater improvement for the computation of modular exponentiation. In this method first the exponent is converted to the binary form. Then this binary representation of the exponent is read from right-to-left for the computation of the modular exponentiation. The lowest bits of \( e \) are considered first. The result is calculated using Eq. 3.4.

\[
res = \prod_{i=0}^{n-1} (b^{2^i}) \mod m
\]  

(3.4)

The algorithm used for this method is described as Algorithm 3.1.
Algorithm 3.1: Right-to-left binary modular exponentiation

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> an element ( g ) and integer ( e \leq 1 ), and a modulus ( m ).&lt;br&gt;<strong>Output:</strong> ( g^e \mod m ).&lt;br&gt;1. ( A = 1 ), ( S = g ), ( E = e ).&lt;br&gt;2. While ( E \neq 0 ) do the following:&lt;br&gt;   2.1. If ( E ) is odd, then ( A = (A \cdot S) \mod m ), ( E = E - 1 ).&lt;br&gt;   2.2. ( E = E/2 ).&lt;br&gt;   2.3. If ( E \neq 0 ), then ( S = (S \cdot S) \mod m ).&lt;br&gt;3. Return ( (A) ).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.3.2 Modular reduction<br>A modular reduction (Bosselaers et al., 1994) is the process of calculating remainder of an integer division operation. This is shown in the Eq. 3.5 which is given below –

\[
A \mod m = A - \left\lfloor \frac{A}{m} \right\rfloor X m
\]

(3.5)

3.3.2.1 Naïve Modular Reduction<br>In the naïve modular reduction algorithm (Nedjah and Mourelle, 2006) the modulus is successively subtracted from the base until the remainder found is non-negative and smaller than the modulus. The naïve sequential division algorithm displayed as Algorithm 3.2, successively subtracts the modulus until a remainder that is non-negative and smaller than the modulus is found.

Algorithm 3.2: Naive Modular Reduction Algorithm

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm NaiveReduction(P, M)&lt;br&gt;Int R = P;&lt;br&gt;Do R = R - M; While R &gt; 0;&lt;br&gt;If R ≠ 0 Then R = R + M;&lt;br&gt;Return R;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
During the execution of the algorithm at every step the remainder is stored temporarily because it may be the final possible remainder.

3.4 **First Form: Sequential Implementation of RSA Algorithm**

Practically the RSA algorithm is the combination of three distinct algorithms: Key Generation algorithm (Gilboa, 1999), Encryption algorithm and the Decryption algorithm (Bleichenbacher, 1998). Therefore, the sequential RSA algorithm has been divided into three parts as described in the next three sections.

3.4.1 **Key Generation**

The key generation part of RSA algorithm (Gilboa, 1999) has been implemented as a multi-step process which is given below –

1. Choose two large random prime integers: p and q.

2. Compute \( m = p \times q \), which is used as a modulus for the modular reduction part.

3. Compute \( \phi(n) = (p-1)(q-1) \).

4. Choose an integer e, the public key, \( 1 < e < \phi(n) \) such that: \( \text{GCD}(e, \phi(n)) = 1 \) (GCD is greatest common denominator)

5. Compute d, the private key, \( 1 < d < \phi(n) \) such that: \( ed \equiv 1 \pmod{\phi(n)} \)

Since in the above procedure e is the public or encryption exponent and d is the private or decryption exponent, thus e and n have been published as the public key and d and n have been kept as the secret key.

3.4.2 **RSA Encryption**

In order to encrypt, the plain text data has been raised to the power of encryption key and then divided by the product of the prime numbers to calculate the remainder (Bleichenbacher, 1998). The remainder is sent as cipher text as per Eq. 3.6.
\[ C = M^e \% m \]  

(3.6)

Where,

C is the Cipher Text,

M is the plain text,

e is the Private Key, and

m is the modulus.

**3.4.3 RSA Decryption**

In order to encrypt, the cipher text data is raised to the power of decryption key and then divided by the product of the prime numbers to calculate the remainder (Bleichenbacher, 1998). The remainder is the original plain text as per Eq. 3.7.

\[ M = C^d \% m \]  

(3.7)

Where,

C is the Cipher Text,

M is the plain text,

e is the Private Key, and

m is the modulus.

**3.5 Second Form: Parallel RSA based on Repeated-Square and Multiply Method**

This form of parallel implementation of RSA has been based on the repeated square-and-multiply method shown by Eq. 3.1. This method improves the performance of RSA to a great extent for larger keys. The algorithm used for the method is given as Algorithm 3.3.

The modular exponentiation part of RSA has been parallelized in a manner described in next two sections.
3.5.1 For Even Exponent

For even exponent the final cipher has been calculated as –

- e was divided into two or four parts as per the availability of the cores in the system.
- Each part of the exponentiation computation was assigned to multiple threads
- finally the results of each thread were multiplied together to get the final result.
- Final result was reduced by applying modular reduction using modulus.

3.5.2 For Odd Exponent

For odd exponent final cipher has been calculated as –

- The value of Message M was stored in one variable A.
- Subtracted 1 from the e to make it even.
- e was divided into 2/4 parts depending upon the number of cores present in the target machine.
- Each exponentiation computation was allocated to a separate thread.
- The results obtained from each thread were multiplied together to get a combined result.
- Finally the result was multiplied by A to get the final result.
- Final result was reduced by applying modular reduction using modulus.
Algorithm 3.3: Parallel RSA based on Repeated Square-and-Multiply Method

Procedure: square_repeat
Model: Thread Model based on repeated square-and-multiply
Input: base, Power, modulus
Output: Result
Declare:
    Global: N, Number_of_Thread, Cipher
    Local: None
Parbegin
    Declare Result := 1
    Assign N := Power / Number_of_Thread
    for Pi, i = 1 to Num_proc do In Parallel
        if Power mod Number_of_Thread := 0
            for i := 1 to Power
                for j := 1 to N
                    Assign Result := Result * Base
                end for
            end for
        else
            Assign N := N - 1
            for i := 1 to Power
                for j := 1 to N
                    Assign Result := Result * Base
                end for
            end for
            Assign Result := Result * Base;
        end if
    end In Parallel
    Assign Cipher := Result mod Modulus
Parend
### 3.6 Third form: Parallel RSA based on Right-to-Left Binary Method

This form of parallel implementation is based on right-to-left binary method which is in turn based on the principle of exponentiation by squaring or binary exponentiation. The method is called right-to-left binary method because the binary representation of the exponent is computed from right to left. Firstly the exponent is converted into its binary representation and the right most bit is considered first. Thereafter the algorithm given as Algorithm 3.4 is used to calculate the value of expression given in Eq. 3.8.

\[
Base^{exponent} \mod modulus
\]  

(3.8)

**Algorithm 3.4: Right-to-left binary modular exponentiation Algorithm**

<table>
<thead>
<tr>
<th>Procedure: power_binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model: Thread Model based on repeated square-and-multiply</td>
</tr>
<tr>
<td>Input: base, BinaryNumber, length, modulus</td>
</tr>
<tr>
<td>Output: Result</td>
</tr>
<tr>
<td>Declare:</td>
</tr>
<tr>
<td>Global: Num_proc</td>
</tr>
<tr>
<td>Local: None</td>
</tr>
<tr>
<td>Parbegin</td>
</tr>
<tr>
<td>Assign Result := 1</td>
</tr>
<tr>
<td>for Pi, i = 1 to Num_proc do In Parallel</td>
</tr>
<tr>
<td>for j = length to 0</td>
</tr>
<tr>
<td>if BinaryNumber[i] mod 2 == 1</td>
</tr>
<tr>
<td>Assign Result := (result * base) mod modulus;</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>Assign base := (base * base) % modulus;</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>return Result</td>
</tr>
<tr>
<td>end In Parallel</td>
</tr>
<tr>
<td>Parend</td>
</tr>
</tbody>
</table>
The loop used in the algorithm is executed for the number of times equal to the number of bits present in the binary notation of the exponent. The calculation performed on the principle is based on Eq. 3.4.

Here, first the exponent was converted into binary form and then calculations were performed. Then the bits were verified whether it is 0 or 1. For this verification the bits were considered from right to left. If it was found to be 0, then no calculation was performed. But if bit was 1 then calculation was performed.

For example if base = 4, exponent = 13, and modulus = 497 then the cipher text is calculated as the method given in Table 3.2. The binary equivalent of exponent 13 is 1101. Because exponent is four binary digits in length, the loop executes only four times.

<table>
<thead>
<tr>
<th>n^{th} Iteration</th>
<th>Bit</th>
<th>Value of Result</th>
<th>Value of Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>(1*4) % 497 = 4</td>
<td>(4*4) % 497 = 16</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>No calculation, Value will remain same</td>
<td>(16*16) % 497 = 256</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>(4*256) % 497 = 30</td>
<td>(256*256) % 497 = 429</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>(30*429) % 497 = 445</td>
<td>(429*429) % 497 = 151</td>
</tr>
</tbody>
</table>

The loop then terminates since exponent is zero, and the result 445 is achieved.

### 3.7 Methodology

The experiments are performed on dual core computers using OpenMP GCC infrastructure in Linux environment. Three different forms of RSA implementation are developed – first is based on Sequential algorithm
developed using C Language and executed on GCC infrastructure on Linux platform, second and third are redesigned Parallel algorithms based on repeated-square-and-multiply method, and right-to-left binary method using the combination of OpenMP API and C language on GCC infrastructure.

The performance gained is measured / analyzed in terms of time. Time is measured using time utility of Linux. Each program is executed for 25 times using the same input and average of time is taken.

Both forms are divided into three parts - Key generation, Encryption and Decryption. To start with the process of key generation, same set of prime numbers is taken in all three forms to generate the same set of private and public keys. Also the same message is used for encryption / decryption to observe the correct differences in terms of performance gained for all the forms.

### 3.8 Platform used for Experiments

The configuration of the computer system which was used during the experimentations is described in the Table 3.3.

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Processor</td>
<td>Intel Core 2 Duo CPU</td>
</tr>
<tr>
<td>2.</td>
<td>Frequency</td>
<td>1.40 GHz</td>
</tr>
<tr>
<td>3.</td>
<td>RAM</td>
<td>2.00 GB</td>
</tr>
<tr>
<td>4.</td>
<td>System Type</td>
<td>32 Bit Operating System</td>
</tr>
<tr>
<td>5.</td>
<td>Environment</td>
<td>Ubuntu Linux</td>
</tr>
<tr>
<td>6.</td>
<td>Platform</td>
<td>GCC Infrastructure</td>
</tr>
</tbody>
</table>
3.9 Results and Discussion

The comparison of results obtained during the experiments is displayed in Table 3.4. The graphical representation of the comparison of results is given in Fig. 3.3.

Table 3.4: Comparative Results of the three forms of RSA

<table>
<thead>
<tr>
<th>S. No</th>
<th>Type of Implementation</th>
<th>Implementation Based on</th>
<th>Time taken sequential / parallel execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>First Form</td>
<td>Sequential implementation</td>
<td>0m4.640s</td>
</tr>
<tr>
<td>2</td>
<td>Second Form</td>
<td>Repeated square and multiply method</td>
<td>0m3.700s</td>
</tr>
<tr>
<td>3</td>
<td>Third Form</td>
<td>Right-to-Left Binary Method</td>
<td>0m3.409s</td>
</tr>
</tbody>
</table>

Fig. 3.4: Graphical Representation of Time of various Implementations

It has been observed during the experiments that the repeated square-and-multiply method of modular exponentiation is more efficient than the naive method and the right-to-left binary method of repeated multiplication.
Therefore, in the later research only repeated square-and-multiply method has been used because the right-to-left binary method results in overheads when used for the computations involving large integers.

3.10 Conclusions

The RSA algorithm is based on modular exponentiation involving very large numbers. As a result, it becomes compute-intensive and takes longer time to execute. In addition to this, due to the involvement of large numbers, the memory consumption of the algorithm also increases. To overcome these problems, RSA algorithm was redesigned as parallel algorithm using two distinct memory-efficient methods to increase its overall efficiency.

The experimental result shows that incorporating memory-efficient methods into the parallel RSA algorithm improves its performance significantly. Two methods were used to implement parallel RSA – repeated square-and-multiply method and right-to-left binary method. Both methods improved the performance of RSA significantly. However, Right-to-left binary method had shown better result over repeated square-and-multiply method using small key sizes. But when it has been used with larger key size, the performance reduced drastically.

Therefore for the further experiments in this research work, repeated square-and-multiply method has been used as it provides consistent results.
CHAPTER 4
PRSA2
PARALLEL RSA
ALGORITHM BASED ON
REPEATED SQUARE-AND-
MULTIPLY METHOD
Chapter 4

PRSA 1 – PARALLEL RSA ALGORITHM BASED ON REPEATED SQUARE-AND-MULTIPLY METHOD

This chapter includes the development of efficient parallel RSA algorithm capable of executing the RSA with faster speed and optimum memory utilization. It is addressing objective number 5 and 6 of this research work. In addition to this, the redesigned parallel algorithm should also be tested using large data sets to prove the efficiency of the algorithm performance of the algorithm. The results of these tests have been discussed also. A description of PRSA1 algorithm which has been proposed during this research work is included first. The PRSA1 algorithm has been designed by incorporating repeated square-and-multiply method along with the parallel computing techniques into the conventional RSA algorithm in order to make it time and memory efficient. The PRSA1 has been tested using large data sets divided into two distinct categories. During experimentation, PRSA1 has shown promising results and leads to a speedup of approximately 5X as compared to its sequential counterpart.

4.1 Introduction

RSA, as described in previous chapter, is one of the most important PKC-based algorithms (Salomaa, 1996) which is widely used for the cryptography as well as for digital signing. RSA is based on the factorization technique, which is a technique of finding two numbers whose product is given. And it becomes almost impractical to find two large prime numbers whose product is composed of 1024 bits or more. Therefore it is not possible to break the security of the RSA algorithm easily (William and Stallings, 2006).

In the RSA algorithm, larger size keys of about 1024 bits or more are used. Moreover, the RSA encryption and decryption is based on repeated modular exponentiation and modular reduction on very large numbers which make it compute-intensive and energy-intensive as well
(Schroeppel et al., 2003). In addition to this compute-intensive behavior, the memory consumption of such computations also becomes very high (Shamir, 1996). They are much slower than the symmetric key algorithms and take a lot of time to execute. But despite all these problems RSA provides stronger security to the message and therefore it is a widely used algorithm (Jonsson and Kaliski Jr, 2002). The need of the hour is to increase the speed of RSA encryption, decryption and improve its memory consumption.

To find the solutions of above mentioned problems two distinct technologies have been used – parallel programming to speed up the RSA encryption and decryption involving larger keys and repeated square-and-multiply method (Hui and Lam, 1994) to improve the memory efficiency of the certain parts of the algorithm. Using above mentioned techniques a new parallel RSA algorithm, PRSA1 has been designed and proposed during this research work. In order to test the PRSA1 algorithm it is implemented using OpenMP and GNU’s MP Library (Miyaji et al., 1997) on the GCC infrastructure in the Linux environment.

As it is already been described that RSA works on very large integers but it is difficult and impractical to work on such large numbers on the GCC infrastructure directly. Therefore, it is vital to have some third party software that may assist in manipulating large numbers on GCC infrastructure. GNU provides an excellent library called GNU’s Multi Precision Library – GMP (Granlund, 1996) to handle large number having arbitrary precision. Thus to handle and manipulate such large numbers during the execution of PRSA 1 the functions of GMP Library has been used.

Recently, the OpenMP (Chandra, 2001) API on the GCC infrastructure has been gaining popularity at global level for parallelizing algorithms. A lot of computational problems have gained a significant performance increase by using the OpenMP parallel API. The OpenMP API has been made available to general programmers by the GCC infrastructure. The OpenMP API, which is based on “pragma” approach simplifies the job of
programmer to design parallel programs (Chapman et al., 2008). Therefore the OpenMP API has been used in collaboration with GMP Library to implement PRSA 1 algorithm.

During the experiments the PRSA 1 algorithm has been tested using large test case sets. These test cases are divided into two groups that have been described in the methodology section. The results obtained during the experiments were promising and shown approximately 5x speed up as compared to its sequential counterpart.

4.2 PRSA 1 – The Parallel RSA Algorithm

The PRSA 1 algorithm has been designed in order to increase the memory and time efficiency of the conventional RSA algorithm. In order to increase the memory efficiency of the algorithm, as described in the last chapter, the algorithm was tested using two memory efficient methods – right-to-left binary method and repeated square-and-multiply method. Both forms were implemented in parallel using OpenMP on the GCC infrastructure. The results obtained from the forms had shown promising results as described in last chapter. For the small key sizes the right-to-left method was giving better results as compared to repeated square-and-multiply method but when they were tested for keys with larger size the form based on repeated square-and-multiply method had better results. During performance analysis it has been observed that the right-to-left binary method shows overheads when used for the computations involving large integers, because the computations involved in it are composed of two major parts. And, therefore these parts can be parallelized in at the most two sections efficiently whereas the repeated square-and-multiply method can be distributed into as many sections as needed or the number of cores available. Thus repeated square-and-multiply method allows more efficient parallelization as compared to right-to-left binary method; therefore during further research work only repeated square-and-multiply method has been used.
To redesign PRSA 1 algorithm the repeated square-and-multiply method was chosen in collaboration with parallel computing.

PRSA 1 has been divided into three parts – the key generation algorithm, the encryption algorithm and the decryption algorithm. The key generation part was designed sequentially because it is a step-wise process and it cannot be made parallel. The process of key generation is same as described in Chapter 2.

The encryption and decryption parts were parallelized to increase the speed and reduce the time involved in the processes. In order to compute the cipher text form the plain text (encryption) or to compute the plain text from the cipher text (decryption) the operations that were required to be performed were – modular exponentiation and modular reduction. Therefore, the repeated square-and-multiply method was used to increase the time and memory efficiency of the algorithm.

The repeated square-and-multiply modular exponentiation algorithm is based on Eq. 4.1, as stated for an even value of $e$,

$$g^e \mod m = (g^{e/2} \times g^{e/2}) \mod m$$ (4.1)

The PRSA1 improved the performance of encryption and decryption involving large size keys to the greater extent. The modular exponentiation part of PRSA1 was redesigned in the fashion described in the next two subsections.

### 4.1.1 PRSA1 for Even Key

In case of even key, the key was divided into two, four or more parts based on the number of cores available on the system. Now, each fraction of the exponentiation computation was assigned to different threads. Thereafter the results obtained by different cores were multiplied together to find the final result. Then the process of modular reduction was applied to the result in order to find the final cipher text, for which the “mod” operation was applied to it. The process of modular exponentiation for even exponent is displayed in Fig. 4.1.
4.1.2 PRSA1 for Odd Exponent

For odd keys the value of Message $M$ was stored in one variable $A$. 

![Diagram of Modular Exponentiation for Odd Keys](image)
Then 1 was subtracted from the key to make it even. Thereafter the key was divided into two, four or more divisions as per the number of cores present on the computer system. Each exponentiation computation was allocated to a separate thread. The result obtained from each threads were multiplied together to get the combined result. At the end, the result obtained so far has been multiplied to $A$ which results into the final result. Final result was reduced by applying modular reduction using modulus. The process of modular exponentiation for odd exponent is displayed in Fig. 4.2.

The proposed algorithm for parallel RSA is given in Algorithm 4.1.

**Algorithm 4.1: Proposed PRSA1 Algorithm**

```
Procedure: square_repeat
Model: Thread Model based on repeated square-and-multiply
Input: base, Power, modulus
Output: Result
Declare:
    Global: N, Number_of_Cores, Cipher
    Local: None
Assign N:=Power/Number_of_Cores
Divide the whole modular exponentiation and reduction into N parts
Parbegin
Declare Result:=1
If Power mod 2 := 0
    for j := 1 to N
        Assign Result := Result * Base
    end for
else
    for j := 1 to N
        Assign Result := Result * Base
    end for
    Assign Result := Result * Base;
end if
Assign Cipher := Result mod Modulus
Parend
```
4.3 Methodology

The proposed PRSA 1 algorithm has been divided into three sub algorithms - Key Generation algorithm, Encryption algorithm and Decryption algorithm. The parallel RSA algorithm is based on popular memory efficient method, repeated square-and-multiply method to implement modular exponentiation and modular reduction. The key generation part was implemented sequentially because of its step-wise nature. Refer Appendix A for code snippets of PRSA1.

The encryption and decryption part is parallelized to increase the speed of the compute-intensive portions of PRSA 1. The diagrammatical representation of PRSA 1 is described in Fig 4.3.

![Diagram of PRSA1 Methodology]

Fig.4.3: PRSA1 Methodology

In order to parallelize the PRSA1 algorithm the principle of instruction level parallelism was used. As described earlier the encryption and
decryption is solely based on the combination of modular exponentiation and modular reduction as below –

\[ C = M^e \% m \]  

(4.2)

To use the instruction level parallelism in the process described in Eq. 4.2 the computations were divided into multiple parts. To perform this chosen memory efficient method was applied to it. For this the key which is used as exponent in the process was split into multiple parts as described in Eq. 4.3.

\[ C = M^e \% m = ((M^{\frac{e}{2}} \times M^{\frac{e}{2}}) \% m) \]  

(4.3)

Then each of this part was computed in parallel.

To divide the exponent into multiple parts data decomposition technique was used. To perform this first the exponent was divided into multiple parts using data partitioning and then these were incorporated into each part of the computation. A coarse grain granularity was used for the division of key because the fine grained granularity was resulting in excessive overheads to the procedure. Thereafter each was converted into tasks and then these tasks were assigned to different cores in order to execute them concurrently.

The PRSA1 was based on threads model or shared memory model. As described in the last paragraph, different parts of instructions were converted into tasks. These tasks were mapped to the cores using static mapping. The static mapping was used to map the tasks onto cores because these tasks involved the array operations; and to map array operations onto the cores, generally, static mapping is used.

All threads that were mapped onto the cores run in the shared memory area and share the memory space. Therefore the variables which were required to be shared among the threads were assigned the memory in the
shared memory area. In contrast to this the thread dependent variables were assigned memory in the cache memory to increase the speed of the reading and writing variables.

To test the algorithm it was implemented using the OpenMP API on the GCC infrastructure in the presence of GNU’s MP Library and in the Linux environment. The algorithm was tested using two distinct test cases. The experiments performed in order to improve the performance of RSA on multi-core machines had shown some encouraging results.

While executing PRSA1 on multi-core machine it was observed that the execution time decreased drastically. The performance gained was measured in terms of time that was spent by various forms during the experimentation. The “time” utility of Linux has been used to measure the time taken by all three forms.

In order to find the exact improvement in the performance of encryption and decryption performed by PRSA1 over the sequential RSA, the implementation was executed on a dual core, a quad core, a 6 cores and a dual quad core configuration of a computer. Each experiment was performed repeatedly 25 times using same set of keys and message and the average was taken as the final time.

The time was measured in terms of two distinct categories – encryption time and decryption time. Encryption time is the time measured during the encryption process whereas the decryption time is the time taken during the decryption process.

In RSA algorithm the encryption is performed using small size public keys whereas the decryption involves large size private keys. Therefore as compared to encryption the decryption process takes larger run times. The PRSA1 algorithm has shown the significant improvement in the case of the decryption process as well.
4.4 Platform used for Experiments

The configuration of the computer system which was used during the experimentations is described in the Table 4.1.

Table 4.1: Configuration of Computer used for Experiments

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Processor</td>
<td>AMD FX (tm) - 8120 Eight-Core Processor</td>
</tr>
<tr>
<td>2.</td>
<td>Frequency</td>
<td>3.10 GHz.</td>
</tr>
<tr>
<td>3.</td>
<td>RAM</td>
<td>4.00 GB</td>
</tr>
<tr>
<td>4.</td>
<td>System Type</td>
<td>64 Bit Operating System</td>
</tr>
<tr>
<td>5.</td>
<td>Environment</td>
<td>Ubuntu Linux</td>
</tr>
<tr>
<td>6.</td>
<td>Platform</td>
<td>GCC Infrastructure</td>
</tr>
<tr>
<td>7.</td>
<td>Big Integer Library</td>
<td>GNU's MP Library</td>
</tr>
</tbody>
</table>

4.5 Experimental Results

During the experimentation, two different set of test cases were used to test and prove the strength of the PRSA1. Each test case composed of two parameters – the key size and the message size. In the first test case set the key size was varied whereas the message size remained constant for all the test cases. On the contrary, in the second test case set, the key size remained constant whereas the message size was varied for the test cases. For each test case, PRSA1 provide promising results in terms of execution time. The test cases along with their respective execution times are described in the following two sections.

4.5.1 Test case Set 1

Under first set of test cases variety of experiments were performed by taking different key sizes varying from 128 bits to 2048 bits. For each key
size same set of message having size of 1,000 characters was taken for encryption and decryption to find the exact difference between the execution time of serial RSA and PRSA1. The execution times of encryption and decryption modules obtained for Test Case Set 1 are given in the Table 4.2 and Table 4.3 respectively. The Table 4.2 and Table 4.3 shows the performance comparison and enhancement in parallel runtimes using 2, 4, 6 and 8 cores versus that of a sequential implementation using a single core for different set of test cases. It is evident through the results shown in the tables that the speedup increased when the number of cores increased to execute the PRSA1 because when the number of cores increased the computations were distributed among all available cores due to its scalability feature and execution time decreased.

Table 4.2: Execution time of Encryption Test case Set 1

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size (in Bits)</th>
<th>Encryption Time taken with respect to the serial and parallel execution of the same code on varied number of cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 Core</td>
</tr>
<tr>
<td>1.</td>
<td>128</td>
<td>0.01025</td>
</tr>
<tr>
<td>2.</td>
<td>256</td>
<td>0.02001</td>
</tr>
<tr>
<td>3.</td>
<td>512</td>
<td>0.04934</td>
</tr>
<tr>
<td>4.</td>
<td>768</td>
<td>0.1126</td>
</tr>
<tr>
<td>5.</td>
<td>1024</td>
<td>0.18261</td>
</tr>
<tr>
<td>6.</td>
<td>1280</td>
<td>0.2665</td>
</tr>
<tr>
<td>7.</td>
<td>1536</td>
<td>0.40371</td>
</tr>
<tr>
<td>8.</td>
<td>1792</td>
<td>0.53012</td>
</tr>
<tr>
<td>9.</td>
<td>2048</td>
<td>0.69422</td>
</tr>
</tbody>
</table>
Fig. 4.4: Execution time of Encryption on 2, 4, 6 and 8 cores (Test Case Set 1)

Table 4.3: Execution time of Decryption Test Case Set 1

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size (in Bits)</th>
<th>Decryption Time taken with respect to the serial and parallel execution of the same code on varied number of cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 Core</td>
</tr>
<tr>
<td>1.</td>
<td>128</td>
<td>0.06553</td>
</tr>
<tr>
<td>2.</td>
<td>256</td>
<td>0.29416</td>
</tr>
<tr>
<td>3.</td>
<td>512</td>
<td>1.37108</td>
</tr>
<tr>
<td>4.</td>
<td>768</td>
<td>4.81935</td>
</tr>
<tr>
<td>5.</td>
<td>1024</td>
<td>10.988</td>
</tr>
<tr>
<td>7.</td>
<td>1536</td>
<td>34.8094</td>
</tr>
<tr>
<td>8.</td>
<td>1792</td>
<td>54.349</td>
</tr>
<tr>
<td>9.</td>
<td>2048</td>
<td>82.5997</td>
</tr>
</tbody>
</table>

It is evident in the fig. 4.4 and 4.5 that as and when the key size and the number of processing elements are increased simultaneously, the PRSA1
starts giving the better results by taking lesser execution time. The graphs also show the graphical comparison between the execution times of PRSA1 on 2, 4, 6 and 8 cores. It can also be observed in the graphs that comparative speedup also increases when key size increased.

Fig. 4.5: Execution time of Decryption on 2, 4, 6 and 8 cores (Test Case Set 1)

The PRSA1 gave encouraging results for the test cases when the key size was taken sufficiently large such as 2048 bits. In such cases the execution time decreased at consistent rate when the number of cores had increased from 2 cores to 8 cores as compared to the serial version of the RSA executed on single core. Whereas the execution time of sequential code has been increased notably when the key size increased from 128 bits to 2048 bits.

### 4.5.2 Test case Set 2

Under second set of test cases the experiments have been performed by taking fixed key size of 1024 bits and message of variable sizes from 1 KB to 10 KB. Each test case was run on 2, 4, 6 and 8 cores machine repeatedly 25 times and the average of all readings was taken as the final execution time. The encryption and decryption execution time for test case set 2 are given in Table 4.4 and Table 4.5 respectively.
Table 4.4: Execution time of Encryption Test case Set 2

<table>
<thead>
<tr>
<th>S. No</th>
<th>Message Size</th>
<th>Encryption Time taken with respect to the serial and parallel execution of the same code on varied number of cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 Core</td>
</tr>
<tr>
<td>1.</td>
<td>1 KB</td>
<td>0.03016</td>
</tr>
<tr>
<td>2.</td>
<td>2 KB</td>
<td>0.06294</td>
</tr>
<tr>
<td>3.</td>
<td>3 KB</td>
<td>0.09095</td>
</tr>
<tr>
<td>4.</td>
<td>4 KB</td>
<td>0.1202</td>
</tr>
<tr>
<td>5.</td>
<td>5 KB</td>
<td>0.1506</td>
</tr>
<tr>
<td>6.</td>
<td>6 KB</td>
<td>0.19511</td>
</tr>
<tr>
<td>7.</td>
<td>7 KB</td>
<td>0.21291</td>
</tr>
<tr>
<td>8.</td>
<td>8 KB</td>
<td>0.24487</td>
</tr>
<tr>
<td>9.</td>
<td>9 KB</td>
<td>0.27096</td>
</tr>
<tr>
<td>10.</td>
<td>10 KB</td>
<td>0.30198</td>
</tr>
</tbody>
</table>

Fig. 4.6: Execution time of Encryption on 2, 4, 6 and 8 cores (Test case Set 2)
Table 4.5: Execution time of Decryption Test case Set 2

<table>
<thead>
<tr>
<th>S. No</th>
<th>Message Size</th>
<th>Encryption Time taken with respect to the serial and parallel execution of the same code on varied number of cores</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 Core</td>
</tr>
<tr>
<td>1.</td>
<td>1 KB</td>
<td>1.87211</td>
</tr>
<tr>
<td>2.</td>
<td>2 KB</td>
<td>3.73262</td>
</tr>
<tr>
<td>3.</td>
<td>3 KB</td>
<td>5.63559</td>
</tr>
<tr>
<td>4.</td>
<td>4 KB</td>
<td>7.50242</td>
</tr>
<tr>
<td>5.</td>
<td>5 KB</td>
<td>9.30783</td>
</tr>
<tr>
<td>6.</td>
<td>6 KB</td>
<td>11.2475</td>
</tr>
<tr>
<td>7.</td>
<td>7 KB</td>
<td>13.1198</td>
</tr>
<tr>
<td>8.</td>
<td>8 KB</td>
<td>15.0225</td>
</tr>
<tr>
<td>9.</td>
<td>9 KB</td>
<td>16.8589</td>
</tr>
<tr>
<td>10.</td>
<td>10 KB</td>
<td>18.6845</td>
</tr>
</tbody>
</table>

Fig. 4.7: Execution time of Decryption on 2, 4, 6 and 8 cores (Test Case Set 2)
The Table 4.4 and 4.5 show the improvements and performance comparison in parallel execution time using 2 cores, 4 cores, 6 cores and 8 cores versus that of a sequential implementation using a single core for test case set 2. It is evident through the results shown in the tables that the speedup increased when the number of cores increased to execute the PRSA1 because when the number of cores increased the computations were distributed among all available cores due to its load balancing feature. In addition to this, the speedup also increased when the message size increased from 1 KB to 10 KB because the computations are distributed to all the available cores and PRSA1 started performing better and better due to its optimal parallel behavior.

It is evident in the fig. 4.6 and 4.7 that as and when the message size and the number of processing elements are increased simultaneously, the PRSA1 starts giving the better results by taking lesser execution time. The graphs also show the graphical comparison between the execution times of PRSA1 on 2, 4, 6 and 8 cores. It can also be observed in the graphs that comparative speedup also increases when message size increased.

4.6.4 Performance Analysis

The PRSA1 algorithm has been redesigned by incorporating repeated square-and-multiply method. While designing it, the focus was on increasing the memory efficiency and speedup of the algorithm. The algorithm was implemented using OpenMP API and tested using various parameters as discussed in experimental results section. The performance of PRSA1 has been analyzed using various performance metrics and the same is mentioned in the next four subsections.

4.5.3 Speedup

Speedup is one of the most important performance metrics for a parallel algorithm. It is comparative measure of solving the problem in parallel as compared to sequential which is calculated by taking the ratio of the execution time taken by the serial algorithm on a single processing
element with respect to the execution time taken by the parallel algorithm on a parallel computing platform having p number of identical processing elements. Speedup is calculated by following expression –

\[ S = \frac{T_s}{T_p} \]  

Where \( T_s \) is the execution time of sequential algorithm and \( T_p \) is the execution time of the parallel algorithm. The speedup obtained by the PRSA1 for both test cases has been calculated using the expression given in Eq. 4.3.

The speedups obtained while testing PRSA1 on dual quad core (8 cores) based on Test case set 1 are given in Table 4.6. It is evident through the results that the speedup of approximately 5X has been obtained when PRSA1 executed on 8 core machine as compared to the serial version of the RSA executed on single core. In addition to this, the execution time of PRSA1 also increased when the key size increased from 128 bits to 2048 bits because the computations are distributed to all the available cores and PRSA1 started performing better and better due to its optimal parallel behavior.

Table 4.6: Speedup obtained for Test case Set 1

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size (in Bits)</th>
<th>Encryption Speed up</th>
<th>Decryption Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>128</td>
<td>2.42</td>
<td>4.10</td>
</tr>
<tr>
<td>2.</td>
<td>256</td>
<td>2.83</td>
<td>4.12</td>
</tr>
<tr>
<td>3.</td>
<td>512</td>
<td>3.66</td>
<td>4.51</td>
</tr>
<tr>
<td>4.</td>
<td>768</td>
<td>4.08</td>
<td>4.69</td>
</tr>
<tr>
<td>5.</td>
<td>1024</td>
<td>4.59</td>
<td>5.23</td>
</tr>
<tr>
<td>6.</td>
<td>1280</td>
<td>4.84</td>
<td>5.26</td>
</tr>
<tr>
<td>7.</td>
<td>1536</td>
<td>5.09</td>
<td>5.50</td>
</tr>
<tr>
<td>8.</td>
<td>1792</td>
<td>5.26</td>
<td>5.27</td>
</tr>
<tr>
<td>9.</td>
<td>2048</td>
<td>5.51</td>
<td>5.25</td>
</tr>
</tbody>
</table>
The speedups obtained while testing PRSA1 on dual quad core (8 cores) based on Test case set 2 are given in Table 4.7. It is evident through the results that the speedup of approximately 5X has been obtained when PRSA1 was executed on 8 core machine as compared to the serial version of the RSA executed on single core. In addition to this, the execution time of PRSA1 also increased when the message size increased from 1 KB to 10 KB because the computations are distributed to all the available cores and PRSA1 started performing better due to its optimal parallel behavior.

Table 4.7: Speedup obtained for Test case Set 2

<table>
<thead>
<tr>
<th>S. No</th>
<th>Message Size</th>
<th>Encryption Speed up</th>
<th>Decryption Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1 KB</td>
<td>2.90</td>
<td>4.17</td>
</tr>
<tr>
<td>2.</td>
<td>2 KB</td>
<td>3.55</td>
<td>4.36</td>
</tr>
<tr>
<td>3.</td>
<td>3 KB</td>
<td>3.73</td>
<td>4.42</td>
</tr>
<tr>
<td>4.</td>
<td>4 KB</td>
<td>3.74</td>
<td>4.48</td>
</tr>
<tr>
<td>5.</td>
<td>5 KB</td>
<td>4.02</td>
<td>4.53</td>
</tr>
<tr>
<td>6.</td>
<td>6 KB</td>
<td>4.37</td>
<td>4.62</td>
</tr>
<tr>
<td>7.</td>
<td>7 KB</td>
<td>4.42</td>
<td>4.65</td>
</tr>
<tr>
<td>8.</td>
<td>8 KB</td>
<td>4.59</td>
<td>4.76</td>
</tr>
<tr>
<td>9.</td>
<td>9 KB</td>
<td>4.63</td>
<td>5.18</td>
</tr>
<tr>
<td>10.</td>
<td>10 KB</td>
<td>5.07</td>
<td>5.21</td>
</tr>
</tbody>
</table>

4.5.4 Total Overhead

The total overhead of a parallel algorithm is expressed by the difference of the total time taken by all the processing elements collectively to solve the given problem in parallel and the time taken by the best known sequential algorithm to perform that problem sequentially. The total overhead $T_O$ is calculated by Eq. 4.4.

$$T_O = \rho T_P - T_S$$  \hspace{1cm} (4.4)
The total overheads have been calculated for the largest values of both test cases by using the expression given in Eq. 4.4.

First set of overheads has been calculated for the combination of key size 2048 bits and message size of 10 KB on 8 cores which results into the value of 0.39 and 43.17 approximately for encryption and decryption respectively.

Second set of overheads has been calculated for the combination of key size 1024 bits and message size of 1000 characters on 8 cores which results into the value of 0.27 and 14.81 approximately for encryption and decryption respectively.

It is evident that the total overhead is a function of problem size and number of processing element. While calculating the total overheads of PRSA1 for various combinations of problem size and number of processors it is observed that the total overhead increases with the increase in size of the two parameters.

4.5.5 Efficiency and Scalability

As discussed in Chapter 2, the efficiency is the measure of the part of time during which the processing elements were involved performing some vital tasks. The efficiency can be calculated by taking the ratio of the speedup with respect to the number of processing elements involved during the parallel execution of the algorithm. The efficiency is calculated by the expression given by Eq. 4.5.

\[ E = \frac{S}{P} \]  

(4.5)

Where \( E \) is the efficiency, \( S \) is the speedup and \( P \) is the number of processors. The efficiency of PRSA1 was calculated for each test case set that has been used to test the algorithm as described in previous sections.

The efficiencies calculated for Test case Set 1 are given in Table 4.8 and 4.9 for encryption and decryption respectively.
Table 4.8: Efficiency of Encryption for Test case Set 1

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size</th>
<th>2 cores</th>
<th>4 cores</th>
<th>6 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>128</td>
<td>0.60</td>
<td>0.53</td>
<td>0.37</td>
<td>0.30</td>
</tr>
<tr>
<td>2.</td>
<td>256</td>
<td>0.60</td>
<td>0.56</td>
<td>0.48</td>
<td>0.35</td>
</tr>
<tr>
<td>3.</td>
<td>512</td>
<td>0.63</td>
<td>0.57</td>
<td>0.49</td>
<td>0.46</td>
</tr>
<tr>
<td>4.</td>
<td>768</td>
<td>0.71</td>
<td>0.70</td>
<td>0.62</td>
<td>0.51</td>
</tr>
<tr>
<td>5.</td>
<td>1024</td>
<td>0.76</td>
<td>0.75</td>
<td>0.62</td>
<td>0.57</td>
</tr>
<tr>
<td>6.</td>
<td>1280</td>
<td>0.79</td>
<td>0.70</td>
<td>0.69</td>
<td>0.61</td>
</tr>
<tr>
<td>7.</td>
<td>1536</td>
<td>0.79</td>
<td>0.72</td>
<td>0.69</td>
<td>0.64</td>
</tr>
<tr>
<td>8.</td>
<td>1792</td>
<td>0.80</td>
<td>0.77</td>
<td>0.71</td>
<td>0.66</td>
</tr>
<tr>
<td>9.</td>
<td>2048</td>
<td>0.82</td>
<td>0.80</td>
<td>0.72</td>
<td>0.69</td>
</tr>
</tbody>
</table>

Table 4.9: Efficiency of Decryption for Test case Set 1

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size</th>
<th>2 cores</th>
<th>4 cores</th>
<th>6 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>128</td>
<td>1.37</td>
<td>2.55</td>
<td>3.33</td>
<td>4.10</td>
</tr>
<tr>
<td>2.</td>
<td>256</td>
<td>1.38</td>
<td>2.72</td>
<td>3.61</td>
<td>4.12</td>
</tr>
<tr>
<td>3.</td>
<td>512</td>
<td>1.39</td>
<td>2.69</td>
<td>3.78</td>
<td>4.51</td>
</tr>
<tr>
<td>4.</td>
<td>768</td>
<td>1.49</td>
<td>2.85</td>
<td>4.03</td>
<td>4.69</td>
</tr>
<tr>
<td>5.</td>
<td>1024</td>
<td>1.51</td>
<td>2.89</td>
<td>4.27</td>
<td>5.23</td>
</tr>
<tr>
<td>6.</td>
<td>1280</td>
<td>1.51</td>
<td>2.98</td>
<td>4.32</td>
<td>5.26</td>
</tr>
<tr>
<td>7.</td>
<td>1536</td>
<td>1.53</td>
<td>2.95</td>
<td>4.45</td>
<td>5.50</td>
</tr>
<tr>
<td>8.</td>
<td>1792</td>
<td>1.60</td>
<td>3.09</td>
<td>4.77</td>
<td>5.27</td>
</tr>
<tr>
<td>9.</td>
<td>2048</td>
<td>1.67</td>
<td>3.27</td>
<td>4.80</td>
<td>5.25</td>
</tr>
</tbody>
</table>

The efficiencies calculated for Test case Set 2 are given in table 4.10 and table 4.11 for encryption and decryption respectively.
Table 4.10: Efficiency of Encryption for Test case Set 2

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size</th>
<th>2 cores</th>
<th>4 cores</th>
<th>6 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1 KB</td>
<td>0.56</td>
<td>0.46</td>
<td>0.43</td>
<td>0.36</td>
</tr>
<tr>
<td>2.</td>
<td>2 KB</td>
<td>0.58</td>
<td>0.52</td>
<td>0.49</td>
<td>0.44</td>
</tr>
<tr>
<td>3.</td>
<td>3 KB</td>
<td>0.60</td>
<td>0.54</td>
<td>0.52</td>
<td>0.47</td>
</tr>
<tr>
<td>4.</td>
<td>4 KB</td>
<td>0.63</td>
<td>0.54</td>
<td>0.52</td>
<td>0.47</td>
</tr>
<tr>
<td>5.</td>
<td>5 KB</td>
<td>0.68</td>
<td>0.58</td>
<td>0.56</td>
<td>0.50</td>
</tr>
<tr>
<td>6.</td>
<td>6 KB</td>
<td>0.71</td>
<td>0.64</td>
<td>0.61</td>
<td>0.55</td>
</tr>
<tr>
<td>7.</td>
<td>7 KB</td>
<td>0.71</td>
<td>0.64</td>
<td>0.61</td>
<td>0.55</td>
</tr>
<tr>
<td>8.</td>
<td>8 KB</td>
<td>0.72</td>
<td>0.67</td>
<td>0.64</td>
<td>0.57</td>
</tr>
<tr>
<td>9.</td>
<td>9 KB</td>
<td>0.73</td>
<td>0.67</td>
<td>0.64</td>
<td>0.58</td>
</tr>
<tr>
<td>10.</td>
<td>10 KB</td>
<td>0.76</td>
<td>0.74</td>
<td>0.70</td>
<td>0.63</td>
</tr>
</tbody>
</table>

Table 4.11: Efficiency of Decryption for Test case Set 2

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size</th>
<th>2 cores</th>
<th>4 cores</th>
<th>6 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1 KB</td>
<td>0.59</td>
<td>0.57</td>
<td>0.56</td>
<td>0.52</td>
</tr>
<tr>
<td>2.</td>
<td>2 KB</td>
<td>0.60</td>
<td>0.59</td>
<td>0.57</td>
<td>0.55</td>
</tr>
<tr>
<td>3.</td>
<td>3 KB</td>
<td>0.62</td>
<td>0.60</td>
<td>0.59</td>
<td>0.55</td>
</tr>
<tr>
<td>4.</td>
<td>4 KB</td>
<td>0.63</td>
<td>0.63</td>
<td>0.60</td>
<td>0.56</td>
</tr>
<tr>
<td>5.</td>
<td>5 KB</td>
<td>0.65</td>
<td>0.65</td>
<td>0.60</td>
<td>0.57</td>
</tr>
<tr>
<td>6.</td>
<td>6 KB</td>
<td>0.68</td>
<td>0.66</td>
<td>0.65</td>
<td>0.58</td>
</tr>
<tr>
<td>7.</td>
<td>7 KB</td>
<td>0.71</td>
<td>0.66</td>
<td>0.64</td>
<td>0.58</td>
</tr>
<tr>
<td>8.</td>
<td>8 KB</td>
<td>0.73</td>
<td>0.69</td>
<td>0.68</td>
<td>0.60</td>
</tr>
<tr>
<td>9.</td>
<td>9 KB</td>
<td>0.76</td>
<td>0.71</td>
<td>0.69</td>
<td>0.65</td>
</tr>
<tr>
<td>10.</td>
<td>10 KB</td>
<td>0.77</td>
<td>0.75</td>
<td>0.69</td>
<td>0.65</td>
</tr>
</tbody>
</table>
It can be seen in the Tables 4.9, 4.10, 4.11 and 4.12 that the efficiency decreases when the number of processing cores increases while keeping problem size constant. It can be seen in fig. 4.8 that the efficiency increases when the problem size increases while keeping the number of processing element constant.

Fig. 4.8: Efficiencies of PRSA1 on 8 cores for increasing problem size

After observing the values given in above tables, it is evident that the similar efficiency can be obtained by increasing both the problem size and the number of processing elements. For example, the efficiency of encrypting 4 KB data on 2 cores is 0.63, 6 KB on 4 cores 0.64, 8 KB on 6 cores is 0.64 and 10KB on 8 cores is 0.64. Therefore increasing the data size to 10 KB and number of cores to 8 results in the same efficiency as 4 KB data on 2 cores. This ability of maintaining the efficiency by increasing the data size as well as number of processing elements simultaneously is called the scalability. Therefore, the PRSA1 algorithm has exhibited the scalability feature and it is capable of utilizing the increasing processing resources effectively.

4.6.5 Complexity

The complexity of a given algorithm describes how the runtime of the algorithm grows with increasing size of its inputs.
But in the case of any parallel algorithm it is the function of both the input size and the number of processing elements. To obtain the time complexity, it is important to design a parallel algorithm in such a manner that it can utilize the number of processing elements in an efficient manner and distribute the input data among the processing elements optimally.

Suppose PRSA1 has number of processing elements, $p=4$ and the exponent $n=4$ to encrypt the data. In such a case, PRSA1 is required to perform four multiplications. At each step, one multiplication operation is to be performed in addition to one communication operation. Suppose multiplication operation takes constant time $T_m$ and communication operation takes $T_s+T_w$ (where $T_s$ is the serial time and $T_w$ is the communication time for one word) therefore the complexity of the PRSA1 can be expressed as Eq. 4.6.

$$T_m = \theta (\log n) \quad (4.6)$$

But for the cases when the exponent increases hugely as compared to the number of processing elements it is important to distribute the computations efficiently among the available number of processing elements. For example if $p=4$ but the $n=16$ then in such cases it is important to exploit granularity efficiently to control the complexity of the algorithm. To perform such computations coarse-grained granularity has been used for PRSA1 which is demonstrated in the Fig. 4.9.

It is evident in the Fig. 4.8 that, at first step each processing core multiplies $n/p$ numbers locally in time $\Theta(n/p)$. Same operations were performed on each processing element locally and then the problem is reduced to multiply $p$ partial products on $p$ processing elements as demonstrated in the Fig. 4.8 which takes $\Theta(\log p)$ time. Therefore the parallel runtime of the PRSA1 can be given by the Eq. 4.7.

$$T_p = \theta \left( \frac{n}{p} + \log p \right) \quad (4.7)$$
Where \( n/p \) is a constant therefore the time complexity of PRSA1 can be given by Eq. 4.8.

\[
T_p = \Theta (\log p)
\]  

(4.8)

\[\Theta (n + p \log p)
\]

(4.9)

Fig. 4.9: Computations using Coarse Grained Granularity for PRSA1

### 4.6.6 Cost Optimality

The cost of the parallel algorithm is the cost of solving it on a parallel platform. The execution time of best known sequential algorithm is considered as cost of algorithm for single processing element. A parallel algorithm is considered to be cost-optimal, if the cost of solving that problem in parallel has same asymptotic growth as a function of the input size, is same as the cost of the sequential algorithm. It has already been shown in the last subsection that the parallel runtime of the PRSA1 is –

\[
T_p = \Theta \left(\frac{n}{p} + \log p\right)
\]

Therefore the cost of the PRSA1 is given in Eq. 4.9.

\[\Theta (n + p \log p)
\]  

(4.9)

It is observed during the experiments performed on PRSA1 that as long as the input \( n \) is \( \Omega (p \log p) \), the resulting cost is \( \Theta (n) \) which is same as the
serial runtime. Therefore PRSA1 is cost-optimal as compared to its sequential counterpart.

4.7 Conclusion

The PRSA1 algorithm has been designed by using instruction level parallelism. The experimental results had shown that the PRSA1 gave the encouraging results while executing. It was based on repeated square-and-multiply method which itself is a memory efficient method and capable of ensuring optimum use of memory. In addition to this the parallel design of the PRSA1 algorithm supports in the faster execution of the algorithm.

The PRSA1 algorithm was implemented using OpenMP in combination with GCC infrastructure and GNU’s MP Library. The programs used in the experiments were executed in dual-quad core environment. The algorithm was tested using two distinct large data sets to prove the efficiency of PRSA1. During experimentation PRSA1 had shown the speedup of approximately 5x as compared to the sequential RSA therefore it is more efficient in terms of time and memory as compared to its sequential counterpart.

Therefore, a faster version of the RSA algorithm can be implemented on multi-core machines if implemented parallel. OpenMP in combination with GCC infrastructure allows implementation of PRS1 algorithm that decreases the execution time and improves the performance in terms of time.
CHAPTER 5
PRSA2
PARALLEL RSA
ALGORITHM
BASED ON
SYMMETRIC MULTIPROCESSOR
ARCHITECTURE
Chapter 5

PRSA 2 – PARALLEL RSA ALGORITHM
BASED ON SYMMETRIC MULTIPROCESSOR
ARCHITECTURE

This chapter addresses objectives 4 and 5 of this research work which include the enhancement of efficient parallel RSA algorithm capable of executing the RSA algorithm with faster speed and optimum memory utilization. In addition to this, the redesigned parallel algorithm should also be tested using large data sets to prove the efficient performance of the algorithm. The chapter describes the PRSA2 which has been proposed during this research. The PRSA2 uses multi-core Symmetric Multiprocessor Architecture (SMP) to make the optimum use of available resources. It has been designed by incorporating PKCS#1 provided by RSA laboratories for the extra armoring of the encryption and decryption process.

5.1 Introduction

As described in earlier chapters the foundation of RSA (Barrett, 1987) algorithm is in modular arithmetic which is the combination of modular exponentiation (Viot et al., 2008) to be performed on very large integers (Mclvor et al., 2003). Due to this fact the RSA algorithm become compute intensive and takes long run time to execute (Almasi andA, 1989). This research is based on finding the efficient parallel RSA algorithm capable of executing compute intensive parts at a faster pace. In addition to this the RSA algorithm takes longer runtime during the decryption process especially because it involves large private keys (Hoffstein et al., 1998). Due to the usage of large private keys the intermediate plain text becomes very large and complicates the computation process to a larger extent. Therefore, while designing the new version of parallel RSA algorithm the objective was to increase the speed of RSA decryption process.
To achieve this objective PRSA2 algorithm has been designed which is based on Symmetric Multiprocessor Architecture (SMP) (Kumar et al., 1994, Flynn, 1972, Jones and O'connell, 2002). The process of PRSA2 algorithm is based on PKCS#1 scheme proposed by RSA laboratories in order to provide extra armoring to the encryption and decryption process. The PKCS#1 is a padding scheme that is used during the encryption and decryption processes to provide extra security to the message.

To test the strength of PRSA2 algorithm it has been implemented using GNUs MP Library (GMP) in collaboration with GCC infrastructure on Linux Platform. GMP Library (Granlund, 1996) is used to handle large integers efficiently while computing the cipher text because GCC infrastructure cannot handle such large integers using its basic data type. Therefore to handle large integers it is required to either design a data structure to accommodate large integers or to use any third party library. But the objective of this research was to design the parallel algorithm therefore, instead of designing new data structure, the GMP library have been used.

For the parallel implementation, the OpenMP (Chandra, 2001) API has been used which is an explicit parallel programming model that support shared multiprocessor architecture. The OpenMP API has been chosen for the parallel implementation (Valiant, 1990) because it’s an Open Source tool and is available as a part of GCC infrastructure.

During the experiments the PRSA2 algorithm has been tested using large test case sets. These test cases have been divided into three groups that are being described in the results section. The results obtained during the experiments are promising and have shown approximately 7.5X speed up as compared to the sequential counterpart.

### 5.2 PRSA 2 - Parallel RSA Algorithm

The conventional RSA algorithm takes a lot of time to perform encryption and specially decryption routines because of the involvement of very large keys operating on large size integers. Due to this fact, it is a
challenge in front of today’s researchers to increase the speed of RSA algorithm without compromising its security. In this research work the concepts of parallel programming have been used to redesign the RSA algorithm in order to increase the speed of encryption and decryption (decipherment) routines. While redesigning the RSA algorithm, the main focus was on the optimum utilization of the power of multiple cores available in the target machine.

The redesigned Parallel RSA algorithm is divided into three sub-algorithms – the Key Generation Algorithm, the Encryption Algorithm and the Decryption Algorithm. The Key Generation part is sequential whereas the Encryption and Decryption parts have been redesigned as parallel algorithms. Refer Appendix B for code snippets of PRSA2. The mechanism of the PRSA2 is demonstrated in Fig. 5.1.
The three parts of PRSA2 algorithm are described in next three subsections.

5.2.1 Key Generation

The key generation in Parallel RSA algorithm is a step-wise process which is given below:

1. The first step is to select a positive integer e as the public key. Usually 65537 is taken as e.

2. Now, randomly select two distinct odd prime numbers p and q in such a manner that there should not be the common divisors of e and (p-1) and e and (q-1).

3. Then n is taken as the product of p and q, given in Eq. 5.1.

   \[ n = p \cdot n \]  

\[ 2^8 (l - 1) \leq n < 2^8 (8l) \]  

5. The length of the modulus, l, must be at least 12 octets to accommodate the block format as described in PKCS#1.

Here, the (e, n) are taken as public key and (d, n) are taken as private key.

5.2.2 Parallel RSA Encryption

The encryption process has been implemented in parallel where the message stored in file or some other form is first transferred to a vector. Then the data stored in vector is divided into chunks of data which is converted into the blocks of data. These blocks are assigned to threads which are in turn assigned to multiple cores.
Algorithm 5.1: PRSA2: Encryption Algorithm

Procedure: encryption_smp
Model: Data Parallel Model based on symmetric multiprocessor
Input: publicExponent, plain text, modulus
Output: Result
Declare:
Global: N, Number_of_Cores, Cipher, time
Local: loop variables
Generate Public Key: modulus n and publicExponent e
Generate Private Key: modulus n and privateExponent e
Read the Number_Of_Cores available on Target_Machine
Declare Number_Of_Threads to be executed parallel on each Core
Parbegin
Decompose the message into ‘N’ number of Chunk_Size in Octet Format
Declare No_OfParallelSections equal to N / Number_Of_Threads
Assign each data of Chunk_Size to each Thread of the each Parallel Section
Declare Shared Variables to measure Time and Assign them to Shared Memory
Declare Loop Variables and assign them to the Local Memory of Each threads
for (Processor i), i:=1 to No_OfParallelSections do In Parallel
Format ENCRYPT BLOCK as described by PKCS#1
    EB = 00 || BT || PS || 00 || D
Convert Padded String Message Block to Integer
Encrypt Integer Block
    C = Me % n
Convert Encrypted Integer Block to String to get Final Cipher
end In Parallel
Parend

The Parallel RSA Encryption comprises of 4 steps: formatting of encryption block, string to integer conversion, calculation of cipher text, integer to octet string conversion. The RSA encryption function takes the data D, modulus n, the public key e as an input. The encryption function
outputs an string ED n octet format, the encrypted message. The length of data D should not be more than block size-11 octets, which guarantees that the length of padding string will be at least eight octets. The PRSA2 Encryption algorithm developed and proposed during this research is given as Algorithm 5.1.

5.2.2.1 Encryption Block Formatting

The EB, encryption block is formatted as an octet and EB takes the form shown in Eq. 5.3.

\[
EB = 00 || BT || PS || 00 || D
\]  

(5.3)

Where the BT is a block type and it should be single octet comprising of the value 00, 01 or 02. If it is a private key operation then it should be either 00 or 01 but if it is a public key operation then it should be 02. The PS is a padding string and it consists of block size-3-||D|| non-zero, randomly generated bytes. The data D must be less than block size -11 that guarantees for having at least 8 bytes of PS. The overall formatting makes the EB equal to block size.

5.2.2.2 Octet String to Integer Conversion

After formatting the EB, it is converted to an integer x (the integer encryption block). For this each character is converted into the respective sequence number of the alphabets. For example, if we have data block consisting of string “RSA” it will be converted as following procedure – R into 17 because it comes at 17th position in alphabets. Similarly S is converted into 18 and A into 00.

\[
\begin{array}{ccc}
R & S & A \\
17 & 18 & 00
\end{array}
\]

Then all pairs of digits concatenated to get a single number x. For the above example the x will be - 171800.
5.2.2.3 Calculation of Cipher Text

In this step, the cipher text is calculated by raising to the power $e$ to integer $x$ modulo $n$ to give cipher text an integer $c$. The procedure to calculate the cipher text is given in Eq. 5.4.

$$c = x^e \mod n$$  \hspace{1cm} (5.4)

Where, $0 \leq c < n$.

5.2.2.4 Integer to Octet string Conversion

Now the integer $c$ should be converted to an octet string encrypted data, ED of length block size. The reverse of the procedure given in step 2 is used here to get back the string of characters.

The Octet string obtained after all steps is published as Cipher Text.

5.2.3 Parallel RSA Decryption

The RSA decryption is also implemented in parallel. Here the Cipher text is divided into blocks of data which in turn is assigned to the threads. These threads are later assigned to the multiple cores. The decryption function takes an octet string encrypted data ED, a modulus $n$ and a private key $d$ (exponent) as the input. The function outputs the data string D. The RSA decryption comprises of 4 steps as described in next four sections. The PRSA2 Decryption algorithm developed and proposed during this research is given as Algorithm 5.2.

5.2.3.1 Octet String to Integer Conversion

First the encrypted data ED is converted to an integer. The same procedure is applied here to convert the string into the integer $c$ which is described in the subsection – “Octet String to Integer Conversion” of the Parallel RSA encryption.

5.2.3.2 Calculation of Message

In this step the plain text is calculated by raising to the power $d$ to integer $c$ modulo $n$ to give plain text an integer $m$. The procedure to calculate the cipher text is given in Eq. 5.5.
\[ m = c^d \mod n \]  \hspace{1cm} (5.5)

Where, \( 0 \leq c < n \).

Algorithm 5.2: PRSA2: Decryption Algorithm

```plaintext
Procedure: decryption_smp
Model: Data Parallel Model based on symmetric multiprocessor
Input: publicExponent, plain text, modulus
Output: Result
Declare:
    Global: N, Number_of_Cores, Cipher, time
    Local: loop variables
Generate Public Key: modulus n and publicExponent e
Generate Private Key: modulus n and privateExponent e
Read the Number_Of_Cores available on Target_Machine
Declare Number_Of_Threads to be executed parallel on each Core
Parbegin
    Decompose the message into ‘N’ number of Chunk_Size in Octet Format
    Declare No_Of_Parallel_Sections equal to N / Number_Of_Threads
    Assign each data of Chunk_Size to each Thread of the each Parallel Section
    Declare Shared Variables to measure Time and Assign them to Shared Memory
    Declare Loop Variables and assign them to the Local Memory of Each threads
    for (Processor i), i:=1 to No_Of_Parallel_Sections do In Parallel
        Format ENCRYPT BLOCK as described by PKCS#1
        \[ EB = 00 \ || \ BT \ || \ PS \ || \ 00 \ || \ D \]
        Convert Padded String Message Block to Integer
        Encrypt Integer Block
        \[ C = Me \mod n \]
        Convert Encrypted Integer Block to String to get Final Cipher
    end In Parallel
Parend
```
5.2.3.3 Integer to Octet string Conversion

Now the integer m should be converted to an octet string plain text, EB of length block size.

5.2.3.4 Encryption Block Parsing

In this step, the EB has been parsed according to the Eq. 5.1. The components in which it is split are - a block type BT, a padding string PS, and the data D according to the Eq. 5.1.

5.3 Methodology

The proposed Parallel RSA algorithm is based on Symmetric Multiprocessor (SMP) Architecture. In SMP architecture multiple identical processors or cores are connected with single shared memory and are controlled by single Operating System. While designing the PRSA2, the major concern was to utilize the power of all the available cores of the processor. Therefore new algorithm was designed as scalable and thus it adjusts itself according to the number of cores available in the processing system.

The next biggest challenge was to not compromise the security of conventional RSA therefore PKCS#1 (provided by RSA Laboratories) scheme was used. The whole data to be encrypted or decrypted was first decomposed into chunks of the size equal to the BlockSize-11 which is one of the security conditions of PKCS#1. For the data decomposition and distribution in the form of chunks; data decomposition technique of parallel models were used. The granularity selected for the data distribution was coarse grained, because fine grained decomposition increases the overheads for the parallel processes. Now each chunk was converted into the fixed size block of the EB (Encryption Block) by padding it using PKCS#1.

During execution of algorithm, multiple threads were generated in shared memory environment so that all threads could share the same instruction set. All blocks of data were equally distributed among the threads to
encrypt or decrypt the data. Now these threads were assigned to different cores of the processor in the shared memory environment using static mapping technique.

The private thread dependent variables were allocated to local cache memory of the cores or in other words, each thread was assigned with their personal copies of the private variables. The global variables were allocated to shared memory, which was equally shared among the threads. Each thread of the operation executes the same set of instructions to operate upon the data to perform encryption and decryption. Once the encryption and decryption was performed the data is again concatenated to get the final cipher or plain message.

The PRSA2 was tested using the OpenMP API on the GCC Infrastructure on Linux platform. The OpenMP API is an explicit programming model that allows the programmer to design and execute the parallel program in shared memory environment. It is based on explicit multithreading model in which programmer can generate multiple threads that can be allocated to the available multiple cores of the computer system to perform parallel programming.

The OpenMP API allows testing programs based on data level parallelism as well as task level parallelism. The proposed PRSA2 was based on data level parallelism. It is tested using massive data organized into two separate test cases – Test case Set 1 and Test case Set 2. The results obtained during the experiments are discussed in results section.

5.4 Techniques used for the Parallelization

In order to parallelize the algorithms in this research various parallel programming techniques are used which are discussed next.

5.4.1 Data Decomposition Technique

Data decomposition techniques are powerful techniques for deriving concurrency in the parallel programs. It is used in this research to divide the large input data into multiple chunks of small data. This technique is
performed in two steps – firstly the data on which the computation is to be performed is divided and secondly these data partitions are converted into the tasks.

5.4.2 Granularity

The number and size of the tasks according to which the problem is decomposed is called the granularity of the decomposition. In this research coarse grained granularity is used in which decomposition is performed in order to obtain small number of large size tasks.

5.4.3 Static Mapping

Static mapping is used in this project which is usually used to map the tasks generated by data partitioning to the available cores. In the first parallel RSA algorithm array distribution scheme is used whereas in the second approach, block distribution scheme is used.

5.4.4 Memory Allocation

In this research work, during the parallel implementations of various algorithms use two types of memory allocation – memory allocated to thread dependent variables and memory allocated to the variables which are shared by all the threads equally. The thread dependent variables are allocated memory in the cache memory whereas the global variables are allocated global shared memory area provided by OpenMP.

5.4.5 Scalability

The scalability refers to the system’s or program’s capability to adapt itself to the increasing number of processing resources. This technique is used in the parallelization to large scale in order to utilize all the available processing elements or cores available in the system.

5.5 Platform used for Experiments

The configuration of the computer system which was used during the experimentations is described in the Table 5.1.
Table 5.1: Configuration of Computer used for Experiments

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Processor</td>
<td>AMD FX (tm) - 8120 Eight-Core Processor</td>
</tr>
<tr>
<td>2.</td>
<td>Frequency</td>
<td>3.10 GHz.</td>
</tr>
<tr>
<td>3.</td>
<td>RAM</td>
<td>8.00 GB</td>
</tr>
<tr>
<td>4.</td>
<td>System Type</td>
<td>64 Bit Operating System</td>
</tr>
<tr>
<td>5.</td>
<td>Environment</td>
<td>Ubuntu Linux</td>
</tr>
<tr>
<td>6.</td>
<td>Platform</td>
<td>GCC Infrastructure</td>
</tr>
<tr>
<td>7.</td>
<td>Big Integer Library</td>
<td>GNUs MP Library</td>
</tr>
</tbody>
</table>

5.6 Experimental Results

The OpenMP is an API based on shared memory parallel architecture that supports the programming languages like C, C++ and FORTRAN.

The experiments were performed to test the PRSA2 algorithm extensively by using two distinct sets of data. To find the improvement in terms of time and energy the parallel RSA was executed on single core, 2 cores, 4 cores and 8 cores processor. The time recorded in terms of two categories – encryption time and decryption time. Experiments related to each Test case was performed exactly 25 times and final reading was taken as the average of it.

The three test cases used for the testing were – first by varying the key size from 1024 to 3072, second by varying message size from 1 MB to 10 MB and third for large data sets varying from 250 MB to 1 GB. The data sets which were used to test PRSA2 algorithm and their respective results are presented in the next three subsections.

5.6.1 Test case Set 1

In the first test case set, experiments were performed by varying the key size from 1024 bits to 3072 bits. The data taken for each key size is 10
Find the exact difference between the execution time of encryption and decryption of sequential RSA and PRSA2. Two different sets of time were taken as the results – encryption time and decryption time during experiments. The encryption time recorded during the experiments is given in Table 5.2 and the decryption time is given in Table 5.3. The results given in Table 5.2 and Table 5.3 shows the improvements and performance comparison in parallel runtimes using 2 cores, 4 cores and 8 cores with respect to the sequential implementation using a single core for Test case set 1.

Table 5.2: Encryption Time for Test case Set 1

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Sequential Code</th>
<th>Parallel Code (2 Cores)</th>
<th>Parallel Code (4 Cores)</th>
<th>Parallel Code (8 Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA1024</td>
<td>3.50148</td>
<td>2.67712</td>
<td>1.48621</td>
<td>0.79095</td>
</tr>
<tr>
<td>RSA1280</td>
<td>4.01448</td>
<td>2.92239</td>
<td>1.62486</td>
<td>0.89972</td>
</tr>
<tr>
<td>RSA1536</td>
<td>4.67206</td>
<td>3.25679</td>
<td>1.82430</td>
<td>1.02755</td>
</tr>
<tr>
<td>RSA1792</td>
<td>6.48508</td>
<td>4.34537</td>
<td>2.35122</td>
<td>1.25756</td>
</tr>
<tr>
<td>RSA2048</td>
<td>7.11674</td>
<td>4.64793</td>
<td>2.45177</td>
<td>1.36929</td>
</tr>
<tr>
<td>RSA2304</td>
<td>7.48441</td>
<td>4.86540</td>
<td>2.50121</td>
<td>1.42530</td>
</tr>
<tr>
<td>RSA2560</td>
<td>8.49329</td>
<td>5.34537</td>
<td>2.80038</td>
<td>1.48498</td>
</tr>
<tr>
<td>RSA2816</td>
<td>9.67380</td>
<td>6.01538</td>
<td>3.10368</td>
<td>1.66914</td>
</tr>
<tr>
<td>RSA3072</td>
<td>11.72997</td>
<td>6.84390</td>
<td>3.56367</td>
<td>1.91154</td>
</tr>
</tbody>
</table>

It may be observed in the graphs shown in fig. 5.2 and 5.3 that as and when the size of the key is increased from 1024 bits to 3072 bits there is no drastic change reflected in the encryption execution time because of the involvement of small encryption key. But as far as the execution time involved in decryption routine is concerned, the speedup involved increases gradually with the increase in the key size as shown in fig. 5.3.
Fig. 5.2: Encryption Execution Time on 2, 4, 8 cores for Test case Set 1

Table 5.3: Decryption Time for Test case Set 1

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Sequential Code</th>
<th>Parallel Code (2 Cores)</th>
<th>Parallel Code (4 Cores)</th>
<th>Parallel Code (8 Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA1024</td>
<td>20.45379</td>
<td>16.62843</td>
<td>8.76932</td>
<td>4.50942</td>
</tr>
<tr>
<td>RSA1280</td>
<td>27.92371</td>
<td>22.23780</td>
<td>11.59470</td>
<td>5.83702</td>
</tr>
<tr>
<td>RSA1536</td>
<td>37.95357</td>
<td>28.68320</td>
<td>14.93382</td>
<td>7.37368</td>
</tr>
<tr>
<td>RSA1792</td>
<td>54.69334</td>
<td>39.94837</td>
<td>20.94375</td>
<td>10.55144</td>
</tr>
<tr>
<td>RSA2048</td>
<td>64.40224</td>
<td>46.38746</td>
<td>23.93200</td>
<td>11.95679</td>
</tr>
<tr>
<td>RSA2304</td>
<td>74.29724</td>
<td>52.47483</td>
<td>26.45610</td>
<td>13.62980</td>
</tr>
<tr>
<td>RSA2560</td>
<td>83.47953</td>
<td>58.84638</td>
<td>29.97280</td>
<td>15.09530</td>
</tr>
<tr>
<td>RSA2816</td>
<td>94.29724</td>
<td>65.84370</td>
<td>33.24979</td>
<td>16.73816</td>
</tr>
<tr>
<td>RSA3072</td>
<td>103.78470</td>
<td>71.47830</td>
<td>35.47820</td>
<td>17.89825</td>
</tr>
</tbody>
</table>
It can also be observed in the graphs that execution time decreases when the number of processing elements increases from 2 to 8. Therefore, it is obvious that PRSA2 shows better time efficiency on more number of processing elements.

Fig. 5.3: Decryption Execution Time on 2, 4, 8 cores for Test case Set 1

5.6.2 Test Case Set 2

For the Test Case Set 2 the experiments were performed for the message size varying from 1 MB to 10 MB. The fixed key size of 2048 bits was taken for all the cases. The encryption time and decryption time obtained from Test case Set 2 are given in Table 5.5 and 5.6 respectively that shows the performance comparison between sequential code on single core and PRSA2 on 2, 4 and 8 cores in terms of the execution time involved. In the results shown in Table 5.4 and Table 5.5 it can be seen that as and when the data size is increased from 1 MB to 10 MB for the encryption or decryption the speedup increases gradually in the presence of key size as large as 2048 bits.
Table 5.4: Encryption Time for Test case Set 2

<table>
<thead>
<tr>
<th>No of Characters</th>
<th>Sequential Code</th>
<th>Parallel Code (2 Cores)</th>
<th>Parallel Code (4 Cores)</th>
<th>Parallel Code (8 Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>0.70553</td>
<td>0.5412486</td>
<td>0.27804</td>
<td>0.14399</td>
</tr>
<tr>
<td>2 MB</td>
<td>1.42126</td>
<td>1.05179572</td>
<td>0.52948</td>
<td>0.27801</td>
</tr>
<tr>
<td>4 MB</td>
<td>2.8342</td>
<td>1.98610706</td>
<td>1.03919</td>
<td>0.53747</td>
</tr>
<tr>
<td>6 MB</td>
<td>4.26249</td>
<td>2.85447216</td>
<td>1.51878</td>
<td>0.75133</td>
</tr>
<tr>
<td>8 MB</td>
<td>5.67591</td>
<td>3.75559448</td>
<td>1.95308</td>
<td>0.99119</td>
</tr>
<tr>
<td>10 MB</td>
<td>7.12236</td>
<td>4.45807518</td>
<td>2.35274</td>
<td>1.1879</td>
</tr>
</tbody>
</table>

Fig. 5.4: Encryption Execution Time on 2, 4, 8 cores for Test Case Set 2
Table 5.5: Decryption Time for Test case Set 2

<table>
<thead>
<tr>
<th>No of Characters</th>
<th>Sequential Code</th>
<th>Parallel Code (2 Cores)</th>
<th>Parallel Code (4 Cores)</th>
<th>Parallel Code (8 Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>6.74926</td>
<td>4.48625</td>
<td>2.28458</td>
<td>1.15248</td>
</tr>
<tr>
<td>2 MB</td>
<td>13.6282</td>
<td>8.71160</td>
<td>4.35790</td>
<td>2.23458</td>
</tr>
<tr>
<td>4 MB</td>
<td>25.9555</td>
<td>15.66198</td>
<td>7.99185</td>
<td>3.99656</td>
</tr>
<tr>
<td>6 MB</td>
<td>38.0431</td>
<td>22.44066</td>
<td>11.27403</td>
<td>5.63523</td>
</tr>
<tr>
<td>8 MB</td>
<td>51.5126</td>
<td>28.72662</td>
<td>14.53132</td>
<td>7.35869</td>
</tr>
<tr>
<td>10 MB</td>
<td>65.5961</td>
<td>35.41247</td>
<td>17.78355</td>
<td>8.98663</td>
</tr>
</tbody>
</table>

Fig. 5.5: Decryption Execution Time on 2, 4, 8 cores for Test Case Set 2

It may be observed in the graphs shown in fig. 5.4 and 5.5 that as and when the message is increased from 1MB bits to 10MB there is no drastic change reflected in the encryption execution time because of the involvement of small encryption key. But as far as the execution time involved in decryption routine is concerned, the speedup involved increases gradually with the increase in the message size as shown in fig. 5.5. It can also be observed in the graphs that execution time decreases when the number of processing elements increases from 2 to 8. Therefore,
it is obvious that PRSA2 shows better time efficiency on more number of processing elements.

5.6.3 Test Case Set 3

In the third test case set experiments were performed on large size data sets by varying message size from 250 MB to 1 GB. The fixed key size of 2048 bits was taken for all the cases. The encryption time and decryption time obtained from Test Case Set 3 are given in Table 5.6 and Table 5.7 respectively. The tables also show the performance comparison between sequential RSA on single core and PRSA2 on 2, 4 and 8 cores in terms of execution time involved.

Table 5.6: Encryption Time for Test case Set 3

<table>
<thead>
<tr>
<th>No of Characters</th>
<th>Sequential Code</th>
<th>Parallel Code (2 Cores)</th>
<th>Parallel Code (4 Cores)</th>
<th>Parallel Code (8 Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250 MB</td>
<td>134.1209</td>
<td>86.45486</td>
<td>44.49901</td>
<td>23.10554</td>
</tr>
<tr>
<td>500 MB</td>
<td>279.1831</td>
<td>172.9038</td>
<td>88.99527</td>
<td>45.41601</td>
</tr>
<tr>
<td>750 MB</td>
<td>462.2551</td>
<td>278.0193</td>
<td>142.2357</td>
<td>72.82904</td>
</tr>
<tr>
<td>1 GB</td>
<td>657.9938</td>
<td>389.2144</td>
<td>199.1347</td>
<td>101.6337</td>
</tr>
</tbody>
</table>

Fig. 5.6: Encryption Execution Time on 2, 4, 8 cores for Test Case Set 3
<table>
<thead>
<tr>
<th>No of Characters</th>
<th>Sequential Code</th>
<th>Parallel Code (2 Cores)</th>
<th>Parallel Code (4 Cores)</th>
<th>Parallel Code (8 Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>250 MB</td>
<td>701.7398</td>
<td>386.6279</td>
<td>199.0005</td>
<td>103.3284</td>
</tr>
<tr>
<td>500 MB</td>
<td>1592.992</td>
<td>844.6752</td>
<td>434.7625</td>
<td>221.8677</td>
</tr>
<tr>
<td>750 MB</td>
<td>2605.707</td>
<td>1370.149</td>
<td>700.9731</td>
<td>358.9198</td>
</tr>
<tr>
<td>1 GB</td>
<td>3655.292</td>
<td>1877.45</td>
<td>960.5643</td>
<td>490.2498</td>
</tr>
</tbody>
</table>

Fig. 5.7: Decryption Execution Time on 2, 4, 8 cores for Test Case Set 2

In the graphs shown in Fig. 5.6 and 5.7 it can be seen that as and when the data size is increased from 250 MB to 1 GB for the encryption or decryption the speedup increases gradually in the presence of key having size 2048 bits. It may be observed in the graphs shown in fig. 5.6 and 5.7 that as and when the message is increased from 1MB bits to 10MB the decryption routine shows better performance as compared to the encryption. It can also be observed in the graphs that execution time decreases when the number of processing elements increases from 2 to 8.
Therefore, it is observed during the experiments that the speedup increases when the number of cores increases to execute the PRSA2. Similarly PRSA2 gives encouraging results for the test cases where the key size is taken sufficiently large such as 2048 bits and the message size is increased from 1 MB to 10 MB. In such cases speedup of approximately 7X is achieved when PRSA2 is executed on 8 core machine as compared to the serial version of the RSA executed on single core. Similarly, the speedup also increased when the data size has been increased from 250 MB to 1 GB.

Hence it is shown that the proposed PRSA2 algorithm is time efficient and it is appropriate for performing encryption, decryption or digital signing in cases where it is required to perform these operations quickly depending upon the type of machine being used. The algorithm is also suitable for battery operated device as it takes less time to encrypt large data as compared to the sequential RSA.

5.7 Performance Analysis

The PRSA2 algorithm has been redesigned by exploiting data level parallelism. First the data to be encrypted or decrypted has been divided into fixed size chunks. Then these chunks have been converted into tasks which were assigned to the processing cores later to perform encryption or decryption concurrently. While designing PRSA2, the focus was on increasing the memory efficiency and speedup of algorithm as compared to its sequential counterpart. The algorithm has been implemented using OpenMP API and tested using various parameters as discussed in experimental results section. The performance of PRSA2 has been analyzed using various performance metrics and the same is mentioned in the next four subsections.

5.7.1 Speedup

Speedup is one of the most important performance metrics for parallel algorithm. It is a comparative measure of solving the problem in parallel
as compared to sequential which is calculated by taking the ratio of the execution time taken by the serial algorithm on a single processing element with respect to the execution time taken by the parallel algorithm on a parallel computing platform having \( p \) number of identical processing elements. The procedure to calculate the speedup of any parallel algorithm is given in Eq. 5.6.

\[
S = \frac{T_S}{T_P}
\]  

(5.6)

Where \( T_S \) is the execution time of sequential algorithm and \( T_P \) is the execution time of the parallel algorithm. The speedup obtained by PRSA2 for both the test cases has been calculated using the expression given in Eq. 5.6.

The speedups obtained while testing PRSA2 on dual quad core (8 cores) based on Test case set 1 are given in Table 5.8. It is evident through the results that the speedup of approximately 6X has been obtained for encryption and approximately 7.5X for decryption when PRSA2 was executed on 8 core machine as compared to the serial version of the RSA executed on single core. In addition to this the execution time of PRSA2 also increased when the key size increased from 1024 bits to 3072 bits because the computations are distributed to all the available cores and PRSA2 started performing better due to its optimal parallel behavior.

The speedups obtained while testing PRSA2 on dual quad core (8 cores) based on Test case set 2 are given in Table 5.9. It is evident through the results that the speedup of 5X approximately for encryption and 7X approximately for decryption has been obtained when PRSA2 was executed on 8 core machine as compared to the serial version of the RSA executed on single core. In addition to this the execution time of PRSA2 also increased when the message size increased from 1 MB bits to 10 MB because the computations are distributed to all the available cores and PRSA2 started performing better due to its optimal parallel behavior.
Table 5.8: Speedup obtained for Test case Set 1

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size (in Bits)</th>
<th>Encryption Speed up</th>
<th>Decryption Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>RSA1024</td>
<td>4.43</td>
<td>4.54</td>
</tr>
<tr>
<td>2.</td>
<td>RSA1280</td>
<td>4.46</td>
<td>4.78</td>
</tr>
<tr>
<td>3.</td>
<td>RSA1536</td>
<td>4.55</td>
<td>5.15</td>
</tr>
<tr>
<td>4.</td>
<td>RSA1792</td>
<td>5.16</td>
<td>5.18</td>
</tr>
<tr>
<td>5.</td>
<td>RSA2048</td>
<td>5.20</td>
<td>5.39</td>
</tr>
<tr>
<td>6.</td>
<td>RSA2304</td>
<td>5.25</td>
<td>5.45</td>
</tr>
<tr>
<td>7.</td>
<td>RSA2560</td>
<td>5.72</td>
<td>5.53</td>
</tr>
<tr>
<td>8.</td>
<td>RSA2816</td>
<td>5.80</td>
<td>5.63</td>
</tr>
<tr>
<td>9.</td>
<td>RSA3072</td>
<td>6.14</td>
<td>5.80</td>
</tr>
</tbody>
</table>

Table 5.9: Speedup obtained for Test case Set 2

<table>
<thead>
<tr>
<th>S. No</th>
<th>Message Size</th>
<th>Encryption Speed up</th>
<th>Decryption Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1 MB</td>
<td>4.90</td>
<td>5.86</td>
</tr>
<tr>
<td>2.</td>
<td>2 MB</td>
<td>5.11</td>
<td>6.10</td>
</tr>
<tr>
<td>3.</td>
<td>4 MB</td>
<td>5.27</td>
<td>6.49</td>
</tr>
<tr>
<td>4.</td>
<td>6 MB</td>
<td>5.67</td>
<td>6.75</td>
</tr>
<tr>
<td>5.</td>
<td>8 MB</td>
<td>5.73</td>
<td>7.00</td>
</tr>
<tr>
<td>6.</td>
<td>10 MB</td>
<td>6.00</td>
<td>7.30</td>
</tr>
</tbody>
</table>

5.7.2 Total Overhead

The total overhead of a parallel algorithm is expressed by taking the difference of the total time taken by all the processing elements collectively to solve the given problem in parallel and the time taken by the best known sequential algorithm to perform that problem sequentially. The total overhead $T_o$ is calculated by Eq. 5.7.

$$T_o = pT_p - T_S$$  \hspace{1cm} (5.7)
The total overheads have been calculated for the largest values of both test cases by using the expression given in Eq. 5.7.

First set of overheads has been calculated for the combination of key size 3072 bits and message size of 10 MB on 8 cores which results into the value of 3.56 and 39.40 approximately for encryption and decryption respectively.

Second set of overheads has been calculated for the combination of key size 2048 bits and message size of 10 MB characters on 8 cores which results into the value of 2.38 and 6.29 approximately for encryption and decryption respectively.

Third set of overheads has been calculated for the combination of key size 2048 bits and message size of 1 GB on 8 cores which results into the value of 155.07 and 266.70 approximately for encryption and decryption respectively.

It is evident that the total overhead is a function of problem size and number of processing element. While calculating the total overheads of the PRSA2 for various combinations of problem size and number of processors it is observed that the total overheads increases with the increase in the size of the above two parameters.

### 5.7.3 Efficiency and Scalability

The efficiency is the measure of the part of time during which the processing elements were involved performing some vital work. The efficiency can be calculated by taking the ratio of the speedup with respect to the number of processing elements involved during the parallel execution of the algorithm. The efficiency is calculated by the expression given by Eq. 5.8.

\[
E = \frac{s}{p}
\]  

(5.8)
Where $E$ is the efficiency, $S$ is the speedup and $P$ is the number of processors. The efficiency of PRSA1 has been calculated for each test case set that has been used to test the algorithm as described in previous sections.

The efficiencies calculated for Test case Set 1 are given in Table 5.10 and 5.11 for encryption and decryption respectively.

Table 5.10: Efficiency of Encryption for Test case Set 1

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>RSA1024</td>
<td>0.65</td>
<td>0.59</td>
<td>0.55</td>
</tr>
<tr>
<td>2.</td>
<td>RSA1280</td>
<td>0.69</td>
<td>0.62</td>
<td>0.56</td>
</tr>
<tr>
<td>3.</td>
<td>RSA1536</td>
<td>0.72</td>
<td>0.64</td>
<td>0.57</td>
</tr>
<tr>
<td>4.</td>
<td>RSA1792</td>
<td>0.75</td>
<td>0.69</td>
<td>0.64</td>
</tr>
<tr>
<td>5.</td>
<td>RSA2048</td>
<td>0.77</td>
<td>0.73</td>
<td>0.65</td>
</tr>
<tr>
<td>6.</td>
<td>RSA2304</td>
<td>0.77</td>
<td>0.75</td>
<td>0.66</td>
</tr>
<tr>
<td>7.</td>
<td>RSA2560</td>
<td>0.79</td>
<td>0.76</td>
<td>0.71</td>
</tr>
<tr>
<td>8.</td>
<td>RSA2816</td>
<td>0.80</td>
<td>0.78</td>
<td>0.72</td>
</tr>
<tr>
<td>9.</td>
<td>RSA3072</td>
<td>0.86</td>
<td>0.82</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Table 5.11: Efficiency of Decryption for Test case Set 1

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>RSA1024</td>
<td>0.62</td>
<td>0.58</td>
<td>0.57</td>
</tr>
<tr>
<td>2.</td>
<td>RSA1280</td>
<td>0.63</td>
<td>0.60</td>
<td>0.60</td>
</tr>
<tr>
<td>3.</td>
<td>RSA1536</td>
<td>0.66</td>
<td>0.64</td>
<td>0.64</td>
</tr>
<tr>
<td>4.</td>
<td>RSA1792</td>
<td>0.68</td>
<td>0.65</td>
<td>0.65</td>
</tr>
<tr>
<td>5.</td>
<td>RSA2048</td>
<td>0.69</td>
<td>0.67</td>
<td>0.67</td>
</tr>
<tr>
<td>6.</td>
<td>RSA2304</td>
<td>0.71</td>
<td>0.70</td>
<td>0.68</td>
</tr>
<tr>
<td>7.</td>
<td>RSA2560</td>
<td>0.71</td>
<td>0.70</td>
<td>0.69</td>
</tr>
<tr>
<td>8.</td>
<td>RSA2816</td>
<td>0.72</td>
<td>0.71</td>
<td>0.70</td>
</tr>
<tr>
<td>9.</td>
<td>RSA3072</td>
<td>0.73</td>
<td>0.73</td>
<td>0.72</td>
</tr>
</tbody>
</table>

The efficiencies calculated for Test case Set 2 are given in Tables 5.12 and 5.13 for encryption and decryption respectively.
Table 5.12: Efficiency of Encryption for Test case Set 2

<table>
<thead>
<tr>
<th>Message Size</th>
<th>2 Cores</th>
<th>4 Cores</th>
<th>8 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>0.65</td>
<td>0.63</td>
<td>0.61</td>
</tr>
<tr>
<td>2 MB</td>
<td>0.68</td>
<td>0.67</td>
<td>0.64</td>
</tr>
<tr>
<td>4 MB</td>
<td>0.71</td>
<td>0.68</td>
<td>0.66</td>
</tr>
<tr>
<td>6 MB</td>
<td>0.75</td>
<td>0.70</td>
<td>0.71</td>
</tr>
<tr>
<td>8 MB</td>
<td>0.76</td>
<td>0.73</td>
<td>0.72</td>
</tr>
<tr>
<td>10 MB</td>
<td>0.80</td>
<td>0.76</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Table 5.13: Efficiency of Decryption for Test case Set 2

<table>
<thead>
<tr>
<th>Message Size</th>
<th>2 Cores</th>
<th>4 Cores</th>
<th>8 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MB</td>
<td>0.75</td>
<td>0.74</td>
<td>0.73</td>
</tr>
<tr>
<td>2 MB</td>
<td>0.78</td>
<td>0.78</td>
<td>0.76</td>
</tr>
<tr>
<td>4 MB</td>
<td>0.83</td>
<td>0.81</td>
<td>0.81</td>
</tr>
<tr>
<td>6 MB</td>
<td>0.85</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td>8 MB</td>
<td>0.90</td>
<td>0.89</td>
<td>0.88</td>
</tr>
<tr>
<td>10 MB</td>
<td>0.93</td>
<td>0.92</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table 5.14: Efficiency of Encryption for Test Case Set 3

<table>
<thead>
<tr>
<th>Message Size</th>
<th>2 Cores</th>
<th>4 Cores</th>
<th>8 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>250 MB</td>
<td>0.78</td>
<td>0.75</td>
<td>0.73</td>
</tr>
<tr>
<td>500 MB</td>
<td>0.81</td>
<td>0.78</td>
<td>0.77</td>
</tr>
<tr>
<td>750 MB</td>
<td>0.83</td>
<td>0.81</td>
<td>0.79</td>
</tr>
<tr>
<td>1 GB</td>
<td>0.85</td>
<td>0.83</td>
<td>0.81</td>
</tr>
</tbody>
</table>
The efficiencies calculated for Test case Set 3 are given in Tables 5.14 and 5.15 for encryption and decryption respectively.

Table 5.15: Efficiency of Decryption for Test Case Set 2

<table>
<thead>
<tr>
<th>Message Size</th>
<th>2 Cores</th>
<th>4 Cores</th>
<th>8 Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>250 MB</td>
<td>0.91</td>
<td>0.88</td>
<td>0.85</td>
</tr>
<tr>
<td>500 MB</td>
<td>0.94</td>
<td>0.92</td>
<td>0.90</td>
</tr>
<tr>
<td>750 MB</td>
<td>0.95</td>
<td>0.93</td>
<td>0.91</td>
</tr>
<tr>
<td>1 GB</td>
<td>0.97</td>
<td>0.95</td>
<td>0.93</td>
</tr>
</tbody>
</table>

It can be observed in the tables 5.10, 5.11, 5.12, 5.13, 5.14 and 5.15 that the efficiency decreases when the number of processing cores is increased while keeping problem size constant. Whereas, it is also evident in the tables that efficiency increases when the problem size as well as number of processing elements have been increased simultaneously. For example, the efficiency of encrypting 250 MB data on 2 cores is 0.78, 500 MB on 4 cores 0.78, 750 on 8 cores is 0.79 and 1 GB on 8 cores is 0.81. Therefore, increasing the data size to 1 GB and number of cores to 8 exhibits approximately same efficiency as encrypting 250 MB data on 2 cores. This ability of maintaining the efficiency by increasing the data size as well as number of processing elements simultaneously is called the scalability. Hence the PRSA2 algorithm was exhibiting the scalability feature therefore it is capable of utilizing the increasing processing resources effectively.

5.7.4 Time Complexity

The time complexity of a given algorithm describes how the runtime of the algorithm grows with increasing size of its inputs. But in the case of any parallel algorithm, it is the function of both input size and the number of processing elements. To obtain the time complexity it is important to design a parallel algorithm in such manner that it can utilize the number
of processing elements efficiently and distribute the input data among the processing elements optimally.

As described earlier in this chapter that PRSA2 has been based on data level parallelism. To exploit data parallel model the message has been divided into the fixed size chunks depending upon the size of key. Then these data chunks were converted into tasks which have been assigned to processing cores later to be executed concurrently.

Let’s suppose the size of the data chunks is \( m \) for a given message, the number of chunks is \( n \) and \( p \) is the number of processing cores available in the system then the parallel time of the PRSA2 can be computed as Eq. 5.9.

\[
T_p = \Theta \left( \frac{n \cdot m}{p} \right) \tag{5.9}
\]

Where \( m \) is a constant (fixed) size of chunk therefore it can be omitted from the Eq. 5.9. Hence the Eq. 5.9 can be rewrite as Eq. 5.10.

\[
T_p = \Theta \left( \frac{n}{p} \right) \tag{5.10}
\]

But for the cases when the exponent increases immensely as compared to the number of processing elements it is important to distribute the computations well among the available number of processing elements. For example if \( p=4 \) but the \( n=16 \) then in such cases it is important to exploit granularity efficiently to control the time complexity of the algorithm. To perform such computations coarse-grained granularity has been used for PRSA2.

### 5.7.5 Cost Optimality

The cost of the parallel algorithm is the cost of solving it on a parallel platform. The execution time of the best known sequential algorithm is considered as the cost of the algorithm for single processing element. A parallel algorithm is considered to be cost-optimal, if the cost of solving that problem in parallel has same asymptotic growth as a function of the input size, is same as the cost of the sequential algorithm.
The cost of the parallel algorithm is usually obtained by multiplying the parallel time with the number of processing elements. It has already been derived in Eq. 5.6 that the parallel runtime of PRSA2 is –

\[ T_p = \Theta \left( \frac{n \times m}{p} \right) \]

Let’s suppose the number of processing elements on a given architecture is \( p \) then the cost of the PRSA2 can be written as per Eq. 5.11 because cost of any parallel algorithm is computed by multiplying parallel time to the number of processing elements, i.e. \( T_p \times p \).

\[ Cost = \Theta \left( \frac{n \times m}{p} \right) \times p \] \hspace{1cm} (5.11)

Because, both expressions are multiplied and both are containing \( p \) therefore it can be omitted from the Eq. 5.11. Hence, after reducing the expression written in Eq. 5.11, now it will become as per given in Eq. 5.12.

\[ Cost = \Theta (n \times m) \] \hspace{1cm} (5.12)

Where, \( m \) is a constant size of chunk, therefore it can also be omitted and the Eq. 5.12 becomes –

\[ Cost = \Theta (n) \] \hspace{1cm} (5.13)

Therefore, the cost of the PRSA2 is \( \Theta (n) \) as given in Eq. 5.13.

It is observed during the experiments performed on PRSA2, that, despite of increasing the size of input \( n \), the resulting cost remains \( \Theta (n) \) which is same as the serial runtime. Therefore PRSA2 is cost-optimal, because it has the same cost as the sequential RSA while executing on parallel architecture.

5.8 Comparative Analysis of PRSA1 and PRSA2

Both variants of parallel RSA – PRSA1 and PRSA2 designed during this research have been compared with respect to the execution time taken during encryption and decryption module. PRSA2 has been shown some fine improvements over PRSA1 while doing the performance comparison.
between PRSA1 and PRSA2. The experiments have been performed using massive data sets in the presence of key with size 2048 bits. The experiments have been performed on same size of message composed of 10 KB for both the versions of Parallel RSA. The Table 5.16 shows the parallel execution time for encryption using 8 cores for both the versions of parallel RSA.

<table>
<thead>
<tr>
<th>RSA Key Size</th>
<th>Execution Time in Seconds by PRSA1</th>
<th>Execution Time in Seconds by PRSA2</th>
<th>Speedup shown by PRSA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA1024</td>
<td>0.01991</td>
<td>0.01024</td>
<td>10.07</td>
</tr>
<tr>
<td>RSA1280</td>
<td>0.02753</td>
<td>0.00997</td>
<td>12.24</td>
</tr>
<tr>
<td>RSA1536</td>
<td>0.03967</td>
<td>0.00872</td>
<td>15.44</td>
</tr>
<tr>
<td>RSA1792</td>
<td>0.050405</td>
<td>0.00829</td>
<td>16.03</td>
</tr>
<tr>
<td>RSA2048</td>
<td>0.06795</td>
<td>0.00797</td>
<td>19.85</td>
</tr>
</tbody>
</table>

PRSA1 itself has shown the significant improvement over the sequential implementation. It represented approximately 5x speedup in the execution runtime as compared to the sequential one.

Whereas the PRSA2 algorithm which is based on SMP architecture represents approximately 19X speedup with respect to the encryption runtime as compared to the PRSA1. It is apparent in the Table 5.17 that the proposed PRSA2 provides faster encryption for longer messages. As and when the data size as well as key size was increased it gave better performance for the encryption as compared to the PRSA1. Hence PRSA2 is better for encrypting the large messages as compared to PRSA1.

5.9 **Comparison of PRSA2 with other Researches**

Various researchers are trying to parallelize RSA algorithm in order to increase the performance in terms of time and energy. The objective behind these researches is to make RSA algorithm more efficient and make it suitable for battery operated devices such as mobile phones, tablets, etc. The researchers used various platforms to increase the
performance of RSA. Majority of these parallel versions of RSA were hardware specific, for example some of them were designed especially for FPGA, and some of them were GPGPU dependent. Table 5.17 shows some of the important versions of parallel RSA developed by different researchers worldwide along with their results.

Table 5.17: Different versions of Parallel RSA proposed by various researchers

<table>
<thead>
<tr>
<th>S. No</th>
<th>Author</th>
<th>Paper Title</th>
<th>Technique</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hong Zhang et al</td>
<td>“Comparison and Analysis of GPGPU and Parallel Computing on Multi-Core CPU”</td>
<td>Parallel implementation of RSA on GPGPU</td>
<td>Speedup of 45x as compared to multi-core CPU</td>
</tr>
<tr>
<td>2</td>
<td>Qing Liu et al</td>
<td>“The research of the Batch RSA Decryption Performance”</td>
<td>Parallelize RSA using OpenMP based on CRT</td>
<td>Speedup of 1.97</td>
</tr>
<tr>
<td>3</td>
<td>Bielecki &amp; Burak</td>
<td>“Parallelization of Encryption Algorithms”</td>
<td>Used OpenMP</td>
<td>Speedup of 3.5</td>
</tr>
<tr>
<td>4</td>
<td>Qiang Liu et al</td>
<td>“A regular RSA processor”</td>
<td>VLSI implementation of the RSA</td>
<td>7% increase in the throughput</td>
</tr>
<tr>
<td>5</td>
<td>Mathieu Ciet et al</td>
<td>“Parallel FPGA Implementation of RSA with Residue Number Systems - Can side-channel threats be avoided?”</td>
<td>FPGA Implementation of RSA with Residue Number Systems</td>
<td>Execution of 1024-bit RSA in less than 150 ms</td>
</tr>
</tbody>
</table>

As it can be seen in the Table 5.17 that two versions of parallel RSA shown at serial number 2 and 3 are software specific and suitable for general hardware available in the market. Both of these versions have been developed using OpenMP. Because the PRSA2 is also software specific and suitable for general machines therefore the PRSA2 has been compared with these two algorithms. The comparison performed has been based on various parameters such as key size, message size, method etc. which is described in the next two sections.
5.9.1 Comparison with the BEA1RSA proposed by Q. Liu et al

Q. Liu et al presented a paper titled “The research of the Batch RSA Decryption Performance” in 2011. In this paper they proposed a variant BEA1RSA, a parallel RSA which was implemented using OpenMP API. They had tested their algorithm on 2 core processor and got the speedup of 1.97 as described in their paper. The PRSA2 was compared with the variant proposed by above scientists. The comparison was made on the basis of various parameters such as number of cores, technology, key size, message size as described in Table 5.18. The PRSA2 was executed on 8 core machine and had shown the speedup of approximately 7.02X as compared to the speedup of 1.97X shown by BEA1RSA. In addition to this the PRSA2 also incorporated PKCS#1 as extra security to the encryption / decryption process performed by PRSA2.

Table 5.18: Comparison between PRSA2 and BEA1RSA by Q.Liu et al

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Parameter</th>
<th>Q. Liu’s BEA1RSA</th>
<th>Proposed PRSA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of Processors / Cores</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Technology</td>
<td>OpenSSL and OpenMP</td>
<td>OpenMP and GNU's MP</td>
</tr>
<tr>
<td>3</td>
<td>Key Size</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>4</td>
<td>Message Size</td>
<td>Batch Decryption</td>
<td>Block Decryption</td>
</tr>
<tr>
<td>5</td>
<td>Extra Security</td>
<td>--</td>
<td>PKCS#1</td>
</tr>
<tr>
<td>6</td>
<td>Speed up</td>
<td>1.97</td>
<td>7.02</td>
</tr>
</tbody>
</table>

5.9.2 Comparison with the Parallel RSA proposed by Bielecki and Burak

Bielecki & Burak presented a paper titled “Parallelization of Encryption Algorithms” in 2007. In this paper they proposed a variant of parallel RSA which was implemented using OpenMP API. They had tested their
algorithm on 8 core processor and got the speedup of 3.5 as described in their paper. The PRSA2 was compared with the variant proposed by the above two scientists. The comparison was made on the basis of various parameters such as number of cores, technology, key size, message size as described in Table 5.19. The PRSA2 was executed using same parameters as it had shown the speedup of approximately 8X as compared to the speedup of 3.5X shown by Bielecki and Burak’s variant.

Table 5.19: Comparison between PRSA2 and Parallel RSA by Belecki and Burak

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Parameter</th>
<th>Bielecki &amp; Burak’s Parallel RSA</th>
<th>Proposed PRSA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of Processors / Cores</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>Technology</td>
<td>OpenMP</td>
<td>OpenMP</td>
</tr>
<tr>
<td>3</td>
<td>Key Size</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>4</td>
<td>Message Size</td>
<td>1 MB</td>
<td>1 MB</td>
</tr>
<tr>
<td>5</td>
<td>Extra Security</td>
<td>--</td>
<td>PKCS#1</td>
</tr>
<tr>
<td>6</td>
<td>Speed up</td>
<td>3.5</td>
<td>7.00</td>
</tr>
</tbody>
</table>

5.10 Conclusion

The PRSA2 algorithm has been designed to overcome the shortcomings of the PRSA1 algorithm. It was designed on the principle of data level parallelism and based on Shared Multiprocessor architecture model. In order to provide it with extra armoring, the PKCS#1 scheme was also incorporated into it.

To test the potential of PRSA2 algorithm, it has been implemented using the OpenMP API on the GCC infrastructure. The GMP Library was used to handle the large numbers during the execution. The results obtained from the experiments performed had shown excellent improvements over
PRSA1 especially for decryption. The PRSA2 was proved to be time efficient because it was giving speedup of approximately 7.5X as compared to its sequential counterpart.

The PRSA2 was designed especially for battery operated mobile devices that had limited power backup and therefore they could supply limited time to execute compute-intensive program. The PRSA2 algorithm has increased the speed of algorithm to a greater extent and therefore it is more suitable for such battery operated devices. It is discussed more in Chapter 7.

Therefore, a faster version of the RSA algorithm can be implemented on multi-core machines if implemented parallel. OpenMP in combination with GCC infrastructure allows implementation of PRS2 algorithm that decreases the execution time and improves the performance in terms of time.
CHAPTER 6
PRSADSA
PARALLEL RSA
BASED
DIGITAL SIGNATURE
ALGORITHM
Chapter 6
PRSADSA – PARALLEL RSA BASED DIGITAL SIGNATURE ALGORITHM

This chapter is based on the objectives 4 and 5 of this research work which include the development of an efficient algorithm capable of handling activities like faster digital signing and a faster digital signature verification with optimum memory utilization. This chapter describes the PRSADSA algorithm proposed in this research. In order to make PRSADSA time-efficient and memory-efficient, the repeated square-and-multiply method along with the parallel computing techniques has been incorporated into it. The PRSADSA has been tested using large data set having two parameters – key size and message size. During experimentation, PRSADSA has shown promising results and gave a speedup of approximately 5X as compared to its sequential counterpart.

6.1 Introduction

In the modern era, Internet has become an integral part of the business. All type of e-commerce applications in areas such as banking, trading, investments, online shopping etc. need to share electronic documents among different parties involved in the business. The use of significant e-commerce transactions is making the situation more complicated.

Traditionally, the physical paper documents are authenticated by means of signature. It is difficult to tamper, forge, or alter the signature once it is done on the paper document. In addition to this, after signing the signatory cannot deny the signing later on. Similarly, the electronic documents can also be authenticated by using digital signatures where tampering with or forging the digital signature is difficult (Bellare and Rogaway, 1996).

A digital signature (Johnson et al., 2001) is based on mathematical calculations, used for the representation of the integrity and authenticity
of a message to be transmitted on network. Once the message is digitally signed, the sender cannot deny that he or she has sent the message to the recipient. In other words, a valid digital signature gives a recipient the reason to believe that the message was created by a known sender. Digital signatures (Juels et al., 1997) are used whenever money transactions are used over the Internet, in the distribution of software, and to detect tampering and forgery.

In general, digital signature algorithm is the combination of two distinct algorithms – hashing algorithm (Van Oorschot and Wiener, 1994) and the signing or verification algorithm. To generate hash code (Nakajima and Matsui, 2002) (hashing) of the message, usually SHA-1 algorithm was used which has been implemented sequentially in PRSADSA. For signature generation and verification, RSA algorithm was used which has been parallelized (Hwang and Xu, 1998).

Usually, RSA algorithm uses larger size keys to ensure the security. Moreover, the RSA signing and verification is based on repeated modular exponentiation (Markov and Saeedi, 2012, Montgomery, 1985) and modular reduction on very large numbers which make it compute-intensive and energy-intensive (Schroeppel et al., 2003). These complex calculations consumes large amount of memory, which makes the algorithm memory-intensive as well (Shamir, 1996). Due to all these problems they become very slow takes lot of time to execute.

In order to overcome all of the problems that have been mentioned above, two distinct technologies have been used – parallel programming to speed up the RSA signing and verification involving larger keys and repeated square-and-multiply method (Hui and Lam, 1994) to improve the memory-efficiency of the compute-intensive parts of the algorithm. Using above mentioned techniques, a new parallel digital signature algorithm, PRSADSA has been designed and proposed in this research.

In order to test the PRSADSA algorithm, it has been implemented using the OpenMP and the GNUs MP Library (Miyaji et al., 1997) on the GCC
infrastructure in the Linux environment. During the experiments the PRSADSA algorithm has been tested using large test case. The test case has been described in the experimental results section. The results obtained during the experiments are promising and have shown approximately 5X speed up as compared to its sequential counterpart.

### 6.2 Digital Signature Algorithms

The Digital Signature Algorithm (DSA) (NIST, 1991) is a standard of proposed by the United States Federal Government to implement digital signatures. The digital signature algorithm was proposed in August 1991 by NIST (National Institute of Standards and Technology) to be used in Digital Signature Standard (DSS). The proposed DSS provides the facility to verify the integrity of the data using public key. In addition to this the public key can also be used to verify the sender of the data. This standard proposed the use of a public-key signature using couple of transformations for signature generation and verification.

Digital Signature (Fischer, 1989) helps to ensure following security features –

- **Authentication:** The message is the one that was originally sent by the sender.

- **Non-Repudiation:** If the sender has sent a digitally signed message then he cannot deny that he is the original sender of the message.

- **Integrity:** The message is the original one and nobody has tampered or altered it during transmission.

Just as there are a number of algorithms for creating hash values, there are also a number of digital signature algorithms. Two of the most commonly used are RSA and DSA.

- **RSA:** The RSA digital signature algorithm was developed by Ron Rivest, Adi Shamir and Leonard Adleman at Massachusetts
Institute of Technology (MIT) in 1977. RSA can also be used to encrypt and decrypt the data being signed. RSA does not mandate the use of a particular hash function, so the security of the signature and encryption are partly dependent on the choice of hash function used to compute the signature.

- **DSA:** The DSA (Digital Signature Algorithm) is defined by the Digital Security Standard (DSS) and was developed in 1991 by NIST. The algorithm requires a SHA-1 digest to compute its digital signature. The DSA algorithm does not encrypt the data being signed; it purely produces a signature that allows the recipient to verify the authenticity and provenance of the data. DSA signatures can be created as quickly as RSA signatures, but their verification can take much longer.

The DSA is assumed to be slower than RSA, therefore in this research the RSA algorithm is used as the key generation algorithm for the digital signatures instead on the DSA algorithm.

### 6.3 RSA Digital Signatures

Diffie and Hellman have proposed general guidelines for digital signatures. The digital signature model was designed in such a manner that it can be used by incorporating any PKC based algorithm. The RSA algorithm, developed by Rivest, Shamir, and Adleman in 1977, has now become the most popular and most trusted algorithm being used with the digital signatures today.

In the RSA digital signature (Cao and Fu, 2008, Somani et al., 2010) scheme, firstly the private key is applied to the message to generate a signature. Secondly, it is verified by applying the corresponding public key on the signature. The verification process gives the result as either valid or invalid signature. The two basic operations that comprise the RSA digital signature scheme are signing and verification.
6.4 Digital Signature Mechanism

The digital signature scheme is a three step mechanism. In the first step, a pair of private key and public key is generated using any key generation algorithm, such as the RSA or the DSA. Then two operations are performed – signature generation and verification, these are discussed in the following sections.

6.4.1 Digital Signing

In order to perform digital signature generation the sender performs the following steps:

1. The first step to generate an RSA signature is to apply a cryptographic hash function (Barreto and Rijmen, 2000) to the message. The hash function is a specially designed message that compresses the message to the fixed size short number (160 bits long), known as hash value. It is performed in such manner that following two conditions have to be satisfied :

   • It is difficult to find a message with a specific hash value.
   • It is difficult to find two messages with the same hash value.

   Usually SHA-1 hash function is used with RSA to generate digital signatures for any message.

2. Sender uses his private key \((n, d)\) to calculate the signature as given in Eq. 6.1.

   \[
   S = M^d \mod n
   \]  
   \[(6.1)\]

3. Sends the message along with signature to recipient.

The mechanism of Digital Signing is demonstrated in Fig. 6.1.
6.4.2 Digital Signature Verification

The recipient performs the following steps in order to verify the message:

1. He uses the public key \((n, e)\) of sender to compute the hash value as per Eq. 6.2.

\[
V = S^e \mod n
\]  
(6.2)

2. Extracts the original hash value from the message.

3. If both the hash values are found to be identical then the signature is considered to be valid.

The mechanism of Digital Signature Verification is demonstrated in Fig. 6.2.
6.5 PRSDSA – The Parallel RSA Based Digital Signature Algorithm

The digital signature algorithms are the combination of hashing and encryption algorithms. The PRSADSA algorithm has been designed such that first it uses SHA-1 algorithm to generate the hash value for the message and subsequently it applies RSA algorithm in order to generate the pair of private and public key. Thereafter, it performs signature generation and signature verification. The SHA-1 algorithm has been applied sequentially to generate the hash value of the given message. Whereas, the signature generation and verification processes has been designed parallel. Refer Appendix C for code snippets of PRSADSA.

The proposed PRSADSA algorithm is based on the repeated square-and-multiply modular exponentiation (Tang et al., 2003) algorithm which describes that for an even exponent the modular exponentiation can be computed as per Eq. 6.1.

\[
base^{exp} \mod modulus = (base^{exp/2} \times base^{exp/2}) \mod modulus \text{ (6.1)}
\]

The PRSADSA algorithm has been divided into two parts which are described further in next two sections.

6.5.1 Parallel Digital Signing

This part of the algorithm has been divided into two parts – generation of hash code and digital signing. To generate the hash code SHA-1 algorithm has been used which has been implemented sequentially. Thereafter the process of digital signing is performed which has been implemented in parallel. In order to sign a message the algorithm performs the following steps:

1. Generates the hash code, H of the message sequentially using SHA-1 algorithm.
2. Uses private key \((d, n)\) to compute the signature in parallel. The Eq. 6.2 is used to perform the same.

\[
S = H^d \mod n
\]  

(6.2)

where \(S\) represents signature, \(H\) is the hash code and \((d, n)\) is the private key.

3. Appends the Signature \(S\) to the message \(M\).

4. Encrypts the Message \(M\) along with signature using public key \((e, n)\) in parallel to get the cipher text \(C\).

5. Sends the encrypted message along with the signature to the recipient.

The block diagram of Parallel Digital signature Module is given in Fig. 6.3. The proposed algorithm is given as Algorithm 6.1.

![Fig. 6.3: Parallel Digital Signing Process](image-url)
Algorithm 6.1: PRSADSA-Digital Signature generation algorithm

 Procedure: signature_generation
 Model: Thread Model based on repeated square-and-multiply
 Input: message, private_key, public_key, modulus
 Output: Signature
 Declare:
         Global: N, Number_of_Cores, Cipher
         Local: None
 Result=1
 Read the Number Of_Cores available on Target_Machine
 Declare Vector Str
 Read Message from File to Vector Str
 Apply SHA-1 to Vector Str and Generate HashCode
 Generate Signature by applying private key on message as –
 Declare N := Power / Number_Of_Cores
 Assign No_Of_Iterations equal to Chunk_Size to each Thread of the each Parallel Section
 Assign N:=Power / Number_of_Cores
 Divide the whole procedure into N parts
 Parbegin
         Declare Result:=1
         If Power mod 2 := 0
                 for j := 1 to N
                         Assign Result := Result * Base
                 end for
         else
                 for j := 1 to N
                         Assign Result := Result * Base
                 end for
                 Assign Result := Result * Base;
         end if
         Assign Cipher := Result mod Modulus
 Parend
 Concatenate Signature to Message
### 6.5.2 Parallel Digital Signature Verification

This part of the algorithm has been divided into three parts – extraction of signature from the message, computation of hash code at the receivers end, and verification of signatures by matching the received signature with the generated one. The process of hash code generation using SHA-1 algorithm has been implemented sequentially whereas the other two processes have been implemented in parallel. In order to perform signature verification the algorithm performs the following steps –

1. The private key (d, n) is applied to message to get the plain text. To perform the same Eq. 6.3 has been used.

\[ M = C^d \mod n \]  \hspace{1cm} (6.3)

Where M is the message, C is the signed document and n is the modulus.

2. Extracts the Signature S from the message M.

3. The public key (e, n) applied to Signature S to get hash code for which Eq. 6.4 has been used.

\[ H = S^e \mod n \]  \hspace{1cm} (6.4)

4. Generates the hash code from the message once again using SHA-1 algorithm.

5. Now both hash codes (extracted from the message and the generated one) were verified for the similarity. If they found to be identical then the signature is considered to be valid otherwise it is discarded due to its invalidity.

The block diagram of Parallel Digital signature Module is given in Fig. 6.4. The proposed algorithm is given as Algorithm 6.2.
6.6 Methodology

The proposed PRSADSA algorithm has been implemented using the OpenMP API on the GCC infrastructure in the Linux environment. The GNUs MP Library has been used to handle large integers because it is capable of accommodating large size intermediate results obtained during computations.

Since the RSA has been used as the base algorithm for PRSADSA, the major part of the algorithm involved series of modular exponentiation and modular reduction routines to be performed on very large integers. Therefore, the signature generation and verification parts of algorithm have been designed using repeated square-and-multiply method. The design of the parallel parts of PRSDSA is on instruction level parallelism.

The parallel distribution of data among the multiple cores is performed by the data distribution technique. Thereafter, these data chunks have been assigned to the tasks which were later mapped to the available cores using static mapping technique.
Algorithm 6.2: PRSADSA – Signature Verification algorithm

Extract Signature from Message
Procedure: signature_verification
Model: Thread Model based on repeated square-and-multiply
Input: Digitally Signed Message, private_key, public_key, modulus
Output: Authenticated Message
Declare: Vector Str
    Global: N, Number_of_Cores, Cipher
    Local: None
Result=1
Read the Number_Of_Cores available on Target_Machine
Read Message from File to Vector Str
Apply SHA-1 to Vector Str and Generate HashCode
Generate Signature by applying private key on message as -
N = Power / Number_Of_Cores
Divide the whole modular exponentiation and reduction into N parts
Parbegin
    Declare Result:=1
    If Power mod 2 := 0
        for j := 1 to N
            Assign Result := Result * Base
        end for
    else
        for j := 1 to N
            Assign Result := Result * Base
        end for
        Assign Result := Result * Base;
    end if
    Assign Cipher := Result mod Modulus
Parend
Compare the generated signature G_Sign with extracted signature E_Sign
if G_Sign equals E_Sign
    Message Accepted
else
    Message Rejected
End if
The model which has been followed in redesigning the parallel algorithm is “Shared Multiprocessor Architecture”. Each thread has been assigned with their personal copies of the private variables which were allocated to local cache memory of the cores. The global variables have been allocated to the shared memory, which were equally shared among the threads.

The proposed PRSADSA algorithm has been designed as scalable thus it was capable of adapting itself to any multi-core machine either 4 cores or 8 cores and so on.

For the experiments, the large data set has been used to test the algorithm. The algorithm has been tested for various key sizes varying from the 512 bits to 2048 bits. The same set of message with size 10 MB has been used for every key size to test the strength of the algorithm.

The performance gained during the execution of PRSADSA algorithm has been measured in terms of time. To measure the time conserved during the execution of algorithm “time” utility of Linux has been used. The time utility measures the elapsed time in milliseconds. To record the time taken during the various segments of the algorithm, it has been divided into two parts, the signature generation part comprised of key generation and the signing part and the signature verification part. To record the execution time of the various parts, the algorithm has been executed for exactly 25 times for each data set and the average of the results has been taken as the final time.

In order to demonstrate the speed up of the algorithm, a comparative analysis was done with respect to the time taken during the serial and parallel implementation of PRSADSA algorithm.

6.7 Platform Used

The configuration of the computer system which was used during the experimentations is described in the Table 6.1.
Table 6.1: Configuration of Computer used for Experiments

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Processor</td>
<td>AMD FX (tm) - 8120 Eight-Core Processor</td>
</tr>
<tr>
<td>2.</td>
<td>Frequency</td>
<td>3.10 GHz.</td>
</tr>
<tr>
<td>3.</td>
<td>RAM</td>
<td>4.00 GB</td>
</tr>
<tr>
<td>4.</td>
<td>System Type</td>
<td>64 Bit Operating System</td>
</tr>
<tr>
<td>5.</td>
<td>Environment</td>
<td>Ubuntu Linux</td>
</tr>
<tr>
<td>6.</td>
<td>Platform</td>
<td>GCC Infrastructure</td>
</tr>
<tr>
<td>7.</td>
<td>Big Integer Library</td>
<td>GNUs MP Library</td>
</tr>
</tbody>
</table>

6.8 Experimental Results

The PRSADSA algorithm has been developed to improve the performance of RSA based digital signature on multi-core machines. After executing the parallel algorithm on 2-core, 4-core and 8-core machine, the promising results have been observed in terms of time conserved during execution. After getting the final results, it has been found that fast implementation of RSA based Signing/Verification is possible on multi-core machines if implemented in a parallel fashion.

The Table 6.2 shows the execution time measured while executing the Digital Signature generation part of PRSADSA algorithm on 2 cores, 4 cores and 8 cores machine along with the execution time of the serial implementation. The Table 6.3 shows the execution time measured while executing the Digital Signature verification part of PRSADSA algorithm on 2 cores, 4 cores and 8 cores machine along with the execution time of the serial implementation. The results shown in the Table 6.2 and Table 6.3 are the comparative results of sequential and parallel variants. The test case which has been taken had two parameters – fixed message size of 10 MB and varied key size ranging from 512 bits to 2048 bits.
It is evident in the results shown in the fig 6.5 and 6.6 that as and when the key size increases the algorithm starts giving the better result as compared to the serial algorithm. It can be observed in the graphs that the execution time decreases when the number of processing elements increased.

Table 6.2: Execution time for the Digital Signature Generation of PRSADSA

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Sequential Code</th>
<th>Parallel Code (2 Cores)</th>
<th>Parallel Code (4 Cores)</th>
<th>Parallel Code (8 Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.01310</td>
<td>0.01250</td>
<td>0.00797</td>
<td>0.00422</td>
</tr>
<tr>
<td>1024</td>
<td>0.07755</td>
<td>0.07251</td>
<td>0.03992</td>
<td>0.02050</td>
</tr>
<tr>
<td>1280</td>
<td>0.13766</td>
<td>0.12600</td>
<td>0.06322</td>
<td>0.03250</td>
</tr>
<tr>
<td>1536</td>
<td>0.27458</td>
<td>0.20733</td>
<td>0.10957</td>
<td>0.05707</td>
</tr>
<tr>
<td>1792</td>
<td>0.43585</td>
<td>0.32571</td>
<td>0.16227</td>
<td>0.08813</td>
</tr>
<tr>
<td>2048</td>
<td>0.67535</td>
<td>0.49270</td>
<td>0.24385</td>
<td>0.13167</td>
</tr>
</tbody>
</table>

Fig. 6.5: Execution time of Signature Generation on 2, 4 and 8 cores
Table 6.3: Execution time for the Digital Signature Verification of PRSADSA

<table>
<thead>
<tr>
<th>Key Size</th>
<th>Sequential Code</th>
<th>Parallel Code (2 Cores)</th>
<th>Parallel Code (4 Cores)</th>
<th>Parallel Code (8 Cores)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.01389</td>
<td>0.01325</td>
<td>0.00861</td>
<td>0.00473</td>
</tr>
<tr>
<td>1024</td>
<td>0.07926</td>
<td>0.07411</td>
<td>0.04112</td>
<td>0.02121</td>
</tr>
<tr>
<td>1280</td>
<td>0.13994</td>
<td>0.12808</td>
<td>0.06472</td>
<td>0.03333</td>
</tr>
<tr>
<td>1536</td>
<td>0.27819</td>
<td>0.21057</td>
<td>0.11165</td>
<td>0.05812</td>
</tr>
<tr>
<td>1792</td>
<td>0.44107</td>
<td>0.33033</td>
<td>0.16467</td>
<td>0.08943</td>
</tr>
<tr>
<td>2048</td>
<td>0.68141</td>
<td>0.49794</td>
<td>0.24655</td>
<td>0.13313</td>
</tr>
</tbody>
</table>

Fig. 6.6: Execution time of Signature Verification on 2, 4 and 8 cores

6.9 Performance Analysis

The PRSADSA algorithm has been redesigned by incorporating repeated square-and-multiply method. While designing it, the focus was on increasing the memory efficiency and speedup of the algorithm. The algorithm was implemented using the OpenMP API and tested using various parameters as discussed in experimental results section. The performance of PRSADSA has been analyzed using various performance metrics and the same is mentioned in the next four subsections.
6.9.1 Speedup

Speedup is one of the most important performance metrics for a parallel algorithm. It is comparative measure of solving the problem in parallel as compared to sequential. It is calculated by taking the ratio of the execution time taken by the serial algorithm on a single processing element with respect to the execution time taken by the parallel algorithm on a parallel computing platform having p number of identical processing elements. Speedup is calculated by the expression given as Eq. 6.5.

\[ S = \frac{T_S}{T_P} \]  

(6.5)

Where TS is the execution time of sequential algorithm and TP is the execution time of the parallel algorithm. The speedup obtained by the PRSADSA for both the test cases has been calculated using the expression given in Eq. 6.5.

Table 6.4: Speedup obtained of Signature Generation on 8 core machine

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size (in Bits)</th>
<th>Signature Generation Speed</th>
<th>Signature Verification Speed</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>512</td>
<td>3.10</td>
<td>2.94</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>1024</td>
<td>3.78</td>
<td>3.74</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>1280</td>
<td>4.24</td>
<td>4.20</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>1536</td>
<td>4.81</td>
<td>4.79</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>1792</td>
<td>4.95</td>
<td>4.93</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>2048</td>
<td>5.13</td>
<td>5.12</td>
<td></td>
</tr>
</tbody>
</table>

The speedups obtained for signature generation and verification while testing PRSADSA on dual quad core (8 cores) is given in Table 6.4. It is evident through the results that the speedup of approximately 5x has been obtained when PRSADSA was executed on a 8 core machine as compared to the serial version of the RSA based digital signature algorithm executed
on single core. In addition to this, the execution time of PRSADSA also increased when the key size increased from 128 bits to 2048 bits because the computations are distributed to all the available cores and PRSADSA start performing better due to its optimal parallel behavior.

6.9.2 Total Overhead

The total overhead of a parallel algorithm is expressed by the difference of the total time taken by all the processing elements collectively to solve the given problem in parallel and the time taken by the best known sequential algorithm to perform that problem sequentially. The total overhead $T_o$ is calculated by Eq. 6.6.

$$T_o = pT_p - T_s$$  \hspace{1cm} (6.6)

The total overheads have been calculated for the largest values of both test cases by using the expression given in Eq. 6.6.

First set of overheads has been calculated for the combination of key size 2048 bits and message size of 10 MB on 8 cores which results into the value of 0.37 and 0.005 approximately for Signature generation and signature verification respectively.

It is evident that the total overhead is a function of problem size and number of processing element. While calculating the total overheads of the PRSADSA for various combinations of problem size and number of processors, it is observed that the total overhead increases with the increase in the size of above two parameters.

6.9.3 Efficiency and Scalability

As discussed in Chapter 2, the efficiency is the measure of the part of time during which the processing elements were involved performing some vital work. The efficiency can be calculated by taking the ratio of the speedup with respect to the number of processing elements involved during the parallel execution of the algorithm. The efficiency is calculated by the expression given by Eq. 6.7.
\[ E = \frac{s}{p} \]  

where E is the efficiency, S is the speedup and P is the number of processors. The efficiency of PRSADSA has been calculated for the test case set that has been used to test the algorithm as described in previous sections.

The efficiencies calculated for the signature generation module and the signature verification module are given below in Table 6.5 and 6.6 respectively.

Table 6.5: Efficiency of Signature Generation in PRSADSA

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>512</td>
<td>0.52</td>
<td>0.41</td>
<td>0.39</td>
</tr>
<tr>
<td>2.</td>
<td>1024</td>
<td>0.53</td>
<td>0.49</td>
<td>0.47</td>
</tr>
<tr>
<td>3.</td>
<td>1280</td>
<td>0.55</td>
<td>0.54</td>
<td>0.53</td>
</tr>
<tr>
<td>4.</td>
<td>1536</td>
<td>0.66</td>
<td>0.63</td>
<td>0.60</td>
</tr>
<tr>
<td>5.</td>
<td>1792</td>
<td>0.67</td>
<td>0.67</td>
<td>0.62</td>
</tr>
<tr>
<td>6.</td>
<td>2048</td>
<td>0.69</td>
<td>0.69</td>
<td>0.64</td>
</tr>
</tbody>
</table>

Table 6.6: Efficiency of Signature Verification in PRSADSA

<table>
<thead>
<tr>
<th>S. No</th>
<th>Key Size</th>
<th>2 cores</th>
<th>4 cores</th>
<th>8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>512</td>
<td>0.52</td>
<td>0.40</td>
<td>0.37</td>
</tr>
<tr>
<td>2.</td>
<td>1024</td>
<td>0.53</td>
<td>0.48</td>
<td>0.47</td>
</tr>
<tr>
<td>3.</td>
<td>1280</td>
<td>0.55</td>
<td>0.54</td>
<td>0.52</td>
</tr>
<tr>
<td>4.</td>
<td>1536</td>
<td>0.66</td>
<td>0.62</td>
<td>0.60</td>
</tr>
<tr>
<td>5.</td>
<td>1792</td>
<td>0.67</td>
<td>0.67</td>
<td>0.62</td>
</tr>
<tr>
<td>6.</td>
<td>2048</td>
<td>0.68</td>
<td>0.69</td>
<td>0.64</td>
</tr>
</tbody>
</table>
It can be seen in Table 6.5 and 6.6 that the efficiency decreases when the number of processing cores is increased while keeping the problem size constant. It can also be seen in the tables that the efficiency increases when the problem size increases while keeping the number of processing elements constant. After observing the values given in the above tables it is evident that the constant efficiency can be obtained by increasing both the problem size and the number of processing elements. For example, the efficiency of digital signature generation using key size 1536 bits and data size 10 MB on 2 cores is 0.66, using key size 1792 and data size of 10 MB on 4 cores is 0.67 and key size 2048 and data size of 10 MB on 8 cores is 0.64. Therefore, it is evident that increasing the key size to 2048 and data size to 10 MB and number of cores to 8 gives the same efficiency for digital signature generation module for 1536 bits on 2 cores. This ability to maintain the efficiency by increasing the data size as well as number of processing elements simultaneously is called the scalability. As the PRSADSA algorithm has exhibited the scalability feature therefore it is capable of utilizing the increasing processing resources effectively.

### 6.9.4 Complexity

The complexity of a given algorithm is describes how the runtime of the algorithm grows with the increasing size of inputs.

But in the case of any parallel algorithm, it is a function of both input size and the number of processing elements. To obtain the minimum time complexity, it is important to design a parallel algorithm in such a manner so that it can utilize the number of processing elements efficiently and distribute the input data among the processing elements optimally.

Suppose PRSADSA has number of processing elements, \( p = 4 \) and the exponent \( n = 4 \) in order to encrypt the data. In such a case, PRSADSA is required to perform four multiplications. At each step one multiplication is to be performed in addition to one communication. Suppose
multiplication takes constant time, $T_m$ and communication takes $T_s+T_w$ therefore the complexity of the PRSADSA can be expressed as Eq. 6.8.

$$T_m = \Theta (\log n)$$ (6.8)

But for the cases when the exponent increases immensely as compared to the number of processing elements it is important to distribute the computations efficiently among the available number of processing elements. For example, if $p=4$ and $n=16$ then it is important to exploit granularity efficiently to control the complexity of the algorithm. To perform such computations, coarse-grained granularity has been used for PRSADSA which is demonstrated in the Fig. 6.7.

![Diagram](image)

Fig. 6.7: Computations using Coarse Grained Granularity for PRSADSA

It is evident in the Fig. 4.8 that at first step, each processing core multiplies $n/p$ numbers locally in time $\Theta (n/p)$. Same operations were performed on each processing element locally and then the problem was reduced to multiply $p$ partial products on $p$ processing elements as demonstrated in the Fig. 6.7 which takes $\Theta (\log p)$ time. Therefore, the parallel runtime of the PRSADSA can be given by the Eq. 6.9.

$$T_p = \Theta \left( \frac{n}{p} + \log p \right)$$ (6.9)
6.9.5 Cost Optimality

The cost of the parallel algorithm is the cost of solving it on a parallel platform. The execution time of best known sequential algorithm is considered as the cost of the algorithm for single processing element. A parallel algorithm is considered to be cost-optimal if the cost of solving that problem in parallel has same asymptotic growth as a function of the input size is same as the cost of the sequential algorithm. It has already been shown in the last subsection that the parallel runtime of the PRSADSA is –

\[ T_p = \Theta \left( \frac{n}{p} + \log p \right) \]

Therefore the cost of the PRSADSA is given in Eq. 6.10.

\[ \Theta \left( n + p \log p \right) \quad (6.10) \]

It is observed during the experiments performed on PRSADSA that as long as the input \( n \) is –

\[ n = \Omega \left( p \log p \right) \quad (6.11) \]

The resulting cost is \( \Theta \left( n \right) \) which is same as the serial runtime. Therefore PRSADSA is cost-optimal as compared to its sequential counterpart.

6.10 Conclusion

The PRSADSA algorithm has been designed to increase the speed of the digital signing and the digital signature verification processes. These parts of the algorithms were based on repeated square-and-multiply method and gave promising results upon execution. The experiments were performed under dual quad core environment and the experimental results have shown that the PRSADSA algorithm gives promising results if implemented using the OpenMP API in the GCC environment. The PRSADSA algorithm has been found to be more time-efficient as well as energy-efficient as compared to its sequential counterpart. The energy efficiency will be discussed later in the document.
A faster version of the RSA-based digital signature algorithm has been implemented on multi-core machines. The OpenMP API in combination with the GCC infrastructure allows implementation of the PRSADSA algorithm that decreases the overall execution time and improves the performance significantly.
CHAPTER 7
ENERGY EFFICIENCY
OF
PROPOSED PARALLEL ALGORITHMS
Chapter 7

ENERGY EFFICIENCY OF PROPOSED PARALLEL ALGORITHMS

This chapter is based on the objective 3 of this research which includes the development of energy efficient parallel RSA algorithm capable of executing the RSA with less energy consumption. The chapter describes the energy efficiency aspects of the proposed parallel algorithms during this research. The energy consumption of all proposed algorithms was measured using Joulemeter and it is demonstrated in this chapter that the proposed parallel algorithms show better energy efficiency on the lower power states of the processor while maintain the same performance level as the sequential algorithm.

7.1 Introduction

One of the objectives of this research was to use parallelism leading to more energy-efficient algorithms for intensive computations and making them more applicable to portable mobile devices (Jaffe and Stavins, 1994). Energy efficiency (Jaffe and Stavins, 1994) means doing the same amount of work using less energy as compared to another program which is performing same amount of work using more energy (Benini et al., 2003). Therefore, while designing parallel algorithm, the main focus was, the new parallel variant must consume less energy as compared to its sequential counterpart and should perform same amount of work in less time.

As described in earlier chapters that the PKC based cryptographic algorithm are compute-intensive due to their roots in the modular arithmetic (Gaubatz et al., 2005). In addition to this the situation becomes more critical because these modular operations are usually performed on very large integers in order to provide ample security to the information (Potlapally et al., 2006). Consequently, their sequential implementation
takes a lot of time as well as energy to execute (Potlapally et al., 2003). But with the growth of mobile commerce on battery operated mobile devices such as smart phones, tablets, etc. the biggest challenge in front of the researchers is now to provide same security to the information as on the desktop computers. But as these devices have limited power backup they require the energy efficient algorithms that can help the devices to save their power from compute-intensive programs and utilize it for other works (Potlapally et al., 2006, Chandrakasan et al., 1992). Since the portable devices come with the support for parallel architecture through their use of multi-core processor, therefore, one solution to this problem is to design energy efficient parallel variants of these security algorithms that can help in achieving high performance on these mobile devices (Gaubatz et al., 2005). These high performance variants will also help the community to secure their crucial information while conserving the limited power backup of their mobile devices.

To measure the energy efficiency of the proposed parallel variants, first requirement was to measure the energy consumption of the both parallel RSA algorithms – PRSA1, PRSA2 and parallel digital signature algorithm - PRSADSA. This has been performed using the Microsoft’s software “Joulemeter” (Aggarwal et al., 2014). Joulemeter is software that measures the energy consumption in the terms of Joules (Bordbar, 2014).

Every computer system works on some default frequency and voltage combination which is called the high power state for the given processor (Goodman et al., 1998). These power states can also be considered as the maximum limit of the performance up to which the processor can perform. But it is rarely required that the processor has to work according to its maximum performance limit. On the contrary, sometimes it is required to reduce the supply voltage and clock frequencies based on the work load to obtain the benefit in terms of power consumption rather than speedup (Tschanz et al., 2003). For example, in the case of battery operated devices, to reduce energy consumption may be more valuable
than improving the performance. This is due to the fact that the battery operated devices usually have limited battery backup therefore they can afford less speedup but cannot afford extra energy consumption.

Nowadays, the new processors which are coming in the market are incorporated with Dynamic Voltage and Frequency Scaling (DVFS) (Potlapally et al., 2006). This feature allows to reduce the frequency of the processor explicitly up to certain extent to decrease the energy consumption of the given computer system or application, because as and when the frequency of the processor is reduces the corresponding voltage also get reduced and system start consuming the less energy. Joulemeter allows this operation by the process known as “Calibration”.

During the experiments the energy consumption of parallel variants has been measured on two distinct environments. Firstly, on the desktop that has been used throughout this research. Secondly, on laptop to prove that the proposed parallel variants are also suitable for battery operated hand held devices. In this chapter, the energy consumption by all parallel variants on high power state is presented. In addition to this the comparison between the energy consumption by serial and parallel variants is also presented on high power state. Thereafter, the energy consumption by the parallel variants on calibrated frequency states is presented. All parallel variants have been shown speedup of 1.2X approximately on laptop and 3X approximately on desktop even after reducing the frequency of the processor of host machine.

### 7.2 Joulemeter

Joulemeter is a software tool developed by Microsoft Research. It can be used to measure the energy consumption of the given system. Joulemeter can be used to measure the energy consumption in terms of –

- Computer system as a unit
- Key hardware components of a given system
- Application running on a given computer system
Joulemeter estimates the energy consumption using a power model that maintains the relationship between the computer resource usage and hardware power state. The hardware power state is interpreted by utilization of processor, frequency of processor, level of screen brightness; power utilized by monitor, I/O devices, etc.

Since the main objective of this research was to make the parallel variants applicable to battery operated mobile devices. Therefore the first target was to fine tune the parallel variants in such manner that they consume less energy while running on the battery operated devices as they have limited battery backup. Thus Joulemeter has been used to estimate the exact energy consumption by different parallel variants developed in this research work.

### 7.3 Working of Joulemeter

As mentioned above, Joulemeter is also capable of measuring the energy consumption by a specific software application running on the given computer system. Therefore, it has been used to measure the energy consumption of all the parallel variants developed during this research.

To monitor the power usage of any application following steps can be performed –

1. Choose the executable file of that application whose power is to be estimated.

2. Enter that name of executable file in the textbox shown in the Application Power (CPU only) portion of the Power usage tab as shown in Fig.7.1.

3. Browse the name of .CSV file in the text box to save the data regarding energy consumption.

4. Click on “Start saving” button.

5. Click on “Start” button.
6. Within about a second, the power usage for that program will start displaying. If the application is not yet running, Joulemeter will wait for the application to start. Power data will be shown after the application is started.

7.4 Calibration in Joulemeter

As mentioned in the last section that, the relationship estimation between computer system resources and the power state of a given computer system is maintained by the power model of the Joulemeter. This process is called calibration.
Calibration is a process of changing the values of the base variables of measuring application to estimate the variation between the actual values and the changed values of the variables (Tschanz et al., 2003).

Joulemeter allows calibrating the frequency by reducing it from the base value. The process of calibration can be performed by clicking on one of the three main tabs of Joulemeter which is “Calibration” (Carvalho et al., 2013). After clicking on the calibration tab, the child dialog box is opened where the old values can be substituted by the newer ones. After performing this, Joulemeter starts giving the reduced values of energy consumption (Carvalho et al., 2013).

Fig. 7.2: Calibration in Joulemeter
7.5 Results and Discussions

As described in previous sections that the Joulemeter was used to measure the energy consumption of various versions of sequential and parallel algorithms. The Joulemeter measures the energy consumption of different key components of computer system used during the execution of a program, such as CPU, monitor, disk, application, etc. It returns the excel sheet pertaining to the power consumption of all key components and the total energy consumed by the application. The example of such excel sheet is shown in Fig. 7.2.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TimeStamp</td>
<td>Total Power</td>
<td>CPU (W)</td>
<td>Monitor (W)</td>
<td>Disk (W)</td>
<td>Base (W)</td>
<td>Application (W)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6.356E+13</td>
<td>25.7</td>
<td>0.7</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>Waiting for [rsa-bin] application data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6.356E+13</td>
<td>26.3</td>
<td>1.3</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>Waiting for [rsa-bin] application data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6.356E+13</td>
<td>31</td>
<td>6</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>Waiting for [rsa-bin] application data</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6.356E+13</td>
<td>42</td>
<td>17</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6.356E+13</td>
<td>38.1</td>
<td>13.1</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>2.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6.356E+13</td>
<td>40.5</td>
<td>15.5</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>2.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6.356E+13</td>
<td>37.7</td>
<td>12.7</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>1.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6.356E+13</td>
<td>39.6</td>
<td>14.6</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>6.356E+13</td>
<td>39.4</td>
<td>14.4</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>0.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>6.356E+13</td>
<td>26.3</td>
<td>1.3</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>6.356E+13</td>
<td>25.8</td>
<td>0.8</td>
<td>10</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7.3: Example of Excel sheet created by Joulemeter

As described above, the energy consumption for the different parallel variants has been performed on two distinct devices –

1. Test case 1 – On a Desktop, to measure the energy consumption of parallel variants on different frequencies and voltage combinations.
2. Test Case 2 – On a Laptop, to measure the energy consumption of parallel variants on different frequencies.
7.5.1 Test case 1

The Laptop (hand held battery operated device) which is used for the Test case 1 has been based on Intel Core 2 Duo. The details of the machine used are shown in Table 7.1. This processor works on the frequency of 1400 MHz.

Table 7.1: Configuration of Laptop used in Experiments

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Processor</td>
<td>Intel (R) Core (TM) 2 Duo T5270</td>
</tr>
<tr>
<td>2.</td>
<td>Frequency</td>
<td>1.4 GHz.</td>
</tr>
<tr>
<td>3.</td>
<td>RAM</td>
<td>2.00 GB</td>
</tr>
<tr>
<td>4.</td>
<td>System Type</td>
<td>32 Bit Operating System</td>
</tr>
<tr>
<td>5.</td>
<td>Environment</td>
<td>Windows 7</td>
</tr>
<tr>
<td>6.</td>
<td>Platform</td>
<td>GCC Infrastructure</td>
</tr>
<tr>
<td>7.</td>
<td>Big Integer Library</td>
<td>GNUs MP Library</td>
</tr>
</tbody>
</table>

The readings recorded for the various versions of sequential and parallel algorithms are displayed in Table 7.2. To find the exact differences between the sequential and parallel versions of algorithms the same size of key and message has been used.

Table 7.2: Energy Consumption recorded on Laptop before Calibration

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Key size</th>
<th>Energy Consumption in Joules</th>
<th>Execution Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential RSA1</td>
<td>2048</td>
<td>845.5</td>
<td>86.409</td>
</tr>
<tr>
<td>PRSA1</td>
<td>2048</td>
<td>949.8</td>
<td>55.872</td>
</tr>
<tr>
<td>Sequential RSA 2</td>
<td>3072</td>
<td>1159.0</td>
<td>124.73</td>
</tr>
<tr>
<td>PRSA2</td>
<td>3072</td>
<td>1326.7</td>
<td>75.734</td>
</tr>
<tr>
<td>Sequential Digital Signature</td>
<td>2048</td>
<td>963.0</td>
<td>103.409</td>
</tr>
<tr>
<td>PRSADSA</td>
<td>2048</td>
<td>1163.5</td>
<td>80.643</td>
</tr>
</tbody>
</table>
It can be observed in the data presented in Table 7.2 that sequential variants are consuming much less power as compared to the power consumed by the parallel variants. It is due to the fact that, in order to execute parallel variants the computer system is required to use eight cores instead of single core and it is quite obvious that system has to spend extra power in order to operate eight cores of the processor as compared to single core. These results were obtained on the high power state of the host machine.

Thereafter the Joulemeter has been calibrated in order to reduce the frequency of the processor and the second set of results were taken on approximately two third of the original frequency. After calibration, promising results has been obtained which are shown in table 7.3.

Table 7.3: Energy Consumption recorded on Laptop after Calibration

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Key size</th>
<th>Energy Consumption in Joules</th>
<th>Execution Time in Seconds</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRSA1</td>
<td>2048</td>
<td>589.1</td>
<td>59.633</td>
<td>1.44</td>
</tr>
<tr>
<td>PRSA2</td>
<td>3072</td>
<td>834.0</td>
<td>81.744</td>
<td>1.52</td>
</tr>
<tr>
<td>PRSADSA</td>
<td>2048</td>
<td>654.5</td>
<td>85.763</td>
<td>1.20</td>
</tr>
</tbody>
</table>

It can be observed in table 7.3 that after reducing the frequency of the processor, the energy consumption by the parallel variants started getting reduced. However as the frequency of the host machine is reduced, the speedup of the parallel variants also gets reduced while keeping it equal to or higher than the sequential counterpart.

Hence on low power states, although the speedup get reduced but the parallel variants become more energy efficient and show their suitability for the battery operated hand held devices.
7.5.2 Test case 2

The desktop machine which has been used for the Test Case 2 was AMD FX(tm)-8120 Eight-Core Processor. The details of machine are given in table 7.4. This processor works on following power states –

- Frequency: 3400 MHz
- Voltage: 1.3125

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Processor</td>
<td>AMD FX(tm)-8120 Eight-Core Processor</td>
</tr>
<tr>
<td>2.</td>
<td>Frequency</td>
<td>3.10 GHz.</td>
</tr>
<tr>
<td>3.</td>
<td>RAM</td>
<td>4.00 GB</td>
</tr>
<tr>
<td>4.</td>
<td>System Type</td>
<td>64 Bit Operating System</td>
</tr>
<tr>
<td>5.</td>
<td>Environment</td>
<td>Ubuntu Linux</td>
</tr>
<tr>
<td>6.</td>
<td>Platform</td>
<td>GCC Infrastructure</td>
</tr>
<tr>
<td>7.</td>
<td>Big Integer Library</td>
<td>GNUs MP Library</td>
</tr>
</tbody>
</table>

The readings recorded for the various versions of sequential and parallel algorithms are displayed in Table 7.5. Here again, to find the exact differences between the sequential and parallel versions of algorithms the same size of key and message were used.

It can be observed in the data presented in Table 7.5 that on desktop machine the sequential variants were consuming much less power as compared to the power consumed by the parallel variants. It was due to the same fact as described in previous section, that, the host is required to spend extra power in order to operate eight cores of the processor as compared to single core. These results were obtained on the high power state of the host machine.
Table 7.5: Energy Consumption recorded on Desktop before Calibration

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Key size</th>
<th>Energy Consumption in Joules</th>
<th>Execution Time in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential RSA1</td>
<td>2048</td>
<td>85.5</td>
<td>26.971</td>
</tr>
<tr>
<td>PRSA1</td>
<td>2048</td>
<td>102.8</td>
<td>5.452</td>
</tr>
<tr>
<td>Sequential RSA 2</td>
<td>3072</td>
<td>109.5</td>
<td>67.332</td>
</tr>
<tr>
<td>PRSA2</td>
<td>3072</td>
<td>152.7</td>
<td>9.672</td>
</tr>
<tr>
<td>Sequential Digital Signature</td>
<td>2048</td>
<td>95.0</td>
<td>31.637</td>
</tr>
<tr>
<td>PRSADSA</td>
<td>2048</td>
<td>121.5</td>
<td>9.643</td>
</tr>
</tbody>
</table>

Thereafter the Joulemeter has been calibrated to reduce the frequency of the processor and the second set of results were taken on approximately two third of the original frequency. After calibration, promising results has been obtained which are shown in table 7.6.

Table 7.6: Energy Consumption recorded on Desktop after Calibration

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Key size</th>
<th>Energy Consumption in Joules</th>
<th>Execution Time in Seconds</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRSA1</td>
<td>2048</td>
<td>59.3</td>
<td>23.744</td>
<td>1.13</td>
</tr>
<tr>
<td>PRSA2</td>
<td>3072</td>
<td>89</td>
<td>62.274</td>
<td>1.08</td>
</tr>
<tr>
<td>PRSADSA</td>
<td>2048</td>
<td>73.6</td>
<td>26.873</td>
<td>1.17</td>
</tr>
</tbody>
</table>

It can be observed in table 7.6 that as the frequency of the processor is reduced, the energy consumption by the parallel variants reduce much faster. However after reducing the frequency of the host machine, speedup of the parallel variants also gets reduced while keeping it higher than the sequential counterpart.

Hence on low power states, although the speedup get reduced but the parallel variants become energy efficient and become suitable for the battery operated hand held devices.
7.6 Conclusion

The experiments results shows that the parallel algorithms developed during this research are giving excellent speedup while running on the multi core computers. These algorithms also shows the energy efficiency when the frequency of the processor is reduced. However, as the speedup is reduced while decreasing the frequency, energy consumption reduces at a faster rate. For the same performance level, parallel implementations uses much lower energy compared to the sequential implementation And the experiments show that these algorithms may be optimized by changing some of the system’s configurations. Thus these algorithms are suitable for the battery operated portable devices.
CHAPTER 8
CONCLUSIONS
AND
FUTURE SCOPE
Chapter 8

CONCLUSIONS AND FUTURE SCOPE

This chapter is based on the conclusions drawn by overall objectives of this research work which includes the development of time efficient and energy efficient parallel PKC based algorithms. In addition to this the chapter also describes the future scope for the work.

8.1 Introduction

This research has been started with the need to design parallel algorithms for the existing public-key infrastructure based security algorithms. The need arose because the public-key based algorithms are compute-intensive and take a lot of time and energy to execute. Parallel programming may be one of the solutions to overcome such complications that can be used to make the PKC based algorithms more applicable to mobile devices such as smart phones, tablets, etc.

Therefore the main focus of this research was on the development of parallel algorithms that can be used to speed up the process of encryption and decryption as well as to speed up digital signing so that both the time and energy involved in these processes can be reduced.

For this research two most important PKC based algorithms have been chosen to redesign as parallel algorithms –

1. RSA Algorithm (Rivest, Shamir and Adleman)
2. DSA Algorithm (Digital Signature Algorithm)

During the research three algorithms have been designed and proposed:

1. PRSA1
2. PRSA2
3. PRSADSA
All the three designed parallel algorithms have been shown to be time efficient as well as energy efficient. Moreover these algorithms are architecture independent; therefore, they don’t require any specific hardware to execute. They can be incorporated in any software or protocol that is required to be executed on any general architecture that is usually available in the market these days.

PRSA1 and PRSA2 algorithms have been based on conventional RSA algorithm and have been redesigned by parallelizing the complex part of the algorithm while keeping the security of the algorithm intact. These algorithms are time efficient hence they can be used to encrypt or decrypt message at a very fast pace on diversified architectures. Similarly PRSADSA can be used to digitally sign the message or any electronic documents to ensure the security and integrity of the message at a faster pace.

Moreover these algorithms are also energy efficient hence they can be used on mobile devices because of their suitability for limited battery backup.

8.2 Conclusions of the research work

This research has been an attempt to develop the new parallel variants of the most important PKC based algorithms to speed up the process of Cryptography and Digital Signatures as well as to decrease their energy consumption.

All research objectives that were laid down before the start of this research have been achieved and excellent results have been obtained at the end of research work. All the algorithms which were proposed during this research work have been proved to be useful and they may be incorporated in the software or protocols related to security and cryptology.

The public key infrastructure based security algorithms have been studied during research work and it has been found that the PKC algorithms have
their working based on modular arithmetic which is the combination of the series of modular exponentiation and modular reduction. Usually these modular computations are performed on very large integers due to which these algorithms become compute-intensive and take lot of time and energy to execute. To overcome such situations, there is a need of technologies that can speed up the computation process. One such technology is parallel programming that can be used to design new parallel algorithms based on existing sequential ones to speed up the process of computation. Therefore, for the further research, parallel programming has been used to increase the time efficiency and energy efficiency of the two most popular PKC based algorithms - RSA and DSA.

The first objective of this research was to increase the memory efficiency of the PKC based algorithms. The memory efficient methods which have been used and tested in this research were repeated square-and-multiply method and right-to-left binary method to implement modular exponentiation and modular reduction on large numbers. Both methods were found to be efficient enough as they gave approximately 20% to 25% speedup as compared to their sequential counterpart. During performance analysis it has been observed that the right-to-left binary method shows overheads when used for the computations involving large integers, because the computations involved here are composed of two major parts. However, these parts can only be parallelized in at the most two sections efficiently whereas the repeated square-and-multiply method can be distributed into as many sections as needed or based on the number of cores available. Thus repeated square-and-multiply method allows more efficient parallelization as compared to right-to-left binary method; therefore during further research work only repeated square-and-multiply method was used.

PRSA1 algorithm has been designed by incorporating repeated square-and-multiply method to increase its memory efficiency. The parallel structure of PRSA1 algorithm uses instruction-level parallelism. The
exponent was divided into separate parts and each part has been mapped with a separate task and then to separate process. Then each process has been assigned to a different core to execute concurrently. Final results have been obtained by multiplying results of each process which is then later converted to a string to obtain the final Cipher text. After using instruction-level parallelism, it has been concluded that it can be used to decrease the time involved in the intensive modular computations of the PKC based algorithms in order to increase their speed. In addition to this, it has also been observed that instruction-level parallelism also support the programmer to provide scalability feature to the parallel algorithm. But the limitation of this method is that the efficiency of the parallel algorithm decreases with increase in the number of processing elements when working with constant message size. However, it has been observed that the efficiency started increasing when the message size and the number of processing elements increased simultaneously. Apart from this, it has also been observed that the PRSA1 algorithm is suitable for small messages only and its performance started degrading with increase in the message size.

PRSA2 algorithm has been designed in such a manner that it starts giving better performance with the increase in message and key sizes. PRSA2 has been designed by incorporating SMP architecture based on data-level parallelism. For this experiment, the data has been divided into fixed size chunks. Each chunk of data has been converted into the fixed size blocks using PKCS#1 scheme. The PKCS#1 scheme has been incorporated in order to provide extra security. The PRSA2 algorithm has also possessed the scalability feature because it has been found to be able to maintain the efficiency while increasing the problem size as well as number of processors. It has been observed during experimentations that data-level parallelism technique is more efficient than that of instruction-level parallelism because it can be used to divide the whole data to be encrypted/signed into the number of parts equal to the number of cores available with the host machine, which supports in the efficient
distribution of task among the processing elements. In addition to this, the efficiency of the PRSA2 increases with increase in the number of processing elements when working with a constant message size. Therefore it is concluded that the data-level parallelism is more efficient method as compared to the instruction-level parallelism.

The third algorithm which was redesigned during the research is PRSADSA based on RSA Digital Signature Algorithm. The PRSADSA algorithm is also based on repeated square-and-multiply method and its design is based on instruction-level parallelism. The message was first converted to Message Authentication code (hash code) using SHA-1 algorithm. Then this MAC was signed using the private key and concatenated with message which was again encrypted using public key and then sent to the receiver. The PRSADSA was redesigned by using instruction-level parallelism again because it is based on repeated square-and-multiply method where instructions are distributed among different parts. Thus, it has been concluded that, for an algorithm, if the instructions performed on the data are more compute-intensive rather than the size of the data itself then it is beneficial to parallelize it by using instruction-level parallelism as compared to the data-level parallelism. In addition to this, it has also been observed that instruction-level parallelism also supports the programmer to provide scalability feature to the parallel algorithm.

The energy consumption of all parallel variant proposed during this research has been measured. To perform this, the Microsoft’s Joulemeter software has been used which measures energy in terms of joules. It has been observed during the energy measurement that all parallel variants were consuming more energy as compared to their serial counterparts due to the fact that system is required to spend extra energy in order to operate all available cores of the host machine as compared to the single core. However, according to the relationship between energy and frequency, with the reduction in frequency, the energy consumption can be reduced at a faster rate by also scaling the voltage down. Next, Joulemeter was
calibrated to reduce the frequency of the host machine on which it works. After scaling down the frequency it has been observed that the energy involved in the algorithms also decreased while decreasing the frequency of the target machine. We can operate at the same performance level as sequential implementation but at a much reduced energy level. Hence on low power states, although the speedups gained are sacrificed but the parallel variants becomes more energy-efficient. Therefore these algorithms are suitable for battery operated devices as they consume less energy as compared to their sequential counterpart.

8.3 Limitations of the Study

Although the proposed algorithms have been proved to be time efficient as well as energy efficient but there are some limitations of this study which are mentioned below:

1. The improved performance of the proposed parallel variants have been observed and analyzed on a single machine only.
2. The energy efficiency of the parallel variants has been tested using software. It can be tested by physically configuring the host machine to find the actual behavior of parallel variants.
3. The SHA1 algorithm has been used in PRSADSA to generate the hash code which was implemented serially.
4. The study has been limited to the RSA and Digital Signature algorithm. There are some other important PKC-based algorithms which have not been a part of this research.

8.4 Future scope of Research

This research presents a lot of scope for further investigations for upcoming researchers. This research can be extended to the following –

1. The proposed parallel variants can also be tested and analyzed on different configurations. The proposed algorithms can also be tested on mobile devices such as smart phones, tablets, etc. to test their performance on limited battery scenarios.
2. However, the energy efficiency of the parallel variants has been shown using software Joulemeter, but it can be proved practically by reconfiguring the target machine by reducing the frequency and voltage. And on these low power states the algorithm can be tested and results may be observed and analyzed.

3. The SHA1 algorithm has been implemented serially in PRSADSA algorithm to generate the hash code. It can also be parallelized in order to obtain even faster versions of PRSADSA.

4. The research involved only RSA and DSA whereas some other important PKC-based algorithms can also be parallelized in order to obtain their time and energy efficient variant, for example, the Diffie Hellman Key Exchange Protocol, the Elliptic Curve, etc.

5. The PRSA1 and PRSA2 can also be incorporated in SSL / TLS protocol, extensively used in electronic commerce, where the RSA algorithm as the part of overall process of securing the transactions.
References


172


LIN, Y.-S., LIN, C.-Y. & LOU, D.-C. Efficient Parallel RSA Decryption Algorithm for Many-core GPUs with CUDA. *International Conference
on Telecommunication Systems, Modeling and Analysis (ICTSM2012), 2012.


OKSUZOGLU, E. & SAVAS, E. Parametric, secure and compact implementation of RSA on FPGA. International Conference on


APPENDIX – A

Parallel Code of PRSA1 Algorithm

1. Code of Encryption Algorithm

//*****CONVERTING STRING INTO NUMBERS**************************
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// letter is an array containing letters equivalent to
// numbers
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
for(len=0;s[len] != '\0';len++){
    for(a=0; letter[a] != '\0'; a++){
        if(s[len]==letter[a]){
            v[len] = a+1;
        }
    }
}
//***** INITIALIZING ARRAYS ****************************************
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// plain array to hold plain text
// cipher array to hold encrypted text
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
mpz_t plain[10000], cipher[10000];
for(x=0;x<10000;x++){
    mpz_init(plain[x]);
    mpz_init(cipher[x]);
}
//***** ENCRYPTION PROCESS ****************************************
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// omp_set_dynamic(1) sets dynamic threading ON
// omp_num_procs() returns number of processors
omp_set_dynamic(1);
int iter = e / omp_get_num_procs();

// Parallel code using OpenMP
// aplain and cipher are shared among the threads
// x is a private variable, each thread will be assigned separate copy of it

#pragma omp parallel for schedule (guided, iter) \
shared (plain, cipher) private (x) \
default (shared)
for(x = 0 ; x < = e ; x+= iter) {
    result = result * plain[x];
}
if(e %2 != 0)
    result = result * plain[x];
cipher[x] = result mod n

//***** DISPLAY ENCRYPTED TEXT****************************************************************************
for(x=0;x<=len;x++) {
    de[x]=mpz_get_ui(cipher[x]);
}
printf("\n The Plain text is (in characters) \n");
for(x=0;x<=len;x++) {
    printf("%c", letter[de[x]-1]);
}

2. Code of Decryption Algorithm

//***** INITIALIZING ARRAYS******************************************************************************
// aplain array to hold decrypted text
mpz_t aplain[10000];
for(x=0;x<10000;x++){
        mpz_init(aplain[x]);
    }

//**** DECRYPTION PROCESS ***********************************************

// omp_set_dynamic(1) sets dynamic threading ON
// omp_num_procs() returns number of processors

omp_set_dynamic(1);
int iter = e / omp_get_num_procs() ; //

// Parallel code using OpenMP
// aplain and cipher are shared among the threads
// x is a private variable, each thread will be assigned separate copy of it

#pragma omp parallel for schedule (guided, iter) \ shared (plain, cipher) private (x) \ default (shared)
for(x = 0 ; x < = d ; x+= iter) {
        result = result * aplain[x];
}

if(d %2 != 0)
        result = result * aplain[x];

cipher[x] = result mod n

//**** DISPLAY ENCRYPTED TEXT***********************************************

for(x=0;x<=len;x++) {
        de[x]=mpz_get_ui(aplain[x]);
}

printf("\n The Plain text is (in characters) \n ");

for(x=0;x<=len;x++) {
        printf("%c", letter[de[x]-1]);
}
APPENDIX – B

Parallel Code of PRSA2

1. Code of Encryption Algorithm

// omp_set_dynamic(0) sets dynamic threading OFF
// omp_num_procs() returns number of processors

#pragma omp parallel for
shared(e_time, d_time)
private(k, j, s, cip, C)
for(k = 0; k < strlen(str); k += 256){
    s = (char *) calloc (2000, sizeof (char));
    for(j = 0; j < 256; j++){
        s[j] = str[i];
        i++;
    }
}

// encrypt() is a function that implements encryption process
// mpz_import() is used to convert number into character

cip = (char *) calloc (2000, sizeof(char));
x = encrypt (cip, s, strlen(s), kp);
mpz_import (C, (256), 1, sizeof(s[0]), 0, 0, cip);
2. Code of Decryption Algorithm

    // omp_set_dynamic(0) sets dynamic threading OFF
    // omp_num_procs() returns number of processors

    //omp_set_dynamic(0);
    //omp_set_num_threads(omp_get_num_procs()*2);

    //***** DECRYPTION PROCESS *************************

    // Parallel code using OpenMP
    // e_time and d_time are shared among the threads to measure the time
    // ku is a private key which is shared among threads
    // k, j, s, cip, msg, M, C, DC is a private variable, each thread will be
    // assigned separate copy of it

    #pragma omp parallel for
    shared(e_time, d_time, ku) private(k, j, s, msg, DC)
    for(k = 0; k < strlen(cip); k += 256){
        s = (char *) calloc (2000, sizeof(char));
        for(j = 0; j < 256; j++){
            s[j] = str[i];
            i++;
        }
    }

    // decrypt() is a function that implements encryption process
    // mpz_import() is used to convert number into character

    msg = (char *) calloc (2000, sizeof(char));
    x = decrypt(msg, cip, strlen(cip), ku);
    mpz_import(DC, (1536), 1, sizeof(s[0]), 0, 0, msg);
    free(msg);
    free(s);
APPENDIX – C

Parallel Code of PRSADSA

1. Code of Signature Generation Algorithm

//****CONVERTING HASH CODE INTO NUMBERS*******************************
//---------------------------------------------------------------
// letter is an array containing letters equivalent to
// numbers
//---------------------------------------------------------------
for(len=0;hash[len] != '0';len++){
    for(a=0; letter[a] != '0'; a++){
        if(hash[len]==letter[a]){
            num[len] = a+1;
        }
    }
}

//**** SIGNATURE GENERATION ****************************************
//---------------------------------------------------------------
// omp_set_dynamic(1) sets dynamic threading ON
// omp_num_procs() returns number of processors
//---------------------------------------------------------------
omp_set_dynamic(1);
int iter = e / omp_get_num_procs(); //
// Parallel code using OpenMP
// sign and num arrays are shared among the threads
// x is a private variable, each thread will be assigned separate copy of it
//---------------------------------------------------------------
#pragma omp parallel for schedule (guided, iter) \
shared (sign, num) private (x) \

default (shared)
for(x = 0 ; x < = d ; x+= iter) {
    result = result * num[x];
}
if(d %2 != 0)
    result = result * num[x];
sign[x] = result mod n

2. Code of Signature Verification Algorithm

//**** DECRYPTION PROCESS ***********************************************
// Signature extracted from message to sign array
// sign converted into number in num array
// omp_set_dynamic(1) sets dynamic threading ON
// omp_num_procs() returns number of processors
omp_set_dynamic(1);
int iter = e / omp_get_num_procs(); //
// Parallel code using OpenMP
// num, e_sign and g_sign are shared among the threads
// x is a private variable, each thread will be assigned separate copy of it
#pragma omp parallel for schedule (guided, iter) \
shared (e_sign, g_sign, num) private (x) \
default (shared)
for(x = 0 ; x < = e ; x+= iter) {
    result = result * num[x];
}
if(e % 2 != 0)
    result = result * e_num[x];
e_sign[x] = result mod n

// Once again SHA-1 algorithm is used to generate Signature in array g_sign
// e_sign and g_sign are compared, if they found to be identical then it is
// assumed that the signature is valid otherwise it is discarded.
//--------------------------------------------------------------------------