A Very Fast Electronic Number Sieve

by

Richard F. Lukes

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Winnipeg, Manitoba, Canada, 1995

©Richard F. Lukes 1995

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-13321-4

Canada

Name RICHARD F. LUKES

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE — SUBJECT TERM

0984 — SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology ... 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology .. 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics ... 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Nom _____

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

_____ ☐☐☐ U·M·I

SUJET — CODE DE SUJET

Catégories par sujets

HUMANITÉS ET SCIENCES SOCIALES

COMMUNICATIONS ET LES ARTS
Architecture 0729
Beaux-arts 0357
Bibliothéconomie 0399
Cinéma 0900
Communication verbale 0459
Communications 0708
Danse 0378
Histoire de l'art 0377
Journalisme 0391
Musique 0413
Sciences de l'information 0723
Théâtre 0465

ÉDUCATION
Généralités 515
Administration 0514
Art 0273
Collèges communautaires 0275
Commerce 0688
Économie domestique 0278
Éducation permanente 0516
Éducation préscolaire 0518
Éducation sanitaire 0680
Enseignement agricole 0517
Enseignement bilingue et multiculturel 0282
Enseignement industriel 0521
Enseignement primaire. 0524
Enseignement professionnel 0747
Enseignement religieux 0527
Enseignement secondaire 0533
Enseignement spécial 0529
Enseignement supérieur 0745
Évaluation 0288
Finances 0277
Formation des enseignants 0530
Histoire de l'éducation 0520
Langues et littérature 0279
Lecture 0535
Mathématiques 0280
Musique 0522
Orientation et consultation 0519
Philosophie de l'éducation 0998
Physique 0523
Programmes d'études et enseignement 0727
Psychologie 0525
Sciences 0714
Sciences sociales 0534
Sociologie de l'éducation 0340
Technologie 0710

LANGUE, LITTÉRATURE ET LINGUISTIQUE
Langues
- Généralités 0679
- Anciennes 0289
- Linguistique 0290
- Modernes 0291

Littérature
- Généralités 0401
- Anciennes 0294
- Comparée 0295
- Médiévale 0297
- Moderne 0298
- Africaine 0316
- Américaine 0591
- Anglaise 0593
- Asiatique 0305
- Canadienne (Anglaise) 0352
- Canadienne (Française) 0355
- Germanique 0311
- Latino-américaine 0312
- Moyen-orientale 0315
- Romane 0313
- Slave et est-européenne 0314

PHILOSOPHIE, RELIGION ET THÉOLOGIE
Philosophie 0422
Religion
- Généralités 0318
- Clergé 0319
- Études bibliques 0321
- Histoire des religions 0320
- Philosophie de la religion 0322

Théologie 0469

SCIENCES SOCIALES
Anthropologie
- Archéologie 0324
- Culturelle 0326
- Physique 0327

Droit 0398
Économie
- Généralités 0501
- Commerce-Affaires 0505
- Économie agricole 0503
- Économie du travail 0510
- Finances 0508
- Histoire 0509
- Théorie 0511

Études américaines 0323
Études canadiennes 0385
Études féministes 0453
Folklore 0358
Géographie 0366
Gérontologie 0351
Gestion des affaires
- Généralités 0310
- Administration 0454
- Banques 0770
- Comptabilité 0272
- Marketing 0338

Histoire
- Histoire générale 0578
- Ancienne 0579
- Médiévale 0581
- Moderne 0582
- Histoire des noirs 0328
- Africaine 0331
- Canadienne 0334
- États-Unis 0337
- Européenne 0335
- Moyen-orientale 0333
- Latino-américaine 0336
- Asie, Australie et Océanie 0332

Histoire des sciences 0585
Loisirs 0814
Planification urbaine et régionale 0999
Science politique
- Généralités 0615
- Administration publique 0617
- Droit et relations internationales 0616

Sociologie
- Généralités 0626
- Aide et bien-être social 0630
- Criminologie et établissements pénitentiaires 0627
- Démographie 0938
- Études de l' individu et de la famille 0628
- Études des relations interethniques et des relations raciales 0631
- Structure et développement social 0700
- Théorie et méthodes. 0344
- Travail et relations industrielles 0629

Transports 0709
Travail social 0452

SCIENCES ET INGÉNIERIE

SCIENCES BIOLOGIQUES
Agriculture
- Généralités 0473
- Agronomie. 0285
- Alimentation et technologie alimentaire 0359
- Culture 0479
- Élevage et alimentation 0475
- Exploitation des pâturages 0777
- Pathologie animale 0476
- Pathologie végétale 0480
- Physiologie végétale 0817
- Sylviculture et faune 0478
- Technologie du bois 0746

Biologie
- Généralités 0306
- Anatomie 0287
- Biologie (Statistiques) 0308
- Biologie moléculaire 0307
- Botanique 0309
- Cellule 0379
- Écologie 0329
- Entomologie 0353
- Génétique 0369
- Limnologie 0793
- Microbiologie 0410
- Neurologie 0317
- Océanographie 0416
- Physiologie 0433
- Radiation 0821
- Science vétérinaire 0778
- Zoologie 0472

Biophysique
- Généralités 0786
- Medicale 0760

SCIENCES DE LA TERRE
Biogéochimie 0425
Géochimie 0996
Géodésie 0370
Géographie physique 0368
Géologie 0372
Géophysique 0373
Hydrologie 0388
Minéralogie 0411
Océanographie physique 0415
Paléobotanique 0345
Paléoécologie 0426
Paléontologie 0418
Paléozoologie 0985
Palynologie 0427

SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT
Économie domestique 0386
Sciences de l'environnement 0768
Sciences de la santé
- Généralités 0566
- Administration des hôpitaux 0769
- Alimentation et nutrition 0570
- Audiologie 0300
- Chimiothérapie 0992
- Dentisterie 0567
- Développement humain 0758
- Enseignement 0350
- Immunologie 0982
- Loisirs 0575
- Médecine du travail et thérapie 0354
- Médecine et chirurgie 0564
- Obstétrique et gynécologie 0380
- Ophtalmologie 0381
- Orthophonie 0460
- Pathologie 0571
- Pharmacie 0572
- Pharmacologie 0419
- Physiothérapie 0382
- Radiologie 0574
- Santé mentale 0347
- Santé publique 0573
- Soins infirmiers 0569
- Toxicologie 0383

SCIENCES PHYSIQUES
Sciences Pures
Chimie
- Généralités 0485
- Biochimie 487
- Chimie agricole 0749
- Chimie analytique 0486
- Chimie minérale 0488
- Chimie nucléaire 0738
- Chimie organique 0490
- Chimie pharmaceutique 0491
- Physique 0494
- PolymÈres 0495
- Radiation 0754

Mathématiques 0405
Physique
- Généralités 0605
- Acoustique 0986
- Astronomie et astrophysique 0606
- Électronique et électricité 0607
- Fluides et plasma 0759
- Météorologie 0608
- Optique 0752
- Particules (Physique nucléaire) 0798
- Physique atomique 0748
- Physique de l'état solide 0611
- Physique moléculaire 0609
- Physique nucléaire 0610
- Radiation 0756

Statistiques 0463

Sciences Appliqués Et Technologie
Informatique 0984
Ingénierie
- Généralités 0537
- Agricole 0539
- Automobile 0540
- Biomédicale 0541
- Chaleur et thermodynamique 0348
- Conditionnement (Emballage) 0549
- Génie aérospatial 0538
- Génie chimique 0542
- Génie civil 0543
- Génie électronique et électrique 0544
- Génie industriel 0546
- Génie mécanique 0548
- Génie nucléaire 0552
- Ingénierie des systèmes 0790
- Mécanique navale 0547
- Métallurgie 0743
- Science des matériaux 0794
- Technique du pétrole 0765
- Technique minière 0551
- Techniques sanitaires et municipales 0554
- Technologie hydraulique 0545

Mécanique appliquée 0346
Géotechnologie 0428
Matières plastiques (Technologie) 0795
Recherche opérationnelle 0796
Textiles et tissus (Technologie) 0794

PSYCHOLOGIE
Généralités 0621
Personnalité 0625
Psychobiologie 0349
Psychologie clinique 0622
Psychologie du comportement 0384
Psychologie du développement 0620
Psychologie expérimentale 0623
Psychologie industrielle 0624
Psychologie physiologique 0989
Psychologie sociale 0451
Psychométrie 0632

A VERY FAST ELECTRONIC NUMBER SIEVE

BY

RICHARD F. LUKES

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

DOCTOR OF PHILOSOPHY

© 1995

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis, to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film, and LIBRARY MICROFILMS to publish an abstract of this thesis.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or other-wise reproduced without the author's written permission.

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Abstract

Number sieves are special-purpose devices used for solving certain problems in computational number theory. In this thesis we give a brief history of number sieves. We describe a single-chip sieve device, called the Scalable Sieve Unit (SSU), which has been used to implement a parallel sieving device, called the Manitoba Scalable Sieve Unit (MSSU). The MSSU achieves sieving rates at least an order of magnitude greater than those of previous sieves through the use of parallelism and a tightly-coupled dedicated control processor. We discuss the design of the hardware and software components which make up the MSSU sieving system.

A number of applications of sieving are described, including many recent results obtained using MSSU. Some results on problems of quadratic character are presented, extending results of Lehmer, Lehmer and Shanks. The list of known pseudosquare values is extended, and some evidence is presented which suggests that the growth of this sequence is sufficiently rapid to ensure that primality testing can be performed in deterministic polynomial time. New results are presented with regard to quadratic polynomials with high asymptotic densities of prime values. We also extend the numerical investigation of Shanks on Littlewood's bounds on the Dirichlet L-function $L(1, \chi)$. Some results on other problems are presented, including a search for a successor to Euler's famous prime-producing polynomial, further computations of the Erdös-Selfridge function, a new pseudocube value, and computation of pseudopowers for bases 2, 3, 5 and 7.

A new method of employing parallel sieves is described which we call multi-variable sieving. We use this method on the MSSU to find three new solutions to the sums of three cubes problem, finding representations of the integers 110, 435 and 478 as the sum of three integer cubes.

An investigation into Waring's problem for cubes presents new evidence to suggest that all numbers are representable as sums of four positive or negative cubes, and that all numbers greater than approximately 10^{13} can be represented as the sum of four positive cubes.

Acknowledgements

There are a number of individuals who I would like to acknowledge for their assistance in the preparation of this thesis.

Foremost, I would like to thank my supervisor, Dr. Hugh C. Williams, for giving me the opportunity to be involved in interesting and rewarding research. I consider myself very fortunate to have been one of his students. His financial support of my studies and in the building of OASiS II and the MSSU are greatly appreciated. Financial support was also provided by NSERC and the University of Manitoba Graduate Fellowship.

I would also like to thank my committee members Dr. Duncan Buell, Dr. John Bate and Dr. Bob McLeod for their careful reading of my thesis.

I am very grateful to Dr. Cameron Patterson, the designer of the SSU single-chip sieve devices. Dr. Patterson's ingenuity and diligence in the design and development of the SSU made the MSSU project possible. The speed and reliability of these devices are truly remarkable. In the many thousands of hours of sieving all of the SSU devices have performed flawlessly.

I would like to thank my fellow student Mike Jacobson for his invaluable computations which are used throughout the thesis. I wish him the very best as he continues his studies in Germany. As well, I thank Dr. Renate Scheidler for her collaboration on computations of the Erdös-Selfridge function which provided me with an opportunity to meet both of these distinguished mathematicians.

For answering scores of questions, I am grateful to the technical expertise and sense of humour of Gilbert Detillieux, Tom Dubinski, and Professor Daniel Salomon.

Computer time on an Alpha workstation used for some computations in Chapter 7 was kindly provided by Guy Durocher and the Physics Department of the University of Manitoba. I would also like to thank Vaclav Zvonik for assistance in the mechanical construction of the MSSU chassis.

I would also like to acknowledge the support and encouragement of my tennis friends: Ken Hotz, Andy Kuc, Doug Stone, and Tony Weekes. I wonder if they have realized yet that I don't actually play tennis myself? There are many, many other people who helped me, or helped to change me, during my six years of PhD studies, and I am glad I met each and every one.

Finally, and most importantly, I would like to acknowledge my parents who provided me with the encouragement and motivation to undertake my PhD in the first place. I dedicate this thesis to them. I shall always be grateful that they sacrificed everything by leaving Czechoslovakia in 1968 to ensure that I was raised in a free and democratic country.

Contents

1 Introduction 1

1.1 The sieve problem 1

1.2 Principles of Sieve Automation 3

1.3 Some Terms and Measures 4

1.3.1 Basic Components of a Sieve Device 4

1.3.2 Measuring Sieve Performance 5

1.3.3 Density of Sieve Problems 5

1.3.4 The Filter Mechanism 6

1.4 Optimizations 7

1.4.1 Single Residue Optimization 7

1.4.2 Multiple Residue Optimization 8

1.5 Factoring using Sieves 8

1.6 History of Sieving 10

1.6.1 Early Sieve Devices 13

1.6.2 Lehmer's Sieves 14

1.6.3 Electronic Sieves 16

1.6.4 A Summary 23

2 The Manitoba Scalable Sieve Unit Architecture 25

2.1 Motivation 25

2.2 The Scalable Sieve Unit 26

2.3 MSSU Overview 30

2.4 Comparison of MSSU with Recent Systems 31

3 MSSU Hardware 34

3.1 Control Processor 34

3.2 Sieve Array 38

3.3 Sieve Chassis 42

4 MSSU Software: The Control Processor Software 44

4.1 The Function of SSUMon 44

4.2 The Choice of Programming Language 46

4.3 SSUMon Overview 48

4.3.1 CPU32BUG Resources 49

4.4 Implementation of Residue Optimization 50

4.4.1 Partitioning 51

4.4.2 The Counter 51

4.4.3 Ring Loading 53

4.5 Checkpoints 56

4.6 SSUMon Variables 57

4.6.1 The Enable Vector 58

4.6.2 The Output Mode 59

4.7 SSUMon Command Set 59

4.7.1 Host Communication Protocol 60

4.7.2 The Reset Command 62

4.7.3 Sieve Loading and Unloading 62

4.7.4 The Run and Halt States 63

4.8 Solution Processing 63

4.8.1 QSPI Routines 64

4.8.2 The `Which()` Routine 65

4.8.3 Solution Pipelining 66

4.8.4 A Responsive Halt Mechanism 67

4.9 Filter Routines 68

4.9.1 The Virtual Filter – `VIRT` 70

5 MSSU Software: The Host Software **72**

5.1 Host Software Overview 72

5.2 Format of Problem Files 74

5.2.1 PROBLEM Section 75

5.2.2 CONGRUENCES Section 75

5.2.3 Specifying Parallel Execution of Sieve Jobs 76

5.2.4 PARAMETERS Section 79

5.2.5 LOG Section 82

5.3 The Programs 84

5.3.1 The Configuration File 84

5.3.2 The `Sieve` Command 85

5.3.3 `talker` 86

5.3.4 `stat` 86

5.3.5 `pull` 87

5.3.6 Off-Line Solution Filters 88

5.3.7 Problem Generators 89

5.4 Implementation of the Sieve Management Software 89

5.4.1 Multiple Precision Arithmetic 90

5.4.2 Shared Files 90

5.4.3 Use of Signals 91

5.4.4 Problem File Parser 93

5.4.5 The Daemon Process 94

5.4.6 The Sender 95

5.4.7 Solution Processing 96

5.4.8 Porting to Other Hosts 97

6 Problems Involving Quadratic Characters **98**

6.1 Pseudosquares 99

6.1.1 Pseudosquares and primality testing 101

6.1.2 Pseudosquare results 107

6.1.3 Negative Pseudosquares 115
6.2 Quadratic Polynomials Having a High Density of Primes 117
6.2.1 The conjecture of Hardy and Littlewood 117
6.2.2 Some results 119
6.3 Polynomials of the Form $x^2 + A$ 125
6.4 Lehmer-Lehmer-Shanks Problem V 130
6.5 Examination of Littlewood's Bounds on $L(1, \chi)$ 132
6.5.1 The Analytic Class Number Formula 132
6.5.2 The Search for Unusually Large Regulators 133
6.5.3 Littlewood's Bounds on $L(1, \chi)$ 135

7 Some Results on Other Problems **151**
7.1 On a Successor to Euler's Polynomial $X^2 + X + 41$ 151
7.1.1 Computing $A'(n)$ 155
7.2 The Erdös-Selfridge Function 158
7.2.1 The Sieving Algorithm 159
7.2.2 Implementation 160
7.2.3 Computational Results 162
7.3 Pseudocubes 165
7.4 Pseudopowers 169
7.4.1 A Heuristic Estimate for $P_2(x)$ 170
7.4.2 Pseudopowers of Other Bases 171
7.4.3 The c_b Values 173
7.5 Two-Dimensional Sieving Problems 177
7.5.1 Sums of Three Cubes 177
7.6 Sums of Four Cubes and Waring's Problem 184

8 Conclusion **193**
8.1 A Summary 193
8.2 Improving the MSSU 194

8.3 A Successor to the MSSU 196
8.3.1 More Hardware Rings 196
8.3.2 Greater Parallelism 198
8.3.3 Newer VLSI Technologies 198
8.4 New Applications 200

A Summary of SSUMon Commands **202**

B SSUMon Filter Algorithms **212**
B.1 The Perfect Square Filter – `PSQR` 212
B.2 The Perfect Cube Filter – `PCUB` 216
B.2.1 Description of the Initial Algorithm 216
B.2.2 A Sequence of Transformations 220
B.3 The Probable-Prime Filter – `PBPM` 229

List of Tables

2.1 SSU Commands 28
2.2 SSU Signal Lines 30
2.3 Comparison of Recent Electronic Sieves 33

4.1 SSUMon Modules 49
4.2 Sieve State Information (SSI) 58
4.3 Sieve Global Variables 58
4.4 SSUMon Commands 60
4.5 SSUMon Message Types 61
4.6 SSUMon Filters 69

5.1 Sieving Parameters 79
5.2 Log Message Types 82
5.3 Sieve Command Options 85
5.4 SMS Signals 92

6.1 Pseudosquares 112
6.2 Negative Pseudosquares 116
6.3 $C(D)$-hichamps ($D < 0$) 121
6.4 $C(D)$-hichamps ($D > 0$) 121
6.5 Selected $C(D)$-hichamps 122
6.6 Computations to $x = 10^7$ for Select $C(D)$-hichamps 122
6.7 Values of N_p and their sources 123
6.8 Values of M_p and their sources 124

6.9 $C(-A)$-hichamps ($A > 0$) 126
6.10 Computations to $x = 10^6$ for $C(-A)$-hichamps 126
6.11 $C(-A)$-lochamps ($A > 0$) 127
6.12 Computations to $x = 10^6$ for $C(-A)$-lochamps 128
6.13 Lehmer-Lehmer-Shanks Problem IV 129
6.14 Lehmer-Lehmer-Shanks Problem V 131
6.15 $D \equiv 5 \pmod 8$ — LLI-lochamps 138
6.16 $D \equiv 1 \pmod 8$ — ULI-hichamps 138
6.17 $D \equiv 3 \pmod 8$ — ULI-hichamps 139
6.18 $D \equiv 6 \pmod 8$ — ULI-hichamps 139
6.19 $D \equiv 7 \pmod 8$ — ULI-hichamps 140
6.20 $5N_p$ - Least Solutions 141
6.21 $5N_p$ - Least Prime Solutions 142
6.22 $-7R_p$ - Least Solutions (Problem II) 143
6.23 $-7R_p$ - Least Prime Solutions (Problem II) 144
6.24 $1R_p$ - Least Solutions (Problem I) 145
6.25 $1R_p$ - Least Prime Solutions (Problem I) 146
6.26 $6R_p$ - Least Solutions 147
6.27 $6R_p$ - Least $2\times$ Prime Solutions 148
6.28 $3R_p$ and $7R_p$ - Least Solutions 149
6.29 $3R_p$ and $7R_p$ - Least Prime Solutions 150

7.1 Tabulation of $A'(n)$ 157
7.2 Tabulation of the Erdös-Selfridge Function 166
7.3 Pseudocubes 168
7.4 Pseudopowers to base 2 172
7.5 Pseudopowers to base 3 174
7.6 Pseudopowers to base 5 175
7.7 Pseudopowers to base 7 176
7.8 Numbers d requiring m cubes for $0 \leq d < 3.43 \times 10^8$ 189
7.9 Non-4-cubes in selected ranges of size 10^8 192

List of Figures

2.1 SSU Block Diagram 27
2.2 SSU Scan Paths 28
2.3 MSSU Block Diagram 31

3.1 MC68332 Block Diagram 36
3.2 Simplified Schematic of the Sieve Array Board 39
3.3 Sieve Array Board Layout 41
3.4 Two MSSU sieves (one extended from rack with cover removed). 43

4.1 Calculation of the `COUNT` Value 52
4.2 Layout of `RINGDAT` 54
4.3 Operations on the Ring Data 54
4.4 Simultaneous Read and Write of SSU Registers 64

5.1 The Sieve Management Software 73
5.2 Partitioning Example 77
5.3 Multi-Variable Sieving Example 78
5.4 Sample Log Output 83

6.1 Pseudosquare growth vs. p 113
6.2 Pseudosquare growth vs. n 114

Chapter 1

Introduction

1.1 The sieve problem

Consider the simple problem of determining L_7, where L_7 is the least positive non-square integer such that

$$\begin{aligned} L_7 &\equiv 1 \pmod 8, \\ L_7 &\equiv 1 \pmod 3, \\ L_7 &\equiv 1 \text{ or } 4 \pmod 5, \\ L_7 &\equiv 1, 2 \text{ or } 4 \pmod 7. \end{aligned}$$

The solution of this problem is $L_7 = 1009$. This is a solution, for the primes less than or equal to 7, to an important problem in number theory known as the *pseudosquares* problem. Later, in chapter 6, we shall discuss this problem in detail.

This problem is also an instance of a much larger problem known as the Generalized Sieving Problem (GSP), in which we are given:

1. bounds A, B with $B > A$;
2. k moduli $m_1, m_2, m_3, \ldots, m_k$ $(m_i > 1, i = 1, 2, 3, \ldots, k)$ which we usually assume are relatively prime in pairs;
3. k sets of *acceptable residues* $\mathbf{R}_i = \{r_{ij} \mid 0 \leq r_{ij} < m_i\}$ $(i = 1, 2, 3, \ldots, k)$.

We are then to determine all $x \in \mathbb{Z}$ such that

$$A \leq x < B$$

and

$$x \pmod{m_i} \in \mathbf{R}_i \qquad (i = 1, 2, 3, \ldots, k).$$

Some problems may place further restrictions on x in addition to satisfying this set of linear congruences; in our example we want x to not be a perfect square. Other possibilities include restricting x to prime values, values for which $x^2 - N$ is a perfect square, or just about any other mathematical condition. The process of searching for values of x satisfying such additional restrictions is called *filtering* and is considered a problem apart from the GSP itself.

We call this a sieve problem because each successive modulus m_i can be regarded as acting like an ordinary kitchen sieve with an increasingly finer mesh, thereby further restricting the possible solutions to the problem. Notice that the well known sieve of Eratosthenes for finding primes is an instance of the GSP. In this case, if we have $B = p_k^2 + 1$ where p_k is the k^{th} prime, we see that the problem of finding all the primes between A and B is that of solving the GSP for $m_i = p_i$ and $\mathbf{R}_i = \{1, 2, 3, \ldots, p_i - 1\}$ $(i = 1, 2, 3, \ldots, k)$.

Patterson [Pat91b] has shown that, when translated into a decision problem, the GSP is NP-complete and is therefore likely to be a very difficult problem in general. Certainly, there is no efficient method known for solving the GSP. The best methods that are

currently available involve the construction of mechanisms which automate the process of searching for solutions. In order to see how such a device might be constructed, we will now discuss some of the principles of sieve automation. For further information on this we refer the reader to [Pat91b] and to Stephens and Williams [SW90].

1.2 Principles of Sieve Automation

Consider the problem mentioned at the beginning of this section and observe the tableau below.

X	0	1	2	3	4	5	6	7	...	1009
mod 8	×	✓	×	×	×	×	×	×	...	✓
mod 3	×	✓	×	×	✓	×	×	✓	...	✓
mod 5	×	✓	×	×	✓	×	✓	×	...	✓
mod 7	×	✓	✓	×	✓	×	×	×	...	✓

For each of the moduli 8, 3, 5, 7 we have put a check mark (✓) under the acceptable residues and a cross (×) under the others. When we find a value of X for which there is a column consisting only of check marks, this value of X represents a *solution candidate* to the problem. In this example, the solution candidate $X = 1$ is rejected because it is a perfect square, and it is not until $X = 1009$ that we find a valid solution. The difficulty with this approach is that we may have to go a very long way before we find a solution, particularly if we have many moduli. We can solve this problem by noting that each row of the tableau exhibits a cyclic pattern corresponding to the list of acceptable and non-acceptable residue classes for its associated modulus, information that can be compressed by the expedient of recording the results for each modulus only once. Any column of the original table can be generated by shifting each of the rows to the left in order to advance the appropriate set of residues to the first column, the leftmost residue wrapping around to the rightmost end. In general, then, column t can be obtained by performing t of these shift operations.

To see how a machine that performs this sieving operation can be fabricated, we note that each modulus m_i can be represented by a loop or *ring* which is divided into m_i positions. Those positions that represent acceptable residues are tagged in some manner to differentiate them from non-acceptable residues, and the machine examines one position from each ring at a fixed location called a *window* or *tap*. The rings are advanced in unison and a solution is detected when the window is filled exclusively with tags identifying the residues as being acceptable for all the rings of interest. A *trial counter* records the number of shifts performed by the machine. Thus, the machine can be set up to start searching from some point N by setting the i^{th} ring such that $N \pmod{m_i}$ is in the window when the mechanism is turned on; after s shifts the value being tested is $N + s$. A device of this type is called a *number sieve* or, frequently, just a *sieve*.

1.3 Some Terms and Measures

Before we can continue in our discussion of sieve devices, it is important to define some terms and measures which will aid our discussion.

1.3.1 Basic Components of a Sieve Device

Inherent in the design of all sieve devices are three basic components: the *characteristic vector generator* (CVG), a *solution detector*, and a trial counter. The CVG corresponds to the rings discussed above. The purpose of the CVG is to generate the patterns of residues corresponding to our sieve problem. Solution taps route residue information from the CVG to a solution detection mechanism. In most digital logic implementations, the solution detector is a wide AND-gate having one input per ring. To prevent the counter value from being out of synchronization with the residue information, the shifting of the rings is kept in lock-step with incrementing of the trial counter.

1.3.2 Measuring Sieve Performance

To avoid confusion, we shall define two measures for describing the speed of a sieve device:

Definition 1.3.1 *We define the* **canvass rate** *of a sieve problem to be the number of values of x included within a specified sieving range, $A \leq x < B$, divided by the number of seconds required to sieve that range.*

Because sieving problems can be subject to various optimizations and transformations which we will discuss later, we also define a measure of the raw sieving rate of the underlying sieve hardware in order to make valid comparisons between various devices.

Definition 1.3.2 *We define the* **maximum hardware sieving rate** *to be the maximum number of values tested per second by the solution detection mechanism. This value is given in terms of* **trials per second**.

1.3.3 Density of Sieve Problems

Clearly, the ratio of acceptable to non-acceptable residues will have an effect on the solution rate of a given sieve problem. Often, it is useful to define some measure of the density of a sieve problem, and for this it is useful to compare the density of a given problem to that of a quadratic sieve. Although we shall not discuss quadratic sieves until chapter 6, it suffices to say that the number of acceptable residues per modulus of a quadratic sieve problem is 1 for modulus 8, and $(p_i - 1)/2$ for the odd primes $p_i \leq p_k$, where p_k is the largest modulus specified by our sieve problem. From this it is very easy to see that the density of solutions for quadratic sieves is

$$\frac{1}{8}\prod_{i=2}^{k}\frac{p_i - 1}{2}.$$

Lehmer [Leh53, p. 8] and Cantor et al. [CEFT62, p. 151] have used the quadratic sieve as a benchmark when discussing the solution density of sieving problems, identifying

problems as being "less than quadratic" or "more than quadratic". For our purposes we shall consider the quadratic sieve to be a *moderate* density sieve problem. We shall call the extreme of one residue per modulus, equivalent to a Chinese Remainder Theorem (CRT) type of problem, a *sparse* sieve problem. At the other extreme, when all but one residue per modulus is acceptable, as is the case with the sieve of Eratosthenes, a problem shall be considered a *dense* sieving problem. In between these extremes we define "moderate to low density" sieves and "moderate to high density" sieves.

1.3.4 The Filter Mechanism

When filtering is required, a very high solution rate may overwhelm the resources available for filtering and cause a sieve problem to be *filter bound.* When a problem is filter bound, the sieve will operate at a much lower speed than the maximum hardware sieving rate, as the process of filtering, or merely outputting solutions, becomes a bottleneck.

Commonly, there are two different types of filtering mechanisms employed by sieve systems. With *on-line* filtering, filtering is performed almost immediately after the generation of a solution candidate by using a computer connected directly to the sieve. This is in contrast to *off-line filtering*, where the solution candidates are collected in a batch, and at some later time transferred to a computer system for processing. Clearly, before the advent of digital computers, all solution filtering was performed off-line, using manual methods such as hand-computation.

Often, even when on-line solution filtering is provided, some off-line post-processing of the data may be required. For example, it may be necessary to perform primality testing of solutions, a process generally too time consuming to be performed on-line. Instead, a fast probable-prime test can be performed on-line, with the much slower primality test being performed off-line.

1.4 Optimizations

The performance of this basic model can be improved further by testing for solutions in parallel. We can improve the sieving rate by a factor of t by using t sieves, starting each at a different point in the range to be sieved, or by constructing a single sieve with a t position solution window. Using an idea which originated from the work of Cantor et al. [CEFT62], it is possible to test t solution taps at once. This window would test t consecutive positions from each ring instead of just one and then the sieve would advance the rings by t positions. However, in order to support t solution taps, a ring must accommodate more than t positions, hence many such designs support very small moduli with rings corresponding to prime powers (see [Pat91b, pp. 112-114]).

1.4.1 Single Residue Optimization

We can often increase the sieving rate even more by making use of the idea of *single residue optimization.* Possibly, the best way to illustrate this idea is to look once again at the example at the beginning of this section. Notice that if we combine the moduli which have only a single acceptable residue, we need to find a value for X such that $X \equiv 1 \pmod{24}$. Since $X = 1 + 24Y$, instead of sieving on X, we could sieve on Y. In this case all residues modulo 8 and 3 are acceptable for Y and we simply rearrange our remaining residues modulo 5 and 7 so that

$$\begin{aligned} Y &\equiv 0 \text{ or } 2 \pmod{5}, \\ Y &\equiv 0, 1 \text{ or } 5 \pmod{7}. \end{aligned}$$

This simple idea, known to Lehmer (see [Leh53]), effects a 24 fold speedup for the problems because any solution Y can easily be converted to one for X by computing $X = 1 + 24Y$ and the sieving range has been reduced by a factor of 1/24. An algorithm to perform the required residue reordering for single residue optimization will be presented in Section 4.4.3.

1.4.2 Multiple Residue Optimization

The idea of single residue optimization can be applied to multiple residues to obtain an additional increase in the sieving rate. This idea was used by Lehmer [Leh28, p. 891] to divide a factoring problem into two separate sieve problems based on the congruence $a \equiv 2, 3 \pmod 5$. By separating results into two cases and using single residue optimization on both subproblems, a speedup factor of $5/2$ was achieved over that of a single run.

In 1975, Stark [Sta75, p. 301] used this method to divide a quadratic type problem into 30 subproblems with respect to the *implicit modulus* $m_1 = 8 \cdot 3 \cdot 5 \cdot 7 \cdot 11 = 9240$. Using single residue optimization modulo 9240, each subproblem achieves a speedup of $9240/30$ over a non-optimized problem, not including any additional costs incurred in running multiple subproblems over that of a single problem.

1.5 Factoring using Sieves

One of the earliest applications of sieve devices was in the factoring of integers. In fact, it was the interest in factoring which spurred much of the early development of sieve devices. Although modern factoring algorithms programmed on present-day computers have made factoring by sieving obsolete, up until about 1970, sieving was the fastest general method available for factoring large integers [SW90, p. 43]. In order that the reader is not left with the impression that sieves are only of historical interest, we shall be presenting examples of contemporary sieve applications in Chapters 6 and 7 of this thesis.

The factoring algorithm we shall describe is based on an idea which began with Fermat in 1643 [Dic66] who suggested that if one can find a representation for a number N as a difference of squares, $x^2 - y^2$, then one can easily determine two factors of N. Assuming $N = U \cdot V$, where $U \geq V$, and U, V are both odd, then we can define

$$x = \frac{1}{2}(U + V), \qquad y = \frac{1}{2}(U - V),$$

and see that

$$U = (x + y), \qquad V = (x - y),$$

and

$$N = (x + y)(x - y) = x^2 - y^2.$$

Stated as a sieving problem, we must find a value x such that $x^2 - N$ is a perfect square. It is a fairly straightforward matter to specify the necessary congruence relations.

For example, in 1938 Lehmer [Leh39, p. 137] used a mechanical sieve to factor the integer N_1, where

$$N_1 = \frac{(2^{120} + 1)(2^8 + 1)}{(2^{24} + 1)(2^{40} + 1)} = 18518800563924107521.$$

In fact, an examination of Lehmer's unpublished sieve log shows that he employed a good deal of mathematical ingenuity to restrict the possible values of x to numbers of the form $806400k + 4304146561$ or $806400k + 4302188161$. Nevertheless, the following description will suffice to describe the general method.

Recall that we determine the set of congruences modulo p_i which must be satisfied so that $x^2 - N$ is a perfect square. We start by tabulating values of x, x^2 and $x^2 - N$ modulo the small primes p_i. These calculations for small values of p_i are shown below. We have crossed off those values of x which correspond to values of $x^2 - N$ which cannot possibly be perfect squares.

p_i	x	$x^2 \pmod{p_i}$	$N \pmod{p_i}$	$x^2 - N \pmod{p_i}$
2	0, 1	0, 1	1	0, 1
3	~~0~~, 1, 2	0, 1, 1	1	~~2~~, 0, 0
5	0, 1, ~~2~~, ~~3~~, 4	0, 1, 4, 4, 1	1	4, 0, ~~3~~, ~~3~~, 0
7	~~0~~, 1, 2, ~~3~~, ~~4~~, 5, 6	0, 1, 4, 2, 2, 4, 1	4	~~3~~, 4, 0, ~~5~~, ~~5~~, 0, 4

From the above, we see that modulo the primes 2, 3, 5, and 7 the resulting congruences are:

$$\begin{aligned} x &\equiv 0,1 \pmod 2 \\ x &\equiv 1,2 \pmod 3 \\ x &\equiv 0,1,4 \pmod 5 \\ x &\equiv 1,2,5,6 \pmod 7 \end{aligned}$$

If we continue this process for all moduli up to and including the prime 229, and run this problem on a suitable sieve device, the first solution occurs for $x = 23651756161$. This corresponds to a y-value of 23256972480, which yields the two factors $U = x + y = 46908728641$, and $V = x - y = 394783681$.

We note that the complexity of this method is $O(N)$; however, variations on this method were developed by Lehmer and Lehmer [Leh33b] that were $O(\sqrt{N})$.

1.6 History of Sieving

It appears that F.W. Lawrence [Law96] in 1896 was the first individual to actually consider constructing a number sieve. This device, which could be used to factor integers (see Williams and Shallit [WS]), was to be made up of from 15 to 20 rings. These rings, representing the small primes p_i ($p_1 = 2$, $p_2 = 3$, $\ldots$ p_k), would be wooden or cardboard gears with p_i (or, when p_i is very small, some power or multiple of p_i) teeth, with the distance between these teeth being the same for all. As these gears must be of differing diameters, it would be necessary for each of them to be supported by a different axle in order for all of them to engage a common driving gear. The teeth on each of these ring gears would be numbered and acceptable residues indicated by a brass stud just within the appropriate tooth and projecting through the gear on both sides. The machine was to be set up in such a way that the ring gears were side by side but spread out in a slightly

fan shaped arrangement in order that they be close together where they were to be turned by the common driving gear, but were farther apart at the other side. This would ensure not only that the brass studs would pass easily under the supports of the different axles, but also that they would all come in contact with each other near the driving wheel when a solution occurred. This line of contact of the brass studs at a solution could be indicated to the operator by having it complete an electrical circuit which would cause a bell to ring or the rotation of the driving gear to cease.

Lawrence did not build this machine, but was content merely to give a very brief sketch of how it might be constructed. Very likely it would not have worked well at high speeds because of the lack of precision implicit in its design.

Later, in 1910 Gérardin published a French version of Lawrence's paper in his monthly mathematical review, *Sphinx-Oedipe*. This appears to have acted as a stimulus to Kraitchik who announced (see Gérardin [Gér12a]) in February of 1912 that he had constructed a model of a number sieve. This device, made of wood, is described more fully by Gérardin [Gér12c] and much later by Kraitchik [Kra22, pp. 43-44]. It was composed of 15 gears with differing numbers of teeth representing the moduli: 64, 54, 50, 49, 44, 52, 51, 57, 46, 58, 62, 37, 41, 43, 47. These gears were mounted on a single shaft A' but could turn freely about this shaft. A second group of 15 gears, each with 64 teeth, were mounted on another shaft A and fastened on that shaft. The radii of the gears on shaft A and those of their corresponding gears on shaft A' were fixed at 200 mm. This meant that the two shafts could be arranged in parallel with the gears representing the moduli engaged by the gears on shaft A. The radii of the gears were also determined such that the gears representing the moduli would all turn at the same rate when driven by rotating shaft A. For example, if r_1 is the gear radius of a gear on shaft A and r_2 is the radius of the gear on shaft A' corresponding to it, then we would want $r_1 + r_2 = 200$ mm and $r_1/r_2 = 64/m$ where m is the number of teeth on the gear of radius r_2.

A numbered hole was placed a fixed distance (60 mm) from the centre beneath each

gear tooth of the gears on shaft A'. For a given sieve problem the non-acceptable residues (mod m) had the holes corresponding to those on the gear with m teeth plugged. A light source was placed at a fixed point near the end of shaft A' and a screen was placed at the other end. As shaft A rotated, a solution would occur when a set of unplugged holes on the 15 gears lined up; this would be detected when a spot of light appeared on the screen. While this mechanism probably worked reasonably well at low speeds it would very likely have been useless at high speeds because the gear teeth, being of different sizes, would produce meshing problems. In any event it seems that Kraitchik abandoned his sieve shortly after he constructed it.

In March 1912, Gérardin [Gér12c] reported that P. Carissan had developed a machine for factoring and that he (Gérardin) had sent a posting to the French Association for the Advancement of Science describing a sieve which he had constructed. Although the subsequent paper [Gér12b] is rather vague, as are all of Gérardin's papers on his machines, some information is provided in his note [Gér12d]. From this we are informed that his mechanism was made up of an unspecified number of strips, likely of paper (see [Gér12b]) and each represented a ring for some modulus. A strip representing modulus m was marked off in a multiple of m spaces of fixed size. These spaces were coloured white or black according to whether the residue represented by any particular space was acceptable or not. A solution to a sieve problem which required several of these strips would be indicated by a completely white column. The strips were set up to be almost equal in length, usually about 3 meters, and were wound up between two similar rollers, with an operator checking for blank columns. This is certainly not as sophisticated a device as Kraitchik's, but it was likely useful for small problems.

Carissan's machine, constructed by his brother E.-O. Carissan, is described very inadequately in [Gér12c] and mentioned only in passing by Gérardin [Gér13]; but in Carissan [Car20a] we learn that it was made up of hanging, flexible modular loops (rings) which were driven simultaneously by a grooved roller and scanned for solutions by eye. We are also informed that it was rather inefficient.

All three of these machines suffered from the same inadequacies: they were inefficient, existed only as prototypes, required the human eye to scan for solutions, and produced no significant result. Thus, it is difficult to regard them as being anything other than experiments that failed. In the next section we will discuss some successfully completed number sieves.

1.6.1 Early Sieve Devices

In 1913-14, E.-O. Carissan, who was a lieutenant in the French infantry, began his development of an automatic sieve device. Unfortunately, his work was interrupted by World War I and not completed until 1919. The resulting precision machine, fabricated by the firm of Chateau Frères of Paris, incorporated 14 circular brass rings representing the moduli: 19, 21, 23, 26, 29, 31, 34, 37, 41, 43, 47, 53, 55 and 59. These rings were driven at the same linear rate by a pinion gear underneath them. A ring representing modulus m had m studs on its top side with the spacing of these studs being the same for all the rings. Non-conducting caps were placed on those studs which represented acceptable residues for the modulus m. These caps would press on contact switches which were mounted on an arm placed above the rings. The switches were connected in series and the completion of a circuit (each stud under the arm is capped) would cause a sound to be made in a telephonic receiver placed over the ears of an operator, alerting him to a solution. The mechanism was driven by hand, canvassed numbers at the rate of 35 to 40 per second and displayed solutions in a 6 digit counter. Further details on this remarkable machine, together with example problems, can be found in Carissan's well-written and detailed paper [Car20a].

Carissan had plans for improving this device such as driving it by means of an electric motor. Indeed, in an unpublished manuscript [Car20b] he refers to it as a first model and provides a list of 4 possible improvements that could be made to subsequent versions. Unfortunately, no further work seems to have been done and this machine was totally

forgotten until fairly recently; this was very likely the result of the premature deaths of both P. Carissan (in 1923) and E.-O. Carissan (in 1925) and the rather obscure journal in which [Car20a] appeared. What is, nevertheless, particularly remarkable is that this beautifully constructed machine, the earliest example extant, of a special purpose automatic mechanism for solving number theoretic problems, still exists today. It was recently donated by a daughter of E.-O. Carissan, France Silber of Grenoble, to the museum of the Conservatoire Nationale des Arts et Métiers in Paris.

1.6.2 Lehmer's Sieves

The next important developments in sieving technology are almost exclusively due to D. H. Lehmer. As his several devices have been fully described elsewhere (see [SW90], [Pat91b] and Lehmer [Leh28], [Leh33a], [Leh34], [Leh80]), we will only provide brief descriptions of some of them here. Lehmer's first sieve, completed in 1927, was made up of 19 bicycle chain loops containing the following numbers of links: 64, 27, 25, 49, 22, 26, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61 and 67. These loops were suspended from a common shaft with 19 10-tooth sprockets to engage the chains. The zero link for each chain was painted red and acceptable residues were indicated by the presence of a small pin (a piece of chicken wire, actually) attached to their corresponding links. The shaft, driven by an electric motor, rotated at the rate of 300 r.p.m. (50 links per second). When a pin projected above the shaft it pressed on a spring loaded contact. As in Carissan's sieve (of which Lehmer had no knowledge until 1989) the contacts were connected in series and completion of the circuit would trip a relay which shut off the power to the motor. A revolution counter on the shaft was used to determine the solution value. This machine was used for a year or so, but was disassembled by Lehmer in anticipation of a move to Providence, R.I. Regrettably, it was stolen before it could be used again. The machine has recently been reconstructed by Robert Canepa of CMU (see Pomerance [Pom82]), and this model is now in storage at the Computer Museum in Boston.

In 1932, encouraged by the success of the Bicycle Chain Sieve, Lehmer went on to construct a much more ambitious device: the Photoelectric Sieve. This mechanism represented the fruition of the somewhat impractical ideas of sieve construction put forward by Lawrence and Kraitchik. As in Kraitchik's model, a modulus m is represented by a gear of m teeth. The 15 driving gears all had 100 teeth and were mounted on a common shaft. Each of the 30 modulus gears had an individual axle and meshed with one of the driving gears. The gears were arranged in two sets of 15 on either side of the driving gears with two modular gears engaging one driving gear. A hole was drilled at a fixed distance beneath each modulus gear tooth, and a toothpick was used to plug holes that represented unacceptable residues. A light source was placed near the end of the driving shaft and so was a photodetector, with a pair of prisms used to redirect the light beams from the first set of 15 modular gears to the second. When a solution occurred, there would be an alignment of the unplugged holes, allowing the photodetector to trigger a series of amplifiers and relays which ultimately opened the circuit breaker for the electric motor which drove the gears. This sieve, which had gears representing 64, 27, 25, 49 and the primes from 11 to 113, performed at the rate of 5000 numbers per second. The main difficulties with the machine, which performed some very impressive feats of calculation [Leh33b], were: (1) it was very difficult to transport because of its size and sensitivity, (2) loading a complete problem took over two hours [Ste89, p. 17], (3) its photoamplifier system, which amplified the light signal by a factor of over 7×10^8, was at the very forefront of the technology available at the time, causing all kinds of difficulties in running the device (see D. N. Lehmer [Leh33]).

Two years later Lehmer developed another sieve with about the same simplicity, speed, capacity and cost of the Bicycle Chain Sieve, but it was much more compact and easy to move. In the Movie Film Sieve the chains were replaced by 18 loops of 16mm movie film, with quarter inch holes punched in the loops to represent unacceptable residues. Adjustable rollers (one for each loop) were used to provide tension at the bottom of the loops which were driven by a conducting brass drive shaft 18 metallic brushes were

attached to a brass bar mounted above this drive shaft. When there were no holes in any of the film loops as they passed over the drive shaft, none of the brushes would be in contact with the shaft, resulting in a shut-down of the driving motor. It was this simple device, very reminiscent of Carissan's first attempt at a sieve, that Lehmer used to produce his table in [Leh37]. We should also point out that a year after this, Gérardin [Gér37], apparently totally unaware of Lehmer's previous work, constructed a number sieve. Although there is a photograph of this machine which suggests that it was based on an adding machine, the description provided is both vague and all too brief. We are, however, informed that it was electric, automatic, able to print and executed at the rate of about 25000 operations per working day.

1.6.3 Electronic Sieves

Up to this point, the sieve devices which have been discussed were primarily mechanical devices, with chains, gears or loops for moving parts. Although, Lehmer's photo-electric sieve made use of rather sophisticated electronics [1], this was strictly limited to the detection of solutions. Obviously, the speed of mechanical sieves was greatly limited by the constraints imposed by moving parts.

It was not until the advent of electronic computers in 1945 that Lehmer experimented with the problem of implementing sieves in software and produced several programs which implemented sieves on various computers. In fact, experience with the ENIAC (the Electronic Numerical Integrator and Calculator) caused Lehmer [Leh46] to propose the construction of what he called an Electronic Sieve. This device would have used vacuum tubes to achieve an estimated sieving rate of about 100000 to 200000 numbers per second. In this same proposal, he also discussed the possible construction of an Acoustic Sieve in which the periodic element would be a tube of ethylene glycol with a

[1] In fact, Lehmer noted that this may have been the first instance in which the digital flip-flop was used in computing (see [Leh80])

piezo-electric crystal at each end. This instrument would, if constructed, sieve at the rate of 10^8 numbers per minute. Lehmer's plans for the Electronic Sieve were quite advanced, but his proposal was apparently not funded, as the device was never constructed.

Software Sieves

The next twenty years saw a lull in sieve building, as Lehmer and others concentrated on implementing sieves in software on various digital computers. In the 1950's, Lehmer [Leh53] programmed the Standards Western Automatic Computer (SWAC) to solve the GSP. This sieve implemented 17 rings in software and canvassed at a rate of approximately 1450 trials per second. In the 1960's, Brillhart [BS67] used an IBM 7094 to achieve a canvass rate of 150000 trials per second with a software sieve supporting 21 or 22 congruences. A much faster software sieve would later be implemented by Lehmer on the experimental ILLIAC IV supercomputer in the early 1970's. This single instruction, multiple data (SIMD) architecture computer contained 64 processors and was able to sieve at a rate of 15 million trials per second.

The Delay Line Sieve

In 1965, Lehmer would build one of his most successful sieves, the Delay Line Sieve (DLS-127). This machine made use of delay lines as memory elements. Delay lines transmit electrical pulses at a fixed speed; these were used to build loops whose lengths were proportional to the sieve moduli. By carefully initializing each loop with pulses for the acceptable residues, solutions would be indicated by the simultaneous arrival of pulses from all the delay line taps. Circuitry would carefully resynchronize and reshape the pulses, allowing the ring data to recirculate indefinitely without loss of information. The first version of the machine used the primes or powers of primes less than or equal to 127 as moduli; later, in the early 1970's the machine was renamed the DLS-157 when 6 more moduli (constructed from shift registers instead of delay lines) were added. This device

was very cheap to construct, was very flexible and ran at the rate of 10^6 numbers per second.

Loading of problems on the Delay Line Sieve was automated by a program on an IBM 7094 computer which accepted punched cards containing the specification of a diophantine equation $f(x, y) = 0$, where f was a polynomial of arbitrary degree with integer coefficients of arbitrary size (see [Leh66]). Using this information, the 7094 would produce a punched paper tape suitable for input directly to the Delay Line Sieve. After outputting a solution, the sieve would automatically load the next problem mounted on its paper tape drive, allowing multiple sieve problems to be enqueued.

The Delay Line Sieve featured a *solution counting mode* used to output the number of solutions within a specified range without actually outputting individual solutions. This mode was found to be useful for certain problems. For example, for the pseudosquares problem, filtering could be avoided by predicting the number of perfect squares in the search range and comparing this with the observed solution count. A discrepancy would alert the operators to the presence of a solution. This was a particularly valuable feature for this sieve because solution filtering was performed off-line.

The Movie Film Sieve, parts of the Photoelectric Sieve, and the DLS-157 are also now in storage in the Computer Museum in Boston.

Cantor, Estrin, Fraenkel and Turn

One of the most impressive early proposals for the construction of an electronic sieve device was provided by Cantor, Estrin, Fraenkel and Turn [CEFT62] in 1961. This called for a sieve to be built from basic digital building blocks similar to those which were being used in the transistorized digital computers of the time. Basing the speed of the sieve on the shift times required by the IBM 7090 digital computer, Cantor et al. envisioned that the shift registers would be clocked at a rate of 5 *megahertz* (MHz), or one shift every 0.2μsec. They also state that the sieving rate would have been in excess of 10^{10} numbers

per minute, or 167 million per second. From this we assume that 32 solution taps per ring were planned.

Clearly, this machine would have required the use of the latest state-of-the-art components in digital electronics and would have been extremely expensive to build; nevertheless, it is clear the technology existed at that time to build such a device. As it turned out, the proposal was not acted on, and it would be another 20 years before a machine of similar capabilities was to be built. Several revolutionary ideas were put forward in this proposal, including the idea of multiple solution taps, which Cantor et al. referred to as *observation posts*, and the idea of combining a sieve with a digital computer as part of a "fixed plus variable" $(F + V)$ structure computer system. A conventional digital computer represented the fixed part of this architecture, while the variable part consisted of a set of reconfigurable modules or digital building blocks which acted as a special purpose computer. In this case, the special purpose computer would implement a shift-register sieve.

A major advantage of the $(F + V)$ architecture is that the conventional computer can be used to simulate in software those rings which are not implemented in hardware. Using our own terminology, the computer implements *virtual rings* corresponding to moduli not supported by the *hardware rings*. Somewhat surprisingly, in [CEFT62] there is no mention of how a digital computer system connected directly to the special purpose sieve hardware could also be used to perform a filtering function (in particular, perfect square testing), nor how it could greatly simplify the loading of sieve problems and the gathering of solutions.

The SRS-181

Lehmer went on to develop a sieve that used only shift registers to implement the modular rings, the Shift Register Sieve, SRS-181, but unfortunately this uncompleted machine was mistakenly scrapped when Lehmer was away from Berkeley. According to Morrison and

Brillhart [MB75, p. 199], the SRS-181 was expected to be operational by the end of 1974.

From several descriptions of the machine (see [Leh80], [Pat91b, p. 70], [Ste89, pp. 22–23]), it appears the SRS-181 was intended to be a stand-alone machine, with a complex array of displays and switches to allow operator control of the device. It was capable of operating in both regular search and solution counting modes. The sieve was to feature 42 hardware rings corresponding to the primes less than or equal to 181. The shift-registers were constructed from off-the-shelf 8-bit shift-register integrated circuits (ICs). The original design called for a single solution tap with a shift rate of 16 million per second resulting in a hardware sieving rate of 16 million trials per second.

UMSU

A more sophisticated version of the shift-register sieve, called the University of Manitoba Sieve Unit (UMSU), was constructed by Patterson and Williams [PW83] in 1983. Originally, it was planned that this project would simply reconstruct the SRS-181 [Pat83, p. 8]. However, these plans would be modified to allow the sieve to be connected directly to a minicomputer host, initially a DEC PDP-11/03, as an attached processor. This would be the first realization of the "fixed plus variable structure" sieve system as proposed by Cantor et al.

The eventual design boasted a complement of more than 500 medium-scale integration (MSI) integrated circuits using readily available transistor-transistor-logic (TTL) components. The electronic circuitry was constructed on three wire-wrapped circuit boards interconnected via a backplane. Shift-register rings corresponding to the first 32 primes were implemented with the first four rings corresponding to the prime powers $2^6 = 64$, $3^4 = 81$, $5^2 = 25$, and $7^2 = 49$. A shift clock rate of 16.67 MHz, or 16.67 million cycles per second, combined with eight solution taps per ring, resulted in a maximum hardware sieving rate of 133 million trials per second.

UMSU was the first sieve to implement virtual ring emulation, on-line filtering, and

automatic problem restart following a power-failure. A special feature called *checkpointing* served the dual purpose of periodically testing for hardware errors and recording the hardware state of the machine. In the event of a hardware error or power-failure, the sieve problem could be restarted using the state information of the last successful checkpoint. Testing for the presence of a hardware error was performed by predicting the contents of the shift-register rings based on the initial contents of the rings and the counter value and comparing this to the actual ring contents.

A simple user editable file format was used for specifying the congruences of a sieve problem, as well as various parameters including: start and stop counter values, specification of a filter function, and the interval to be used between successive checkpoints. These *problem files* could be edited using a text editor on the multi-user host system and submitted to a *problem queue* for execution.

OASiS

All of these sieves described above have the disadvantage of operating with fixed moduli. In 1985, Stephens and Williams [SW90] constructed a variable modulus sieve called the Open Architecture Sieve System (OASiS), which utilized 16 8K RAM chips. This would permit the user to employ up to 16 arbitrary combined moduli, each of which could be no larger than 8192. For example, up to 37 moduli corresponding to all of the primes less than or equal to 157 could be fit into the 16 rings, with the moduli 5, 7, 11 and 13 being combined in one ring of modulus 5005. The OASiS design has a shift rate of 13.3 MHz, with 16 solution taps, resulting in a hardware sieving rate of 215 million trials per second.

The major advantages of OASiS over its predecessor, UMSU, are its enhanced speed, variable size ring capability and modular design. Designed with future expandability in mind, a second OASiS sieve was constructed in 1991 at minimal cost using a spare set of printed circuit boards. The modular design allowed "board swapping" to be used to

isolate faults, simplifying maintenance.

The major shortcoming of OASiS is the very low bandwidth connection of 9600 Bits Per Second (BPS) between OASiS and its host computer system, a microVAX II. For example, it can take as long as 10 minutes to load and verify a problem, depending on the size and number of rings required by the particular problem. Even worse, since the consistency check portion of checkpoints are actually performed by the host system, checkpoints can take up to 10 minutes as well. For problems such as the pseudosquares, which required a significant amount of filtering, the slow communication path between sieve and host, combined with the relatively low CPU performance of the microVAX II, result in filtering being a significant bottleneck.

The Bronson and Buell Sieve

Very recently, Bronson and Buell [BB93] have made use of Field Programmable Gate Array (FPGA) technology to develop a very fast number sieve. The Bronson and Buell sieve consists of an attached processor, called SPLASH, interconnected via a VME bus interface to a Sun workstation host. SPLASH is a reconfigurable systolic array processor, developed at the Supercomputing Research Center in Maryland, which consists of a linear array of 32 early generation FPGA chips. These FPGA devices consist of a large number of logic building blocks which can be dynamically rewired to perform various functions. The possible circuit combinations are limited only by the number of available logic blocks and the interconnectivity of the specific FPGA devices. Reconfigurable architectures such as SPLASH blur the conventional distinction between hardware and software by allowing a designer to write "code" which specifies a hardware architecture.

The real power of this sieve architecture is that it allows a designer to theoretically custom configure the hardware resources for the problem at hand. In Lehmer's [Leh46] preliminary proposal for an Electronic Sieve using vacuum tube technology, he advocated designing the ring circuitry utilizing several hundred identical logic modules allowing

the ring configuration to be changed via physically plugging and unplugging modules and cables, as required. The proposal of Cantor et al [CEFT62] states that, "by an appropriate rearrangement of the building blocks, problems with different moduli can be run". FPGAs allow such reconfiguration to take place on-the-fly, at hardware speeds.

Bronson and Buell [BB93] have implemented a sieve using SPLASH which possesses shift-register rings whose moduli correspond to 22 primes, or powers of primes, between 3 and 127. This sieve has 64 solution taps and a shift rate of 16 million per second, for a maximum hardware sieving rate of 1024 million trials per second. This is the highest rate ever achieved by a single sieve, and is almost 5 times faster than the previous record holder, OASiS.

1.6.4 A Summary

In Table 1.1 below we present a very brief summary of the history of sieve construction. The Scalable Sieve Unit (SSU), which appears near the bottom of this table, will be described in detail in the following chapter.

Sieve Systems

Machine	Year	Rings	Trials/Sec
E. Carissan	1919	14	35–40
Bicycle Chain	1927	19	50
Optical Gears	1932	30	5,000
16mm Movie Film	1936	18	50
A. Gérardin	1937	?	?
SWAC†	1950's	?	1,450
IBM 7094†	1960's	21 or 22	100,000
DLS-127	1966	31	1,000,000
DLS-157	1969(?)	37	1,000,000
ILLIAC IV†	1960's	64	15,000,000
Shift Register	1975	42	20,000,000
UMSU	1983	32	133,000,000
OASiS	1989	16 variable	215,000,000
Sun4/280†	1989	32	2,000,000
SSU	1991	30	192,000,000
Bronson, Buell	1992	22	1,024,000,000

†Denotes software running on a general purpose computer

Table 1.1

Chapter 2

The Manitoba Scalable Sieve Unit Architecture

The *Manitoba Scalable Sieve Unit* (MSSU) is the third of a succession of sieve units designed and built at the University of Manitoba. As discussed above, the first such device was the UMSU sieve [Pat83], completed in 1982, and the second was the OASiS sieve [Ste89], completed in 1988. The first MSSU sieve was completed in 1993. Soon thereafter, a second MSSU sieve was built, and now both devices are in operation at the Computer Science Department at the University of Manitoba.

2.1 Motivation

From the outset, the goal of the MSSU design was to achieve a sieving rate of at least an order of magnitude faster than that of OASiS, which at the time was the world's fastest electronic sieving device. The pseudosquares problem was chosen as a benchmark by which to measure sieve performance, both for its mathematical importance and for its great computational difficulty. It is also representative of a larger family of problems known collectively as problems of quadratic character.

Since 1988, first UMSU, and later OASiS, were used to extend the known list of pseudosquares from L_{157} to L_{223}. In 1991, in an attempt to extend the pseudosquares search further, a second OASiS sieve was constructed, and both OASiS sieves were used to compute the values of the pseudosquares L_{227} and L_{229}. At that point it was obvious that a further pseudosquare search using OASiS would be futile since we could expect to wait months, if not years, for the next solution.

At about this time, Cam Patterson at the University of Calgary was completing the fabrication of a small number of VLSI circuit devices which essentially placed an electronic sieve device, similar in many respects to UMSU, on a single chip. It was clear that by operating many of these sieve chips in parallel, a very fast sieve device could be built. The original outline of the design envisioned a parallel architecture consisting of 16 VLSI sieve devices, a powerful control microprocessor capable of performing solution-filtering without intervention of a host computer, and a multi-user host computer which would provide file storage for specification of problems and for solution capture.

2.2 The Scalable Sieve Unit

In [Pat91b], Patterson describes the *Scalable Sieve Unit* (SSU), a *Very Large Scale Integration* (VLSI) implementation of an electronic sieve device. The SSU combines the circuitry for hardware rings, a counter, a solution detector, and associated circuitry onto a single integrated circuit, or "chip". A simplified block diagram taken from Figure 5.18 of [Pat91b] is given in Figure 2.1.

The SSU utilizes shift-registers to implement the hardware rings. Eight solution taps per ring are routed to separate solution detectors, allowing eight solution candidates to be tested per clock cycle. Each VLSI chip possesses hardware rings whose size correspond to the first 30 prime numbers (2 through 113). However, to accommodate the solution taps, the size of the first four rings are larger than the corresponding primes. These rings are of size $2^4 = 16$, $3^2 = 9$, $5^2 = 25$, and $7^2 = 49$.

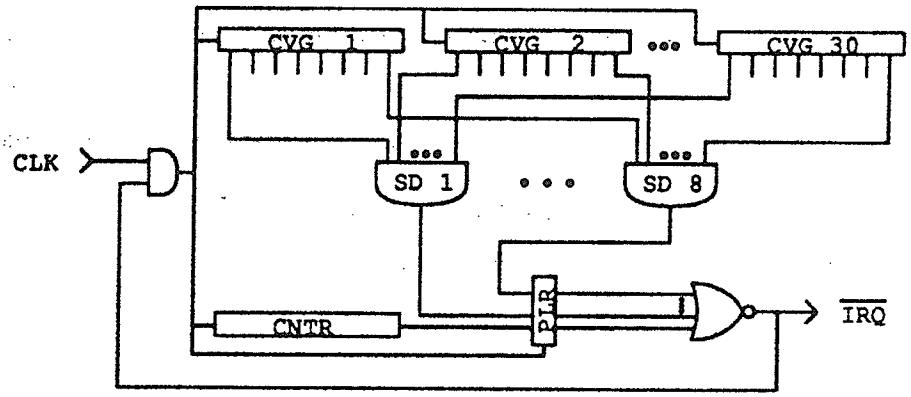


Figure 2.1: SSU Block Diagram

A pipelined solution register (PLR) is utilized to store the outputs from each of the solution detectors (SD). By gating this register with the sieve clock (CLK), a simple two-stage hardware pipeline is formed. In the first stage, solutions are detected using the 8-input AND gates and clocked into the pipeline register. In the second stage, the interrupt request output (IRQ) is generated through the 8-input NOR gate. By dividing the required operations into two stages as opposed to one, fewer gate delays are incurred per clock cycle, allowing a greater clock rate to be used. A minor disadvantage of this approach is that when the interrupt request is generated, and the clock is gated off, the counter (CNTR) will have advanced by one count beyond the corresponding solution.

A solution counter of 44 bits is provided on-chip. The most significant bit of this counter is reserved for indicating counter overflow; thus, the terminal count is limited to 2^{43}. This is sufficiently large that we can expect several days of continuous operation between counter-overflow interrupts. Since one or more solutions may be present during counter-overflow, it is important that the contents of the pipelined solution register be tested for solutions. The contents of the pipelined solution register are referred to as the *Solution Mask*.

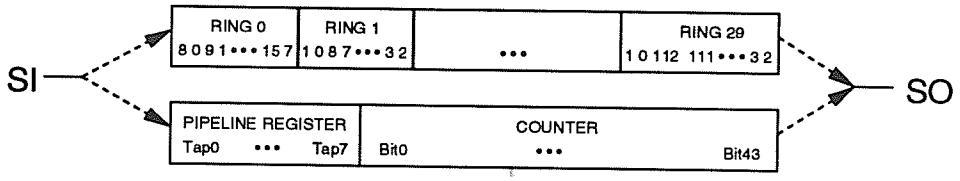


Figure 2.2: SSU Scan Paths

The contents of the SSU rings, counter, and pipelined solution register can be read and written using one of two so-called *scan paths*. These are shown pictorially in Figure 2.2. One scan path is formed by concatenating the contents of the rings, while the other consists of the solution mask concatenated with the value of the counter. Both scan paths are read in a bit serial fashion using a fast synchronous serial link operating at a data rate of over 1 MBPS. The data is clocked using an external bit clock referred to as the *Slow Clock Pulse* (SCP). This is to differentiate it from the *Fast Clock Pulse* (FCP) which controls the shifting of the rings. With each pulse of SCP a single bit of the selected scan path is input via the *Serial Input* (SI), while simultaneously one bit is output via the *Serial Output* (SO).

During operation, the SSU is said to be in one of two states. In the instruction state, the SSU accepts one of four commands via control signal inputs and FCP is gated off to the shift-registers and counter. In the search state, the SSU will search for a solution by clocking the shift-registers and counter using FCP. A transition to the search state will only initiate a search if there are no pending solutions in the solution register, the counter has not overflowed, and the most recent command was not a disable search command.

CMD0	*CMD1*	*Command*
0	0	Scan Rings
0	1	Scan Counter
1	0	Disable Search
1	1	Enable Search

Table 2.1: SSU Commands

Although the SSU is not a stored program computer, it does accept commands via

two signal inputs, CMD0 and CMD1. The *Execute* (EXL) signal is used to initiate the command specified by the CMD0 and CMD1 inputs when in the instruction state. The commands, listed in Table 2.1, are divided into two groups of two instructions each. The two scan instructions are used to simultaneously read and write either the contents of the rings, or the solution mask and counter. The remaining two instructions enable or disable search mode. When search mode is enabled a transition to the search state will initiate sieving for a solution. When it is disabled, the chip is essentially idle until a new command is received. Once a solution is found, or on a counter-overflow, the chip will assert the *Interrupt Request* (IRQL) signal and search mode is disabled.

The *Chip Select* (CSL) control signal input is used to select between the instruction state and the search state. When CSL is deasserted, the SSU enters the instruction state and the IRQL output is placed into a high-impedance state, essentially disconnecting the SSU from the IRQL signal, and the SO output is active. When CSL is asserted, the SSU enters the instruction state and the SO output is placed in a high-impedance state, while the IRQL output is active.

The *Clear* (CLRL) signal is used to perform an asynchronous reset by clearing the pipelined solution register and counter, and disabling the search mode. A complete list of the input/output signals required for interfacing to the SSU device is given in Table 2.2.

The SSU VLSI devices were implemented using a channeled gate array with a circuit complexity equivalent to 10,000 gates. These gate array chips were constructed using a 2μm CMOS double-metal process. This technology was considered mature, even by the standards of 1991, when they were first manufactured. The dies, 1 cm^2 in size, are mounted in a conventional ceramic 48-pin *Dual Inline Package* (DIP). The SSU devices were designed and tested for a maximum clock rate of 33 MHz, but, as we shall see in Section 3.2, due to excessive power dissipation at these speeds, normal operation is at somewhat lower clock rates.

Signal	*Function*	*Type*
CLRL	Clear(Reset)	Control Input
CMD0	Command 0	Control Input
CMD1	Command 1	Control Input
CSL	Chip(Mode) Select	Control Input
EXL	Execute Command	Control Input
FCP	Fast Clock Pulse	Clock Input
IRQL	Interrupt Request	Control Output
SCP	Slow Clock Pulse	Clock Input
SI	Serial In	Data Input
SO	Serial Out	Data Output

Table 2.2: SSU Signal Lines

2.3 MSSU Overview

The MSSU hardware consists of two major subunits housed in a single chassis (see Figure 2.3). The first of these is the control processor which handles various control operations and performs on-line solution filtering. The control processor consists of a high speed microprocessor, memory, and various I/O interfaces. The second major subunit is the sieve array which contains up to 32 SSU devices and associated circuitry to interface these devices to the control processor.

Because of the limited number of moduli supported by the SSU devices it was decided to implement fast emulation of virtual rings in the control processor software. It was also decided to place the filtering functions of the sieve in the control processor, thereby eliminating the need for a fast communication channel between the sieve hardware and the multi-user host computer which had traditionally been the bottleneck with OASiS. As much functionality as possible was placed in the control processor, without affecting overall system flexibility, to reduce the response time requirements of the multi-user host. These factors implied that the control processor would have to be quite powerful, with a fast synchronous serial interface to communicate directly with the SSU devices.

Software executing on the controller recognizes a small number of high level commands

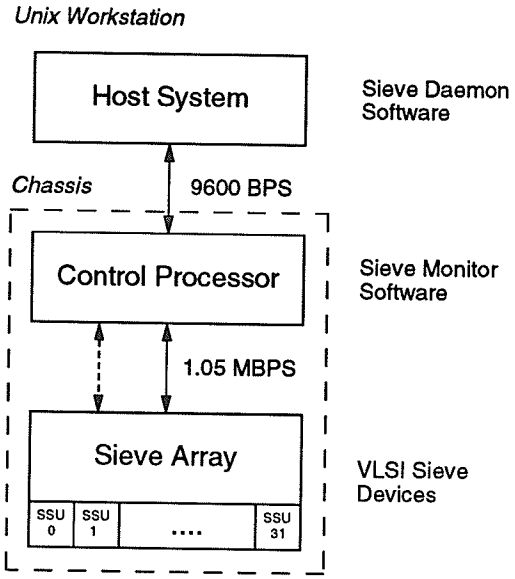


Figure 2.3: MSSU Block Diagram

from a host via a conventional serial port and translates them into the necessary low level operations to be performed by the individual sieve chips. Typically, the host is a multi-user workstation; however, if necessary, a dumb terminal can be used. Since the control processor reduces the level of interaction required between the host and the sieve hardware, adequate communication bandwidth is provided by a 9600 Bits Per Second (BPS) asynchronous serial line. However, to maximize performance, data rates in excess of 1.05 Million BPS are provided for data transfer between the sieve array and the control processor.

2.4 Comparison of MSSU with Recent Systems

Table 2.3 lists some of the key features of the UMSU, OASiS and MSSU sieves. Each system has benefited from various technologies which were available and economically feasible when these systems were designed. Clearly the major features present in the MSSU sieve which were not available in the UMSU and OASiS sieves are the parallel

architecture and the powerful dedicated control processor capable of performing on-line solution filtering.

In retrospect, some design decisions of the earlier sieves would not be duplicated today (1995). For example, one of the main goals of the OASiS system was software portability. Unfortunately, the software system relies heavily on many features of VAX/VMS, an operating system whose relative popularity has declined with the emergence of Unix as the operating system of choice for minicomputers and workstations. Today most portable software packages are written for Unix platforms. However, one portability feature of OASiS which was very successful was the use of a standard serial port for interfacing the sieve to the host system. This feature was retained in the design of the MSSU to allow the sieve to be easily and quickly connected to any suitable Unix workstation, obviating the need for any special interface cards, like those required by UMSU.

The design of the MSSU hardware was significantly simpler than that of the UMSU and OASiS sieves. Since most of the circuit complexity of the MSSU sieve is contained within the VLSI sieve chips themselves, comparatively very little digital logic circuitry is required when compared to UMSU and OASiS. This reduced complexity made feasible the construction of a wire-wrapped 16-chip prototype, used to debug the hardware prior to implementing the printed circuit boards used in the final version of the hardware.

Feature	*UMSU*	*OASiS*	*MSSU*
Year Completed	1982	1988	1993
Rings	32 fixed	16 variable	30 fixed
Number of independent sieves	one	one	1–32
Trials per second (millions)	133	215	192 (6144 aggregate)
Control CPU	AMD 2901	Motorola 6801	Motorola 68332
Control CPU technology	bit-slice	8-bit μP	32-bit μP
Interconnect technology	wire-wrap	2-layer PCB	4-layer PCB
Circuit technology	MSI	MSI & LSI	MSI & VLSI
Number of ICs	$\approx$ 500	$\approx$ 300	34
Host interface	16-bit parallel	9600 baud serial	9600 baud serial
Filtering	host-based	host-based	on-board
Host CPU	PDP-11/45	μVAX-II	Sun workstation
Host operating system	Unix V7	VAX/VMS	SunOS 4.1.x

Table 2.3: Comparison of Recent Electronic Sieves

Chapter 3

MSSU Hardware

This chapter describes the implementation of the MSSU hardware. This includes the dedicated control processor, the sieve array circuit board, and the chassis in which the MSSU is housed. The software components of the system will be described in the following chapter(s).

3.1 Control Processor

The major factors considered in selecting a suitable processor for control of the MSSU were: speed of computation, ease of programming and ease of interfacing to the SSU devices. To prevent the functions of the control processor from being a bottleneck a very fast processor was required. An 8-bit microprocessor like the one used in the OASiS design would simply be overwhelmed by the amount of computation required. With respect to ease of programming, should assembly language programming be necessary, the author favoured a Complex Instruction Set Computer (CISC) over that of a Reduced Instruction Set Computer (RISC). For high-level language programming, the availability of high-quality development tools such as compilers and debuggers were the main concern. Additionally, since the control processor would be required to interface directly to the SSU

devices, the availability of certain on-chip input/output (I/O) interfaces would obviate the need for the design and construction of complex external circuitry.

The nature of sieving is such that the greatest part of the processing time will likely be devoted to multi-precise integer operations during solution filtering. To support the 128-bit or 256-bit integers required by the MSSU software, fast integer execution was considered more important than the availability of fast floating point hardware. The size of these integers suggested performance would benefit from a large processor word-length of 32 bits or greater. Another useful capability is that of hardware multiply and divide since multi-precise multiplies and divides can be easily synthesized from these instructions.

The control processor that was selected for the MSSU is the Motorola MC68332, a variant of the popular 680x0 family of microprocessors designed for use in embedded control applications. The MC68332 is a high performance microprocessor which utilizes the CPU32 core, an enhanced version of the MC68020 processor, with internal 32-bit data paths and a 16-bit external data bus.

The CPU32 provides many features to enhance processing speed. The on-chip Arithmetic Logic Unit (ALU) is based on the MC68030 processor, with full 32-bit address and data paths, a barrel-shifter and fast multiply and divide instructions. The CPU32 implements a two word instruction pipeline, a prefetch queue, and a write-pending buffer to reduce the bandwidth requirements over the 16-bit external data bus. A so-called *fast-termination* feature allows two-clock memory access, whereby memory can be read or written within two system clock periods. At 16.78 MHz the CPU32 benchmarks at 6742 Dhrystones (V1.1) per second, or approximately 3 million instructions per second (MIPS).

The 680x0 family is based on a CISC architecture which dates back to the introduction of the original device, the M68000, in 1979. The instruction set is not unlike that of the VAX in that a rich number of addressing modes are available which behave in a consistent

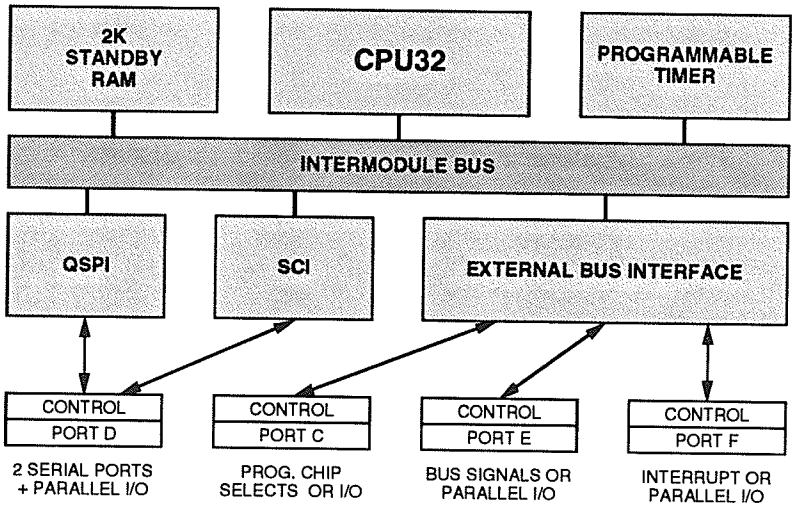


Figure 3.1: MC68332 Block Diagram

manner independent of the instruction being performed. The 680x0 instruction set is considered well suited for assembly language programming, and in fact is often used for teaching introductory assembly language programming to undergraduates.

The CPU32 supports many of the advanced addressing modes of the 680x0 architecture, including the scaled index, 32-bit branch displacements, and address register indirect with base displacement and index; to name a few. The instruction set enhancements include a 32-bit by 32-bit multiply instruction yielding a 64-bit result, with a corresponding 64-bit divide instruction. The only MC68020 instructions not implemented by the CPU32 are the bit-field instructions, the CALL and RTM instructions, the PACK and UNPK binary coded decimal (BCD) instructions, and the CAS and CAS2 read-modify-write instructions. Fortunately, these instructions are used infrequently in most applications.

The MC68332 processor provides many hardware features which make it ideally suited for use as the MSSU control processor. It integrates numerous peripherals on-chip to improve system throughput and reduce the external part-count (see Figure 3.1). The most important of these, with regard to the MSSU, are the two on-chip serial interfaces. The Serial Communications Interface (SCI) is an asynchronous serial interface used to imple-

ment the communication link between the Unix host system and the control processor. The Queued Serial Peripheral Interface (QSPI) is a synchronous serial interface capable of data rates in excess of 1 million bits per second suitable for data transfer between the SSU devices and the control processor.

In addition to the on-chip serial ports, the MC68332 provides approximately 32 programmable parallel I/O pins which are available for control signals between the control processor and the sieve array. Control registers are provided for each of the four ports which determine whether the specified bit of that port is a general purpose input, or output, or assumes some other function such as an external bus signal or interrupt input. Since some of these pins are required for their other functions, only 18 parallel I/O pins are available to interface to the MSSU sieve array.

The external bus interface provides a programmable chip select feature which eliminates the need for address decoding circuitry for external RAM or EPROM memory. For applications requiring non-volatile memory, 2KB of very low-power standby RAM is provided on-chip.

The MC68332 also provides a very versatile programmable timer unit known as the Timer Processing Unit (TPU). However, since the programmable timer feature is not required by the MSSU software, the control processor can be replaced by the somewhat less expensive MC68331 microprocessor. The MC68331 is identical to the MC68332 except it replaces the complex TPU module with a simpler General Purpose Timer (GPT) unit.

When the MSSU design was being finalized, the fastest CPU32 processors available were rated at 16.78 MHz; however, at that time, 25 MHz parts were under development. It was envisioned that the MSSU control processor would be upgraded as soon as these faster parts became available. To allow for this future upgradability it was decided that the functions of the sieve array and the control processor should be located on separate circuit boards. Rather than design such a processor board from scratch, incurring additional cost

and development time, a commercially available processor board, the Motorola 68332 EVS, was used.

The Motorola 68332 EValuation System (EVS) provides up to 64KB (kilobytes) of fast static RAM, 32KB of slow static RAM, 64KB of user programmable EPROM, and 64KB of EPROM reserved for the CPU32BUG monitor/debugger software provided with each kit. The fast static RAM devices have an access time of under 25 nanoseconds allowing two cycle memory access using fast-termination bus cycles. The remaining 32KB of RAM, which is not socketed, has an access time of 85 nanoseconds requiring two wait states for every memory access. The EVS also provides an RS232 interface for the asynchronous serial port, as well as access to all MC68332 on-chip resources such as the QSPI synchronous serial interface and the parallel input/output pins.

Since the development of the MSSU, a second 68332 single board computer has become commercially available. The Vesta SBC332 provides a 68332 processor board on a very compact circuit board. The main advantages of the Vesta SBC332 over the Motorola EVS are increased RAM and EPROM capacity and a socketed CPU chip. The standard Vesta SBC332 has only a 16.78MHz processor, however this can be easily removed and replaced with a 20 or 25MHz device. On the other hand, the 16.78MHz MC68332 in the Motorola EVS is not socketed and thus cannot be upgraded to a faster processor.

3.2 Sieve Array

The sieve array is composed of one or two sieve array boards each of which contains up to 16 SSU devices, allowing for a maximum configuration of 32 sieve chips per MSSU. Jumper settings are used to identify the sieve array board as either the first or second board in a system.

Removable flat (ribbon) cables interconnect the required signals between the sieve array board(s) and the control processor. Since processor boards such as the Motorola

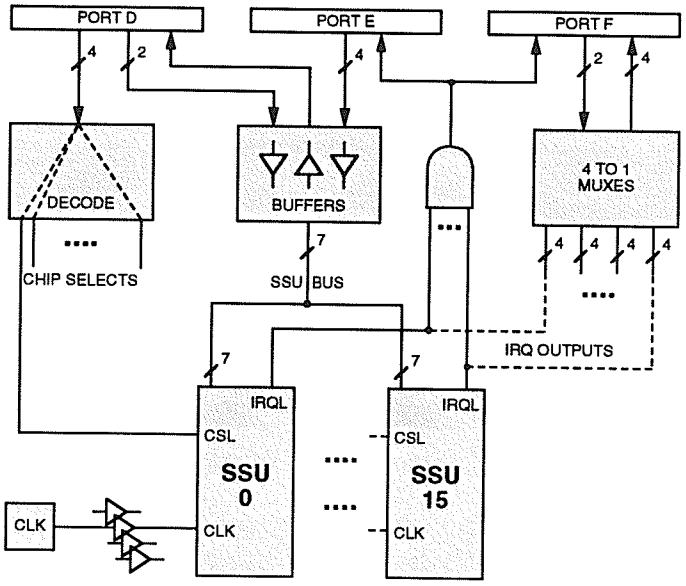


Figure 3.2: Simplified Schematic of the Sieve Array Board

EVS do not provide a single connector for accessing all of the desired 68332 signals, it was convenient to use a wire-wrapped adapter board to mate the interconnecting cable to the processor board. An adapter board also provides a mechanism for easily wiring a future replacement processor, such as the Vesta SBC332, to the sieve array.

Each sieve array board contains: a high-speed clock, a single interrupt request output, circuitry to identify sieve chips requesting an interrupt, a decoder to select an individual sieve chip, and drivers which form an *SSU bus* which interconnects all the sieve chips on an array board (see Figure 3.2). For most of the signals on the sieve bus to have effect, the desired SSU chip must first be enabled via the decoder. This allows the control processor to service interrupts on one SSU device, while the others continue sieving.

To determine which device requires interrupt servicing, the control processor can determine whether board 0, or board 1, or both boards have pending interrupts. Unfortunately, it can only retrieve the interrupt status of 4 sieve chips from one board at a time, therefore, up to 4 probes may be required to find a pending interrupt. Ideally 16

inputs should be used; however, the control processor simply does not have that many parallel I/O pins. Eight interrupt inputs with differing priority levels are provided, but from the standpoint of fairness, it is desirable to assign the same priority to all sieve chips. As it turns out, multiplexing the interrupt status of 16 sieve chips onto 4 status lines slows down interrupt processing very little and is the simplest scheme to implement in hardware.

The sieve array board was implemented as a four-layer *Printed Circuit Board* (PCB) consisting of two outer signal layers and two inner power planes. This is more costly than the conventional two-layer PCB design, but it greatly simplifies the layout of the board by eliminating the need to route power traces. The dedicated ground and power (Vcc) layers provide excellent power distribution and the metal planes act as heatsinks for attached components. By reserving the two outer board layers exclusively for signal routing, the length of critical signal paths can be minimized and potential sources of cross-talk avoided by the careful placement of traces.

Design of the sieve array was aided by the use of Tango PCB layout software. This software runs on IBM PC-compatible microcomputers and can perform automated routing. A number of critical signals were routed manually, after which auto-routing of the remaining signals was performed. The two or three connections which could not be auto-routed were routed manually. Component placement was determined visually and by trial-and-error, using the number of signals which could be auto-routed as a measure of optimal placement.

In particular, great care was taken in the routing of the high-speed clock to each of the VLSI sieve chips. The crystal oscillator and clock buffers were placed in the center of the array board to minimize trace lengths. One clock buffer was provided for every four adjacent sieve chips to provide adequate buffering.

Figure 3.3 shows the artwork on the component side of the final version of the PCB. Connectors at the top left corner of the board are used to interconnect the sieve array to

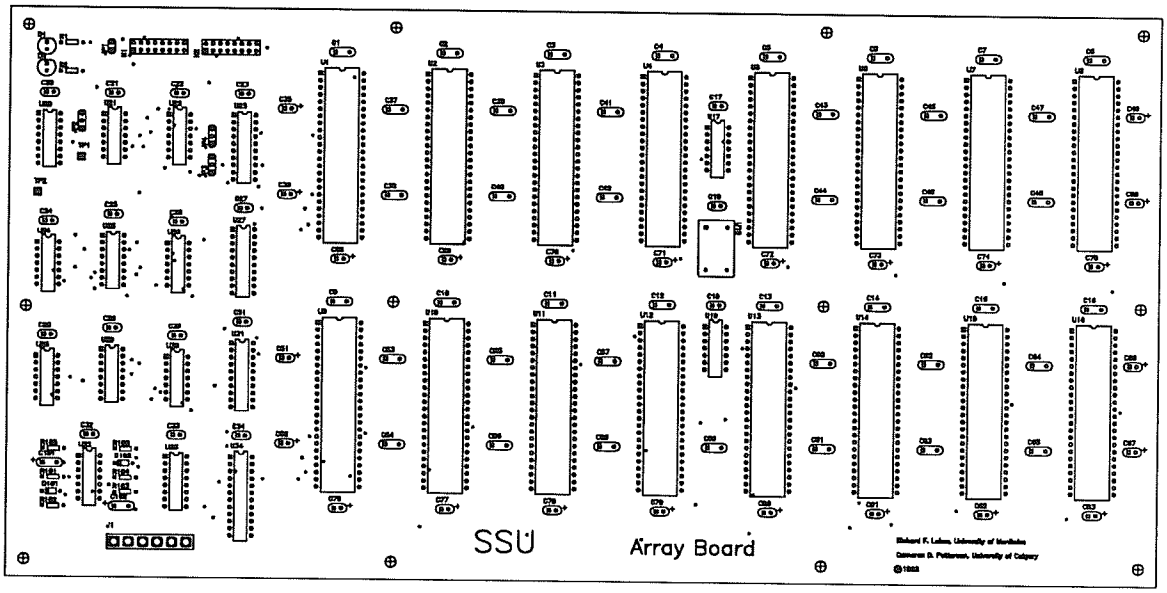


Figure 3.3: Sieve Array Board Layout

the control processor. A power connector at the lower left corner of the board connects to a 5 Volt DC power supply.

Although the SSU devices were designed and tested with a maximum clock rate of 33 MHz, at this speed the chips require approximately $400ma$ of current and consume nearly 2 Watts of power. This is considered near the limit of power dissipation of the ceramic 48-pin DIP. Since power dissipation of CMOS circuitry is a linear function of switching frequency, we limited the clock rate to 24 MHz to prevent the possibility of overheating and to increase the long-term reliability of the devices. At 24 MHz the SSU device draws approximately $250ma$ of current, equivalent to 1.25 Watts of power consumption, enough to make the chip warm to the touch. To ensure adequate heat dissipation, two black anodized aluminum heatsinks are bonded to the surface of each chip using a heat-conductive glue. At 24 MHz the effective sieving rate of a single SSU device is 192 million tests per second.

3.3 Sieve Chassis

The control processor and two sieve array boards are housed in a rack-mount chassis along with a power supply and six high volume cooling fans. The chassis is 19 inches wide, 10.5 inches high, and 20 inches deep, weighing approximately 40 pounds. Chassis sliders allow easy access to the chassis cover and interior of the unit by sliding the unit in and out of the rack (see Figure 3.4).

The airflow in the chassis is directed across the length of the sieve chips to increase heat dissipation. Three fans on one side of the chassis direct air into the chassis, while the three fans on the opposing side expel air out of the chassis. To pack 32 sieve chips into the chassis it was necessary to stack the sieve array PCBs; unfortunately this tends to restrict the flow of air to the center of the bottom circuit board. This problem is overcome by concentrating the airflow to this area of the chassis.

Since the dynamic power dissipation of the CMOS sieve devices increases with the square of the supply voltage, the power supply voltage setting was carefully adjusted to slightly less than 5V, yet within operating range, to minimize power dissipation. For electrical safety, the sieve unit is fused and to prevent electrical noise on the power line from entering the unit, a line filter was installed.

A power on-off switch and reset button provide the only external controls. Two LED lamps on each array board are visible through a transparent front panel. A green lamp gives indication that DC power is applied to the board. A red lamp is used to indicate that an interrupt request is pending. Monitoring this lamp provides a good indication of the processing load on the control processor.

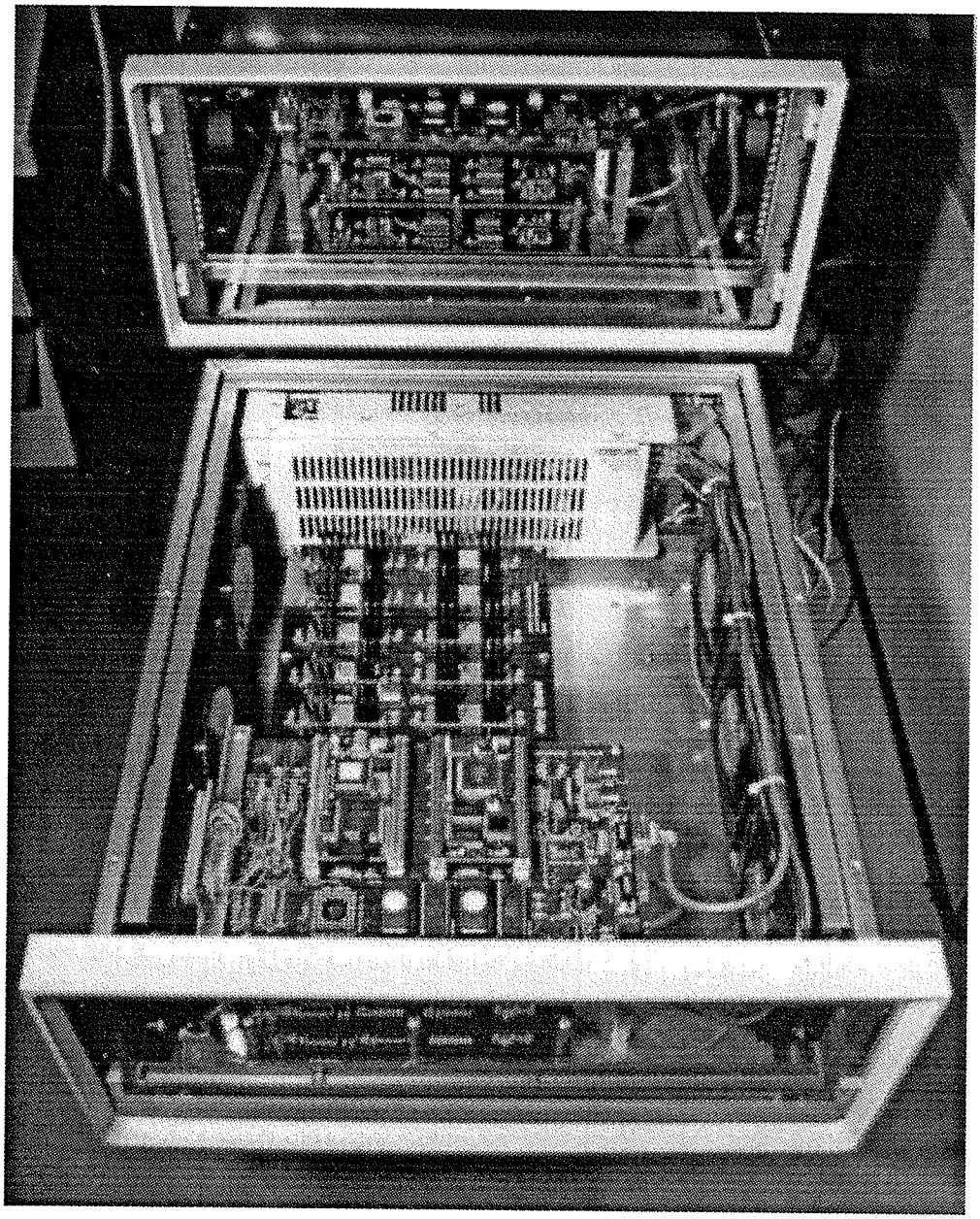


Figure 3.4: Two MSSU sieves (one extended from rack with cover removed).

Chapter 4

MSSU Software: The Control Processor Software

In order to present this material in a logical manner, the description of the MSSU software has been divided into two chapters. In this chapter we detail the software which executes on the control processor called SSUMon (*SSU Monitor*).

4.1 The Function of SSUMon

The main function of SSUMon is to supervise the low-level operations required for sieving; this includes providing a means to load a problem onto the sieve, to initiate sieving, to output solutions, and to stop sieving. Inherent in the process of producing solutions is the ability to specify some filter function and have the control processor apply it to all solution candidates.

Whereas earlier sieve designs such as OASiS relied on a multi-user host computer system to perform the function of filtering, the MSSU utilizes a dedicated control processor with obvious performance advantages. First, since no other processes are competing for

resources, response time can be almost instantaneous with a dedicated control processor. Second, by situating the control processor in close proximity to the sieve hardware, data rates in excess of 1MBPS can be reliably supported over inexpensive serial lines. Finally, a fast control processor can rival the performance of a workstation at a fraction of the cost.

To prevent the control processor from being the bottleneck in the performance of the MSSU system, SSUMon implements many features to enhance speed and efficiency. Computationally intensive code, such as filter routines, were carefully coded for maximum speed. Instead of employing inefficient polling for solutions, SSUMon solution processing is interrupt-driven, thereby reducing the latency from interrupt arrival to interrupt servicing. A one solution pipeline is implemented so that solution processing can be performed concurrently with sieving, enabling the sieve chips to generate the next solution while the current solution is processed.

Like the previous UMSU and OASiS sieve systems, MSSU provides a *checkpoint* capability for use in detecting hardware errors. Checkpoints are also used to establish the counter value to which sieving has progressed, so that if a hardware fault is detected, computation can be rolled back to the location of the previous successful checkpoint. In the case of the MSSU, the checkpoint is initiated by the host system, but is performed locally by the control processor. This requires minimal transfer of data to and from the host system. In the OASiS system, checkpoints are particularly time-consuming, as the contents of all relevant hardware registers are first transferred to the host system where all *consistency checks* are performed. Due to the amount of data involved, and the slow data rate of the link between OASiS and the host computer, on average, checkpoints take 3-4 minutes to complete. An equivalent checkpoint operation in the MSSU system takes 2-3 seconds.

Another function of SSUMon is the implementation of capabilities in software which are not available in the underlying hardware. For example, due to limitations of the VLSI

technology used, the SSU devices contain rings corresponding only to the first 30 primes. For some sieve problems, this necessitates a lot of post-processing to eliminate solutions which would most likely have been rejected had a greater number of hardware rings been available. To circumvent this limitation, SSUMon implements 20 additional virtual rings which are emulated in software.

Finally, because MSSU is a parallel machine, SSUMon must implement mechanisms which permit sieve problems to be easily adapted to the parallel architecture. To facilitate this, SSUMon implements two different strategies for running sieving problems on the MSSU. The first method is referred to as *partitioning*, or *single-variable sieving*, and allows a single sieve problem to be partitioned into sub-problems each of which is executed in parallel on an SSU device. The second method is intended for *multivariable sieving* whereby one distinct sieve problem is assigned to each SSU device and solutions over a given range are output.

4.2 The Choice of Programming Language

One important decision in the design of the MSSU was the choice of programming language for implementing the control processor software. The Motorola EVS provides a cross-assembler named AS32 which uses a PC-compatible computer as a development platform. This assembler is of the "no frills" variety, however it is reliable and provides support for the complete CPU32 instruction set. For high-level language development, there exist several commercial C cross-compilers which target the CPU32 processor. Another option is to adapt the free GNU C Compiler (GCC) which can, with some customization, generate CPU32 code in a cross-compiler configuration. Regardless of the language, it is necessary to employ cross-development on a Unix or PC platform since it is not feasible to provide hardware support for a full-scale operating system with secondary storage on the control processor.

Due to the low-level nature of SSUMon, it was assumed that some routines, where

speed of execution was critical, would have to be written in assembler for optimum performance. However, it was unclear whether assembler should be used throughout, or used in conjunction with a high-level language. The reduced code size generated by assembly code versus compiler-generated code was considered an advantage, as RAM space on the Motorola EVS is limited. On the other hand, the most obvious disadvantages of the assembler approach are the difficulty of programming and the lack of portability.

At the time when the implementation language was being chosen, several commercial C Language cross-compilers existed, however their reliability and functionality were unknown, and their cost was considerable. Most cross-compilers provided language extensions or in-line assembler capability to perform low-level tasks, such as interrupt processing. Unfortunately, it was felt using these features could result in non-portable code. Furthermore, it was unclear which language extensions were necessary or desirable prior to starting the implementation of SSUMon.

Use of the freely-distributable GNU GCC cross-compiler was examined, however this would have required significant effort in customization for CPU32 code generation, support for the C standard library, and a suitable loader and linker. More discouraging, however, was the news that 68K support in GCC was slated to be dropped in the near future. At that time (1993), 68K support in GCC was suffering from "bit-rot" as the majority of more recent development efforts had concentrated on newer architectures such as the Power-PC and Alpha. Since that time, however, the situation has improved dramatically with several CPU32-specific ports of the GCC cross-compiler and associated tools being made available under the GNU General Public License.

Development of SSUMon began with small test programs written in assembler to test the suitability of the MC68332 QSPI for interfacing with the SSU synchronous serial interface. As the author became more familiar with the CPU32 architecture, concerns about difficulty in programming in assembler diminished. And whereas the programmer has no control over software bugs introduced by a poor cross-compiler, by careful

programming, the assembly language programmer can have total control over the quality of generated code. The design of the hardware was so closely tied to the Motorola 68332/68331 microprocessor and the QSPI interface that questions about portability were rendered moot. Thus, development proceeded exclusively in assembler using the Motorola AS32 cross-assembler.

One of the most problematic issues relating to the portability and maintainability of SSUMon has to do with the difficulty in adding new solution filters. In the final design, new solution filters must be written in assembler, certain internal SSUMon tables must be updated to reflect the addition of the filter, and the new filter code must be incorporated into the SSUMon object code. This is significantly more involved than writing a filter in the UMSU or OASiS systems, because the filter now executes on the dedicated control processor and not the host system. Unfortunately, little would have been gained by using a high-level language such as C, since the filter code would still have to be cross-compiled for the CPU32 and then somehow transferred onto the control processor. No simpler mechanism to permit the use of (dynamically loaded) user-defined filters was apparent. Nevertheless, writing a new solution filter is not as daunting as it may sound, and the knowledge required by the programmer of the inner-workings of SSUMon or of any subroutine linkage protocols is minimal.

4.3 SSUMon Overview

SSUMon consists of approximately 8,000 lines of CPU32 assembler, translating into approximately 28 kilobytes (KB) of object code. The code was written to be modular and follow accepted software engineering techniques. A great deal of programming discipline was employed to make SSUMon maintainable and reliable. All local variables are dynamically allocated on the stack. All registers modified by a subroutine call are saved and then restored on exit. Parameters are passed either via the stack or by CPU registers.

In order to impose a logical structure which is both easier to understand and maintain,

the SSUMon source code was organized into a number of code modules. Each module corresponds to a file containing source code with similar routines or data. The names of the these modules are listed in Table 4.1.

Module	*Description*
`GLOBDEFS`	Global definitions.
`START`	Start of code execution.
`RTNS332`	Routines accessing 68332-specific resources.
`INTERRUP`	Interrupt handler routines.
`LOWLEVEL`	Low level routines common to various modules.
`VLSIRTNS`	Routines which manipulate the SSU devices.
`SIVRTNS`	Pertaining to the logical sieve device.
`FILTER`	Filter routines.
`MONITOR`	The SSUMon command interpreter.
`IORTNS`	Input/Output routines.
`STATIC`	Static data.
`VARIABLE`	Variable definitions.
`ROMSTART`	Boot code which copies EPROM to fast RAM.

Table 4.1: SSUMon Modules

4.3.1 CPU32BUG Resources

SSUMon is almost completely self-contained, however, it does make use of the Motorola EVS CPU32BUG monitor software to provide routines to manipulate the SCI serial interface. The CPU32BUG routines include `InStat()` which returns the status of the input character buffer. `InStat()` also allows a program to determine if an input character is present without actually inputting it. The routine `InChr()` is used to perform the actual read. A corresponding `OutChr()` routine is used to output a single character. Buffered I/O is not provided, however the XON-XOFF protocol is implemented for *flow-control*. This feature is utilized by the sieve system software to prevent loss of characters between the control processor and the Unix host.

Another feature of CPU32BUG which is utilized by SSUMon is the *boot-load* feature. On boot-up, CPU32BUG will automatically transfer control to an address specified in a

user-customizable area of the CPU32BUG EPROM. This permits SSUMon code to reside in EPROM located in a convenient area of the control processor address space. Although it is easy to write code for the CPU32 which can be executed from read only memory (i.e. ROMable code), the slow memory access time of the ROM makes it advantageous to first copy the contents of the ROM to RAM, and then transfer execution to the faster RAM memory. This scheme is utilized by SSUMon since EPROM is subject to 4 wait-states per memory access on the EVS, while the fast RAM incurs no wait-states.

Reliance on these features of CPU32BUG does not imply that SSUMon is tied to use of the Motorola EVS processor board. CPU32BUG can be easily customized to work with just about any MC68332 CPU board, including the Vesta SBC332.

4.4 Implementation of Residue Optimization

One of the most important concepts in understanding the MSSU system is how residue optimization is implemented. The MSSU supports a form of single-residue optimization whereby one residue and one implicit modulus are specified per chip. With this method, if a suitable congruence $x \equiv A \pmod{M}$ exists, instead of sieving on x one can sieve on y, where $x = My + A$, and thus effect an M-fold speedup of sieving.

To specify single-residue optimization, SSUMon provides two variables named `OPTMUL` and `OPTADD`, also referred to as the *optimization registers*. `OPTMUL` corresponds to the modulus of the single-residue congruence used for optimization, while `OPTADD` corresponds to the value of the residue. For example, in the pseudosquares problem, the congruence $x \equiv 1 \pmod{8}$ can be "optimized-out" by setting `OPTMUL` $= 8$ and `OPTADD` $= 1$. In this case, SSUMon automatically transforms the ring data and counter value to assume only solution values which are $1 \pmod{8}$, for an 8-fold speedup in the sieving rate. Of course, by setting `OPTMUL` $= 1$ and `OPTADD` $= 0$, residue optimization is effectively disabled and sieving proceeds at the expected rate.

4.4.1 Partitioning

The MSSU can perform a parallel form of single-residue optimization called partitioning to achieve extremely high sieving rates. This mechanism can be used to divide a single problem into a number of sub-problems executing in parallel on multiple sieve chips.

For example, the pseudosquares problem specifies the following congruences:

$$x \equiv 1 \pmod 8$$
$$x \equiv 1 \pmod 3$$
$$x \equiv 1,\ 4 \pmod 5$$
$$x \equiv 1,\ 2,\ 4 \pmod 7$$

By combining these congruences into a single congruence, we have the equivalent congruence having 6 residues modulo $8 \cdot 3 \cdot 5 \cdot 7 = 840$:

$$x \equiv 1,\ 121,\ 169,\ 289,\ 361,\ 529 \pmod{840}$$

By assigning each of these residues modulo 840 to one sieve chip using single-residue optimization, the MSSU can partition the pseudosquares problem to run in parallel using 6 sieve chips. The hardware sieve rate is increased by a factor of 840 over that of a single sieve chip without single-residue optimization.

In the case of the pseudosquares, this idea can be extended by combining the congruence modulo 11 with the above set to form 30 residues modulo 9240, etc.

4.4.2 The Counter

For each of the SSU devices, SSUMon maintains a `COUNT` variable which corresponds to the current trial counter. Figure 4.1 shows the variables and operations involved in calculating the `COUNT` value.

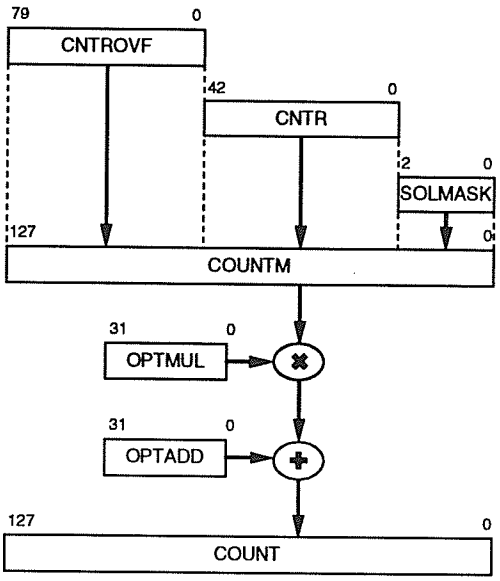


Figure 4.1: Calculation of the COUNT Value

The SSUMon routine Hw2Cnt() is used to concatenate the 80-bit CNTROVF (counter overflow), the 43 bits of the SSU hardware counter CNTR, and 3 bits derived from the 8-bit SOLMASK (solution mask) to form the COUNTM value. The Optimize() routine is used to multiply the COUNTM value by OPTMUL and add the OPTADD value to form the 128-bit COUNT variable. If no overflow is detected from these operations, then COUNT will hold the value of the solution.

Recall that the SSU has a solution window size of 8, so up to 8 solutions can be detected at one time. Effectively, the hardware counter is counting up in multiples of 8. If no solution is present, the last 3 binary digits of COUNTM are considered to be zero. However, for each solution detected, the lowest 3 binary digits of COUNTM will correspond to the bit position (0–7) of a particular solution in the solution mask register.

The CNTROVF variable is merely a software extension of the SSU hardware counter. Every time the hardware counter overflows the CNTROVF variable is incremented. Running at the maximum hardware sieving rate, we can expect the SSU counter to overflow

approximately every 4.24 days of continuous sieving.

The COUNT variable is not only useful to calculate solution values, but also for determining how far sieving has progressed. For example, when an upper bound is specified in a sieving problem, the sieve can be periodically halted, and the COUNT value compared to the bound. If COUNT exceeds this bound, the sieve job has completed and should be terminated.

For initiating a sieve problem, it is required to perform the reverse operation to that of Figure 4.1. These two operations are somewhat asymmetrical, since the Hw2Cnt() routine updates the COUNT and COUNTM variables based on the hardware registers; however, the corresponding routine Cnt2Hw() updates CNTROVF, CNTR, and SOLMASK based only on the COUNTM value. It really would not make sense to use the value of the COUNT variable instead, and thus "hide" the underlying optimization, because the host system must be aware of this process in any event to set the values of the optimization registers. If the sieve problem is to start sieving at a counter value of S, then COUNTM must be initialized to $\lfloor S/\texttt{OPTMUL} \rfloor$ by the host.

4.4.3 Ring Loading

Before the residue information for a sieve problem can be loaded onto the SSU hardware rings, the ring data must be organized in a specific manner. For SSUMon, the residue information is specified as a string of 1675 binary bits where a binary "0" represents an unacceptable residue, and a "1" represents an acceptable residue. The residues are assigned for each ring starting with the largest hardware ring of size 113. Here the residues are assigned sequentially from residue 0 to 112. This continues for each ring in turn, in order of decreasing modulus, with the exception of the last four rings. The last four rings correspond to the primes 7, 5, 3 and 2, but are actually of size $7^2 = 49$, $5^2 = 25$, $3^2 = 9$, and $2^4 = 16$ (see Figure 4.2).

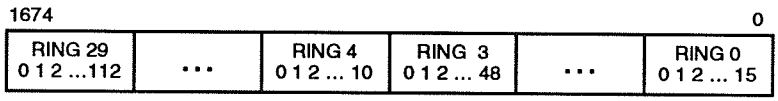


Figure 4.2: Layout of `RINGDAT`

Figure 4.3 shows how the ring data is transformed from the sequential ordering in the `RINGDAT` variable, to the ordering required by the SSU hardware in `RING`. This operation is performed by the routine `Ring2Hw()`. First, the binary information corresponding to the 1675 residues which make up `RINGDAT` are converted to byte-oriented data for easy manipulation. The data is then re-ordered according to the optimization register values to support single-residue optimization. The `ReOrder()` routine is called once for each of the 30 hardware rings which make up `RING` to perform this function. Next, the ring data must be mapped to align the data with respect to the starting `COUNTM` value, and then aligned again to assume the bit ordering required by the hardware rings. This too, must be performed separately for each of the hardware rings. Once this is complete, it is again converted to bit-oriented data and stored in the `RING` variable.

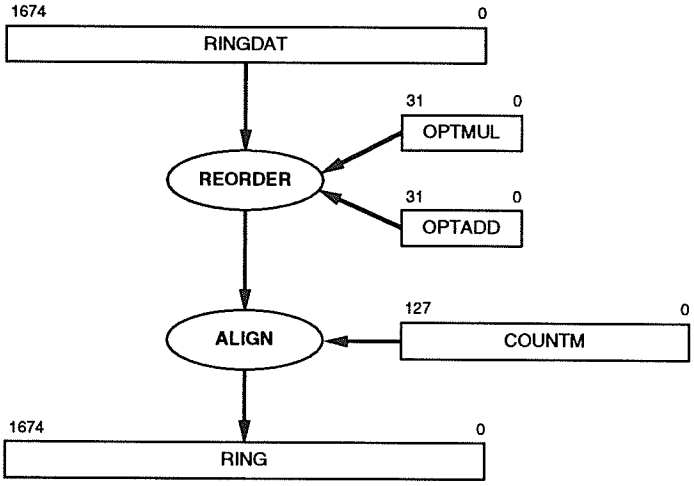


Figure 4.3: Operations on the Ring Data

A pseudocode description of `ReOrder()` is given in Algorithm 4.1.

Note that if the implicit modulus (`OPTMUL`) is a multiple of a hardware modulus, then all of the residues of that hardware ring will be set to acceptable. Effectively, that hardware ring is "optimized-out" since it will not reject any solution candidates. In fact, it is possible that the value of `OPTMUL` can be divisible by several of the hardware moduli, thus eliminating several hardware rings from contributing to the sieving problem. Where alternatives exist for the implicit modulus, the user should carefully consider the benefits of increased sieving rate against that of maximizing the use of hardware rings to eliminate solution candidates.

Algorithm 4.1 *Ring Re-Ordering*

FUNCTION *ReOrder*(*OptMul*, *OptAdd*, *Cong*) **RETURNS** Congruence ;
INPUT *OptMul* : Int32 ;
INPUT *OptAdd* : Int32 ;
INPUT *Cong* : Congruence ;

i : Int32 ;
loop : Int32 ;
residues : Array of Residues;

$i \leftarrow OptAdd \bmod Cong.Modulus$;
FOR $loop \leftarrow 0$ **TO** $(Cong.Modulus - 1)$ **DO**

    $residues[loop] \leftarrow Cong.Residues[i]$;
    $i \leftarrow (i + OptMul) \bmod Cong.Modulus$;

ENDFOR
{Copy the re-ordered residues back to the congruence}
FOR $loop \leftarrow 0$ **TO** $(Cong.Modulus - 1)$ **DO**

    $Cong.Residues[loop] \leftarrow residues[loop]$;

ENDFOR
RETURN (*Cong*) ;

End of Algorithm

4.5 Checkpoints

Two approaches were considered in the design of the SSUMon checkpoint mechanism. Checkpoints could be initiated locally by the control processor's making use of the on-board programmable timer facility. Or, alternatively, checkpoints could be initiated by the host system by a sequence of SSUMon commands. The latter approach was chosen for two reasons. First, the checkpoints need to be logged in the problem file, and it is natural that this should be initiated by the host. Second, various routines for determining the date and time and for scheduling various events (alarms) were available on the Unix host system, but most of these would have to be written for use on the control processor.

A checkpoint operation involves several separate SSUMon commands. First, the host must halt the sieve, and then the state of all hardware registers must be read for each enabled chip. Next, the consistency check portion of a checkpoint is required to determine whether a hardware fault has occurred. After this, the host issues a command to determine the count value for which sieving has completed. The random nature of sieving problems means that some sieve chips will have progressed farther in the sieving, as others are slowed by more frequent solution candidates, or perhaps solution candidates which are more time-consuming to filter. The lowest count value from all of the enabled chips is returned. At this point, if no hardware errors were encountered, the host system will restart the sieve from the point were sieving last left off. Should errors be detected, sieving will restart at the counter value of the last successful checkpoint maintained by the host software.

To perform a checkpoint the current value of `RING` is compared to that expected by calling `Ring2Hw()` with the initial residue information stored in `RINGDAT`, the present value of `COUNTM` and the optimization registers. If there is any discrepancy, then a hardware error has occurred, and an error message of the form "E-CHECKPOINT FAILED n BYTE(S) DISCREPANCY, CHIP m" is generated.

Two types of error are possible. First, the hardware counter may have miscounted.

This likely will cause a large number of bits to be found in error during a checkpoint. Second, one or more bits could have been "flipped" while they were being shifted in the ring hardware. In either case, it is highly unlikely that an error can occur without being detected by a checkpoint, particularly since over 1675 bits are being tested. The chances of a hardware fault going undetected are even further decreased by regularly performing checkpoints.

4.6 SSUMon Variables

SSUMon requires a number of variables to maintain the state information of the MSSU. The variables are divided into two main categories. The first group consists of per-chip state information listed in Table 4.2. The second group of variables are common to all of the sieve chips and contain information pertaining to the system as a whole. These variables are listed in Table 4.3.

The status of each individual sieve chip is stored in a data structure called the Sieve State Information (SSI). SSUMon commands are provided to read and write many of these variables, while other commands access these values indirectly. Internally, SSUMon can quickly access this data by using the routine `Index()` which is passed a chip number from 0 to 31 and which returns a pointer to the structure.

Additional global information which must be maintained by SSUMon is listed in Table 4.3. These variables affect all of the sieve chips in an MSSU system. The variables `BOARD`, `CHIPSEQ`, `IRQLST0`, `IRQLST1`, and `PIPED`, are maintained by SSUMon itself and are not readily accessible by the user. For changing the values of the remaining variables, SSUMon commands are provided.

Variable	*Description*
COUNT	Count corresponding to a solution candidate.
COUNTM	Master counter value.
CNTR	Hardware counter value.
SOLMASK	Solution mask.
CNTROVF	Overflow counter for hardware counter.
OPTMUL	Optimization register multiply value.
OPTADD	Optimization register add value.
RINGDAT	Ring data for SSU in sequential order.
RING	Ring data for SSU in hardware order.
SOLCNT	Solution count.
SOLLMT	Solution limit (currently not used).

Table 4.2: Sieve State Information (SSI)

Variable	*Description*
BOARD	Used by Which().
CHIPSEQ	Sequence in pipeline.
ENABVEC	The enable vector.
IRQLST0	Used by Which().
IRQLST1	Used by Which().
OUTMODE	Output mode.
PBPARM	Problem parameter.
PIPED	Pipeline flag.
POSTFNM	Post-filter name.
PREFNM	Pre-filter name.
VRINGS	Virtual ring data.

Table 4.3: Sieve Global Variables

4.6.1 The Enable Vector

Since not all of the SSU chips in the MSSU system may be required when running a particular problem, some method is required to enable or disable them. This is the purpose of the *Enable Vector* (ENABVEC), a 32-bit array where a logic "0" value is used to identify an inactive SSU chip, and a "1" indicates that the corresponding chip is "enabled".

The value of ENABVEC affects many of the SSUMon commands. For example, when a checkpoint is performed, only the enabled chips take part in the checkpoint. Also, many commands check that a chip is enabled before executing and will return an error message

if it is not.

4.6.2 The Output Mode

The value of `OUTMODE` is used to select one of three possible formats for the output of solutions: `D`, `H` or `S`. The default mode, `D`, outputs solutions in decimal, convenient for human readers. Hexadecimal (`H`) is also provided as a faster, more compact mode of output, intended for computer hosts. The third option does not output the solution values at all. This mode is intended for solution counting mode, whereby a count of the number of solutions encountered is maintained in the Sieve State variable `SOLCNT`.

4.7 SSUMon Command Set

SSUMon recognizes and executes a number of commands input via the asynchronous serial port (see Table 4.4). All commands take a fixed number of parameters (0, 1 or 2) with no intervening whitespace on a command line. Values representing SSU registers or SSUMon variables are represented by hexadecimal character strings, with the exception of `OUTMODE` and the filter names which are specified as alphanumeric strings. Some commands also return a result in the form of a hexadecimal or an alphanumeric string.

Many commands require that a chip specification be provided as a parameter. To specify one of the 32 possible sieve chips, a two character hexadecimal string is used corresponding to a number from 0 to 31 (00–1F). Some commands allow a form of *wildcard specification*, interpreting the character * to signify that the operation be applied to all enabled chips, as per the enable vector. Two commands in particular, perform very useful functions when used with this wildcard specification feature. The `CK*` command performs a hardware consistency check for all enabled chips, and the `CN*` command returns the least `COUNT` value for all enabled chips. The host need only perform these two commands

Command	*Description*
!	Halt the sieve.
CK	Perform consistency check.
CN	Read COUNT value.
CR,CW	Read/Write COUNTM value.
EN	Set Enable Vector.
EX	Exit SSUMon.
FI	Specify filters.
GO	Enter Run state.
LO	Load sieve.
OM	Specify output mode.
OR,OW	Read/Write Optimization Register.
PP	Specify Problem Parameter.
RD	Output RING data.
RE	Reset sieve.
RR,RW	Read/Write RINGDAT data.
SR,SW	Read/Write Solution Registers.
ST	Read Enable Vector.
UL	Unload sieve.
VR,VW	Read/Write Virtual Ring data.

Table 4.4: SSUMon Commands

to perform a checkpoint operation for the entire system.

4.7.1 Host Communication Protocol

Since it is useful for debugging purposes to use SSUMon from a dumb terminal, the communication protocol that is used between SSUMon and the host system was kept very simple. All commands to SSUMon are one or two characters long, some followed by a parameter string. All command lines are terminated by a carriage return. SSUMon always responds with a carriage-return and line-feed character pair to start each line of output. The purpose of the carriage-return and line-feed characters is to make the output readable on a standard terminal. Output will consist of zero or more lines of status messages terminated with a colon (:) character. The colon character doubles as an SSUMon prompt character for human readers, as well as an end of message character for computer hosts.

Each status message is preceded by a one character code (see Table 4.5). `I` indicates an informational message which the host system may choose to ignore. Errors are always preceded by the character `E` and generally indicate that some further action is required by the host to respond to the error. Some commands request data from SSUMon, these responses are preceded by the letter `D`. The final message type is reserved for solution messages which are preceded by `S`.

Type	*Description*
`D`	Data value or status code (Halt state only).
`E`	Error condition occurred.
`I`	Informational or warning message.
`S`	Solution value (Run state only).

Table 4.5: SSUMon Message Types

The XON-XOFF software handshaking protocol is employed to ensure that data to and from the host system does not overrun serial I/O buffers. This enforces flow-control when either the control-processor or the host system are too busy to process the incoming data stream. Even at the rather modest baud rate of 9600 baud, it is not uncommon for the control processor to present data to the host faster than the host can process it.

It was decided not to employ a full-blown end-to-end communication protocol because of the overhead it would incur. Error-checking is provided at two levels. First, the number of solutions generated can be compared to the value of the `SOLCNT` variable to ensure that none were missed. Second, by the very nature of sieving problems, solutions are generally quite rare and must satisfy a number of congruence relations; thus, it is highly unlikely that a corrupted solution containing some digits in error will pass a verification test employing the initial congruence conditions.

4.7.2 The Reset Command

The purpose of the Reset command is twofold: to perform a hardware reset of the sieve hardware and to set all internal variables to their defaults. This command should be executed at the beginning of every sieve job to return the system to a known state and prevent previous settings from affecting later jobs. After a reset, all of the Sieve State Variables and the SSUMon global variables are set to zero, with the exception of `OUTMODE` which is set to `D`, and the prefilter and postfilter which are both set to `NULL`.

4.7.3 Sieve Loading and Unloading

Whenever the value of an SSI variable is changed by an SSUMon command, only that specific SSUMon variable will be updated. For example, for a `CW` (Counter Write), only the `COUNTM` variable will be updated; the value of the SSU hardware registers will be unaffected. To propagate the change to other affected software variables and hardware registers, the Load Command (`LO`) is provided. A *load* operation will, for all chips, call the `Cnt2Hw()` and `Ring2Hw()` routines to update the necessary variables and the hardware registers based on the settings of `COUNTM`, `RINGDAT`, and the optimization registers.

Similarly, before an operation such as a consistency check can be performed, it is necessary for the hardware register values to be read and the SSI to be updated correspondingly. This operation is termed sieve *unloading*, and is very much the reverse of the load operation. Undoing the work of its counterpart, the `UL` (unload) command invokes `Hw2Cnt()` to update variables based on the SSU register values. However, to prevent the overwriting of the initial residue information `RINGDAT`, no analogous `Hw2Ring()` routine exists. The value of `RINGDAT` can only be changed by an explicit `RW` (Ring Write) command.

4.7.4 The Run and Halt States

At any moment, the sieve is either in the *run* state or the *halt* state. In the run state, the MSSU hardware is sieving, SSU devices are in solution mode, candidate solutions are processed by SSUMon, and any solutions are output as solution messages. Responding to unexpected conditions, SSUMon may output informational messages or error messages.

The only command recognized by the command interpreter in the run state is the single character (`!`) halt command. When the halt command is received, hardware sieving is stopped, sieve chips enter command mode, and a sieve unload operation is performed to update the SSI. The MSSU is now in the halt state, and all SSUMon commands are recognized by the command interpreter.

In the halt state, the host processor can execute commands to load the hardware registers with a new sieve problem or perform operations such as a checkpoint on an existing problem. To start or resume sieving, the host will execute the `GO` command. The `GO` command causes a load operation to be performed, the sieve chips to enter solution mode, and the MSSU to enter run mode.

Because the `GO` and `!` commands perform load and unload operations indirectly, the host need not perform `LO` and `UL` commands explicitly. Nevertheless, the `LO` and `UL` commands are provided for debugging purposes.

4.8 Solution Processing

SSUMon solution processing is interrupt-driven. A maskable, level 6 hardware interrupt is generated by the sieve array board whenever a solution is pending for a sieve chip. The SSUMon interrupt handler first calls the `Which()` subroutine to determine which chip requires servicing. In the case of multiple outstanding interrupts, `Which()` applies a fairness criterion to select which outstanding interrupt request should be serviced next.

Then, the interrupt handler will invoke a routine to retrieve the hardware trial counter corresponding to the solution value and update the SSI.

A one-deep solution pipeline is maintained for each sieve chip to permit the processing of a solution to overlap with sieving for the next solution. Thus, once a solution has been stored in the SSUMon variables, the sieve chip is restarted to continue sieving.

4.8.1 QSPI Routines

The QSPI routines are used to read and write the SSU hardware counter and ring registers using the MC68332 synchronous serial interface. They are invoked whenever SSUMon commands are executed to read or write hardware registers, and in the case of the hardware counter, also invoked during solution processing.

A single routine is used to simultaneously read the current contents of an SSU register into an incoming data buffer and to update it with a new value stored in an outgoing data buffer (see Figure 4.4). In order to perform a non-destructive read, two operations must be performed. The same buffer must be used for the incoming data buffer for the first QSPI operation and as the outgoing data buffer for the second operation. This ensures that the data stored in the register initially is restored.

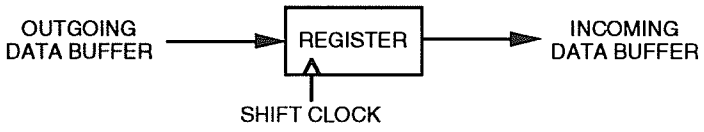


Figure 4.4: Simultaneous Read and Write of SSU Registers

The reason that this read operation is destructive is a function of the hardware. The SSU registers are implemented as serial shift registers where data is simultaneously clocked into the registers as data is clocked out (see Figure 4.4). Circuitry could have been used to recirculate the data for non-destructive reads; however, as we shall see, because

the SSU counter registers must be read and then written during solution processing, this additional circuitry is not needed.

The routine to read and write the SSU hardware counter and solution mask is named `ScanCtr()`. It simultaneously reads and writes the 52 low-order bits from 7-byte buffers. This includes 8 bits of the solution mask **SOLMASK**, 43 bits of the counter, and 1 bit for counter overflow held in **CNTR**. The counter data must be read every time a solution or counter-overflow is detected; but before the sieve chip can be restarted, the solution mask or counter overflow bit must be cleared. This is accomplished by reading the contents of the counter into a temporary area, clearing the necessary bits and writing this data to the sieve chip. This requires that `ScanCtr()` be executed twice for every interrupt.

Since `ScanCtr()` is invoked twice for every solution detected by the MSSU hardware, great care was taken in coding this routine. All loops were unrolled and data movement instructions were carefully selected for maximum speed of execution. Successive refinement has produced a version of `ScanCtr()` which executes in very few processor cycles.

The routine `ScanRng()` is used to read and write simultaneously the low order 1675 bits from 210 byte buffers. Since this operation is performed only during the loading of a problem or during a checkpoint, it is not necessary for `ScanRng()` to be particularly fast. The performance penalty resulting from the additional write operation after the destructive read is inconsequential.

4.8.2 The `Which()` Routine

The `Which()` routine is used to determine which of the SSU chips with pending interrupts should be serviced next. `Which()` maintains three variables to enforce a round-robin priority on pending interrupts which are intended to allow for a fair distribution of the control processor among the set of sieve chips. The variable **BOARD** takes on the values 0 or 1 corresponding to the array board which was last serviced. Two additional variables,

`IRQLST0` and `IRQLST1`, store the value of the chip number which was last serviced for respective array boards.

When `Which()` is invoked, the `BOARD` variable is incremented modulo 2. The status of the interrupt request outputs for the 16 sieve chips on the corresponding board are input. If there are no pending interrupts for this board, the `BOARD` variable is again incremented modulo 2, and the status of the interrupt request outputs for that board are retrieved. A search for the chip number of an interrupt request is performed modulo 16, starting from the position of the last interrupt serviced as specified by the `IRQLST` variable of that board. This enforces a circular ordering which prevents an SSU device from being serviced twice on consecutive interrupts unless there are no other interrupts pending.

4.8.3 Solution Pipelining

To implement the one-deep solution pipeline, solution processing was divided into two stages: the retrieval of a solution, and the processing of that solution.

The routine `GetSoln()` is used to retrieve the SSI for a chip which was determined to have generated an interrupt by `Which()`. `GetSoln()` then determines whether the interrupt was caused by a counter overflow or by the solution detector. If a counter overflow occurs, it is processed immediately and sieving is resumed. If a solution is detected, the status of the `PIPED` variable is tested to determine whether a previous solution is still pending in the pipeline. If `PIPED` is not set, then the SSI is updated from the hardware registers, and the sieve chip is released to resume sieving. If `PIPED` is set, an unprocessed solution is currently being stored in the SSI which must be processed first, before the new solution can be processed. This is called a *solution collision* because two solutions are outstanding for one sieve chip.

The routine `ProcSol()` is used to process a solution. `ProcSol()` can be called directly by `GetSoln()` in the event of a solution collision, but usually it will be invoked in the main solution loop of SSUMon. The solution loop repeatedly tests for the presence of

solutions by monitoring the status of the `PIPED` variable. Here, the variable `CHIPSEQ` is incremented modulo 32 to enforce a round-robin ordering on the processing of solutions. Thus a fair distribution of resources is enforced at two levels: at the time of a solution interrupt and at the time of retrieving solutions from the solution pipeline.

`ProcSol()` examines the SSI to determine whether multiple solutions are present (recall there are 8 solution taps). If multiple solutions exist, they are processed in numerical order. `ProcSol()` invokes `Hw2Cnt()` to determine the solution value of each solution. The two filter subroutines specified by `PREFNM` and `POSTFNM` are then invoked. If the solution passes both filters, it is output by the routine `OutSoln()` using the output mode specified by `OUTMODE`. `OutSoln()` also outputs the chip number of each solution so that solutions can be associated with the generating sieve chips.

Note that since MSSU is a parallel machine, the solutions output by SSUMon will not necessarily be sorted, because not all sieve chips will count up at exactly the same rate. It is the responsibility of post-processing software to ensure that solutions are in sorted order.

4.8.4 A Responsive Halt Mechanism

For proper functioning of the host software, it is very important that SSUMon provide a responsive halt mechanism to suspend sieving. The host software requires that SSUMon respond to a command within some reasonable *time-out* period. If the halt mechanism took several minutes to respond, users would have intolerable delays in terminating sieve jobs, and checkpoints would not be performed at the desired intervals.

Ideally, the SCI routines used to send and receive information from the host system should have been interrupt-driven, preferably at an interrupt level higher than solution interrupts. This would have enabled a halt command to have precedence over any solution processing. Unfortunately, the current implementation of SSUMon utilizes the CPU32BUG routines which are not interrupt-driven and thus requires polling using

`InStat()` to test for the presence of the halt character (`!`).

An early version of SSUMon implemented the halt mechanism poorly, causing it not to respond to a halt while interrupts were pending. When running problems having high candidate solution rates, such as the pseudosquares problem, starting from a low search bound, where the number of false solutions (perfect squares) is so great that filtering is the bottleneck in sieving, halt commands were ignored for ten minutes or more. Under these conditions, SSUMon would not execute the command to call `InStat()` to test for the presence of a halt command until control was relinquished by the interrupt service routine.

The solution to this problem was to ensure that `Instat()` be invoked at least once for every N solutions (currently, N=16) to test for the presence of a halt request. Although this method degrades performance of the system to a small degree, since the processor must now test for the presence of a halt far more frequently, it ensures that a halt request is acted upon almost immediately.

4.9 Filter Routines

Solution filters are selected by the user from a pre-defined set of filter routines using the specify filters command `FI`. Each filter is identified by a unique four letter filter name, a list of which is maintained by the system. If a filter name is specified which is not recognized, SSUMon will display an error message. The available filters are shown in Table 4.6.

The virtual ring filter `VIRT` is available to emulate 20 fixed size rings in software which are not available in hardware. This increases the number of rings which can be supported from the first 30 prime numbers provided by the SSU hardware to the first 50 prime numbers ($\leq$ 229). Since we expect the virtual filter to be specified for many sieving

Filter	*Description*
ERDO	Erdös-Selfridge function.
ICUB	Is a perfect cube.
ISQR	Is a perfect square.
NULL	All solutions accepted.
PBPM	Strong pseudoprime to base 2 (spsp(2)) test.
PBP2	Probably twice a prime (spsp(2)).
PCUB	Not a perfect cube.
PSQR	Not a perfect square.
QDPM	Probable quad (0-2-6-8) prime (spsp(2)).
TRPM	Probable triple (0-2-6) prime (spsp(2)).
TWPM	Probable twin-prime (spsp(2)).
VIRT	Virtual rings.

Table 4.6: SSUMon Filters

problems, it is necessary to provide the capability to specify a second filter for solution filtering in the traditional sense. To differentiate between the two, the filters are named the *prefilter* and the *postfilter* according to the order in which they are applied. For some problems the order in which filtering is performed can affect performance, thus the user should be conscious of the filter order.

If no filter is required the user can specify the NULL filter which passes all solution candidates. Although the same filter can be specified as both the prefilter and postfilter, the only time when such a combination may be useful is when both filters are NULL.

Since some filters consist of a function based on some parameter of the problem, SSUMon provides a 32-bit signed binary variable, named the *Problem Parameter* (PBPARM), which is settable by the user. For example, the ERDO filter which filters out invalid solutions to the Erdös-Selfridge function, requires that a value of the parameter k be assigned to the problem parameter variable PBPARM. Currently, SSUMon allows only one parameter to be specified for all sieve chips.

Each filter routine is passed a 128-bit solution candidate value. The only accepted return values are *pass*, when the solution satisfies the filter requirement, or *fail*, in which case the solution candidate is discarded. The result of filtering can be thought of as

the logical *and* of the prefilter and the postfilter. A solution is considered to be a valid solution only if it passes both filters.

4.9.1 The Virtual Filter – VIRT

Algorithm 4.2 *The Virtual Filter (4 Rings Shown)*

FUNCTION $VirtFil(Count, Ring1, Ring2, Ring3, Ring4)$ **RETURNS Flag** ;
INPUT $Count$: 128-bit integer (Solution candidate) ;
INPUT $Ring1, Ring2, Ring3, Ring4$: Virtual rings ;
OUTPUT : $PASS$ or $FAIL$;

Rem : 32-bit integer ;

$Rem \leftarrow Count \bmod (Ring1.size \times Ring2.size \times Ring3.size \times Ring4.size)$;
IF $Ring1.residues[Rem \bmod Ring1.size] \neq ACCEPTABLE$ **THEN**

    RETURN $(FAIL)$;

IF $Ring2.residues[Rem \bmod Ring2.size] \neq ACCEPTABLE$ **THEN**

    RETURN $(FAIL)$;

IF $Ring3.residues[Rem \bmod Ring3.size] \neq ACCEPTABLE$ **THEN**

    RETURN $(FAIL)$;

IF $Ring4.residues[Rem \bmod Ring4.size] \neq ACCEPTABLE$ **THEN**

    RETURN $(FAIL)$;

RETURN $(PASS)$;

End of Algorithm

The virtual rings filter provided by SSUMon emulates 20 rings corresponding to the primes 127 through 229. This is a total of 3524 residues which must be stored for each chip. Although it is almost always fastest to manipulate binary data represented as individual bytes as opposed to accessing individual bits, it would require $3524 \cdot 32 = 112768$ bytes of memory to store the residue information. Instead, SSUMon allocates an array of 3524

longwords where each bit of the longword corresponds to a residue for each of the 32 sieve chips. This incurs a small performance penalty, but requires only 14096 bytes of storage.

Much of the computation in the `VIRT` filter involves remainder operations to calculate the residue classes of the solution for each of the virtual moduli. Since calculating the remainder of a 128-bit number is expensive, `VIRT` groups the virtual rings/moduli into groups of 4 to reduce the number of 128-bit remainder operations which need to be performed. We describe this technique by using the pseudocode shown in Algorithm 4.2. Note that the temporary value *Rem* is calculated using a 128-bit remainder operation yielding a 32-bit result. This precomputation allows the following operations to be performed using only 32-bit remainder operations. This is a significant savings over the obvious approach using four 128-bit operations.

More information on various filters found in SSUMon can be found in Appendix B.

Chapter 5

MSSU Software: The Host Software

This chapter describes the software which executes on the Unix host system. In addition to the program which is used to run sieve problems on the MSSU, a number of ancillary programs are described.

5.1 Host Software Overview

Users interact with the MSSU via the *sieve command* which is part of the MSSU *Sieve Management Software* (SMS). The SMS maintains a queue of sieve problems and controls the loading, running and termination of *sieve jobs*. Users can add a sieve problem to the queue, terminate the current sieve problem, and display the status of the queue via the sieve command. The program code to implement the SMS is organized into four main code modules: the sieve command mainline, the sieve daemon, the problem file parser, and the sender module (see Figure 5.1).

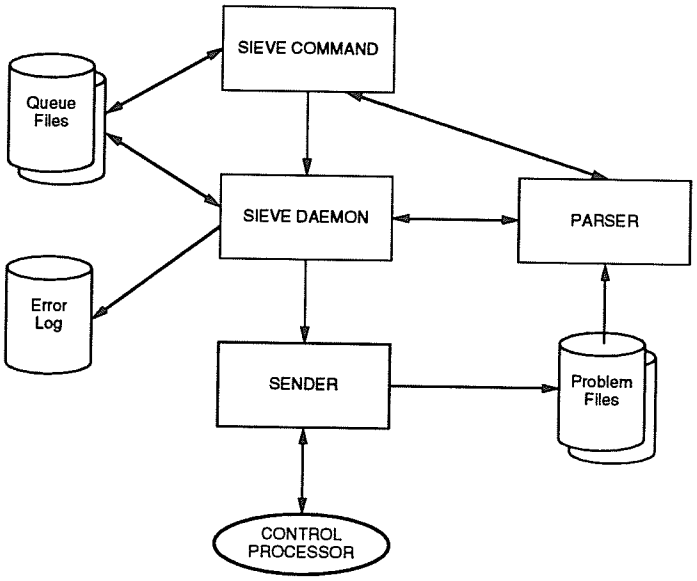


Figure 5.1: The Sieve Management Software

Sieving problems are specified to the system using standard Unix text files called problem files. Problem files contain the system of congruences to be solved plus additional parameters of the problem, such as the sieving range. Users can prepare these files manually using a Unix editor such as `vi` or write a program called a *problem generator* to generate the file.

Information about the sieve problem is extracted from the problem file by the problem file *parser* module. Whenever a user submits a problem file for execution, the parser must first verify that the problem file contains no syntactic errors and that a number of additional rules are followed. If no errors are detected, a data structure is returned containing the congruence information and the parameters specified in the problem file.

The sieve command mainline acts merely as an interface between the user and the underlying system. Much of the actual work of the host software is carried out by the sieve *daemon* process. The daemon process executes in the background and conducts a dialog with the SSUMon software executing on the control processor via the *sender* module.

Any errors encountered are automatically logged to an *error log* since the daemon has no controlling terminal to which it can output error messages.

The sender module communicates with the SSUMon software over a standard Unix *tty* port. It oversees the loading of the problem data onto the sieve hardware, the performing of checkpoints, and collects the solutions generated and places them at the end of the problem file. When a sieve problem completes, the sender will inform the sieve daemon to dequeue a new problem file from the job queue.

In addition to the sieve command, a number of ancillary programs are provided. These include *stat* which checks for syntax errors and displays statistics about problem files, *pull* which is used to extract a list of solutions from the problem file, and *talker* which is used to enter SSUMon commands to the control processor directly, bypassing the sieve daemon.

5.2 Format of Problem Files

The MSSU problem file format is based on the format used by the OASiS system. This permits OASiS problem files to be easily converted for use on the MSSU.

A number of problem file *keywords* are defined to denote the various sections of the problem file or identify problem file parameters. These keywords can be specified in either upper or lower case. The space, tab, and newline characters are treated as whitespace characters by the parser. Comments are line-oriented, with text following a comment character ! (exclamation point) treated as a comment up to a matching comment character, or until the end of line.

A problem file is divided into 4 sections. The first is the PROBLEM section, which specifies a name for the sieve problem; the second is the CONGRUENCES section which describes the congruences to be used for sieving; and the third section is called PARAMETERS which specifies values to be used for parameters such as the starting trial counter

value, the interval between checkpoints, and the names of the solution filters, to name a few. The final section is reserved for output generated by the sieve software and is called the LOG section. Here, solutions and system-generated messages are logged with *timestamps* giving the elapsed time since the sieve job was submitted.

5.2.1 PROBLEM Section

Like the OASiS problem file format, the sieve problem name is limited to 32 characters. Although the user can choose to use any name he or she desires, the intention of the problem name is that it be a meaningful mnemonic for describing the sieving problem.

`PROBLEM` *problem_name*

5.2.2 CONGRUENCES Section

The CONGRUENCES section specifies the congruences of the sieving problem. This section immediately follows the PROBLEM section. Congruences are specified using the form:

$$keyword\ (\ \ modulus_i\ :\ \ \ residue_1\ residue_2\ \ldots\ residue_{n_i}\ ;\ \ )$$

The modulus can assume any value in the range 2 to $2^{32} - 1$. While, residues can be any value from 0 to $modulus - 1$. At least one residue must be specified for each congruence. A range of residues such as 1, 2, 3...6 can be specified with the shorthand form of "1:6". Residues may appear in any order, however, repeated residue values are not permitted. As well, no two congruences can have the same modulus value.

Three different types of congruences can be specified using the keywords `HARDWARE`, `VIRTUAL`, and `PARTITION`. The hardware congruences are sieved using rings on the underlying sieve hardware. On the other hand, virtual congruences are congruences which are not supported by the hardware, but can be emulated in software on the control processor. Congruences specified by the keyword `PARTITION` are used to parallelize the execution of

the sieve problem in such a way that these congruences need not be tested in hardware nor software.

Hardware congruences have moduli corresponding to the first 30 primes, 2 through 113. The primes 11 through 113 are directly supported by the underlying SSU architecture, as are the prime powers 16, 9, 25 and 49. The host software will also accept the modulus values 2, 4, 8, 3, 5, and 7 and map those to prime powers supported by the hardware.

Because the hardware congruences are likely the most common congruence type to be specified, the keyword `HARDWARE` and the enclosing brackets are optional when specifying hardware congruences. To ensure that there is the possibility of solutions being generated by the hardware, the system requires at least one hardware partition to be specified. Otherwise, an error message will result.

The `VIRTUAL` keyword is used to specify congruences whose moduli are a prime value between 127 and 229 inclusive. These congruences will be tested using virtual rings emulated in software if the user specifies the `VIRT` filter in the parameter section of the problem file. If the `VIRT` filter is not specified, none of the virtual congruences will be sieved, and the virtual congruences are effectively treated as comments.

Congruences denoted by `PARTITION` are used to specify parallel execution of a sieve job and are explained in detail in the following section. Unlike the hardware and virtual congruences, a partition congruence can assume any value between 2 and $2^{32} - 1$ such that the product of the moduli of the partition congruences is less than or equal to $2^{32} - 1$.

5.2.3 Specifying Parallel Execution of Sieve Jobs

To take full advantage of the parallel architecture of the MSSU system, two new methods of specifying congruences were developed. These are called partitioning and multi-variable sieving.

Partitioning (Single Variable Sieving)

Partitioning allows the user to partition, or divide, a single sieving problem into a number of sub-problems, each of which executes on a separate sieve chip. Partitioning is specified in a problem file by use of the keyword `CONGRUENCES` followed by the specification of the satisfying congruences. Partitioning requires that the user specify a single *partition congruence*, whose number of residues is less than or equal to the number of available sieve chips. To identify the congruences which make up the partition congruence, the keyword `PARTITION` is specified for one or more congruences (see Figure 5.2). When multiple congruences are specified, the host software combines these to form an equivalent single partition congruence.

```
PROBLEM PS30

CONGRUENCES
!   0 !   PARTITION( 8:      1; )
!   1 !   PARTITION( 3:      1; )
!   2 !   PARTITION( 5:      1    4; )
!   3 !   PARTITION( 7:      1    2    4; )
!   4 !   PARTITION( 11:      1    3    4    5    9; )
!   5 !   13:      1    3    4    9   10   12;
!   6 !   17:      1    2    4    8    9   13   15   16;
!   7 !   19:      1    4    5    6    7    9   11   16
                  17;
!   8 !   23:      1    2    3    4    6    8    9   12
                  13   16   18;
:
!  45 !  VIRTUAL( 199:     1    2    4    5    7    8    9   10 ...
:
PARAMETERS
        RECORD FROM 0
        TO 10000000000000000000
        PREFILTER = 'VIRT'
        POSTFILTER = 'PSQR'
        CHECKPOINT INTERVAL = 60 MINUTES

LOG
```

Figure 5.2: Partitioning Example

In Figure 5.2, we show a sample file corresponding to the Psuedosquares problem. The smallest 5 moduli are combined into a partition congruence consisting of 30 residues

modulo 9240. In addition to specifying the remaining 25 hardware rings, a number of virtual congruences were specified (only the modulus 199 is shown in Figure 5.2). The resulting sieve problem will utilize 30 sieve chips operating in parallel. The parameters section specifies: start and stop values, a perfect-square filter, a virtual ring filter, and that checkpoints be performed at hourly intervals.

Multi-Variable Sieving

The second method provided by the MSSU host software for executing parallel sieve jobs is called multi-variable sieving. In this case, a separate set of congruences is specified for each chip used (see Figure 5.3). This allows each sieve chip to execute a completely different sieving problem, however sieving parameters such as the solution filter are common to all chips.

```
PROBLEM sotc3p2

CHIP=0
! d=3, m=2 !
2:  0 1 ;
3:  2 ;
5:  0 2 4 ;
7:  3 5 6 ;
11:  1 3 4 6 8 10 ;
13:  1 3 4 7 8 9 10 11 12 ;
17:  3 4 5 8 11 12 13 15 16 ;
19:  0 2 3 8 12 14 18 ;
⋮
CHIP=31
! d=3, m=137 !
⋮
PARAMETERS
        RECORD FROM 0
        TO 1000000000
        SOLUTION LIMIT = 20
        FILTER = 'VIRT'
        CHECKPOINT INTERVAL = 10 MINUTES

LOG
```

Figure 5.3: Multi-Variable Sieving Example

The keyword CHIP is used to signify that the following congruences are to be associated

with the given chip. Chips are specified with a decimal number from 0 to 31. To allow solutions to be matched with the generating congruences, each solution message contained in the log section of the problem file gives the chip number of the generating sieve chip.

In Figure 5.3, we show a problem file corresponding to the sums of three cubes problem (see Section 7.5.1). Because this is a two-variable problem, we sieve in both the x-dimension, given by the trial counter of each sieve chip, and an m-dimension, corresponding to each individual sieve problem. Here, a comment following the specification of the congruences of each chip lists the corresponding value of m to be associated with any solutions generated.

5.2.4 PARAMETERS Section

There are a large number of parameters which can be specified for a sieve problem. Most parameters are optional; however, the `RECORD FROM` value of the search description parameter is required. A list of the parameters, listed in the order in which they must be specified, is given in Table 5.1.

Parameter	*Values*	*Description*
`RECORD FROM`	*large_int*	Start of search interval (required).
`TO`	*large_int*	End of search interval.
`SOLUTION LIMIT`	*small_int*	Max. number of solutions before termination.
`FILTER`	*filt_nam*	Name of filter (one filter only).
`PREFILTER`	*filt_nam*	Name of first filter (two filters).
`POSTFILTER`	*filt_nam*	Name of second filter (two filters).
`CHECKPOINT INTERVAL`	*time_val*	Time interval between checkpoints.
`PROBLEM PARAMETER`	*signed_int*	Problem-specific parameter value.
`DELETE`	`YES` | `NO`	File deletion option.

Table 5.1: Sieving Parameters

We now describe the function of each of these parameters, and define the values which can be specified for each.

Search Description Parameter (RECORD FROM, TO)

The *search description* consists of a required start value and an optional end value for the search interval. Trial counter values are specified as large (multiprecise) unsigned integers in the range 0 to $2^{128} - 1$. The value can be specified in either a longhand format, or using a restricted form of scientific notation, of the form:

$$mantissa\,\mathrm{E}\,exponent$$

where the resultant value is calculated as

$$mantissa \times 10^{exponent}$$

The mantissa and exponent must both be integer values. For example, one possible search description is:

```
RECORD FROM = 2000
TO = 1E19
```

Solution Limit Parameter

The user has the option of enforcing a limit on the number of solutions before a sieve job is terminated. The parameter value must be in the range 1 to $2^{32} - 1$. This is accomplished by including a line of the form:

SOLUTION LIMIT = *small_int*

Filter Specification Parameter

Optionally, each sieve problem can specify one or two solution filters. If only one filter is to be specified, the keyword FILTER is used. By default, the unspecified second filter will be set to NULL. If two filters are to be used, a PREFILTER assignment followed by a POSTFILTER assignment must be specified.

Filter names are exactly four characters long, can be in either upper or lower case, and must be enclosed in single quotes. The problem file parser does not check whether a

filter exists having the name specified. The assignment of filters according to these four character strings is performed by the control processor software SSUMon which maintains an internal table of valid filter names. Thus, an invalid but syntactically correct filter name will not result in an error message until loading of the problem onto the sieve is attempted. This error message will appear in the log section of the problem file.

Checkpoint Interval Parameter

The checkpoint interval determines how frequently a checkpoint operation is performed. The checkpoint operation performs a verification of the sieve contents utilizing SSUMon consistency checks. Checkpoints are also used to record in the problem log the trial counter values to which computation can be safely *rolled back* in the event of an error. Checkpoints are also important because they determine the times at which the sieve daemon tests if the end value of the search interval has been reached. This check is performed only during checkpoints.

The time interval can be specified using the time units `SECONDS`, `MINUTES` or `HOURS` (`HOUR` is also accepted). By default, if no units are given, values are assumed to be in seconds. The system enforces limits on the checkpoint interval to within reasonable values. Currently, it cannot be less than 3 minutes, nor greater than 5 hours.

The Problem Parameter

The problem parameter is a 32-bit signed integer quantity passed to the filter routines. This allows filters to be written which are based on some function of the sieve problem. For example, the sieving problem to compute the Erdös-Selfridge function for different values of the parameter k requires that the k value be passed to the `ERDO` filter. This can be accomplished by specifying `PROBLEM PARAMETER =`*kvalue*. Negative values are specified with a minus sign directly preceding the first digit of the value.

Delete Parameter

The delete parameter specifies whether the problem file should be deleted after it has executed subject to special conditions. If `DELETE=YES` is specified, the file will be deleted if the end of the search range has been reached with no solutions being found and with no errors being generated. This feature is particularly useful for multi-variable sieving where it may be desirable to discard the problem file once it has been shown to contain no solutions. This simplifies the maintenance of problem files by the deletion of the uninteresting ones.

5.2.5 LOG Section

Output from the execution of the sieve problem is placed in the problem file following the keyword `LOG`. The different message types which can be produced are listed in Table 5.2.

Type	*Description*
`DATE`	Informational date message.
`DEBUG`	Informational debug message (reserved).
`ERROR`	Non-fatal error message.
`FATAL`	Fatal error message.
`MESSAGE`	Informational message.
`SOLUTION`	Solution message.
`VERIFIED`	Successful checkpoint message.

Table 5.2: Log Message Types

All messages generated by the SMS are preceded by a timestamp giving the elapsed time since the sieve job was first read by the system. When logging some important events, a `DATE` message is output with the time of day, day of the week, name of the month, and the year. The date message is output whenever a problem file is read, a checkpoint is performed, or whenever a problem is terminated, enabling the reader of the problem file to associate an absolute date to these events.

In the sample log output (see Figure 5.4), taken from a run of the Sums of Three

Cubes problem (see Section 7.5.1), a single solution value of 5 is found by chip number 2. The time to load the problem was 1 minute and 40 seconds, and the solution was found almost immediately thereafter. For this example, the checkpoint interval was set to 10 minutes, and the end of the search interval was 1 billion. When the checkpoint was performed it was determined that the end value had been exceeded, and the problem was terminated.

```
LOG
     0 00:00:00 MESSAGE Problem file read successfully
     0 00:00:00 DATE Mon Nov 21 09:50:12 1994
     0 00:01:40 MESSAGE "sotc3p2" utilizes 32 sieve chips
     0 00:01:42 SOLUTION        1 AT 5-02
     0 00:11:42 MESSAGE Verifying sieve contents
     0 00:11:42 DATE Mon Nov 21 10:01:54 1994
     0 00:11:44 VERIFIED        1 SOLUTIONS TO 115132716680
     0 00:11:44 MESSAGE End value reached
     0 00:11:44 MESSAGE Problem terminated
     0 00:11:44 DATE Mon Nov 21 10:01:56 1994
```

Figure 5.4: Sample Log Output

Solution Order and Duplicate Solutions

Recall that the sieve chips are running at slightly different rates; consequently, solutions will not necessarily be in sorted order. It is also possible for multiple solution messages to appear in the log for the same solution. This situation can occur when a sieve job is terminated by a user request and later restarted, or if a hardware fault causes a restart from a previous checkpoint, resulting in a small amount of overlap in computation. If a solution was generated at a trial counter value higher than the last successful checkpoint, a restart will cause that solution to be recomputed.

5.3 The Programs

Users access the sieve programs and problem files using the "sieve" Unix login account on a SPARCstation workstation connected to an MSSU. The executables which make up the MSSU host software are located in the `bin` directory of this account. In addition to the `sieve` command, which is the user interface to the SMS, a number of ancillary programs are provided for users of the MSSU. These include the programs `talker`, `stat`, and `pull`.

5.3.1 The Configuration File

The sieve configuration file `/var/spool/sieve/sieve.con` contains the number of chips available on the sieve and the name of the *tty* port that is connected to the MSSU host port. By default, if the configuration file is not present, the SMS will assume that 32 chips are present, and that the sieve is attached to `/dev/ttya`.

Creating or modifying the configuration file is useful when moving the host software from one port to another, or when running the MSSU without the full complement of SSU devices. If a problem file is submitted to the SMS for execution which requires more SSU devices than the value of the `CHIPS` parameter, that problem file will be rejected and an error message given.

The format of the configuration file is:

```
! comments look like this
! or, this !
TTY = /dev/ttya     !this is the tty port used!
CHIPS = 32          !number of installed chips!
```

Filename Extensions

Although filename extensions are not mandatory under Unix, a filename extension of "`.siv`" is assumed for all problem files. This scheme was used by the OASiS system which

ran under the VMS operating system where all filenames must be assigned extensions. The `.siv` extensions are retained in the MSSU system because they aid in organizing collections of files by differentiating between problem files and other types of files.

5.3.2 The `Sieve` Command

The syntax of the sieve command is:

`sieve` *commandname* [*argument*]

This program accepts a number of commands to perform various functions via the SMS. A complete list of the commands is given in Table 5.3. Most commands have a single letter abbreviation, however, to prevent disastrous effects from simple typographical errors, the `kill`, `purge` and `restart` commands must be typed in full.

Command	*Description*
`e` | `errorlog`	Display entire contents of error log.
`kill`	Terminate the currently executing sieve job.
`purge`	Purge the queue of all pending jobs.
`q` | `queue` *filename*	Enqueue a given problem file.
`restart`	Restart daemon after system crash/shutdown.
`s` | `status`	Display status of sieve queue.

Table 5.3: Sieve Command Options

The `errorlog` command will display the entire contents of the error log file. Since the error log file can be very long, the user may choose to view only the most recent entries by piping the output through the Unix `tail` filter, with:

```
sieve e | tail
```

The SMS identifies the current problem being sieved as *executing*, and all other queued problem files as *pending*. The `kill` command will terminate the currently executing job. The sieve job will perform a final checkpoint before terminating. If a problem file is pending, it will be dequeued and placed in the executing state. The `purge` command can be used to delete all pending jobs from the queue, leaving the executing job unchanged.

If a sieve job is currently executing, the **queue** command will add the given problem file to the pending list. Otherwise, it will make the given problem file the executing job. If the format of the problem file is found to be invalid, an error message describing the error is output, and the file is not queued.

The `restart` command will attempt to restart the SMS following a reboot of the Unix host, or a crash of the SMS. It will try to create a new sieve daemon process to re-initiate the problem file which was executing at the time the system went down.

The `status` command displays the status of the sieve queue, listing the currently executing problem file, and the pending list of problem files. If the sieve queue is empty, the following message is displayed:

```
status: no daemon is currently running
no currently executing job
no pending jobs
```

5.3.3 talker

The `talker` program allows a user to bypass the SMS and enter commands directly to SSUMon from a Unix terminal session. This is a useful tool for debugging the sieve without having to physically connect a dumb terminal to the MSSU host port.

`talker` Uses the "raw" mode of character input, as opposed to the "cooked" mode. This allows each character typed by the user to be sent to the control processor immediately, without requiring the input line to be terminated with a newline character. The key sequence *control-C* is used to exit the program.

5.3.4 stat

The `stat` program outputs statistics for a given problem file. The syntax is:

`stat` *filename*[*.siv*]

As part of its output, `stat` displays the maximum canvass rate and an estimate of the filter rate. The maximum canvass rate is a measure of how quickly the sieving range is being searched assuming the system is running at 100% of full-speed, and there are no slowing effects due to filtering or other control processor overhead. The filter rate is a measure of the density of solutions and is given in terms of solutions per second. The filter rate is computed as:

$$filter\ rate = canvass\ rate \times \prod_{i=1}^{50} \frac{(\#\ residues\ modulo\ p_i)}{p_i}$$

The same parser routines are used for both `stat` and the sieve command. If the specified problem file contains a syntax error or violates any of the other rules for MSSU problem files, a message describing the error is output. In this manner, a user can test the validity of problem files before submitting them with the sieve command.

5.3.5 `pull`

The `pull` program extracts solutions from a problem file and outputs them one per line to a solution output file:

`pull` *probfile*[*.siv*] [*solutionfile*[*.out*]]

Although for some problems it may suffice to extract solutions manually using a text editor, or even pencil and paper, such methods are tedious and error-prone. The `pull` program is a convenient means for extracting all of the verified solutions from a problem file. Because `pull` knows all about checkpoint messages, it can determine which solutions are verified, and which are not, and place only verified solutions in the output file. `pull` also displays useful summary information on the standard output about the solutions it has extracted, including whether the solution file is in sorted order and whether any inconsistencies were found in the problem file.

Because of its usefulness, the "`.out`" extension is reserved for solution output files. If no extension is specified for the solution file, `.out` is assumed. If no name is specified

for the solution output file, the name is taken from the problem file name, and the `.out` extension is substituted.

5.3.6 Off-Line Solution Filters

Off-line solution filters are often used in conjunction with solution output files generated by `pull` (i.e. one solution per line). These programs usually perform some sort of classification of the solutions, generating a line of output containing the value of each solution and the associated filter value.

A number of off-line solution filters are provided by the MSSU system. For example, the off-line filter `psall` is used to determine the highest prime for which a solution satisfies the congruences for the pseudosquare problem. Programmers will find that it is not difficult to modify the source code of `psall` to create new filters for different problems.

To illustrate the operation of `psall` we present some sample output. The Unix command "`psall <ps30.out`", was used to generate the following output:

```
6450045516630769--199
11641399247947921--223
30819994040589121--211
43782503675213521--199
61414601754207601--199
⋮
```

The program `best` can then be used to post-process this output to yield the "best" solutions to the problem. (A `worst` program is also available to search for least values.) The `best` program filters the standard input and outputs those lines containing the largest number seen so far following the two dash characters (`--`). Using the data from the above example, `best` generates the following output:

```
6450045516630769--199
11641399247947921--223
⋮
```

5.3.7 Problem Generators

As discussed earlier, problem files can be generated using a text editor, but the preferred method is to write a program to generate them called a *problem generator.* Often, the set of congruences for a sieving problem are non-trivial to calculate, and it is easier to write a problem generator than it is to perform the calculations by hand. Problem generators can also be written to create problem files based on various parameters and thus can be used over and over again for numerous problems.

Currently, no program has been written to perform the job of automatically selecting partition congruences for problem files specifying partitioning. It is feasible that a program could be written to evaluate all possible selections for the partitioning set, and select the optimal set resulting in the highest sieving rate. The number of possible combinations is actually quite small, and usually selection of the optimal set will appear obvious to a human user. Unfortunately, until such a program is implemented, partition congruences must be assigned manually with the use of a text editor.

5.4 Implementation of the Sieve Management Software

The SMS was written for use on SPARCstation workstations running the SunOS 4.1.x operating system. This platform was chosen because of its widespread use and the stability of the SunOS operating system. Also, these workstations provide two built-in asynchronous ports, identified as `ttya` and `ttyb`, both of which are suitable for connecting to the MSSU host port. The SMS was written entirely in the C programming language, except for the parser routines which were generated using automated tools. Not including the parser routines, the code is approximately 10,000 lines of C.

The SMS is based on a client-server model. The sieve command is a client process that enqueues problem files which are processed by the server process, the sieve daemon. The daemon process is spawned by the sieve command whenever a problem is queued

and there is no sieve daemon running. The daemon persists as long as there are problems in the queue.

Communication between the client and server processes is performed via signals and shared files. Signals are a means for one process to notify another process that an event has occurred, while shared files are used to transfer information between processes.

5.4.1 Multiple Precision Arithmetic

Support for arbitrary precision integer arithmetic is provided by GNU MP, the GNU Multiple Precision Arithmetic Library, Edition 1.2.93. This library is free and implements quite a comprehensive collection of integer arithmetic operations, as well as rational arithmetic operations. Names of GNU MP integer arithmetic routines are identified by the prefix "`mpz_`". The complete source code, written in the C language, is also distributed with the package.

5.4.2 Shared Files

A number of shared files located in the `/var/spool/sieve` directory are used to maintain the sieve queue.

`.ERROR` contains a log of the name of each problem file, the date it was executed on the sieve and any errors which may have occurred while enqueuing or dequeuing sieve jobs. Because the daemon has no controlling terminal, this file is used to output error messages.

`.EXECUTING` stores the name of the problem file which is currently executing on the sieve. This file only exists while a problem file is executing on the sieve.

`.PENDING` holds the pending list of problem files to be executed after the current sieve job completes. Problem files are listed one per line and jobs are dequeued beginning at the first line of the file. If no pending files exist, this file is not present.

`.PID` contains the process identification number (`pid`) of the current sieve daemon process. When the sieve daemon process is spawned, it places its `pid` in this file. When it completes, the sieve daemon deletes `.PID`.

File Locking

File locking is used to enforce exclusive access when updating the shared files. The SMS utilizes the Unix `flock()` system call to provide *advisory* file locks. With advisory locks, all processes cooperate and make use of the same locking mechanism to guarantee exclusive access. However, processes outside of the cooperating group can still gain access to the file by choosing not to use the advisory lock. Thus, the problem files are not locked to external programs such as the Unix `vi` editor.

Both the sieve command and the sieve daemon invoke `flock()` requesting an *exclusive lock* before accessing shared files. Exclusive locks cause the caller to block until the lock is acquired, enforcing mutual exclusion. After the update has completed, the `flock()` routine is invoked once more to perform an *unlock* operation.

5.4.3 Use of Signals

The SMS uses a style of signal facilities found in 4.3BSD Unix. This signal mechanism is said to be reliable, in the sense that signals cannot get lost by the system.

Signals are defined by the operating system for many abnormal events which may occur during the execution of a process. Signals can be ignored or assigned user-written handler routines using the `signal()` system call. One notable exception to this is `SIGKILL`, used to terminate a process on system shutdown, which cannot be caught or ignored. To allow processes to exit gracefully prior to system shutdown, the operating systems sends a `SIGTERM` signal to each process. If the `SIGTERM` is not acted upon, the operating system will send a `SIGKILL` which leads to termination of the process.

User processes can also generate certain signals and deliver them to specific processes based on their process identification, or `pid`. The `kill()` system call delivers a specified signal to a process with a given `pid`. This method is used to deliver user-defined signals such as `SIGUSR1`. Another means to deliver a signal is via the `alarm()` system call which generates `SIGALRM` signals.

Table 5.4 lists the signals which are used by the SMS.

Name	*Description*
`SIGALRM`	used to schedule checkpoints.
`SIGHUP`	used to signal a problem termination request.
`SIGKILL`	special (cannot be caught, blocked or ignored).
`SIGTERM`	this signal causes a graceful exit of the daemon.
`SIGUSR1`	used to check if the daemon process is alive (ignored).

Table 5.4: SMS Signals

The SMS contains a number of signal handler routines for the abovementioned signals, with the exception of `SIGKILL`, for reasons explained earlier, and `SIGUSR1`. The `SIGUSR1` signal is used to determine whether the sieve daemon is running. To implement this, the sieve daemon ignores the `SIGUSR1` signal by setting the handler to the built-in function `SIG_IGN`. The sieve command client process can determine whether or not the daemon process is running by attempting to deliver a `SIGUSR1` signal to the process specified by `.PID` with a `kill()`. If the `kill()` fails, the daemon has passed away under mysterious circumstances and the `.PID` file is out of date. Otherwise, it can be assumed that the `.PID` file holds the `pid` of the sieve daemon.

Often, during critical processing, it is desirable to block certain signals. To specify which signals are to be blocked or unblocked, one specifies a *signal mask*. The system call `sigsetmask()` is used to retrieve the current signal mask. Specific bits of the signal mask, corresponding to signals which need to be blocked or unblocked, can then be set or cleared appropriately. The updated signal mask is then set using the system call `sigblock()`.

5.4.4 Problem File Parser

The problem file parser was written using the Lex lexical analysis program generator and the Yacc parsing program generator. These automated tools greatly simplified the implementation of the parser. Unfortunately, when parsing very large problem files with multi-variable sieving data, the parser was found to be quite slow. A 300 kilobyte problem file would require several minutes to parse. In an attempt to increase the performance of the parser, GNU Flex, the fast lexical analyzer generator which employs improved scanner algorithms, was substituted for Lex, greatly improving performance. The only ramification of this change was the loss of the Lex built-in line number variable `yylineno`. Unfortunately, this eliminated the ability to output line numbers when displaying syntax errors.

The routine `parse_problem_file()` is invoked twice during the processing of each problem file: initially by the sieve command mainline to confirm that the problem file is valid, and once more by the sieve daemon process to extract the problem information.

Because a daemon persists as long as the job queue is not empty, the parser can be called repeatedly by the same incarnation of the daemon process. Therefore, it is crucial that any memory allocated by the parser be freed after use, preventing daemon memory from growing without bound. This requires that all tokens allocated by the lexical analyzer be deallocated by the parser routines after being processed.

The `parse_problem_file()` routine reads the contents of the problem file and returns the parsed information in the `problem_info` data structure. `problem_info` contains all of the congruence information, parameter values and flags denoting whether partitioning or multi-variable sieving is to be used.

Processing of Congruence Information

The parser calls the `process_congruences()` routine to verify that the information just read is consistent with the rules for specifying congruences, and calculates the hardware and virtual ring data to be loaded using SSUMon. If an error is detected, an appropriate error message is stored in the `problem_info` structure and later output, either to the standard output if invoked by the sieve command, or to the error log file if invoked by the sieve daemon.

Hardware congruences in the problem file must be mapped to equivalent hardware rings on the SSU chip. Congruences having a modulus value of 2, 4, 8, 3, 5, or 7 are exceptional cases and must first be converted to congruences modulo 16, 9, 25 or 49, as required by the underlying hardware.

For problem files where partitioning is indicated, the `combine_congruences()` routine is used to evaluate the partition congruence. `combine_congruences()` uses a straightforward algorithm which combines two congruences at a time, creating a congruence whose modulus is equal to the product of the moduli. The first congruence is used to generate residues up to the resultant modulus, and then the second congruence is applied to eliminate residues which do not satisfy both congruences. This process is continued until a single congruence remains, having a modulus equal to the product of the specified moduli.

5.4.5 The Daemon Process

The sieve daemon process was based on the skeleton daemon described by W. Richard Stevens, in his book "UNIX Network Programming" [Ste90]. According to Stevens:

> *A daemon process is a process that executes "in the background", (i.e. without an associated terminal or login shell) either waiting for some event to occur, or waiting to perform some specific task on a periodic basis.*

The sieve daemon is created by a `fork()` system call from within the sieve command mainline. A new sieve daemon is created whenever a new problem file is enqueued and no existing daemon process is executing. The daemon process disassociates itself from the originating process group (i.e. the parent process) and the controlling terminal so that it will execute even after the user which initiated the daemon has logged out.

The daemon process *working directory* is set to `/var/spool/sieve`. All of the shared files are located here, as is the configuration file. Should the daemon process crash, due to a system-detected error, a core file will be placed in this directory.

5.4.6 The Sender

Communication between the sieve and the SMS is controlled by code in the sender module. The `sender()` routine is invoked once for each problem file and executes for the duration of that problem.

The asynchronous communication routines of the sender module make use of the Unix *termio* general terminal interface. Calls to the `ioctl()` system call are made to initialize the communication parameters and test the status of the input buffers. The `read()` and `write()` system calls are used for the actual input and output. For proper operation of the SMS, the "ownership" of the *tty* device file must be set to the sieve userid, and the `/etc/ttytab` entry for that port must be set to disable interactive logins.

Input is performed using timed reads with a time-out value of 30 seconds. Timed reads have the advantage of causing the program to block until the read is completed, without the read blocking indefinitely. Should the communication link between the host system and the MSSU be disrupted, the SMS will eventually time-out and try to re-establish communication.

Problem Loading

Once the sender module has established communication with the sieve and confirmed that SSUMon is running on the control processor, the sender code will load the problem onto the sieve hardware. First, an SSUMon reset command is sent, followed by commands to set the problem parameter and specify the filters. Next, the hardware ring data and virtual ring data are written to each sieve chip. The enable vector is then set, and an informational message is written to the log giving the problem name and a count of the number of chips enabled. At this point, the counter is set to either the start value specified or to the last successful checkpoint value. Finally, an SSUMon command to load the sieve and initiate sieving is sent.

Since no error-checking is performed on the communication link between the host and the control processor, the SMS requires that this link be 100% error-free. During loading, the sender will perform a read-after-write of all data to verify that there were no errors in transmission. If all 32 chips are enabled this amounts to writing and reading approximately 42 kilobytes of data, requiring approximately one minute of data transmission. Any errors detected during problem loading would indicate that the operation of the sieve would be, to say the least, unreliable. Fortunately, after several years of sieving on two MSSU systems, no serial communication errors have been detected.

5.4.7 Solution Processing

After loading the problem and initiating sieving, the `sender()` routine enters a loop to await solutions. Solutions generated by the control processor are assigned a solution number, and a solution message is output to the problem file log. Processing continues within this loop until one of the following conditions occurs: a checkpoint occurs, the solution limit is exceeded, a kill command is received, or repeated SSUMon errors result in a fatal error.

To prevent a stream of solutions from overrunning I/O buffers, the XON-XOFF protocol is used between the control processor and the `sender()` routine. To ensure that no solutions are missed, the number of solutions received is compared to that expected by the control processor at the time of a checkpoint. If a discrepancy is found, the checkpoint fails and a roll-back is performed.

Checkpoints

Checkpoints are scheduled using the `alarm()` system call which delivers a `SIGALRM` signal after a specified number of seconds. When a job is first loaded, the alarm is set according to the checkpoint interval parameter. When the alarm goes off, the `SIGALRM` signal is handled by the `handle_sigalrm()` routine. `handle_sigalrm()` blocks all maskable signals, schedules the next checkpoint using `alarm()`, and performs the checkpoint operation. If an end value was specified, the trial counter value is tested during the checkpoint to see if it has been reached. Should the end value be exceeded, the problem is terminated.

If a checkpoint fails, the problem data are re-loaded onto the control processor, and the trial counters are set to the value of the last successful checkpoint. If there was no previous successful checkpoint, the search start parameter is used.

Whenever a problem is terminated, a final checkpoint is performed to ensure that all of the computation up to the point of termination has been verified.

5.4.8 Porting to Other Hosts

Programmers will find that the SMS utilizes many features from the 4.3 Berkeley Software Distribution (4.3BSD) Unix system. With minimal change this software should run on any modern Unix system with BSD support. An earlier version of the software ran successfully under Ultrix V4.2 on a MIPS based DEC workstation and required minimal changes to execute under SunOS 4.1.x.

Chapter 6

Results on Problems Involving Quadratic Characters

In 1970, Lehmer, Lehmer and Shanks[LLS70] examined the problem of determining integer sequences N_p such that the Legendre symbol $(N_p/p_i) = \varepsilon_i$, for all primes $p_i \leq p$, for a prescribed set of values of ε_i with $|\varepsilon_i| = 1$. The six problems which they identified as being of special interest and application were:

I. $N_p \equiv 1 \pmod 8$, $(N_p/p_i) = 1$

II. $N_p \equiv -1 \pmod 8$, $(N_p/p_i) = (-1/p_i)$ or equivalently $(-N_p/p_i) = 1$

III. $N_p \equiv 3 \pmod 8$, $(N_p/p_i) = -(-1/p_i)$ or $(-N_p/p_i) = -1$

IV. $N_p \equiv -1 \pmod 8$, $(N_p/p_i) = -(-1/p_i)$ or $(-N_p/p_i) = -1$

V. $N_p \equiv 5 \pmod 8$, $(N_p/p_i) = -(-2/p_i)$ or $(-2N_p/p_i) = -1$

VI. $N_p \equiv 5 \pmod 8$, $(N_p/p_i) = -1$

Definition 6.0.1 *The* **Legendre symbol** (a/p) *for a prime* p*, is defined as*

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{if } p \mid a \\ +1 & \text{if } a \text{ is a quadratic residue of } p \\ -1 & \text{if } a \text{ is a quadratic non-residue of } p \end{cases}$$

Also,

$$\left(\frac{a}{p}\right) = a^{(p-1)/2} \pmod{p}.$$

Problems of quadratic character are particularly well suited to hardware sieving since about half of all residues tend to be excluded (are inadmissible) with respect to each modulus, resulting in high sieving rates and infrequent solutions. We consider quadratic sieves to be moderate density sieves because the solution rate is approximately halved for each modulus specified.

In the following sections we shall present some results on these problems and several related problems. We have divided this chapter into sections dealing with the positive and negative pseudosquares (problems I and II), quadratic polynomials having a high density of primes (III and VI), polynomials of the form x^2+A (IV and II), Lehmer-Lehmer-Shanks problem V, and an examination of Littlewood's bounds on $L(1,\chi)$.

6.1 Pseudosquares

Definition 6.1.1 *Let* p *be an odd prime. The (positive)* **pseudosquare** L_p *is defined as follows:*

1. $L_p \equiv 1 \pmod{8}$
2. *the Legendre symbol* $(L_p/q) = 1$ *for all odd primes* $q \leq p$,
3. L_p *is the least positive non-square integer satisfying (1) and (2).*

Thus, the pseudosquare L_p behaves locally like a perfect square modulo all primes $\leq p$, but is nevertheless not a perfect square. Hall [Hal33] showed that the values of L_p must be unbounded as p increases.

Kraitchik [Kra24, pp. 41–46] seems to have been the first to consider these numbers, and in [Kra24] provides a table of them up to L_{47}. Since then, various authors, most notably D.H. Lehmer, who gave the pseudosquares their name (see Lehmer [Leh54]), have extended Kraitchik's list up to L_{223} (Stephens and Williams [SW90]). Two observations can be made from this list: first, the values of L_p seem to grow very rapidly with respect to p; second, in spite of the first observation, there are many pairs of consecutive primes p, q such that $L_p = L_q$. This latter phenomenon first happens when $L_{59} = L_{61}$, and then occurs a further 14 times among the 30 pseudosquares between L_{67} and L_{223} inclusive.

The growth rate of the pseudosquares is of great importance in two problems in computational number theory: square recognition and primality testing. Cobham [Cob66] has shown that if a number is not a perfect square, then under the Extended Riemann Hypothesis (ERH) it must fail to be a square modulo a small prime p. Thus, we expect that the pseudosquares should grow quickly. More recently, Bach and Sorenson [BS93] discussed the problem of perfect power recognition, which can be of importance in primality testing and factoring. Also, as will be shown in the following section, if pseudosquare growth is sufficiently rapid, then there exists a deterministic polynomial time (in $\log N$) algorithm for determining the prime character of N. At the moment the best unconditional results on the primality testing problem are those of Adelman, Pomerance and Rumely [APR83], who showed that the problem can be solved for a given N by a deterministic algorithm of time complexity $O((\log N)^{c \log\log\log N})$, and of Adelman and Huang [AH92], who show that there is a probabilistic polynomial time algorithm for solving this problem. However, both of these algorithms are very complicated and difficult to implement; whereas the algorithm we shall present, which is conditional on the ERH, is very simple.

6.1.1 Pseudosquares and primality testing

Throughout this section we will use the symbol N to denote an odd, positive integer. If p is an odd prime, we say that $(-1)^{(p-1)/2}p$ is an apparent [1] (quadratic) residue of N when $(N/p) = 1$; we say that $(-1)^{(p-1)/2}p$ is an apparent non-residue of N when $(N/p) = -1$. Furthermore, if the Jacobi symbol $(-1/N) = 1$ (-1), then -1 is said to be an apparent residue (non-residue) of N; also, the apparent character of 2 and -2 can be defined in a similar fashion through the values of the Jacobi symbols $(2/N)$ and $(-2/N)$. Note that by the law of quadratic reciprocity, we see that if N is a prime then any apparent residue of N is in fact a quadratic residue of N and any apparent non-residue of N is a non-residue of N.

In 1929 Kraitchik [Kra29] wrote at some length on how this concept of apparent residue character could be used in factorization and primality testing. In particular, he noticed that if N is not a prime, then an apparent residue of N need not be a quadratic residue of N. Indeed, he went so far as to assert that if enough values of p (possibly including -1 and 2) could be found such that the corresponding apparent residue is in fact a quadratic residue of N, then N must be a prime. He did not specify the number of such apparent residues that would be required, but seemed instead to have contented himself with determining that almost all p values such that $(N/p) = 1$, for p less than some vague bound, actually produced apparent residues that were also quadratic residues of N. For example, he considered the number

$$N_1 = \frac{(2^{120}+1)(2^8+1)}{(2^{24}+1)(2^{40}+1)} = 18518800563924107521.$$

After much labour he determined that for 56 of the 76 primes < 1000 for which $(N_1/p) = 1$, the corresponding apparent residues of N_1 were also quadratic residues of N_1. From this he concluded that N_1 must be a prime. In fact, as noted by Kraitchik in his note at the end of [Leh38], he made an error in his calculations because N_1 is composite.

[1] This is Hall's [Hal33] translation of Kraitchik's [Kra29] "résidu éventuel."

While, as we shall see, the basic idea behind all this has some merit, both Lehmer [Leh30] and Beeger [Bee30] criticized this proof as being mathematically unsatisfactory. In fact, in 1938 Lehmer [Leh38], [Leh39] found that N_1 is composite and exhibited its factors. In fairness to Kraitchik, however, one must observe that he was more interested in his ideas as they applied to the problem of integer factorization, where he was quite successful, than to primality testing; the above technique undoubtedly arose from his frustration at being unable to factor particular values of N (see Kraitchik [Kra31]). Nevertheless, his primality criterion, as explained in [Kra29], can at best only be regarded as empirical.

In [Hal33] Hall produced a mathematically correct version of Kraitchik's idea which is provided in the following theorem.

Theorem 6.1 *If all the factors (not necessarily prime factors) of N are below L_p and if $\left\{-1, 2, -3, \ldots, (-1)^{(p-1)/2}p\right\}$ can be divided into two classes $A = \{a_1, a_2, \ldots, a_r\}$, the apparent residues of N, and $B = \{b_1, b_2, \ldots, b_n\}$, the apparent non residues of N, such that every a_i is also a quadratic residue of N and every $b_i b_j$ is a quadratic residue of N, then N is a prime or a power of a prime.* □

Notice that this is a primality test that involves the pseudosquares. Furthermore, Beeger [Bee39], [Bee46] actually used this test to prove that a 12 digit divisor of the numerator of the Bernoulli number B_{34} and that a 13 digit factor of $12^{45} + 1$ are primes. For example, he first noted that the latter number N_2 had previously been shown to have no prime divisor less than 10^5; hence, any prime divisor of N_2 must be less than $N_2/10^5 < L_{67}$. Further,

$$-1, 2, -3, 5, -11, 13, 17, -23, 29, -31, -43, -47, 53, -59, 61$$

are all apparent residues of N_2 and $-7, -19, 37, 41, -67$ are apparent nonresidues of N_2. By using Kraitchik's method [Kra29] to find quadratic residues of N_2, he was able to show that $-1, 2, 3, 5, 11, 13, 17, 23, 29, 31, 43, 47, 53, 59, 61, 7 \cdot 19, 7 \cdot 37, 7 \cdot 41, 7 \cdot 67$ are all quadratic residues of N_2; hence, N_2 must be a prime or prime power. Since $N_2 < 10^{15}$, and N_2 is not a perfect square, N_2 is a prime.

The main difficulty in utilizing Hall's test is the problem of determining whether a given integer m is a quadratic residue of N. This was usually done by rather tedious hit-or-miss methods which attempted to express certain multiples of N in the form $x^2 - mky^2$. This latter problem can be avoided by making use of a test which is easily derived from the ideas of Selfridge and Weinberger (the S-W test) presented in [Wil78] (see Lukes, Patterson, and Williams [LPWb]). In order to discuss this test we must first define the numbers that we shall denote by M_p, where p is any prime. These must follow the following two properties:

i) the Jacobi symbol $(q/M_p) = 1$ for all primes q such that $2 \leq q \leq p$;

ii) M_p is the least such positive non-square integer.

Theorem 6.2 *If*

1. *All prime divisors of N exceed B,*
2. *$N/B < M_p$ for some prime p,*
3. *$p_i^{(N-1)/2} \equiv \pm 1 \pmod{N}$ for all primes p_i such that $2 \leq p_i \leq p$,*
4. *$p_j^{(N-1)/2} \equiv -1 \pmod{N}$ for some prime p_j such that $2 \leq p_j \leq p$,*

then N is a prime or prime power. □

Of course, this would be a most useful test of primality if it could be shown that M_p grows very quickly as a function of p. At the moment this seems very far from being achieved. We can, however, appeal to a result of Bach [Bac90]

Theorem 6.3 *(Theorem 2 in [Bac90, p. 372]) Denote by $\mathbb{Z}/(m)^*$ the group of reduced residues modulo m. Let G be a nontrivial subgroup of $\mathbb{Z}/(m)^*$ such that $n \in G$ for all $0 < n < x$. Then if the ERH holds, we must have*

$$x < 2(\log m)^2. \quad \square$$

We shall make use of this result to bound M_p from below.

Consider the subgroup G of $\mathbb{Z}/(M_p)^*$ which is made up of all k such that $(k/M_p) = 1$. Since M_p is not a perfect square, there must be some odd prime q such that $q^a \,||\, M_p$ and a is odd. Let t be a quadratic non-residue of q and put $r \equiv t \pmod{q^a}$, $r \equiv 1 \pmod{M_p/q^a}$. Clearly, $r \in \mathbb{Z}/(M_p)^*$ and $(r/M_p) = -1$; thus, G is a non-trivial subgroup of $\mathbb{Z}/(M_p)^*$. Also, for all $0 < n < p$, we get $n \in G$; hence,

$$p < 2(\log M_p)^2,$$

or

$$M_p > e^{\sqrt{p/2}}. \tag{6.1}$$

Notice that if $N < M_p$, there must, by definition of M_p, exist some prime q such that $2 \leq q \leq p$ and $(q/N) \neq 1$. If N is a prime, then either $(q/N) = -1$ and $q^{(N-1)/2} \equiv -1 \pmod{N}$ or $q = N$. If $q = N$, there must be some prime $r < p$ such that $(r/q) = -1$; hence $r^{(N-1)/2} \equiv -1 \pmod{N}$. It follows that if $B = 1$, and N is a prime $< M_p$, then the conditions 3) and 4) of the S-W test must be satisfied. Thus, by (6.1), the S-W test is a deterministic polynomial time prime test for N when $B = 1$.

In order to provide a proof for the S-W test, we require three simple lemmas.

Lemma 6.1 *Let $2^s \,||\, N - 1$ $(s \geq 1)$ and suppose that there exists some $c \in \mathbb{Z}$ such that*

$$c^{(N-1)/2} \equiv -1 \pmod{N}.$$

If q is a prime divisor of N, then $2^s \mid q - 1$.

Proof. Let d be the multiplicative order of c mod q. Since $d \nmid (N-1)/2$ and $d \mid N - 1$, we have $2^s \mid d$; hence, $2^s \mid q - 1$. □

Lemma 6.2 *Let $2^s \,||\, N - 1$ $(s \geq 1)$ and suppose that there exists some $c \in \mathbb{Z}$ such that*

$$c^{(N-1)/2} \equiv \pm 1 \pmod{N}.$$

If $(c/q) \neq 1$ for some prime factor q of N such that $q \equiv 1 \pmod{2^s}$, then $2^s \,||\, q - 1$.

Proof. Let d be the multiplicative order of c mod q. Since

$$c^{(q-1)/2} \equiv -1 \pmod{q},$$

we have $d \mid q-1$, $d \nmid (q-1)/2$. Thus, if $2^r \,||\, q-1$, then $2^r \mid d$; but $d \mid N-1$ means that $r = s$. □

Lemma 6.3 *If $2^s \,||\, N-1$ $(s \geq 1)$, $2^s \,||\, q_1 - 1$, $2^s \,||\, q_2 - 1$, where q_1, q_2 are primes and c is some integer such that $(c/q_1 q_2) \neq 1$, then*

$$c^{(N-1)/2} \not\equiv \pm 1 \pmod{q_1 q_2}.$$

Proof. Suppose

$$c^{(N-1)/2} \equiv \pm 1 \pmod{q_1 q_2}.$$

We may then assume with no loss of generality that $(c/q_1) = -1$, $(c/q_2) = 1$. Let d_1 be the multiplicative order of c modulo q_1. We have $d_1 \mid q-1$, $d_1 \nmid (q_1-1)/2$; thus, $d_1 = 2^s e_1$ where $2 \nmid e_1$. Since,

$$c^{d_1/2} \equiv -1 \pmod{q_1}$$

and $(N-1)/d_1$ is odd, we get

$$c^{(N-1)/2} \equiv -1 \pmod{q_1}.$$

Also, it is easy to see that

$$c^{(N-1)/2} \equiv 1 \pmod{q_2},$$

but this contradicts the assumption that

$$c^{(N-1)/2} \equiv \pm 1 \pmod{q_1 q_2}. \quad \square$$

We are now able to present the proof of the S-W test.

Proof. Let N satisfy the conditions (1), (2), (3), (4) and suppose that N is not a prime or prime power. If q is any prime divisor of N, we get $q \equiv 1 \pmod{2^s}$ if $2^s \,||\, N-1$ by

Lemma 6.1 and condition (4). Since $q < N/B < M_p$, we see that $(p_k/q) \neq 1$ for some prime $p_k (2 \leq p_k \leq p)$; thus, by Lemma 6.2 and condition (3), we find that

$$N = 1 + 2^s t = \prod_{i=1}^{k} q_i$$

where $q_i = 1 + 2^s t_i$, $2 \nmid t_i$ and q_i is a prime for $i = 1, 2, \ldots, k$. Since t is odd, we must have k odd; hence, $k \geq 3$. Since N is not a prime power, there must be at least two distinct primes q_1, q_2 such that $q_1 q_2 \mid N$ and $q_1 q_2 < N/B$. Thus, there must be some $p_k (2 \leq p_k \leq p)$ such that $(p_k/q_1 q_2) \neq 1$. By condition (3) we have

$$p_k^{(N-1)/2} \equiv \pm 1 \pmod{q_1 q_2},$$

which by Lemma 6.3 is impossible. □.

Returning to our definition of M_p, it is well known that $(2/M_p) = 1$ only if $M_p \equiv 1 \pmod 8$ or $M_p \equiv 7 \pmod 8$. If we consider the pseudosquare values $L_p \equiv 1 \pmod 8$, by the quadratic reciprocity theorem, for odd primes q we have the result $(L_p/q) = (q/L_p) = 1$, thus,

$$L_p \geq M_p.$$

Hence, from (6.1) we have

$$L_p > e^{\sqrt{p/2}}. \tag{6.2}$$

We can also combine the ideas of Hall and the S-W test to obtain

Theorem 6.4 *If*

1. *All prime divisors of N exceed B,*
2. *$N/B < L_p$ for some prime p,*
3. *$p_i^{(N-1)/2} \equiv \pm 1 \pmod N$ for all primes p_i such that $2 \leq p_i \leq p$,*
4. *$2^{(N-1)/2} \equiv -1 \pmod N$ when $N \equiv 5 \pmod 8$,*
   $p_j^{(N-1)/2} \equiv -1 \pmod N$ for some odd $p_j \leq p$ when $N \equiv 1 \pmod 8$,

then N is a prime or prime power.

Proof. As before assume that N is not a prime or prime power. If $N \equiv -1 \pmod 4$, then $(-1)^{(N-1)/2} \equiv -1 \pmod 4$. If $N \not\equiv -1 \pmod 4$, then by condition (4) there must be some $c \in \mathbb{Z}$ such that

$$c^{(N-1)/2} \equiv -1 \pmod N.$$

It follows that if $2^s \,||\, N-1$, then $q \equiv 1 \pmod{2^s}$ for all primes $q \mid N$. If $q \mid N$, we must have $q < N/B$; thus, if $q \equiv 1 \pmod 8$, there must exist some p_k such that $(p_k/q) = (q/p_k) \neq 1$. If $q \equiv 5 \pmod 8$, then $(2/q) = -1$, and if $q \equiv -1 \pmod 4$, then $(-1/q) = -1$; thus by Lemma 6.2, condition (3) and the same reasoning as that used in the proof of the S-W test, we must have at least two distinct prime divisors q_1, q_2 of N such that $q_1 q_2 < N/B$. If $q_1 q_2 \equiv 1 \pmod 8$, there must exist some prime p_k such that $(p_k/q_1 q_2) = (q_1 q_2/p_k) \neq 1$; if $q_1 q_2 \equiv 5 \pmod 8$, then $(2/q_1 q_2) = -1$, and if $q_1 q_2 \equiv -1 \pmod 4$, then $(-1/q_1 q_2) = -1$. By condition (3) and Lemma 6.3 we get a contradiction. □

Also note that condition (3) of Theorem 6.4 must hold if N is prime. Also, if $N \equiv 5 \pmod 8$, then $2^{(N-1)/2} \equiv -1 \pmod N$ when N is a prime, and if $N \equiv 1 \pmod 8$, we see by the same reasoning as used earlier that $(p_k/N) = -1$ for some odd $p_k \leq p$ when N is a prime. Thus, if $B = 1$, and $N < L_p$, then conditions (3) and (4) must hold.

From the above we can see that if the growth rate of the pseudosquares is sufficiently rapid, then primality testing of N could be achieved in deterministic polynomial time. By (6.2), this is certainly the case under the ERH, but it might be possible to show that a result like (6.2) holds without appealing to the ERH. Thus, it is of some interest to examine, at least empirically, the growth rate of L_p.

6.1.2 Pseudosquare results

Because of the importance of the question of the pseudosquare growth rate, it would be very helpful if the table in [SW90] could be extended. Unfortunately, this is very difficult to do simply because the pseudosquares do seem to grow very quickly.

We first used a larger version of OASiS called OASiS II (OASiS with an additional sieve unit, see [SW90, p. 63] to search for pseudosquares beyond the limit of Table 2 in [SW90]; however, after many months of continuous running of the device, we were able to find only two additional pseudosquares: L_{227} and L_{229}. Indeed, it was this slow sieving rate of OASiS II (for the pseudosquare problem) that motivated the construction of the MSSU.

The parallel architecture of the MSSU is extremely well suited to computing pseudosquares. The problem of determining the values of L_p can be partitioned to combine the moduli 8, 3, 5, 7 and 11, producing an implicit modulus of 9240 with 30 associated residue classes. The resulting effective sieving rate utilizing 30 sieve chips (one employed per residue class with single residue optimization implemented for the modulus 9240) is thus:

$$9240 \times 192 \text{ million/sec} = 1.774 \times 10^{12}/\text{sec} = 6.387 \times 10^{15}/\text{hour}$$

This optimization can be extended by combining the moduli 8, 3, 5, 7, 11 and 13, to form 180 residues, partitioning the problem into 6 sub-problems of 30 chips each, realizing an additional two-fold increase in the sieving rate.

The pseudosquare problem specifies a set of linear congruences which generate a large number of perfect square solutions which must be discarded using a perfect square filter. At low count values, where the density of perfect squares is greatest, sieve hardware is often idle while filter resources are overwhelmed. To prevent filtering from being a bottleneck, the perfect square filter PSQR described in Section B.1 was used. This filter can determine whether or not a given 128-bit integer is a perfect square in under 1 millisecond.

When running the pseudosquares problem on the MSSU, starting from an initial count of 1, we observed during the first hour that sieving proceeded at 35.6% of the theoretical maximum speed. As the density of perfect squares diminished, this value increased to 86.5% during the second hour, and 94.1% during the third. At higher values, sieving

rates exceed 99% of the maximum speed of the hardware, attesting to the efficiency of filtering on the MSSU.

After running the MSSU for a few months on the pseudosquare problem we produced the results [2] in Table 6.1 below.

Sieving was extended up to the bound of $2^{64} \doteq 1.8447 \times 10^{19}$. This allows the primality test described above to be used to the limit of unsigned 64-bit integer math, now available on many modern computer systems such as the DEC Alpha architecture [Sit92].

In Figure 6.1 we have plotted the values of $\log L_p$ against p. We have also plotted the values of $\sqrt{p/2}$. Note that the growth rate of $\log L_p$ is much greater than that of $\sqrt{p/2}$, supporting (6.2). Thus, from the numerical data that we have so far been able to determine, it appears that L_p has a growth rate well in excess of $e^{\sqrt{p/2}}$.

Let p_i denote the i^{th} prime ($p_1 = 2$). If we make the somewhat naive assumption (there will always be fluctuations) that the solutions of

$$x \equiv 1 \pmod 8,\ (x/p_i) = 1\ (i = 1, 2, \ldots, n) \tag{6.3}$$

are equi-distributed in the region $0 < x < 8p_2p_3p_4 \ldots p_n$, then we would expect that if $p = p_n$, then[3]

$$L_p \approx 8p_2p_3 \ldots p_n/S,$$

where S is the number of solutions of (4.1) in the given region. Since

$$S = \prod_{i=2}^{n} (p_i - 1)/2,$$

we get

$$L_p \approx 2^{n+1} \prod_{i=1}^{n} p_i/(p_i - 1).$$

At this point we make use of Mertens's Theorem (see [HW79, p. 351]).

[2] Further data on the pseudosquares, including a list of least prime solutions, can be found in Tables 6.24 and 6.25 in Section 6.5.3.

[3] Of course we are assuming here that the least solution of (6.3) is not a perfect square.

Theorem 6.5 *(Mertens's Theorem)*

$$\prod_{p \leq x} \frac{(p-1)}{p} \sim \frac{e^{-\gamma}}{\log x},$$

where $\gamma \doteq 0.57721$ *is Euler's constant.*

By Mertens's Theorem we get

$$L_p \approx c_1 2^n \log p$$

for a constant $c_1 = 2e^{\gamma}$, suggesting that the n^{th} pseudosquare should grow exponentially in n.

We now state the Prime Number Theorem [HW79, p. 9].

Theorem 6.6 *(Prime Number Theorem) The number of primes not exceeding* x*, denoted by* $\pi(x)$*, is asymptotic to* $x/\log x$*:*

$$\pi(x) \sim \frac{x}{\log x}.$$

An equivalent statement of the above theorem is

$$n \sim p_n / \log p_n.$$

Indeed, our heuristic analysis for the growth rate of the pseudosquares suggests that since, $n \sim p_n/\log p_n$, we should have a growth rate for L_p of the form $2^{(p/\log p)(1+o(1))}$, a suggestion confirmed numerically in Figure 6.1. We should also point out that this growth rate for L_p could also be predicted from the plausible assumption, made by Bach and Huelsberger in [BH93], that pseudosquares will provide extreme values of $G(p)$. Here $G(n)$ represents the smallest value of x such that the primes $\leq x$ generate the multiplicative group modulo n.

In Figure 6.2, we present a plot of the values of $\log(L_p/\log n)$ for the L_p given in Table 6.1 against the values of n. The straight line represents the least squares line fitted to these data. For this line we have

$$y = 0.67987n + 4.4835$$

Although there are very few sample points for our graph, the slope of this line appears to be approaching the value of $\log 2 \doteq 0.69315$, which we would expect from our heuristic analysis.

Thus, these numerical data lend some support to the heuristic belief that L_p should grow exponentially in n. While nothing has been proved, the data presented here make a very strong empirical case for a fast growth rate of L_p, suggesting that it may be possible to provide a proof, without using the ERH, that L_p grows sufficiently quickly that Theorem 6.4 provides a deterministic polynomial primality test when $B = 1$.

p	L_p	Source
3	73	Kraitchik (1924)
5	241	moveable strips
7	1 009	
11	2 641	
13	8 089	
17	18 001	
19	53 881	
23	87 481	
29	117 049	
31	515 761	
37	1 083 289	
41	3 206 641	
43	3 818 929	
47	9 257 329	
53	22 000 801	Lehmer (1928)
59,61	48 473 881	bicycle chains
67	175 244 281	Lehmer (1954)
71,73	427 733 329	SWAC
79	898 716 289	
83,89,97	2 805 544 681	Lehmer, Lehmer,
101	10 310 263 441	and Shanks (1970)
103	23 616 331 489	DLS-127
107,109	85 157 610 409	
113,127	196 265 095 009	
131,137,139	2 871 842 842 801	Lehmer (1973)
149,151	26 250 887 023 729	DLS-157
157,163,167	112 434 732 901 969	Patterson and
173,179	178 936 222 537 081	Williams (1988)
181,191	696 161 110 209 049	UMSU
193	2 854 909 648 103 881	Stephens and
197,199	6 450 045 516 630 769	Williams (1989)
211,223	11 641 399 247 947 921	OASiS
227	190 621 428 905 186 449	Lukes and Williams
229	196 640 248 121 928 601	(1991) OASiS II
233	712 624 335 095 093 521	MSSU (1994)
239	1 773 855 791 877 850 321	
241	2 327 687 064 124 474 441	
251	6 384 991 873 059 836 689	
257	8 019 204 661 305 419 761	
263,269,271	10 198 100 582 046 287 689	
277	>40 000 000 000 000 000 000	

Table 6.1: Pseudosquares

(Problem I in [LLS70])

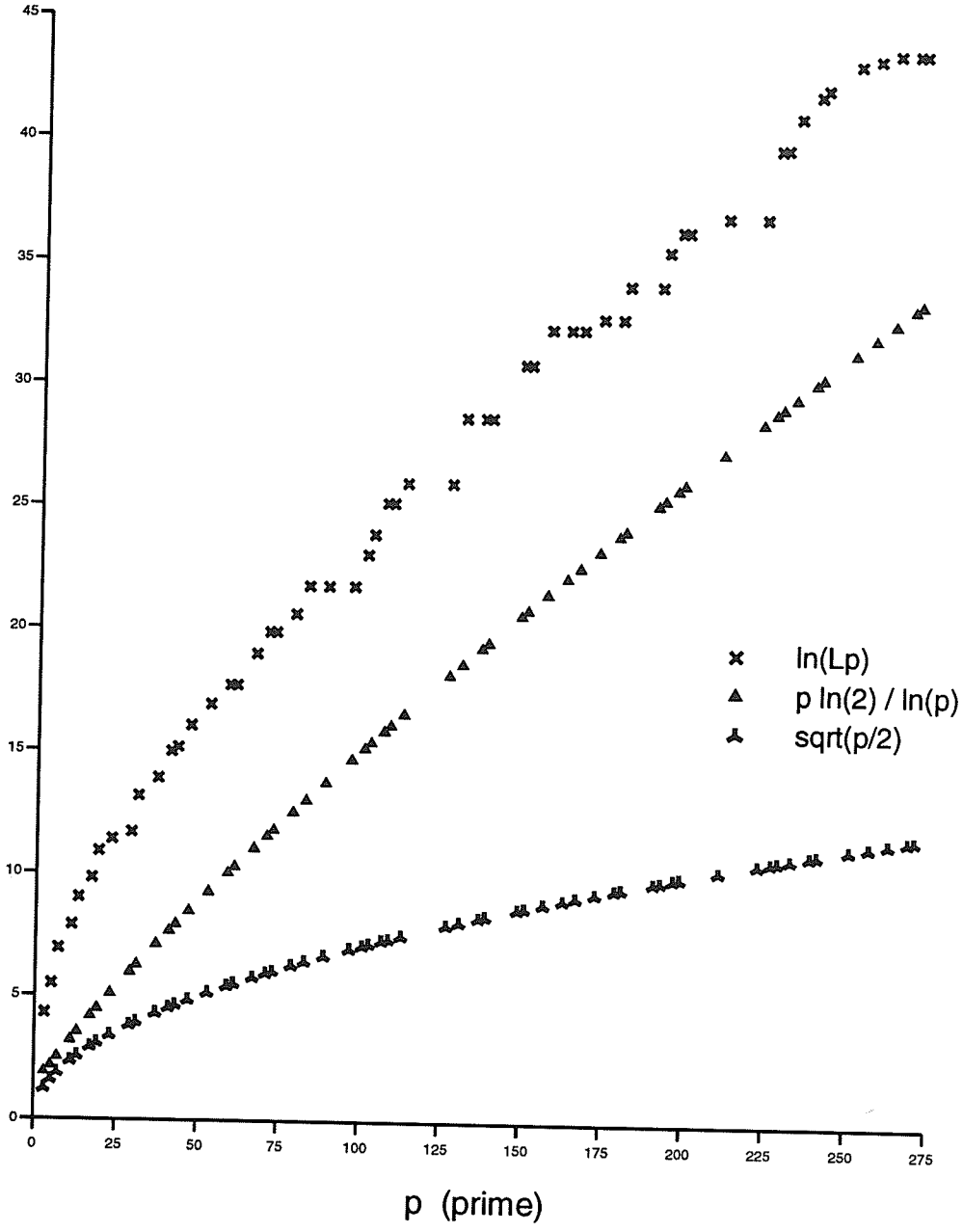


Figure 6.1: Pseudosquare growth vs. p

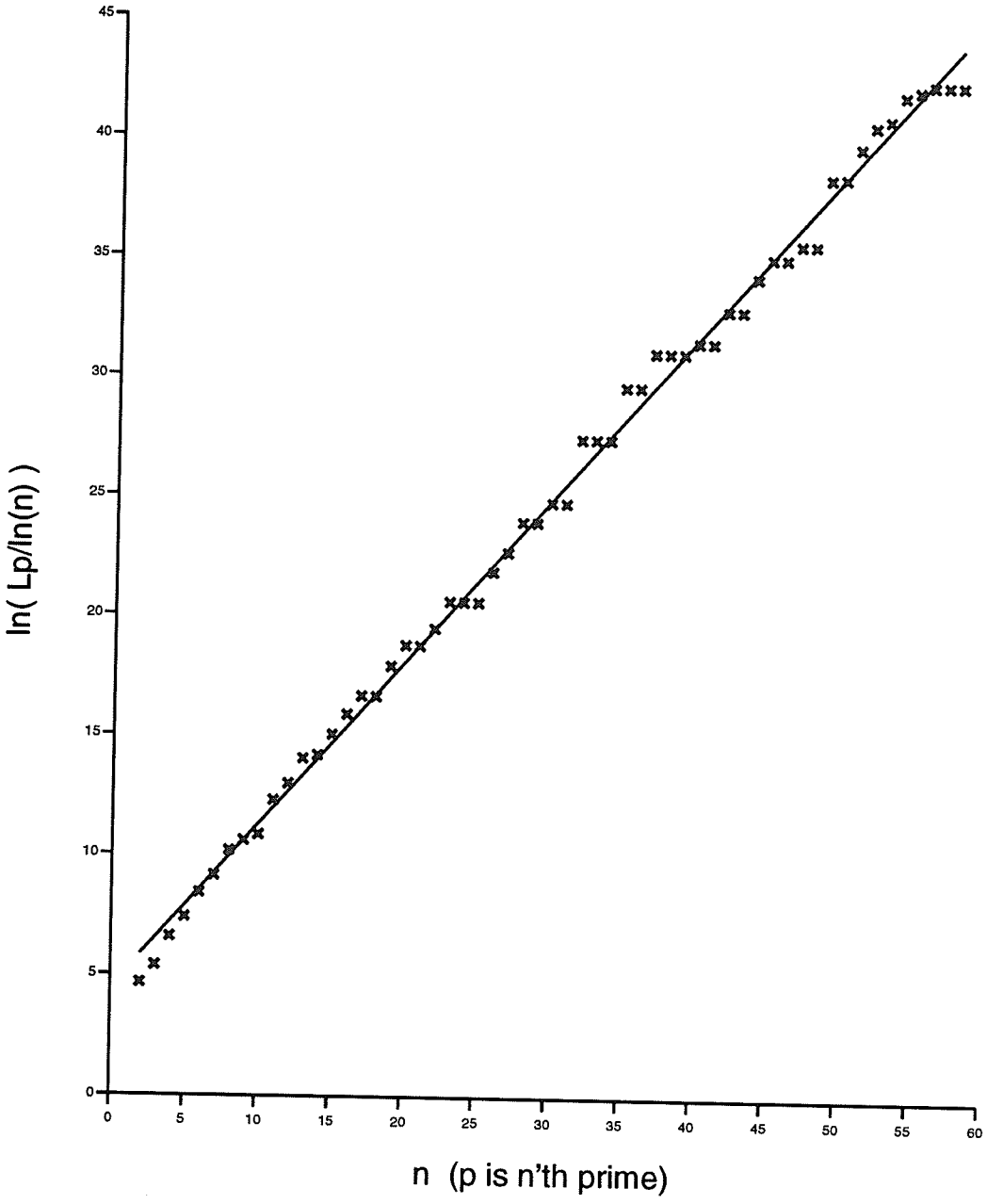


Figure 6.2: Pseudosquare growth vs. n

6.1.3 Negative Pseudosquares

Definition 6.1.2 *Let p be an odd prime. The* **negative pseudosquare** N_p *is defined as follows:*

1. $N_p \equiv -1 \pmod 8$
2. *the Legendre symbol* $(-N_p/q) = 1$ *for all odd primes* $q \leq p$,
3. N_p *is the least positive non-square integer satisfying (1) and (2).*

Recently the table of N_p given in [Ste89] has been extended by Bronson and Buell [BB93] from N_{211} to N_{227} using their FPGA-based sieve. Using MSSU, we have extended this list even further, to N_{269}. The updated list [4] is given in Table 6.2.

By quadratic reciprocity it follows that

$$(p_i/L_p) = (p_i/N_p) = 1$$

for all primes p_i such that $2 \leq p_i \leq p$; thus it is easy to see that

$$M_p = min\{L_p, N_p\}.$$

It is possible, then, for the test of Theorem 6.4 to be slightly better than the S-W test, in that L_p seems often to exceed N_p. Shanks [Sha73] noted that the pseudosquare values appear to be delayed by competition with the perfect squares, a phenomenon which does not occur for the negative pseudosquares.

[4] Further data on the negative pseudosquares, including a list of least prime solutions, can be found in Tables 6.22 and 6.23 in Section 6.5.3.

p	N_p	Source
3	23	Lehmer, Lehmer
5	71	and Shanks (1970)
7	311	DLS-127
11	479	
13	1 559	
17	5 771	
19	10 559	
23	18 191	
29	31 391	
31	†118 271	
37,41	366 791	
43,47,53	2 155 919	
59,61	6 077 111	
67	98 538 359	
71	120 293 879	
73,79	131 486 759	
83	508 095 719	
89,97,101,103,107,109	2 570 169 839	
113,127	328 878 692 999	
131	513 928 659 191	
137	844 276 851 239	Stephens
139	1 043 702 750 999	OASiS (1989)
149	4 306 732 833 311	
151	8 402 847 753 431	
157	47 375 970 146 951	
163	52 717 232 543 951	
167	100 535 431 791 791	
173	251 109 340 045 079	
179,181,191	493 092 541 684 679	
193,197,199,211	1 088 144 332 169 831	
223	71 608 584 429 428 591	Bronson and Buell
227	88 163 809 868 323 439	SPLASH (1991)
229	218 748 706 425 968 039	MSSU (1994)
233	423 414 931 359 807 911	
239	695 681 268 077 667 119	
241,251	1 116 971 853 972 029 831	
257,263,269	3 546 374 752 298 322 551	
271	>10 000 000 000 000 000 000	

Table 6.2: Negative Pseudosquares

† value first appeared in [BB93]
(Problem II in [LLS70])

6.2 Quadratic Polynomials Having a High Density of Primes

Let $f_A(x)$ denote the quadratic polynomial $x^2 + x + A$ ($A \in \mathbb{Z}$) and let $P_A(n)$ denote the number of prime values assumed by $f_A(m)$ for $0 \leq m \leq n$. In 1772 Euler discovered that $P_{41}(39) = 40$; since then it has been learned (see, for example, Ribenboim [Rib88a]) that $P_A(A-2) = A-1$ can only occur for $A = 2, 3, 5, 11, 17, 41$.

In 1938, Beeger [Bee39] found the polynomials $f_A(x)$ with $A = 19421$, 27941, and 72491. He did this by computing all the positive integers $N_p < 10^6$ of the form $8t+3$ such that $(-N_p/q) = -1$ for $q = 3, 5, 7, \ldots, 43$; the only such numbers are $N_p = 77683, 111763, 289963$. Poletti [Pol39] computed tables of $P_A(n)$ for these values of A and various values of n up to 11000. He discovered that the polynomials $f_{27941}(x)$ and $f_{72491}(x)$ seemed to be better at producing primes than the famous $f_{41}(x)$. For example, he found that $P_{41}(11000) = 4506$; whereas $P_{27941}(11000) = 4819$ and $P_{72491}(11000) = 4923$ [5]. Today we know that $P_{41}(10^6) = 261081$ and $P_{27941}(10^6) = 286129$. Thus, it appears that $f_{27941}(x)$ has a larger density of prime values than $f_{41}(x)$.

6.2.1 The conjecture of Hardy and Littlewood

Neither Beeger nor Poletti seems to have been aware of Hardy and Littlewood's [HL23] conjecture F. This is somewhat surprising, given the relevance of this conjecture to the problems that they were considering. If we denote by $\pi_f(n)$ the number of prime values assumed by the quadratic polynomial $f(x) = ax^2 + bx + c$ for $x = 0, 1, 2, \ldots, n$, this conjecture can be given in the form below.

[5] Poletti's calculations contain some errors since the actual values of $P_{41}(11000) = 4507$, $P_{27941}(11000) = 4831$ and $P_{72491}(11000) = 4943$.

Conjecture 6.1 (F) *Let $a > 0, b, c$ be integers such that $\gcd(a,b,c) = 1$, $d = b^2 - 4ac$ is not a square and $a+b, c$ are not both even. Then there are infinitely many primes of the form $f(m)$ and*

$$\pi_f(n) \sim \frac{1}{2}\varepsilon\, C_f\, Li(n),$$

where

$$Li(n) = \int_2^n \frac{dx}{\log x},$$

$$\varepsilon = \begin{cases} 1 & \text{when } 2 \nmid a+b. \\ 2 & \text{otherwise,} \end{cases}$$

and

$$C_f = \prod_{\substack{p>2 \\ p|(a,b)}} \frac{p}{p-1} \prod_{\substack{p>2 \\ p\nmid a}} (1 - (d/p)/(p-1)).$$

The products in the expression for C_f are taken over primes only. Note here that it is really the value of εC_f which determines the density of prime values assumed by f.

We can derive a special version of Conjecture F which applies to $f_A(x)$:

$$P_A(n) \sim C(D)\, L_A(n),$$

where

$$L_A(n) = 2\int_0^n \frac{dx}{\log f_A(x)},$$

$$C(D) = \prod_{p\geq 3} (1 - (D/p)/(p-1)). \tag{6.4}$$

Here $D = 1 - 4A$. It is clear that to maximize the value of $C(D)$ we want $(D/p) = -1$ for as many small primes as possible.

For example, when $A = 41$, we have $C(-163) \doteq 3.3197732$; whereas, if $A = 27941$, then $C(-111763) \doteq 3.6319998$ and if $A = 72491$, then $C(-289963) \doteq 3.6947081$. Thus, the numerical values that we have for $C(D)$ here tend to confirm Poletti's observations about the superiority of Beeger's polynomials to $f_{41}(x)$ in producing primes. In fact, it seems that Conjecture F provides us with a very accurate predictor of prime densities

for quadratic polynomials. Indeed, Fung and Williams [FW90] found that values of $C(D)$ corresponded very closely to the actual values of $P_A(10^6)/L_A(10^6)$. Previously, experiments by Shanks [Sha59, Sha60, Sha63] have tended to confirm the truth of Hardy-Littlewood Conjecture F.

Shanks [Sha75] developed a method for rapidly computing values of $C(D)$ by transforming the very slowly convergent infinite products of the Dirichlet functions into more rapidly converging ones. He used this technique to make the first accurate computation of $C(-163) \doteq 3.3197732$. Fung and Williams [FW90] developed a faster method of computing $C(D)$ however, unlike Shanks's method, theirs was contingent on the Extended Riemann Hypothesis (ERH). For this experiment, values of $C(D)$ were computed by Michael Jacobson Jr. [Jac95] using a similar algorithm. To permit a broader search, Jacobson extended the methods to allow $C(D)$ to be computed for both positive and negative discriminants D.

Under the ERH, using a result of Littlewood [Lit28] it can be shown that

$$C(D) < \{1 + o(1)\}e^{\gamma} \log\log|D| \tag{6.5}$$

where $e^{\gamma} \doteq 1.781072418$ (see [LPWa]). Thus, we would not expect to see a large rate of growth in $C(D)$, but it is nevertheless of some interest to find values of D for which $C(D)$ is relatively large.

6.2.2 Some results

In order to find values of D for which $C(D)$ is large, we used the MSSU to search for all values of D such that $|D| < 10^{19}$, $D \equiv 5 \pmod 8$, and $(D/p) = -1$ for all odd primes ≤ 199. Tables 6.3 and 6.4 list the $C(D)$ hichamps, those D having a higher $C(D)$ than all preceding values of D. After examining the several thousand numbers produced by this search, the largest $C(D)$ value we found is

$$C(D) \doteq 5.3670819$$

for

$$D = -13598858514212472187.$$

Thus, if conjecture F holds, then

$$x^2 + x + A,$$

where

$$A = 3399714628553118047$$

has the largest asymptotic density of primes for any polynomial of this type currently known.

This surpasses the previous record set by Stephens [Ste89] using the OASiS sieve of

$$C(D) \doteq 5.0978921$$

for

$$D = -2068660612674307.$$

In Table 6.5 we have listed some values of A corresponding to champion prime-producing polynomials, ordered by increasing value of $C(D)$. The values of L_A were computed using the GP/PARI CALCULATOR Version 1.38 [BBCO93] to a precision of 8 digits, while a simple program written in Maple V Release 2 [CGG+88] was used to calculate the values of P_A. Note that the first polynomial attributed to Fung & Williams, extracted from Table 4.2 of [FW90], which appears towards the middle of our list, has the highest value of $P_A(10^6)$ known. Up to this value of A, the value of $P_A(10^6)$ appears to be increasing, however, values of A following this entry appear to have decreasing values of $P_A(10^6)$. This table also illustrates that the values of $P_A(10^6)/L_A(10^6)$ appear to be in close agreement with the computed values of $C(D)$, with the notable exception of the second last entry where there is some slight discrepancy. In order to determine whether this discrepancy persisted, we computed $P_A(10^7)/L_A(10^7)$ for $A = 1712329866165608771$ shown in Table 6.6. Here the value of P/L is in very close agreement with the computed value of $C(D)$.

D	$h(D)$	$L(1,\chi)$	$C(D)$
-4311527414591923	3791896	0.181422319	4.5293043
-5513463660887323	4214276	0.178303796	4.6086597
-8842819893041227	5188215	0.173329485	4.7414735
-11779882219755787	5904498	0.170907994	4.8086435
-14363876114143483	6478729	0.169825875	4.8393795
-15326624594334307	6664840	0.169128322	4.8590033
-30462609261723907	9340770	0.168131466	4.8883007
-32779240456803163	9520419	0.165198649	4.9753684
-50792117776428667	11782274	0.164240716	5.0043010
-221328140358231307	24591656	0.164217459	5.0050646
-234391954943494723	24980688	0.162099963	5.0706939
-369885383792662483	31346105	0.161919767	5.0766794
-441899002218793387	33684408	0.159190549	5.1635912
-554395014308976163	37602038	0.158654209	5.1814176
-803608018073876563	45224688	0.158490531	5.1864453
-2038991582966171563	71351592	0.156980541	5.2369507
-2039953459173530587	70825967	0.155787373	5.2765336
-6849319464662435083	128288704	0.153997822	5.3384020
-13598858514212472187	179800672	0.153175728	5.3670819

Table 6.3: $C(D)$-hichamps ($D < 0$)

D	$R(D)$	$h(D)$	$L(1,\chi)$	$C(D)$
370095509388197	794079.6472497177	2	0.165107631	4.9779328
16710980998953317	5296924.2425040266	2	0.163901444	5.0144216
1897400302017991 7	2737025.3979827850	4	0.158960540	5.1711431
587108439330001613	30377994.3008864805	2	0.158584203	5.1831340
2430946649400343037	30781378.0110772471	4	0.157939344	5.2048129
3512773592849667053	146959147.1762363224	1	0.156820032	5.2422843
4927390995446922917	86988957.8224337937	2	0.156752908	5.2437622

Table 6.4: $C(D)$-hichamps ($D > 0$)

A	Source	$P_A(10^6)$	$L_A(10^6)$	P/L	$C(D)$
41	Euler	261081	78716.099	3.317	3.3197732
27941	Beeger	286129	78620.480	3.639	3.6319998
21425625701	Fung & Williams	361841	76877.960	4.707	4.7073044
132874279528931	Fung & Williams	312975	61495.079	5.089	5.0870883
517165153168577	Stephens & Williams	300923	59031.829	5.098	5.0976398
1712329866165608771	Lukes & Williams	253677	47636.756	5.325	5.3384021
3399714628553118047	Lukes & Williams	251841	46871.092	5.373	5.3670819

Table 6.5: Selected $C(D)$-hichamps

A	$P_A(10^7)$	$L_A(10^7)$	P/L	$C(D)$
41	2208197	666062.07	3.3153	3.3197732
1712329866165608771	2543261	476367.34	5.3389	5.3384021

Table 6.6: Computations to $x = 10^7$ for Select $C(D)$-hichamps

In Table 6.7 below we extend the list of known values for N_p. We also list the value of $C(-N_p)/e^{\gamma}\log\log N_p$ which from (6.5) we would expect to be not too far from 1. Notice that with the exception of $N_p = 163$ (see Shanks [Sha73]), the value of $C(-N_p)/e^{\gamma}\log\log N_p$ is always less than 1. In fact, it does not seem to vary very much from .8, a phenomenon that also occurs in Table 6.8.

If we define M_p to be the least positive integer such that $M_p \equiv 5 \pmod 8$ and $(M_p/q) = -1$ for all primes $q \leq p$, we would expect $f_A(x)$ to have a large density of primes when $A = (-M_p + 1)/4$. In fact, according to Poletti [Pol51], Beeger was the first to make a table of M_p values. Apparently, without knowledge of Beeger's earlier work, Lehmer [Leh37] computed M_p values up to M_{109} [6]. This table was extended by Lehmer, Lehmer and Shanks [LLS70] using the DLS-127 in 1970. In Table 6.8 we give the current known list of M_p values.

[6]The evidence in Lehmer's unpublished sieve log indicates that the photoelectric sieve was used for these computations.

p	N_p	$C(-N_p)$	$C/e^\gamma \log\log N_p$	Source
3	19	.94222046	.4898691072	Lehmer
5, 7	43	1.6297209	.6907193690	(1936)
11, 13	67	2.0873308	.8160080075	
17, 19, 23, 29, 31, 37	163	3.3197732	1.144902784	
41, 43	77683	3.3003388	.7652967791	
47	1333963	3.8123997	.8088270556	
53, 59	2404147	3.7793704	.7896114897	
61	20950603	3.8410195	.7634423948	
67	36254563	3.6365197	.7146963938	
71	51599563	3.8514289	.7516514185	
73, 79	96295483	3.8528890	.7430185677	
83	114148483	4.0332358	.7753453468	
89, 97, 101, 103	269497867	4.1092157	.7778960664	
107	585811843	4.1185705	.7694918661	
109, 113	52947440683	4.3245257	.7572257271	Lehmer,
127	71837718283	4.6097143	.8040827662	Lehmer,
131, 137	229565917267	4.2679170	.7341016471	Shanks
139	575528148427	4.4746374	.7616017156	(1970)
149, 151, 157, 163	1432817816347	4.4163429	.7442057577	
167	6778817202523	4.5565681	.7555810581	Lehmer
173	16501779755323	4.7524812	.7812226289	(unpublished)
179, 181	30059924764123	4.8379057	.7907472063	
191, 193, 197	110587910656507	4.9711959	.8029226034	UMSU (1988)
199	4311527414591923	4.5293043	.7096431032	OASiS (1989)
211, 223	10472407114788067	4.6162389	.7183819661	MSSU
227, 229, 233, 239, 241	22261805373620443	4.8576312	.7517308637	
251	132958087830686827	4.9146545	.7509543608	
257	441899002218793387	5.1635913	.7826008352	
263, 269	2278509757859388307	5.0669199	.7598318105	
271	5694230275645018963	5.2163043	.7777801235	
277	9828323860172600203	5.2552050	.7809755689	

Table 6.7: Values of N_p and their sources

(Problem III in [LLS70])

p	M_p	$C(M_p)$	$C/e^\gamma \log\log M_p$	Source
3	5	1.7733051	2.092183985	Beeger
5	53	1.3831458	.5632129467	(1938)
7, 11	173	2.0427655	.6995030019	
13	293	2.4386997	.7882818434	
17	437	2.7933935	.8689101049	
19, 23	9173	3.1227858	.7930297198	
29	24653	3.1631443	.7675628664	
31, 37, 41	74093	3.0809338	.7156647901	
43	170957	3.3299831	.7511667053	
47, 53, 59	214037	3.2704656	.7323038660	
61	2004917	4.0077796	.8412252523	Lehmer,
67	44401013	3.8743032	.7583553099	Lehmer,
71	71148173	4.1026493	.7957213588	Shanks
73, 79	154554077	3.6684052	.7013201448	(1970)
83, 89, 97	163520117	3.8307572	.7316144432	
101, 103	261153653	4.3158954	.8174685445	
107, 109, 113	1728061733	4.2447622	.7795290236	
127	9447241877	4.5541813	.8158487534	
131	19553206613	4.6250203	.8203779580	
137, 139	49107823133	4.8420287	.8486487840	
149, 151, 157, 163	385995595277	4.7144914	.8060573959	Lehmer (unpublished)
167	13213747959653	4.5147795	.7437444245	MSSU
173	14506773263237	4.7257867	.7778006998	
179, 181	57824199003317	4.7059530	.7645367358	
191, 193	160909740894437	4.7279560	.7611110267	
197, 199	370095509388197	4.9779329	.7956168667	
211	1409029796180597	4.9274990	.7788910126	
223	4075316253649373	4.9577054	.7771038698	
227, 229, 233	18974003020179917	5.1711431	.8011871230	
239, 241	224117990614052477	4.7415726	.7219229456	
251, 257, 263	637754768063384837	4.7753226	.7220023457	
269, 271, 277, 281, 283	4472988326827347533	5.0085747	.7479198393	

Table 6.8: Values of M_p and their sources

(Problem VI in [LLS70])

6.3 Polynomials of the Form $x^2 + A$

Lehmer, Lehmer and Shanks [LLS70] defined problem IV as those values of $N_p \equiv -1 \pmod 8$ with $(N_p/p_i) = -(-1/p_i)$. These numbers have application in the study of the density of primes of the corresponding polynomials $f_{x^2+A}(x) = x^2 + A$. Shanks [Sha60, Sha63] specialized Hardy and Littlewood's [HL23] conjecture F to apply to polynomials of the form $f_{x^2+A}(x)$. Using terminology consistent with the previous section (on prime-producing polynomials of the form $x^2 + x + A$), if A is an integer which is not a negative square, $A \neq -k^2$, and if $P_{x^2+A}(N)$ is the number of primes of the form $x^2 + A$ for $1 \leq x \leq N$, then

$$P_{x^2+A}(N) \sim C(-A)\, L_{x^2+A}(N),$$

where

$$L_{x^2+A}(N) = \int_2^N \frac{dx}{\log f_{x^2+A}(x)},$$

and the constant $C(-A)$ is the infinite product

$$C(-A) = \prod_{p \geq 3} \left(1 - \left(\frac{-A}{p}\right) \frac{1}{p-1}\right),$$

taken over all odd primes p, and for which $(-A/p)$ is the Legendre symbol.

From the above we see that to maximize the value of $C(-A)$ we want the Legendre symbol $(-A/p) = -1$ for as many of the small primes p as possible. These are the conditions specified by Problem IV of Lehmer-Lehmer-Shanks. To minimize the value of $C(-A)$, it is desirable to have the Legendre symbol $(-A/p) = 1$ for as many small primes p as possible. Thus, we would expect solutions of the negative pseudosquares problem (Lehmer-Lehmer-Shanks problem II) to correspond to values of A for which the quadratic polynomial $x^2 + A$ has an unusually low asymptotic density of prime values.

Shanks developed a rapidly convergent method for computing $C(-A)$ and determined that $C(-1) \doteq 1.37281346$. Thus we would expect that

$$P_{x^2+1}(N) \sim 1.37281346 \int_2^N \frac{dx}{\log(x^2+1)}. \tag{6.6}$$

In 1973, Wunderlich [Wun73] computed the actual values of $P_{x^2+1}(10^6) = 54110$ and $P_{x^2+1}(10^7) = 456362$ which agree very closely with the values of 53969.76 and 456403.75 respectively, assumed by (6.6). Using a simple program written in Maple [CGG+88] we computed $P_{x^2+1}(10^8) = 3954181$, which also agrees very closely with the value of 3966660.2 predicted by (6.6).

In Tables 6.9 and 6.10 we present some statistics on those values of A for which the asymptotic density of primes assumed by $f_{x^2+A}(x)$, as given by $C(-A)$, is unusually high. Again, the data in this list were computed by Michael Jacobson Jr. [Jac95], using techniques similar to those described in the previous section. The data were compiled by computing all values of $N_p \equiv -1 \pmod 8$ such that $(-N_p/p_i) = -1$ for all odd primes $p \leq 199$ and $0 \leq N_p \leq 10^{18}$. These are the conditions specified by problem IV of Lehmer, Lehmer and Shanks.

A	$h(-A)$	$L(1,\chi)$	$C(-A)$
2216664949125463	7784956	0.519464661	4.7458184
2832651136748623	8322068	0.491229666	5.0198174
10594288233662863	15421007	0.470681051	5.2394369
225086816184343327	69841925	0.462477964	5.3321934

Table 6.9: $C(-A)$-hichamps $(A > 0)$

A	$P_{x^2+A}(10^6)$	$L_{x^2+A}(10^6)$	P/L	$C(-A)$
2216664949125463	134547	28300.550	4.754	4.7458184
2832651136748623	141314	28105.536	5.028	5.0198174
10594288233662863	142132	27100.861	5.245	5.2394369
225086816184343327	133443	25027.941	5.332	5.3321934

Table 6.10: Computations to $x = 10^6$ for $C(-A)$-hichamps

Tables 6.11 and 6.12 list the same statistics for those values of A for which the asymptotic density of primes assumed by $f_{x^2+A}(x)$, as given by $C(D)$, is unusually low (also computed by Michael Jacobson Jr.). The data correspond to all values of $N_p \equiv -1 \pmod 8$

such that $(-N_p/p_i) = 1$ for all odd primes $p \leq 199$ and $0 \leq N_p \leq 10^{19}$. These are the conditions specified by problem II of Lehmer, Lehmer and Shanks, the negative pseudosquares (see Section 6.1.3).

Note that the approximation P/L is in close agreement to the value of $C(-A)$ for both Tables 6.10 and 6.12. The values of L_{x^2+A} were computed using the GP/PARI CALCULATOR Version 1.38 [BBCO93] to a precision of 8 digits, while Maple V Release 2 was used to calculate the P_{x^2+A} values.

A	$h(-A)$	$L(1,\chi)$	$C(-A)$
1088144332169831	95504500	9.09558120	0.14528180
2085366568194719	135018400	9.28862920	0.14225381
34571718084771479	562662108	9.50685916	0.13897783
121036243821187439	1054370893	9.52106570	0.13877611
140477714238295199	1138177140	9.54016784	0.13849255
391724890376888591	1901072300	9.54240670	0.13846496
427407728556837551	2008951026	9.65379421	0.13685911
695681268077667119	2582308636	9.72641656	0.13582738
1026219868955993351	3148812910	9.76509607	0.13529083
1687542416179693919	4038842988	9.76741170	0.13526493
1811049832613761871	4189547100	9.78029038	0.13508619
2503702890956049719	4931770683	9.79177334	0.13492130
2570304063059644799	5012639100	9.82254571	0.13450393
4995724305819922919	7031222944	9.88283420	0.13368059

Table 6.11: $C(-A)$-lochamps $(A > 0)$

In an attempt to find long strings of composite values assumed by quadratic polynomials, we tested all of the values in Table 6.12 for $x = 0, 1, 2, \ldots 10^6$. The largest string of composite values was for the polynomial

$$(x + 216529)^2 + 1687542416179693919 = x^2 + 433058x + 1687542463064501760$$

which is composite for $x = 0, 1, \ldots 2782$, the first 2783 values of the polynomial. This, of course is simply the polynomial

$$x^2 + 1687542416179693919 \tag{6.7}$$

A	$P_{x^2+A}(10^6)$	$L_{x^2+A}(10^6)$	P/L	$C(-A)$
1088144332169831	4212	28882.012	0.1458	0.14528180
2085366568194719	4007	28349.531	0.1413	0.14225381
34571718084771479	3619	26259.195	0.1378	0.13897783
121036243821187439	3532	25422.685	0.1389	0.13877611
140477714238295199	3524	25326.774	0.1391	0.13849255
391724890376888591	3387	24685.617	0.1372	0.13846496
427407728556837551	3337	24632.606	0.1355	0.13685911
695681268077667119	3227	24340.523	0.1326	0.13582738
1026219868955993351	3237	24112.365	0.1342	0.13529083
1687542416179693919	3286	23826.605	0.1379	0.13526493
1811049832613761871	3226	23786.573	0.1356	0.13508619
2503702890956049719	3263	23604.731	0.1382	0.13492130
2570304063059644799	3205	23590.112	0.1359	0.13450393
4995724305819922919	3127	23225.998	0.1346	0.13368059

Table 6.12: Computations to $x = 10^6$ for $C(-A)$-lochamps

translated by 216529, which is where the longest string of composite values assumed by (6.7) begins.

Using a similar format to that used by Lehmer, Lehmer, Shanks [LLS70, p. 444] we present an updated list of the least and least prime solutions to Problem IV in Table 6.13.

p	Least Solution N_p	Least Prime Solution N_p'	Source
3,5	7	7	Lehmer, Lehmer
7	127	127	and Shanks (1970)
11	247	463	DLS-127
13	463	463	
17	487	487	
19	1 423	1 423	
23	33 247	33 247	
29	56 743	73 327	
31	74 743	118 903	
37,41	118 903	118 903	
43	348 727	454 183	
47	348 727	773 767	
53,59,61	773 767	773 767	
67	2 430 943	86 976 583	
71	2 430 943	125 325 127	
73	2 430 943	132 690 343	
79	242 675 623	788 667 223	
83	393 292 183	788 667 223	
89	393 292 183	1 280 222 287	
97	393 292 183	2 430 076 903	
101	1 656 835 783	10 703 135 983	
103	2 713 676 023	10 703 135 983	
107	4 352 137 927	10 703 135 983	
109	8 133 814 327	10 703 135 983	
113,127	8 133 814 327	15 605 135 527	
131	8 363 603 623	148 202 808 007	
137	8 363 603 623	569 078 186 623	
139	1 128 864 945 583	3 506 439 768 967	
149,151	3 402 396 344 407	3 506 439 768 967	
157,163	3 402 396 344 407	† 18 730 616 733 343	
167	17 382 121 592 383	28 155 078 320 623	MSSU
173,179,181,191	36 557 608 400 527	36 557 608 400 527	
193	79 844 545 926 247	234 950 661 025 543	
197	79 844 545 926 247	1 338 417 926 295 487	
199	2 216 664 949 125 463	3 295 223 684 316 127	
211,223	5 504 869 618 441 447	10 594 288 233 662 863	
227	6 797 859 698 206 063	54 257 601 099 038 527	
229,233	57 658 533 878 198 983	57 658 533 878 198 983	
239	404 974 660 368 988 543	405 504 561 130 381 927	
241	590 811 814 251 783 727	>1 000 000 000 000 000 000	
251	757 792 008 436 904 527	>1 000 000 000 000 000 000	
257	>1 000 000 000 000 000 000	>1 000 000 000 000 000 000	

Table 6.13: Lehmer-Lehmer-Shanks Problem IV

† value computed using MSSU

6.4 Lehmer-Lehmer-Shanks Problem V

Lehmer, Lehmer, Shanks [LLS70] appear to have been the first to compute values for $N_p \equiv 5 \pmod 8$ with $(N_p/p_i) = -(-2/p_i)$ for all $p_i \leq p$. They remarked that the resulting list was quite short as many of the least solutions are valid for relatively long strings of p_i values. This is particularly noteworthy for the solutions N_5 through N_{23} which is a string of 7 primes, and the solutions N_{29} through N_{47} which has length 6. They also commented that they were unsure whether this phenomenon was of significance, or merely a fluke. Using the MSSU, we were able to extend these computations and correct one error present in the original table. The correct value for N_{157}, N_{163}, and N_{167} is 1068031603781. With this correction, we break a string of 6 values of p_i which was believed to be valid for the same least solution, instead creating two smaller sequences of repeated solutions of length 3. For least solutions which we were able to compute beyond the range given by Lehmer, Lehmer and Shanks, the tendency for least solutions to repeat does not appear to be as pronounced as in the cases of earlier values. The updated table of values is presented in Table 6.14.

p	Least Solution N_p	Least Prime Solution N_p'	Source
3	5	5	Lehmer, Lehmer
5,$\cdots$,23	29	29	and Shanks (1970)
29,$\cdots$,47	23 669	23 669	DLS-127
53	1 508 789	1 508 789	
59	5 025 869	5 025 869	
61,67	7 841 261	9 636 461	
71	9 636 461	9 636 461	
73	18 127 229	37 989 701	
79,83	31 839 341	37 989 701	
89,97	37 989 701	37 989 701	
101,103,107	240 511 301	240 511 301	
109	23 739 440 141	41 868 418 349	
113	44 913 466 781	90 664 613 309	
127,131,137	60 664 576 541	123 464 393 861	
139,149,151	123 464 393 861	123 464 393 861	
157,163,167	† 1 068 031 603 781	1 833 287 692 469	
173,179,181	1 833 287 692 469	1 833 287 692 469	
191	* 551 844 211 352 981	1 715 227 855 420 661	MSSU
193	551 844 211 352 981	1 826 528 575 689 869	
197	1 826 528 575 689 869	1 826 528 575 689 869	
199,211	3 560 698 230 077 381	3 560 698 230 077 381	
223	7 022 891 714 254 061	7 022 891 714 254 061	
227	7 251 154 827 581 981	17 216 381 212 527 389	
229,233	17 216 381 212 527 389	17 216 381 212 527 389	
239	68 761 250 344 570 109	224 548 122 529 827 461	
241	363 383 971 194 708 221	>400 000 000 000 000 000	

Table 6.14: Lehmer-Lehmer-Shanks Problem V

† value computed using MSSU

* repeat value for $p = 191, 193$

6.5 Examination of Littlewood's Bounds on $L(1,\chi)$

In work performed in close collaboration with D.H. and Emma Lehmer, Shanks [Sha73] conducted an extensive numerical examination of Littlewood's bounds on $L(1,\chi)$. In order to discuss this work, and present new experimental data generated using the MSSU, we shall briefly define some of the terms which will be used in this section and describe some earlier related work.

6.5.1 The Analytic Class Number Formula

Let D denote a square-free positive integer, and let $\mathcal{K} = \mathbb{Q}(\sqrt{D})$ be the quadratic field formed by adjoining $\sqrt{D}$ to the rationals $\mathbb{Q}$. If we define

$$r = \begin{cases} 2 & \text{when } D \equiv 1 \pmod 4 \\ 1 & \text{otherwise} \end{cases}$$

then $\Delta = (2/r)^2 D$ is the *discriminant* of $\mathcal{K}$. If

$$\omega = \left(\Delta + \sqrt{\Delta}\right)/2,$$

then

$$\mathcal{O} = \mathbb{Z} + \omega\mathbb{Z}$$

is the maximal order (the ring of algebraic integers) of $\mathcal{K}$.

If $\mathcal{O}^*$ is the group of units in $\mathcal{O}$ and $\Delta > 0$, then $\mathcal{O}^* = \langle -1, \varepsilon_o \rangle$, where $\varepsilon_o (> 1)$ is the *fundamental unit* of $\mathcal{K}$. Let $R = \log\ \varepsilon_o$ denote the *regulator* of $\mathcal{K}$. Since $\varepsilon_o \in \mathcal{O}$, we have $\varepsilon_o = \left(x + y\sqrt{\Delta}\right)/2$, where $x, y \in \mathbb{Z}$.

Let h denote the class number of $\mathcal{K}$. The above fundamental invariants of $\mathcal{K}$ are related by the analytic class number formula

$$2Rh = \sqrt{\Delta} L(1, \chi_\Delta), \tag{6.8}$$

where

$$L(1,\chi_\Delta) = \lim_{s\to 1} L(s,\chi_\Delta),$$

and the Dirichlet L-function is defined by

$$L(s,\chi_\Delta) = \sum_{n=1}^{\infty} \chi_\Delta(n) n^{-s} = \prod_p (1-\chi_\Delta(p)/p^s)^{-1}. \tag{6.9}$$

The character χ_Δ here is the Kronecker symbol (Δ/n); the Euler product on the right of (6.9) is taken over all the primes p. Thus, in order for R to be large, say, it is necessary for h to be small and $L(1,\chi_\Delta)$ to be large. How often h can be small and how large (and how often) $L(1,\chi_\Delta)$ can become are very deep and difficult questions in number theory. For example, the famous Gauss Conjecture asserts that $h = 1$ infinitely often and the Extended Riemann Hypothesis (ERH) provides us with quite close bounds on $L(1,\chi_\Delta)$.

6.5.2 The Search for Unusually Large Regulators

Define by $p(D)$ the length of the period of the continued fraction expansion of $\sqrt{D}$. In [Wil81], Williams conducted a numerical investigation which suggested that $p(D)$ is $O(\sqrt{D}\log\log D)$. Defining

$$f(D) = \begin{cases} \sqrt{D}\log\log D & \text{for } D \equiv 1 \pmod 8 \\ \sqrt{D}\log\log 4D & \text{otherwise,} \end{cases}$$

Williams showed that

$$G(D) = \frac{p(D)}{f(D)} < K_1 + o(1),$$

where

$$K_1 \approx 3.701223297,$$

provided the ERH is true.

Using $G(D)$ as a relative measure of the period length, the numerical data provided in [Wil81] suggests that in order to find values of D for which $G(D)$ is large, it is desirable

to consider values of D of the form q or $2q$ where q is a prime for which $q \equiv -1 \pmod 4$, such that the Legendre symbol $(D/p_i) = 1$ for as many of the small primes p_i as possible.

Patterson and Williams [PW85] defined the following three D-value types

(i.) $D \equiv 3 \pmod 8$, D prime,

(ii.) $D \equiv 7 \pmod 8$, D prime,

(iii.) $D \equiv 6 \pmod 8$, $D/2$ prime.

Using the UMSU sieve they provided tables of $G(D)$ values for which $G(D)$ exceeds $G(d)$ for all values of $d < D$ for each of these three types. These computations were later extended for each of these three types of D-values by Stephens [Ste89] using the OASiS sieve for all $D \leq 5 \times 10^{15}$.

Since $R(D)$, the regulator of $\mathbb{Q}(\sqrt{D})$, is expected by Lévy's Law (see [Wil81]) to be proportional to $p(D)$, a large $G(D)$ value is a likely indicator of large $R(D)$. Also, from the analytic class number formula (6.8), we see that for R to be large we desire that the associated Dirichlet L-function, $L(1,\chi)$, be large and that h, the class number, be as small as possible; ideally the class number should be 1. Cohen and Lenstra [CL84, CL83] have provided heuristics which suggest that for prime *radicand* values D, $h = 1$ approximately 75% of the time. Thus we would expect that to find unusually large regulator values, we should conduct a search for the same three types of D-values as was used to find long period lengths $p(D)$.

Instead of conducting a search for unusually large values of $p(D)$ or $R(D)$, we decided to concentrate our energies on the related problem of finding extremal values of the Dirichlet L-function $L(1,\chi)$. Because earlier work in this area was performed by Shanks [Sha73] using the comparatively slow DLS-157 sieve, our hope was that the MSSU could provide valuable new experimental data on the size of $L(1,\chi)$.

6.5.3 Littlewood's Bounds on $L(1,\chi)$

Littlewood [Lit28] and Shanks [Sha73] have shown that under the ERH we have

$$\{1+o(1)\}\,(c_1 \log\log \Delta)^{-1} < L\,(1,\chi_\Delta) < \{1+o(1)\}\,c_2 \log\log \Delta, \tag{6.10}$$

where the values of the constants c_1 and c_2 depend upon the parity of Δ:

$$c_1 = \begin{cases} 12e^\gamma/\pi^2 & \text{when } 2\nmid\Delta, \\ 8e^\gamma/\pi^2 & \text{when } 2|\Delta, \end{cases}$$

$$c_2 = \begin{cases} 2e^\gamma & \text{when } 2\nmid\Delta \\ e^\gamma & \text{when } 2|\Delta. \end{cases}$$

In [Sha73], Shanks performed a numerical examination of (6.10) to study the possibility of approaching these bounds, or even surpassing them. He noted that Littlewood gave no description of the $o(1)$ term in (6.10), giving neither its sign, nor the manner in which it approaches zero as a function of Δ. Ignoring the $o(1)$ term, Shanks defined for a fixed Δ the *upper* and *lower Littlewood indices* as

$$ULI = L\,(1,\chi_\Delta)\,/\,(c_2 \log\log \Delta) \tag{6.11}$$

and

$$LLI = L\,(1,\chi_\Delta)\,c_1 \log\log \Delta \tag{6.12}$$

If (6.10) is true, then as Δ increases, we would expect that extreme values of the ULI and LLI would tend to approach 1.

In fact, Chowla [Cho49] has shown that for any positive $\epsilon < 1$,

$$ULI > \frac{1-\epsilon}{2} \quad \text{and} \quad LLI < 2(1-\epsilon)$$

hold, each for an infinite sequence of values of Δ. Furthermore, Joshi [Jos70] showed that if c and d are relatively prime positive integers and $8|d$, then as Δ runs through prime values congruent to $c \pmod d$, we have

$$ULI > \frac{1-\epsilon}{2} \prod_{p|d} \frac{1-1/p}{1-\left(\frac{p}{c}\right)1/p}$$

and

$$LLI < 2(1-\epsilon)\prod_{p|d}\frac{1-1/p}{1-\left(\frac{p}{c}\right)1/p}$$

infinitely often. Thus, if Δ is a prime and $\Delta \equiv 5 \pmod 8$, we would have

$$LLI < (4/3)(1-\epsilon)$$

infinitely often. Also, if Δ is a prime and $\Delta \equiv 1 \pmod 8$, we would have

$$ULI > (1/2)(1-\epsilon)$$

infinitely often.

Shanks tested (6.10) by attempting to produce values of Δ for which he might find locally extreme values for the LLI and ULI. For example, if $\Delta \equiv 5 \pmod 8$ and $(\Delta/q) = -1$ for all of the small primes q less than some bound p, then we would expect by (6.9) that $L(1,\chi_\Delta)$ would be small. On the other hand, if $\Delta/4 \equiv 7 \pmod 8$ and $(\Delta/q) = 1$ for all the primes $q \leq p$, then we would expect $L(1,\chi_\Delta)$ to be large.

Shanks made use of Lehmer's numerical sieving device, the DLS-157, to find such special values of Δ. He found no ULI larger than 1; in fact, the largest ULI that he found was .7333. Also, he found only a few LLI's less than 1, however, these only occurred for small values of Δ where we would expect the affect of the $o(1)$ term to be most significant. The values of the LLI's tended to remain stable on average, or change very slowly; whereas the ULI's tended to increase very slowly for these special Δ values; thus, these numerical trials lent support to (6.10).

Numerical Experiments

For both positive and negative discriminants $|\Delta| \leq 50000$, Shanks was able to determine those values of Δ having smaller values of $L(1,\chi_\Delta)$ than all preceding values of Δ, which he termed *lochamps*. Similarly, Shanks found those values of Δ having values of $L(1,\chi_\Delta)$ larger than any preceding values of Δ, which he termed *hichamps*. For larger

discriminants, it was not feasible to compute $L(1,\chi_\Delta)$ for each possible discriminant, therefore, Shanks identified *tentative* hichamps and lochamps as the $L(1,\chi_\Delta)$ values of several types of quadratic characters which were likely to produce locally extreme values. For our investigation we constructed lists of ULI hichamps for those types of discriminants most likely to have large values of $L(1,\chi_\Delta)$ and LLI lochamps for those most likely to have small values of $L(1,\chi_\Delta)$.

With MSSU, we were able to conduct this search for the kind of numbers that we sought at the rate of over 4×10^{12} per second, a considerably faster search rate than that of the DLS-157. For $D\equiv 5 \pmod 8$ and $(D/q)=-1$, for $q=3,5,7,\ldots,199$, we found all values of D such that $0<D<10^{19}$. For $D\equiv 6 \pmod 8$ and $D\equiv -1 \pmod 4$ we found all the values of D such that $0<D<10^{19}$ and $(D/q)=1$ for $q=3,5,7,\ldots,199$. For $D\equiv 1 \pmod 8$ and $(D/q)=1$ for $q=3,5,7,\ldots,199$, we re-used the list of values generated in the pseudosquare search described in Section 6.1, providing us with a list of D values for $0<D<4\times 10^{19}$ For the several thousand numbers that resulted, the class number, regulator and $L(1,\chi_\Delta)$ values were computed by using the Shanks heuristic [MW92, p. 283]. These values were computed by Michael Jacobson Jr. (see [Jac95]). For all of these values of D, both prime and non-prime, we evaluated the ULI's or LLI's. The largest ULI we found is $ULI = .741429825\ldots$ $(L(1,\chi_\Delta)=4.98741315\ldots, h=2)$ for

$$D = 2323617473234474719.$$

The least LLI we found is $LLI = 1.24745080\ldots$ $(L(1,\chi_\Delta)=0.158960540\ldots)$, $h=4$ for

$$D = 18974003020179917$$

Following Shanks, we define the symbols aR_p (aN_p) to represent the least integers congruent to a modulo 8 such that

$$\left(\frac{aR_p}{q}\right)=1, \qquad \left(\frac{aN_p}{q}\right)=-1$$

D	R	h	$L(1,\chi)$	LLI
370095509388197	794079.6472497177	2	0.165107631	1.25601176
1897400302017991 7	2737025.3979827850	4	0.158960540	1.24745080

Table 6.15: $D \equiv 5 \pmod 8$ — LLI-lochamps

D	R	h	$L(1,\chi)$	ULI
6450045516630769	90756597.6965676137	4	9.04038245	0.706023128
11641399247947921	491322389.3502864803	1	9.10739051	0.708086652
30819994040589121	807859472.4921805586	1	9.20342636	0.710441965
116307279575913409	1603698785.2191524967	1	9.40479529	0.719187024
309361186961076121	659373276.4916228329	4	9.48393166	0.720412930
2305984263805598401	7377563911.9100995120	1	9.71661078	0.728492958
6476334166896360649	1245808525.9986525809	10	9.79076596	0.729352585
11917728321713151001	4247532486.6139021970	4	9.84305599	0.730537438

Table 6.16: $D \equiv 1 \pmod 8$ — ULI-hichamps

for all odd primes $q \leq p$. We provide tables (6.20 through 6.29) of aR_p for $a = 1$, $a = 6$, $a = 3$ and 7, and aN_p for $a = 5$ and also similar tables of aR_p and aN_p when we added the extra constraint that aR_p and aN_p be prime, together with the corresponding ULI and LLI values. We have also included the data for $-7R_p$ which we gathered during our search for negative pseudosquare values described in Section 6.1.3. Notice that the tendency is for the ULI's to very slowly increase and for the LLI's to remain stable with minor fluctuations about 4/3. Thus, the results that we have obtained completely support Shanks' earlier findings and therefore support the truth of (6.10). At the very least, we have not found anything that would lead us to believe that the ERH has been violated.

Although such values of D surely must exist, it seems to be very difficult to produce a value of D with a ULI close to 1. We attempted to do this by finding a D value with a large $L(1,\chi_\Delta)$ value. We modified slightly an unpublished idea of Lehmer which he employed to find the 20 digit value of D with a small $L(1,\chi_\Delta)$ value that appears in

D	R	h	$L(1,\chi)$	ULI
5851505127988699	177728821.6530873557	2	4.64680087	0.718840689
17062309974303619	614369707.8820124509	1	4.70338879	0.722024977
19369243730308651	66039864.2223467303	10	4.74514884	0.727785680
20602619993714851	170879672.4976781991	4	4.76199755	0.730053523
134428259280597811	442693041.8869178770	4	4.82966935	0.731040320
370695282126782419	1483547793.6632413161	2	4.87329843	0.732795816
686289874744590691	4066647990.4097152107	1	4.90888418	0.735269175
4217347918605865891	2555341969.2165341982	4	4.97724826	0.737272524

Table 6.17: $D \equiv 3 \pmod 8$ — ULI-hichamps

D	R	h	$L(1,\chi)$	ULI
8265289127640406	410994766.1996174020	1	4.52071643	0.697582853
9442925344429966	457439232.0709417274	1	4.70738946	0.725690793
164913261020266126	1958556197.5128538633	1	4.82289989	0.729034169
382903329914535886	188017733.9645143813	16	4.86154405	0.730876784
574911115184562766	3718539673.4666251606	1	4.90424396	0.735394142
2249618179698381886	744815083.7612317213	10	4.96585525	0.738369948

Table 6.18: $D \equiv 6 \pmod 8$ — ULI-hichamps

Lehmer, Lehmer, Shanks [LLS70, p. 439]. We examined numbers of the form

$$D = A + BX$$

where

$$B = \prod_{i=j}^{k} p_i,$$

p_i is the i^{th} prime, and $(A/p_i) = 1$ $(i = j, j+1, \ldots, k)$. In our case we used

$$B = 271 \cdot 277 \cdots 313 \approx 5.277 \times 10^{19}$$

and the least nonsquare value of A. We then employed the MSSU to sieve on values of X by using as moduli 8 and primes $p_1, p_2, \ldots, p_m$ with $p_m \leq 269$ such that $A + XB \equiv 6 \pmod 8$ and $((A + XB)/p_i) = 1$, $i = 1, 2, \ldots, m$. Henri Cohen used the technique of

D	R	h	$L(1,\chi)$	ULI
2871159201832639	246120736.6299490883	1	4.59324455	0.714308970
11343192346627639	126443772.9651967759	4	4.74886462	0.731122630
1250929971496804351	2741367017.0974877715	2	4.90208322	0.731510386
1548668327679299479	3055545592.8796488176	2	4.91065649	0.731827607
2323617473234474719	3801260801.6072624237	2	4.98741315	0.741429825

Table 6.19: $D \equiv 7 \pmod 8$ — ULI-hichamps

[CDyDO93] to evaluate the $L(1,\chi_\Delta)$ values for some of these D values. The largest ULI occurred for

$$D = 1320870879580760303352202626252612243246,$$

where $L(1,\chi_\Delta) = 5.324999338\ldots (h = 1)$. This is a large $L(1,\chi_\Delta)$, but when we evaluate the ULI we only get

$$ULI = .669706597,$$

a ULI value which is less than many others that we computed previously.

p	N_p	$R(N_p)$	$h(N_p)$	$L(1,\chi)$	LLI
3	5	0.4812118251	1	0.4304089410	0.4435538722
5	53	1.9657204716	1	0.5400249451	1.6124663300
7,11	173	2.5708146781	1	0.3909108412	1.3879957859
13	293	2.8366557290	1	0.3314383943	1.2466975854
17	437	3.0422471121	1	0.2910608308	1.1376863198
19,23	9173	12.4722335398	1	0.2604465046	1.2469607005
29	24653	5.0562863725	4	0.2576242629	1.2908426649
31,37,41	74093	7.2159755425	5	0.2650980070	1.3875884595
43	170957	16.9391811055	3	0.2458103797	1.3249131722
47,53,59	214037	28.9536703424	2	0.2503336255	1.3593113661
61	2004917	48.2972218460	3	0.2046563391	1.1854909180
67	44401013	352.5078318367	2	0.2116081169	1.3144216275
71	71148173	140.5395212093	6	0.1999391491	1.2533798811
73,79	154554077	694.9131538316	2	0.2235889265	1.4219776752
83,89,97	163520117	152.1367911189	9	0.2141515205	1.3633426429
101,103	261153653	512.3272362372	3	0.1902175756	1.2210459376
107,109,113	1728061733	4021.1400400380	1	0.1934637347	1.2808614165
127	9447241877	1252.3775392061	7	0.1803893063	1.2243143804
131	19553206613	6209.5055811147	2	0.1776266063	1.2175566563
137,139	49107823133	18804.6808640657	1	0.1697151536	1.1773388854
149,...,163	385995595277	27068.0628101250	2	0.1742713488	1.2392997833
167	13213747959653	330785.2663571069	1	0.1819965450	1.3432519058
173	14506773263237	331149.0061978068	1	0.1738873705	1.2845627585
179,181	57824199003317	165998.4596144430	4	0.1746381943	1.3069830652
191,193	160909740894437	275610.2629809000	4	0.1738177824	1.3128077070
197,199	370095509388197	794079.6472497177	2	0.1651076319	1.2560117612
211	1409029796180597	3130386.6897110315	1	0.1667891884	1.2829190333
223	4075316253649373	5291574.7242182097	1	0.1657808151	1.2859302600
227,229,233	18974003020179917	2737025.3979827850	4	0.1589605408	1.2474508061
239,241	224117990614052477	10257518.4583975435	4	0.1733380610	1.3842259724
251,257,263	637754768063384837	22908547.7970583540	3	0.1721162807	1.3841002940
269,...,283	4472988326827347533	14462868.4419286840	12	0.1641219979	1.3363105513

Table 6.20: $5N_p$ - Least Solutions

p	N_p	$R(N_p)$	$h(N_p)$	$L(1,\chi)$	LLI
3	5	0.4812118251	1	0.4304089410	0.4435538722
5	53	1.9657204716	1	0.5400249451	1.6124663300
7,11	173	2.5708146781	1	0.3909108412	1.3879957859
13	293	2.8366557290	1	0.3314383943	1.2466975854
17	2477	6.4723486834	1	0.2600931403	1.1580285644
19,23	9173	12.4722335398	1	0.2604465046	1.2469607005
29	61613	36.2337073341	1	0.2919487298	1.5176456292
31,37,41	74093	7.2159755425	5	0.2650980070	1.3875884595
43	170957	16.9391811055	3	0.2458103797	1.3249131722
47	360293	68.2369106192	1	0.2273638631	1.2550449432
53	679733	92.0434915506	1	0.2232820974	1.2559265869
59,61	2004917	48.2972218460	3	0.2046563391	1.1854909180
67	69009533	869.6964342466	1	0.2093838307	1.3118207226
71	138473837	1369.2976968388	1	0.2327255204	1.4771395282
73	237536213	1725.6409680824	1	0.2239316727	1.4350861306
79	384479933	2087.3575497300	1	0.2129070310	1.3758049972
83	883597853	3018.2647139931	1	0.2030766526	1.3304141174
89, ...,113	1728061733	4021.1400400380	1	0.1934637347	1.2808614165
127	9447241877	1252.3775392061	7	0.1803893063	1.2243143804
131,137,139	49107823133	18804.6808640657	1	0.1697151536	1.1773388854
149	1843103135837	119080.8535909752	1	0.1754272744	1.2691535621
151,157	4316096218013	192239.8325709201	1	0.1850665020	1.3507860483
163,167	15021875771117	344898.8085837201	1	0.1779752480	1.3152047235
173,179	82409880589277	804942.5146297817	1	0.1773393937	1.3314686188
181	326813126363093	1551603.4111066732	1	0.1716567855	1.3044518825
191,193	390894884910197	1650908.4884553084	1	0.1670025115	1.2710155484
197	1051212848890277	547589.0434995716	5	0.1688921326	1.2960099142
199,211,223	4075316253649373	5291574.7242182097	1	0.1657808151	1.2859302600
227	274457237558283317	45653225.9568753417	1	0.1742866502	1.3937104898
229	443001676907312837	6097479.6722440887	9	0.1648998198	1.3228799251
233	599423482887195557	65388978.2285433153	1	0.1689147872	1.3578014995
239	614530964726833997	64783176.9720669673	1	0.1652800183	1.3288014118
241,...,263	637754768063384837	22908547.7970583540	3	0.1721162807	1.3841002940

Table 6.21: $5N_p$ - Least Prime Solutions

p	R_p	$h(-R_p)$	$L(1,\chi)$	ULI
3	23	3	1.96520	0.482759
5	71	7	2.60986	0.505323
7	311	19	3.38472	0.543767
11	479	25	3.58857	0.553535
13	1559	51	4.05785	0.571024
17	5711	109	4.53127	0.589579
19	10559	153	4.67767	0.589863
23	18191	213	4.96136	0.610004
29	31391	289	5.12442	0.615461
31	118271	606	5.53583	0.632266
37,41	366791	1121	5.81495	0.640061
43,47,53	2155919	2968	6.35034	0.665221
59,61	6077111	5092	6.48917	0.662786
67	98538359	21934	6.94168	0.669054
71	120293879	24503	7.01854	0.673968
73,79	131486759	25817	7.07317	0.678107
83	508095719	51460	7.17210	0.671580
89,...,109	2570169839	122106	7.56669	0.690615
113,127	328878692999	1499699	8.21553	0.703613
131	513928659191	1870227	8.19582	0.698368
137	844276851239	2449750	8.37585	0.709779
139	1043702750999	2722860	8.37310	0.707902
149	4306732833311	5610919	8.49395	0.707476
151	8402847753431	7881499	8.54172	0.706692
157	47375970146951	19213042	8.76934	0.713643
163	52717232543951	20080160	8.68841	0.706364
167	100535431791791	27781594	8.70457	0.703556
173	251109340045079	44300560	8.78269	0.704194
179,181,191	493092541684679	63175230	8.93783	0.712534
193,...,211	1088144332169831	95504500	9.09558	0.720378
223	71608584429428591	770037780	9.04022	0.693655
227	88163809868323439	883621890	9.34912	0.716310
229	218748706425968039	1389387790	9.33255	0.710576
233	423414931359807911	1972365462	9.52257	0.721831
239	695681268077667119	2582308636	9.72641	0.734870
241,251	1116971853972029831	3235883052	9.61881	0.724506
257,263,269	3546374752298322551	5849622617	9.75855	0.729663

Table 6.22: $-7R_p$ - Least Solutions (Problem II)

p	R_p	$h(-R_p)$	$L(1,\chi)$	ULI
3	23	3	1.96520	0.482759
5	71	7	2.60986	0.505323
7	311	19	3.38472	0.543767
11	479	25	3.58857	0.553535
13	1559	51	4.05785	0.571024
17	5711	109	4.53127	0.589579
19	10559	153	4.67767	0.589863
23	18191	213	4.96136	0.610004
29	31391	289	5.12442	0.615461
31,37,41	366791	1121	5.81495	0.640061
43	4080359	3997	6.21634	0.640943
47	12537719	7457	6.61614	0.664792
53	30706079	12017	6.81292	0.671737
59	36415991	12719	6.62150	0.650614
61	82636319	20299	7.01518	0.678376
67,71	120293879	24503	7.01854	0.673968
73,79	131486759	25817	7.07317	0.678107
83,89	2929911599	128755	7.47285	0.680716
97	7979490791	219207	7.70933	0.692165
101,103	33857579279	456929	7.80136	0.686925
107	89206899239	761619	8.01103	0.696823
109	121560956039	883537	7.96118	0.689880
113,127	328878692999	1499699	8.21553	0.703613
131	513928659191	1870227	8.19582	0.698368
137,139,149	4306732833311	5610919	8.49395	0.707476
151	8402847753431	7881499	8.54172	0.706692
157	70864718555231	23298345	8.69480	0.704981
163	317398900373231	48337375	8.52374	0.682060
167,...,181	501108392233679	63401301	8.89779	0.709245
191,193,197	5551185799073591	217709085	9.17980	0.717736
199,211	7832488789769159	255145331	9.05706	0.706280
223,227	102097158739597271	957852561	9.41763	0.720820
229	315759454565514431	1702318729	9.51727	0.722846
233	868116409360316399	2849113533	9.60662	0.724771
239	3412527725201978759	5672915337	9.64757	0.721538
241,...,269	3546374752298322551	5849622617	9.75855	0.729663

Table 6.23: $-7R_p$ - Least Prime Solutions (Problem II)

p	R_p	$R(R_p)$	$h(R_p)$	$L(1,\chi)$	ULI
3	73	7.6666904193	1	1.7946364837	0.3459282982
5	241	18.7714934907	1	2.4183563839	0.3988910181
7	1009	6.9847171775	7	3.0784441441	0.4468651590
11	2641	90.1229885158	1	3.5073683588	0.4770019440
13	8089	178.2283912928	1	3.9633229771	0.5064209380
17	18001	282.9788464803	1	4.2182824066	0.5188848171
19	53881	541.5581275917	1	4.6661344671	0.5484838791
23	87481	711.1635824938	1	4.8088655542	0.5551442792
29	117049	846.0999760519	1	4.9461561691	0.5651217648
31	515761	1907.4520693120	1	5.3120160651	0.5787450482
37	1083289	2874.3057076656	1	5.5232002116	0.5892030098
41	3206641	1695.1784696826	3	5.6799106982	0.5890869201
43	3818929	5713.8264256957	1	5.8477156539	0.6039032227
47	9257329	9230.3898997539	1	6.0674637842	0.6137764192
53	22000801	2086.0812795921	7	6.2264429761	0.6181510043
59,61	48473881	22267.3855259820	1	6.3965430604	0.6249458203
67	175244281	8755.3143391106	5	6.6137810900	0.6307799850
71,73	427733329	70323.9694075417	1	6.8005936576	0.6386300530
79	898716289	9537.4088853210	11	6.9990931932	0.6493042832
83,89,97	2805544681	19261.8308091777	10	7.2730900489	0.6629482221
101	10310263441	380018.3647427944	1	7.4851360649	0.6696424663
103	23616331489	30603.4417531708	19	7.5674167131	0.6694690869
107,109	85157610409	571883.7911846249	2	7.8389166334	0.6822425198
113,127	196265095009	1724813.5179962550	1	7.7866491811	0.6709050799
131,137,139	2871842842801	7023729.3598930115	1	8.2892918290	0.6933157491
149,151	26250887023729	5468833.1146211778	4	8.5391111438	0.6987414036
157,163,167	112434732901969	45498659.9770372017	1	8.5817993364	0.6929425011
173,179	178936222537081	905318.6655122233	65	8.7982377693	0.7075170717
181,191	696161110209049	29450368.8819946960	4	8.9294733746	0.7098228696
193	2854909648103881	121507633.5651132951	2	9.0963543491	0.7148977642
197,199	6450045516630769	90756597.6965676137	4	9.0403824576	0.7060231286
211,223	11641399247947921	491322389.3502864803	1	9.1073905127	0.7080866524
227	190621428905186449	1011534665.9919609451	2	9.2673358433	0.7062716759
229	196640248121928601	2074591515.4625055871	1	9.3567793106	0.7129370864
233	712624335095093521	263678486.7019505920	15	9.3705617701	0.7078731164
239	1773855791877850321	506310756.6777648463	12	9.1236631113	0.6851762605
241	2327687064124474441	7232768052.8702567520	1	9.4813949390	0.7108158377
251	6384991873059836689	2004168448.1338770521	6	9.5177718982	0.7090778996
257	8019204661305419761	6728169524.1991089359	2	9.5036682582	0.7070422414
263,269,271	10198100582046287689	3783707520.3950194471	4	9.4786808275	0.7041556389

Table 6.24: $1R_p$ - Least Solutions (Problem I)

p	R_p	$R(R_p)$	$h(R_p)$	$L(1,\chi)$	ULI
3	73	7.6666904193	1	1.7946364837	0.3459282982
5	241	18.7714934907	1	2.4183563839	0.3988910181
7	1009	6.9847171775	7	3.0784441441	0.4468651590
11	2689	90.3456893558	1	3.4845125300	0.4733698982
13	8089	178.2283912928	1	3.9633229771	0.5064209380
17	33049	395.0737146868	1	4.3463927738	0.5209116675
19	53881	541.5581275917	1	4.6661344671	0.5484838791
23	87481	711.1635824938	1	4.8088655542	0.5551442792
29	483289	1817.6595926533	1	5.2292462452	0.5708252190
31	515761	1907.4520693120	1	5.3120160651	0.5787450482
37	1083289	2874.3057076656	1	5.5232002116	0.5892030098
41,43	3818929	5713.8264256957	1	5.8477156539	0.6039032227
47	9257329	9230.3898997539	1	6.0674637842	0.6137764192
53	22000801	2086.0812795921	7	6.2264429761	0.6181510043
59,61	48473881	22267.3855259820	1	6.3965430604	0.6249458203
67	175244281	8755.3143391106	5	6.6137810900	0.6307799850
71,73	427733329	70323.9694075417	1	6.8005936576	0.6386300530
79	898716289	9537.4088853210	11	6.9990931932	0.6493042832
83	8114538721	331798.4694693130	1	7.3666910346	0.6612462398
89	9176747449	351603.8308992703	1	7.3407292144	0.6577849731
97,101,103	23616331489	30603.4417531708	19	7.5674167131	0.6694690869
107,109,113,127	196265095009	1724813.5179962550	1	7.7866491811	0.6709050799
131,137,139	2871842842801	7023729.3598930115	1	8.2892918290	0.6933157491
149	26437680473689	21737796.4313188107	1	8.4553937843	0.6918446788
151	89436364375801	13405886.4246995330	3	8.5053015875	0.6881706881
157,163,167	112434732901969	45498659.9770372017	1	8.5817993364	0.6929425011
173,179	178936222537081	905318.6655122233	65	8.7982377693	0.7075170717
181,191,193	6072205049848081	343020804.2126533863	1	8.8039434062	0.6878756324
197,199,211,223	11641399247947921	491322389.3502864803	1	9.1073905127	0.7080866524
227,229	196640248121928601	2074591515.4625055871	1	9.3567793106	0.7129370864
233	781158046093912369	830497955.1883735070	5	9.3965647828	0.7094115069
239	6938117179828687609	4215890391.9340709644	3	9.6032834211	0.7150847819
241	9064125655411231729	14430633177.9254913048	1	9.5863310826	0.7126613881
251	15559176909429792409	18941964091.6491117336	1	9.6042110555	0.7116720874
257	18539153100230615161	20896666758.0590150037	1	9.7064867761	0.7185001674

Table 6.25: $1R_p$ - Least Prime Solutions (Problem I)

p	R_p	$R(R_p)$	$h(R_p)$	$L(1,\chi)$	ULI
3	22	5.9763444674	1	1.2741609216	0.4772356074
5,7	46	10.7928181024	1	1.5913142134	0.5409893881
11	214	27.9608415496	1	1.9113637791	0.5618960324
13	1054	36.4425470915	2	2.2450106929	0.5940462388
17	4174	153.0173430331	1	2.3684511642	0.5846547786
19	5014	86.8113705157	2	2.4519659982	0.6003392697
23	9454	125.4203677627	2	2.5798251163	0.6150019700
29	34654	508.5862719586	1	2.7320456249	0.6206640694
31	166846	1188.8221393565	1	2.9104422394	0.6294462049
37,41	189814	1283.9621566596	1	2.9470540629	0.6350231729
43	2185726	4732.0598097346	1	3.2007545089	0.6484022341
47,53	2237134	4898.7805440843	1	3.2752313716	0.6631418565
59	12020446	5852.6129297062	2	3.3761326908	0.6598054998
61	30628966	18845.3636087480	1	3.4051665099	0.6537540622
67	45735286	23224.6896756745	1	3.4341892160	0.6545588488
71	103345246	36324.1970214383	1	3.5731460735	0.6714799796
73	193438606	12646.2805382172	4	3.6370663694	0.6764488578
79	302673526	15594.7382233309	4	3.5855093740	0.6621139478
83	1399951606	71257.6997494259	2	3.8089500990	0.6873976022
89	1493483566	74632.7843077108	2	3.8624192745	0.6964030995
97	8813799094	370333.7974090586	1	3.9446799718	0.6943106082
101	8932573654	373945.6035732992	1	3.9565816157	0.6962851607
103	11391294814	208190.3399218793	2	3.9012521771	0.6844123343
107,109,113	16692514294	65489.3575575022	8	4.0550838481	0.7079819674
127,131	490184082166	1438149.3744667363	2	4.1082266003	0.6899767584
137,139	771038637814	938914.0876846836	4	4.2770839621	0.7149410094
149,151	1052385901774	443737.1880325219	10	4.3255199181	0.7207287357
157,163	27266351212006	1448975.3984422674	16	4.4398416630	0.7171894017
167	46075643128414	7562458.7664037368	4	4.4564322467	0.7165501371
173	101596847251054	11240507.7537534517	4	4.4607286474	0.7123880112
179	111010920394126	5975308.1009565997	8	4.5369859957	0.7240246579
181,191	186677562227374	10370523.2619880009	6	4.5541345752	0.7236159062
193	2769113411231974	9937273.1452172189	24	4.5321898896	0.7050053676
197	7796793440819254	15101865.9096864397	27	4.6178131873	0.7128664511
199	8265289127640406	410994766.1996174020	1	4.5207164332	0.6975828539
211	9442925344429966	457439232.0709417274	1	4.7073894627	0.7256907935
223	27130689396477286	127224468.6626642538	6	4.6343782946	0.7091249540
227	62414818040620774	1187337674.0593696079	1	4.7525904803	0.7230714826
229	77991421972487566	330118687.3920430143	4	4.7283180497	0.7182990644
233	230816476295404294	1145918413.8038441345	2	4.7703505070	0.7195099422
239,241	5749111115184562766	3718539673.4666251606	1	4.9042439697	0.7353941426
251,..., 269	1441979855850505414	1410649668.1724776074	4	4.6989363866	0.7005822902

Table 6.26: $6R_p$ - Least Solutions

p	R_p	$R(R_p)$	$h(R_p)$	$L(1,\chi)$	ULI
3	22	5.9763444674	1	1.2741609216	0.4772356074
5,7	46	10.7928181024	1	1.5913142134	0.5409893881
11	214	27.9608415496	1	1.9113637791	0.5618960324
13	1486	17.4517660960	5	2.2636015528	0.5877935245
17	4174	153.0173430331	1	2.3684511642	0.5846547786
19	10774	265.9615141646	1	2.5623019742	0.6076452315
23	13126	298.6426033189	1	2.6066670535	0.6134158918
29	34654	508.5862719586	1	2.7320456249	0.6206640694
31	166846	1188.8221393565	1	2.9104422394	0.6294462049
37,41	189814	1283.9621566596	1	2.9470540629	0.6350231729
43	2185726	4732.0598097346	1	3.2007545089	0.6484022341
47,53	2237134	4898.7805440843	1	3.2752313716	0.6631418565
59	25056574	16886.1067717626	1	3.3734065692	0.6500663311
61	30628966	18845.3636087480	1	3.4051665099	0.6537540622
67	45735286	23224.6896756745	1	3.4341892160	0.6545588488
71	103345246	36324.1970214383	1	3.5731460735	0.6714799796
73	548033014	87004.9198478119	1	3.7165540272	0.6800524999
79,83,89	1998450094	55854.7212378159	3	3.7483012667	0.6730487468
97	8813799094	370333.7974090586	1	3.9446799718	0.6943106082
101	8932573654	373945.6035732992	1	3.9565816157	0.6962851607
103,107,109	26026453534	647111.4921003566	1	4.0111750552	0.6964904169
113	31416841054	723672.4155659130	1	4.0828249625	0.7073144760
127	762216630646	3635072.1570969908	1	4.1636462684	0.6960621408
131	1058804919286	4298691.6955030604	1	4.1776141281	0.6960411031
137	1321651385014	4795026.1808554558	1	4.1709263874	0.6933635780
139,149	3499659258286	2635944.0116697690	3	4.2271197912	0.6959670315
151	25070229597526	22138339.9330482741	1	4.4214619921	0.7147553709
157	48888369417694	30359479.4996157916	1	4.3420172580	0.6977931789
163	129143129979406	50279000.9115376854	1	4.4243653791	0.7051552920
167	184022456828926	12173479.5622951690	5	4.4869302976	0.7130221009
173	256397742215806	71809801.1109845836	1	4.4846300804	0.7107169880
179	600206879107606	112267896.3002947161	1	4.5825277176	0.7212823926
181	2457721162815286	224676759.2628898814	1	4.5320202993	0.7056112649
191	4142956695812806	289298680.3684517510	1	4.4946021551	0.6970637106
193,197	7796793440819254	15101865.9096864397	27	4.6178131873	0.7128664511
199	8265289127640406	410994766.1996174020	1	4.5207164332	0.6975828539
211	9442925344429966	457439232.0709417274	1	4.7073894627	0.7256907935
223,227	62414818040620774	1187337674.0593696079	1	4.7525904803	0.7230714826
229	150726700798733614	1854144221.5935088719	1	4.7758251191	0.7223451197
233,239,241	574911115184562766	3718539673.4666251606	1	4.9042439697	0.7353941426
251	2380303014240673006	7394016196.7648550161	1	4.7925237558	0.7123534720
257,263	4693358374126530406	10456361930.7709409468	1	4.8265715502	0.7144982071

Table 6.27: $6R_p$ - Least $2\times$ Prime Solutions

p	R_p	$R(R_p)$	$h(R_p)$	$L(1,\chi)$	ULI
3	7	2.7686593833	1	1.0464548848	0.4881400578
5	19	5.8289369670	1	1.3372498520	0.5122417374
7	79	5.0751347504	3	1.7129918110	0.5495238232
11	331	36.2563832043	1	1.9928310511	0.5672551030
13	751	57.9421480629	1	2.1143390118	0.5706179805
17	1171	25.3728064560	3	2.2243919456	0.5851342427
19	5251	89.6157067802	2	2.4733929630	0.6043596044
23	10651	270.8720689096	1	2.6246357047	0.6227107192
29	18379	367.1977320417	1	2.7085636844	0.6293491820
31,37	78439	813.5634679127	1	2.9048613936	0.6425763221
41	399499	1890.8635586713	1	2.9915918664	0.6316501081
43	1234531	3537.8678099198	1	3.1841286161	0.6536169887
47	1394611	1901.1078275627	2	3.2196614114	0.6590124154
53	1427911	3841.3976895553	1	3.2146870537	0.6576303475
59	4355311	6958.9983656651	1	3.3345493442	0.6653680285
61	5715319	8109.8013153157	1	3.3922666024	0.6730177892
67	12807391	6078.5198036139	2	3.3970152569	0.6630606931
71	117678031	38495.7079874423	1	3.5486618262	0.6654255555
73,79	133826599	10617.1445316298	4	3.6711024130	0.6869124037
83	452980999	78083.7491911782	1	3.6687715128	0.6732611819
89,97	505313251	83941.6234166887	1	3.7341952366	0.6841233484
101	5273095699	144314.3922955479	2	3.9747225064	0.7043189046
103,107,109	9248561191	127289.8015022314	3	3.9707978385	0.6984739720
113	38816991931	36197.2005113960	22	4.0419108071	0.6984501801
127	257057640739	1042866.4276665458	2	4.1138023671	0.6957141056
131	348113924239	2445102.4600630270	1	4.1441585740	0.6985536327
137,139	782893951711	66997.9685271023	56	4.2403157062	0.7086830471
149,151,157	963864514519	2107959.7675975439	2	4.2942202694	0.7161568992
163	20044941740191	453137.8910269472	42	4.2508645843	0.6885577345
167	35984570527819	6600795.7661293467	4	4.4014738399	0.7092476786
173,179	46257585588439	30459726.6874839799	1	4.4785200370	0.7200768951
181,191,193	53009903964319	16254586.3732151659	2	4.4650611346	0.7170629964
197	2726829078460579	59344254.1621878925	4	4.5457935599	0.7072031159
199,...,229	2871159201832639	246120736.6299490883	1	4.5932445578	0.7143089705
233	97915624862375191	184534021.4196572267	8	4.7178098548	0.7156122281
239	286657540188128671	249648958.6310063776	10	4.6628138080	0.7023046350
241,...,263	632590969227841471	3833565622.4249429725	1	4.8199375134	0.7223165935
269,...,281	1905834685957869991	3348155946.0528215876	2	4.8505731682	0.7219566218

Table 6.28: $3R_p$ and $7R_p$ - Least Solutions

p	R_p	$R(R_p)$	$h(R_p)$	$L(1,\chi)$	ULI
3	7	2.7686593833	1	1.0464548848	0.4881400578
5	19	5.8289369670	1	1.3372498520	0.5122417374
7	79	5.0751347504	3	1.7129918110	0.5495238232
11	331	36.2563832043	1	1.9928310511	0.5672551030
13	751	57.9421480629	1	2.1143390118	0.5706179805
17	1171	25.3728064560	3	2.2243919456	0.5851342427
19	7459	73.0534161596	3	2.5375901542	0.6108320549
23	10651	270.8720689096	1	2.6246357047	0.6227107192
29	18379	367.1977320417	1	2.7085636844	0.6293491820
31,37	78439	813.5634679127	1	2.9048613936	0.6425763221
41	399499	1890.8635586713	1	2.9915918664	0.6316501081
43	1234531	3537.8678099198	1	3.1841286161	0.6536169887
47,53	1427911	3841.3976895553	1	3.2146870537	0.6576303475
59	4355311	6958.9983656651	1	3.3345493442	0.6653680285
61	5715319	8109.8013153157	1	3.3922666024	0.6730177892
67	49196359	24407.9038423932	1	3.4798778739	0.6624065327
71	117678031	38495.7079874423	1	3.5486618262	0.6654255555
73	180628639	49263.4242643637	1	3.6654836886	0.6824922473
79,83	452980999	78083.7491911782	1	3.6687715128	0.6732611819
89,97	505313251	83941.6234166887	1	3.7341952366	0.6841233484
101,...,109	9248561191	127289.8015022314	3	3.9707978385	0.6984739720
113	152524816291	6690.8406742078	239	4.0945703195	0.6964584023
113,127,131	348113924239	2445102.4600630270	1	4.1441585740	0.6985536327
137	916716646759	3976755.5379911352	1	4.1534724643	0.6930406209
139	1086257787619	637789.4742419198	7	4.2836002584	0.7135133432
149	4606472154439	707977.1594319826	13	4.2882329941	0.7041628149
151	4726529308939	9447793.5416737310	1	4.3456947426	0.7134228217
157	35032713351619	8533304.3173078870	3	4.3251542526	0.6971142316
163,...,179	46257585588439	30459726.6874839799	1	4.4785200370	0.7200768951
181	251274765020899	23977422.8668851469	3	4.5378414684	0.7192684281
191	316934672172031	81024861.1746746802	1	4.5512785800	0.7200367702
193,...,229	2871159201832639	246120736.6299490883	1	4.5932445578	0.7143089705
233,...,263	632590969227841471	3833565622.4249429725	1	4.8199375134	0.7223165935

Table 6.29: $3R_p$ and $7R_p$ - Least Prime Solutions

Chapter 7

Some Results on Other Problems

7.1 On a Successor to Euler's Polynomial $X^2 + X + 41$

Let $f_A(x)$ denote the quadratic polynomial $x^2 + x + A$ ($A \in \mathbb{Z}$) and let $P_A(n)$ denote the number of prime values assumed by $f_A(m)$ for $0 \leq m \leq n$. In 1772 Euler discovered that $f_{41}(x)$ has the rather amazing property that $P_{41}(39) = 40$. Since then it has been learned (see, for example, Ribenboim [Rib88a]) that $P_A(A-2) = A-1$ can only occur for $A = 2$, 3, 5, 11, 17, and 41, where the corresponding imaginary quadratic number fields $\mathbb{Q}(\sqrt{1-4A})$ have class number 1. Before it had been established that 41 is the largest value of A for which this phenomenon can occur, Lehmer [Leh37] made an attempt to find a larger such value of A. Lehmer was able to show that if this larger A value existed, it would exceed 1.25×10^9.

Lehmer's approach to this problem was to first point out that if $f_A(x)$ is to represent primes, then A must be odd. Also,

$$4f_A(x) = (2x+1)^2 - D,$$

where $D = 1 - 4A$. Thus, if q is an odd prime and D is a quadratic residue with respect to q, or equivalently, that the Legendre symbol $(D/q) = 1$, then there must exist some

x such that $0 \leq x < q$ and $f_A(x) \equiv 0 \pmod{q}$. It follows that if $(D/q) = 1$ and $q < A$, then there must be some value of x such that $0 \leq x \leq A-2$ and $q \mid f_A(x)$; thus, $f_A(x)$ cannot be a prime and $P_A(A-2) \neq A-1$.

Lehmer tabulated the values of N_p, where p is an odd prime and N_p denotes the least positive integer of the form $8t+3$ such that $(-N_p/q) = -1$ for all odd primes $q \leq p$. He computed all the values of N_p for $p \leq 107$ and also determined that $N_{109} > 5 \times 10^9$. Now suppose that $A > 41$ and $P_A(A-2) = A-1$; then we must have $-D = 4A-1 \geq N_{41} = 77683$ and $A \geq 19421$. For if $-D < N_{41}$, then there exists some odd $q \leq 41 < A$ such that $(D/q) = 1$, which is impossible. This implies that $P_A(A-2) \geq 19420$. Thus, if $-D < N_{109}$, then for some odd $q \leq 109 < A$, we get $(D/q) = 1$; it follows, then, that $A = (-D+1)/4 > 1.25 \times 10^9$. Lehmer noted that the values of N_p seem to increase so rapidly that it would appear most unlikely that we would ever have $p > (N_p+1)/4$.

Since it seemed most unlikely to Lehmer that this value of A existed at all, he suggested that instead of searching for a polynomial which represents $A-1$ consecutive prime values, that a search should be undertaken for a polynomial better than Euler's in the sense that it is a prime for at least 41 consecutive values of x. In this case we would be searching for a value of A such that $P_A(40) = 41$. An observation due to Granville (see [LMW92]) is that if the prime k-tuples conjecture holds, then for any $n \in \mathbb{Z}^+$ there exists a value for A such that $f_A(x)$ is prime for $x = 0, 1, \ldots, n$. Denote by $A(n)$ the least positive integral value of A such that $P_A(n) = n+1$; then $A(0) = 2, A(1) = 3, A(2) = A(3) = 5, A(4) = A(9) = 11, A(10) = A(15) = 17, A(16) = A(39) = 41$.

Lehmer was also unsuccessful in his attempt to find $A(40)$. Indeed, until fairly recently no other quadratic polynomial of any kind which would produce more than 40 consecutive prime values was known. Fung found a few examples in 1988 (see [MW90]) and a little later Ruby discovered a polynomial which produced 45 consecutive prime values, the current record holder in this respect. [1]

[1] In this we are assuming that the polynomial should produce distinct prime values. Otherwise, Escott's

MSSU was used to search for the value $A(40)$. If we put $K = A(40)$, then we want a value of K such that $K, K+2, K+6, K+12, K+20, K+30, K+42, K+56, \ldots, K+1640$ are all primes.

This means that

$$\begin{aligned} K &\equiv 1 \pmod 2 \\ K &\equiv 2 \pmod 3 \\ K &\equiv 1,2 \pmod 5 \\ K &\equiv 3,4,6 \pmod 7 \\ K &\equiv 1,4,6,7,8 \pmod{11} \\ &\vdots \end{aligned}$$

If we continue this process for all of the moduli supported on the MSSU, we see that the number of admissible residues for $A(40)$ is $(p-1)/2$ for all primes $p < (2 \times 41) = 82$, and $p - 41$ for all p such that $82 < p < A(40)$. Here, the smaller primes have a greater effect on constraining our solutions than do the larger primes. Algorithm 7.1 was used to generate these congruences.

Since we know that $A(40) > 229$, we make use of all of the MSSU virtual rings to eliminate as many false solutions as possible. Unfortunately, a great number of false solutions are still generated which must then be filtered using a fast prime test. The congruences having moduli 2, 3, 5, 7 and 11 were combined, resulting in a single congruence of 30 residue values modulo 2310. The resulting maximum hardware sieving rate for the $A(40)$ problem on the MSSU is 192 million $\times$ 2310 = 4.435×10^{11} per second.

Initially, a fast twin prime test was utilized using the strong pseudoprime to base 2 test on K and $K+2$. For low values of K, where we would expect the density of twin primes to be high, the twin prime filter (the SSUMon `TWPM`) proved to be a bottleneck and

polynomial $x^2 - 79x + 1601 = f_{41}(x - 40)$, which is prime for $n = 0, 1, 2, \ldots, 79$, would hold the record.

Algorithm 7.1 *Generate A(40) Residues*

```
FUNCTION GenA40Residues(Modulus) RETURNS Set ;
INPUT Modulus : Integer ;

i : Integer ;
x : Integer ;
flag : Boolean ;
Residues : Set;

Residues ← {};
FOR i ← 0 TO (Modulus − 1) DO

    flag ← TRUE;
    FOR x ← 0 TO (Modulus − 1) DO

        IF ( (i + x * x + x) mod modulus = 0) THEN
        flag ← FALSE;
        ENDIF

    ENDFOR
    IF (flag = TRUE) THEN Residues ← Residues ∪ {i}

ENDFOR
RETURN(Residues);
```

End of Algorithm

sieving proceeded significantly slower than the maximum hardware rate. A large number of false solutions were generated which then had to be eliminated using an off-line filter, in this case, using a program written in Maple [CGG+88], to calculate the number of consecutive primes assumed by $f_K(x)$ for each value of K. For larger values of K, where we would expect the density of primes to be lower, it was noticed that filtering had a decreasing effect on the sieving rate. Once this was realized, a triple prime filter on K, $K + 2$ and $K + 6$ was utilized which significantly reduced the number of false solutions which had to be processed off-line.

After approximately 20 days of sieving on one MSSU, we were able to establish that

if $A(40)$ exists, then

$$A(40) > 10^{18}.$$

This is the highest bound so far known. We estimate that it would require an additional 180 days of sieving on one MSSU to extend the search to 10^{19}.

7.1.1 Computing $A'(n)$

It is apparent that it is quite a difficult problem to compute $A(40)$. To get some indication of how large we can expect the value of $A(40)$ to be, we propose that values of $A'(n)$ be tabulated where we define $A'(n)$ to be the least positive integer $A' > 41$ such that $f_{A'}(x)$ is prime for $x = 0, 1, 2, \ldots, n$. Here the exceptional values of $q = 2, 3, 5, 11, 17$ and 41 are excluded from consideration for values of A'. We note that by our definition of $A'(n)$, the value of $A'(40)$ will coincide with that of $A(40)$.

The selection of $A'(0) = 43$ is somewhat arbitrary, and we could just as easily have defined $A'(n)$ to be the least positive integer A' not equal to 2, 3, 5, 11, 17 and 41, such that $f_{A'}(x)$ is prime for $x = 0, 1, 2, \ldots, n$. This would result in an initial value of 7 for our sequence. However, such a change in our definition would be of little consequence, affecting only the first few values of the sequence $A'(n)$.

Sieve programs were generated to compute successive values of $A'(n)$. Observing the resulting congruences, we note that the number of acceptable residues decreases gradually for each successive value of n. For the prime moduli less than twice the n value, the resulting congruences are quadratic in character. As with the $A(40)$ case, we were able to partition our sieve problems across 30 sieve chips modulo 2310, however all of the resulting sieve problems were heavily filter-bound. To prevent generating enormous solution files, a triple prime filter (`TRPM`) was used.

Unfortunately, the difficulty in computing this sequence limits the amount of numerical evidence which we can provide to make claims about the growth rate of $A'(n)$,

nevertheless, the resulting tabulation of $A'(n)$ shown in Table 7.1 seems to indicate an exponential growth rate. If this is true, we would expect $A(40) = A'(40)$ to be very, very large, and there is little chance of finding it using sieving techniques.

Computation of $A'(13)$, the largest entry in our table, required in excess of 16 days of sieving using a single MSSU, an amount of computation nearly rivaling that of determining that $A(40) > 10^{18}$. The average canvass rate in computing $A'(13)$ was approximately 14.5 million per second. The values of $A'(8)$ through $A'(12)$ were confirmed by software using a segmented version of the sieve of Eratosthenes which we were able to program to run at a sieving rate of approximately 2×10^6 per second on a SPARCstation 10/51 workstation.

In 1989, Young and Potler [YP89] programmed a segmented sieve of Eratosthenes on a Cray-2 supercomputer to search for large prime gaps, or long strings of consecutive composite numbers. This extensive search tested all numbers up to 7.2×10^{13} at a sieving rate of approximately 8 million per second. This would indicate that a similar computer search for $A'(14)$ up to the range 7×10^{13} would require a significant expenditure of computing power.

n	$A'(n)$
0	43
1	59
2	101
3	101
4	347
5	1 277
6	1 277
7	21 557
8	844 427
9	844 427
10	180 078 317
11	1 761 702 947
12	8 776 320 587
13	27 649 987 598 537
14	$>$ 70 000 000 000 000

Table 7.1: Tabulation of $A'(n)$

7.2 The Erdös-Selfridge Function

For $k \geq 1$, denote by $g(k)$ the least integer $> k+1$ such that no prime $p \leq k$ divides $\binom{g(k)}{k}$. This function grows rapidly with increasing k, is non-monotonic, and is difficult to compute for even modest values of k. The behavior of $g(k)$ was first studied by Ecklund, Erdös and Selfridge [EES74] who tabulated $g(k)$ for $k \leq 40$ as well as $g(42)$, $g(46)$, and $g(52)$. These are all the values of $g(k) \leq 2500000$ when $k \leq 100$. The table was extended to include all the values of $g(k)$ for $k \leq 140$ by Scheidler and Williams [SW92] using sieving techniques. These computations were performed using the OASiS sieve. The largest of these values, $g(139)$, is a 17 digit number. Sieving was continued for $141 \leq k \leq 155$ but the results were never published.

Erdös, Lacampagne and Selfridge [ELS93] showed that $g(k) > ck^2/\ln k$ for some absolute constant $c > 0$, and based on previous numerical evidence, conjectured much larger lower bounds, such as $g(k) > k^3$ for $k > 35$ and $g(k) > k^5$ for $k > 100$. They defined the following

Definition 7.2.1 *The binomial coefficient $\binom{N}{k}$ is said to be* **good** *if the least prime factor of $\binom{N}{k}$ is $> k$.*

Definition 7.2.2 *For $1 \leq i \leq k$ write $N-k+i = a_i b_i$ where b_i contains those prime factors $> k$, and a_i contains those prime factors $\leq k$. Define the* **deficiency** *of a good binomial coefficient as the number of i for which $b_i = 1$.*

Thus the function $g(k)$ can be expressed as the least integer N for which $\binom{N}{k}$ is good. One of the outstanding conjectures of Erdös, Lacampagne and Selfridge is that the deficiency of $\binom{g(k)}{k}$ is 0 for $k > 46$. Using the results of Scheidler and Williams, they were able to verify this conjecture to $k = 149$.

To extend the computations of values of $g(k)$ for $k \leq 200$ the algorithm of Scheidler and Williams was adapted for the MSSU sieve (see [LSW94]). Recomputation of previous

values revealed an error in the value of $g(138)$ given in [SW92]. These new results are consistent with the lower bounds conjectured in [ELS93] and possibly suggest an even larger bound. Indeed, recently, Granville and Ramare [GR94] proved that there exists an absolute constant $c > 0$ such that

$$g(k) > \exp\left(c\left(\log^3 k/\log\log k\right)^{1/2}\right).$$

7.2.1 The Sieving Algorithm

The sieving method used for tabulating $g(k)$ was first described in detail by Scheidler and Williams in [SW92]. The algorithm makes use of a result which is due to an observation by Kummer in 1855 (see [GR94]) about the divisibility of binomial coefficients. This was stated as Theorem 1 in [SW92] as

Theorem 7.2.1 *Let*

$$n = \sum_{i=0}^{t} b_i p^i, \qquad k = \sum_{i=0}^{t} a_i p^i$$

be the base p representations of n and k, respectively, where p is a prime. Then $p \nmid \binom{n}{k}$ if and only if $b_i \geq a_i$ $(i = 0,\ 1, \ldots,\ t)$.

This is the basis of determining a set of possible residues modulo p^m to be used for our sieving problem. For a detailed explanation of the algorithm and how it was derived see [SW92].

To compute the base p representation of an integer k, Scheidler and Williams use the following:

If

$$k = \sum_{i=0}^{m} a_i p^i, \qquad 0 \leq a_i \leq p-1 \quad \text{for} \quad i = 0, 1, \ldots, m; \quad a_m \neq 0$$

is the base p representation of k where $m = \lfloor \log_p k \rfloor \geq 1$, then the coefficients a_i can be computed by a simple iteration putting

$$s_0 = k, \quad a_0 \equiv k \pmod{p}$$

$$s_{i+1} = \frac{s_i - a_i}{p}$$
$$a_{i+1} \equiv s_{i+1} \pmod{p}.$$

For each fixed positive integer k, the moduli of our sieving problem are the values p^m where p is a prime, $p \leq k$, and $p^m \leq k < p^{m+1}$, i.e. $m = \lfloor \log_p k \rfloor$. For each modulus p^m, the corresponding admissible residues are determined as follows:

For $i = 0, 1, \ldots, m$, set

$$C_i = \{a_i, a_i + 1, \ldots, p - 1\}$$

and recursively define the sets

$$B_0 = C_0, \quad B_i = B_{i-1} + C_i p^i = \{b + cp^i \mid b \in B_{i-1}, c \in C_i\} \quad (i = 1, 2, \ldots, m - 1).$$

Then $B_i = C_0 + C_1 p + C_2 p^2 + \cdots + C_i p^i$ for $i = 0, 1, \ldots, m - 1$. The set of admissible residues corresponding to modulus p^m is B_{m-1}. Each solution x of

$$x \pmod{p^m} \in B_{m-1} \tag{7.1}$$

is checked for the additional filter condition

$$\left\lfloor \frac{x}{p^m} \right\rfloor \pmod{p} \geq a_m. \tag{7.2}$$

The least integer x satisfying (7.1) and (7.2) for all primes $p \leq k$ is $g(k)$.

7.2.2 Implementation

To generate problem files for the MSSU, a problem generator program was written which accepts a k value as a parameter. An on-line filter named ERDO was written to test each value x which satisfies (7.1) for condition (7.2). Since the value of k is a parameter of the filter routine, used to determine values for p^m and a_m, the problem files specify the k-value using the PROBLEM PARAMETER feature in the parameters section of the problem file.

Often the problem generation algorithm will require the use of moduli which are prime powers. For k values near 200, we can expect moduli of $2^7 = 128$, $3^4 = 81$, $5^3 = 125$, $7^2 = 49$, $11^2 = 121$ and $13^2 = 169$ to be specified. Since many of these moduli are not available in hardware, a congruence modulo p^m may be reduced to a congruence modulo p^n where $n < m$. The residues in B_{m-1} are then mapped, or *folded*, onto a possibly smaller set of residues modulo p^n. The congruences "lost" in the process of residue folding are implemented in software using an off-line filter that screens out "false" solutions. This does not slow down the sieving process, as the number of false solutions is sufficiently small to avoid a bottleneck.

Partitioning the problem

Partitioning of the sieve problem on the MSSU is complicated by the fact that for moduli which are prime powers there can be two or more possible partitionings due to residue folding. Note also that a partition congruence can be "optimized out" by the MSSU system, and thus a partition congruence can be specified which is not supported by the underlying hardware. For example, for $g(179)$ we have the following eight residue classes modulo 128

```
PARTITION( 128 :  51  55  59  63  115  119  123  127 ; )
```

This is equivalent to the four residue classes modulo 64

```
PARTITION( 64 :  51  55  59  63 ; )
```

Although this 4-residue congruence has half the modulus value of the congruence modulo 128, it is more likely that we will be able to combine this congruence with others to achieve a higher overall rate. However, if other partitionings exist which would result in a higher sieving rate, it may be preferable to partition the problem by folding the above residues onto the modulus 4, resulting in the following single-residue congruence

```
PARTITION( 4 :  3 ; )
```

Subject to limitations of the system, namely that the combined modulus be $< 2^{32}$, a single-residue congruence can always be combined with an existing set of partition congruences.

Therefore, the partition congruences should be selected from the many different possibilities in such a way that M is maximal. Clearly, congruences with few admissible residues (single-residue congruences in particular) and large moduli are most desirable. The number of choices is further increased when partitioning is combined with the effects of residue folding, since for partitioning purposes, a reduced modulus $m_j = p^n$ need not actually be supported by the underlying hardware. However, the total number of congruences is sufficiently small to make an exhaustive search for the optimal combination of residue folding and congruence partitioning computationally feasible.

We conclude this section with a comment on the speed-up suggested in [SW92] in the case where $k+1$ is composite. In this case, each solution candidate n for $g(k)$ satisfies $n \equiv -1 \pmod{k+1}$, so one can sieve on $(n+1)/(k+1)$ rather than n and speed up the process by a factor of $k+1$. The MSSU algorithm achieves essentially the same speed-up as follows. For each prime divisor p of $k+1$, modulus p^m is folded onto modulus p^α where α is the largest exponent such that $p^\alpha \,||\, k+1$. This results in a single residue congruence $n \equiv -1 \pmod{p^\alpha}$ (see the proof of Lemma 2 in [SW92]). Combining these congruences for all primes dividing $k+1$ yields a single residue congruence $n \equiv -1 \equiv k \pmod{k+1}$.

7.2.3 Computational Results

We recomputed $g(k)$ for all $k \leq 140$ and found an error in the table given in [SW92] for $k = 138$. The correct value is $g(k) = 601242167764223$. We also computed $g(k)$ for $141 \leq k \leq 200$. These values are given in Table 7.2. To show the enormous increase in speed of MSSU versus OASiS, the device used for the computations in [SW92], we point

out that OASiS required 11 days 11 hours for computing $g(139)$, whereas MSSU achieved this task in a mere 4 minutes (including time to load and verify the problem).

We verified that the deficiency of $\binom{g(k)}{k}$ is 0 for $135 \leq k \leq 200$ which is consistent with the conjecture of Erdös, Lacampagne and Selfridge that the deficiency of $\binom{g(k)}{k}$ is 0 for $k > 46$.

Sieving rates varied greatly for various values of k. The fastest sieving rate occurred for $k = 199$ with a hardware count rate of 7.5×10^{15} per second, requiring less than 20 hours to compute $g(199)$. One of the more difficult values of k to compute was 198 with a hardware sieving rate of 3.3×10^{13}. This would have taken more than 50 days to compute using only 32 sieve chips. However, we were able to re-partition the problem into 5 subproblems requiring 24 sieve chips each and were able to verify a solution in under 10 days. Due to the very low rate at which solution candidates were generated, solution filtering had a negligible effect on the observed sieving rate. Surprisingly, even with the introduction of false solutions by residue folding, and optimizing out many of the sparse congruences using partitioning, sieving proceeded at essentially the maximum theoretical hardware sieving rate. Using a 6-way partitioning, it took approximately 30 days to compute the largest value found, $g(200)$, which is a 23 digit number.

The table shows that there are frequently significant gaps in size between $g(q-1)$ and $g(q)$ where q is a prime. To highlight this property, prime values of k appearing in our table are displayed in a bold font.

We now provide some evidence to explain this phenomenon. First, we state a Lemma and Corollary which were first proved in [LSW94]. Denote by $r_p(k)$ the number of residues specified for modulus p when computing $g(k)$.

Lemma 7.2.2 *(Lemma 3.7 in [LSW94]) Let $k = q^t$, q a prime, $t \geq 1$. Then for any prime $p \leq k$, the number of admissible residues for modulus p^m, $m = \lfloor \log_p k \rfloor$, satisfies the following properties.*

1. *If $p = q$, then $r_p(k) = p^m$, $r_p(k-1) = 1$ if $m > 1$, and no power of p is included in the moduli for $k-1$ if $m = 1$.*

2. *If $p \neq q$, then*
$$r_p(k) = r_p(k-1)\left(1 - \frac{1}{p - k_p}\right),$$
where $k_p \equiv k - 1 \pmod{p}$, $1 \leq k_p \leq p-1$.

Corollary 7.2.3 *(Corollary 3.8 in [LSW94]) Let $k = q$ be a prime. Then the moduli in the congruences for both $g(q)$ and $g(q-1)$ are exactly the powers p^m where p is a prime less than q and $m = \lfloor \log_p k \rfloor$. Furthermore, for each such prime p,*
$$r_p(q) = r_p(q-1)\left(1 - \frac{1}{p - q_p}\right),$$
where $q_p \equiv q - 1 \pmod{p}$, $1 \leq q_p \leq p-1$.

Corollary 7.2.3 implies that
$$r_p(q) \leq r_p(q-1)\left(1 - \frac{1}{p}\right)$$
for each modulus p^m, $p < q$. Denote by $P_p(k)$ the probability that a solution candidate x satisfies (7.1). Then under the reasonable assumption that each admissible residue is equally likely we get
$$P_p(k) = \frac{r_p(k)}{p^m},$$
so the probability that x satisfies (7.1) for all moduli p^m is
$$P(k) = \prod_{p \leq k} P_p(k).$$
It follows that
$$P(q) \leq P(q-1) \prod_{p \leq q-1} (1 - \frac{1}{p}).$$
By Mertens's theorem
$$\prod_{p \leq k} (1 - \frac{1}{p}) \sim \frac{e^{-\gamma}}{\log k};$$

hence the two probabilities $P(q-1)$ and $P(q)$ will tend to differ by at least a factor which is proportional to $\log q$. Thus, we expect the gaps between $g(q-1)$ and $g(q)$ to increase significantly for large primes q.

7.3 Pseudocubes

Pseudocubes are defined to be those integers of the form $9k \pm 1$ which are cubic residues of all primes less than or equal to some prime p, but are not perfect cubes. This problem was first examined by Cobham [Cob68] who used a conventional computer search to produce an unpublished table of pseudocube values up to $p = 139$. Stephens [Ste89] used the OASiS sieve to extend this search to $p = 313$.

Although pseudocubes appear to have far less applicability than the pseudosquares, their computation is instructive as they are an example of a sieving problem of cubic character. More to the point, because features of the OASiS sieve make it particularly well suited for computing pseudocubes, it is a good problem for comparing the relative performance of the OASiS and MSSU sieves. Also, the pseudocubes problem gives us the opportunity to test the perfect cube filter routines of Section B.2.

In an attempt to extend the list of known pseudocubes we expended 258 hours of sieving time on one MSSU to search the interval from 0 to 10^{17}. One new pseudocube value (not including repeats) was discovered to extend the table to $p = 349$. The complete list of known pseudocube values is given in Table 7.3. Adhering to the format used by Stephens (see Table 7-9 [Ste89, pp. 117]), we have omitted from our table listing pseudocube values N_p for primes $p \equiv 2 \pmod 3$ because they are all repeats with the exceptions of N_2 and N_{83} which are first occurrences.

Stephens noted that the relative scarcity of cubes as opposed to squares resulted in far less filtering for the pseudocubes problem versus the pseudosquares problem. Stephens also noted that since every number except 0 is a cubic residue modulo p if $p \equiv 2 \pmod 3$,

Table 7.2: Tabulation of the Erdös-Selfridge Function

k	$g(k)$
140	908985 4222866845
141	6333152 3816662671
142	1990465 6320115423
143	1542289 5461804543
144	139719 3586455769
145	339515 6674599871
146	17509 5016485374
147	72531 1731192223
148	1180 8400809148
149	542394 5342959799
150	47313 8520098551
151	3258989 9217872863
152	15549796 7465547419
153	53518499 5256751839
154	17864690 7528990874
155	4129798 4000013467
156	2527233 4970944959
157	104130829 7102375167
158	4702566 0758882783
159	6485551 8266246559
160	1664745 6280932287
161	342159 0108339941
162	20212 9337635322
163	5188127 2225707439
164	14664726 1829992439
165	9865227 4401898671
166	347584 7868933047
167	277308556 4165092343
168	48669223 2365306798
169	72698097 9380669099

k	$g(k)$
170	5942841 5007516671
171	28496594 9074228671
172	11223206 5794463997
173	138175311 6390427373
174	11057733 6695616174
175	194409219 4361247743
176	22625530 3912072703
177	19246523 8561441207
178	684480 9280136434
179	90787419 7930300859
180	43976301 6255983614
181	2 8336150170 1232528573
182	2898883863 4918997183
183	5351624705 6143575999
184	2662687844 8827721469
185	3713603655 0263266493
186	1 4274157994 6200597438
187	220884991 2824359867
188	127198106 5611178943
189	295629805 3153332989
190	45565223 2192890367
191	939688321 4719852991
192	106365072 4436901873
193	4 3525141972 8230720249
194	1 2638743588 3753706219
195	2632591216 1870817495
196	78151666 4215365373
197	4 2796097712 6350089949
198	1 3533936460 3654686198
199	4 0316886886 7096129999
200	520 8783889271 0191382732

it was beneficial to use the variable-size ring feature of OASiS to omit these moduli and instead load the hardware rings for moduli p where $p \equiv 1 \pmod 3$, excluding roughly two-thirds of their residues. Using this technique OASiS was able to load 28 of these "desirable" sparse moduli, while MSSU is only able to load 14 such moduli in its fixed-size rings. Nevertheless, over the interval from 0 to 10^{15}, OASiS required 700 hours of sieving time, whereas MSSU required less than 3 hours. OASiS was however much closer to achieving its maximum hardware sieving rate of 4.25×10^8 per second versus an actual rate of 3.97×10^8 over this range. This compared to the MSSU maximum hardware rate of 3.145×10^{11} per second versus an observed rate of 9.25×10^{10}.

We attribute the very good showing of the MSSU on the pseudocubes problem to the efficiency with which it performs filtering. The fast virtual ring filter gives MSSU the capability to overcome much of the hardware limitations of the underlying SSU devices. The MSSU perfect cube filter utilizes the fastest algorithm currently known for determining whether a given 128-bit binary number is a perfect cube. The PCUB filter is described in Section B.2. In comparison, the OASiS algorithm was implemented using a custom multi-precise math package which performed a version of Newton's method followed by a cube operation, requiring two multiplies and a comparison. It should be noted that the OASiS rate could possibly have been significantly increased by re-partitioning the pseudocubes problems into sub-problems as is done on the MSSU and employing a more efficient perfect cube filter. However, it is our belief that it would not be long before the solution rate would overwhelm the very inefficient filter capability of the OASiS system.

By experimentation, it was discovered that the sieving rate of the pseudocubes problem on the MSSU was very sensitive to the ordering of the virtual and perfect-cube filters. If the perfect-cube filter was chosen as the prefilter and the virtual filter as the postfilter then the sieving rate was one fourth of that of the alternate ordering. This illustrates the importance of the user being aware of the relative costs of the filter routines when specifying multiple filters. We expect that for the majority of problems where a virtual ring will be used, it should be specified as the prefilter.

k	p_k	L_p	Source
* 1	2	17	Cobham (1968)
4	7	71	computer search
6	13	181	
8	19	2 393	
11	31	3 457	
12	37	5 669	
14	43	74 339	
18	61	74 339	
19	67	166 249	
21	73	2 275 181	
22	79	7 235 857	
* 23	83	†7 298 927	
25	97	8 721 539	
27	103	8 721 539	
29	109	91 246 121	
31	127	91 246 121	
34	139	98 018 803	
36	151	1 612 383 137	Stephens (1989)
37	157	1 612 383 137	OASiS
38	163	7 991 083 927	
42	181	7 991 083 927	
44	193	7 991 083 927	
46	199	20 365 764 119	
47	211	2 515 598 768 717	
48	223	6 440 555 721 601	
50	229	29 135 874 901 141	
53	241	29 135 874 901 141	
58	271	29 135 874 901 141	
59	277	406 540 676 672 677	
61	283	406 540 676 672 677	
63	307	406 540 676 672 677	
65	313	406 540 676 672 677	
67	331	75 017 625 272 879 381	MSSU
68	337	75 017 625 272 879 381	
70	349	75 017 625 272 879 381	
73	367	$>$ 100 000 000 000 000 000	

Table 7.3: Pseudocubes

* primes are $\equiv 1 \pmod 3$

† value first appeared in [Ste89]

7.4 Pseudopowers

A problem proposed by Bach et al. [BLSW95] is that of computing pseudopowers. For example, a pseudopower of the base 2 is a positive integer which is not a power of 2 but looks like a power of 2 modulo all primes less than some bound. The pseudopower problem is very much the analogue of the pseudosquares problem (see Section 6.1), with powers of 2, instead of squares.

Definition 7.4.1 *We define an integer n to be an* x-pseudopower *of the base b if n is not a power of b (hence $n > 1$), but looks like a power of b modulo all primes $\leq x$, i.e. if for all primes $p \leq x$ there exists an integer exponent $e_p \geq 0$ such that $n \equiv b^{e_p} \pmod{p}$. We denote the least such x-pseudopower as $P_b(x)$.*

For simplicity, we restrict our definition of pseudopowers to bases where b is prime. If n is the least x-pseudopower to base b and $x \geq b$ then $n \equiv 1 \pmod{b}$. Clearly if n is divisible by b then n cannot be the least x-pseudopower since n/b is also a pseudopower.

Denote by $\langle b \rangle_p$ the multiplicative group generated by b modulo p. Thus for base 2, the least x-pseudopower must satisfy the following set of congruences for primes $p_1 = 2,\ p_2, \ldots,\ p_k$ where $p_k \leq x$

$$\begin{aligned} n &\equiv 1 \pmod{2} \\ n &\equiv 1,2 \pmod{3} \\ n &\equiv 1,2,3,4 \pmod{5} \\ n &\equiv 1,2,4 \pmod{7} \\ &\vdots \\ n &\in \langle 2 \rangle_{p_k} \pmod{p_k} \end{aligned} \tag{7.3}$$

It is not necessary to specify any additional filter conditions in addition to the set of congruences (7.3) since the congruence $n \equiv 1 \pmod 2$ ensures that no actual power of 2 is generated as a solution. Of course, the solution candidate 1 is treated as a special case and must be discarded.

7.4.1 A Heuristic Estimate for $P_2(x)$

In [BLSW95] Bach et al. considered a probabilistic model for estimating the value of $P_2(x)$. This is similar to the heuristic analysis performed in Section 6.1.2 for the pseudosquares. The fraction of integers n satisfying the system of congruences (7.3) for $p \leq n$ is

$$\frac{1}{2} \prod_{3 \leq p \leq x} \frac{|\langle 2 \rangle_p|}{p}.$$

If we make the somewhat simplistic assumption that solutions to (7.3) are equi-distributed throughout the region 0 to $2 \cdot 3 \cdot 5 \cdots p_k$ then we would expect that

$$P_2(x) \approx P_2'(x),$$

where

$$P_2'(x) = 2 \prod_{3 \leq p \leq x} \frac{p}{|\langle 2 \rangle_p|} = \left(\prod_{2 \leq p \leq x} \frac{p}{p-1} \right) \left(\prod_{3 \leq p \leq x} \frac{p-1}{|\langle 2 \rangle_p|} \right). \tag{7.4}$$

By Mertens's theorem (see Theorem 6.5), the first factor is asymptotic to $e^\gamma \log x$, where γ is Euler's constant. The second product can be expressed as

$$\prod_{3 \leq p \leq x} \frac{p-1}{|\langle 2 \rangle_p|} = \exp \left(\sum_{3 \leq p \leq x} \log \frac{p-1}{|\langle 2 \rangle_p|} \right).$$

Let $c_2^{(k)}$ denote the average value of $\log \frac{p-1}{|\langle 2 \rangle_p|}$ over the odd primes $p \leq p_k$. Assuming that the value of $c_2^{(k)}$ for odd primes $p_k \leq x$ has a limit c_2 as $x \to \infty$, we can approximate $\sum_{3 \leq p \leq x} \log \frac{p-1}{|\langle 2 \rangle_p|}$ by $c_2(\pi(x) - 1)$.

In [BLSW95], arguments are presented to show that the average value of $\log \frac{p-1}{|\langle 2 \rangle_p|}$ does in fact converge to a limit value of $c_2 \doteq 0.927346$ (correct to six figures). This suggests

that

$$P_2(x) = e^{\gamma} \log(x) \left(e^{(\pi(x)-1)}\right)^{c_2+o(1)}. \tag{7.5}$$

In fact, Bach et al. conjecture that

$$P_2(x) \sim \exp \frac{c_2 x}{\log x}.$$

Table 7.4 lists the pseudopowers to base 2 computed using the MSSU sieve. Only the "record-setting" values of k are listed, that is, those k for which $P_2(p_k) \neq P_2(p_{k-1})$. To give some numerical evidence in support of the heuristic estimate for $P_2(x)$, we have also tabulated the integer part of P'_x, the value of $c_2^{(k)}$ and the value of the ratio

$$R_2^{(k)} = \frac{\gamma + \log\log(p_k) + c_2^{(k)}(k-1)}{\log P_2(k)}.$$

We see that the value $c_2^{(k)}$ appears to converge slowly to the asymptotic value of $c_2 \doteq 0.927346$. Nevertheless, as we would expect, it appears that $R_2^{(k)} \to 1$.

As it turns out, (7.5) is not a very good predictor for the value of $P_2(x)$. This is due not only to the slow convergence of the average value of $\log \frac{p-1}{|\langle 2 \rangle_p|}$ for odd primes $p \leq x$ to c_2, but also the slow convergence of $\prod \frac{p}{p-1}$ over the primes $p \leq x$ to $e^{\gamma} \log x$. Within the range of this table, $P'_2(x)$ is a much more accurate predictor of the value of $P_2(x)$.

7.4.2 Pseudopowers of Other Bases

In Tables 7.5, 7.6, and 7.7 we present our computations of the pseudopowers for bases 3, 5, and 7.

Recall that $P_b(p_k)$ is the least number greater than 1 satisfying $n \in \langle b \rangle_{p_i} \pmod{p_i}$ for $i \leq k$, $p_i \neq b$, and $n \equiv 1 \pmod{b}$ (if $b \leq p_k$).

Using the same heuristic argument as before, we would expect

$$P_b(x) \approx b \prod_{2 \leq p \leq x, p \neq b} \frac{p}{|\langle b \rangle_p|}, \tag{7.6}$$

k	p_k	$P_2(p_k)$	$\lfloor P_2'(p_k) \rfloor$	$c_2^{(k)}$	$R_2^{(k)}$
1	2	3	2	0.000000	0.192
2	3	5	3	0.000000	0.417
3	5	7	3	0.000000	0.541
4	7	11	8	0.231049	0.807
5	11	23	9	0.173287	0.684
7	17	43	22	0.231049	0.799
9	23	127	48	0.259930	0.784
11	31	1087	314	0.387120	0.813
14	43	2209	2032	0.435612	0.982
15	47	2837	4152	0.454008	1.042
20	71	7603	9014	0.371013	1.016
21	73	115669	73115	0.456435	0.957
24	89	1062839	1212924	0.517447	1.007
25	97	4007837	2451119	0.524768	0.966
30	113	38863631	61670169	0.543879	1.024
31	127	101665279	1118873083	0.622095	1.129
33	137	234556697	2271540184	0.604875	1.117
36	151	1848054121	23188164852	0.618817	1.118
48	223	3131990286049	7128868199136	0.581310	1.028
50	229	41398091214971	21575451536777	0.580003	0.979
51	233	335444151885977	173347593381698	0.609992	0.980
52	239	663176716985449	348151889228789	0.611623	0.981
53	241	10600009924847711	3496025221005765	0.644141	0.970
54	251	28185732773917153	17550046609448938	0.662354	0.987
55	257	306313044048233909	281897623664273568	0.701433	0.998

Table 7.4: Pseudopowers to base 2

and,

$$P_b(x) = (b-1)e^{\gamma}\log(x)\left(e^{(\pi(x)-1)}\right)^{c_b+o(1)} \tag{7.7}$$

where c_b is the asymptotic average value of $\log \frac{p-1}{|\langle b\rangle_p|}$. To compare the data with heuristic predictions, we also tabulated $c_b^{(k)}$, (the average value of $\log \frac{p-1}{|\langle b\rangle_p|}$ over the primes different from b and $\leq p_k$) and the ratio

$$R_b^{(k)} = \frac{\gamma + \log(b-1) + \log\log(p_k) + c_b^{(k)}(k-1)}{\log P_b(k)}$$

for $b = 3, 5, 7$.

7.4.3 The c_b Values

Bach et al. [BLSW95] have presented arguments to show that the limit

$$\lim_{x\to\infty} \frac{1}{\pi(x)-1} \sum_{p\leq x, p\neq b} \log \frac{p-1}{|\langle b\rangle_p|} \tag{7.8}$$

has a value c, where

$$c \doteq 0.89846489937400140618.$$

The observed averages for $\log \frac{p-1}{\langle b\rangle_p}$ for odd primes $p \leq 10^6$, $p \neq b$, for the bases 2, 3, 5, and 7 are 0.923, 0.896, 0.895, and 0.893, respectively. In fact, due to the quadratic character of 2, the resulting constant for $b = 2$ is given by

$$c_2 \doteq 0.92734603189733246 07.$$

Bach et al. conjecture that the limit of the average value of (7.8) is c_2 for $b = 2$, and c for all other bases, $b = 3, 5, 7, \ldots$

k	p_k	$P_3(p_k)$	$c_3^{(k)}$	$R_3^{(k)}$
1	2	5	0.000000	0.562
2	3	7	0.000000	0.701
4	7	13	0.000000	0.755
5	11	31	0.173287	0.826
6	13	157	0.415888	0.849
9	23	841	0.346574	0.770
10	29	859	0.308065	0.778
12	37	1543	0.315067	0.820
13	41	6241	0.422931	0.876
18	61	36481	0.485484	1.041
19	67	170041	0.519547	1.001
20	71	241081	0.528684	1.030
21	73	1515361	0.591837	1.023
27	103	16226731	0.550833	1.032
28	107	32913169	0.556104	1.030
29	109	52078027	0.585753	1.082
36	151	872200213	0.519796	1.024
41	179	1327190419	0.506807	1.104
42	181	8479278889	0.528258	1.075
44	193	89400402001	0.577596	1.101
58	271	384810485528569	0.559402	1.039
63	307	2346816388490401	0.572087	1.087
65	313	150139363999760521	0.597531	1.043

Table 7.5: Pseudopowers to base 3

k	p_k	$P_5(p_k)$	$c_5^{(k)}$	$R_5^{(k)}$
1	2	3	0.000000	1.454
2	3	7	0.000000	1.057
3	5	11	0.000000	1.017
5	11	31	0.173287	1.028
8	19	311	0.354987	0.963
10	29	961	0.353117	0.925
11	31	3931	0.548064	1.048
17	59	32761	0.429183	0.985
19	67	96721	0.481038	1.050
20	71	2048071	0.594618	1.012
22	79	3962941	0.570995	1.016
24	89	15942061	0.551480	0.974
34	139	1049824801	0.543683	1.035
42	181	537343041691	0.592626	1.033
43	191	6791126548441	0.633339	1.023
53	241	28764591571409101	0.641573	0.977
54	251	88428973201069961	0.672913	1.008

Table 7.6: Pseudopowers to base 5

k	p_k	$P_7(p_k)$	$c_7^{(k)}$	$R_7^{(k)}$
1	2	3	0.000000	1.823
2	3	13	0.346574	1.095
4	7	43	0.173287	0.945
8	19	463	0.354987	0.967
10	29	4453	0.430133	0.887
12	37	6889	0.540968	1.087
14	43	10837	0.607427	1.248
17	59	433777	0.623500	1.059
21	73	1639597	0.553730	1.041
23	83	2026627	0.534898	1.076
27	103	2767129	0.479265	1.103
28	107	45401119	0.461515	0.929
30	113	540936733	0.549194	0.987
33	137	2502103171	0.541029	0.983
34	139	7329561037	0.545639	0.967
42	181	147395795779	0.565830	1.058
43	191	2125965901123	0.622464	1.063
44	193	91729981475959	0.656347	1.003
46	199	258884537383081	0.657982	1.014
54	251	5016296353717489	0.631706	1.039
57	269	24275674347323869	0.597865	0.996
60	281	1546083343852656253	0.635691	0.993
61	283	1761176025342168541	0.636649	1.007

Table 7.7: Pseudopowers to base 7

7.5 Two-Dimensional Sieving Problems

In Section 5.2.3 we described the multi-variable sieving capability of the MSSU. In the following, we describe how this capability was used to tackle a problem which has long been the domain of some of the world's fastest digital computers.

7.5.1 Sums of Three Cubes

It is of some interest to determine those integers d which can be represented as the sum of three integer cubes

$$x^3 + y^3 + z^3 = d. \tag{7.9}$$

Very little is known about the solutions of (7.9), except for a small number of d values. Since any integer cube is $\equiv \pm 1,\ 0 \pmod 9$, it is well known that $d \not\equiv \pm 4 \pmod 9$. However, this is the only local solvability condition which has been deduced for d. Of course, for $d = 0$, it has long been known that there are no solutions, other than the trivial solutions where $xyz = 0$.

This is a longstanding problem which has been the subject of many computer searches. In 1955, at the suggestion of L.J. Mordell, whose primary interest was the case $d = 3$, Miller and Woollett [MW55] used the EDSAC computer to search for solutions of (7.9) for $0 < |d| \leq 100$ and $max(|x|, |y|, |z|) \leq 3200$. The result was a table of all solutions over this range. Solutions were identified as *primitive* if x, y and z possessed no common factors (i.e. $(x, y, z) = 1$), and *derived* if $(x, y, z) > 1$.

Later, in 1964, S. Chowla's interest, again in the case $d = 3$, prompted Gardiner, Lazarus and Stein [GLS64] to employ an IBM STRETCH computer to extend the search for solutions for $0 < |d| < 1000$, for $0 \leq x \leq y \leq 2^{16} = 65536$, $0 < z - x \leq 2^{16}$. Like the earlier search of Miller and Woollett, Gardiner et al. used a logical search approach whereby solutions are found with increasing x, y and z, with no apparent ordering with

respect to d. Thus for effort not much greater than that of searching for solutions to $d = 3$, the entire range $0 < |d| < 1000$ was covered.

More recently, Scarowsky and Boyarsky [SB84] have investigated the case $d = 3$ along the planes $x+y+z = 3m$, $m \in \mathbb{Z}$ for $|m| \leq 50000$ finding no new solutions. Heath-Brown, Lioen and te Riele [HBLtR93] used a Cyber 205 vector supercomputer to investigate the cases $d = 2,\ 3,\ 30,\ 39$ and 42, finding a new solution for $d = 39$. Koyama [Koy91] used a Cray vector supercomputer to search for $|d| < 1000$ to the range $max(|x|, |y|, |z|) < 2^{21}$ finding numerous solutions, including the first known example for $d = 75$.

It is well known (see [MW55]) that there are an infinite number of solutions for $d = 1$ and $d = 2$. For $d = 1$ an infinitude of solutions are given by the parametric equation

$$x = 9t^4, \quad y = -9t^4 + 3t, \quad z = -9t^3 + 1.$$

Indeed, Lehmer [Leh56] exhibited an infinite sequence of explicit parametric solutions for $d = 1$. For $d = 2$, an infinitude of solutions are given by

$$x = 6t^3 + 1, \quad y = -6t^3 + 1, \quad z = -6t^2.$$

For $d = 3$, little is known, other than the existence of the two solutions $(1, 1, 1)$ and $(4, 4, -5)$. It is not known whether the number of solutions for $d = 3$ is just 2, a finite number greater than 2, or an infinite number.

Miller and Woollett [MW55] commented at some length on how the relative difficulty of computing solutions to various values of $d \equiv 9n \pm \kappa$ may be dependent on the value of κ. Since all cubes are of the form 9λ or $9\lambda \pm 1$, there are 27 combinations of (α, β, γ) for any solution $x = 3\lambda_1 + \alpha$, $y = 3\lambda_2 + \beta$, and $z = 3\lambda_3 + \gamma$. Tabulating the number of combinations for each of the possible forms of d modulo 9 results in the following table.

Form	No. of Combinations
$9n$	7
$9n \pm 1$	12
$9n \pm 2$	6
$9n \pm 3$	2
$9n \pm 4$	0

From this table we would expect d-values of the form $9n \pm 1$ to have relatively abundant solutions, and solutions for d-values of the form $9n \pm 3$ to be relatively scarce. This could explain why the majority of values of d for which no solutions are known (most notably, 30, 33 and 42) are of the form $9n \pm 3$.

At the present time, there are 5 values of $|d| < 100$ for which no solutions are known. These are

$$30,\ 33,\ 42,\ 52,\ 74.$$

From a list provided by Guy in [Guy94, p. 151] combined with more recent results, we have the following d values for $100 < d < 1000$ for which no solutions are known at present

$$114, 156, 165, 195, 290, 318, 366, 390, 420, 444, 452, 462, 501, 530,$$

$$534, 564, 579, 588, 606, 609, 618, 627, 633, 732, 735, 758, 767, 786,$$

$$789, 795, 830, 834, 861, 894, 903, 906, 912, 921, 933, 948, 964, 969, 975.$$

Bremner's Idea

Bremner [Bre94] observed that for many of the known solutions of the three cubes problem the sum $y + z = m$ tends to be quite small. He suggested making use of this observation by searching along the planes $y + z = m$ ($m \in \mathbb{Z}$, $m \neq 0$). Putting this into (7.9) results in the following elliptic curve

$$3mY^2 = 4X^3 - (m^3 - 4d) \tag{7.10}$$
$$Y = 2y - m,\ X = -x.$$

Bremner used two filters to restrict the values of m which are tested. To ensure an integer solution, it is necessary that (7.10) be solvable locally for all primes (p-adic integer solvability). This restricts the number of possible m-values somewhat; however,

to reduce this list further, Bremner decided to consider only those values of m for which the Mordell-Weil rank of (7.10) is odd.

Using a DEC 5000 workstation, Bremner used this search technique to search for solutions to the remaining values of $|d| < 1000$ for which no solutions were known for $|X| < 10^7$ and $|m| < 20,000$. His search uncovered a primitive solution for $d = 75$, with a corresponding derived solution for $d = 600$, and another primitive solution for $d = 768$.

Implementation for MSSU

Bremner's search is 2-dimensional, in that searching proceeds in both the x and m directions. To perform this search on the MSSU we made use of its multi-variable search capability as described in Section 5.2.3. 32 m-values were tested in parallel for each sieve, with sieving along the x direction proceeding at a rate of 1.92×10^8 per second. Each sieve job was limited to 3 minutes of sieving time, bounding x by $|x| \leq 3.456 \times 10^{10}$. Taking into account the overhead of filtering, which is very slight for this particular problem, we can confidently assert that all x values were searched in the range $0 \leq |x| \leq 3 \times 10^{10}$.

Due to the large number of congruences which need to be specified, problem files were approximately 300 kilobytes in size. Up to 2 minutes were required by the host computer system to parse a problem file and download the problem onto the sieve. In total, approximately 384 different values of m can be tested per hour specifying these problem parameters.

Since odd Mordell-Weil rank is not a necessary condition for a solution, to ensure that no solutions were missed over a particular range, we chose to restrict m solely by local solvability conditions. A large number of m values can be eliminated by making use of a simple property of our elliptic curve. From (7.10) we have

$$3mY^2 = 4X^3 - (m^3 - 4d)$$

we see that if $p \mid m$, then $4x^3 + 4d \equiv 0 \pmod{p}$ and

$$4(x^3 + d) \equiv 0 \pmod{p}$$

$$x^3 \equiv -d \pmod{p}$$

Thus if there is no x such that $x^3 \equiv -d \pmod{p}$ then $p \nmid m$.

We can further restrict our m values by examining admissible residues of m modulo the prime powers $2^3 = 8$ and $3^2 = 9$. For example, to compute the acceptable m values for $d = 3$ we have

$$x^3 + y^3 + z^3 \equiv 3 \pmod{9}$$

$$x^3 \equiv y^3 \equiv z^3 \equiv 1 \pmod{9}$$

$$x \equiv y \equiv z \equiv 1,\ 4,\ 7 \pmod{9}$$

Therefore

$$m = y + z \equiv 2,\ 5,\ 8 \pmod{9}.$$

combining these two techniques, a large number of inadmissible values of m can be rejected.

From our elliptic curve equation (7.10) it is fairly straightforward to program a routine to generate admissible residues of X modulo primes p. To ensure that the corresponding value for Y^2 will be positive, we assume that m and X are of the same sign. Thus if a solution is generated for $m < 0$ we assume that X represents a negative value. This leaves a small region which is not covered by our search for small X. This "missing" region consists of three separate parts. The first region consists of those values for which $m = y + z = 0$. In this case, we are looking for solutions to

$$x^3 = d.$$

We tested that no solutions of this type existed for all d-values of interest within the bounds of our search. The second "missing" region occurs when $m > 0$ and $X < 0$. Here,

our elliptic curve equation (7.10) becomes

$$3|m|Y^2 + 4|X|^3 + |m|^3 = 4d,$$

and a simple program suffices to test all possible combinations of m, X and Y. The final case we must consider is when $m < 0$ and $X > 0$. This, however, results in an impossible situation since from (7.10) we have

$$-3|m|Y^2 = 4|X|^3 + |m|^3 + 4|d|,$$

and it is not possible for the left hand side of this equation to ever equal the right hand side.

There are two possible types of false solutions. Since our sieve only specifies a finite number of moduli, we can expect some solutions will pass the congruences specified by our sieve problem, and yet not be valid with respect to some unspecified moduli. Another type of false solution is what is called a *half-integer* solution. This occurs when the value $y = \frac{1}{2}(Y + m)$ has a non-integer part of $\frac{1}{2}$. These solutions can be valuable in that they can be used to derive a new solution for $2^3 d$ by multiplying the values of x, y, and z by 2. In the search range which we examined, the only half-integer solution of interest would be for $d = 114$ which could be used to derive a first known solution for $d = 8 \cdot 114 = 912$. An off-line solution filter was written to identify solutions as being either integer solutions, half-integer solutions, or false solutions.

Computational Results

MSSU was employed to search for solutions to $d = 3$, and the 5 cases of d less than 100 for which no solutions are known, namely $d = 30,\ 33,\ 42,\ 52,\ 74$, over the range

$$|m| \leq 100,000$$

$$|x| \leq 3 \times 10^{10}$$

Unfortunately, this failed to uncover any new solutions.

We continued our search for $d =$

$$110, 114, 156, \ldots$$

discovering three new solutions.

For $d = 110$, $m = -1619$,

$$x = 109938919,$$

$$y = 16540290030,$$

$$z = -16540291649.$$

For $d = 435$, $m = 1211$,

$$x = -981038126,$$

$$y = 509795655496,$$

$$z = -509795654285.$$

For $d = 478$, $m = 58111$,

$$x = -4388907,$$

$$y = 22050441,$$

$$z = -21992330.$$

This eliminated 110, 435, and 478 from the list of values of $|d| < 1000$ for which no solutions were known. We note that the unusually small m-value of the solutions for $d = 110$ and $d = 435$ would indicate that Bremner's observation that m tends to be small holds even when x, y, and z are comparatively large. The large x-values of these two solutions put them well outside the range of previous searches. For the solution for $d = 478$, the m-value of 58111 puts it outside the range of Bremner's search.

Since many mathematicians and powerful computers have failed to uncover solutions for the 5 remaining values of $d < 100$, there is a certain amount of "glory" attached to these values. Thus, we made a further attempt to find these solutions by broadening our search to include all values of $m \leq 400000$. This was performed for $d = 33$ and $d = 42$, however no solutions were found.

To give some idea of the sieving times involved, for the case $d = 30$ which encompasses 17594 admissible values of m for $|m| \leq 100000$, approximately 50 hours of sieving was required using one MSSU.

Due to the massive amount of disk space required by the problem files for each value of d, the DELETE feature of the host software was used during sieving. This feature deletes a problem file if sieving completes without finding any solutions or encountering any error conditions. This allows the sieve operator to enqueue several hundred sieve jobs, and when sieving completes, only problem files which require the attention of the operator remain. This frees up what can be several hundred megabytes of disk space (as occurred for $d = 3$) and also eliminates the organization nightmare posed by maintaining several hundred problem files.

7.6 Sums of Four Cubes and Waring's Problem

The following investigation of representing integers as the sum of integer cubes was performed without the aid of an electronic sieve device, nevertheless we include it here as it relates to the work in the previous section. Determining those integers which can be expressed as the sum of 3 positive or negative integer cubes is closely associated to the Easier Waring's Problem for cubes. If it were possible to express every integer $d_3 \not\equiv \pm 4 \pmod 9$ as a sum of 3 cubes then clearly every integer could be expressed as a sum of 4 cubes (the fourth cube being either 0 or 1). However, as we showed in the previous section, this is far from being proved or disproved.

Waring's Problem is to find the least value of $g(k)$ [2] such that every positive integer can be expressed as the sum of at most $g(k)$ kth powers of positive integers (see [Ell71]). For cubes it is a longstanding result that no more than 9 cubes are required to represent every integer [Dic28], and thus $g(3) = 9$. The so-called Easier Waring's Problem is that of determining the least value of $v(k)$, such that every integer (positive or negative) can be expressed as the sum of $v(k)$ positive or negative kth powers. In 1894, Oltramare [Wri34] proved that $4 \leq v(3) \leq 5$. In an attempt to find numerical evidence that $v(3) = 5$, we searched for integers which are difficult to represent as the sum of 4 positive or negative integer cubes (4 cubes for brevity). Instead, we found some numerical evidence to support that $v(3) = 4$ by determining that all integers n, where $|n| < 2 \times 10^8$, can be represented as the sum of 4 cubes.

Positive and Negative Cubes

A computer search was undertaken to determine those integers d which are difficult to represent as the sum of 4 cubes. In our algorithm, we first determined all $d_3 = x^3+y^3+z^3$ for $0 < d_3 < 10^6$ and $max(|x|, |y|, |z|) \leq 1300$ and stored these values of d_3, x, y, and z in an array. We then determined all values of d_4, for $0 < |d_4| < 10^6$, which can be formed by taking an element from the set d_3 and adding or subtracting integer cubes w^3, where $0 \leq w < 100$. This program took approximately 20 minutes of CPU time on a DEC Alpha AXP3300LX workstation. The result of this process was that the only integer d_4 between 0 and 10^6 which could not be represented in this manner was the value 82562. In an attempt to find further examples of integers difficult to represent as the sum of 4 cubes, we extended the search for $0 < d_3, d_4 < 4 \times 10^6$ and $max(|x|, |y|, |z|) \leq 1500$ and $0 \leq w < 150$, and this time found two representations for $d_4 = 82562$ [3] expressed as the tuples (x, y, z, w): (-1498, 1490, 377, 41) and (350, -331, -163, -130). This search uncovered no other values of d_4 in the range $0 < |d_4| < 4 \times 10^6$ which could

[2] This $g(k)$ is not to be confused with the $g(k)$ of the Erdös-Selfridge problem of Section 7.2.

[3] Later, 8 more unique representations for $d_4 = 82562$ were found.

not be represented as the sum of 4 cubes. As a final test, we ran our algorithm for $0 < d_3, d_4 < 10^7$ and $max(|x|, |y|, |z|) \leq 1600$ and $0 \leq w < 215$, and again found no values of d_4 in this range which could not be represented as a sum of 4 cubes. To ensure that no errors occurred during our computations, the complete list of representations for $d_4 \equiv \pm 4 \pmod 9$, stored as tuples (x,y,z,w), were output to a file and double-checked on an Alpha workstation using 64-bit signed integer arithmetic. Only those values of $d_4 \equiv \pm 4 \pmod 9$ need be verified since it is known that all integers $\not\equiv \pm 4 \pmod 9$ can be represented as a sum of 4 cubes [Guy94, PV92].

To further extend this search, another program was written which utilized two separate bit arrays to determine those values which can be represented as the sum of 4 cubes. Initially, both arrays were initialized to 0. In the first array, numbers from 0 to 2×10^8 which could be represented as the sum of 3 cubes were identified by a 1-bit in the array. These values were determined by running a triple nested loop for $max(|x|, |y|, |z|) < 1600$. The second array was used to store those values which could be represented using 4 cubes by adding or subtracting values of w^3 for $0 \leq |w| < 280$. Again, the result of this experiment was that no number d_4 was found which could not be represented as the sum of 4 cubes (positive or negative) in the range $0 \leq |d_4| < 2 \times 10^8$.

We were also able to determine that in the range $0 \leq d_4 \leq 2 \times 10^8$, $max(|x|, |y|, |z|) < 1439$ and $0 \leq |w| < 250$, the only number which could not be represented was 135598172. In some sense, this number represents the integer most difficult to represent using our algorithm over this range. As a matter of interest we determined that 135598172 can be represented in at least two ways, namely (-1204,-1027,1439,-250) and (1061,1031,-1288,-262).

These computations were implemented using double-precision floating point with approximately 52 binary bits of precision. Our program took approximately 15 hours of CPU time on a SPARCstation 10/51 workstation and required 50 megabytes of main memory. The majority of computation time was spent in indexing the elements of the

bit array. These results were partially confirmed using 64-bit integer math on an Alpha AXP3300LX workstation over the interval $0 \leq d_4 \leq 10^8$, the maximum possible due to the memory constraints on the particular Alpha workstation at our disposal.

Because there is heuristic evidence to suggest that it is easier to represent larger integers as the sum of 4 cubes, some of which we shall present in the following section, the lack of difficult values of d_4 in the range $0 \leq |d_4| < 2 \times 10^8$ suggests that either these difficult values are very rare, or all integers can in fact be represented as a sum of 4 cubes.

After conducting this investigation, we became aware of some earlier work in this area from a reference in [Rib88b, p. 240,434]. In 1980, F. Romani [Rom82] determined that all numbers d_4 in the range $0 \leq |d_4| \leq 10^5$ can be represented as the sum of 4 positive or negative cubes. These computations led him to conjecture that four positive or negative cubes suffice to represent any integer.

Positive Cubes

Although as many as 9 positive cubes are required to represent every integer, and hence $g(3) = 9$, Dickson [Dic39] proved that only 23 and 239 require 9 cubes and all other integers can be represented as the sum of 8 positive cubes. From such work was identified the quantity $G(k)$, a more fundamental number than $g(k)$, defined to be the the least positive integer such that all sufficiently large integers can be represented as a sum of at most $G(k)$ kth powers of positive integers. In other words, $G(k)$ kth powers suffice to represent all integers greater than some bound b. For the case $k = 3$, the best known result is $4 \leq G(k) \leq 7$ [Rib88b, p. 240]. Clearly, $G(k) \geq 4$ since numbers of the form $\pm 4 \pmod 9$ require at least 4 cubes in their representations. In 1942, Linnik (see [Wat51]) proved that every sufficiently large integer can be expressed as the sum of seven positive integral cubes, implying $G(K) \leq 7$.

A.E. Western [Wes26] made extensive computations with regards to numbers representable as the sum of four or five positive cubes and made the following predictions:

1. the number of numbers not representable as the sum of five cubes is finite, and the greatest of them is less than 2×10^6.

2. $G(3) = 4$, and the greatest number not representable as the sum of 4 cubes lies between 10^{12} and 10^{14}.

Western based these predictions on extensive computations of whether numbers congruent to 4 (mod 9) up to 810,000 can be represented as a sum of 5 cubes. Using Western's terminology, we refer to an integer which cannot be represented using five cubes as a non-five-cube. His computations showed that the ratio, denoted by y, of the number of non-five-cube numbers congruent to 4 (mod 9) which lie between $x - d$ and $x + d$ to the total number in the range, which is $2d/9$, suggest the empirical formula

$$\log \log y^{-1} \cong \alpha + \beta \log x.$$

Using the method of least squares, Western determined values for α and β which he then used to make his predictions.

In 1981, Bohman and Fröberg [BF81] conducted a numerical investigation of Waring's problem for cubes using a Univac 1100/80 computer. Their program, written in Fortran and assembly code, found all numbers representable as the sum of not more than 4 cubes in selected ranges up to 4×10^{11}. Their basic algorithm selected all possible pairs of cubes (including zero) such that their sum was in the desired range, and then combined these pairs to find all numbers representable as the sum of not more than 4 cubes. Bohman and Fröberg [BF81] conjectured that

> *All numbers $N > M$ where $M \cong 10^{13}$ can be expressed as sums of not more than 4 positive cubes. This is equivalent to $G(3) = 4$.*

They also expressed the desire that these computations be extended in the future, stating:

> *Our impression is that a sample of 10-100 million numbers close to 10^{12} or possibly 2×10^{12} would be desirable.*

Using a very simple C-language program, we were able to examine the representations of all positive integers up to 343 million. This required two bit-arrays of size 343 million, occupying the better part of 90 megabytes of main memory. A count of the number of integers whose representations require m cubes was maintained, denoted by n_m, as well as the value of the largest such integer requiring m cubes, denoted by b_m. The results obtained from this search, listed in Table 7.8, are identical to those of Bohman and Fröberg. Although this by no means constitutes a proof for any integers beyond 343 million, many of the values determined by this search most likely represent the truth for all integers.

We note that the value of $b_6 = 1290740$ agrees very closely with the prediction, made by Western in 1926, that the last non-five-cube occurs before 2×10^6. We also point out that the likelihood of the true value of b_6 being greater than 3.43×10^8 is very small indeed. Thus, it is extremely likely that $b_6 = 1290740$ is the last example of an integer requiring 6 positive cubes.

m	n_m	b_m
9	2	239
8	15	454
7	121	8042
6	3922	1290740
5	$\geq$19189420	$\geq$342999886

Table 7.8: Numbers d requiring m cubes for $0 \leq d < 3.43 \times 10^8$

To extend the numerical investigation of Bohman and Fröberg, we examined various intervals of width 10^8 starting at 10^{10}. Our previous algorithm which utilized a triple loop, followed by the "sprinkling" of a small fourth cube was not suitable because far too few numbers of magnitude 10^{12} were representable in this fashion. This posed a problem, since four nested loops on x, y, z and w would require a tremendous amount of computation. To reduce the amount of computation required, we made use of the simple

fact that by enforcing the ordering $x \geq y \geq z \geq w \geq 0$, we can bound the values of x, y, z and w. We now define a lower limit (inclusive) of our search interval denoted l, and an upper limit (non-inclusive) of our search interval denoted u. The region which we shall examine will be of size 10^8. We see that we can bound the value of x by noting that

$$l \leq d_4 = x^3 + y^3 + z^3 + w^3 \leq u$$

from which we can bound x from below by

$$l \leq x^3 + x^3 + x^3 + x^3$$

$$\frac{l}{4} \leq x^3$$

$$\sqrt[3]{\frac{l}{4}} \leq x.$$

And from above by

$$x^3 \leq u$$

$$x \leq \sqrt[3]{u}.$$

In this manner we restrict the range of our x loop to

$$\sqrt[3]{\frac{l}{4}} \leq x \leq \sqrt[3]{u}.$$

Similarly for y, z, and w we can make use of the fact that if $l \geq x^3$ then

$$\sqrt[3]{\frac{l - x^3}{3}} \leq y \leq x$$

else

$$0 \leq y \leq x.$$

And, if $l \geq x^3 + y^3$ then

$$\sqrt[3]{\frac{l - x^3 - y^3}{2}} \leq z \leq y$$

else

$$0 \leq z \leq y.$$

And, if $l \geq x^3 + y^3 + z^3$ then

$$\sqrt[3]{l - x^3 - y^3 - z^3} \leq w \leq z$$

else

$$0 \leq w \leq z.$$

Another optimization made was to test at all times that the current partial sum $(x^3 + y^3 + z^3 + w^3)$ is less than the value of u. If the value of u is exceeded the loop is exited immediately without performing any additional computation.

Combining these techniques, we examined the various regions listed in Table 7.9. As an indication of the amount of computation required, the region of size 10^8 starting at $l = 2 \times 10^{12}$ consumed 1711 minutes of CPU time on a DEC Alpha AXP3300LX computer. As predicted by Western, and Bohman and Fröberg, the non-4-cubes appear to disappear at around 10^{13}. The largest non-4-cube we encountered in our search was 1600061431756.

l (lower limit)	*count*
1×10^{10}	201048
2×10^{10}	82721
3×10^{10}	46052
4×10^{10}	29147
5×10^{10}	19763
6×10^{10}	14314
7×10^{10}	10530
8×10^{10}	8375
9×10^{10}	6528
1×10^{11}	5182
2×10^{11}	1020
3×10^{11}	302
4×10^{11}	156
5×10^{11}	61
6×10^{11}	41
7×10^{11}	25
8×10^{11}	14
9×10^{11}	5
1×10^{12}	7
1.1×10^{12}	5
1.2×10^{12}	1
1.3×10^{12}	3
1.4×10^{12}	2
1.5×10^{12}	1
1.6×10^{12}	2
1.7×10^{12}	0
1.8×10^{12}	0
1.9×10^{12}	0
2.0×10^{12}	0

Table 7.9: Non-4-cubes in selected ranges of size 10^8

Chapter 8

Conclusion

This chapter discusses the lessons learned from the design and development of the MSSU system, possible enhancements to MSSU, and a brief discussion of what possible sieve designs may in the future succeed MSSU. We also discuss some possible applications for MSSU and future sieves.

8.1 A Summary

In this thesis we have discussed the design and development of the hardware and software components of a very high speed electronic sieve device, named MSSU. This sieve has proven to be a valuable tool in the investigation of various significant problems in number theory. In the introductory chapter we stated that much of the early development of sieves was motivated by the problem of factoring, but that recent computer algorithms have made factoring by sieve obsolete. In Chapters 6 and 7 we attempted to present a large variety of modern-day problems and applications for which sieves are ideally suited. Many of these problems are intractable on present-day general purpose computers.

The MSSU is the first sieve to employ parallel computation on a large scale. The software system is the first to feature methods for parallelizing a single sieve problem across

multiple sieve units. The design of the hardware and software enable many problems to achieve nearly full utilization of the underlying 32 parallel sieve units. The overall speed of the MSSU is at least an order of magnitude greater than any other sieve known to exist — several orders of magnitude faster than the OASiS sieve.

The use of a dedicated control processor for on-board filtering has resulted in significant performance advantages over previous sieves which used non-dedicated hosts for filtering. Reliability of the hardware and software has resulted in over three years of trouble-free operation, with many thousands of solutions generated by MSSU being verified by other sieves and various computer programs.

8.2 Improving the MSSU

As with any device designed for computation, there is inevitably a desire for greater speed. A number of sieving applications were identified, which, when executed on the MSSU, were filter-bound and resulted in relatively low canvass rates. One such problem was the search for $A'(14)$ (see Section 7.1.1), in which the need to perform frequent pseudoprime tests on possible prime twins or prime triples slowed down sieving considerably. The performance of the MSSU control processor is such that approximately 12 (strong) pseudoprime tests can be performed per second on random 128-bit integers. Clearly, filter-bound problems would benefit from a significant boost in the performance of the control processor. Although faster versions of the Motorola MC68332/68331 microprocessors are now available, with enhanced clock rates of 25MHz versus 16.78MHz, a more significant speed increase is desired.

In the near future, two new products from Motorola may be suitable for upgrading the MSSU control processor. The first is a line of more powerful processors based on the CPU32 family of embedded controllers. A number of enhanced products are currently available, including the MC68349 "Dragon I" microprocessor which features a CPU030 core, with full 32-bit external data paths and a 1-Kbyte instruction cache. As of yet,

no member of the CPU030 family has been announced featuring the QSPI hi-speed synchronous serial interface, found so essential in the development of the MSSU. Newer products based on a higher performance, instruction compatible CPU core, similar to the 68040, are planned. A second possibility is a microprocessor from the newer PowerPC family of MPC500 embedded control processors. According to Motorola literature, the MPC500 offers up to ten times the performance of CPU32 processors. Although this product is very promising, until more powerful PowerPC-based controllers are available and the PowerPC product line has had time to evolve, it would be premature to re-write and port SSUMon for the PowerPC architecture.

In retrospect, the decision to implement the control processor software in assembler was made out of the necessity to "wring out" the last drop of performance from the CPU32. To this end, SSUMon was very successful. Nevertheless, if the performance of the control processor were to be significantly increased, the conversion of SSUMon to a high-level programming language like C should be made. It is our opinion that the performance benefits of coding in hand-optimized assembler are outweighed by the disadvantages of non-portability and difficulty of software maintenance. In addition, should a RISC architecture (such as the PowerPC) be used, the difficulty in writing and optimizing assembler code would be reason enough to encourage a programmer to employ a high-level language compiler.

Another possible design improvement in control processor performance could be achieved by using multiple control processors. One possibility would be to provide one control processor for each of the two array boards (16 SSU devices) assigned the task of processing solutions, and to employ a third to oversee operation of these two and also handle communication with the host system. Unfortunately, such a scheme would be very complicated to design and implement, not to mention very difficult to debug.

A number of possible software enhancements would be desirable to improve the flexibility and user-friendliness of the MSSU system. One feature which may be useful some-

time in the future is a solution counting mode. It is possible that some future application may require that the number of solutions in a specific range be computed without actually outputting each solution. This capability has been provided in the SSUMon control software, but no such capability has been programmed in the Unix host software. Another desirable software feature would be automatic assignment of partition congruences when partitioning a single variable sieve problem. Such a feature could assign the optimal set of partition congruences to ensure the highest canvass rates. Currently, these congruences must be assigned manually.

8.3 A Successor to the MSSU

In Chapter 1 we defined the term "hardware sieving rate" as a measure of sieve performance. As we saw, the hardware sieving rate is a function of the clock rate (or shift rate) multiplied by the number of solution taps. Other factors, such as the time to transfer a trial counter value to an external device, can have an impact on overall system performance. The MSSU design proved to be very successful in reducing the amount of overhead in transferring the solution values from the trial counter to the filter mechanism by physically locating a dedicated control processor very near the sieve devices and using high speed serial communication links. Future sieves may consider even higher speed links, or perhaps parallel transfer of solution counter values, while opting to retain a lower speed serial link for non-time-critical operations, such as reading and writing ring contents.

8.3.1 More Hardware Rings

Another very important factor in the overall performance of a sieve device is the hardware support of moduli required by the problem. A large number of hardware supported moduli is extremely desirable. It was found that the rather limited number of rings on the MSSU

often resulted in sieve problems becoming filter-bound, as the process of filtering numerous solution candidates resulted in a bottleneck. As we saw for the pseudocubes problem in Section 7.3, the fast virtual ring capability of MSSU was quite successful in alleviating this problem. Nevertheless, for these types of problems it is important to eliminate as many solution candidates as possible by specifying hardware congruences, thereby shifting the load of rejecting solution candidates from the comparatively slow control processor to the very fast sieve devices.

For example, consider the problem of computing $A'(14)$ described in Section 7.1. Of all of the sieve problems discussed in Chapters 6 and 7, this was the most dense, having a very low canvass rate of approximately 14.5 million per second. If an additional 20 hardware rings corresponding to the primes 127 through 229 were available, then the number of solution candidates requiring processing by the control processor would be reduced to approximately 16% of the rate observed on MSSU, since

$$\prod_{i=31}^{50} \frac{p_i - 15}{p_i} \doteq 0.15823.$$

This would reduce the number of interrupts handled by the control processor, and the number of solutions which need to be passed to the virtual ring filter. If an updated virtual ring filter was provided for the next 16 primes (233 through 317), we would expect to reduce the number of solution candidates by an additional 60%, as

$$\prod_{1=51}^{66} \frac{p_i - 15}{p_i} \doteq 0.40451.$$

Assuming the sieve was still filter-bound after these changes, we would expect that reducing the amount of solution candidates by this amount would result in approximately a 250% increase in the canvass rate.

It was our experience with MSSU that often single-residue optimization was performed on hardware supported moduli, resulting in higher solution rates, and increasing the processing load on the control processor. For example, consider the pseudosquares

problem. Here, single residue optimization is performed on the congruences

$$L_p \equiv 1 \pmod 8$$

$$L_p \equiv 1 \pmod 3,$$

resulting in the first and second rings of the SSU device(s) being loaded with all admissible residues. Effectively, the SSU is now acting like a sieve with 28 rings, further reducing the ability of the sieve hardware to eliminate unwanted solution candidates. In fact the problem is exacerbated when a 180-chip partitioning is used and the moduli 8, 3, 5, 7, 11 and 13 are "optimized-out", reducing the useful complement of rings to 24. Variable-size rings and reconfigurable sieve architectures offer the promise of eliminating this inefficiency by implementing only those rings which are required by the problem at hand. Thus, theoretically, on-chip resources, such as flip-flops or RAM, used to implement rings which are to be optimized-out could be better utilized in implementing a different ring whose modulus would otherwise not be available.

8.3.2 Greater Parallelism

As we saw in Chapters 6 and 7, extremely high canvass rates were achieved on a number of problems by partitioning single-variable sieve problems to execute on parallel sieve devices. As well, the parallel architecture of MSSU was used to implement a multi-variable sieving capability useful for solving some very difficult diophantine problems. This ability to exploit parallelism is the major advantage of the MSSU over all other existing sieve systems. Clearly, employing parallelism on a greater scale, perhaps employing several hundred sieve chips, is a challenge for future sieve designers. The difficulty of such a design would be in providing a sufficiently powerful control processor (or multiple processors) and a suitable switching network to handle the increased data bandwidth necessary to accommodate a large number of sieve devices.

8.3.3 Newer VLSI Technologies

The SSU VLSI sieve devices were designed using gate array technology which is fast becoming obsolete. The SSU sieve chips were implemented with a 2μm CMOS double-metal process on a 1 cm^2 die. Patterson [Pat91, pp. 150–158] utilized 9766 out of a possible 10013 available gates, with the majority of that circuitry occupied by 1675 D-type flip-flops used to implement the shift-register rings. Shift-register rings corresponding to the next 16 primes would have required an additional 2634 flip-flops.

Modern (1995) gate array devices offer much higher densities and speed. Using conservative estimates, VLSI sieve devices similar to the SSU could be manufactured using 0.8μm CMOS gate arrays with clock rates of approximately 50MHz (versus 24MHz for SSU), with 16 solution taps (versus 8 on the SSU), and hardware rings corresponding to the first 40 to 50 primes. Such a device would have 4 times the hardware sieving rate of the SSU devices and offer increased flexibility with the added hardware rings.

Custom gate arrays such as the SSU are extremely expensive to fabricate, costing tens of thousands of dollars in fixed costs in addition to per unit production costs. In large quantities, these one-time fees can be amortized, resulting in a relatively low unit-cost. Due to the relatively low number of sieve devices required to implement a new electronic number sieve, a more economical solution would be the use of one-time programmable devices. Field Programmable Gate Arrays (FPGAs) offer the advantage of being user-programmable, at somewhat higher per unit costs than custom gate arrays. Gate densities of FPGAs, although significantly lower than custom devices, have increased to the point where design of a single-chip sieve device is now feasible. And whereas custom gate arrays are fabricated in batches, requiring a commitment to purchase some minimum quantity of parts, FPGAs can be programmed in any quantity, as needed.

Some Field Programmable Gate Arrays, most notably Xilinx parts, are available which offer re-programmability. These devices are used in the SPLASH systolic processor used by Bronson and Buell [BB93] to implement their sieve. In the near future, the densities

of re-programmable FPGAs will make it feasible to implement a sieve on a single chip. Theoretically, a number of different hardware configurations could be supported utilizing this re-programmability feature. For example, one mode could feature n solution taps with $2r$ hardware rings for maximum speed on sparse sieves, while another mode could offer $2n$ solution taps with only r rings, useful for denser, filter-bound sieves. Another possibility would be to offer a mode where all the hardware supported moduli are p^2, for primes p. This would be a useful configuration for computing values of the Erdös-Selfridge function.

Conceivably, a single-chip version of the OASiS sieve could be built which would offer the variable-size ring feature regardless of whether or not it is implemented in re-programmable logic. Some programmable devices are being manufactured with large on-board memories, however, a design along the lines of OASiS would ideally require many smaller blocks of memory in order to implement 16 or so variable-size rings. A hybrid design employing a single FPGA containing the bulk of the sieve functions with external RAM devices may also be feasible. The major limitations of this approach are the requirement for a large number of external input/output pins on the FPGA for interfacing to multiple RAM devices with wide data paths, and the increased size of the resulting sieve.

At present the FPGA-based sieves of Bronson and Buell have suffered from a lack of hardware rings, requiring that an attached workstation process a large number of solution interrupts and perform virtual ring emulation. This is not computationally burdensome unless a filter condition is specified, as in the pseudosquares problem. However, the major limitation with sieves of this type is that they implement a single sieve unit. Unless many such sieves are operated in parallel, it is unlikely that any single sieve unit will achieve the canvass rates necessary to surpass the performance of the MSSU.

8.4 New Applications

The success of the MSSU in computing several new solutions to the sums of three cubes problem (see Section 7.5.1) demonstrates the power of multi-variable sieving. Applications of this type are ideally suited to the parallel architecture of the MSSU. As well, these types of problems parallelize across multiple sieve units, allowing two MSSU sieves to be deployed on a single problem. It is our belief that this method holds great promise for solving a number of very difficult diophantine problems.

In addition to extending the search for sums of three cubes for both $m > 100000$ and $x > 3 \times 10^{10}$, it is possible that further analysis, perhaps for specific values of d, may lead to a focused search along a plane other than $m = y + z$. Another variation of this problem which has received some attention (see [Guy94, p. 151],[LRB69]) is determining which integers d can be expressed as

$$d = x^3 + y^3 + 2z^3.$$

Another possible application of multi-variable sieving is a problem suggested by Melvyn J. Knight and analyzed by Bremner, Guy and Nowakowski [BGN93].

Appendix A

Summary of SSUMon Commands

CK - check

Format: CK* | CK < # >
Parameters: chip #
Outputs: DF | DP

Description: Checks whether the current contents of the rings and counters of the specified chips are consistent with the initial ring contents. If * is specified, all enabled chips will be tested. A DF signifies that the counter and ring values indicate a hardware error has occurred, or the sieve chip(s) were loaded incorrectly. A DP signifies that the consistency check was successful.

Error Messages:

```
E-CHECKPOINT DID NOT VERIFY
E-INVALID PARAMETERS
E-NO CHIPS CURRENTLY ENABLED
```

CN - count

Format: CN* | CN< # >
Parameters: chip #
Outputs: D< 32*h* >

Description: This command returns the 128-bit trial counter value COUNT. This is calculated using the COUNTM value and the value of the optimize registers.

Error Messages:

E-INVALID PARAMETERS
E-NO CHIPS CURRENTLY ENABLED

CR - counter read

Format: CR< # >
Parameters: chip #
Outputs: D< 32*h* >

Description: Outputs the counter value COUNTM for the specified chip.

Error Messages:

E-INVALID PARAMETERS
E-NO CHIPS CURRENTLY ENABLED

CW - counter write

Format: CW*< 32*h* > | CW< # >< 32*h* >
Parameters: chip #, 128-bit COUNTM value
Outputs: none

Description: Write the 128-bit counter value (COUNTM) for the specified chip(s).

Error Messages:

E-INVALID PARAMETERS
E-NO CHIPS CURRENTLY ENABLED

DU - variable dump

Format: DU< # >
Parameters: chip #
Outputs: a variable dump of sieve information

Description: Performs a variable dump having the following format:

```
CHIP: 0
H/W CNTR =000000000000  00
CNTROVF  =0000000000000000
COUNT    =00000000000000000000000000000000
COUNTM   =00000000000000000000000000000000
OPTMUL   =00000001       OPTADD   =00000000
SOLCNT   =00000000       SOLLMT   =00000000

OUTMODE  =D              PBPARM   =00000000
PREFNM   =PSQR           POSTFNM  =PSVR

ENABVEC  =0000           PIPED    =00000000
IRQLST0  =00000000       IRQLST1  =00000000
CHIPSEQ  =00000000
```

Error Messages:

E-INVALID PARAMETERS

EN - enable sieve chips

Format: EN< $8h$ >
Parameters: 32-bit enable vector
Outputs: none

Description: Enables/disables chips from entering solution mode. The 32-bit enable vector contains a binary "1" for each enabled chip, and a "0" for each disabled chip. The least significant bit corresponds to chip #0.

Error Messages:

E-INVALID PARAMETERS

EX - exit

Format: EX
Parameters: none
Outputs: none

Description: Causes sieve monitor to exit and returns execution to CPU32BUG.

Error Messages:

none

FI - filter specify

Format: FI< *name* >< *name* >
Parameters: name of pre-filter, post-filter
Outputs: none

Description: Specifies the filters to be used for filtering solutions.

Error Messages:

E-INVALID PARAMETERS

GO - go

Format: `GO`
Parameters: none
Outputs: none

Description: All enabled chips enter solution mode and solutions are displayed as they are encountered.

Error Messages:
none

HA|! - halt

Format: `HA | !`
Parameters: none
Outputs: none

Description: Causes sieve to exit solution mode and await further commands.

Error Messages:
none

LO - load

Format: `LO`
Parameters: none
Outputs: none

Description: Loads the hardware rings and counters of all enabled chips using the `RINGDAT` and `COUNTM` variables.

Error Messages:
`E-NO CHIPS CURRENTLY ENABLED`

OM - output mode

Format: `OM<H | D | S>`
Parameters: output mode type
Outputs: none

Description: Specifies the output mode which will be used for the output of solutions. Must be one of the following:

`D` – decimal
`H` – hexadecimal
`S` – solution counting mode, no output

Error Messages:

`E-INVALID PARAMETERS`

OR - read optimize registers

Format: `OR`$< 2h >$
Parameters: chip #
Outputs: `D`$< 8h >< 8h >$ (multiply, add)

Description: Reads the single-residue optimization values for the specified chip. The multiply value is output first, followed by the add value.

Error Messages:

`E-INVALID PARAMETERS`

OW - write optimize registers

Format: OW*< $8h$ >< $8h$ > | OW< # >< $8h$ >< $8h$ >
Parameters: chip #, multiply value, add value
Outputs: none

Description: Writes the given multiply and add values to the specified chip.

Error Messages:

E-INVALID PARAMETERS

PP - problem parameter

Format: PP< $8h$ >
Parameters: 32-bit signed integer parameter value
Outputs: none

Description: This command allows 32-bits of problem specific data to be specified.

Error Messages:

none

RD - ring dump

Format: RD< # >
Parameters: chip #
Outputs: D< $420h$ >

Description: This command outputs the contents of the 1675-bit RING buffer of the specified chip.

Error Messages:

E-INVALID PARAMETERS

RE - reset

Format: RE
Parameters: none
Outputs: none

Description: This command performs a hardware reset of all sieve chips. All state variables are reset to their defaults.

Error Messages:
none

RR - ring read

Format: RR< $2h$ >
Parameters: chip #
Outputs: D< $420h$ >

Description: This command outputs the contents of the 1675-bit RINGDAT buffer of the specified chip.

Error Messages:
E-INVALID PARAMETERS

RW - ring write

Format: RW*< $420h$ > | RW< # >< $420h$ >
Parameters: chip #, 1675 bits of ring data
Outputs: none

Description: This command sets the contents of the 1675-bit RINGDAT buffer to the given value for the specified chip.

Error Messages:
E-INVALID PARAMETERS

SR - read solution registers

Format: SR< # >
Parameters: chip #
Outputs: D< 8*h* >< 8*h* > (SOLCNT,SOLLMT)

Description: This command reads the solution count (SOLCNT) and solution limit (SOLLMT) values for the specified chip.

Error Messages:

E-INVALID PARAMETERS

ST - return chip status

Format: ST
Parameters: none
Outputs: D< 8*h* > (ENABVEC)

Description: Displays the 32-bit enable vector value.

Error Messages:

none

SW - write solution registers

Format: SW*< 8*h* >< 8*h* > | SW< # >< 8*h* >< 8*h* >
Parameters: chip #, 32-bit SOLCNT, 32-bit SOLLMT
Outputs: none

Description: This command sets the contents of the optimize registers for the specified chip.

Error Messages:

E-INVALID PARAMETERS

UL - unload

Format: UL
Parameters: none
Outputs: none

Description: This command unloads the contents of the hardware rings and counters of all enabled chips and updates the values of the RING and COUNTM variables.

Error Messages:
none

VR - virtual ring read

Format: VR< # >
Parameters: chip #
Outputs: D< 882*h* >

Description: This command outputs the contents of the 3524-bit virtual ring data of the specified chip.

Error Messages:
E-INVALID PARAMETERS

VW - ring write

Format: VW*< 882*h* > | VW< # >< 882*h* >
Parameters: chip #, 3524 bits of ring data
Outputs: none

Description: This command sets the contents of the 3524-bit virtual ring data to the given value for the specified chip.

Error Messages:
E-INVALID PARAMETERS

Appendix B

SSUMon Filter Algorithms

B.1 The Perfect Square Filter – PSQR

The integer square-root of an integer X is the largest positive integer r such that $r^2 \leq X$. More precisely:

Definition B.1.1 *We define the* **square-root** *of a positive integer value X to be that integer value r such that*

$$0 \leq r^2 \leq X < (r+1)^2.$$

Most software math packages utilize a version of Newton's method to evaluate square roots. This technique finds the zero of a function $f(x)$ by starting with an initial approximation and refining that approximation using an iterative process. The iteration used to find an approximation x_{i+1} of the zero of $f(x_i)$ is

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

where $f'(x_i)$ is the derivative of $f(x_i)$. For computing the square root of A, the resulting iteration is

$$x_0 = A,$$

$$x_{i+1} = x_i - \frac{{x_i}^2 - A}{2x_i} = \frac{1}{2}(x_i + \frac{A}{x_i})$$

where A is the radicand, and the x_i are successive approximations to $\sqrt{A}$. Newton's method is said to have "quadratic convergence", meaning that the number of digits of precision doubles with each iteration [Con65], however this is an asymptotic property and dependent on the initial approximation used.

Although, Newton's method is generally considered to be the fastest square root algorithm asymptotically, it has several disadvantages. First, running time of the algorithm is dependent on the input and the accuracy of the initial approximation. Second, some stopping condition must be tested after each iteration to determine whether the iteration has converged to a solution. Most importantly, however, Newton's method requires a relatively expensive divide operation per iteration.

Since division is more complex than multiplication, addition or subtraction, it will run more slowly [PH90, p. A-26]. Even on the most modern of RISC processors, such as the 64-bit MIPS R10000 processor, the relative costs of an integer add operation versus a multiply operation are 1:5, while the relative cost of an integer multiply operation versus a divide are 5:35 [MIP94]. The relative costs of these operations is much higher for so-called "multi-precise" operations in which operands or return values are in excess of the hardware supported word-lengths. For example, for the 128-bit integers required by the SSUMON perfect square filter, the cost of a 128-bit by 64-bit divide operation using a bit-at-a-time shift and subtract algorithm is of the same order as performing an entire 128-bit square root operation by more direct means. Other variants of the iterative method can be used to replace the division operation by three multiplication operations, incurring costs roughly on the order of a single division operation [PH90, p. A-26]. Others eliminate multiply or divides altogether at the expense of their corresponding rate of convergence.

Algorithm B.1 *32-bit square root algorithm with multiplies*

```
FUNCTION Sqrt(X) RETURNS Y ;
INPUT X : integer ;
OUTPUT Y : integer ;

xr : integer ;
q2 : integer ;

xr ← 0 ;
q2 ← 0x8000 ; { hexadecimal constant}

DO

    IF ((xr + q2) * (xr + q2) ≤ x) THEN

        xr ← xr + q2 ;

    q2 ← lsr(q2) ;

WHILE (q2 ≠ 0) ;

RETURN (xr) ;
```

End of Algorithm

The perfect square filter of SSUMon `PSQR` uses a square root algorithm published by Peter [Pet90] [1]. It closely resembles the "classical" method of root extraction known in ancient times [Smi73, pp. 144–148]. One binary bit of the solution is generated per iteration using only basic operations: assignment, comparison and branch, addition, subtraction, and logical (single binary bit) right shifts. We shall denote a single logical right shift of a variable x using the symbol $lsr(x)$. The pseudocode for a 32-bit version of the algorithm is presented in Algorithm B.1. Making use of the fact that $(a + b)^2 = (a^2 + 2ab + b^2)$, Peter was able to produce an optimized version of this algorithm shown in Algorithm B.2. It is shown here without proof, however, in the next section we will develop and prove a very similar algorithm for the fast computation of cube-roots.

[1] We would like to thank Volker Grupe [Gru94] for bringing this algorithm to our attention.

Algorithm B.2 *32-bit square root algorithm without multiplies*

```
FUNCTION Sqrt(X) RETURNS Y ;
INPUT X : integer ;
OUTPUT Y : integer ;

xr : integer ;
q2 : integer ;
tmp : integer ;

xr ← 0 ;
q2 ← 0x40000000 ; { hexadecimal constant}

DO

    tmp ← xr + q2 ;
    IF (tmp ≤ X) THEN

        X ← X − tmp ;
        xr ← lsr(xr) ;
        xr ← xr + q2 ;

    ELSE
        xr ← lsr(xr) ;

    q2 ← lsr(lsr(q2)) ;

WHILE (q2 ≠ 0) ;

RETURN (xr) ;
```

End of Algorithm

The obvious manner in which this algorithm can be used to implement a perfect square filter is to first compute the integer square root, square that value, and then compare it to the original value. If the values are identical, then the number is a perfect square. As it turns out, it is not necessary to perform the additional computation of squaring since the value of the local variable x holds the value of $X - r^2$ and it suffices to test if $x = 0$ to identify whether or not X is a perfect square.

B.2 The Perfect Cube Filter – PCUB

Similar to the square-root algorithm above, an integer cube-root algorithm exists which computes one bit of the solution per iteration, and consists of only simple operations. The cube-root algorithm is somewhat more complicated than the one used to determine square-roots.

Before we can proceed any further, we must define precisely what we mean by the cube-root function:

Definition B.2.1 *We define the* **integer cube-root** r *of a positive integer value* X *to be that value of* r *such that*

$$0 \leq r^3 \leq X < (r+1)^3.$$

To introduce and explain the derivation of the algorithm, we shall describe two versions of the cube-root algorithm. The first contains a number of multiplication operations but is very simple to prove correct. We shall then perform a sequence of program transformations to derive an equivalent algorithm using only the simple operations of assignment, comparison and branch, addition, subtraction and logical right shifts. We shall denote the logical shift right function by $lsr(x)$.

B.2.1 Description of the Initial Algorithm

Algorithm B.3 is a pseudocode implementation of the cube-root algorithm. It computes one bit of the solution per iteration. The variable p_1 acts as a *scan-bit*, representing the position of the solution bit currently being tested. The **IF** statement determines whether the current bit position represented by the scan-bit should be a "0" or a "1". The variable r holds the partial integer root. When the algorithm completes, the variable r will contain the solution, namely, the integer cube-root of X.

So that we can discuss the values of a given variable v at various times throughout the execution of this algorithm, we define the variable i which serves no purpose other than to identify the current iteration of the algorithm. So that we can uniquely identify the value of any variable at any given iteration of the algorithm, we shall use the symbol $v^{(i)}$ to denote the value of variable v at iteration i. To avoid ambiguity, we enforce a "single-assignment rule" whereby a variable can be assigned a value only once per iteration. The value of a variable is undefined until an assignment operation has taken place for that iteration. If we require two operations to be performed on a variable during an iteration, we can either combine these operations in one assignment statement or introduce a temporary variable. For example, consider the following code segment:

$$x \leftarrow x + 1 \ ;$$
$$x \leftarrow lsr(x) \ ;$$

This can be re-written as:

$$i \leftarrow i + 1 \ ;$$
$$x^{(i)} \leftarrow lsr(x^{(i-1)} + 1) \ ;$$

Or:

$$i \leftarrow i + 1 \ ;$$
$$y^{(i)} \leftarrow x^{(i-1)} + 1 \ ;$$
$$x^{(i)} \leftarrow lsr(y^{(i)} + 1) \ ;$$

Note, however, that this convention of labeling variables by iteration number does not change the semantics of the algorithm, but merely allows us to refer to the value of a variable in a precise manner.

Algorithm B.3 *cube root for arbitrary-size integers with multiply operations*

FUNCTION $cbrt(X)$ **RETURNS** arbitrary_integer ;
INPUT X : arbitrary_integer ;

i : timestamp ;
r : arbitrary_integer ;
p_1 : arbitrary_integer ;
$temp$: arbitrary_integer ;
x : arbitrary_integer ;

$i \leftarrow 0$;
$r^{(0)} \leftarrow 0$;
$p_1^{(0)} \leftarrow 2^{(\lceil bits(X)/3 \rceil)}$; { highest possible result bit }
$x^{(0)} \leftarrow X$;

DO

$temp^{(i)} \leftarrow (r^{(i)} + p_1^{(i)}) * (r^{(i)} + p_1^{(i)}) * (r^{(i)} + p_1^{(i)})$;
$i \leftarrow i + 1$;
IF $(temp^{(i-1)} \leq x^{(i-1)})$ **THEN**

$r^{(i)} \leftarrow r^{(i-1)} + p_1^{(i-1)}$; { output 1-bit }

ELSE

$r^{(i)} \leftarrow r^{(i-1)}$; { output 0-bit }

ENDIF
$x^{(i)} \leftarrow x^{(i-1)}$;
$p_1^{(i)} \leftarrow lsr(p_1^{(i-1)})$;

WHILE $(p_1^{(i)} \neq 0)$;
RETURN $(r^{(i)})$;

End of Algorithm

The cube-root algorithm does not enforce a limit on the word-size of the input X. Since we want to prove that this algorithm will work for arbitrary-size integers, we shall merely identify all variables as being of type "arbitrary_integer". For now, we shall assume that the function $bits(Y)$ simply returns an integer value h such that $Y < 2^{h+1}$. This

roughly corresponds to the number of bits required to represent the value X. To ensure that the algorithm tests all possible bit-positions of the solution we initialize the scan-bit variable to $2^{(\lceil bits(X)/3 \rceil)}$. Note, the algorithm will work correctly if our initial value for the scan-bit variable is too large. However, we want the value of $bits(X)$ to be as low as possible to avoid unnecessary work in our algorithm. When the algorithm is programmed for fixed-length values of X we shall replace the value of $2^{(\lceil bits(X)/3 \rceil)}$. by a constant which is a function of the word-size.

Theorem B.1 *Algorithm B.3 correctly computes the integer cube root of X.*

Proof: We shall prove that the algorithm is correct using induction on the number of binary bits of the solution which are output correctly.

For our initial case, we shall prove that Algorithm B.3 outputs the first binary digit correctly for any integer X. When the algorithm performs the first iteration, we have the following two cases:

(A.) $temp^{(0)} \leq X$, thus $r^{(1)} \leftarrow r^{(0)} + {p_1}^{(0)}$ (a "1" bit is output).

(B.) $temp^{(0)} > X$, thus $r^{(1)} \leftarrow r^{(0)}$ (a "0" bit is output).

Consider case (A.). By contradiction, assume that a 0-bit is output and therefore $r^{(1)} \leftarrow r^{(0)}$. Thus the algorithm will generate a solution, r, whose binary string is of the form $0y$, where y is any arbitrary string of 0's and 1's. Since $temp^{(0)} \leq X$, we know that $({p_1}^{(0)})^3 \leq X$. However, $r^3 < ({p_1}^{(0)})^3 \leq X$, and thus r cannot be a valid solution because it violates the condition that $0 \leq r^3 \leq X < (r+1)^3$ in Definition (B.2.1) and thus our original assumption must be wrong.

Consider case (B.). By contradiction, assume that a 1-bit is output and therefore $r^{(1)} \leftarrow r^{(0)} + {p_1}^{(0)}$. Since $temp^{(0)} = (r^{(0)} + {p_1}^{(0)})^3$ we must have $(r^{(1)})^3 = (temp^{(0)})^3 > X$ and $(r^{(1)})^3 > X$. However this violates the condition that r^3 cannot exceed X in Definition (B.2.1) and thus our original assumption must be wrong.

We now assume that the algorithm correctly generate the first m digits of the solution given by the binary string, z, for any arbitrary value of input X. We now prove that the algorithm will generate the correct $m+1$'th digit.

Recall that $temp^{(m)} = (r^{(m)} + {p_1}^{(m)})^3$. Again, we have two cases:

(A.) $temp^{(m-1)} \leq X$, thus $r^{(m)} \leftarrow r^{(m-1)} + {p_1}^{(m-1)}$ (a "1" bit is output).

(B.) $temp^{(m-1)} > X$, thus $r^{(m)} \leftarrow r^{(m-1)}$ (a "0" bit is output).

Consider case (A.). Since $temp^{(m-1)} \leq X$ we know that $(r^{(m)})^3 = (r^{(m-1)} + {p_1}^{(m-1)})^3 \leq X$. By contradiction, assume that a 0-bit is output and the remaining bits of the solution, r, form a string of the form $0y$, where y is any arbitrary string of 0's and 1's. This generates a solution of the form $z0y$. Since, $(r^{(m)})^3 \leq X$ we know that for some integer, q, of the form $z10\cdots0$, consisting of the prefix $z1$ followed by a string of binary 0's, that $r^3 < q^3 \leq X$. This again violates the condition that $0 \leq r^3 \leq X < (r+1)^3$ of Definition (B.2.1), and thus our original assumption must be wrong.

We now consider case (B.). By contradiction, assume that a 1-bit is output and $r^{(m)} = r^{(m-1)} + {p_1}^{(m-1)}$. Since $temp^{(m-1)} = (r^{(m-1)} + {p_1}^{(m-1)})^3 > X$, we must have $(r^{(m)})^3 > X$. However this violates the condition that r^3 cannot exceed X in Definition (B.2.1) and, once more, our original assumption must be wrong. □

B.2.2 A Sequence of Transformations

We now perform a sequence of transformations on Algorithm B.3 to derive an equivalent algorithm using only the simple operations of assignment, comparison and branch, addition, subtraction and logical right shifts.

We start by evaluating $(r+p_1)^3$,

$$(r+p_1)^3 = \underbrace{r^3}_{1} + \underbrace{{p_1}^3}_{2} + \underbrace{3r{p_1}^2}_{3} + \underbrace{3r^2p_1}_{4}.$$

We shall attempt to eliminate the multiplication operations required in computing the above four terms, in the order shown, beginning with the first term, r^3.

Define

$$tmp^{(i-1)} = 3(r^{(i-1)})^2 p_1{}^{(i-1)} + 3r(p_1{}^{(i-1)})^2 + (p_1{}^{(i-1)})^3.$$

If the **THEN** portion of the **IF** statement is executed,

$$(r^{(i)})^3 = (r^{(i-1)})^3 + tmp^{(i-1)},$$

otherwise, if the **ELSE** portion of the statement is executed,

$$(r^{(i)})^3 = (r^{(i-1)})^3.$$

Since

$$(r^{(0)})^3 = 0,$$

we have

$$(r^{(j)})^3 = \sum_{i,\ \text{THEN taken}}^{i<j} tmp^{(i)},$$

where we sum over the iterations i for which the **THEN** portion of our **IF** statement is executed. Thus, we see that by decrementing x by the value of tmp each time the **THEN** statement is executed that we eliminate the need to recompute the value of $(r^{(i)})^3$. Our updated pseudocode becomes the following:

```
DO
    tmp(i) ← 3(r(i))^2 p1(i) + 3r(i)(p1(i))^2 + (p1(i))^3 ;
    i ← i + 1 ;
    IF (tmp(i-1) ≤ x(i-1)) THEN
        x(i) ← x(i-1) − tmp(i-1) ;
        r(i) ← r(i-1) + p1(i-1) ;
    ENDIF
    p1(i) ← lsr(p1(i-1)) ;
WHILE (p1(i) ≠ 0) ;
RETURN (r(i)) ;
```

Second, we see that we can eliminate the computation of ${p_1}^3$ by defining a variable p_3 whose initial value is $2^{3(\lceil bits(X)/3 \rceil)}$, the cube of the initial value of p_1, and which is shifted right three times every iteration instead of once like p_1. We can also use p_3 as the test condition for terminating the loop because p_3 and p_1 will both equal zero at the same iteration.

DO
$tmp^{(i)} \leftarrow 3(r^{(i)})^2 {p_1}^{(i)} + 3r^{(i)}({p_1}^{(i)})^2 + {p_3}^{(i)}$;
$i \leftarrow i + 1$;
IF $(tmp^{(i-1)} \leq x^{(i-1)})$ **THEN**
$x^{(i)} \leftarrow x^{(i-1)} - tmp^{(i-1)}$;
$r^{(i)} \leftarrow r^{(i-1)} + {p_1}^{(i-1)}$;
ENDIF
${p_1}^{(i)} \leftarrow lsr({p_1}^{(i-1)})$;
${p_3}^{(i)} \leftarrow lsr(lsr(lsr({p_3}^{(i-1)})))$;
WHILE $({p_3}^{(i)} \neq 0)$;
RETURN $(r^{(i)})$;

Third, we shall attempt to eliminate the multiplications in the term $3r(p_1)^2$. First, define a variable t' which is the sum of the p_3 values in the inner loop. The initial value of t' is $t'^{(0)} = 0$.

DO

$tmp^{(i)} \leftarrow 3(r^{(i)})^2 p_1^{(i)} + 3r^{(i)}(p_1{}^{(i)})^2 + p_3{}^{(i)}$;

$i \leftarrow i + 1$;

IF $(tmp^{(i-1)} \leq x^{(i-1)})$ **THEN**

$x^{(i)} \leftarrow x^{(i-1)} - tmp^{(i-1)}$;

$t'^{(i)} \leftarrow t'^{(i-1)} + p_3{}^{(i-1)}$;

$r^{(i)} \leftarrow r^{(i-1)} + p_1{}^{(i-1)}$;

ENDIF

$p_1{}^{(i)} \leftarrow lsr(p_1{}^{(i-1)})$;

$p_3{}^{(i)} \leftarrow lsr(lsr(lsr(p_3{}^{(i-1)})))$;

WHILE $(p_3{}^{(i)} \neq 0)$;

RETURN $(r^{(i)})$;

If we denote the value of the highest order binary bit of r by the symbol $r_{(0)}$, and the kth most significant bit by $r_{(k)}$, and similarly define symbols for the value of variable t', we have at iteration i,

$$r = r_{(0)}r_{(1)}r_{(2)}\cdots p_1{}^{(i)},$$

and

$$t' = t'_{(0)}00t'_{(1)}00t'_{(2)}00\cdots p_3{}^{(i)}.$$

Immediately, we note that $r_{(i)} = t'_{(i)}$ for all values of i, thus

$$t' = r_{(0)}00r_{(1)}00r_{(2)}00\cdots p_3{}^{(i)}.$$

Furthermore, by performing two logic right shifts every iteration, we can "squeeze-out" these additional pairs of zeroes. Thus we define a variable t such that

$$t = r_{(0)}r_{(1)}r_{(2)}\cdots p_3{}^{(i)},$$

or equivalently that

$$t = r^{(i)}(p_1{}^i)^2.$$

Note that at the completion of our algorithm that

$$r_{(i)} = t_{(i)},$$

and thus we can use the value of t as the function return value.

Rewriting our algorithm to reflect this change, we now define a variable t which is the sum of the p_3 values and subjected to two logical right shifts per iteration. To adhere to our "single-assignment rule", we split these two operations into two assignment operations by introducing a temporary variable s. Remember, we must assign the value of s in the **ELSE** portion of our **IF** statement to avoid its value being undefined.

We also note that at any iteration i, the variable $t^{(i)} = r^{(i)}(p_1{}^{(i)})^2$. Since we require three times this quantity, we make use of the fact that

$$s = 4t,$$

and that

$$s^{(i)} - t^{(i)} = 3r^{(i)}(p_1{}^{(i)})^2.$$

This results in the following changes:

DO
  $tmp^{(i)} \leftarrow 3(r^{(i)})^2 p_1^{(i)} + s^{(i)} - t^{(i)} + p_3{}^{(i)}$;
  $i \leftarrow i + 1$;
  IF $(tmp^{(i-1)} \leq x^{(i-1)})$ **THEN**
    $x^{(i)} \leftarrow x^{(i-1)} - tmp^{(i-1)}$;
    $s^{(i)} \leftarrow t^{(i-1)} + p_3{}^{(i-1)}$;
    $r^{(i)} \leftarrow r^{(i-1)} + p_1{}^{(i-1)}$;
  ELSE
    $s^{(i)} \leftarrow t^{(i-1)}$;
  ENDIF
  $t^{(i)} \leftarrow lsr(lsr(s^{(i)}))$;
  $p_1{}^{(i)} \leftarrow lsr(p_1{}^{(i-1)})$;
  $p_3{}^{(i)} \leftarrow lsr(lsr(lsr(p_3{}^{(i-1)})))$;
WHILE $(p_3{}^{(i)} \neq 0)$;
RETURN $(t^{(i)})$;

Finally, we shall eliminate the computation of $3rp_1{}^2$, thus eliminating all multiplica-

tions from our algorithm. We see that if the **THEN** statement is executed that

$$(r^{(i)})^2 = (r^{(i-1)})^2 + 2r^{(i-1)}p_1{}^{(i-1)} + (p_1{}^{(i-1)})^2.$$

This yields a recurrence relation for $(r^{(i)})^2$. Since, $r^{(0)} = 0$ we have have,

$$(r^{(j)})^2 = \sum_{i,\ \text{THEN taken}}^{i<j} \left(2r^{(i)}p_1{}^{(i)} + (p_1{}^{(i)})^2\right).$$

Thus,

$$(r^{(j)})^2 p_1{}^{(j)} = \sum_{i,\ \text{THEN taken}}^{i<j} \left(2r^{(i)}p_1{}^{(i)}p_1{}^{(j)} + (p_1{}^{(i)})^2 p_1{}^{(j)}\right).$$

If $i < j$, we have

$$p_1{}^{(j)} = \frac{p_1{}^{(i)}}{2^{(j-i)}},$$

and

$$(r^{(j)})^2 p_1{}^{(j)} = \sum_{i,\ \text{THEN taken}}^{i<j} \frac{1}{2^{(j-i)}} \left(2r^{(i)}p_1{}^{(i)2} + (p_1{}^{(i)})^3\right).$$

This is equivalent to

$$(r^{(j)})^2 p_1{}^{(j)} = \sum_{i,\ \text{THEN taken}}^{i<j} \frac{1}{2^{(j-i)}} \left(s^{(i)} + p_3{}^{(i)}\right).$$

We define a variable u which will equal the value of $(r^{(i)})^2 p_1{}^{(i)}$ at iteration i. The initial value of $u^{(0)} = 0$. Each iteration, u is incremented by the value of

$$s^{(i)} + p_3{}^{(i)}.$$

At the end of every iteration we perform a single logical right shift of the contents of u to account for the shift factor of

$$\frac{1}{2^{(j-i)}}.$$

Since we are trying to eliminate the term $3(r^{(i)})^2 p_1^{(i)}$, we define a temporary variable $v = 2u$, and thus eliminate multiplication by three, making use of the fact that $3u = v + u$. This is shown in the following:

DO

$tmp^{(i)} \leftarrow u^{(i)} + v^{(i)} + s^{(i)} - t^{(i)} + p_3{}^{(i)}$;

$i \leftarrow i + 1$;

IF $(tmp^{(i-1)} \leq x^{(i-1)})$ **THEN**

$x^{(i)} \leftarrow x^{(i-1)} - tmp^{(i-1)}$;

$s^{(i)} \leftarrow t^{(i-1)} + p_3{}^{(i-1)}$;

$r^{(i)} \leftarrow r^{(i-1)} + p_1{}^{(i-1)}$;

$v^{(i)} \leftarrow u^{(i-1)} + t^{(i-1)} + t^{(i-1)} + p3^{(i-1)}$;

ELSE

$s^{(i)} \leftarrow t^{(i-1)}$;

$v^{(i)} \leftarrow u^{(i-1)}$;

ENDIF

$t^{(i)} \leftarrow lsr(lsr(s^{(i)}))$;

$u^{(i)} \leftarrow lsr(v^{(i)})$;

$p_1{}^{(i)} \leftarrow lsr(p_1{}^{(i-1)})$;

$p_3{}^{(i)} \leftarrow lsr(lsr(lsr(p_3{}^{(i-1)})))$;

WHILE $(p_3{}^{(i)} \neq 0)$;

RETURN $(t^{(i)})$;

At this point we no longer need the variables p_1 and r, so we eliminate them from our algorithm. We also notice that we can make a simple optimization by noticing that

$$s^{(i)} = t^{(i-1)} + p_3{}^{(i-1)},$$

thus,

$$v^{(i)} = u^{(i-1)} + t^{(i-1)} + t^{(i-1)} + p_3{}^{(i-1)} = u^{(i-1)} + t^{(i-1)} + s^{(i)}.$$

Permitting us to eliminate one addition operation from our algorithm.

DO

$tmp^{(i)} \leftarrow u^{(i)} + v^{(i)} + s^{(i)} - t^{(i)} + p_3{}^{(i)}$;

$i \leftarrow i + 1$;

IF $(tmp^{(i-1)} \leq x^{(i-1)})$ **THEN**

$x^{(i)} \leftarrow x^{(i-1)} - tmp^{(i-1)}$;

$s^{(i)} \leftarrow t^{(i-1)} + p_3{}^{(i-1)}$;

$v^{(i)} \leftarrow u^{(i-1)} + t^{(i-1)} + s^{(i)}$;

ELSE

$s^{(i)} \leftarrow t^{(i-1)}$;

$v^{(i)} \leftarrow u^{(i-1)}$;

ENDIF

$t^{(i)} \leftarrow lsr(lsr(s^{(i)}))$;

$u^{(i)} \leftarrow lsr(v^{(i)})$;

$p_3{}^{(i)} \leftarrow lsr(lsr(lsr(p_3{}^{(i-1)})))$;

WHILE $(p_3{}^{(i)} \neq 0)$;

RETURN $(t^{(i)})$;

At this point we shall abandon our single assignment rule and implement our algorithm in pseudocode suitable for programming in any programming language. The only other modifications which we shall make to our algorithm is eliminating the temporary variables s and u and the timestamp i. This will involve rearranging some of our instructions, and will require multiple assignments to the variable *tmp* per iteration. We now present the final version of our algorithm:

Algorithm B.4 *cube root for arbitrary-size integers without multiply operations*

FUNCTION *cbrt*(X) **RETURNS** arbitrary_integer ;
INPUT X : arbitrary_integer ;

p_3 : arbitrary_integer ;
t : arbitrary_integer ;
tmp : arbitrary_integer ;
v : arbitrary_integer ;
x : arbitrary_integer ;

$p_3^{(0)} \leftarrow 2^{(3\lceil bits(X)/3 \rceil)}$; { cube of p_1 }
$t \leftarrow 0$;
$tmp \leftarrow 0$;
$v \leftarrow 0$;
$x \leftarrow X$;

DO

$tmp \leftarrow t + v$;
$t \leftarrow lsr(lsr(t))$;
$v \leftarrow lsr(v)$;
$tmp \leftarrow tmp + v - t + p3$;
IF $(tmp \leq x)$ **THEN**

$x \leftarrow x - tmp$;
$v \leftarrow v + t$;
$t \leftarrow t + p_3$;
$v \leftarrow v + t$;

ENDIF
$p_3 \leftarrow lsr(lsr(lsr(p_3)))$;

WHILE $(p_3 \neq 0)$;
RETURN (t) ;

End of Algorithm

As in the case for perfect squares, it is possible to identify a perfect cube value by merely examining the value x. The value of the local variable x holds the value of $X - r^3$ and it suffices to test if $x = 0$ to identify whether or not X is a perfect cube.

B.3 The Probable-Prime Filter – PBPM

SSUMon provides a probable-prime filter based on the Strong Pseudoprime Test of Pomerance, Selfridge and Wagstaff [PSW80]. The filter will reject most composite numbers and accept all prime numbers. However, a fairly scarce class of composite values, known as *strong pseudoprimes* are accepted as well; hence, the filter is a probable-prime test and not a true primality test. The actual algorithm used is that described by Bressoud in [Bre89, pp. 75–78]. The algorithm utilizes fast modular exponentiation. The base 2 was chosen because it resulted in a slight performance improvement over that of other bases, due to multiplication by 2 being equivalent to a single left shift. Pomerance, Selfridge and Wagstaff [PSW80] computed all of the *strong pseudoprimes to base 2*, denoted by $spsp(2)$, less than or equal to 2.5×10^{10}, and counted only 4842. Therefore, we can expect very few false-primes to pass the PBPM filter. These false-primes can be detected using an off-line primality test or factoring algorithm implemented in a math package such as Maple.

A number of different filters are available for specifying certain configurations of primes. These include: twin-primes X and $X+2$ (TWPM), triple-primes X, $X+2$ and $X+6$ (TRPM), and quad-primes X, $X+2$, $X+6$ and $X+8$ (QDPM). For specifying solutions which are twice a prime value, the filter PBP2 is provided.

Bibliography

[AH92] L.M. Adleman and M.-D. Huang. *Primality Testing and Abelian Varieties over Finite Fields*, volume 1512 of *Lecture Notes in Mathematics*. Springer-Verlag, 1992.

[APR83] L.M. Adleman, C. Pomerance, and R.S. Rumely. On distinguishing prime numbers from composite numbers. *Annals of Math*, 117:173–206, 1983.

[Bac90] Eric Bach. Explicit bounds for primality testing and related problems. *Math. Comp.*, 55:355–380, 1990.

[BB93] N.D. Bronson and D.A. Buell. Congruential Sieves on FPGA Computers. In *Mathematics of Computation 1943-1993: A half-century of computational mathematics*. AMS Proc. Symp., 1993. To appear.

[BBCO93] C. Batut, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP*. 1993. for version 1.38.

[Bee39] N.G.W.H. Beeger. Report on some calculations of prime numbers. *Nieuw Archief voor Wiskunde*, 20(2):48–50, 1939.

[Bee46] N.G.W.H. Beeger. Note sur la factorisation de quelques grands nombres. *Institut Grand-Ducal de Luxembourg, Sect. d.Sc. nat; phys; math. Archives*, 16:93–95, 1946.

[Bee30] N.G.W.H. Beeger. Sur la décomposition de grands nombres. *Nieuw Archief voor Wiskunde*, 16(2):37–42, 1929/30.

[BF81] Jan Bohman and Carl-Erik Fröberg. Numerical investigation of waring's problem for cubes. *BIT*, 21:118–122, 1981.

[BGN93] Andrew Bremner, Richard K. Guy, and Richard J. Nowakowski. Which integers are representable as the product of the sum of three integers with the sum of their reciprocals? 1993.

[BH93] Eric Bach and L. Huelsbergen. Statistical evidence for small generating sets. *Math. Comp.*, 61:69–82, 1993.

[BLSW95] Erich Bach, Richard Lukes, Jeffrey Shallit, and H.C. Williams. Some results and estimates on pseudopowers. *Math. Comp.* To appear, 1995.

[Bre89] David M. Bressoud. *Factorization and Primality Testing.* Undergraduate Texts in Mathematics. Springer-Verlag, 1989.

[Bre94] Andrew Bremner. On sums of three cubes. To appear, 1994.

[BS67] John Brillhart and J.L. Selfridge. Some factorizations of $2^n \pm 1$ and related results. *Math. Comp.*, 21:87–96, 1967.

[BS93] Eric Bach and J. Sorenson. Sieve algorithms for perfect power testing. *Algorithmica*, 9:313–328, 1993.

[Car20a] E. Carissan. Machine à resoudre les congruences. *Bulletin de la Société d'Encouragement pour l'Industrie Nationale*, 132:600–607, 1920.

[Car20b] E. Carissan. Principe mechanique et description de la machine du C^t E. Carissan. Unpublished MS, Rennes, May 9, 1920.

[CDyDO93] H. Cohen, F. Diaz y Diaz, and M. Olivier. Calculs de nombres de classes et de régulateurs de corps quadratiques en temps sous-exponentiel. In *Séminaire de Théorie des Nombres*, pages 35–46, Paris, 1993.

[CEFT62] D.G. Cantor, G. Estrin, A.S. Fraenkel, and R. Turn. A very high-speed digital number sieve. *Math. Comp.*, 16:141–154, 1962.

[CGG+88] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Michael B. Monagan, and Stephen M. Watt. *Maple reference manual.* WATCOM Publications, 5 edition, mar 1988.

[Cho49] S. Chowla. Improvement of a theorem of Linnik and Walfisz. *Proc. London Math. Soc.*, 50:423–429, 1949.

[CL83] H. Cohen and H.W. Lenstra, Jr. Heuristics on class groups of number fields. In *Lecture Notes in Math*, volume 1068, pages 33–62, Noordwijkerhout, 1983. Number Theory, Springer-Verlag.

[CL84] H. Cohen and H.W. Lenstra, Jr. Heuristics on class groups. In *Number Theory, Lecture Notes in Math.*, volume 1052, pages 26–36. Springer-Verlag, 1984.

[Cob66] A. Cobham. The recognition problem for the set of perfect squares. *IBM Research Report RC1704*, pages 1–16, 1966.

[Cob68] Alan Cobham. Letter to Drs. D. H. and E. Lehmer, March 1968.

[Con65] S.D. Conte. *Elementary Numerical Analysis*, section 2.3, pages 30–38. Information Processing and Computers. McGraw-Hill, New York, 1965.

[Dic28] L.E. Dickson. Simpler proofs of waring's theorem on cubes with various generalizations. *Trans. Amer. Math. Soc.*, 30:1–18, 1928.

[Dic39] L.E. Dickson. All integers except 23 and 239 are sums of eight cubes. *Bull. Amer. Math. Soc.*, 45:588–591, 1939.

[Dic66] L.E. Dickson. *History of the Theory of Numbers*, pages 30–38. Chelsea Publishing, New York, 1966.

[EES74] E. F. Ecklund, P. Erdös, and J. L. Selfridge. A new function associated with the prime factors of $\binom{n}{k}$. *Math. Comp.*, 28(126):647–649, April 1974.

[Ell71] W.J. Ellison. Waring's problem. *Amer. Math. Monthly*, 78:10–36, 1971.

[ELS93] P. Erdös, C. B. Lacampagne, and J. L. Selfridge. Estimates of the least prime factor of a binomial coefficient. *Math. Comp.*, 61(203):215–224, July 1993.

[FW90] G.W. Fung and H.C. Williams. Quadratic polynomials which have a high density of prime values. *Math. Comp.*, 55:345–353, 1990.

[Gér12a] A. Gérardin. Question 335. *Sphinx-Oedipe*, 7:47–48, 1912. Also see p. 31.

[Gér12b] A. Gérardin. Rapport sur diverses méthodes de solutions employées en théorie pour la décomposition des nombres en facteurs. *Assoc. Française pour l'Avancement des Sciences. Comptes Rendus*, 41(II partie):54–57, 1912.

[Gér12c] A. Gérardin. Réponse 335. *Sphinx-Oedipe*, 7:61–64, 1912.

[Gér12d] A. Gérardin. Sur une nouvelle machine algébrique. *British Assoc. Reports*, 82:405–406, 1912.

[Gér13] A. Gérardin. Sur quelques nouvelles machines algébriques. In *Proc. Fifth Int. Congress of Mathematicians, Vol II*, pages 572–573, Cambridge, 1913.

[Gér37] A. Gérardin. Machine à congruences (modèle 1937). In 70^e *Congrès des Sociétés Savantes de Paris et des Départements, Section des Sciences*, pages 14,II, p.37. Gauthier-Villars, Paris, 1937.

[GLS64] V.L. Gardiner, R.B. Lazarus, and P.R. Stein. Solutions of the diophantine equation $x^3 + y^3 = z^3 - d$. *Math. Comp.*, (18):408–413, 1964.

[GR94] Andrew Granville and Olivier Ramaré. Explicit bounds on exponential sums and the scarcity of squarefree binomial coefficients. To appear, 1994.

[Gru94] Volker Grupe. personal communication via electronic mail, January 1994.

[Guy94] Richard K. Guy. *Unsolved Problems in Number Theory*, volume I of *Unsolved Problems in Intuitive Mathematics*. Springer-Verlag, 2 edition, 1994.

[Hal33] M. Hall. Quadratic residues in factorization. *Bull. Amer. Math. Soc.*, 39:758–763, 1933.

[HBLtR93] D.R. Heath-Brown, W.M. Lioen, and H.J.J. te Riele. On solving the diophantine equation $x^3 + y^3 + z^3 = k$ on a vector computer. *Math. Comp.*, 61(203):235–244, July 1993.

[HL23] G.H. Hardy and J.E. Littlewood. Partitio Numerorum III: On the expression of a number as a sum of primes. *Acta Mathematica*, 44:1–70, 1923.

[HW79] G.H. Hardy and E.M. Wright. *An Introduction to the Theory of Numbers.* Oxford University Press, 5 edition, 1979.

[Jac95] M.J. Jacobson, Jr. *Computational Techniques in Quadratic Fields.* MSc dissertation, University of Manitoba, Department of Computer Science, 1995.

[Jos70] P.T. Joshi. The size of $L(1, \chi)$ for real nonprincipal residue characters χ with prime modulus. *J. Number Theory*, 2:58–73, 1970.

[Koy91] Kenyi Koyama. On the solutions of the diophantine equation $x^3+y^3+z^3 = n$. unpublished, 1991.

[Kra22] M. Kraitchik. *Théorie des Nombres.* Gauthier-Villars, Paris, 1922.

[Kra24] M. Kraitchik. *Récherches sur la Théorie des Nombres.* Gauthier-Villars, Paris, 1924.

[Kra29] M. Kraitchik. *Récherches sur la Théorie des Nombres,* T. II. Gauthier-Villars, Paris, 1929.

[Kra31] M. Kraitchik. Factorisation des grands nombres. *Sphinx*, 1:35–37, 1931.

[Law96] F.W. Lawrence. Factorisation of numbers. *Quart. J. Pure Appl. Math.*, 28:285–311, 1896. French translation in *Sphinx-Oedipe*, 5:98-121, 1910.

[Leh28] D.H. Lehmer. The mechanical combination of linear forms. *Amer. Math. Monthly*, 35:114–121, 1928.

[Leh30] D.H. Lehmer. A fallacious principle in the theory of numbers. *Bull. Amer. Math. Soc.*, 36:847–850, 1930.

[Leh33a] D.H. Lehmer. A photo-electric number sieve. *Amer. Math. Monthly*, 40:401–406, 1933.

[Leh33b] D.H. Lehmer. Some new factorizations of $2^n \pm 1$. *Bull. Amer. Math. Soc.*, 39:105–108, 1933.

[Leh34] D.H. Lehmer. A machine for combining sets of linear congruences. *Math. Annalen*, 109:661–667, 1934.

[Leh37] D.H. Lehmer. On the function $x^2 + x + A$. *Sphinx*, 6:212–214, 1936 and 7:40, 1937.

[Leh38] D.H. Lehmer. Sur les essais directs de primalité. *Sphinx*, 8:87–88, 1938.

[Leh39] D.H. Lehmer. A factorization theorem applied to a test for primality. *Bull. Amer. Math. Soc.*, 45:132–137, 1939.

[Leh46] D.H. Lehmer. Preliminary proposal for the design and construction of the electronic sieve. Unpublished MS, U.C. Berkeley, December 1, 1946.

[Leh53] D.H. Lehmer. The sieve problem for all-purpose computers. *MTAC*, 7:6–14, 1953.

[Leh54] D.H. Lehmer. A sieve problem on pseudo-squares. *Math. Tables and other Aids to Computation*, 8:241–242, 1954.

[Leh56] D.H. Lehmer. On the diophantine equation $x^3 + y^3 + z^3 = 1$. *J. London Math. Soc.*, 31:275–280, 1956.

[Leh66] D.H. Lehmer. An announcement concerning the delay line sieve dls-127. *Math. Comp.*, 20:645–646, 1966.

[Leh80] D.H. Lehmer. A history of the sieve process. In *A History of Computing in the Twentieth Century*, pages 445–456. Academic Press, 1980.

[Leh33] D.N. Lehmer. Hunting big game in the theory of numbers. *Scripta Mathematica*, 1:229–235, 1932/33.

[Lit28] J.E. Littlewood. On the class number of the corpus $P(\sqrt{-k})$. *Proc. London Math. Soc.*, 27:358–372, 1928.

[LLS70] D.H. Lehmer, E. Lehmer, and D. Shanks. Integer sequences having prescribed quadratic character. *Math. Comp.*, 24:433–451, 1970.

[LMW92] J.S. Louboutin, R.A. Mollin, and H.C. Williams. Class numbers of real quadratic fields, continued fractions, reduced ideals, prime-producing quadratic polynomials and quadratic residue covers. *Can. J. Math*, 44:1–19, 1992.

[LPWa] R. F. Lukes, C. D. Patterson, and H. C. Williams. Numerical sieving devices: Their history and some applications. *Nieuw Archief v. Wiskunde* To appear.

[LPWb] R.F. Lukes, C.D. Patterson, and H.C. Williams. Some results on pseudosquares. *Math. Comp.* To appear.

[LRB69] M. Lal, W. Russell, and W.J. Blundon. A note on the sums of four cubes. *Math. Comp.*, 23:423–424, 1969.

[LSW94] Richard F. Lukes, Renate Scheidler, and Hugh C. Williams. Further tabulation of the erdös-selfridge function. To appear, 1994.

[MB75] Michael A. Morrison and John Brillhart. A method of factoring and the factorization of F_7. *Math. Comp.*, 29:183–205, 1975.

[MIP94] Technologies Inc. MIPS. R10000 microprocessor product overview. pages 1–12, October 1994.

[MW55] J.C.P. Miller and M.F.C Woollett. Solutions of the diophantine equation $x^3 + y^3 + z^3 = k$. *J. London Math. Soc.*, (30):101–110, 1955.

[MW90] R.A. Mollin and H.C. Williams. Class number problems for real quadratic fields. *Number Theory and Cryptography*, LMS Lecture note series 154:177–195, 1990. Cambridge.

[MW92] R.A. Mollin and H.C. Williams. Computation of the class number of a real quadratic field. *Utilitas Mathematica*, pages 259–308, 1992.

[Pat83] Cameron Douglas Patterson. *Design and Use of an Electronic Sieve.* MSc dissertation, University of Manitoba, Department of Computer Science, 1983.

[Pat91] Cameron Douglas Patterson. *Derivation of a High Speed Sieve Device.* PhD dissertation, University of Calgary, Department of Computer Science, November 1991.

[Pet90] Otto Peter. Prozessor zieht wurzeln. *c't*, (1):300–306, January 1990.

[PH90] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*, section A.6, pages A23–26. Morgan Kaufmann Publishers, San Mateo, California, 1990.

[Pol39] L. Poletti. Au sujet de la décomposition des termes de la série $z = x^2 + x + 146452961$. *Sphinx*, 9:83–85, 1939.

[Pol51] L. Poletti. Il contributo italiano alla tavola dei numeri primi. *Rivista di Matematica della Università di Parma*, 2:417–434, 1951.

[Pom82] C. Pomerance. The search for prime numbers. *Scientific American*, 247:136–147, December 1982.

[PSW80] Carl Pomerance, John L. Selfridge, and Samuel S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Math. Comp.*, 35:1003–1026, 1980.

[PV92] G. Payne and L. Vaserstein. Sums of three cubes. In *The Arithmetic of Function Fields*, pages 443–454. de Gruyter, 1992.

[PW83] C.D. Patterson and H.C. Williams. A report on the university of manitoba sieve unit. *Congressus Numerantium*, 37:85–98, 1983.

[PW85] C.D. Patterson and H.C. Williams. Some periodic continued fractions with long periods. *Math. Comp.*, 44:523–532, 1985.

[Rib88a] P. Ribenboim. Euler's famous prime generating polynomial and the class number of imaginary quadratic fields. *L'Enseignement Math.*, 34:23–42, 1988.

[Rib88b] Paulo Ribenboim. *The Book of Prime Number Records.* Springer-Verlag, New York, 1988.

[Rom82] F. Romani. Computations concerning Waring's problem for cubes. *Calcolo*, 19:415–431, 1982.

[SB84] Manny Scarowsky and Abraham Boyarsky. A note on the diophantine equation $x^3 + y^3 + z^3 = 3$. *Math. Comp.*, 42(165):235–237, January 1984.

[Sha59] D. Shanks. A sieve method for factoring numbers of the form $n^2 + 1$. *MTAC*, 13:78–86, 1959.

[Sha60] D. Shanks. On the conjecture of Hardy and Littlewood concerning the number of primes of the form $n^2 + a$. *Math. Comp.*, 14:320–332, 1960.

[Sha63] D. Shanks. Supplementary data and remarks concerning a Hardy-Littlewood conjecture. *Math. Comp.*, 17:188–193, 1963.

[Sha73] D. Shanks. Systematic examination of Littlewood's bounds on $L(1, \chi)$. In *Proc. Symp. Pure Math. Vol. 24*, pages 267–283, AMS, Providence, 1973.

[Sha75] D. Shanks. Calculation and applications of epstein zeta functions. *Math. Comp.*, 29:271–287, 1975.

[Sit92] Richard L. Sites. Alpha AXP architecture. *Digital Technical Journal*, 4(4):19–34, 1992.

[Smi73] David Eugene Smith. *History of Mathematics: Special Topics of Elementary Mathematics*, volume 2, section 7, pages 144–149. Ginn, Boston, 1973.

[Sta75] H. M. Stark. On complex quadratic fields with class-number two. *Math. Comp.*, 29(129):289–302, 1975.

[Ste89] Allan J. Stephens. *OASiS: An Open Architecure Sieve System for Problems in Number Theory.* PhD dissertation, University of Manitoba, Department of Computer Science, 1989.

[Ste90] W. Richard Stevens. *Unix Network Programming*, section 2.6, pages 72–85. Prentic-Hall, Englewood Cliffs, New Jersey, 1990.

[SW90] A.J. Stephens and H.C. Williams. An open architecture number sieve. *London Math. Soc. lecture Note Series*, 154:38–75, 1990.

[SW92] R. Scheidler and H. C. Williams. A method of tabulating the number theoretic function $g(k)$. *Math. Comp.*, 59(199):251–257, July 1992.

[Wat51] G. L. Watson. A proof of the seven cube theorem. *J. London Math. Soc.*, 26:153–156, 1951.

[Wes26] A. E. Western. Computations concerning numbers representable by four or five cubes. *J. London Math. Soc.*, 1:244–250, 1926.

[Wil78] H.C. Williams. Primality testing on a computer. *Ars Combinatoria*, 5:127–185, 1978.

[Wil81] H.C. Williams. A numerical investigation into the length of the period of the continued fraction expansion of $\sqrt{D}$. *Math. Comp.*, 36:593–601, 1981.

[Wri34] E. M. Wright. An easier waring's problem. *J. London Math. Soc.*, 9:267–272, 1934.

[WS] H.C. Williams and J.O. Shallit. Factoring integers before computers. *Mathematics of Computation 1943-1993: A Half-Century of Computational Mathematics*, AMS Proc. Symp. Appl. Math.

[Wun73] M. C. Wunderlich. On the gaussian primes on the line $im(x) = 1$. *Math. Comp.*, 27:399–400, 1973.

[YP89] Jeff Young and Aaron Potler. First occurrence prime gaps. *Math. Comp.*, 52:221–224, 1989.