2014

Massively parallel solvers for lattice quantum gauge field theories

Dieter John David Beaven
University of Wollongong

UNIVERSITY OF WOLLONGONG

Massively Parallel Solvers For Lattice Quantum Gauge Field Theories

by

Dieter John David Beaven

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Faculty of Engineering
School of Engineering Physics

July 2014

Declaration of Authorship

I, Dieter John David Beaven, declare that this thesis titled, 'Massively Parallel Solvers For Lattice Quantum Gauge Field Theories' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

"Feynman told me about his sum over histories version of quantum mechanics. The electron does anything it likes, he said. It just goes in any direction at any speed, forward or backward in time, however it likes, and then you add up the amplitudes and it gives you the wave-function. I said to him, you're crazy. But he wasn't."

Freeman Dyson

UNIVERSITY OF WOLLONGONG

Abstract

Faculty of Engineering
School of Engineering Physics

Doctor of Philosophy

by Dieter John David Beaven

Gauge field theories have been very successful in their description of quantum many-body field interactions. For interactions with strong coupling constants a computational approach is needed since analytic techniques fail to provide convergent solutions, such as occurs in the study of the strong nuclear force between quarks and gluons of atomic nuclei and other hadrons. The key computational bottleneck in such calculations is the solution of an extremely large linear algebra problem of finding solutions to non-symmetric Wilson-Dirac matrices with maybe a billion unknowns. With a floating point performance requirement that challenges even the fastest petaflop supercomputers, finding algorithms that scale well over massively parallel distributed architectures is an ever present necessity. Results of an investigation into the use of small cost-effective reconfigurable devices is presented here, along with the solver methods currently being used, and those currently being researched as possible improvements. A reconfigurable design is considered and a research software library is presented as a means to efficiently investigate potential new algorithms. Results collected from this research software library are presented leading up to an improved preconditioning technique that addresses the time consuming critical slowing down problem inherent in calculations for low mass spin-half particles near the lattice physical point. This proposed preconditioner is a hybrid combination of existing algorithms factored together in a highly configurable manner that can be dynamically adapted to any specific matrix during the solve to increase performance. Applications to solid state physics problems such as the conductivity and optical properties of graphene are discussed with regards to how these solvers are also of interest due to the existence of massless spin-half quasi-particles interacting strongly with the hexagonal carbon lattice.

Acknowledgements

I would like to thank the engineering team at Cochlear Ltd, Sydney for supporting the early years of my work with FPGAs and generally for being a great place to work. Andrew Saldanha deserves thanks for first introducing FPGAs to me and catalysing the idea that FPGAs may be of use for high performance matrix inversion. Thank you to Jayne Andrews for all those discussions about the philosophy of physics that kept the ideas in motion. Thank you to Adrian Cryer and the team for all the support over the final year or so there.

Professor Chao Zhang I would like to thank for his help and patience over the years as my co-supervisor then principle supervisor, and I still hope to one day magic-up a speed of light eigensolver to solve all your matrices. I would like to thank Professor John Fulcher for taking such a big interest in the original idea in the first place and for his indefatigable enthusiasm from day 1 to day N; where N has finally been determined. Without the assistance of Greg Doherty and his linear algebra expertise the journey would have been even more than impossible: a big thank you.

An special mention goes to Frederick Beaven who has provided immeasurable moral support over the past few years; always being insistent that his little brother reaches the finishing post one way or another. Thank you to Tracey Cull who kept on reminding me that I should keep on trying for the goal line. Finally a loving thanks to James, Nathan and Amber who almost managed to keep me sane at the weekends and holidays.

Contents

Declaration of Authorship i

Abstract iii

Acknowledgements iv

List of Figures viii

List of Tables xi

Abbreviations xii

1 Introduction 1
- 1.1 Feynman Paths . . . 1
- 1.2 Molecular Dynamics Algorithm . . . 4
- 1.3 The Lattice Dirac Matrix . . . 6
- 1.4 Computational Complexity . . . 8
- 1.5 Thesis Rationale . . . 11

2 Matrix Solvers 12
- 2.1 Numerical Investigation . . . 13
- 2.2 Stationary Methods . . . 15
- 2.3 Krylov Subspace Solvers . . . 22
  - 2.3.1 Conjugate Gradient Normal Residual . . . 23
  - 2.3.2 BiConjugate Gradient Stabilized . . . 26
  - 2.3.3 Generalized Conjugate Residual . . . 31
- 2.4 Multigrid . . . 35
- 2.5 Preconditioning . . . 39
- 2.6 Wilson-Dirac Solvers . . . 41

3 Reconfigurable Hardware 43
- 3.1 Software Integrated Co-Development . . . 45
- 3.2 BLAS Applications . . . 46
- 3.3 Toolsets . . . 47
- 3.4 Integration . . . 51

4 Software Development **52**
4.1 SystemC Components . . . 53
4.2 SystemC Algorithm . . . 57
4.3 Werner . . . 60
4.4 Multiprocessing . . . 66
4.5 QUDA . . . 68
4.6 Reconfiguration . . . 69
4.7 FPGA and Werner . . . 71

5 Results and Discussion **72**
5.1 Multigrid and Dirac Linearity . . . 73
5.2 Optimal SSOR Parameter . . . 78
5.3 SSOR Depth Parameter . . . 80
5.4 Preconditioned GCR . . . 81
5.4.1 Matrix Mass Behaviour . . . 84
5.4.2 PGCR Restarts . . . 88
5.5 Hybrid Preconditioned GCR . . . 92
5.5.1 Matrix Size Behaviour . . . 97
5.6 Preconditioned BiCGStab . . . 100
5.6.1 Matrix Size . . . 100
5.6.2 Matrix Mass . . . 102
5.7 Hybrid Preconditioned BiCGStab . . . 104
5.7.1 Matrix Size Performance . . . 109
5.7.2 Preconditioner Restarts . . . 111
5.8 Hybrid Flexible BiCGStab . . . 113
5.9 Multi Processing . . . 115

6 Conclusions and Further Work **117**
6.1 Hybrid Preconditioning . . . 118
6.2 Graphene Physics . . . 122
6.3 Further Work . . . 125

A Feynman Path Lattice QFT **128**
A.1 The Feynman Path Integral . . . 128
A.2 Lattice Path Integration . . . 131
A.3 The Wilson-Dirac Matrix . . . 133
A.4 Gauge Fields . . . 137
A.5 Lattice Simulations . . . 140

B Werner Software Library **143**
B.1 Werner Directory Structure . . . 144
B.2 Main Application Example . . . 146
B.3 Werner Solver . . . 149
B.4 Inverter Interface Base Class . . . 151
B.5 Preconditioned Conjugate Gradient . . . 152
B.6 Conjugate Gradient with Normal Residual . . . 153

B.7 Biorthogonal Conjugate Gradient Stablised 155
B.8 General Conjugate Resdiual (with Restarts) 157
B.9 MPI-pthread PBiCG Stablised 159
B.10 MPI-pthread GCR(restarts) 163
B.11 Stationary Solver Template Class 165
B.12 Smoother Base Class 167
B.13 Symmetric Gauss-Seidel Relaxed 168
B.14 Multigrid Solver 171
B.15 GCR-SSOR Hybrid Preconditioner 173
B.16 MPI-pthread GCR-SSOR Hybrid Preconditioner 176

Bibliography **179**

List of Figures

2.1 Werner matrix solver command line. 13
2.2 Jacobi convergence for high mass (m=-0.4) LQCD matrix. 16
2.3 Jacobi performance against matrix size. 18
2.4 Jacobi convergence for medium mass (m=-0.9) LQCD matrix. 18
2.5 Jacobi divergence for low mass (m=-1.2) LQCD matrix. 19
2.6 Jacobi divergence for lowest mass (m=-2.0) LQCD matrix. 20
2.7 Jacobi iterations against matrix mass. 20
2.8 CGNR time taken against matrix size. 24
2.9 CGNR time taken against mass. 24
2.10 CGNR convergence for medium mass (m=-0.4) LQCD matrix. 25
2.11 CGNR convergence for low mass (m=-1.2) LQCD matrix. 26
2.12 CGNR convergence for lowest mass (m=-2.0) LQCD matrix. 26
2.13 BiCGStab time taken against matrix size. 27
2.14 BiCGStab time taken against mass. 28
2.15 BiCGStab convergence for high mass (m=-0.4) LQCD matrix. 30
2.16 BiCGStab convergence for medium mass (m=-1.0) LQCD matrix. 30
2.17 BiCGStab convergence for low mass (m=-1.1) LQCD matrix. 30
2.18 GCR time taken against matrix size. 32
2.19 GCR time taken against mass. 33
2.20 GCR convergence for medium mass (m=-0.4) LQCD matrix. 34
2.21 GCR convergence for low mass (m=-1.2) LQCD matrix. 34
2.22 GCR convergence for very low mass (m=-1.8) LQCD matrix. 34
2.23 Multigrid time taken against matrix size. 35
2.24 Multigrid command line execution. 36
2.25 Multigrid time taken against mass. 38
2.26 Eigenvalues of the Wilson-Dirac matrix. 42

4.1 VHDL interface for a floating point adder unit. 53
4.2 SystemC module for a floating point adder unit. 54
4.3 SystemC architecture for a floating point adder unit. 55
4.4 SystemC test bench for a floating point adder unit. 56
4.5 SystemC test bench logic trace output. 56
4.6 Conjugte gradient algorithm implementation in SystemC. 58
4.7 Conjugte gradient algorithm logic trace. 59
4.8 Werner solver interface. 60
4.9 Main function for Werner solver testing. 61
4.10 Command line options for Werner. 62
4.11 Werner solver interface. 63

4.12 Werner conjugte gradient solver initialisation. 64
4.13 Werner conjugte gradient solver implementation. 65
4.14 Conjugate Gradient Jumpshot. 66
4.15 Improved Conjugate Gradient Jumpshot. 67
4.16 Primatives linked via C++ templates. 71

5.1 AMG solver time against matrix size. 74
5.2 Matrix reference generation time against matrix size. 75
5.3 Improved generation time against matrix size. 76
5.4 GCR performance against SSOR relaxation parameter. 79
5.5 GCR time against SSOR depth parameter. 80
5.6 Loop unrolling beta calculation in GCR. 81
5.7 PGCR time taken against matrix size. 83
5.8 PGCR convergence N=20 and m=40. 83
5.9 The effect of SSOR smoothing on medium mass fermion matrices. 85
5.10 PGCR time taken against matrix mass. 86
5.11 PGCR convergence N=10 and m=150. 87
5.12 PGCR convergence N=10 and m=200. 87
5.13 PGCR(L) convergence N=10 and m=130. 88
5.14 PGCR(L) convergence N=10 and m=140. 88
5.15 PGCR(L) convergence N=10 and m=140 close-up. 89
5.16 PGCR(L) convergence N=10 and m=200. 89
5.17 GCR(L) convergence against mass, with N=10. 90
5.18 GCR(L) convergence against mass. 90
5.19 GCR(L) convergence with mass m=130. 91
5.20 GCR(L) convergence with mass m=180. 91
5.21 Hybrid PGCR time taken against matrix mass. 93
5.22 Hybrid PGCR convergence N=10 and m=200. 93
5.23 Command line output for GCR Hybrid-GCR-SSOR method. 94
5.24 Command line output for GCR Hybrid-GCR-SSOR with Multigrid. . . . 96
5.25 Hybrid PGCR time taken against matrix rank. 98
5.26 Time taken per GCR outer loop against matrix rank. 98
5.27 PBiCGStab time taken against matrix size. 101
5.28 PBiCGStab convergence N=20 and m=40. 101
5.29 PBiCGStab time taken against matrix mass. 102
5.30 PBiCGStab convergence N=10 and m=110. 103
5.31 PBiCGStab convergence N=10 and m=120. 103
5.32 Hybrid PBiCGStab time taken against mass. 105
5.33 Hybrid PBiCGStab convergence N=10 and m=190. 106
5.34 Hybrid PBiCGStab convergence N=10 and m=200. 106
5.35 Hybrid PBiCGStab command line output. 107
5.36 Hybrid PBiCGStab command line output. 108
5.37 Hybrid PBiCGStab time taken against matrix size, r=100, m=180. 109
5.38 Hybrid PBiCGStab convergence N=12, m=180, restarts=100. 110
5.39 Hybrid PBiCGStab convergence N=15, m=180, restarts=100. 110
5.40 Hybrid PBiCGStab convergence N=16, m=180, restarts=100. 110
5.41 Hybrid PBiCGStab time taken against restarts. 112

5.42 Hybrid PBiCGStab convergence N=10, m=140, restarts=10. 112
5.43 Hybrid PBiCGStab convergence N=10, m=140, restarts=30. 112
5.44 Hybrid PBiCGStab performance for critical LQCD matrix. 113
5.45 Hybrid PBiCGStab performance and matrix size. 114
5.46 Hybrid performance against Pthreads. 115
5.47 Hybrid performance against MPI nodes. 116

6.1 The effect of SSOR-GCR hybrid preconditioning with low mass matrices. 118
6.2 Flexible BiCGStab monotonic convergence. 119

B.1 Werner directory source structure. 144
B.2 Werner directory source structure. 145

List of Tables

2.1 Jacobi performance against LQCD matrix size. 17
2.2 Jacobi performance against LQCD mass. 19
2.3 CGNR performance against matrix size. 23
2.4 CGNR performance against mass. 25
2.5 BiCGStab performance against matrix size. 27
2.6 BiCGStab performance against mass. 28
2.7 GCR performance against matrix size. 31
2.8 GCR performance against mass. 33
2.9 Multigrid performance against matrix size. 36
2.10 Multigrid time against depth dependency. 37
2.11 Multigrid performance against mass. 37

4.1 QUDA test code performance against lattice width. 68
4.2 QUDA test code performance against mass. 69

5.1 Matrix reference generation performance. 73
5.2 Matrix reference generation performance. 75
5.3 Improved matrix generation performance. 76
5.4 Relaxation parameter performance. 78
5.5 GCR performance and SSOR relaxation parameter. 79
5.6 GCR performance and SSOR depth. 80
5.7 PGCR performance and matrix size. 82
5.8 SSOR solver performance against mass. 84
5.9 PGCR performance and matrix mass. 86
5.10 Hybrid PGCR performance and matrix mass. 92
5.11 Hybrid PGCR performance and 100 restarts. 98
5.12 PBiCGStab performance and matrix size. 100
5.13 PBiCGStab performance and matrix mass. 102
5.14 Hybrid PBiCGStab performance and mass. 104
5.15 Hybrid PBiCGStab performance with 100 restarts for mass 180. 109
5.16 Hybrid PBiCGStab performance and restarts. 111
5.17 Hybrid PBiCGStab performance and matrix size. 114
5.18 Hybrid performance against Pthreads. 115
5.19 Hybrid performance against MPI nodes. 116

Abbreviations

AMG	**A**lgebraic **M**ulti**G**rid
BiCG	**Bi**-**C**onjugate **G**radient
BiCGStab	**Bi**-**C**onjugate **G**radient **S**tablised
BLAS	**B**asic **L**inear **A**lgebra **S**ubroutines
CPU	**C**entral **P**rocessing **U**int
CG	**C**onjugate **G**radient
CGNR	**C**onjugate **G**radient **N**ormalised **R**esidual
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DDOT	**D**ouble **DOT** product
DGEMM	**D**ouble **GE**neral **M**atrix **M**ultiply
DGEMV	**D**ouble **GE**neral **M**atrix **V**ector
DSP	**D**igital **S**ignal **P**rocessing
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GCR	**G**eneralised **C**onjugate **R**esidual
GPU	**G**raphical **P**rocessing **U**nit
HDL	**H**ardware **D**escription **L**anguage
HMC	**H**ybrid **M**onte **C**arlo
HPC	**H**igh **P**erformance **C**omputing
IDE	**I**ntegrated **D**evelopment **E**nvironment
LQCD	**L**attice **Q**uantum **C**hromo**D**ynamics
MAC	**M**ultiplay **AC**ummulate
MPI	**M**essage **P**assing **I**nterface
PGCR	**P**reconditioned **G**eneralised **C**onjugate **R**esidual
QED	**Q**uantum **E**lectro**D**ynamics
SSOR	**S**ymmetric **S**uccessive **O**ver**R**elaxation

Dedicated to my parents Kenneth, Käthe Beaven and my children James, Nathan, Amberley.

Chapter 1

Introduction

"Feynman was fond of saying that all of quantum mechanics can be gleaned from carefully thinking through the implications of this single experiment, so it's well worth discussing."

Brian Greene

How does a particle move from A to B? Heisenberg's uncertainty principle results in there generally being no unique answer to this problem and requires calculations involving all possible paths to be considered. When weighted accordingly this produces predictions for measurable quantities. The two original and independent formulations of quantum mechanics, Heisenberg's Matrix mechanics and Schrödinger's wave mechanics, were shown to be equivalent by Dirac but both were based on classical Hamiltonian equations. The quadratic nature of Einstein's energy-mass equivalence equation resulted in negative probabilities that caused early relativistic formulations to be rejected. Dirac found a work-around that preserved probability as positive but at the expense of introducing additional particles that mirrored the original particle; with these being required to form the complete dynamical description. Positron anti-matter was discovered a few years later, and Dirac's elegant theory became scientific fact.

1.1 Feynman Paths

The postulate of quantum mechanics that all available information is contained within the wavefunction leads to a very abstract formalism, the power of which is its ability to produce extremely accurate predictions concerning phenomena in which human everyday experience and common sense are of little use. An area of debate raging from the early

days, personified by Einstein versus Bohr, and causing Schrödinger to renounce his own theory, is the meaning of this mysterious wavefunction. Based on ideas first described by Dirac [5], Feynman postulated that the wavefunction is a superposition of all possible paths a particle can travel from A to B weighted by a factor depending on the classical actions evaluated over those paths.

Feynman demonstrated this approach to building a wavefunction from a history of all possible paths was functionally equivalent to Heisenberg's matrix operator approach and the Schrödinger equation is a derivable result in the case of a single particle [6]. For massive objects with a short de Broglie wavelength, or equivalently as Planck's constant tends to zero, the paths deviating from the most likely cancel each other out to leave the classical path overwhelmingly dominant; from which one recovers the classical principle of least action for the path followed by a deterministic particle.

Whilst considered elegant, and reducing the principles of quantum mechanics to the study of Young's double slit experiment, the primary theoretical significance of the path integral formalism is that the mathematical structures are given in terms of scalar products. Unlike the Schrödinger and Heisenberg matrix versions based on vector derivatives, having a system based on scalars allows Einstein's special relativity to be seamlessly incorporated with quantum uncertainty. The path integral is now considered the starting point for developments in modern quantum field theories such as String theory and Quantum Loop gravity.

An inhibiting drawback of path integrals however as been the intractable mathematics. Variational calculus over infinite dimensional integrals can only be analytically solved for a very small subset of functions; proto-typically gaussian distributions for particles free from force field interactions. Once force field interactions are introduced the standard technique used is that of the perturbative expansion of Green function solutions, and one is lead to a set of Feynman Rules that can be organised with the aid of Feynman Diagrams.

Quantum Electro-Dynamics was the first fully worked theory where both the charged particles and the electromagnetic force field itself are both subject to quantum fluctuations within the path integral framework. Features that can be observed and measured can be seen using the path integral framework as being related to underlying symmetries within the scalar functions of the particle and field actions. Newton's third law relating to momentum conservation results from the symmetry of the laws of physics being the same at point A as they are at point B (homogeneity of space). The electromagnetic force acting on a charged particle can be seen as a symmetry whereby the wavefunction at point A can be out of phase with the wavefunction at B without affecting any measurable observables: that is, the wavefunctions are local, but then causally connected

by an intermediating function subject to the ultimate speed limit postulated by special relativity.

This wavefunction symmetry goes by the name of a gauge symmetry, and the intermediating function by the name of a gauge potential. When the gauge potential has one degree of freedom within a four dimension framework of spacetime, the result is a vector potential whose component's derivatives obey Maxwell's Equations and whose variational quanta intermediate between events A and B at the speed of light: Einstein's photons. QED can be analysed using mathematical techniques since the Green functions can be expanded in powers of the fine structure constant, 1/137; and so forms a converging series when calculating observables.

The quantum gauge symmetries may also be investigated for higher degrees of freedom. The original attempt was by Yang-Mills for two degrees of freedom, to model the almost perfect symmetry between protons and neutrons, and the beta decay force that converts between them. Nature declined to invoke this even-numbered symmetry in its original form, but a decade later a symmetry breaking variant using the Higgs mechanism was able to reconcile the difference and produce a gauge field model for the weak nuclear force intermediated by massive quanta over very short ranges.

The quantum gauge symmetry with three degrees of freedom results in a model of the strong nuclear force. Due to the strength of the interaction only limited calculations can be done since Green function expansions generally diverge and numerical values are hard to extract analytically. This has severely limited the testability of the theory. In 1974 Kenneth Wilson proposed a method to discretise the path integral approach in order to perform the calculation of observables on a digital computer [7]; and applications have been slowly evolving since. A major problem with performing variational calculus over an infinity of possible paths has been the immense computational resources required, even after suitable approximations have been put in place [8].

1.2 Molecular Dynamics Algorithm

The Feynman Path Integral performs a statistical average over all possible variations in the field returning observable measurements. The physical dynamics is governed by actions which provide the weights of each possible path in a manner similar to Boltzmann distributions in the thermodynamics of classical statistical mechanics. Feynman demonstrated the close connection between classical thermodynamics in three spatial dimensions and quantum mechanics in four dimensional spacetime with the mathematical transformation between real and imaginary time.

The mathematical connection between statistical mechanics and quantum field theory has allowed techniques between the two disciplines to be shared, and much of the basic terminology of gauge field theory originates from solid state physics. The observables in gauge field theory are calculated from expressions known as correlation functions and all the relevant physics can be extracted from corresponding partition functions. The algorithms for processing this data are also based on algorithms originally developed for molecular modeling based on classical mechanics [9].

In analytical quantum field theory the Green function propagators moving the field components across a spacetime continuum evaluate as inverses of the field equation operators. For example, the inverse of the Dirac equation for spin half fermions such as quarks for QCD or electrons for QED. When a lattice is introduced these operators become matrices that transport field components from lattice site to lattice site and the fermion propagators become known as hopping matrices, with a hopping parameter dependent on the mass of the spin-half particle. To evaluate measurables over the lattice it is necessary to transport field values to neighbouring lattice sites with these hopping matrices, and as in the continuum case, these propagations require inversion of the underlying field equation operators; but now they are matrices which require inversion.

QED and QCD gauge theories have the spin-half fermions propagating against a background field of gauge bosons. For QED the massless spin 1 bosons generate a gauge field with U(1) symmetry, whilst QCD has massive spin 1 bosons called gluons subject to SU(3) gauge symmetry with a triplet of color charges red, green and blue. The boson contribution is evaluated against the space of all possible values for the gauge field U, but even though the infinite continuum is approximated by a finite lattice, the exhaustive addition of all possibilities would take longer than the lifetime of the universe for any conceivable computational device. However, the main contributors to the overall result will tend to cluster, which allows importance sampling based on a Metropolis algorithm to be utilised. This generates a Markov chain of gauge field configurations

satisfying the principle of detailed balance and allows for an accurate calculation of ensemble averages [10].

Calculating the gauge field contributions allows the studying of the phenomena of quark confinement and demonstrates QCD based on non-abelian SU(3) can reproduce the fact that quarks are always found in colorless combinations and are never observed in isolation. The boson gauge field can be factored out from the fermion contribution, and in the case that the variations of the fermion field are ignored by setting det(Q) = 1, this is known as the quenched approximation; which were the only feasible calculations prior to larger machines becoming available in the 1990's.

To calculate the contribution of fully dynamical fermions, algorithms require the fermion propagator, which in turn requires the evaluation of the Dirac matrix inverse. There are theoretically an infinite number of possible lattice matrices which recover the continuum Dirac matrix in the limit of zero lattice spacing, and there are several that are routinely used for their particular properties. Wilson's original proposition removed over-counted copies known as the fermion doubling problem.

The fermion contribution is analytically calculated based on grassmann algebra resulting in an effective boson field contribution as before, and field updates evaluated from field components hopping to nearest neighbour lattice sites via the propagator matrix inverse. The correlation function between two lattice field values is given by

$$\langle \phi_i \phi_j \rangle = Z^{-1} \int \prod \mathrm{d}x_n \; \phi_i \phi_j \; e^{-\langle \phi_i | Q | \phi_j \rangle} = \left[Q^{-1} \right]_{ij} \tag{1.1}$$

The Hybrid Monte Carlo algorithm combines the molecular dynamics Markov chains of updates with a Metropolis accept-reject test, and the evaluation of the updated field configurations requires the calculation of new field states updated from the hopping propagator matrix inversion

$$\eta = Q^{-1} \left[U \right] \phi \tag{1.2}$$

with the pseudofermionic action being

$$S_{PF} = \sum_{i,j} \phi_j^* \; Q_{ij}^{-1} \left[U \right] \phi_j \tag{1.3}$$

The quantum observables are evaluated as correlation functions over the variations of gauge field values and fermion propagations with the configurations sampled using Markov chains and Metropolis acceptance. The inversion of the Wilson-Dirac matrix is the key computational step in this or any related procedure, and is the computational bottleneck for the overall algorithm.

1.3 The Lattice Dirac Matrix

The Dirac equation was created to model relativistic spin-half fermions, the electron in particular. The first quantization procedure is the same as one performs to create the Schrödinger equation but instead one uses the relativistic energy equation $E = mc^2$. The quadratic nature of Einstein's formula with respect to time caused issues with interpreting the wavefunction as a probability, so Dirac's idea was to factor Einstein's energy formula into two time-linear terms. This is impossible using real or complex numbers, but using matrices which anti-commute results in the following energy-momentum relationship,

$$\gamma^\mu p_\mu - mc = 0 \tag{1.4}$$

Under the rules of first quantization $p_\mu \to i\hbar\partial^\mu$ this classical relationship becomes the quantum wavefunction formula

$$(i\hbar\gamma^\mu\partial_\mu - mc)\,\psi = 0 \tag{1.5}$$

The gamma matrices must obey the anti-commutator relationship

$$\{\gamma^\mu, \gamma^\nu\} = 2g^{\mu\nu} \tag{1.6}$$

where $g^{\mu\nu}$ is the Minkowski metric of special relativity. The smallest possible matrices that obey this relationship are 4×4 matrices, of which there are several equivalent representations, with a common one being

$$\gamma^0 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix},\ \gamma^i = \begin{pmatrix} 0 & \sigma^i \\ -\sigma^i & 0 \end{pmatrix}$$

The sigma matrices are the Pauli 2×2 spin matrices originally used to model spin-half electrons within the Schrödinger equation, and here we can see that the Dirac equation includes the original spin formulation but produces two copies corresponding to matter spinors and their negative energy twins, the anti-matter spinors. The wavefunctions for the Dirac equation are thus comprise of four components grouped into two spinors, and are known as bi-spinors or Dirac spinors.

The negative energy states implied by the Dirac equation caused conceptual problems not that much more acceptable than the negative probabilities it was designed to avoid, but the discovery of the positron gave the theory undisputed validity. The incorporation

of special relativity is the root cause and is related to the break down of simultaneity of non-local events in the region of spacetime between past and future known as 'elsewhere'.

Depending on the relative velocity of two observers some events may be seen as happening in different orders, such as one observer seeing event A before event B, but another observer seeing event B first. These two events are not causally connected in 'elsewhere' so their order does not affect measurements or physical laws written respecting special relativity. It does mean though, that physical laws such as the Dirac equation may describe events that happen in reverse order depending on the observers frame of reference: so that a particle emitted at event A then absorbed at even B would be seen as matter, but then another observer may see the same particle travel backwards from event B to event A, and would interpret the same particle as anti-matter. By respecting special relativity the Dirac equation necessarily causes the admixture of matter and anti-matter states, as well as removing completely the notion of particle conservation since the total number of particles at a given time now becomes dependent on the simultaneity of a particular frame of reference. Feynman saw this as the ability of an electron to travel back and forth in time, and hence traveling backwards in time as permissible paths in the path integral sum of all possible histories. The negative energy states of antimatter are thus positive energy fermions observed traveling backwards in time.

The Dirac equation as given is for a single particle, but it shares the same structure as the many particle quantum field theory version, from which it follows as the single particle limit. For inclusion into a variational path integral the Dirac equation is given as a field Lagrangian density scalar, from which the differential Dirac equation may be recovered from the Euler-Lagrange equation for that density scalar.

$$\mathcal{L} = \overline{\psi} \left(\gamma^{\mu} p_{\mu} - m \right) \psi \tag{1.7}$$

This Lagrangian density needs to be discretized over a four-dimensional hypercubic lattice, where at each site n there is an independent four-component spinor ψ_n. An anti-symmetrically defined discrete derivative can be introduced for the changes in the anti-symmetric spinor values between sites of lattice spacing a in the four directions μ

$$\partial_{\mu} \psi \rightarrow \frac{1}{2a} \left(\psi_{n+\mu} - \psi_{n-\mu} \right) \tag{1.8}$$

The total action is given by adding the Lagrangian over all lattice site transitions

$$S_F = \sum_{n\, m} \psi_n Q_{nm} \psi_m \tag{1.9}$$

where, after a Wick rotation into a Euclidean lattice formulation, Q_{nm} is given by

$$Q_{nm} = \frac{1}{2}a^3 \sum_{\mu} \gamma_\mu \left(\delta_{n+\mu,m} \ - \delta_{n-\mu,m}\right) + a^4 m \ \delta_{nm} \tag{1.10}$$

Two point correlation functions give the transition amplitude for the particle to propagate from site n to site m, and is given by

$$\langle \psi_n | \psi_m \rangle = Z^{-1} \int \left[\mathrm{d}\psi_n \ \mathrm{d}\psi_m \right] \ \psi_n \psi_m \ e^{-\langle \psi_n | Q | \psi_m \rangle} = Z^{-1} \ det \left[Q \right]_{nm} \ \left[Q^{-1} \right]_{nm} \tag{1.11}$$

with the partition function Z

$$Z = \int \left[\mathrm{d}\psi_n \ \mathrm{d}\psi_m \right] \ e^{-\langle \psi_n | Q | \psi_m \rangle} = \ det \left[Q \right]_{nm} \tag{1.12}$$

resulting in the matrix inverse propagator between lattice sites n and m

$$\langle \psi_n | \psi_m \rangle = \left[Q^{-1} \right]_{nm} \tag{1.13}$$

This basic formulation causes fermions to be duplicated in the limit $a \to 0$ and is the unwanted fermion doubling problem that needed to be removed before successful computations could be undertaken.

Kenneth Wilson first used a discrete lattice to show that there are no free quarks in the strong-coupling limit of a quantized non-Abelian gauge field theory [7] and proposed the following workable discrete Dirac operator, known as the Wilson-Dirac operator matrix

$$Q_{nm} = \frac{1}{2}a^3 \sum_{\mu} \left[\ (1 + \gamma_\mu) \, \delta_{n+\mu,m} + \ (1 - \gamma_\mu) \, \delta_{n-\mu,m} \right] \ + \left(a^4 m + 4a^3 \right) \ \delta_{nm} \tag{1.14}$$

As the fermions move from site to site their wavefunctions now acquire a projection of $1 \pm \gamma_\mu$ instead of just γ_μ and this removes spurious doublers. A complete derivation, including Wick rotation into Euclidean space and the additon of the gauge field into the matrix, is provided in Appendix A.

1.4 Computational Complexity

The elements of the Wilson-Dirac matrix represent probabilities for particle transitions from a given state at a given lattice site to another state at any other lattice site. This state hopping between sites is only from site n to its nearest neighbours, with the path integral then building up the entire wavefunction amplitude as it is integrated over all possible paths. This feature results in very sparse matrices when elements are calculated for large lattices.

In the four dimensions of spacetime this results in eight connections for lattice site n to its nearest neighbours, with each site having four components of the Dirac spinor. In the case of QED the gauge interaction is a single complex number based on the U(1) gauge symmetry resulting in a matrix with a total of 8×4 off diagonal entries plus the transition from site n to itself for a total of 33 matrix non-zeros. For QCD the gauge symmetry is SU(3) so each of the four Dirac spinor components is a three element color-charge triplet giving a non-zero matrix element count of $8 \times 4 \times 3$ plus the diagonal gives 97.

Typical matrix sizes on modern systems may use lattice widths of 64, with the time dimension often being double. The matrix rank for this case would be $64^2 \times 128 \times 24 = 805,306,368$. This results in extremely sparse matrices, but unfortunately matrix inverses will generally not be sparse, and whilst they never require actual storage, the operation count to calculate results based on $|\phi\rangle = Q^{-1}|\psi\rangle$ still grows with time complexity $O\left(N^2\right)$. It should be noted also that just doubling the lattice size already has an $O\left(N^4\right)$ effect on the four dimensional spacetime lattice volume before the matrix is even created; thus exhibiting the curse of dimensionality with time complexity being of the order of the space dimensionality. Early LQCD results where performed in the late 1970's on lattice widths as small as 8, leading to matrices of width a few thousand, already pushed the computer hardware limits at that time.

Generation of the matrix elements themselves has only linear time complexity due to the fact that each row has a fixed number of elements. Since matrix row generation is only a linear cost, but storage of the complete matrix can be very costly, it is common that matrix elements are generated as required then discarded, to be regenerated when needed again. This is especially true when huge Dirac matrices are solved over distributed clusters and the movement of element data becomes prohibitive; it's just quicker to generate the elements again from scratch instead over sending it over a network. One issue though, is that the matrix elements are calculated from gauge field configurations, and this gauge field data needs sharing and updating throughout the Monte Carlo process. This necessitates some nearest neighbour communication, but nearest neighbour communication can be done, in principle, in a fixed amount of time, in parallel, irrespective of the lattice size.

The time taken to solve the matrix equation $|\phi\rangle = Q^{-1}|\psi\rangle$ is the key time complexity factor, and can dominate the entire Monte Carlo calculation of any of the main molecular dynamics algorithms; often up to 80% of the overall runtime. Exact algorithms for the solution of matrix equations, such as Gaussian elimination, suffer from the dual problems of having time complexity $O\left(N^3\right)$ but also the potential for numerical instabilities causing large errors in the final result anyway.

Group theoretic algorithms have been devised that take advantage of symmetries in matrix multiplication, with the 1969 Strassen's algorithm performing at time complexity $O\left(N^{2.8074}\right)$: given an efficient matrix multiplication an efficient matrix inversion can also be given [11]. The 1990 Coppersmith-Winograd [12] group theoretic algorithm performs at $O\left(N^{2.376}\right)$, and recent improvement by Virginia Vassilevska- Williams [13] gets the time complexity for exact solution down to $O\left(N^{2.3727}\right)$. The conjecture is that the lower bound is $O\left(N^2\right)$.

Exact methods suffer from a large time complexity, the group theoretic methods can are very involved and not parallel-cluster friendly, and suffer from potential numerical instability. Iterative methods are the primary alternative of choice, and consist of two classes of algorithm, the stationary Jacobi variants and the Krylov non-stationary methods. When these methods converge to a stable answer, they can often do so in $O\left(N^2\right)$ time complexity, and successful iterative algorithms progress by reducing errors so numerical stability is less likely to be a problem.

A key feature of the Wilson-Dirac matrices is that for the main region of physical interest the diagonal terms lose their dominance with respect to the non-diagonal terms; that is the sum of the diagonal terms becomes larger than the diagonal value. This is known as critical slowing down when the system tends towards the critical hopping parameter value with low mass fermions interacting over the entire lattice. This is a key feature of extracting measurables out of LQCD since the continuum limit of lattice spacing $a \to 0$ corresponds to long range interactions of the quarks over a large number of lattice sites (an infinite number of lattice sites when in the continuum limit of lattice sites infinitesimally close together). In condensed matter physics this is seen as a critical point when matter under goes a change of state (e.g. solid to liquid) and the equations of state undergo a second order transition.

Stationary iterative methods can completely break down when matrices lose diagonal dominance, and hence of limited use in their original form. Krylov methods are more robust against critical slowing down, but not immune since their performance is dependent on the condition number of the matrix; this will degrade as the critical hopping parameter is approached. Another aspect of the Krylov methods' performance is that one of the most effective variants relies on matrices being symmetric positive definite, the conjugate gradient method, and the Dirac matrices are anti-symmetric with complex eigenvalues. The choice is thus, either square the matrix to make it symmetric or choose more awkward non-symmetric variants; both approaches having their pros and cons.

1.5 Thesis Rationale

Leading edge investigation into lattice quantum field theory currently requires large supercomputing clusters and is available primarily to researchers in dedicated departments with access to sufficiently large budgets; and they are often in competition for resources with other large projects. Quantum chromodynamics is overwhelmingly the primary target for these resource intensive investigations. The QCD computations on the lattice are providing compelling evidence that the SU(3) gauge description is along the correct lines, whilst the mathematical proofs are the subject of one of the Clay Mathematics Institute's Millennium Prizes [14].

There is a large cross-over in ideas and techniques with many other branches of physics, not least solid state physics and condensed matter physics. Linear time complexity algorithms running on inexpensive hardware could open up research into these areas providing opportunities for research that are not available with current methods. The studies undertaken for massless 2-D Dirac fermions propagating over graphene lattices, being a good example of the areas that could be explored in more detail. Current methods tend to inspect non-time dependent properties using the Schrödinger equation, but a field theory approach could open up research into dynamic properties and ballistic transport.

The current hardware used for LQCD is mainly commodity processor clusters, such as a Beowulf cluster running Linux with MPI for intranode communication. A major bottleneck with such systems, above and beyond the higher than linear time complexity algorithms is the movement of data between computational nodes. High data bandwidth can become a major factor in the over runtime of some of these algorithms, especially when large matrices need to be communicated over distributed clusters.

The focus of this thesis is the design of a highly scalable matrix solver algorithm that has linear time complexity with respect to increasing matrix size. The scalability would ideally be with respect to small inexpensive commodity processing units, so that computational runtime could be halved by doubling the number of cheap processing units. At the outset of this thesis, the original idea was to use low-cost field programmable gate arrays, within which multipliers where available for a cost as low as US fifty cents each. Having software programmable hardware structures, they could be configured to optimize data paths between active arithmetic units, and so address data bandwidth issues by storing matrix element data as close as possible to, or even inside, bespoke arithmetic units required by the algorithm.

Chapter 2

Matrix Solvers

"There are 10 to the power 11 stars in the galaxy. That used to be a huge number. But it's only a hundred billion. It's less than the national deficit! We used to call them astronomical numbers. Now we should call them economical numbers."

Richard Feynman

The lattice Wilson-Dirac equation is normally solved with Krylov subspace algorithms, but there are many from which to choose, and even for a given choice there are a plethora of configurations and options. Algorithm time complexity is the key consideration given that the matrix size itself is dependent on the volume of the spacetime lattice, which itself goes as $O(N^4)$ against the lattice width N. For state of the art calculations with $N = 48$ or more, the calculation needs to be distributed across hundreds or thousands of processing units, in order to be executed within a reasonable length of time. This inextricably involves consideration of how data will be distributed across the network of processors, how much and how often that data requires broadcasting between nodes, and what proportion of the algorithm can be executed in parallel against the fraction that requires synchronization between serial blocks. Algorithms that minimize data bandwidth will necessarily be considered premium, possibly even at the expense of additional FLOPs. For example it is common that the Wilson-Dirac matrix is never directly stored, but regenerated from gauge field configurations each time element values are required.

2.1 Numerical Investigation

To review the properties of known algorithms as detailed in existing literature, a library framework was created for use by the research presented here. This library, named Werner, was written in ANSI standard C++ in such a way that it can be cross-compiled between the Windows-XP/7 Microsoft Visual Studio environment that was used for interactive development and debugging, and also compilable within the UNIX-GNU environment on the University of Wollongong's HPC cluster of 30 x 8 core DELL Power Edge 1435 Servers.

The Werner library will be discussed in greater detail in chapter 4.3, but shown in figure 2.1 is the command line for an application that provides Wilson-Dirac matrices for the Werner library to solve. Some key command line arguments are the target residual ($-uncertainty$), the name of the solver algorithm to use ($-solver$), the lattice width for the creation of the Wilson-Dirac matrix from randomly generated gauge fields ($-arg1$), and the effective mass corresponding to the hopping parameter ($-arg2$).

```
-sh-3.2$ time ./test1 --uncertainty 1e-14 --loops 1000 --depth 1 --solver Jacobi
--omega 0.67 --arg1 10 --arg2 40

MPI node 0 of 1. Processor: hpc24.its.uow.edu.au
Max Level: 64 Max Iterations: 1000
Tolerance: 1e-14 Epsilon: 0 Omega: 0.67 Depth: 1
Dirac.rank() = 240000
A.non_zeros  = 23280000
A.per_row_av = 97
-----------------------------------------------
                  Run Werner
-----------------------------------------------
INPUT RESIDUAL-NORM: 0
Loop: 0  dot(r,r) = 240000
Loop: 1  dot(r,r) = 114335
Loop: 2  dot(r,r) = 62909.3
...
Loop: 139  dot(r,r) = 1.55209e-14
Loop: 140  dot(r,r) = 1.20685e-14
Loop: 141  dot(r,r) = 9.38029e-15


 x = [ 0.31195; 0.262589; 0.237371; 0.682564; 0.40936; 0.00516332; 0.536908; ...

 b = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]
Ax = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]


real    1m19.604s
user    1m18.668s
sys     0m0.895s
```

FIGURE 2.1: Werner matrix solver command line.

The allowed range for the Wilson effective mass is

$$0.0 \leq -m \leq 2.0 \tag{2.1}$$

When the gauge field is completely random the effective mass will be 2.0 corresponding to a hopping parameter of $\kappa = 0.25$. When the gauge field is set to a unit-matrix field this corresponds to a completely ordered field and the effective mass is 0.0 with a hopping parameter of $\kappa = 0.125$. Finite lattice spacing normally has mass values over one; $-m \geq 1.0$ [15].

$$\kappa = \frac{1}{2(-m)a + 8} \tag{2.2}$$

In practice gauge fields are often initialized to random values and then as the Monte Carlo algorithm updates the critical value will lower as the field evolves and loses some randomness. The ill-conditioned nature of the matrices is at this critical point, and therefore the actual value of the critical hopping parameter will vary over the course of the simulation and is dependent on the exact values of the gauge field.

For the purposes of this thesis random matrices will be used so the critical mass value will be $m = -2.0$. This value is specified in the Werner software using $-arg2$, but scaled 0 to 200 for command line convenience and converted inside the program.

$$m = \text{–arg2} \ \div \ -100.0 \tag{2.3}$$

The residual-norm is calculated to measure the algorithm's progress, and is reported after each iterative step or loop. The residual is explicitly calculated rather than inferred from algorithm variables, as is sometimes the procedure; $r = Ax - b$ For an exact solution the magnitude of the residual r, given by the inner product $dot(r, r)$, should be equal to exactly zero. Once this actual value is below the requested $-uncertainty$, this returns the solution vector x at this requested precision. The program reports the first few values of x, along with b and Ax, as a visual sanity check.

All the numerical tests presented in this thesis were obtained on the University of Wollongong's GUR computer provided by UOW Information Technology Services. GUR consists of 30 nodes each with two quad core AMD Opteron 2356 Processors running at 2300MHz with 512kB cache, and each node sharing 16GB of RAM.

2.2 Stationary Methods

The Jacobi method is a simple stationary algorithm that benefits from a very high degree of parallelism. The idea is that the matrix to be solved can be applied to an initial guess in such a way that the error in the guess is reduced on each iteration, until the error has been reduced to the requested amount. The method is stationary in the sense that the same matrix is applied on each iteration, so never needs to be updated: this is an ideal feature from the point of view that there is no matrix element data needing to be updated across a network between each iteration.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} \; x_j^{(k)} \right) \tag{2.4}$$

The fact that each solution element can be updated independently is also a completely parallel operation. The only communication required is to update nearest neighbour vector elements $x_i^{(k+1)}$ between iterations. For a sparse matrix such as the Dirac-Wilson the cost of updating vector solution elements between each iterative loop is also minimal since only the vector elements corresponding to matrix non-zeros in that row are required. In the case of the Wilson-Dirac matrix that means $97 - 1 = 96$ vector elements need to be communicated to nearest neighbours, but this communication can be done in parallel, so the time-complexity cost is constant irrespective of how large the matrix gets.

Figure 2.2 shows the convergence to solution for an LQCD matrix with an effective mass value of $m = -0.4$ on a lattice of size $N = 12$. The boundary condition vector $\vec{b}$ was set to all ones so the entire matrix is involved in the calculation, and the stopping criterion is the usual 1e-14 (absolute, although relative is more common in practice). A key feature is that the residual steadily decreases in a monotonic manner making stopping criteria easy to judge.

For the Jacobi method to converge the standard condition is that the spectral radius ρ of the matrix applied in the iterative process should be less than one

$$\rho(B) = max \;\; |\lambda_i| \tag{2.5}$$

where λ_i are the eigenvalues of the matrix. For the Jacobi method the matrix applied B is

$$B = -D^{-1} \left(L + U \right) \tag{2.6}$$

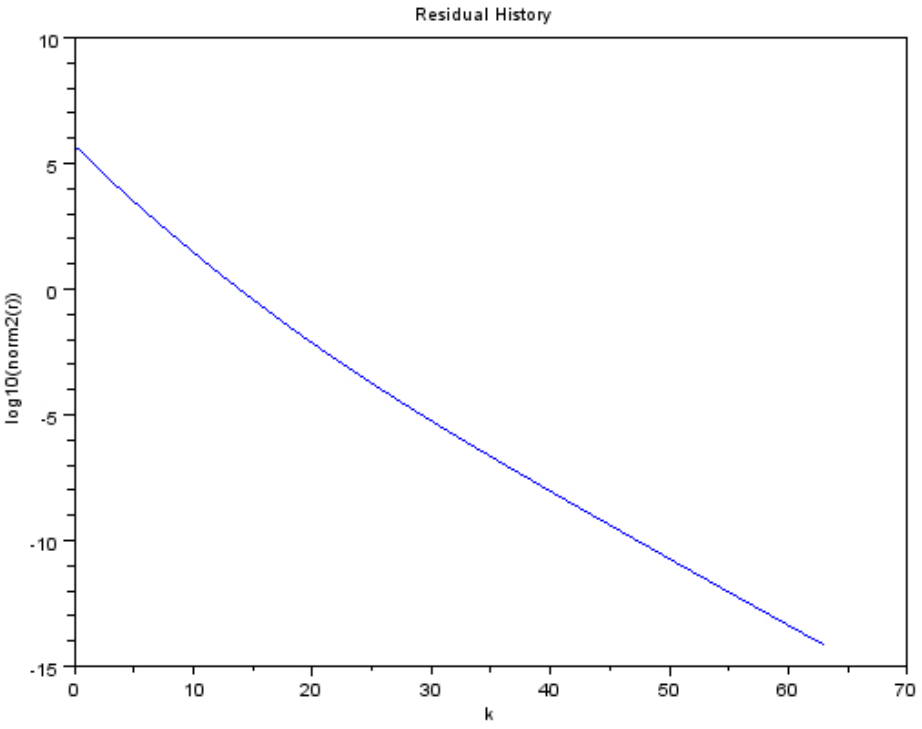


FIGURE 2.2: Jacobi convergence for high mass (m=-0.4) LQCD matrix.

where D are the diagonals, U and L being the upper and lower off-diagonals respectfully such that $A = D+L+U$. Convergence is guaranteed if the matrix is diagonally dominant:

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \tag{2.7}$$

The diagonal terms of the Wilson-Dirac matrix are given by

$$a_{ij} = (4 + m)\, \delta_{ij} \tag{2.8}$$

and with the off-diagonal values calculated from the gauge field links, the Wilson-Dirac matrix will generally be diagonally dominant for effective mass values $0.0 \geq m \geq -1.0$. At around -1.0 diagonal dominance begins to be lost, although the exact position depends on the hopping parameter critical value determined from the gauge fields evolved over the molecular dynamics algorithm.

Table 2.1 shows how the Jacobi method performs with respect to increasing lattice widths up to a maximum lattice width of $N = 20$, which was the maximum size of Wilson-Dirac matrix that could be supported by Werner on a single HPC cluster node. As the lattice increases in size the number of iterations required to find the solution vector

TABLE 2.1: Jacobi performance against LQCD matrix size.

Lattice Width	Matrix Rank	Iterations	Time / s
8	98,304	56	13.9
9	157,464	60	24.9
10	240,000	61	39.8
11	351,384	62	58.0
12	497,664	63	78.0
13	685,464	64	111
14	921,984	64	149
15	1,215,000	65	207
16	1,572,864	65	271
17	2,004,504	66	349
18	2,519,424	66	463
19	3,127,704	67	557
20	3,840,000	67	692

slowly increases, but otherwise the main time complexity dependency is the number of multiplications required to perform the matrix-vector product Ax, and hence is almost linear. Figure 2.3 shows the plot of time against matrix size for the Jacobi method with the diagonally dominant high mass valued LQCD matrices.

The fairly linear time complexity of the Jacobi method and its highly parallel nature are appealing but unfortunately the diagonal dominance issue prevents the Jacobi method from being of any great utility. Table 2.2 shows the effect of decreasing diagonal dominance on the performance of the Jacobi solver, with the number of iterations required to find solutions increasing rapidly as the mass parameter approaches -1.0 (arg2 = 100). For effective masses lighter than -1.0 the solver fails, with the numbers in brackets in the table giving the iteration number at which the algorithm begins to diverge after some initial convergence.

Figure 2.4 shows the convergence for a medium mass LQCD matrix at $m = -0.9$, at a value shortly before the critical value of $m = -1.0$. It can be seen that after an initial faster convergence, the convergence slows down, but does continue to a solution. In Figure 2.5 with a low mass of $m = -1.2$, the initial convergence reaches a minimum residual before starting to diverge again without ever finding a solution of the required accuracy. Figure 2.6 shows the worst case of mass value at its lightest $m = -2.0$, and at this point there is never any convergence: the iterations diverge from the outset. The plot of solver time against mass number is shown in figure 2.7.

The Gauss-Seidel stationary iterative method is a variation of the Jacobi method based on the idea that updating the vector elements $x_j^{(k+1)}$ immediately once they are available

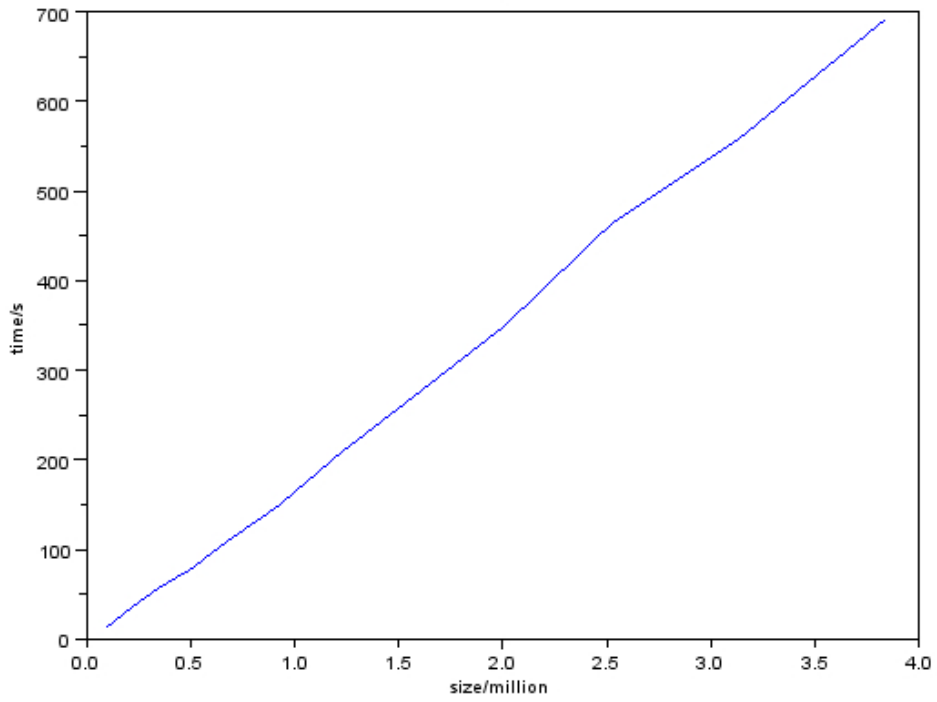


FIGURE 2.3: Jacobi performance against matrix size.

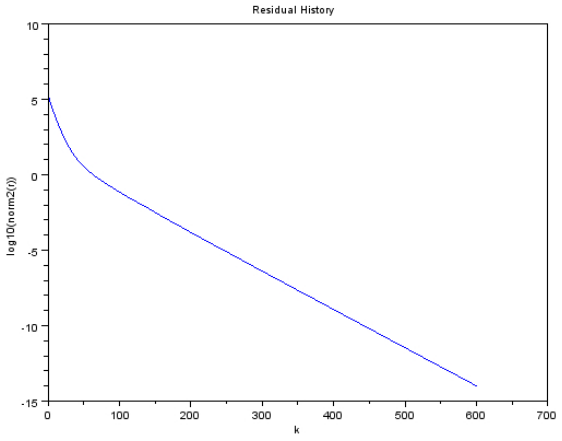


FIGURE 2.4: Jacobi convergence for medium mass (m=-0.9) LQCD matrix.

TABLE 2.2: Jacobi performance against LQCD mass.

mass	Iterations	Time / s
0	94	53
10	102	57
20	112	62
30	124	70
40	141	79
50	164	92
60	197	110
70	251	141
80	350	196
90	601	336
100	2426	1319
110	[60]	
120	[41]	
130	[33]	
140	[22]	
150	[14]	
160	[5]	
170	[3]	
180	[3]	
190	[3]	
200	[1]	

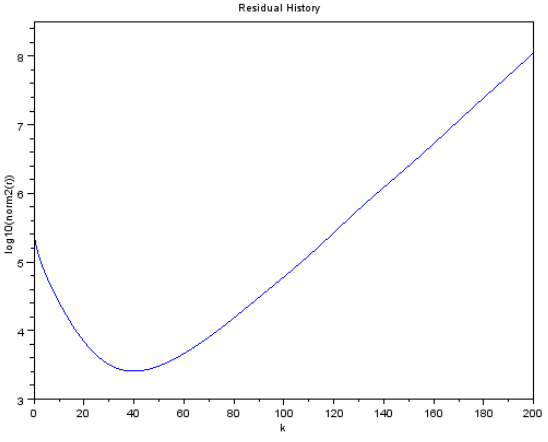


FIGURE 2.5: Jacobi divergence for low mass (m=-1.2) LQCD matrix.

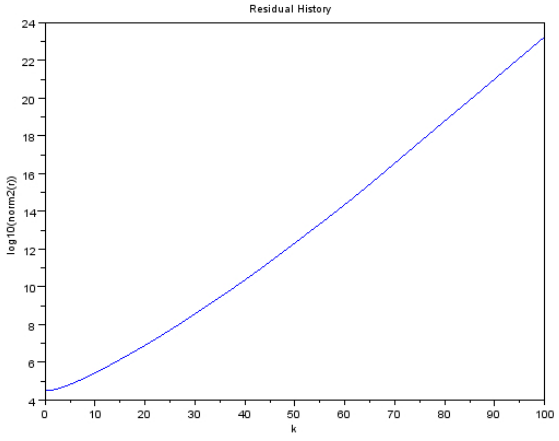


FIGURE 2.6: Jacobi divergence for lowest mass (m=-2.0) LQCD matrix.

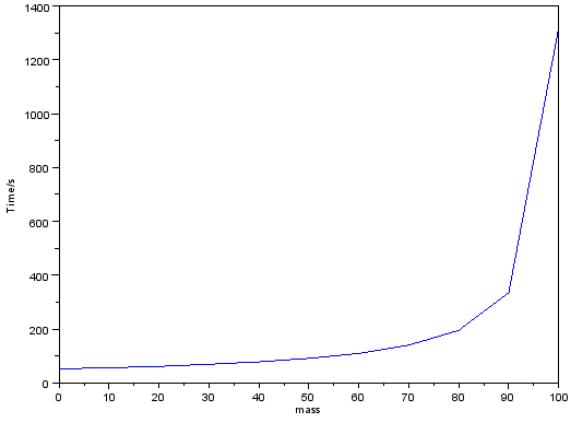


FIGURE 2.7: Jacobi iterations against matrix mass.

to calculate $x_i^{(k+1)}$ should be a improvement over waiting until the end of the iteration for all the updates.

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij}\ x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right) \tag{2.9}$$

The algorithm does show an improvement when executed sequentially in a single process, but for parallel clusters the communication overhead on each $x_j^{(k+1)}$ update becomes a significant overhead.

The stationary method can be further accelerated by applying relaxation, such as the relaxed Gauss-Seidel method known as Successive Over-Relaxation, SOR. ω is known as the relaxation factor and is typically in the range $0 < \omega < 2$.

$$x_i^{(k+1)} = (1-\omega)\, x_i^{(k)} + \frac{\omega}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij}\ x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)} \right) \tag{2.10}$$

Based on the Gauss-Seidel method it has the same parallelization drawbacks. The Jacobi method can also have the relaxation idea applied, and the option is present in the Werner library. When $\omega = 1.0$ the original algorithm is recovered.

2.3 Krylov Subspace Solvers

Krylov subspace methods do not rely as much on diagonal dominance and typically converge with fewer iterations; therefore fewer applications of the potentially expensive matrix-vector multiply operation. By repeated application of the matrix A to be solved, one builds up a subspace sequence of vectors that can be used as the basis for selecting update directions in the search for the global residual minimum [16].

$$K\left(A,r\right)=\text{span}\left\{r,Ar,A^2r,\ \ldots\ ,A^{k-1}r\right\} \tag{2.11}$$

When the matrix A is symmetric the residual minimum is at the lowest point of the hypersurface represented by the quadratic form

$$f\left(\vec{x}\right)=\frac{1}{2}\tilde{x}A\vec{x}-\tilde{b}\vec{x}+c$$

Application of the gradient 1-form, using $\tilde{\nabla}\vec{x}=1$, and $\tilde{\nabla}\vec{x}=I$ where I is the identity matrix, results in

$$\tilde{\nabla}f\left(\vec{x}\right)=\frac{1}{2}\left(\tilde{\nabla}\tilde{x}\right)A\vec{x}+\frac{1}{2}\tilde{x}A\left(\tilde{\nabla}\vec{x}\right)-\tilde{b}\left(\tilde{\nabla}\vec{x}\right) \tag{2.12}$$

$$\tilde{\nabla}f\left(\vec{x}\right)=\frac{1}{2}\tilde{x}A^T+\frac{1}{2}\tilde{x}A-\tilde{b} \tag{2.13}$$

For symmetric matrices the transpose equals the original matrix so

$$\vec{\nabla}f=A\vec{x}-\vec{b} \tag{2.14}$$

The solution of $A\vec{x}=\vec{b}$ can be seen as finding the minimum of the hypersurface, at the point where the gradient has become flat $\vec{\nabla}f=0$. At a given initial guess the direction of greatest decrease is identified and the iterative solution updated in that direction to the coordinate with minimal value along that direction. By repeating this process one eventually arrives at the global minimum required.

The basic idea of steepest descent can be augmented by instead choosing update vectors based on conjugate orthogonality. This algorithm can be shown to give an exact solution in precisely N iterations, but in practice the required tolerance can be reached far sooner. In principle this method requires keeping a history of prior updates directions against which to ensure orthogonality using a Gram-Schmidt procedure. Keeping a history of previous search directions can become expensive when the matrix rank numbers in the

millions, and also there is the overhead of performing the orthogonalization, which will add an additional $O(N)$ time complexity to the overall solver performance.

The requirement to keep the history of update vectors can be circumvented by instead using the concept of conjugate orthogonality with conjugate Gram-Schmidt orthogonalization process [17]. This allows a convenient recursive update trick that only requires the current residual and the previous residual, hence dispenses with the need to keep the entire history.

An update vector along one of the principal axes of the surface would be ideal since that would go immediately to the global minimum, but unfortunately this would require knowledge of the matrix eigenvectors; a related problem just as complicated.

2.3.1 Conjugate Gradient Normal Residual

Faster convergence and small memory overhead make the conjugate gradient method very popular, but for LQCD the requirement of being a symmetric matrix is not met: the Wilson-Dirac matrix is skew-symmetric with complex eigenvalues. Solving $A^T A\vec{x} = A^T\vec{b}$ instead allows the basic conjugate gradient algorithm to be applied to the now symmetric matrix of $A^T A$ [18]. This approach has several drawbacks, not least of which is the fact that the condition number squares, which greatly impedes the convergence rate to solution. A second issue is that the matrix sparsity is also affected, causing additional non-zeros that increase storage requirements and will also affect the communication overheads since there will be a greater number of nearest neighbour non-zero elements. For the systems that create the Wilson-Dirac matrix on the fly from gauge field links, rather store the matrix, there is again an increase computational burden to the creation of $A^T A\vec{x}$ for each iterative loop.

Table 2.3: CGNR performance against matrix size.

Lattice Width	Matrix Rank	Iterations	Time / s
8	98,304	128	167
9	157,464	132	247
10	240,000	133	306
11	351,384	133	484
12	497,664	131	737
13	685,464	138	989
14	921,984	146	1341

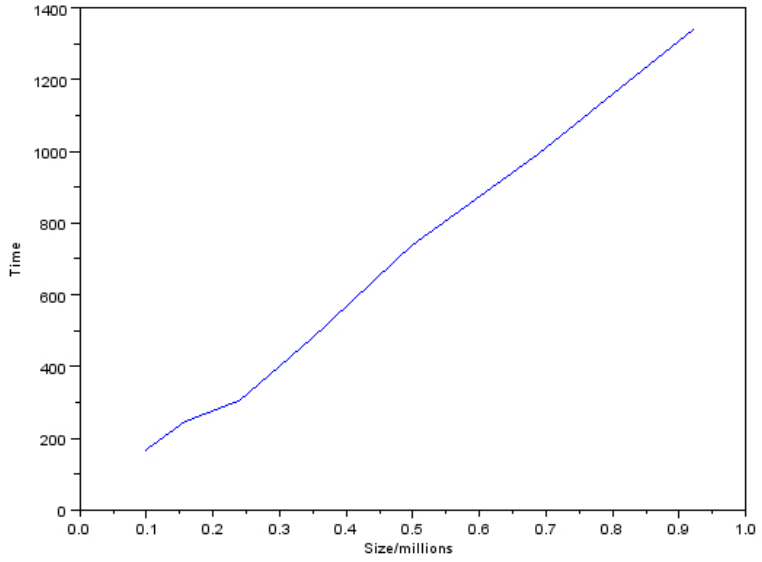


FIGURE 2.8: CGNR time taken against matrix size.

The figures 2.10, 2.11, 2.12 show the performance of CGNR for several difference mass values. For well-conditioned matrices with high mass, thus low hopping parameter, the convergence is smooth. For medium valued masses the convergence has periods where it stalls before recovering again, whilst the lowest mass value of -2.0 can be seen to converge after the longest number of iterations, but the initial convergence rate slows down without recovering again.

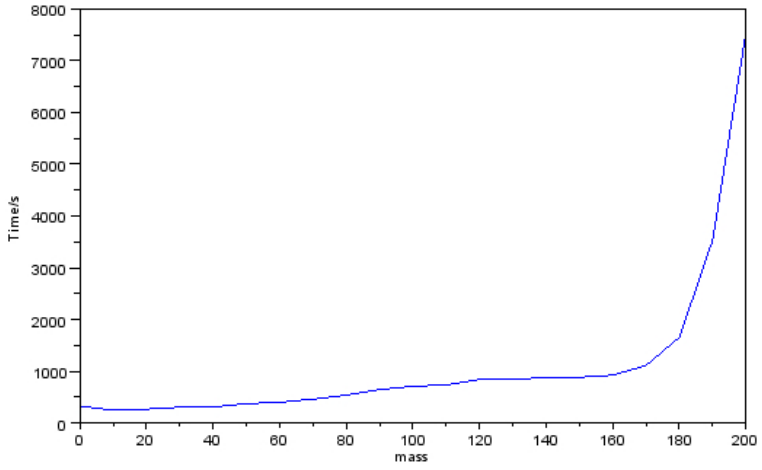


FIGURE 2.9: CGNR time taken against mass.

TABLE 2.4: CGNR performance against mass.

mass	Iterations	Time / s
0	89	322
10	96	250
20	107	268
30	120	305
40	133	311
50	150	373
60	174	401
70	207	460
80	259	537
90	305	645
100	351	711
110	388	734
120	436	844
130	441	853
140	468	869
150	457	883
160	519	924
170	679	1113
180	908	1655
190	1510	3535
200	4420	7545

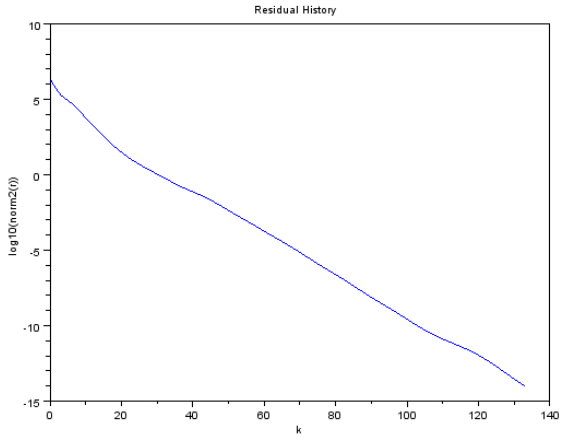


FIGURE 2.10: CGNR convergence for medium mass (m=-0.4) LQCD matrix.

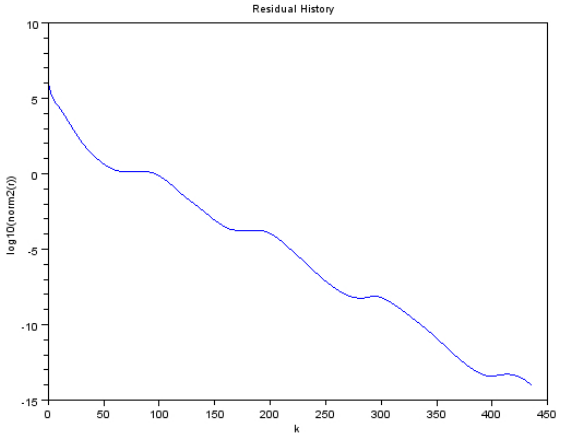


FIGURE 2.11: CGNR convergence for low mass (m=-1.2) LQCD matrix.

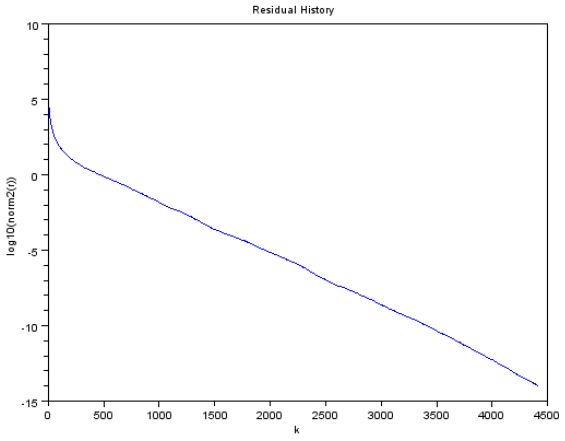


FIGURE 2.12: CGNR convergence for lowest mass (m=-2.0) LQCD matrix.

2.3.2 BiConjugate Gradient Stabilized

Another approach to non-symmetric systems is to use two mutually orthogonal sequences of residuals, one for A and one for A^T, leading to the BiConjugate Gradient method [19]. This method can suffer from numerical instability and breakdown, as well as irregular convergence. Several improvements are available with BiCGStab [20] being popular with faster and smoother convergence. BiCGStab uses a GMRES style stage to repair the tendency towards irregular convergence, but BiCGStab can still be likely to stall if the matrix has large complex eigenvalues. LQCD matrices will have increasingly large complex eigenvalues as the critical hopping parameter is approached, but there is an upper bound, so workarounds such as preconditioning are applicable. Table figure 2.5 shows BiCGStab performance against matrix size.

TABLE 2.5: BiCGStab performance against matrix size.

Lattice Width	Matrix Rank	Iterations	Time / s
8	98,304	33	13.4
9	157,464	36	23.0
10	240,000	38	35.0
11	351,384	41	55.1
12	497,664	38	72.7
13	685,464	38	92.1
14	921,984	39	141
15	1,215,000	40	191
16	1,572,864	39	247
17	2,004,504	41	320
18	2,519,424	41	370
19	3,127,704	43	472
20	3,840,000	42	655

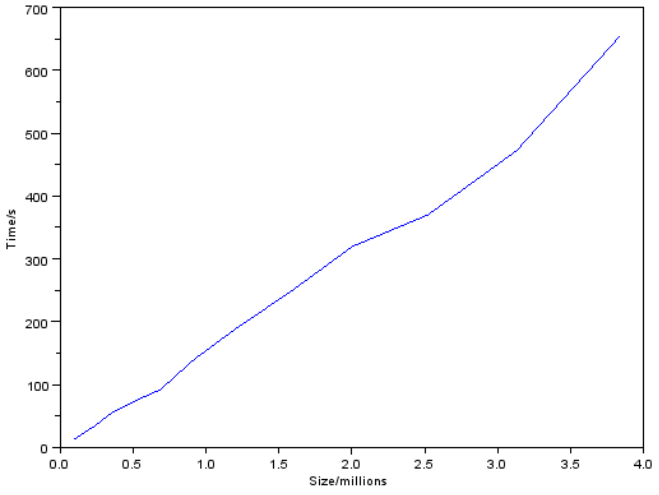


FIGURE 2.13: BiCGStab time taken against matrix size.

TABLE 2.6: BiCGStab performance against mass.

mass	Iterations	Time / s
0	26	23.7
10	29	27.1
20	30	28.8
30	34	31.3
40	38	34.6
50	42	37.5
60	53	46.0
70	60	50.0
80	79	60.8
90	123	88.3
100	195	140.2
110		

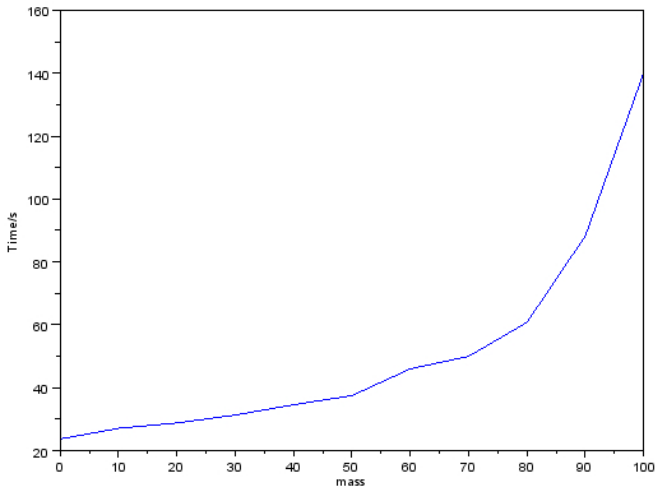


FIGURE 2.14: BiCGStab time taken against mass.

The figures 2.15, 2.16, and 2.17 show the performance of BiCGStab for several different mass values. The BiConjugate Gradient algorithm has not been widely adopted due to the erratic nature of its convergence and the difficulty in proving it converges at all. Even though BiCGStab was specifically designed to address these issues the erratic nature persists and becomes exacerbated by smaller the quark masses.

The ability to handle masses lower than -1.0 is not much better than the stationary methods of Jacobi or Gauss-Seidel, with convergence for mass -1.1 completely stagnating at a modest relative residual of 10^{-2}. The table in figure 2.6 and the graph in figure 2.13 show BiCGStab performance against mass.

The GMRES style part of the BiCGStab algorithm can also be extended to longer sequences of orthogonalized vectors. First developed was BiCGStab2 using two vectors, and then BiCGStab(L) generalizing to L-number of vectors to give a restarted GMRES(L) style procedure. BiCGStab and its offshoots have used extensively as an alternative to the A^TA CG-Normal approach since BiCGStab was introduced [21].

The irregular convergence and tendency to stagnate for critical values are still major issues with its usage to this day. However, for the non-symmetric Wilson-Dirac matrix, when it works, the BiCGStab method is one of the fastest solvers available.

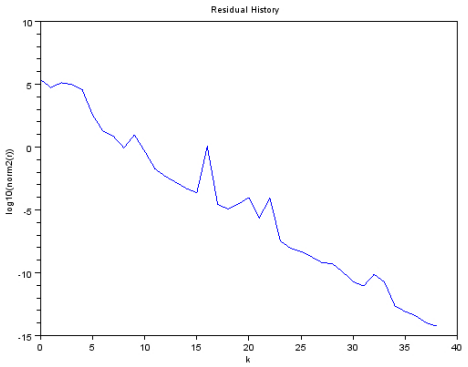


FIGURE 2.15: BiCGStab convergence for high mass (m=-0.4) LQCD matrix.

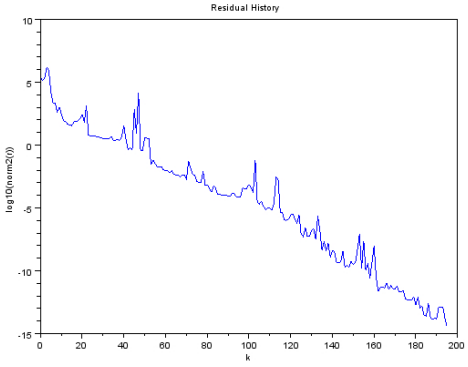


FIGURE 2.16: BiCGStab convergence for medium mass (m=-1.0) LQCD matrix.

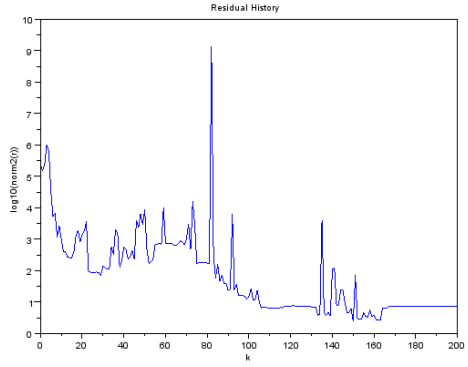


FIGURE 2.17: BiCGStab convergence for low mass (m=-1.1) LQCD matrix.

2.3.3 Generalized Conjugate Residual

Generalized minimal residual method, GMRES, generates a sequence of orthogonal vectors to minimize residuals $\vec{r}^{\,k} = \vec{b} - A\,\vec{x}^{\,k}$, but for non-symmetric matrices the benefit of short recurrences is lost and the history of previous update vectors needs to be stored again [22]. Restarts can be done to place a limit on the number of prior vectors that get taken into account, but at the cost of slowing down convergence when the update vector history is deleted and the algorithm restarts anew, albeit from a more accurate starting place. Faber and Manteuffel demonstrated the necessary and sufficient criteria for the existence of a conjugate gradient method and hence that it is not possible to have short-term recurrences and guaranteed residual minimisation for non-symmetric matrices [23].

A variant of the conjugate gradient known as conjugate residual can be generalized to non-symmetric matrices and is known as the generalize conjugate residual, GCR, method [24]. This method can be shown to converge to the exact solution for a non-symmetric matrix in exactly N iterations; where N is the size of the matrix. Whilst N is prohibitively large, a very useful feature of the GCR method is that after each iteration the residual is guaranteed to be smaller than that of previous loop. The GCR does require orthogonalization over the history of all previous update directions, which is an inhibitive as usual, so typically the method is employed with restarts as with the GMRES method.

Table 2.7: GCR performance against matrix size.

Lattice Width	Matrix Rank	Iterations	Time / s
8	98,304	64	18.3
9	157,464	67	29.9
10	240,000	69	47.5
11	351,384	69	71.0
12	497,664	70	120
13	685,464	72	146
14	921,984	74	224
15	1,215,000	75	268
16	1,572,864	74	346
17	2,004,504	75	424
18	2,519,424	76	555
19	3,127,704	76	711
20	3,840,000	77	895

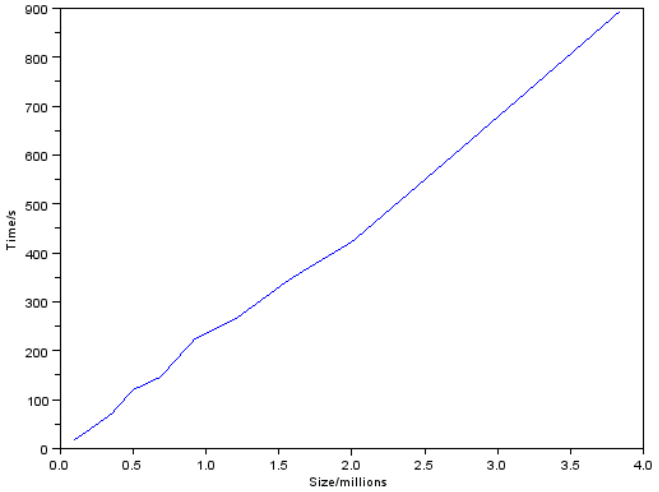


FIGURE 2.18: GCR time taken against matrix size.

The time complexity performance of GCR is almost linear since the dominant operation is the matrix-vector product with the matrix being sparse. The number of iterations required is slowly increasing though, which adds some quadratic nature to the overall time complexity.

Table 2.8 and graph in figure 2.19 show GCR performance against mass where it can be seen that ill-conditioned low mass matrices cause a significant blowout in the number of iterations required for solution. The performance of GCR is similar to CGNR, although roughly twice as good in terms of half the iterations required and half the total time taken.

TABLE 2.8: GCR performance against mass.

mass	Iterations	Time / s
0	49	37.1
10	53	38.6
20	57	43.8
30	62	50.8
40	69	53.1
50	77	61.9
60	89	67.1
70	108	81.2
80	136	110
90	169	146
100	224	215
110	286	318
120	365	492
130	444	665
140	590	1021
150	705	1300
160	919	1983
170	1153	2941
180	1494	4564
190	2163	10053

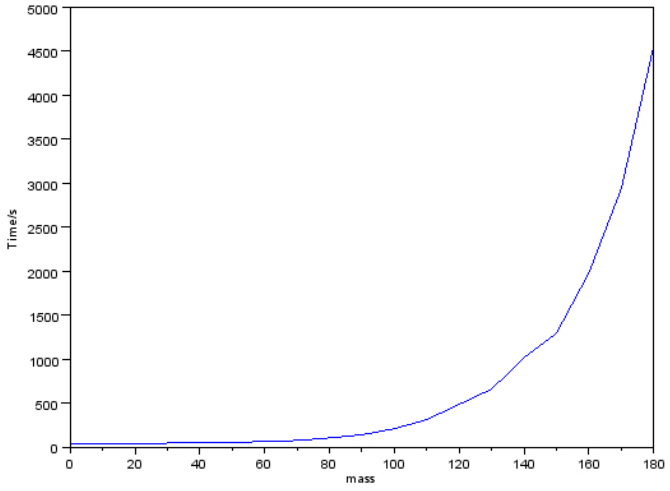


FIGURE 2.19: GCR time taken against mass.

The figures 2.20, 2.21, 2.22 show the performance of CGNR for several different mass values. High mass values perform well with smooth convergence whilst lower mass values still show monotonic convergence, but with possible periods of slow progress.

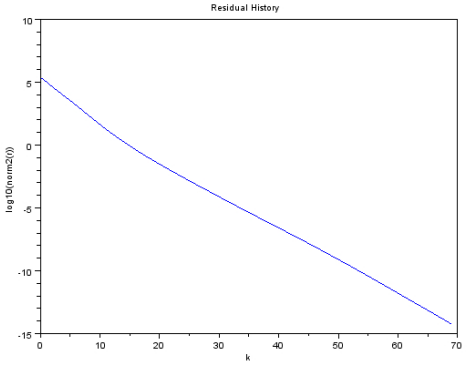


FIGURE 2.20: GCR convergence for medium mass (m=-0.4) LQCD matrix.

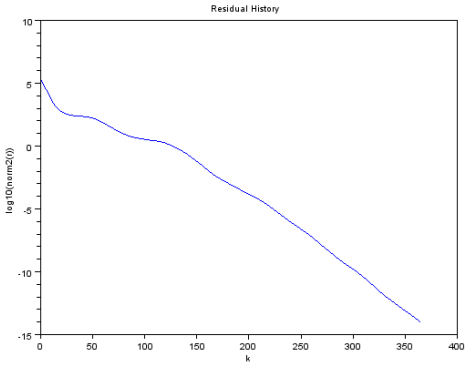


FIGURE 2.21: GCR convergence for low mass (m=-1.2) LQCD matrix.

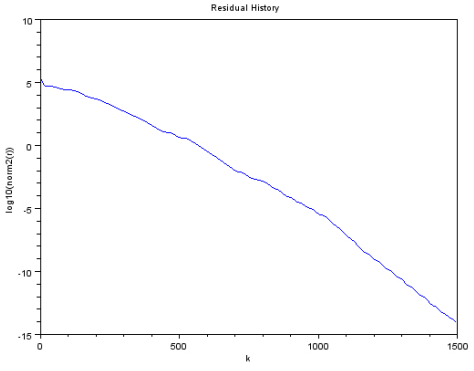


FIGURE 2.22: GCR convergence for very low mass (m=-1.8) LQCD matrix.

2.4 Multigrid

The Jacobi iteration works on the idea that repeated operation of the matrix on an initial guess $\vec{x}^{(0)}$ will cause the error $\vec{e}^{(k)} = \vec{x}^{(k)} - \vec{x}$ in that guess to attenuate, until arriving at the exact solution $\vec{x}$, to within the requested tolerance.

$$\vec{x}^{\,(k+1)} = -D^{-1}(L+U)\left(\vec{x} + \vec{e}^{k}\right) + D^{-1}\vec{b} \tag{2.15}$$

Upon recursive application with $B := -D^{-1}(L+U)$ leads to the result

$$\vec{x}^{\,(k)} = \vec{x} + B^{k}\vec{e}^{(0)} \tag{2.16}$$

The high frequency modes of the matrix will die away faster than the lower frequency modes with higher eigenvalues so the idea behind multigrid is to perform the iteration over many layers of grid coarseness. With coarser grids, what were low frequency modes will effectively become high frequency modes, and the Jacobi iteration, together with the other related stationary methods, become increasingly effective at those frequencies too.

This idea can be recursively applied and one ends up with the remarkable result that by applying a constant number of Jacobi iterations on each layer of grid coarseness, convergence to the solution can be achieved with the total work done being completely linear with respect to the number of matrix vector multiplications. That is, a completely linear time complexity solution.

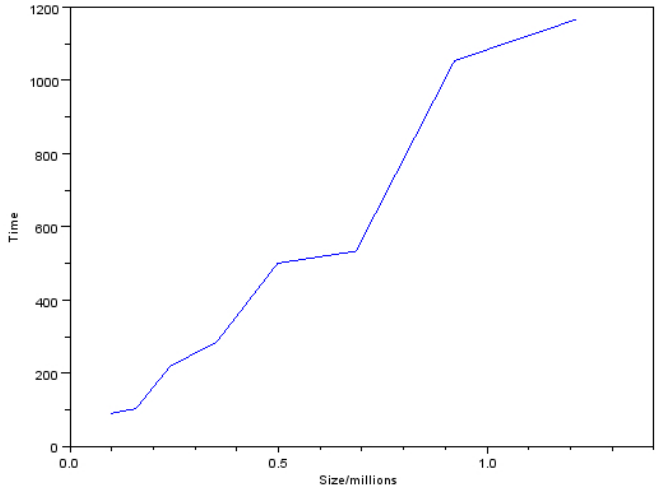


FIGURE 2.23: Multigrid time taken against matrix size.

TABLE 2.9: Multigrid performance against matrix size.

Lattice Width	Matrix Rank	Time / s	Depth	Residual
8	98,304	91	90	1.6E-07
9	157,464	104	90	4.4E-07
10	240,000	220	90	8.3E-07
11	351,384	286	100	9.4E-08
12	497,664	501	100	1.8E-07
13	685,464	534	100	3.8E-07
14	921,984	1054	120	2.3E-08
15	1,215,000	1167	120	2.2E-08

The data given in table 2.9 and plotted in figure 2.23 give an indication of the linear performance, with a caveat regarding the setup. The multigrid implementation is configured to apply the stationary method iteration a fixed number of times per grid-level rather than to terminate once a given residual has been achieved. Hence a value was chosen to return a reasonable approximation to the usual $\langle r|r\rangle = 10^{-14}$. This is specified as *–depth* for the Werner library, with its command line interface being shown in figure 2.24. For this experiment the smoother used was the over-relaxed SOR version of Gauss-Seidel with a relaxation parameter of $\omega = 0.67$.

The multigrid implementation is that of an Algebraic MultiGrid, or AMG, and for this Wilson-Dirac matrix of rank 1,215,000 the grid coarsening algorithm produces six grid levels over which to apply the SOR iterations. Each grid will have *–depth* = 120 repetitions of the SOR iteration applied in order to reduce the error in the solution guess at that level. The solution $\vec{x}$ returned to the command line application by the Werner library is then multiplied back into the matrix A as a sanity check to verify the boundary condition vector $\vec{b}$ is recovered as expected.

For the results given here the *–depth* value is slowly increased to keep the residual value comparable $\langle r|r\rangle = 10^{-14}$ so that timings may be compared to the Krylov and basic Jacobi results in the earlier sections. A linear trend is suggested but further results are restricted by the memory required to build the multigrid matrices for all the levels.

```
-sh-3.2$ time ./test1 --solver Multigrid --smoother SOR --omega 0.67 --depth 120 --arg1 15 --arg2 40

MPI node 0 of 1. Processor: hpc24.its.uow.edu.au
Max Level: 64 Max Iterations: 1000
Tolerance: 1e-14 Epsilon: 0 Omega: 0.67 Depth: 120
Dirac.rank() = 1215000
A.non_zeros  = 117855000
A.per_row_av = 97
--------------------------------------------------
                  Run Werner
--------------------------------------------------
Relative: 0.0    Order: 1215000  Level: 0
Relative: 0.391052      Order: 475128   Level: 1
Relative: 0.00862083    Order: 4096     Level: 2
Relative: 0.0227051     Order: 93       Level: 3
Relative: 0.0215054     Order: 2        Level: 4
Relative: 0.5           Order: 1        Level: 5
Relative: 1             Order: 1        Level: 6
OUTPUT RESIDUAL-NORM: 2.1841e-08

 x = [ 0.0347332; 0.185269; 0.670805; 0.377104; 0.420598; 0.462622; 0.374946; 0.171609; 0.252073; 0.164273; -0.129669; ]

 b = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]
Ax = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]
```

FIGURE 2.24: Multigrid command line execution.

TABLE 2.10: Multigrid time against depth dependency.

depth	time/s	residual
10	126	6.8
20	134	0.46
30	141	0.040000
40	153	0.004500
50	168	0.000700
60	198	0.000120
70	210	2.3E-05
80	220	4.3E-06
90	231	8.3E-07
100	243	1.6E-07
110	251	3.1E-08
120	261	6.1E-09

The *depth* plays a very important part in the multigrid algorithm since the error reduction operator needs to be applied enough times to achieve the desired residual; but excessive repetitions are just wasting time. The effect of the –*depth* parameter is shown in table 2.10. As expected, increasing the *depth* increased the accuracy of the solution. Setup time to create the matrix and multigrid levels was 110 seconds for the example used for figure 2.10, so one can see that each addition 10 in *depth* added about 10 extra seconds to the total runtime for this example.

The performance of multigrid with respect to mass is dependent on the stationary iterations applied to each level, thus follows the same performance against mass profile. For well-conditioned matrices of high quark mass the convergence time was fairly uniform but then the time to solve rapidly increased as the matrices lost diagonal dominance at $m = -1.0$.

TABLE 2.11: Multigrid performance against mass.

mass	depth	time / s	residual
0	80	201	7.0 E-09
10	80	209	3.1 E-08
20	80	210	6.8 E-09
30	100	222	1.9 E-08
40	90	217	6.0 E-10
50	90	237	6.0 E-10
60	90	212	6.0 E-10
70	200	333	1.6 E-08
80	250	398	9.9 E-08
90	500	754	3.2 E-09

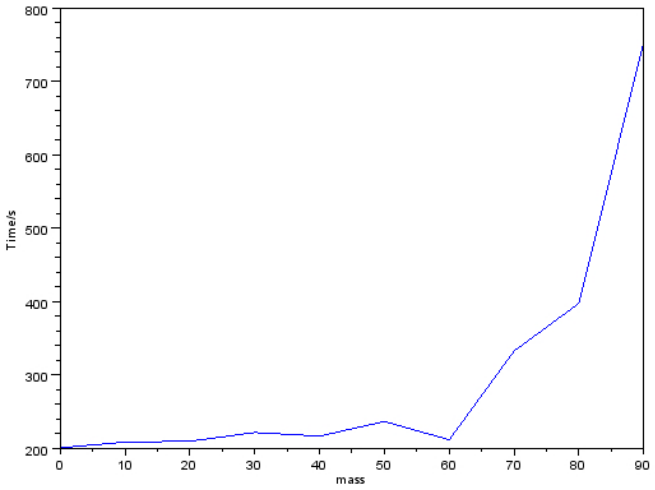


FIGURE 2.25: Multigrid time taken against mass.

The graph of time against mass for multigrid in figure 2.25 is very similar to the graph of time against mass for the Jacobi method shown in figure 2.7.

Multigrid is an extremely promising idea that works well for high mass quarks in the regime of wide lattice spacing, but a simple implementation fails to support the low mass continuum limit that is desirable for full scale LQCD calculations. A number of more sophisticated multigrid implementations have been attempted and have shown some promise [25].

2.5 Preconditioning

The conjugate gradient method can be viewed from a geometrical perspective in terms of the conjugated gradients descent, but an algebraic perspective also reveals the iterative structure of the method. Given an A-conjugate complete basis $|d_i\rangle$ for the matrix A to be solved such that

$$\langle d_i|A|d_j\rangle = 0 \;\forall\; i \neq j \tag{2.17}$$

where Dirac notation is being used for vectors and vector transposes. Any vector $|x\rangle$ can be expressed as a linear sum of the basis so

$$|x\rangle = \sum_{i=1}^{N} \beta^i |d_i\rangle \tag{2.18}$$

where the components β^i are given by the normalized projection of $|x\rangle$ onto $|d_i\rangle$,

$$\beta^i = \frac{\langle d_i|A|x\rangle}{\langle d_i|A|d_i\rangle} \tag{2.19}$$

Considering the meaning of a matrix inverse one can construct the matrix inverse in terms of the basis $|d_i\rangle$,

$$A^{-1}A\ |x\rangle = I\ |x\rangle = \sum_{i=1}^{N} \beta^i |d_i\rangle \tag{2.20}$$

$$\therefore\ A^{-1}A\ |x\rangle = \left(\sum_{i=1}^{N} \frac{\langle d_i|A|x\rangle}{\langle d_i|A|d_i\rangle}\right) |d_i\rangle \tag{2.21}$$

$$\therefore\ A^{-1}\ A|x\rangle = \left(\sum_{i=1}^{N} \frac{|d_i\rangle\langle d_i|}{\langle d_i|A|d_i\rangle}\right) A|x\rangle \tag{2.22}$$

resulting in the expression

$$A^{-1} = \sum_{i=1}^{N} \frac{|d_i\rangle\langle d_i|}{\langle d_i|A|d_i\rangle} \tag{2.23}$$

Thus the inverse A^{-1} can be algebraically constructed from the sum of an N-term series involving the A-conjugate basis $|d_i\rangle$. Each iteration of a Krylov subspace method can construct the k^{th} member of the basis and the sum converges to the answer as $k \to N$. As one would expect the matrix element A_{ii} is in the denominator of the expression.

The residual vectors at the k^{th} iteration $|r_k\rangle = |b\rangle - A|x_k\rangle$ are used to construct the next basis vector $|d_k\rangle$ in the Krylov subspace with

$$|d_k\rangle = K|r_k\rangle = K\left(|b\rangle - A|x_{k-1}\rangle \right) \tag{2.24}$$

$$\therefore \; |d_k\rangle = KA\left(A^{-1}|b\rangle - |x_{k-1}\rangle \right) \tag{2.25}$$

which after repeated application results in a formula for the residual involving a polynomial expression and the starting vector guess $|x_0\rangle$

$$|r_k\rangle = P_k\left(KA\right)\left(A^{-1}|b\rangle - |x_0\rangle \right) \tag{2.26}$$

$$\therefore \; |r_k\rangle = P_k\left(KA\right)|r_0\rangle \tag{2.27}$$

The kernel polynomial $P_k(KA)$ of the iteration method [26] has a set of solutions by virtue of the Cayley-Hamilton theorem stating that a square matrix satisfies its own characteristic equation.

$$P_N\left(KA\right) = 0 \tag{2.28}$$

Thus as $k \to N$ this causes $|r_k\rangle \to |0\rangle$ and the solution with zero residual is achieved after N iterations. If a convenient K can be found such that KA reduces the degree of the polynomial then the Krylov method will converge in fewer iterations, and then K is known as the preconditioning matrix.

In the perfect case of $KA = I$ then the polynomial is of degree one and a single Krylov iteration finds the solution immediately. This was observed in testing with Werner in the situation that a Multigrid preconditioner was configured to such a smoother repetition depth that the preconditioner itself returned to the required tolerance and the preconditioned Krylov step completed on the first iteration; that is the preconditioner had caused $K = A^{-1}$ and the solver was finished.

Grouping eigenvalues together until they become degenerate lowers the degree of the kernel polynomial so bunching the matrix spectrum together provides a potentially successful preconditioner. For diagonally dominant matrices something as simple as just the inverse diagonals may prove to be an effective preconditioner.

2.6 Wilson-Dirac Solvers

As the ill-conditioning of the matrices worsens as one tends towards the critical mass value for continuum observable values, this linearity factor decreases and the convergence decreases. This is described as critical slowing down, and this phenomenon severely limits the accuracy of lattice field theory calculations. There has been a lot of research into how to handle this critical slowing down for physical masses. Increasing the raw speed is a natural approach, and it has indeed been Moore's Law that has enabled dynamic fermions to be finally tackled above and beyond ignoring them in the quenched approximation.

A key numerical algorithms issue lies in the fact that Dirac matrices are anti-Hermitian and hence non-symmetric. The conjugate gradient algorithm performs so well since the symmetric matrices for which it is designed allows one to combine both guaranteed residual minimization with a short three term recurrence relationship. Thus sufficient iterations will definitely converge on the answer without the need to store the history of previous iterations. In theory for a matrix of order N the exact answer will be achieved after N iterations, but in practice the answer is achieved within a given tolerance much sooner. N iterations involving a matrix-vector product means the exact solution is time complexity $O(N^3)$ as with the classical exact methods such as Gaussian elimination. In practice the conjugate gradient will converge to tolerance with a smaller number of iterations so the effective time complexity is $O(kN^2)$ with k being the number of iterations.

For sparse matrices the matrix-vector product often has only a time complexity of $O(N)$, and this is the case for the Wilson-Dirac matrix since each row has 97 elements. Thus the time complexity is directly proportional to the number of rows N: $97 \times N$ $O(N)$. The number of iterations required often has dependency on the matrix size. As can be seen from the data in Chapter 2 the iterations required for the Wilson-Dirac solvers do slowly increase with the rank of the matrix; tables 2.3, 2.5, 2.7. The fact that this increase in iterations required is typically quite gradually with regards to matrix size,is the reason why the corresponding graphs look fairly linear over the range of data presented in Chapter 2.

Multigrid does offer the possibility of truly linear time complexity with respect to matrix size, but the time complexity of increasing matrix size is completely swamped by the critical slowing down issue of light fermions. For algorithms relying on diagonal dominance, such as Jacobi and multigrid smoothers, the light fermion matrices are completely intractable due to the iterative process being divergent. For the Krylov technique the issue is the condition number defined by the ratio of largest to smallest eigenvalues.

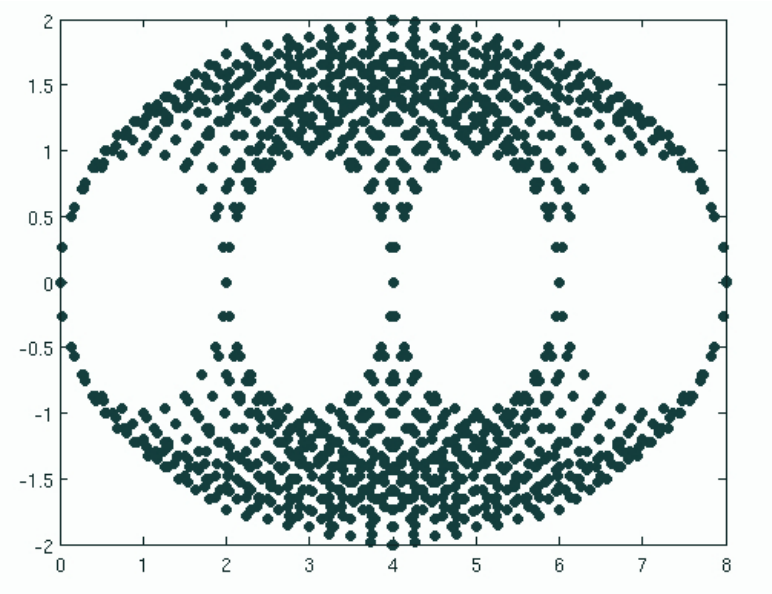


FIGURE 2.26: Eigenvalues of the Wilson-Dirac matrix.

As the critical hopping parameter is approached the smallest eigenvalue tends to zero, causing the condition number to grow extremely large. This very ill-conditioned nature of the critical mass fermions rapidly increases the number of iterations required to solve the LQCD matrices; as shown with tables 2.4, 2.6, 2.8. A plot of the complex eigenvalues for a Wilson-Dirac matrix is shown in figure 2.26, with the smallest eigenvalue visible close to the origin $(0, 0)$.

The conjugate gradient method can be used in the normal-equation form but the squaring of the ill-conditioning factor becomes prohibitively expensive as the convergence effectively stalls for critical values. BiCGStab does not guarantee the reduction of the residual on each iteration, hence the convergence becomes very erratic as fermion mass decreases, eventually to the point where convergence fails completely. Whereas BiCGStab can be fast but unstable, the GCR algorithm guarantees each iteration will produce a smaller residual, but GCR does not have the short-term recurrence relationship of CG or BiCGStab, hence the history of previous residuals must be stored against which to perform expensive orthogonalization. This greatly increases the required memory storage, and the convergence also slows down significantly with greater ill-conditioning as critical mass is approached. There are no known algorithms for non-symmetric matrices that have both the short-term recurrence and guaranteed residual minimization, and it has been proven that none can exist. In the case that the normal-equation variant of conjugate gradient is unacceptable, one is forced to choose between short-term recurrence or the guarantee of residual minimisation.

Chapter 3

Reconfigurable Hardware

"We have found it of paramount importance that in order to progress, we must recognize our ignorance and leave room for doubt."

Richard Feynman

Field Programmable Gate Arrays FPGAs are very large scale integrated circuits that can be configured at the level of transistor logic. Modern FPGAs can have millions of logic gates interconnected by electronic switches. This makes them highly versatile and often used for prototyping integrated circuit designs prior to fixing the final design ready for market. Since FPGAs are completely configurable, hardware level designs can be created especially tailored towards a specific problem. Data paths can be optimised and registers created to hold frequently used key data close to the arithmetic units that will be processing data. This chapter will explore the applicability of FPGAs and the challenges involved in using them for the problem of solving very large sparse matrices.

FPGAs are often used during the design and development stage of an electronic product to test an electronic design as a working prototype. Once the design has been verified the final logic design can be given to a wafer fabrication factory and the application specific integrated circuit ASIC can be manufactured knowing the design already works. This greatly reduces the cost and risk of having to rerun the very expensive wafer fabrication were the electronic design found to contain errors once added to a larger system.

FPGAs have been growing ever larger and more complicated to the point where they can be used in the final product themselves without the need to produce the ASIC at all. Addition of closely connected block memory BRAM and built-in multiply-accumulate

arithmetic MAC units have made them particularly attractive for digital signal processing DSP projects. If the product has room to fit these devices into the product package, then one can save on the cost of producing a more compact ASIC.

The MACs are built as 18 bit integer arithmetic units but can be cascaded together to allow for greater bit operations. Floating point operations can also be supported by configuring several MACs; for example a 64 bit multiplier can be built from nine elemental MACs [27]. Historically FPGAs were rarely considered feasible for floating point work due to insufficient MACs on all but the most expensive FPGAs whilst using raw logic gates resulted in slow performance and often engulfed the entire FPGA just to create a few multipliers [28]. Rapid increases in FPGA sizes and resources began to make floating point scientific computation feasible [29] and entire supercomputing systems such as the Cray XD1 were being built with integrated FPGA coprocessors [30].

FPGA for Lattice Quantum Chromodynamics has mainly been restricted to the area of coprocessor acceleration such as work on logarithmic multipliers [31], with matrix solvers typically being small scale [32] rather than for matrices with rows and columns numbering in the millions. The possibility of implementing Krylov methods within FPGAs eventually became a possibility with the availability of such devices s the Xilinx VirtexPro series of FPGAs [33].

Designing small circuits with FPGAs can be done using logic diagrams with wires connecting transistor gates, but any sizable design is generally performed with some higher level hardware description language, HDL. The main two choices for FPGAs have generally been the European created Verilog language favoured in the US, or the US created VHDL favoured in Europe. These HDLs were created to describe hardware designs so that the hardware could be simulated prior to fabrication. Now these two languages are used by FPGA design environments as input that can be used directly to compile underlying physical component descriptions known as netlists.

Whilst Verilog and VHDL can describe hardware and can be executed as a simulation within FPGA design environments, neither exist as a high level software language that can be executed as a stand alone program. To bridge this gap into high level programming environments additional software libraries have been created. JHDL and MyHDL are Java and Python hardware description language libraries that allow hardware descriptions to be executed as regular programs, but can also me made to output Verilog or VHDL to then be used as input for FPGA netlist compilation [34]. SystemC is a C++ library and can be used as a system level description language, encompassing both hardware and higher level system description.

An objective of this thesis was to investigate the possibility of using the smaller Xilinx Spartan FPGA as a computational accelerator since these can be purchased very cheaply in bulk and hence large homogeneous clusters could be very cost effectively constructed. This chapter will review and discuss some issues involved with building such a system, and in particular, with developing the system across the FPGA hardware platform and the software algorithms that need mapping into that hardware.

3.1 Software Integrated Co-Development

Hardware and software development are often engaged in as disparate disciplines. Even when a single system architect has visibility over an entire project it is not uncommon for separate teams to develop hardware and software components with entirely distinct toolsets, and often with incommensurate time-lines. Hardware and software development models tend to enforce these differences, whereby hardware may involve large turn around times when a bug is found, whilst software can often be patched with a quick fix when an issue is identified. Both classes of development model can have benefits and disadvantages.

With typical hardware development emphasis is often placed on up-front functional and timing specifications, followed by modelling and simulations, before a first tapeout and the delivery of the first version of silicon. The development life-cycle of an ASIC device can become expensive if the final stages need repeating too often. Software on the other hand has relatively small manufacturing overheads and iterative development is common practice. One of the major issues in software engineering is often how to manage the relative ease with which changes can be made: code changes, design changes and requirement changes can all be easily changed at any time. In system architectures involving both software and hardware providing integrated development for both hardware and software components in parallel can be a challenge at all stages, as the pace of development for both varies over the course of the entire development life-cycle. FPGAs can be reconfigured in a matter of minutes providing potential software style development timescales.

A key strength of bespoke hardware over a general purpose CPU is the ability to optimise algorithms in silicon, and in particular to exploit any inherent parallelism in the hardware [35]. Whist most general purpose CPUs will out-perform FPGAs in terms of raw processing clock speed FPGAs have the potential to execute algorithms with a high level of concurrency with an in-silicon design tailored for that algorithm [36]. This can lead to some very cost-effective designs with very high performance, often orders of magnitude better than general purpose CPUs in terms of cost per MFLOP. Also identified

as a key reason for the success of FPGAs in such areas as DSP has been their ability to be programmed in such a way as to sustain a continuously high level of computational throughput with optimaldata-flows implemented directly into the hardware [37].

A limitation for FPGAs has been the amount of resources within the FPGA required to undertake floating point operations, hence most applications have been restricted to fixed point or integer based applications. This is a significant limitation for scientific and engineering work that tends to require the large dynamic range provided by double precision floating point numbers. Improvements in floating point algorithms [38] and available intellectual property could combine to realise the potential of highly configurable hardware based on FPGA technology. Cray Incorporated have the XD1 supercomputer built on such an architecture and some energy efficient designs have been reported [39]. The RAMP project at UC Berkeley is developing massively parallel supercomputers with 1000s of CPUs built from racks of FPGA [40]. These projects highlight the need for integrated development environments to support programmers with compilers, operating systems and other useful abstraction models on these new highly concurrent platforms.

Several companies, along with research departments in both academia and industry, are creating tools for co-development and co-design: the tools and methodology necessary to develop the required hardware and software in parallel as an integrated process. The challenges include developing high level modelling languages that can be used effectively for software and hardware development such that both system development and testing can proceed within an integrated environment. Minimizing the number of tools required for development would have great benefits to both development productivity and overall system reliability. The ideal would be a single language for all designers, developers and testers to use. The current situation is that there tends to be many different tools required for each particular purpose which can lead to specialisms that may hinder communication in large projects spread over many teams, and it becomes increasingly harder to find individuals with good knowledge in all skill areas.

3.2 BLAS Applications

A key consideration for hardware acceleration of Basic Linear Algebra Subroutine BLAS packages is the nature of the inherent bottlenecks in the algorithm, and the performance of FPGAs versus commodity processors may hinge on which suffers least from the particular algorithm bottleneck [41]. The vector dot-product (DDOT) requires 2N memory accesses (where N is the size of the vector) to perform 2N arithmetic operations, and as such tends to be memory bound in modern processors where modern CPUs can perform at several gigaflops but the memory bandwidth tends to be in megabits per second.

Bandwidth communication is also an important limitation for FPGAs, but in the case of FPGAs their customizable nature allows a heavily pipelined architecture together with a specially modified MAC design that can show significant performance gains for larger vector sizes. For Matrix-Vector multiply, DGEMV in BLAS, there is again a memory bottleneck since there are two arithmetic operations per memory retrieval, and again the customisable nature of the FPGA allows an implementation in hardware tailored to the characteristics of the algorithm which in the case of DGEMV also improves the performance for smaller sizes of vector as well as the larger sizes as described above. AXPY, DDOT, and DGEMV are the critical operations in any Krylov subspace implementation.

For matrix-matrix multiply, DGEMM in BLAS, the arithmetic operations per memory access is now $N/2$, which gives an advantage to the faster CPUs for larger matrices, but the observation is that this imposes the often unrealistic requirement that both matrices be in lower level processor cache [42]. It is noted that commodity processors typically achieve only 80-90% of their peak performance on dense matrix multiplies while requiring large cache and memory bandwidth, whereas FPGAs can again take advantage of their reconfigurability to trade external memory bandwidth for internal memory cache in such a way as to construct the optimal setup to exploit the massive parallelization that the DGEMM exhibits when viewed as a set of N matrix-vector DGEMV multiplies.

3.3 Toolsets

Several toolset vendors and research institutions have been active in the area of trying to bring hardware development and software development closer together for the purposes of creating FPGA based applications. Their aims are to provide a system modelling language that is useful for specifying, designing, implementing and verifying both hardware and software modules, so called Co-Specification, Co-Design, Co-Development and Co-Verification. A unified language would help in an environment where it is becoming common for software developers, embedded developers and hardware developers to be using a variety of different design languages, incompatible tools and several design flow procedures. For example, it may be the case that the hardware designer is using VHDL in a hardware vendor specific IDE, with an embedded firmware engineer using assembler and C, with the front-end application developer using a higher level third generation language such as C++, Java or C# in yet a different IDE.

ImpluseC is a proprietary set of C-to-hardware tools that support the creation of custom FPGA hardware accelerators from C language source code. The tools use a process and streams-orientated programming model that is targeted towards FPGA-based mixed hardware and software applications. The design flow involves starting with a design

written in standard C. This design is done using the ImpluseC library and related tools collectively known as CoDeveloper which can be used within a traditional software environment such as Microsoft Visual Studio or GNU's GCC. Integration with existing software tool flows is said to be a key benefit. Once the design has been completed and simulated in the C environment the system is partitioned into hardware and software components, where the CoDeveloper can generate synthesisable hardware descriptions from the C description of the hardware components and automatically generate the required interfaces between hardware and software.

As Moore's Law continues to increase the level of FPGA performance, the more likely it becomes that the abstraction penalty of higher level languages becomes affordable, and low level programming optimizations become less frequently required. High-concurrency generative programming could then become an active area of research whose physical realisation would be feasible through the FPGAs that are now beginning to appear in the marketplace. Several such high level programming libraries are now beginning to emerge.

MyHDL is another example of a hardware description language embedded in a high level language, in this case the Python scripting language. Python is a modern scripting language supportive of the object orientated style of software, and is stated that Python's power and clarity makes MyHDL an ideal solution for high level modelling. MyHDL has the goal of providing high level language support to hardware designers by providing a complete set of embedded libraries to enable hardware to be designed and simulated in Python.

Execution of a simulation creates a standard VCD formatted output of waveform traces that can be displayed in any waveform viewer that supports the standard VCD format such as Gtkwave or BlueHDL. Through the usage of MyHDL, the built-in support that Python has for unit testing can be leveraged on the hardware models in a naturally integrated manner. MyHDL is also able to generate Verilog code from the MyHDL model, which can then be synthesized via the usual vendor tools; a future release of MyHDL also plans to support VHDL. It is also stated that MyHDL can also be used as a hardware verification language for VHDL and Verilog designs by co-simulation with traditional HDL simulators.

SystemC is a C++ class library that can be used for system design, simulation and validation. It has become internationally standardised though the efforts of the Open SystemC Initiative, OSCI, and is now gaining widespread acceptance within industry as a major tool for both the development of hardware systems and the specification of IP libraries. The SystemC standard describes the language as enabling users to write a set of C++ functions to model hardware processes that are executed under the

control of a scheduler that mimics the passage of simulated time, and are synchronized to communicate in a way that is useful for modelling electronic systems containing hardware and embedded software.

The SystemC library supports functional modelling by providing classes that represent: hierarchical decomposition of a system into modules using sc_module classes; structural connectivity between those modules using I/O ports with sc_port classes; scheduling and synchronization of concurrent processes using sc_event classes; the ability to independently refine computational implementation and communication using interfaces; and hardware-orientated data types especially for modelling digital logic and fixed-point arithmetic which is otherwise not directly supported by the native C++ language.

SystemC is a prime example of a domain specific language embedded within a general purpose high level language, and C++ itself provides excellent support for such generative embedded languages with its strongly typed template mechanism [43]. SystemC takes full advantage of C++ templates to provide a rich set of domain specific parameterizable types that the user can construct, and that the SystemC runtime libraries can use to elaborate the model and execute hardware simulations.

One major advantage of SystemC is the ability to describe a system at several levels of abstraction, from a high level functional description down to a synthesizable RTL level description. This allows SystemC to be used as a tool for architects to develop the chip concept as an executable specification, followed by software and hardware developers who can take the SystemC model and refine its component parts into modules that map into the physical hardware actually available.

The key here is that this process can now be undertaken as an integrated process with SystemC abstractions using a common and widely available C++ environment. SystemC as a hardware modelling language provides a very good bridge between hardware and software in that common software environments can interface directly to the SystemC model, hence software development and system testers can progress using an executable hardware specification, whilst the underlying SystemC implementation can be refactored by hardware designers into hardware modules physically present in the target device.

SystemC is currently mainly a simulation tool, and the SystemC libraries contain built-in support for the creation of VCD standard stimulation logic traces. There exists a wide range of tools to convert VHDL and Verilog into SystemC code, hence a common situation is for hardware designers to create their designs in VHDL or Verilog as usual, then the SystemC model can be automatically generated. This manufactured SystemC code is then given to software developers and system testers who can use the more convenient and readily available higher level language tools for their work.

For the reverse procedure of turning SystemC models into VHDL or Verilog for netlist synthesis in the case of turning a SystemC design into an FPGA implementation, it is still common to use the traditional method of handing the design over to a hardware expert for manual coding. There are a limited number of tools available that can turn a SystemC source code model into synthesisable hardware specifications [44], but most are expensive proprietary software such as the Synopsys CoCentric SystemC Compiler which generates RTL, and the Prosilog SystemC Compiler which can be integrated into Microsoft Visual Studio and can output VHDL or Verilog.

SystemC is also viewed as an opportunity to move away from hardware vendor specific HDL development and simulation environments which are typically licensed at some cost. IP held as a SystemC model is accessible through non-proprietary open source libraries and development environments and as such may encourage reuse and hence facilitate faster time-to-market for new hardware [45]. SystemC has also been identified as reducing time-to-market due to the linkage between the systems designers work and the hardware designers work, where the ability to support several levels of abstraction has been crucial, and enabled by the variety of powerful C++ types available within SystemC [46]. Such multi-level abstraction support also allows system testers to create test benches vertically through the layers of the software and hardware design.

The ability of SystemC to be used at several layers of abstraction for hardware specification, design, simulation and verification, together with its inherent ability to integrate into traditional software environments, suggests that SystemC is a good candidate as a language to support hardware and software integrated co-design, co-development and co-verification [47]. SystemC is a natural candidate as the language of choice for an integrated approach to mixed hardware and software co-development. Its applicability cuts vertically through both hardware and software abstraction and implementation, whilst being horizontally applicable to the areas of software design, hardware design and system verification.

By virtue of its host high level language C++, complete support is available for multi-paradigm reuse and development; namely programming to interface and parameterized generative programming. Refinement of high level designs and prototype implementations is supported through well established software techniques of refactoring and unit testing which could be vertically reused right down to the silicon implementation.

3.4 Integration

The increasing size of FPGAs and the progression of Moore's Law for their performance is making FPGA acceleration of software components progressively more attractive. The floating-point capability that will make FPGAs truly viable for general scientific computation is now within reach of many FPGAs, and specialised algorithms to utilise floating-point operations on them are beginning to materialise. FPGA-based supercomputing is in the early stages of research, but both research institutions and commercial companies are building FPGA-based machines with massively parallel potential; typically at a fraction of the prices that commodity processors would cost for similar concurrency. The low clock speed of FPGAs has another significant benefit in that this can make them highly energy efficient, making them low cost to run [48].

For some algorithms the ability to customise hardware at the silicon level enables FPGAs to outperform commodity processors, despite the fact that the raw clock speed is generally much smaller. Add to this the fact that highly scalable algorithms would be able to have their performance increased simply by adding more FPGAs to the setup, the cost-per-FLOP can be significantly better than that of commodity processors.

A common issue with FPGA custom hardware however, is the lack of programming tool support. Most of the existing support consist of low level tools and a disjoint set of higher level tools. The provision of an integrated high level environment would be of immense benefit to the community of scientists and engineers whose primary concern is within the realm of an application domain, or others interested in accelerating general purpose programming, and who are less concerned about low level details of FPGA synthesis.

Chapter 4

Software Development

"There is a computer disease that anybody who works with computers knows about. It's a very serious disease and it interferes completely with the work. The trouble with computers is that you 'play' with them!"

Richard Feynman

SystemC has been chosen to build a hardware model of a potential Krylov method accelerator to be embedded inside a high concurrency FPGA device. The SystemC model can be executed on a desktop environment, and both the logic trace output of the simulator and the numerical results can be inspected. The model should keep to interface blocks that are easily translated into VHDL interfaces with the bottom layer components requiring direct translation into actual FPGA logic blocks.

With the ideas for the FPGA accelerator in place, a C++ framework to investigate solver methods is created. Within this framework algorithms can be rapidly prototyped and investigated for the convergence properties with respect to Wilson-Dirac matrices. The solver methods investigated within the C++ application framework should be kept to the DDOT and DGEMV primitives to be implemented in the FPGA to avoid the necessity of adding further features to the hardware development.

The C++ application framework can provide the interface into the FPGA implementation for LQCD client applications by providing the ability to chose the C++ implementation for solver execution, or instead choosing to delegate the matrix solution to the FPGA accelerator.

4.1 SystemC Components

The basic units required for implementing the conjugate gradient algorithm in an FPGA are adders and multipliers. Figure 4.1 shows a VHDL interface for a floating point adder. In VHDL the interface is known as an entity and represents the wires, or electronic ports, going in or coming out of the component. Common electronic functionality is provided, with the clock input CLK used to synchronise functionality, the reset RST, and enable which allows a component to be switched off when the input is not intended for that target. This interface specifies 64-bit operands for input with the output being in a format of a 56 bit mantissa with 11 bit exponent and a sign bit.

The actual implementation of this component interface entity is achieved in another file that may either be an explicit coding of the logic in terms of FPGA logic blocks, or specialised multiply-accumulate MAC blocks. The MAC blocks being the dedicated arithmetic units are the preferred choice, but they still needed wiring together in floating point format, since they are provided as 18-bit integer components. These implementations of the entity interface are known as architectures and it is possible to have multiple architectures implementing the same interface. For example an FPGA might be configured with 10 adder-units, with 7 made from DSP MACs and the other three composed from FPGA logic. The DSP MACs would be preferred since they are faster than the generic reconfigurable logic blocks.

Figure 4.2 shows a SystemC interface definition for a floating point adder. With SystemC being a C++ library, the syntax is extended using C++ classes and uses a combination of SystemC libraries and in-built types. The same data in and data out ports can be seen in common with the VHDL entity from figure 4.1 along with the clock and reset. A few extra ports relating to the management of arithmetic units such as underflow and overflow are also present. SC_MODULE is a C language macro that maps to a C++ class, and is used to declare the name of the interface with SystemC required prefixes.

```
ENTITY fpu_add IS

PORT(
      clk         : IN     std_logic;
      rst         : IN     std_logic;
      enable      : IN     std_logic;
      opa         : IN     std_logic_vector (63 DOWNTO 0);
      opb         : IN     std_logic_vector (63 DOWNTO 0);
      sign        : OUT    std_logic;
      sum_3       : OUT    std_logic_vector (55 DOWNTO 0);
      exponent_2  : OUT    std_logic_vector (10 DOWNTO 0)
    );
END fpu_add;
```

FIGURE 4.1: VHDL interface for a floating point adder unit.

```
SC_MODULE(FPAdder)
{
  sc_core::sc_in<bool> CLK;
  sc_core::sc_in<bool> port_reset;
  sc_core::sc_in<bool> port_begin;
  sc_core::sc_in< sc_dt::sc_bv<2> > port_rmode;
  sc_core::sc_in< sc_dt::sc_bv<FP_SIZE> >  port_opA;
  sc_core::sc_in< sc_dt::sc_bv<FP_SIZE> >  port_opB;

  sc_core::sc_out< sc_dt::sc_bv<FP_SIZE> > port_outP;
  sc_core::sc_out<bool> port_ready;
  sc_core::sc_out<bool> port_underflow;
  sc_core::sc_out<bool> port_overflow;
  sc_core::sc_out<bool> port_inexact;
  sc_core::sc_out<bool> port_exception;
  sc_core::sc_out<bool> port_invalid;

  void main_thread();
  void reset_process();
  void begin_process();

  SC_CTOR(FPAdder) : result( '0' ), cycle_counter( 0 ), update_reset( false ),
    update_begin( false ), update_counter( false )
  {
    port_outP.initialize( 0 );

    SC_THREAD(main_thread);
    sensitive << CLK.pos();

    SC_METHOD(begin_process);
    sensitive << port_begin.pos();

    SC_METHOD(reset_process);
    sensitive << port_reset.pos();
  }

  const static int CYCLE_COUNT = 20;
  sc_dt::sc_bv<FP_SIZE> result;
  unsigned int cycle_counter;
  bool update_reset;
  bool update_begin;
  bool update_counter;
};
```

FIGURE 4.2: SystemC module for a floating point adder unit.

The SystemC module contains functions that simulate the running of the hardware component: reset, begin, and the main_thread executing the primary function of the component. The class constructor defined by the macro SC_CTOR associates those functions with input signals using an overload of the $\ll$ operator. SystemC uses the C++ template mechanism to configure data types of the data in and data out ports with single logic signals being of type *bool* for a single logic bit. Words and bytes are grouped together as bitvectors, with the template sc_bv instantiated with a predefined compile time constant. In the example of figure 4.2 the data width is that of a floating point number whose width is given by the value FP_SIZE.

The SystemC module can map directly to the VHDL entity and tools are available to translate between them [49], but the most useful tools are being sold by companies such as ImpulseC [50] are targeted at commercial markets.

The hardware architecture of the FPGA component is modeled using SystemC within C++ implementation files. The example for the main_thread function of the FPAdder is shown in figure 4.3. SystemC launches these components within its executing environment as infinite loops to model the hardware in continuous operation. The *wait*() call triggers on each clock cycle to model synchronous clocked behaviour, then executes the body of the code block to update the behaviour of the device.

For the FPAdder the required behaviour is to perform a reset if the reset input is logic high, or to begin a sum if the begin input line is toggled high. The third code block keeps track of how many clock cycles it's been since the begin request was received. After the specified amount of time given by FPAdder::CYCLE_COUNT, the result is written to the data out port and the ready signal is set to true. For the floating point adder provided by the VHDL design, it specified that the FPGA would take 20 clock cycles to perform a floating point addition.

```
void FPAdder::main_thread()
{
  while( true )
  {
    wait();

    if ( this->update_reset )
    {
      port_ready.write( false );
      port_outP.write( this->result );
      this->update_reset = false;
    }

    if ( this->update_begin )
    {
      port_ready.write( false );
      port_outP.write( 0 );
      this->update_begin = false;
    }

    if ( this->update_counter )
    {
      if ( this->cycle_counter < FPAdder::CYCLE_COUNT )
      {
        this->cycle_counter++;
      }
      else
      {
        port_outP.write( this->result );
        port_ready.write( true );
        this->update_counter = false;
        this->cycle_counter = 0;
      }
    }
  }
}
```

FIGURE 4.3: SystemC architecture for a floating point adder unit.

Once the SystemC module has defined the component interface one can create a Test-Bench to verify the module behaves as expected. Test benches in hardware design are the equivalent of unit tests in software engineering practice. Design methodologies often state that unit tests should be created immediately after the interface has been designed and prior the actual implementation of working code. Often the exact order of file creation can vary and in practice is often an iterative process that ends up developing files in parallel. In large teams one may have different people responsible for each aspect of the overall design: interface design, unit testing, implementation coding.

Figure 4.4 shows the test-bench function for the floating point adder. The test creates two data signals $A = 4$ and $B = 7$, writes them to the component input ports, then toggles the port_begin to trigger the execution of the component's functionality. The FPAdder component and its test bench are executed within the SystemC environment and the logic signals can be saved to file. The logic signals are saved in the IEEE-1364 specified Value Change Dump VCD format and can be displayed using the open source GTKwave viewer. Figure 4.5 shows the waveform generated from the FPAdder test bench. The inputs values are set, the begin signal is toggled, and twenty clock cycles later the answer $4 + 7 = B$ arrives at the output port, along with the ready signal switching to true. Client components wait for this ready signal to go high before using the output value.

```
void register_test()
{
  sc_dt::sc_bv<FP_SIZE> A = 4;
  sc_dt::sc_bv<FP_SIZE> B = 7;

  wait();  // cycle 1

  wait();  // cycle 2
  port_opA.write(A);

  wait();  // cycle 3
  port_opB.write(B);

  WAIT_CYCLES( 8 );
  port_begin.write(true);

  wait();  // cycle
  port_begin.write(false);
}
```

FIGURE 4.4: SystemC test bench for a floating point adder unit.

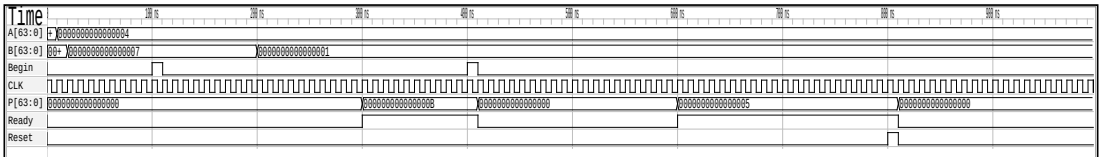


FIGURE 4.5: SystemC test bench logic trace output.

4.2 SystemC Algorithm

With the basic components modelled and tested SystemC can then combine those components into the higher level algorithm and external interface. Figure 4.6 shows the architecture code for a conjugate gradient algorithm in SystemC. This top level component organises the lower level adder and multiplier components by selecting the data to use for input and then toggling the begin ports. This top level component is named CGController.

There are two concurrent processes executing inside the CGController, the one shown here and a second process *cg_mac*() which is the core kernel that implements the matrix-vector product $A\vec{p}$. Implemented in hardware logic blocks, the algorithm becomes selecting vector element data from Block-RAM, copying it into input buffers ready for the arithmetic units to use, toggle the begin, then after the component's latency delay the answer appears at the output port and the ready flag goes high.

The CGController is a sequence of selections such as $port\_mult\_selectA1.write($ "011"$)$. This tells the A1 multiplier block to use the third input at this stage of the calculation, which here would correspond to the conjugate gradient alpha value used for updating solution vectors and update directions. In this way the algorithm is built up in hardware. Figure 4.7 shows the logic trace from GTKWave for an iteration of the CGController.

```
void CGController::cg_algorithm()
{
  this->cg_mac(); // Sum( Aij * Pj )
  port_sum_selectA2.write("111"); port_sum_selectB2.write("100");
  port_sum_selectS2.write("000110");  //Ri & Pi
  port_sum_begin.write( true ); wait(); port_sum_begin.write( false );
  do { wait(); } while ( false == port_sum_ready.read() );
  port_mult_selectA1.write("110"); port_mult_selectB1.write("111");
  port_mult_begin.write( true ); wait(); port_mult_begin.write( false );
  do { wait(); } while ( false == port_mult_ready.read() );

  while ( true )
  {
    this->cg_mac();  // Sum( Aij * Pj )
    //  alpha_i = Pi * Zi;  T1 = A1 * B1;
    port_mult_selectA1.write("100"); //Pi
    port_mult_selectB1.write("101"); //Zi
    port_mult_begin.write( true ); wait(); port_mult_begin.write( false );
    do { wait(); } while ( false == port_mult_ready.read() );
    // Setup the IOMuxes.
    port_mult_selectA1.write("011"); // alpha
    port_mult_selectB1.write("100"); // Pi
    port_mult_selectA2.write("011"); // alpha
    port_mult_selectB2.write("101"); // Zi
    // Accumulate the Dot-Product with the CGSumBlock.
    port_sum_selectA1.write("110");     //Xi
    port_sum_selectB1.write("000");     //T1
    port_sum_selectA2.write("100");     //Ri
    port_sum_selectB2.write("001");     //T2
    port_sum_selectS1.write("100000");  //Xi
    port_sum_selectS2.write("001000");  //Ri
    port_mult_begin.write( true ); wait(); port_mult_begin.write( false );
    do { wait(); } while ( false == port_mult_ready.read() );

    port_mult_selectA1.write("110"); //Ri
    port_mult_selectB1.write("111"); //Ri
    port_mult_begin.write( true ); wait(); port_mult_begin.write( false );
    do { wait(); } while ( false == port_mult_ready.read() );

    // Setup the IOMuxes.
    port_mult_selectA1.write("010"); // invAii
    port_mult_selectB1.write("110"); // Ri
    port_mult_selectA2.write("100"); // Pi
    port_mult_selectB2.write("011"); // beta
    // Accumulate the Dot-Product with the CGSumBlock.
    port_sum_selectA1.write("000");     //T1
    port_sum_selectB1.write("001");     //T2
    port_sum_selectS1.write("000100");  //Pi

    port_mult_begin.write( true ); wait(); port_mult_begin.write( false );
    do { wait(); } while ( false == port_mult_ready.read() );
    port_sum_begin.write( true ); wait(); port_sum_begin.write( false );
    do { wait(); } while ( false == port_sum_ready.read() );
  }
}
```

FIGURE 4.6: Conjugte gradient algorithm implementation in SystemC.

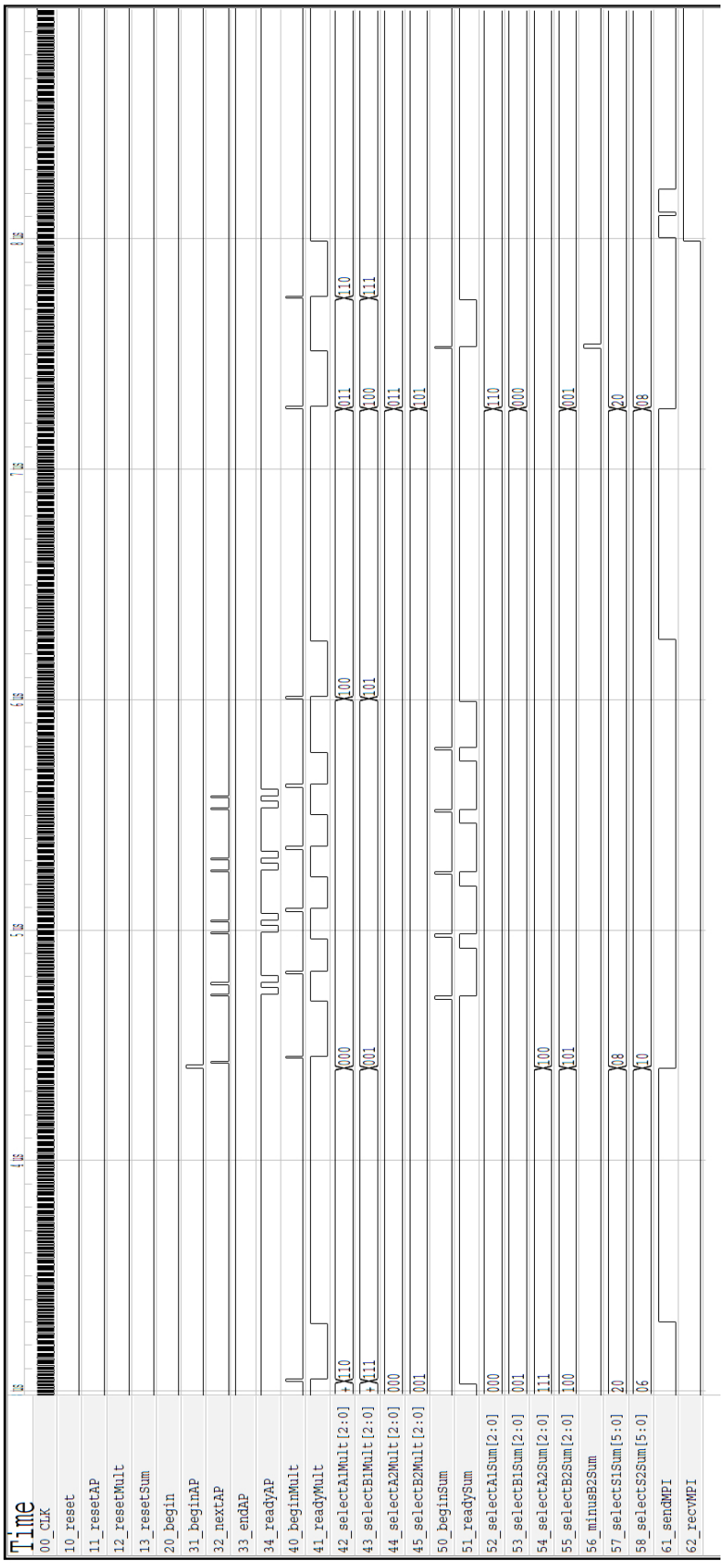


FIGURE 4.7: Conjugte gradient algorithm logic trace.

4.3 Werner

To investigate the convergence properties of potential solver algorithms for the LQCD matrices, a software framework was created into which matrix elements could be passed or matrix element generators could be invoked. For LQCD it is standard practice to generate the matrix elements as required rather than store precalculated values, hence callback functionality is often employed. Since LQCD matrices are generated from gauge configurations it is still a requirement that the gauge matrices are stored and this is often one source of bottleneck in a cluster and coprocessor implementation. These gauge matrices need downloading to the coprocessor and then updating between computational nodes as the molecular dynamics program progresses.

Werner provides a matrix interface that can be implemented for either once off calculation of matrix elements then stored or as a callback to an on demand generator. The former approach is used in this thesis to facilitate comparison between algorithms without the additional overhead of regenerating the matrices on each iteration, which itself is a linear time complexity operation which would add to the solve time complexity. For modern PC clusters the storage available is normally sufficient with the storage being required for the matrix elements being about 1.5GB per million rows. An important aspect of the solver design is to have matrix element updates restricted to nearest neighbour updates so this large volume of data would not contribute to a increase in time complexity due to data bandwidth growth.

Werner consists of a solver class hierarchy with a user interface called *Operator*, shown in figure 4.8, which provides access to matrix solver functionality. Methods are provided to access a reference to the matrix structure A, the boundary condition vector $\vec{b}$, and the solution vector $\vec{x}$ for initialization and the final answer. The method *solve* is called to generate the solution vector $\vec{x}$ from the loaded values of A and $\vec{b}$.

```
class WERNER_API Operator : private handle_body< Operator >
{
  public:
    explicit Operator( Args& PARAMS );

  public:
    ~Operator();

  public:
    Operator& solve();
    blas::Vector& test( blas::Vector& y, const blas::Vector& X ) const;
    const double residual() const;

  public:
    multigrid::Matrix& A();
    blas::Vector& x();
    blas::Vector& b();
};
```

FIGURE 4.8: Werner solver interface.

```
int main( int argc, char * argv[] )
{
  try
  {
    djdb::werner::Args args;
    djdb::werner::Operator solver( args.load( argc, argv ) );

    const int RANK = buildA( solver.A(), args );
    std::cout << "Dirac.rank() = " << RANK << std::endl;

    make_b( solver.b(), RANK );
    init_x( solver.x(), RANK );

    if ( args.arg3() > 0 )
    {
      solver.solve();
    }
    test_x( solver );

  }
  catch ( const std::exception& EX )
  {
    std::cout << "\nException = " << EX.what();
  }

  std::cout << std::endl << std::endl;

  return 0;
}
```

FIGURE 4.9: Main function for Werner solver testing.

Figure 4.9 shows the C++ program entry point used to collect the test data presented in chapter 2 and chapter 5. The function *buildA* generates the Wilson-Dirac matrix and places the elements into the storage provided by the call to *solver.A*(). It is here that the matrix storage can be specified with alternatives being a callback mechanism to a just-in-time matrix element generator for FPGA's or GPU's that cannot store gigabytes of matrix data, or for PC implementations a buffered retrieval from hard-drive may be a possibility to allow the CPU to focus on matrix solving rather than element generation. The program used here generates the matrix elements a single time then stores them in Compressed Row Storage format, but other formats are still being researched [51].

The class *Args* provides the mechanism to pass command line arguments into both *Dirac* for matrix generation, namely lattice width N and fermion mass m as arguments $arg1$ and $arg2$ respectfully, and the solver options for solver selection with parameters for its configuration. The *www.boost.org program_options* library is used with figure 4.10 showing the available options being specified in the code. Alternatively with *Args* being a class, the options can also be set or retrieved within the code: for example if one wanted to write a program to automatically test the same matrix against several different solvers or parameter configurations.

```
boost::program_options::options_description desc( "Options" );

desc.add_options()
  ( "help",        "produce help message" )
  ( "display",     "display parameters" )
  ( "list",        "display available solvers" )
  ( "test",        boost::program_options::value<std::string>(), "unit-tests [test_all, test_namespace, test_class, list]" )
  ( "solver",      boost::program_options::value<std::string>(), "solver name [CG, BiCG, etc]" )
  ( "precon",      boost::program_options::value<std::string>(), "preconditioning type [simple, multigrid]" )
  ( "smoother",    boost::program_options::value<std::string>(), "preconditioning smoother [jacobi, gauss, SSOR]" )
  ( "logtype",     boost::program_options::value<std::string>(), "log type [off, cout, filename, +filename]" )
  ( "datafile",    boost::program_options::value<std::string>(), "matrix data file [binary: Row(int)Col(int)Value(double)]" )
  ( "uncertainty", boost::program_options::value<double>(),      "residual tolerance" )
  ( "omega",       boost::program_options::value<double>(),      "relaxation parameter" )
  ( "gamma",       boost::program_options::value<double>(),      "preconditioning convergence boost" )
  ( "depth",       boost::program_options::value<int>(),         "multigrid smoother repeats" )
  ( "threads",     boost::program_options::value<int>(),         "solver threads for multithreaded solvers (Xmt)" )
  ( "mingrid",     boost::program_options::value<int>(),         "minimum grid size" )
  ( "loops",       boost::program_options::value<int>(),         "maximum iterations" )
  ( "restarts",    boost::program_options::value<int>(),         "gram-schmidt depth" )
  ( "levels",      boost::program_options::value<int>(),         "multigrid depth" )
  ( "arg1",        boost::program_options::value<int>(),         "arg1 for main" )
  ( "arg2",        boost::program_options::value<int>(),         "arg2 for main" )
  ( "arg3",        boost::program_options::value<int>(),         "arg3 for main" )
;

boost::program_options::variables_map vm;
boost::program_options::store( parse_command_line(argc-1, &argv[ 1 ], desc), vm );
boost::program_options::notify( vm );

if ( vm.count( "help" ) )
{
    std::cout << desc << std::endl;
    ::exit( 0 );
}
```

FIGURE 4.10: Command line options for Werner.

The main options are *solver*, *precon* and *smoother* which specify the configure the matrix solver method. The solver can be a stationary method Jacobi, Gauss-Seidel, SOR or SSOR; various the Krylov methods such as CG, BiCGStab, GCR, restarted GCRL, or higher polynomial BiCGStabL; or the solver could be Multigrid. Further variants are easy to add as shown later in figure 4.12.

The Multigrid has several options for its configuration including *smoother* to specify OR for relaxed Jacobi, SOR or SSOR as the V-cycle approximate solver. The number of smoother cycles to apply can affect the speed of the multigrid convergence and this can be controlled with the *depth* parameter. The multigrid configuration itself can be controlled by specifying the V-cycle grid depth with *levels* or by specifying a minimum size to the bottom grid with *mingrid*. The default behaviour for the algebraic multigrid is to construct the lower level from the average values of the current level based on the sparsity pattern down to a matrix size of 1. For LQCD these small matrices at the bottom of the V cycle generally add nothing to the overall convergence and one can use *mingrid* to avoid construction of the smallest grids.

Multigrid is a valid argument for both the *solver* option and as an argument for the *precon* option for use with preconditioned PCG or PCGNR. It is also possible to specify the Krylov methods as a "smoother" for the multigrid sub-grids in order to investigate K-cycle multigrid as an alternative to the standard stationary methods of V, W or Full cycle multigrid [52].

Werner facilitates rapid development and a flexible approach to investigating algorithms and parallel cluster environments. The key to this is the object orientated hierarchy of solver classes implementing the *Inverter* class interface via public inheritance [53]. Figure 4.11 shows the C++ class interface to the generic solver.

There are two public functions. An initialisation method *init* which passes in a reference (pointer) to the matrix and a reference to the *Args* so that the solver can access the parameter options. The second method takes the vectors $\vec{b}$ and $\vec{x}$ to then start the solving process. The class is using the Non-Virtual Interface idiom variant of the Template design pattern [54] to help encapsulate the derived implementation classes in a well defined manner.

The abstract implementations of *init* and *solve*, *initImpl* and *solveImpl*, are declared as pure virtual and private to tell the compiler to reserve virtual table entries for these classes, but require these methods to be provided by derived classes that inherit from the *Inverter* interface. The flexibility is provided when a pointer to the interface is used to create programs with a compiler but the specific solve implementation, may be conjugate gradient, Jacobi or anything else, is not known until requested by the user at runtime. The actual solve algorithm to use is requested using an option value string such as "CG" and using the *SmartPointerRegistry* to create the requested object based on a map between strings and factory functions that can create the algorithm object.

```
class Inverter
{
  public:
    typedef factory::SmartPointerRegistry< Inverter > factory;
    typedef factory::smart_ptr smart_ptr;

  public:
    Inverter& init( const data::CRSMatrix& A, const werner::Args& PARAMS );
    int solve( blas::Vector& x, const blas::Vector& b );

  private:
    virtual void initImpl( const data::CRSMatrix&, const werner::Args& ) = 0;
    virtual int solveImpl( blas::Vector& x, const blas::Vector& b ) = 0;

  protected:
    Inverter();

  public:
    virtual ~Inverter();
};
```

FIGURE 4.11: Werner solver interface.

Figure 4.12 shows how a solver algorithm is registered into the *Inverter* class hierarchy. The two virtual methods are coded here along with any variables needed for the implementation of those methods. The class is self contained inside the source file for this code and the contents of the implementation class are also entirely private in order to maximise and enforce good practice of encapsulation. It can be seen in the *init* implementation how the *Args* class is used to provide runtime configuration with requests about which preconditioner *precon* to use and which *logtype* to use. The preconditioner itself is accessed via the same *Inverter* interface and so one can see how the *Args* string is used by the $Inverter::factory$ to create the runtime requested solver when initialising the preconditioner $M\_$.

The $CRSMatrix$ is passed in by reference as an argument and this data location reference is used to form a matrix operator object of type $MatrixOp$. The matrix operator is used to form the $A\vec{x}$ matrix-vector product and here would be the location to configure alternative variants such as the case when the matrix is not stored but regenerated on each iteration on demand. The $Inverter::factory$ pattern also provides additional benefits beyond convenient flexible object creation and the very tight modular encapsulation; resulting from the class declaration being inside a source file rather than a header file.The factory's map can be used to query its contents and so allow a listing of available solvers to be provided at runtime to the user: this functionality is used for the command line option "list".

```
class PCG : public Inverter
{
  private:
    virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
    virtual int solveImpl( blas::Vector& x, const blas::Vector& b );

  private:
    boost::shared_ptr< operators::MatrixOp > A_;
    Iterate::factory::smart_ptr iterate_;
    Inverter::factory::smart_ptr M_;
};


Inverter::factory& factory = Inverter::factory::instance().add< PCG >( "PCG" );


void PCG::initImpl( const data::CRSMatrix& A, const werner::Args& ARGS )
{
  this->M_ = Inverter::factory::instance().create( ARGS.precon() );
  this->M_->init( A, ARGS );

  std::string s = std::string( Iterate::CLASS_NAME ).append( ARGS.logtype() );
  this->iterate_ = Iterate::factory::instance().create( s );
  this->iterate_->init( ARGS );

  this->A_.reset( new operators::MatrixOp( A ) );
}
```

FIGURE 4.12: Werner conjugte gradient solver initialisation.

The list can also used to unit test all available solvers by creating a suite of tests and then executing those tests for-all available solvers in the factory list; a feature made available through the command line option “test” and useful for installation testing.

Figure 4.13 shows the preconditioned conjugate gradient algorithm implementation itself. The algorithm is constructed from a sequence of calls to underlying BLAS [55] operations and matrix-vector multiplies executed by the matrix operator. GPU or FPGA accelerators would be interfaced through alternative *Inverter* implementations with the code shown here replaced by calls to BLAS primitives located within the accelerator hardware. For example Nvidia’s CUDA provides a BLAS library that would execute on data loaded by the matrix operator object. The SystemC conjugate gradient hardware would be called via a similar mechanism from this location.

```
int PCG::solveImpl( blas::Vector& x, const blas::Vector& b )
{
  double alpha = 1.0;
  double beta  = 0.0;

  blas::Vector z( x.size() );
  blas::Vector mr( b.size() );
  blas::Vector r = b;
  blas::Vector p = x;

  this->A_->multiplyA( z, x );
  blas::daxpy( r, -alpha, z );
  this->iterate_->test( blas::ddot( r, r ) );

  this->M_->solve( mr, r );
  blas::daxpy( blas::dscal( p, beta ), 1, mr );

  double rho1 = blas::ddot( mr, r );
  double rho0 = rho1;

  while ( this->iterate_->next() )
  {
    this->A_->multiplyA( z, p );
    alpha = rho0 / blas::ddot( z, p );

    blas::daxpy( x,  alpha, p );
    blas::daxpy( r, -alpha, z );
    this->iterate_->test( blas::ddot( r, r ) );

    this->M_->solve( mr, r );
    rho1 = blas::ddot( mr, r );

    beta = rho1 / rho0;
    blas::daxpy( blas::dscal( p, beta ), 1, mr );

    rho0 = rho1;
  }

  return this->iterate_->index();
}
```

FIGURE 4.13: Werner conjugte gradient solver implementation.

4.4 Multiprocessing

Once the algorithm has been tested on a single processor the task is then to distribute the algorithm over a cluster of processing nodes. Amdahl's Law indicates that the relation between the overall performance and an algorithm's parallel performance is limited by the section of code that is inherently sequential in nature [56]. For the Krylov subspace solvers the vector element updates can be done with complete parallelism since their calculations depends only on nearest neighbour values, so the update complexity is constant irrespective of the size of the cluster or the size of the matrix distributed over that cluster. Each time a vector element changes due to a matrix-vector multiply, that element update only needs sending to a fixed number of nearest neighbours. The parallelization problem inherent in Krylov subspace algorithms comes primarily from the global nature of the scalar product with its $O(logN)$ time complexity within a tree network setup.

Investigating these communication issues can be done using the MPI tool Jumpshot and in figure 4.14 one can see the relative amount of time spent between data communication and processing time. The $MPI\_Allreduce$ (lower item in the legend) shows the time spent collecting together the dot product sum, the $MPI\_Allgather$ shows the time taken to update element values across the network, whilst the time in between uncoloured is the time spent by the processor in doing the numerical work.

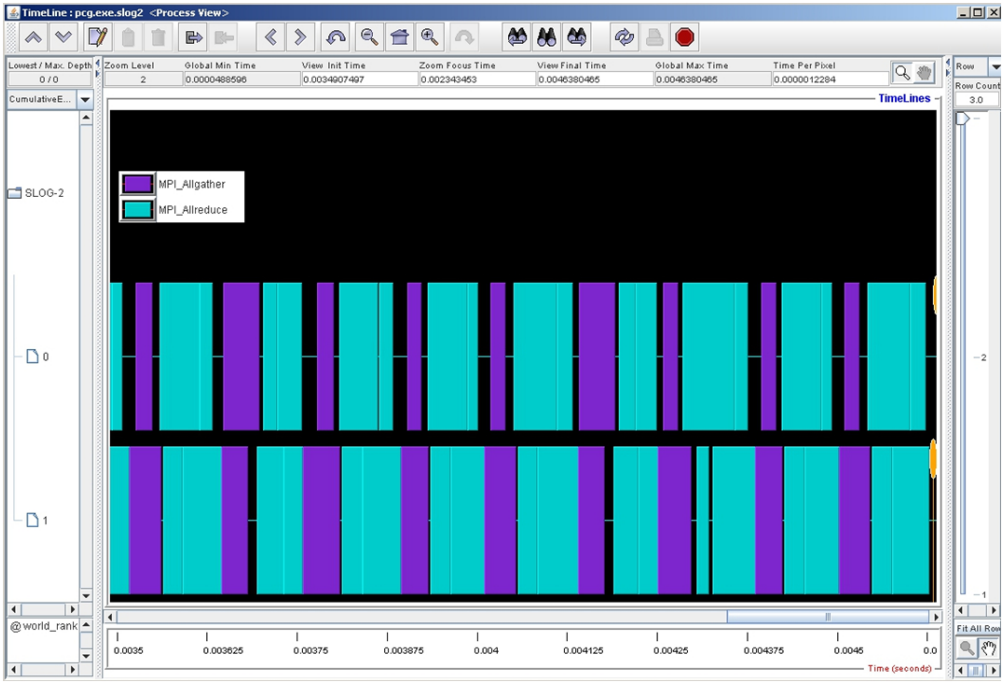


FIGURE 4.14: Conjugate Gradient Jumpshot.

This Jumpshot was collected for a conjugate gradient execution over two processing nodes and shows most of the time spent is in data exchange. Rewriting the conjugate gradient algorithm to now exchange vector elements with nearest neighbours only results in a Jumpshot profile as shown in figure 4.15. With individual elements sent to specific nearest neighbours the communication overhead as a percentage of total time has greatly decreased.

Werner contains both multi-threaded and MPI multi-processor variants of the Krylov methods. A cluster manager class automatically partitions the matrix into rows evenly spread across the designated number of threads or nodes, and allocates the corresponding vector elements so that the matrix-vector product for that subset of indices is calculated locally. This is implemented through matrix and vector classes such as $MatrixOpMT$, $VectorMT$ and $DAXPYmt$, which can be used by the solver algorithms in place of the original single process classes without further modification of the solver algorithm code.

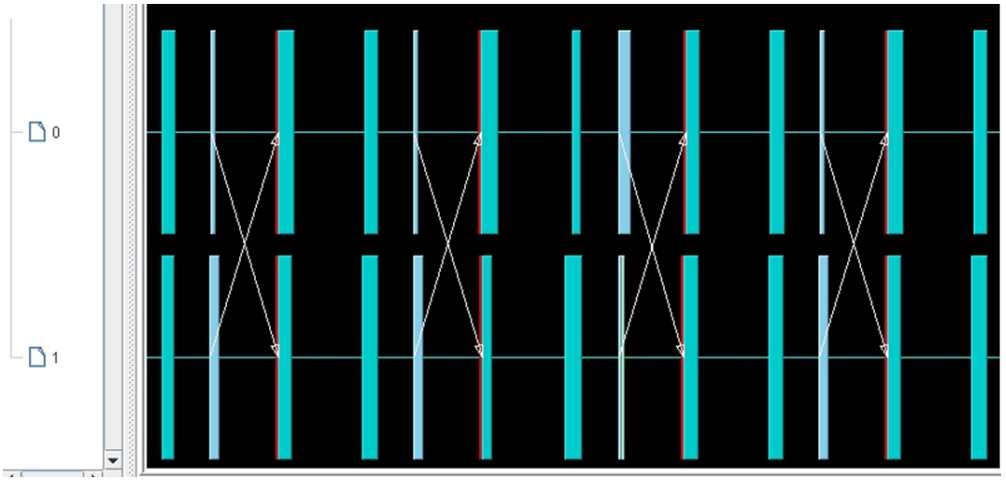


FIGURE 4.15: Improved Conjugate Gradient Jumpshot.

4.5 QUDA

Graphical Processing Units are designed for highly concurrent fast floating point operations for image rendering. Nvidia Corporation provides a C based library interface known as CUDA for the programming of their GPU devices [57]. Image processing is heavily based on matrix calculation and it was realised that LQCD could benefit immensely from the high performance floating point performance of these devices [58]. OpenCL is an alternative programming language available as an open standard for the low-level programming of heterogeneous computing clusters and can be used to program GPUs as well [59].

QUDA is a QCD library built on top of Nvidia's CUDA and has been used to successfully provide GPU coprocessor support for LQCD calculations [60] and is now supported for several of the major LQCD production systems [61]. Scaling to parallel systems with hundreds of GPUs has now been reported for key LQCD bottlenecks of gauge field configuration and Wilson-Dirac matrix solvers [62].

The test applications available for download were complied and executed on a desktop PC containing an Nvidia GPU card and the results are shown in table 4.1. The mass value of 100 (as per the Werner library command line parameter) was used: corresponding to a hopping parameter kappa value of 0.16 and anisotropy set at the isotropic value of 1.0 as set in the main function of the QUDA test application. The uncertainty value at which to stop the solver was set at a value equivalent to Werner's uncertainty equal to 1e-8.

The performance of this test code against mass is shown in table 4.2 for the largest matrix size of just under 8 million. The solver algorithm used was CGNR and for a mass value of 120 or more the solver application caused a timeout.

TABLE 4.1: QUDA test code performance against lattice width.

Lattice Width	Matrix Rank	Iterations	Time / s
8	98,304	155	0.5
12	497,664	177	0.9
16	1,572,864	248	2.4
20	3,840,000	263	5.4
24	7,962,624	317	13.6

TABLE 4.2: QUDA test code performance against mass.

kappa	mass	Time / s
0.10	-100	3.7
0.12	-17	4.4
0.14	43	6.0
0.16	88	13.6
0.18	120	diverges

The raw speed of the GPUs for floating point scientific work is very impressive, and one can compare the time taken to solve the lattice width case of N=12 for easy to solve heavy masses matrix. In table 4.1 the GPU code took 177 iterations in only 0.9s compared to the CPU code with Werner taking 131 iterations in 737s; as given in table 2.3 of chapter 2. The CPU code was with a single thread of execution whereas the GPU has almost 200 parallel threads over which the code is executed. For critical mass values the algorithm's time to solve suffers from the same critical slowing down as with the CPU, but critically the GPU slows down 200 times faster.

4.6 Reconfiguration

The original research of this thesis was aimed at constructing a massively parallel farm of low cost FPGAs. The Xilinx Spartans could be purchased in bulk from as little as US$20 each, potentially providing floating point multipliers for a few dollars each. The matrix elements could be generated inside the FPGA in parallel with the matrix solver code, using a high concurrency design to vector pipeline the matrix data into the solver. With the matrix data inside the FPGA this could reduce large scale data movement to help address bandwidth issues commonly found with hardware accelerators accessing and returning data from the main store RAM.

Aside from moving data between main store and coprocessor, the other critical issue are the inherent communications bottlenecks in the Krylov subspace methods themselves. Sparse matrices have a distinct advantage when their elements are generated from discretized partial differential equations, such as the Dirac equation, due to the vector element updates being to near neighbour elements only: as matrix size increases this scales perfectly without adding to the time taken, as long as the network pattern mirrors the matrix pattern. Small FPGAs with localised data would again be ideal here since they can be mounted on a printed circuit board with their layout mirroring the

sparse matrix rather than the monolithic layout of a large block of RAM with general purpose data buses to provide the random access.

Given good nearest neighbour communication between FPGA matrix elements, the largest Krylov bottleneck would then be the global scalar products required to update search directions or perform orthogonalisations, and these are known to scale poorly on distributed clusters. Bespoke design facilitated by small FPGAs would again be useful in its ability to be customised; for example by proving a tree network structure for the optimal $O(logN)$ performance, and maybe a dedicated high speed bus just for the scalar product data. Having separate data buses for the two types of communication would allow optimal performance for both rather than relying on a general purpose bus.

With matrix element data localised inside FPGA cells this allows bringing into play the idea of minimising data movement by dynamically reconfiguring [63] the FPGA part way through the Hybrid Monte Carlo HMC algorithm. The core data to be stored inside the registers of the FPGA would be the gauge field values from which the Wilson-Dirac matrix elements are generated: over ten times smaller in bytes than holding the matrix elements directly. During the inverter step of the HMC algorithm the FPGA needs to be setup with the MACs configured for Krylov methods and Wilson-Dirac fermion matrix generation, but later the gauge fields are also used to calculate the boson component of the Hamiltonian's action. At this point the FPGA could be partially reconfigured for the boson action instead of the fermion inverter without the need for moving the gauge data. This technique could also be used when switching between an outer Krylov loop and an inner preconditioner so that data paths are optimised around the data registers in a process known as partial reconfiguration [64], thus saving both time for fewer data movements and space with logic cells reused between reconfigurations.

The problem has been that learning to program FPGAs is a very steep learning curve without a strong electronics background. The basic tools are geared towards logic analysis rather than higher level software programming languages. As FPGAs grow larger and larger this becomes a bigger problem since it becomes impractical to program at the logic level for millions of gates, and software tools are slowly becoming available, but can be very expensive. Embedded products such as uBlaze are extremely helpful and provide the opportunity to use a commodity style processor embedded inside the FPGA to act as a controller with bespoke units tailor made around them [65]. From a software programming perspective trying to program a large system with logic traces is not only comparable to writing a large application in assembly, but constructing the assembly language itself as you go.

The FPGA design for LQCD matrix solvers is a huge project and the cost effective Spartan FPGAs are two small to hold sufficient floating point multipliers with enough

```
#include <boost/numeric/ublas/matrix.hpp>
#include <boost/numeric/ublas/io.hpp>
#include <uow/nanostorm/corelib.hpp>

using uow::nanostorm::invert;
using boost::numeric::ublas::matrix;

int main ()
{ matrix< double > m( 4, 4 ) =
    -2, 1, 0, 0,
     1,-2, 1, 0,
     0, 1,-2, 1,
     0, 0, 1,-2;
  std::cout << invert( m ) << std::endl;
  return 0;
}
```

FIGURE 4.16: Primatives linked via C++ templates.

gate fabric around gauge field holding registers. GPUs have since entered the market space and their floating point capabilities far outstretch that available with FPGAs. To implement the intended design would require the more expensive Virtex FPGAs, at which point one might as well use off the shelf GPUs. Another important advantage with GPUs is the vendor supplied C-based programming interface CUDA, or OpenCL as a vendor neutral alternative, which then immediately provides a software development environment free of charge.

4.7 FPGA and Werner

FPGAs have been the subject of research for numerous reasons: the hardware logic can be tailored and optimised for specific applications; they offered the possibility of dynamically altering that tailored logic to new tailored logic during the execution of an application; they have the potential to create hardware with a software development model; physically they use less power because they can be tailored without superfluous logic. With energy costs increasing reducing power consumption has bigger cost saving potential that even the purchase cost per FLOP.

For this project though FPGAs did not progress to the actual implementation of algorithm stage. SystemC models where produced to implement the archetypal Krylov methods and the primitives required but the steep learning curve of design to implementation proved to high. The thesis moved onto the algorithm investigation presented in the following chapter without having the time to return to FPGA development.

The design would have the BLAS required for Krylov subspace algorithms encapsulated behind a C++ template, such as that provided by the *BOOST* library, allowing the user to transparently invoke the FPGA implementation via the inclusion of a software header library (called NanoStorm in an early prototype version). This is shown in figure 4.16.

Chapter 5

Results and Discussion

"We know what kind of a dance to do experimentally to measure this number very accurately, but we don't know what kind of dance to do on the computer to make this number come out, without putting it in secretly!"

Richard Feynman

With the emergence of GPUs this chapter now presents the results of an investigation, using the C++ Werner library, into constructing a highly scalable algorithm for Wilson-Dirac inversion. The algorithm should still scale perfectly for nearest neighbour vector element updates and have the potential to keep the scalar product restricted to complexity $O(logN)$.

GPUs prior to the Nvidia Tesla generation suffered from the drawback that they included little fault detection which made them potentially problematic for accurate scientific work; an inaccuracy in a few pixels that briefly flash on the screen was a good trade-off for the large amount of resources needed to detect faults. Using GPUs for a single precision preconditioner works particularly well since preconditioners are tolerant of inaccuracies by their very nature of being approximate inverses.

The Tesla generation of GPUs now has fault detection and is being aimed at the HPC market as a General Performance GPU (GPGPU), with a single accelerator card capable of providing over 1 Tflop double precision with 6Gb on board memory and almost 4 Tflop peak single precision performance. With this performance available a highly scalable algorithm would enable larger lattices to be investigated with the matrix size reaching billions of rows.

5.1 Multigrid and Dirac Linearity

The reference implementation of the Wilson-Dirac equation was taken from the QUDA software library [18]. Initial investigations indicated that the performance of the QUDA-Dirac implementation was dominating the time complexity performance of the test runs; even at the easier to solve high mass hopping parameter values. The basic test run of QUDA on a GPU used a mass value of $m = 40$ which leads to diagonally dominant matrices and can be solved with linear time complexity using multigrid preconditioned CGNR.

The results of CGNR with multigrid preconditioning are shown in table 5.1. The initial results using simple Algebraic Multigrid [66] as a preconditioner on CGNR were encouraging on the small scale tests initially undertaken. The lattice widths were only modest over the range of $N = 4$ up to $N = 12$ resulting in the largest matrix having a rank just under half a million. The matrix sizes were restricted due to the requirement of multigrid needing to generate transformation matrices to prolongate and restriction between the multigrid matrix levels [67].

The extra memory requirements for multigrid have proved inhibitive with regards to pursuing multigrid further, and the AMG code produced for this thesis has required the Dirac matrix to be explicitly formed, whereas in production code the Dirac matrix is rarely formed, but rather the elements are generated when needed. The gauge field links are required to form the matrix elements, but take up about a hundred times less space than the matrix elements would to store. The matrix generated from a single lattice site has 24 rows to cover the complex color and spinor values of the wavefunction leading to 24 doubles times 97 elements per row times the bytes used to store 1 double and two coordinate integers: 37248 bytes per lattice site. In contrast the gauge field values would

Table 5.1: Matrix reference generation performance.

Lattice Width	Matrix Rank	Iterations	Time / s
4	6,144	23	7.5
5	15,000	28	20.8
6	31,104	32	45.6
7	57,624	36	86.6
8	98,304	39	466.6
9	157,464	37	255.6
10	240,000	38	382.1
12	497,664	36	768.8

take up only 384 bytes per site, and a unitary matrix relationship between elements can be used to generate three elements out of nine if six are given [68].

Adaptive multigrid is being investigated for the use in LQCD [25] and has shown some promise in handling the key issue of critical slowing down for the light mass problem. Classical multigrid suffers from the low modes with the almost zero eigenvalues not being geometrically smooth causing the slow to converge errors not being locally constant. Research into tackling this problem with the multigrid approach is being undertaken, leading to the idea of grouping the eigenvectors of the lower end of the spectrum together in a similar manner to Lüscher 's deflation approach [69]. This attempts to remove the troublesome eigenvalues into a separate subspace and then solve the remaining matrix without the modes that are causing the critical slowing down.

For the purposes of this investigation the simpler Algebraic Multigrid was used whereby the matrix elements within a given row are averaged across as the basis for generating the restriction operators to generate the smaller sub-grid. As with the overall time complexity, the memory requirements are also a linear $O(N)$ since to the total resource usage is a geometric sum whose total is proportional to the original problem size N. The algebraic multigrid breaks down when the critical mass values are approached, not least because the smoothers used become divergent when diagonal dominance is completely lost. The time complexity linearity for the small Wilson-Dirac matrices with heavy fermions can be seen in the graph in figure 5.1.

With heavy fermions on small lattices being solved in linear time it was apparent that the reference implementation was dominating the total application time as the matrices

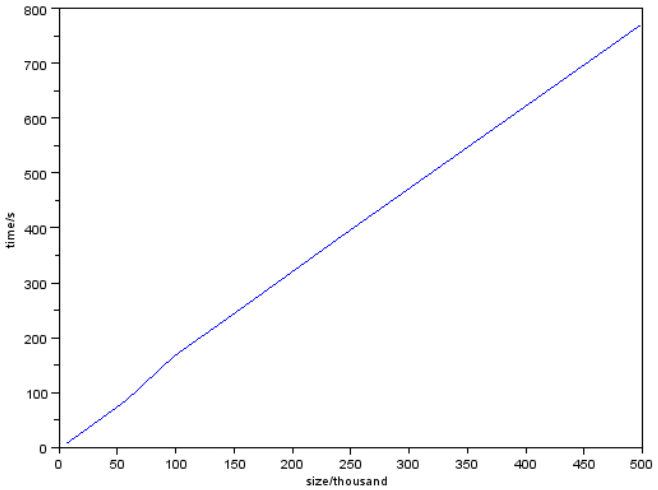


FIGURE 5.1: AMG solver time against matrix size.

TABLE 5.2: Matrix reference generation performance.

Lattice Width	Matrix Size	Setup / s
2	384	0.1
3	1944	1.1
4	6144	4.9
5	15000	20.6
6	31104	73.0
7	57624	216.3
8	98304	574.9

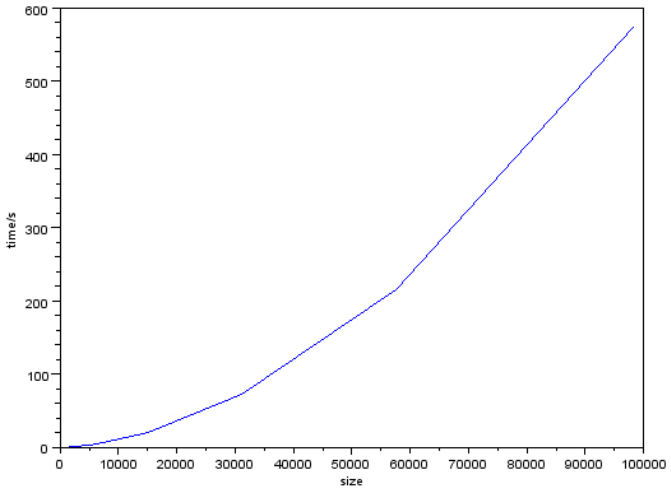


FIGURE 5.2: Matrix reference generation time against matrix size.

increased in size, and thus exhibiting over-linear time complexity solely in the matrix element generation. In the test QUDA code, from whence the matrix code came, the raw speed of the GPUs would appear to mask this issue. The table 5.2 shows the data with the performance of the reference implementation plotted as time against matrix rank in figure 5.2.

The origin of the non-linearity was traced to the fact that for each lattice site the matrix elements generated from that lattice site reflect the probability that a transition will be made to any other lattice site, hence a natural $O(N^2)$ complexity. In practice with the Wilson-Dirac though the probability hopping is between nearest neighbours, which with boundary wrap around, is always eight nearest neighbours. Thus the time complexity should go as $O(N) \times 8$ in being optimal. The GPU reference implementation used a nested for-loop to visit all sites and to calculate the hopping overlap to all other sites, which was the origin of the quadratic behaviour.

To enforce the optimal linear complexity, a mapping function was created so that any given lattice site could call the function to enumerate its nearest neighbours based on an index lookup value, and thus only visit those eight sites which are its nearest neighbours. This nearest neighbour lookup is required in several places, and all double nested loops were replaced with a single loop and the nearest neighbour lookup function. The results table for the linear matrix element generator is given in table 5.3 and the graph to show linearity is in figure 5.3.

Keeping interactions to nearest neighbours is very important to remove a linear time complexity degree of freedom in several aspects of the overall application, not least of which is vector updates between iterations. When distributing the application over

TABLE 5.3: Improved matrix generation performance.

Lattice Width	Matrix Size	Setup / s	Memory / MB
8	98304	2.7	197
9	157464	3.5	246
10	240000	5.1	376
11	351384	8.1	557
12	497664	10.7	786
13	685464	15.2	1098
14	921984	20.0	1442
15	1215000	26.9	1933
16	1572864	36.7	2441
17	2004504	44.4	3195
18	2519424	55.8	4063
19	3127704	70.6	4981
20	3840000	100	6193

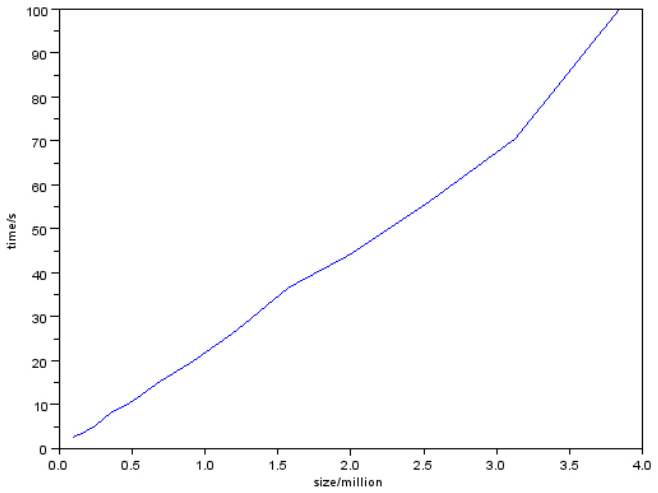


FIGURE 5.3: Improved generation time against matrix size.

an MPI or co-processor network the nearest neighbour aspect is crucial to keeping IO bandwidth at constant-time complexity by respecting the fact that when performing the Ax matrix product only nearest neighbours need to be informed of iteration vector changes, and that the new vector components do not need to be broadcast to all sites over all processor nodes.

Only nearest neighbour sites, and the nodes on which they are located, require IO exchange, and this keeps the IO bandwidth at a constant level independent of the lattice width and final matrix size. The only IO growth factors are the scalar products in the Krylov algorithms and the MPI Reduce() calls that are required to evaluate them. This is a well known factor in the distribution of Krylov solvers over distributed processors [70]. With multigrid possibly providing an upper limit to how many outer Krylov iterations are required independent of matrix size, this could constrain the time dependency of the MPI Reduce() calls by restricting them to a set number.

One significant issue with multigrid is the overhead required in setting up the multi-level grid system with all the restriction and prolongation matrices that require evaluating and storing, but this overhead can be amortized to some extent by the fact that multi-mass LQCD algorithms can reuse the same matrix several times [71].

Another significant issue is that whilst linearity seems clear for well conditioned matrices [4] since the number of outer Krylov iterations appears bounded with the linear multigrid preconditioner, the low mass fermions remove the ability for simple stationary solvers to be applied in the standard multigrid manner resulting in more complicated schemes being researched to again make multigrid feasible [25].

5.2 Optimal SSOR Parameter

Jacobi and Symmetric Successive Over-Relaxation can be effective and relatively simple preconditioners for many matrix systems, but calculating the correct relaxation parameter value requires detailed knowledge of the eigenvalue spectrum a priori [72]. A binary search over the allowable range $0.0 \leq \omega \leq 2.0$ gave optimum values for the Wilson-Dirac matrices and it was observed that this optimum value remained about the same for varying mass parameter values.

For critical mass values Figure 2.6 in Chapter 2 shows that stationary methods diverge from the very start for the worst case critical values and are of little use by themselves. It will be shown here though that as part of a hybrid preconditioning system, a few, or even just one application of a stationary method iteration can be beneficial. In that case it is necessary to chose a relaxation value and the same choice as that optimal for heavy mass fermions still appears to be the correct choice.

Table 5.4 shows the data collected to compare the performance of Jacobi, SOR and SSOR, on heavy fermions with a mass of $m = 0.4$. At this mass value these stationary methods all work well and so provide a good test environment to choose a good omega (relaxation parameter) value. As expected from the literature SSOR performs the best, Jacobi takes the most number of iterations to converge, and SOR is better than Jacobi but not as good as SSOR. Jacobi worked best with the relaxation parameter being less than one at a value of two thirds to give an under-relaxed variant. SOR did not benefit from relaxation with omega being best at a value of 1.0; corresponding to the underlying Gauss-Seidel algorithm with no relaxation at all.

TABLE 5.4: Relaxation parameter performance.

Lattice	Matrix	Jacobi	omega=0.67	SOR	omega=1.00	SSOR	omega=1.26
Width	Rank	Iterations	Time / s	Iterations	Time / s	Iterations	Time / s
8	98,304	10	16.2	4	10.0	3	7.5
9	157,464	11	27.7	5	15.5	3	11.9
10	240,000	11	42.3	5	24.2	3	18.1
11	351,384	11	62.9	5	35.4	3	26.8
12	497,664	11	87.6	5	49.2	3	37.9
13	685,464	12	130	5	67.6	3	51.4
14	921,984	12	178	5	92.9	3	70.8
15	1,215,000	11	221	5	123	3	92.9
16	1,572,864	11	286	5	161	3	122
17	2,004,504	11	367	5	206	3	158
18	2,519,424	12	488	5	232	3	195
19	3,127,704	12	613	5	322	3	246
20	3,840,000	12	756	5	403	3	314

SSOR used a quarter of the number of iterations as compared to Jacobi, but the more complicated algorithm was only twice as quick with respect to actual time taken. The optimum value of omega for SSOR was over one, to give over-relaxation at a value of around $\omega = 1.2$. Table 5.5 shows the performance of SSOR when used as a preconditioner on the GCR algorithm, again in the straight forward configuration of heavy fermions far from the critical mass: $m = 40,\ N = 10,\ depth = 5$. Depth refers to the number of repeats of SSOR is used as the preconditioner prior to returning to the main GCR outer iterative loop. As can be seen from the table and the graphical plot 5.4, a relaxation value of around 1.2 works best. This was found to be the case independent for lighter masses and larger matrices as well.

TABLE 5.5: GCR performance and SSOR relaxation parameter.

--omega	Iterations	Time / s
0.1	46	109
0.2	33	79.1
0.3	26	63.3
0.4	21	52.1
0.5	17	42.8
0.6	15	38.7
0.7	12	32.5
0.8	11	30.5
0.9	9	26.2
1.0	8	23.7
1.1	7	21.3
1.2	6	19.9
1.3	6	23.3
1.4	8	24.2
1.5	10	33.8
1.6	13	35.1
1.7	18	55.4
1.8	26	63.3
1.9	40	122
2.0	69	162

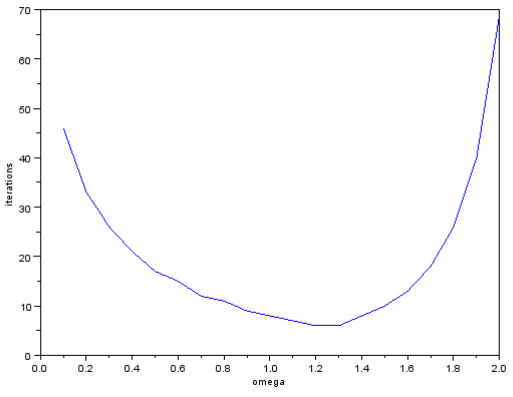


FIGURE 5.4: GCR performance against SSOR relaxation parameter.

5.3 SSOR Depth Parameter

For diagonally dominant matrices corresponding to heavy fermion masses the stationary methods can be used to solve the Wilson-Dirac matrices, so using them to provide preconditioning is highly effective when the hopping parameters are far from critical. Table 5.6 and the plot of solver time against SSOR preconditioning depth (repeats of SSOR iterative steps) in figure 5.5 show this clearly. The breakdown occurs when fermion mass approaches lighter critical values and the application of stationary iterations at any point in the solver algorithm causes the method to diverge as either the primary solver or when used as a preconditioner. This makes basic multigrid techniques unusable without first removing the smallest eigenvalue modes, such as using a deflation scheme [73].

TABLE 5.6: GCR performance and SSOR depth.

--depth	Iterations	Time / s
0	69	49.0
1	15	22.8
2	15	22.7
3	9	21.4
4	9	23.2
5	7	23.6
6	7	23.4
7	5	21.6
8	5	22.0
9	4	21.7
10	4	21.8

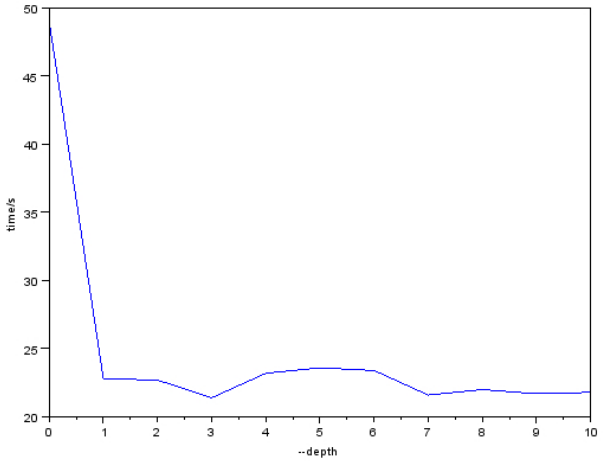


FIGURE 5.5: GCR time against SSOR depth parameter.

5.4 Preconditioned GCR

The General Conjugate Residual algorithm compares favourably to BiCGStab for the easier to solve heavy fermions when one compares the data between the table 2.5 and table 2.7 in chapter two. GCR takes almost twice as many iterations as BiCGStab but only 37% longer in terms of time taken for the largest matrix of rank four million, despite all the orthogonalisations that are required as part of GCR.

The large number of orthogonalisations required could be a major bottleneck for any distributed implementation due to the nature of the scalar products (hence MPI Reductions) involved with any such orthogonalization procedure. GCR does have the advantage though in that the for-loop that performs the scalar products has no dependencies between all the loops, hence can easily be loop-unrolled, and done entirely in parallel. The scalar product components that then need to be broadcast to all the distributed nodes can be grouped together into a single transmission in order to reduce this potential scaling bottleneck.

As can be seen in figure 5.6 the calculation of the i^{th} beta depends on values determined prior to entering the loop and not altered again until the next GCR k^{th} iteration. The update of $p(k+1)$ and $q(k+1)$ can be factored out to a second for-loop to be called once the beta updates have been completed. In this way the expensive beta(i) updates can be grouped into a single aggregate MPI Reduce call. The latter two vector updates, requiring the beta(i) evaluation, will have to wait until the MPI Reduce call returns, but the vector updates involved purely nearest neighbour updates and contribute a fixed communication cost independent of matrix size.

Without preconditioning the $N = 20$ matrix with almost four million rows took 77 iterations and about 15 minutes to reach the required tolerance for even the easier heavy mass fermions. For the more critical lighter fermions the required tolerance took hours to reach with thousands of iterations. The original GCR algorithm requires the complete history of update vectors against which to orthogonalize as shown in figure 5.6 which can cause memory usage to rapidly escalate as each vector has the dimension of the rank of the matrix being solved. These vectors are not sparse so require full storage although

```
for ( int i = 0; i <= k; ++i )
{
  ks->beta( i ) = -1.0 * blas::ddot( ks->q( k + 1 ), ks->q( i ) ) / ks->sigma( i );
  blas::daxpy( ks->p( k + 1 ), ks->beta( i ), ks->p( i ) );
  blas::daxpy( ks->q( k + 1 ), ks->beta( i ), ks->q( i ) );
}
```

FIGURE 5.6: Loop unrolling beta calculation in GCR.

they can be distributed over the cluster, using nearest neighbour communication only when required.

Reducing the number of GCR iterations required will aid this situation, and this is exactly what preconditioning achieves as can be seen in table 5.7; where the N=20 matrix now reaches tolerance in almost a quarter of the iterations taken up without preconditioning. A simple SSOR with only a single up-down cycle and 1.1 as the relaxation parameter was used. Whilst the time taken reduces by 41% the number of iterations reduced by a useful 74% which has a significant impact on the memory required to store the history of update vectors for each iteration.

Figure 5.8 shows a nice smooth linear convergence for GCR with the simple 1-cycle SSOR preconditioner with the matrix size N=20 and heavy fermion m=40. The data presented in this results chapter will be with 1-cycle SSOR and relaxation parameter $\omega = 1.1$ unless otherwise stated, primarily to ensure a fair-test comparison between results. The optimal relaxation parameter appears to be in the range $1.2 \leq \omega \leq 1.3$ as demonstrated in figure 5.4 from section 5.2 but a lot of data had already been collected at $\omega = 1.1$, so all data for purposes of presentation here, have also been collected at $\omega = 1.1$. The effect on the results maybe a few percent, which is significant for full-scale runs, but the focus here is on comparison between solver configurations and 1.1 is close to enough not to affect comparison.

TABLE 5.7: PGCR performance and matrix size.

Lattice Width	Matrix Rank	Iterations	Time / s
8	98,304	17	10.6
9	157,464	18	17.1
10	240,000	19	26.2
11	351,384	19	44.0
12	497,664	19	58.7
13	685,464	19	82.5
14	921,984	19	101
15	1,215,000	20	162
16	1,572,864	20	212
17	2,004,504	20	259
18	2,519,424	20	294
19	3,127,704	20	413
20	3,840,000	20	529

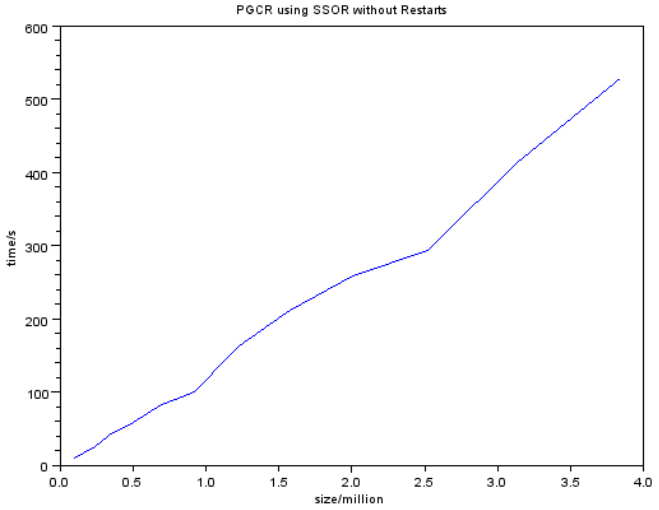


FIGURE 5.7: PGCR time taken against matrix size.

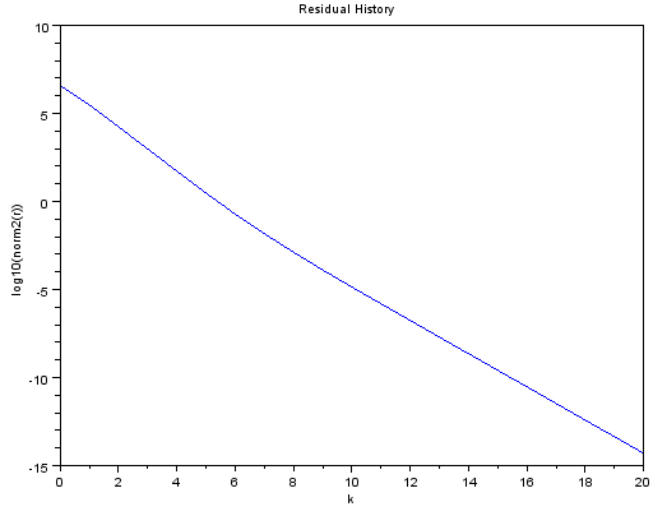


FIGURE 5.8: PGCR convergence N=20 and m=40.

5.4.1 Matrix Mass Behaviour

The repeat depth on the SSOR preconditioner can also affect the convergence rate, but again for a fair-test comparison, the value is kept at a single depth of SSOR preconditioner. For heavy fermions, the easier to solve matrices can be greatly accelerated by increasing any of the stationary method preconditioners, relaxed Jacobi, SOR or SSOR; as shown in figure 5.5 of section 5.3. For the critical mass matrices these preconditioners generally fail and increasing the depth of stationary method preconditioners causes the divergence discussed in relation to figure 2.5 of chapter 2. For the critical mass m=200 even a single application of relaxed-Jacobi diverges the solution as shown in figure 2.6.

SSOR is known to be a significantly better method than relaxed-Jacobi, with data in table 5.4 confirming this for the Wilson-Dirac matrices with heavy masses. Conversely for Wilson-Dirac matrices with lighter fermion masses, SSOR also diverges faster as well; thus implying SSOR would fare worse as a preconditioner in the critical cases. Table 5.8 shows this situation with convergence faster than Jacobi 2.2 for masses less than 100 in chapter two, but diverging faster for the lighter masses up to m=200.

TABLE 5.8: SSOR solver performance against mass.

mass	Iterations	Time / s
0	15	19.4
10	16	18.9
20	17	20.9
30	19	22.4
40	21	24.4
50	24	26.6
60	28	28.9
70	35	35.8
80	46	44.8
90	73	65.1
100	172	198
110	[10]	
120	[7]	
130	[6]	
140	[5]	
150	[4]	
160	[3]	
170	[2]	
180	[1]	
190	[1]	
200	[1]	

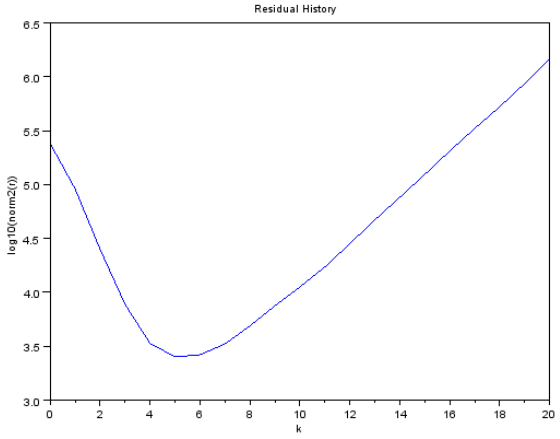


FIGURE 5.9: The effect of SSOR smoothing on medium mass fermion matrices.

In comparison to the Jacobi data in table 2.2 of chapter 2 where only m=200 diverges from the onset, now for SSOR even m=180 diverges from the very beginning. Figure 5.9 shows m=120 already diverging from the seventh iteration as opposed to Jacobi in figure 2.5 which only began to diverge at the 41^{st} iteration.

This would suggest SSOR may not be the best smoother to use as a preconditioner with Krylov methods for critical masses, but counter intuitively with respect to that evidence, SSOR is still the best smoother and the only one that works at all for the most critical matrices of the lightest mass fermions. A major caveat though, is that only a single application of the SSOR smoother is helpful for the critical mass matrices. Two or more applications of SSOR cycles completely destroys any convergence in the outer Krylov loops as well. SOR and Jacobi smoothers as preconditioners on the critical matrices fail to work for any depth when the mass is critical, and will only work for masses where some progress would be made as the primary solver. For example applying SSOR 7 times, or applying Jacobi 41 times, when m=120 is helpful; but not more iterations or at lighter mass values.

Table 5.9 shows the performance of preconditioned GCR with the preconditioner being a single application of SSOR with relaxation parameter $\omega = 1.1$. The number of Krylov loops are reduced by a factor of about five and time taken reduced by a factor of about ten for the lighter critical masses when compared to the data for unpreconditioned GCR in table 2.8 of chapter 2.

Even a single application of SSOR preconditioner has improved the performance of GCR by almost an order of magnitude, despite SSOR being of less than no use, when used as the primary solver by itself.

TABLE 5.9: PGCR performance and matrix mass.

mass	Iterations	Time / s
0	14	23.6
10	15	24.2
20	16	24.7
30	17	26.4
40	19	29.3
50	20	33.8
60	23	33.9
70	27	41.3
80	33	45.2
90	41	60.6
100	52	67.3
110	66	93.3
120	81	110
130	96	148
140	122	176
150	142	209
160	175	277
170	206	364
180	248	394
190	331	607
200	635	1740

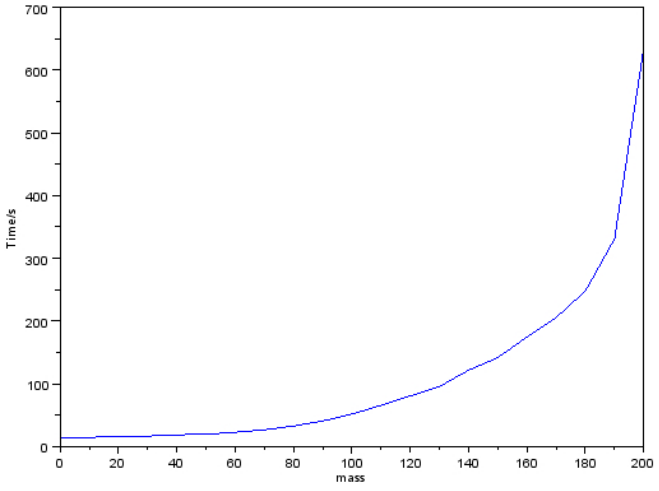


FIGURE 5.10: PGCR time taken against matrix mass.

The critical slowing down phenomenon is still clearly present and can be seen in the plot of time taken against matrix mass in graph 5.10. Figure 5.11 and figure 5.12 show the steady monotonic convergence of the preconditioned GCR algorithm at light and critical masses.

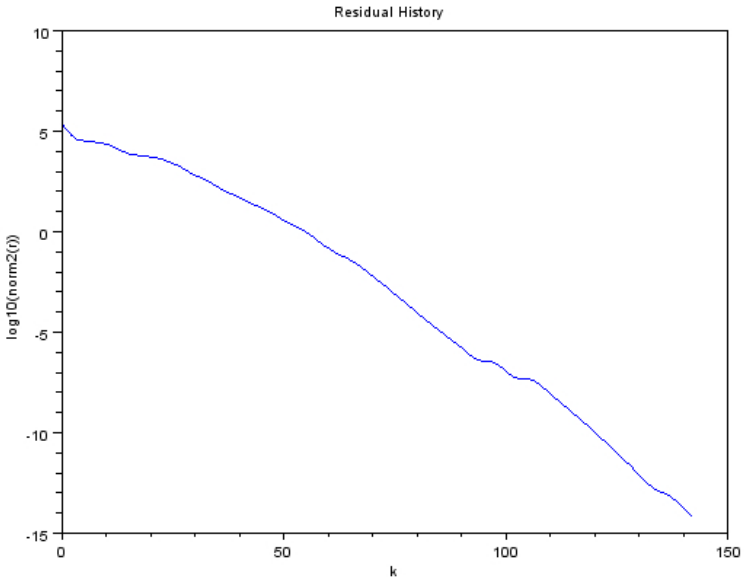


FIGURE 5.11: PGCR convergence N=10 and m=150.

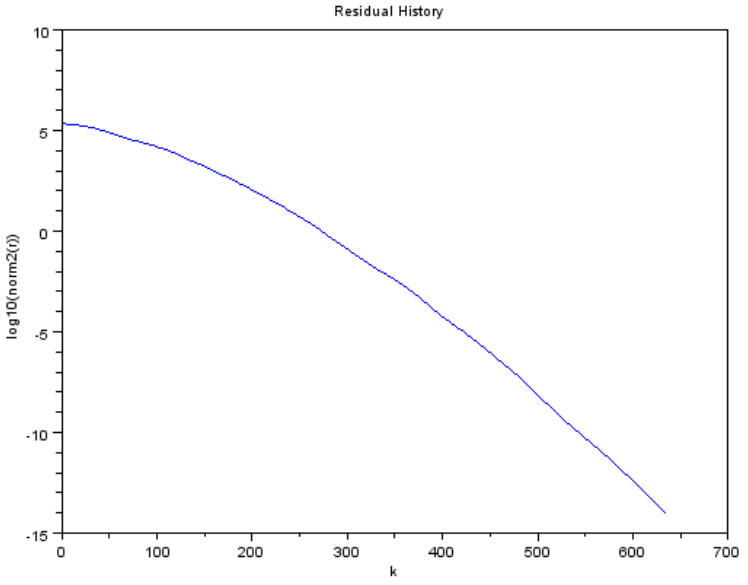


FIGURE 5.12: PGCR convergence N=10 and m=200.

5.4.2 PGCR Restarts

The speedup with a single SSOR cycle as preconditioner is promising but the results presented so far for GCR have been without restart, hence have relied on storing the complete history of update vectors. The figure 5.13 shows the convergence for the Wilson-Dirac matrix with lattice size N=10 but now with the GCR to be restarted after every 100 iterations. This takes the current iterative solution as the initial guess as progresses the GCR algorithm again as if from the very beginning.

With the mass at m=130 only 96 iterations are required to reach the required tolerance and hence the convergence is smooth as before. Figure 5.14 shows the dramatic effect that restarting has on the convergence: it stalls badly. The general pattern was for convergence to go sublinear then gradually improve as update vector history lengthened, until convergence accelerated again.

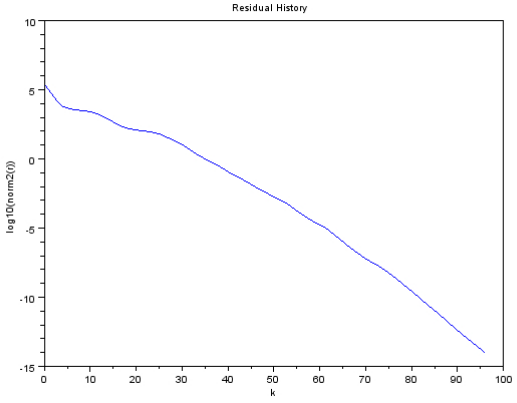


FIGURE 5.13: PGCR(L) convergence N=10 and m=130.

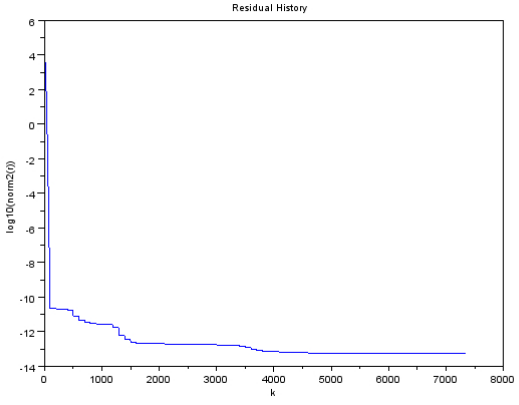


FIGURE 5.14: PGCR(L) convergence N=10 and m=140.

When the limit of 100 iterations was reached again, convergence would stall and become sublinear again. Figure 5.15 zooms into the first few hundred iterations to highlight this effect. Figure 5.16 shows the same effect for the worst case critical mass of m=200 as one would expect. For this worst case critical mass value the residual improved only by a factor of 100 before the sublinear slowing down was encountered, hence rendering this configuration unusable for practical purposes. Various permutations of restart mechanism were attempted, all resulting in the same sublinear slowing down on restart with the single-cycle SSOR preconditioner.

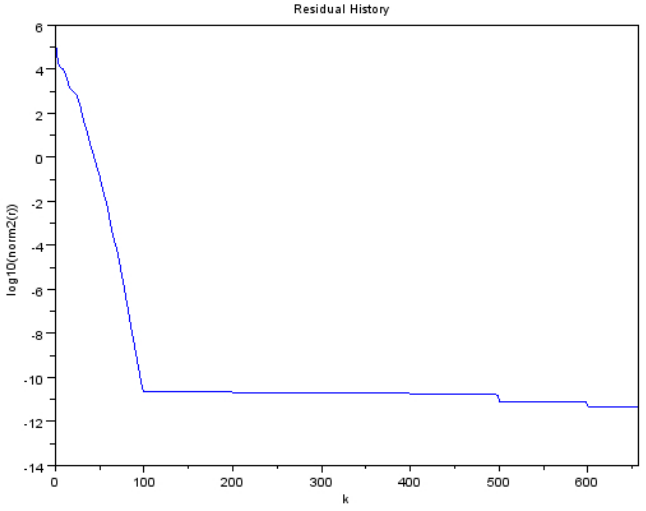


FIGURE 5.15: PGCR(L) convergence N=10 and m=140 close-up.

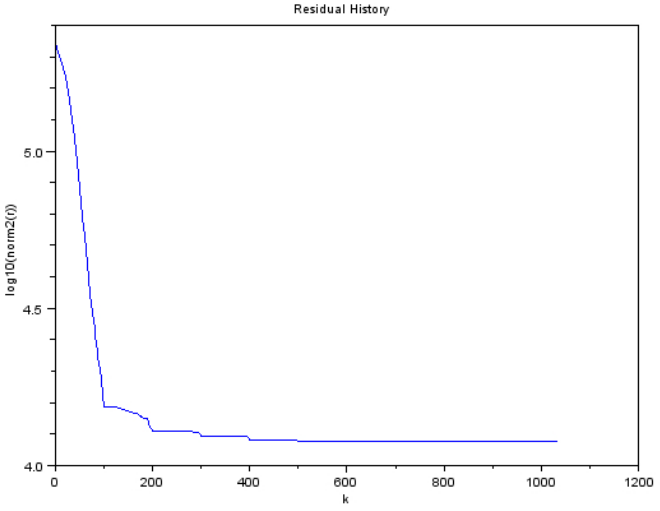


FIGURE 5.16: PGCR(L) convergence N=10 and m=200.

Testing restarted GCR without preconditioning reveals that it is the preconditioning with SSOR that causes this sublinear slowing down. In the absence of SSOR preconditioning, the restart mechanism works successfully, as demonstrated in table 5.17 and figure 5.18. In comparison against the GCR with no restarts shown in table 2.8 one can see that whilst the total number of iterations increases due to the restart, the total time taken actual decreases with restart.

mass	Iterations	Time / s
80	138	103
90	177	146
100	263	180
110	426	332
120	591	411
130	889	599
140	1348	981
150	1898	1360
160	2678	2232
170	3764	2773
180	5427	3606
190	10093	7382
200		

FIGURE 5.17: GCR(L) convergence against mass, with N=10.

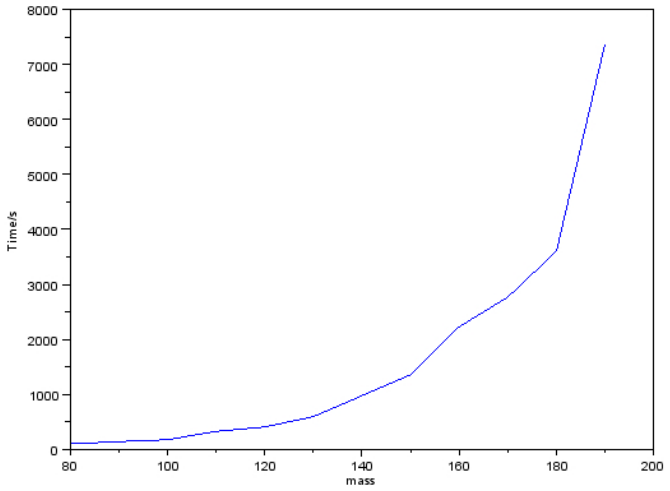


FIGURE 5.18: GCR(L) convergence against mass.

This time taken improvement with the restarts can be put down to the fact that orthogonalizing against a long sequence of prior update vectors in the non-restarted version is a time consuming process, and that the extra iterations required with restart is more than made up for by decreasing the the amount of orthogonalizing required. The restarted GCR is an improvement over the basic GCR method in both time and memory requirements for the harder to solve light fermion mass matrices.

Figure 5.19 shows the restart mechanism in progress for m=130, where for unpreconditioned GCR a restart causes an initial slowing down in convergence, but then the rate of convergence increases again and good progress is made. Figure 5.20 shows this pattern for even lighter mass m=180 where over 50 restarts are required to reach the required tolerance.

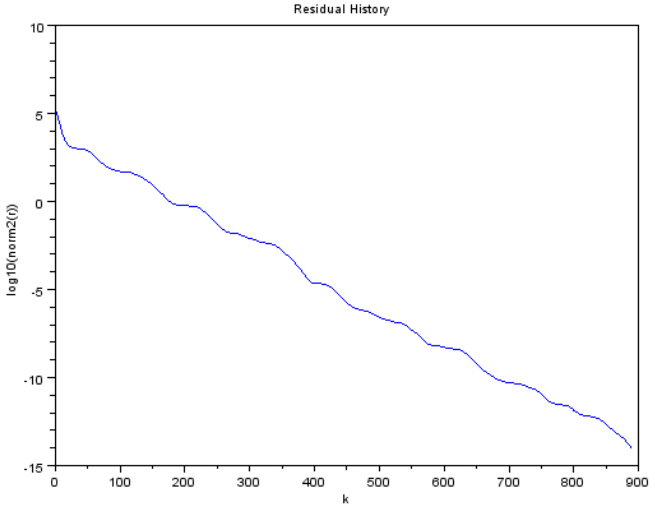


FIGURE 5.19: GCR(L) convergence with mass m=130.

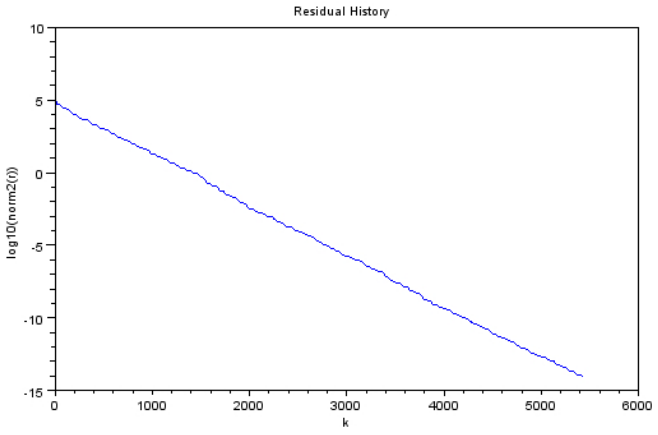


FIGURE 5.20: GCR(L) convergence with mass m=180.

5.5 Hybrid Preconditioned GCR

It would be beneficial to combine the memory savings of restarted GCR with the acceleration of SSOR preconditioning on critical mass matrices.

With the flexibility afforded by the object orientated design of the Werner software library various combinations were trialled and tested. Table 5.10 shows the results against mass with a nested GCR-GCR configuration. The outer GCR method is being preconditioned with an inner GCR solver, and that inner GCR solver is itself preconditioned with a single application of the SSOR smoother.

This double nested doubly preconditioned approach provides an improved speed up over either of the separate approaches. The inner solve is accelerated by the SSOR preconditioner, which then in turn accelerates the main GCR outer method. For masses below a value of 140 the inner GCR does almost all of the work, with only a single outer GCR loop necessary; with the configuration at 100 applications of inner GCR iterations.

Table 5.10: Hybrid PGCR performance and matrix mass.

mass	Iterations	Time / s
0	1	21.8
10	1	22.9
20	1	23.9
30	1	27.9
40	1	29.3
50	1	28.4
60	1	30.9
70	1	40.2
80	1	40.8
90	1	47.9
100	1	58.5
110	1	88.0
120	1	101
130	1	124
140	3	313
150	3	360
160	3	529
170	4	575
180	6	871
190	10	1873
200	31	4117

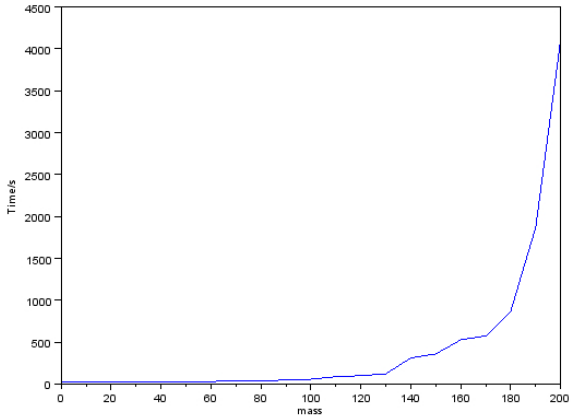


FIGURE 5.21: Hybrid PGCR time taken against matrix mass.

The plot of time taken against mass shown in figure 5.21 shows that this solver resists critical slowing down for longer in comparison to either the unpreconditioned GCR of figure 2.19 or the SSOR preconditioned GCR of figure 5.10.

The effectiveness of the SSOR preconditioner was compromised by a sublinear slowdown on restart of the GCR method. By using an outer GCR loop, effectively as a restart mechanism, this sublinear slowdown has been avoided. The preconditioner is doing most of the convergence, whereas preconditioning is normally expected to be less expensive than the main outer solver, so this is a non-traditional approach. The utility of this approach is primarily as a mechanism to successfully restart the inner loop convergence of SSOR preconditioned CGR; which had its potential speed up destroyed by sublinear restarts. Figure 5.22 shows the converge of the outer Krylov loop for the worst case lightest critical mass fermion of m=200.

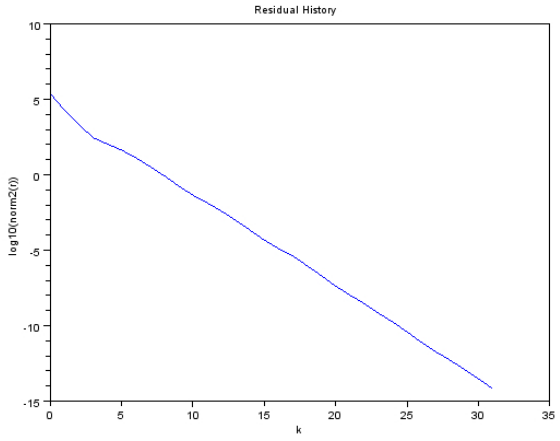


FIGURE 5.22: Hybrid PGCR convergence N=10 and m=200.

```
-sh-3.2$ time ./test1 --loops 100 --uncertainty 1e-14 --arg1 10
--solver PGCR --precon Monogrid --smoother K3 --restarts 100
--omega 1.1 --depth 1 --logtype +PGCR_K3_N10_m200 --arg2 200

MPI node 0 of 1. Processor: hpc24.its.uow.edu.au
Max Level: 64 Max Iterations: 100
Tolerance: 1e-14 Omega: 1.1 Depth: 1
Dirac.rank() = 240000
A.non_zeros  = 23280000
A.per_row_av = 97
-----------------------------------------------
                  Run Werner
-----------------------------------------------

Loop: 0  dot(r,r) = 240000
K3(0) norm(r) = 228726
K3(10) norm(r) = 203851
K3(20) norm(r) = 180590
K3(30) norm(r) = 141891
K3(40) norm(r) = 104483
K3(50) norm(r) = 71114.1
K3(60) norm(r) = 48152.7
K3(70) norm(r) = 32938.9
K3(80) norm(r) = 23278.7
K3(90) norm(r) = 15853.9
K3(100) norm(r) = 8694.31
Loop: 1  dot(r,r) = 18684.4
K3(0) norm(r) = 43436
K3(10) norm(r) = 40190.5
K3(20) norm(r) = 36844.2
K3(30) norm(r) = 32011.9
K3(40) norm(r) = 24071.6
K3(50) norm(r) = 17817.8
K3(60) norm(r) = 13417.1
K3(70) norm(r) = 9443.55
K3(80) norm(r) = 7263.6
K3(90) norm(r) = 5692.53
K3(100) norm(r) = 3757.84
Loop: 2  dot(r,r) = 2159.13
K3(0) norm(r) = 9879.29
K3(10) norm(r) = 9112.98
K3(20) norm(r) = 7912.91
K3(30) norm(r) = 6270.17
K3(40) norm(r) = 4626.24
K3(50) norm(r) = 3284.96
K3(60) norm(r) = 2443
K3(70) norm(r) = 1727.31
K3(80) norm(r) = 1317.35
K3(90) norm(r) = 1062.16
K3(100) norm(r) = 740.232
Loop: 3  dot(r,r) = 287.595

Loop: 31  dot(r,r) = 6.6292e-15
OUTPUT RESIDUAL-NORM: 8.14199e-08


 x = [ 1.72654; 1.75582; -1.79867; 2.20028; -1.06426; 1.61579; -0.766635;
      -1.46393; -1.63947; 3.55729; -1.31318; ]

 b = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]
Ax = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]
```

FIGURE 5.23: Command line output for GCR Hybrid-GCR-SSOR method.

Figure 5.23 shows the command line for the critical mass m=200 matrix on a lattice with width N=10 for a matrix of rank just under a quarter of a million. The inner GCR loop is configured to run 100 times with the inner loops also preconditioned themselves with the single application of SSOR set at an over relaxation of $\omega = 1.1$. The outer loop residuals can be followed with the lines beginning with *Loop* :, whilst the lines beginning with $K3$ show the progress of the inner hybrid SSOR-GCR preconditioner.

The inner loop GCR preconditioning typically causes an order of magnitude reduction with a hundred inner loop iterations being applied with the SSOR preconditioning the preconditioner. The preconditioner solution then returned to the outer GCR loop results in a corresponding ten-fold improvement in the main solution, which is the good progress highlighted in the convergence plot of figure 5.22. Of course the outer loops only requiring 31 iterations covers the fact that there are a hundred iterations within the preconditioning stages. A better measure of the number of iterations reflecting the true time complexity, inherent in the dominant matrix-vector multiply operation and MPI reduction calls, is naturally $31 \times 100 = 3100$.

Whilst the bulk of the convergence is achieved by the inner hybrid SSOR-GCR preconditioning stages and this now effective restart mechanism, it can be seen from figure 5.23 that the outer GCR loops can also contribute some convergence in their own right. Detailed inspect of more logs reveals this is not always the case, but the size of the preconditioner residual on exit is almost always close to the outer loop residual calculated after application of the preconditioner, and typically slightly less; sometimes significantly less.

K3 is the lookup codename for the hybrid GCR-SSOR preconditioner within the Werner software library. K3 is applied as a smoother within the multigrid framework when the multigrid framework configured to use the top level only; hence the name monogrid. The K3 smoother can also be used with multigrid preconditioning as well, which is shown in figure 5.24. The command line application is called with the configuration as in figure 5.23 except now the $--precon$ argument specifies Multigrid; the lattice width is again N=10 with the critical mass m=200.

The algebraic multigrid algorithm generates a total of five grids levels, where relative means the faction of grids points relative to the higher grid level and order give the rank of the matrix so created. The starting grid level is the original matrix, here of size 240,000 and level 0: this would correspond to the Monogrid preconditioner with only the original matrix used. The bottom most matrix is the trivial rank one scalar, and here that is level 5. With this simple algebraic multigrid setup the first new level has half the number of matrix rows with a relative of 0.5 and order 120,000.

```
-sh-3.2$ time ./test1 --loops 1000 --uncertainty 1e-14 --solver PGCR
--precon Multigrid --smoother K3 --restarts 100 --omega 1.1 --depth 1
--arg1 10 --arg2 200


MPI node 0 of 1. Processor: hpc26.its.uow.edu.au
Max Level: 64 Max Iterations: 1000
Tolerance: 1e-14 Omega: 1.1 Depth: 1
Dirac.rank() = 240000
A.non_zeros  = 23280000
A.per_row_av = 97
-----------------------------------------------
                  Run Werner
-----------------------------------------------
Relative: 0.0           Order: 240000   Level: 0
Relative: 0.5           Order: 120000   Level: 1
Relative: 0.00803333    Order: 964      Level: 2
Relative: 0.0238589     Order: 23       Level: 3
Relative: 0.0434783     Order: 1        Level: 4
Relative: 1             Order: 1        Level: 5

Loop: 0  dot(r,r) = 240000
K3(0) norm(r) = 5.42101e-20
K3(1) norm(r) = 5.42101e-20

K3(0) norm(r) = 1.342e+08
K3(10) norm(r) = 2.73583e-17

K3(0) norm(r) = 1.61699e+07
K3(10) norm(r) = 4.24979e-07
K3(17) norm(r) = 7.78918e-16

K3(0) norm(r) = 138927
K3(10) norm(r) = 1.29217e-09
K3(15) norm(r) = 1.42437e-15

K3(0) norm(r) = 4.01013e+06
K3(10) norm(r) = 3.28441e+06
K3(20) norm(r) = 2.62652e+06
K3(30) norm(r) = 1.77086e+06
K3(40) norm(r) = 948193
K3(50) norm(r) = 476982
K3(60) norm(r) = 272083
K3(70) norm(r) = 149301
K3(80) norm(r) = 100423
K3(90) norm(r) = 64631.5
K3(100) norm(r) = 39161
Loop: 1  dot(r,r) = 37847.9

Loop: 32  dot(r,r) = 1.85512e-14
Loop: 33  dot(r,r) = 4.52061e-15

OUTPUT RESIDUAL-NORM: 6.72355e-08


 x = [ 1.72654; 1.75582; -1.79867; 2.20028; -1.06426; 1.61579; -0.766635;
      -1.46393; -1.63947; 3.55729; -1.31318; ]

 b = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]
Ax = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]
```

FIGURE 5.24: Command line output for GCR Hybrid-GCR-SSOR with Multigrid.

After the first multigrid restriction the next levels reduce in size far quicker under the simple algebraic multigrid scheme. Inspection of the logging output shows each of the levels being solved in turn by the hybrid SSOR-GCR preconditioner K3. The small levels require few iterations within the preconditioner to reach the stopping criterion of 1e-14 in the square of the residual. As with the Monogrid setup, the largest matrix level requires the most work. The smaller matrix levels all converge to within the requested tolerance even on the very first iteration.

This multigrid configuration took 33 CGR loop iterations in 5016 seconds to return the answer, compared to 31 GCR loops in 4117 when using the non-multigrid version shown in figure 5.23. The extra overhead involved with setting up multigrid, solving the extra set of smaller matrices and performing the V-cycle restrictions and prolongations, has not resulted in an improved performance for the critical mass LQCD matrices. Double checking the sanity check printouts at the end one can see that the solutions returned are identical to six significant figures, but the multigrid solver took about 20% longer, and also used more memory.

Multigrid can be useful for solving LQCD matrices but requires a more complicated setup than simple algebraic multigrid [71]. Adaptive multigrid looks for the near zero eigenvalue modes and tries to remove them into a small independent subspace prior applying multigrid to the remaining matrix [74].

5.5.1 Matrix Size Behaviour

Table 5.11 shows the performance, of the hybrid SSOR-GCR preconditioner on outer GCR, against matrix size. The graph in figure 5.25 of time to solve versus matrix rank clearly shows non-linear characteristics, with the performance curving up until the lattice width N=16. This particular size of matrix seems to take longer for many of the variants of solver, which is likely to be caused by some aspect of the particular sequence of random numbers generated for the example resulting in above average ill-conditioning for the test trial of that matrix N=16.

The non-linearity is down to the increasing number of iterations required to reach the specified solution tolerance. This can be seen clearly by considering the time take per outer iterative loop. The data is plotted in graph 5.26 and shows a clear linear trend when considering the time taken per GCR outer loop against matrix rank. Each outer loop GCR with its inner hybrid SSOR-GCR is executing in a time proportional to the size of the matrix. This is as one would expect from the linear time complexity of the sparse matrix-vector multiplications within the algorithm.

TABLE 5.11: Hybrid PGCR performance and 100 restarts.

Lattice Width	Matrix Rank	Iterations	Time / s	Time per Iteration/s	Time/iter/million(s)
8	98,304	1	57.6	57.6	586
9	157,464	1	108	108	686
10	240,000	3	313	104	435
11	351,384	2	410	205	583
12	497,664	2	734	367	737
13	685,464	2	772	386	563
14	921,984	4	2016	504	547
15	1,215,000	4	3158	790	650
16	1,572,864	8	7669	959	609
17	2,004,504	6	6765	1128	562
18	2,519,424	7	9937	1420	563
19	3,127,704	7	11964	1709	546
20	3,840,000	7	14933	2133	556

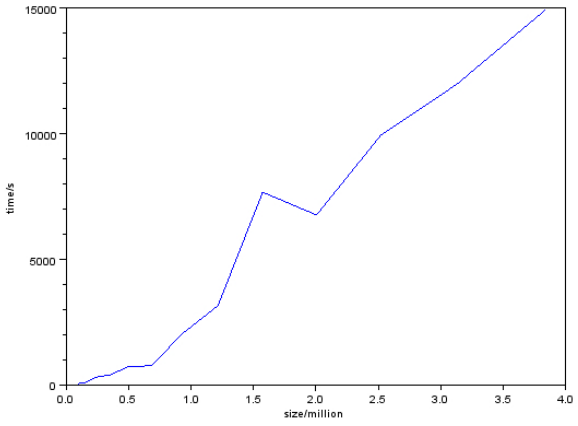


FIGURE 5.25: Hybrid PGCR time taken against matrix rank.

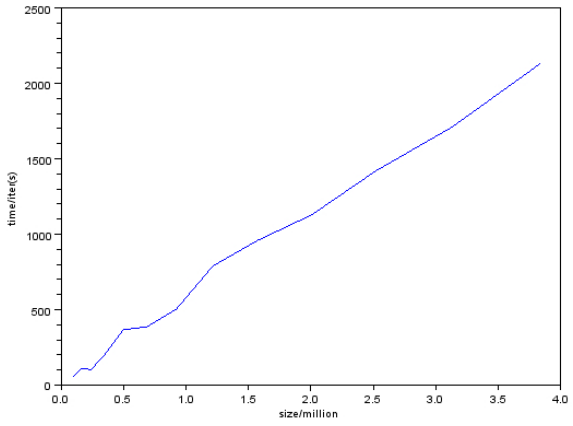


FIGURE 5.26: Time taken per GCR outer loop against matrix rank.

The quadratic nature of the overall solver time against matrix rank is from the slowly increasing number of iterations required to reach solution as the matrices grow in size. The last column in the table 5.11 shows that the time taken per iteration per million matrix rows is pretty constant at around 550 seconds to solve per million per iteration. The graph of total solver time in figure 5.25 also indicates linear for the last three matrix sizes where the number of outer iterations was a constant seven.

5.6 Preconditioned BiCGStab

Whereas GCR guarantees to minimise the residual, but at the expense of storing a history of previous vectors, BiConjugate Gradient based algorithms use a short-term recurrence relationship and do not need to keep that history. For nonsymmetric matrices it is not possible to have both short-term recurrences and a guaranteed residual minimisation [75], so for BiCG and variants the guaranteed minimisation is lost which can cause the convergence to become erratic, unstable, or even completely divergent. That said, if they do converge, BiCG and variants can be amongst the fastest solvers available for nonsymmetric matrices.

5.6.1 Matrix Size

Without preconditioning the easier to solve heavy mass fermion matrix with m=40 on a lattice of width N=20 took 42 iterations in 655 seconds using the improved convergence of the BiCG-stabilised algorithm; as detailed in table 2.5 of chapter 2. Now with single cycle SSOR preconditioner with $\omega = 1.1$, the performance improved to only 9 iterations in 423 seconds 5.12.

With far fewer Krylov iterations one has better scalability in terms of the far fewer scalar-reduce operations with the trade-off being the easier to scale SSOR operations with its nearest neighbour only updates over the sole application of the SSOR preconditioning. The total time taken sequentially is only a 35% improvement, but this

TABLE 5.12: PBiCGStab performance and matrix size.

Lattice Width	Matrix Rank	Iterations	Time / s
8	98,304	7	8.64
9	157,464	8	16.1
10	240,000	9	27.0
11	351,384	8	31.1
12	497,664	8	55.3
13	685,464	9	81.0
14	921,984	8	96.6
15	1,215,000	9	116
16	1,572,864	9	170
17	2,004,504	8	190
18	2,519,424	9	264
19	3,127,704	9	342
20	3,840,000	9	423

scalability improvement trading nearest neighbour updates for scalar-reduce broadcasts can be significant with respect to multi-processor scalability.

Figure 5.27 plots solver time against matrix size showing a scalable solver time of 12 seconds per million rows per iteration. Smooth convergence is shown in figure 5.28 for a far from critical mass m=40 matrix on a lattice of width N=20 corresponding to rows 3.84 million.

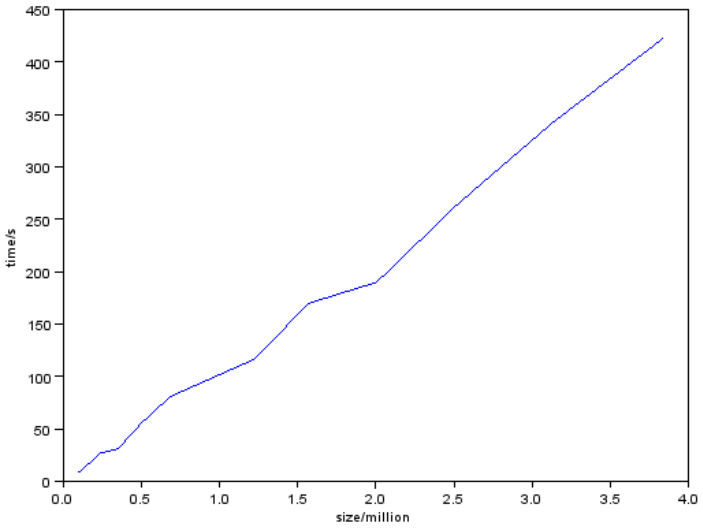


FIGURE 5.27: PBiCGStab time taken against matrix size.

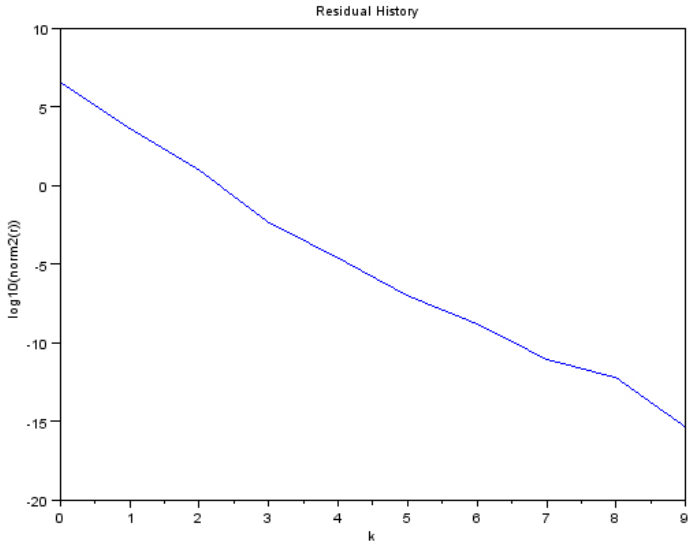


FIGURE 5.28: PBiCGStab convergence N=20 and m=40.

5.6.2 Matrix Mass

In contrast to SSOR preconditioned GCR which suffered critical slowing down but did eventually converge for m=200, for SSOR preconditioned BiCGStab the convergence behaviour with respect to mass broke down in line with ordinary SSOR breaking down. Table 5.13 shows the time to solve dependency against mass, and like SSOR divergence occurred shortly after the m=100 mark. The critical slowing down pattern is plotted in figure 5.29, and demonstrates that this preconditioned BiCGStab variant is of no use for critical mass valued Wilson-Dirac matrices.

TABLE 5.13: PBiCGStab performance and matrix mass.

mass	Iterations	Time / s
0	6	19.6
10	6	20.5
20	7	20.9
30	7	21.8
40	9	22.8
50	9	22.6
60	11	26.3
70	13	29.9
80	16	35.4
90	26	53.5
100	35	80.3
110	103	196
120		

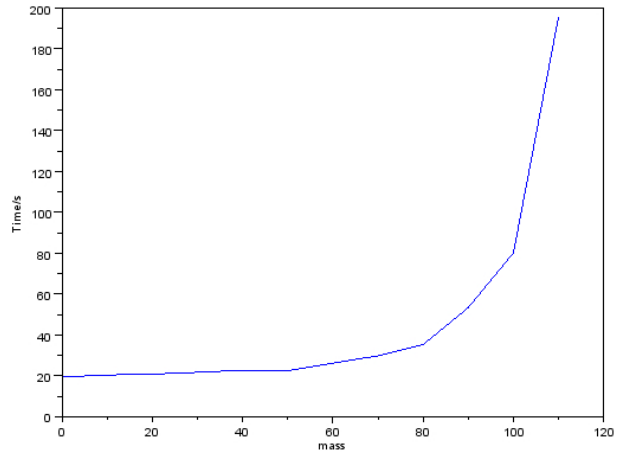


FIGURE 5.29: PBiCGStab time taken against matrix mass.

Figure 5.30 shows the convergence profile when the mass parameter is 110 on a lattice size of N=10. Progress is regularly made, albeit with some divergence that then recovers. Once mass is already only 120, as shown in figure 5.31, the initial progress is disrupted by sharp divergent spikes before completely stalling after 30 iterations and only limited progress to an improvement of only a factor of a thousand. In comparison to the data in chapter 2, table 2.6, preconditioning has raised the converging limit from a mass of 100, up to a mass of 110. This is still far from the required critical mass of m=200.

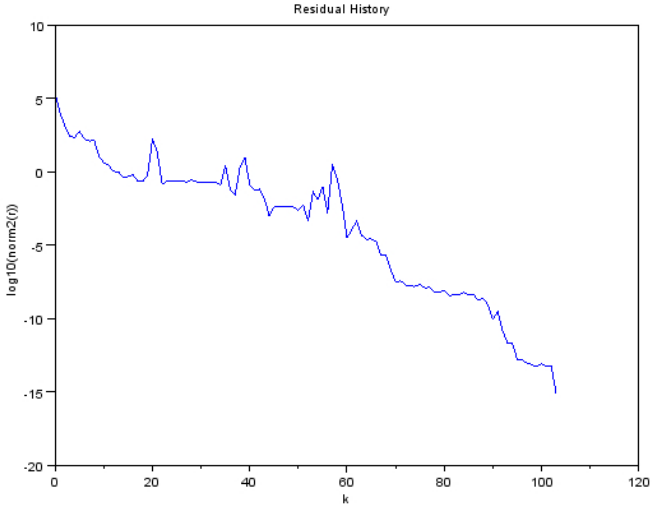


FIGURE 5.30: PBiCGStab convergence N=10 and m=110.

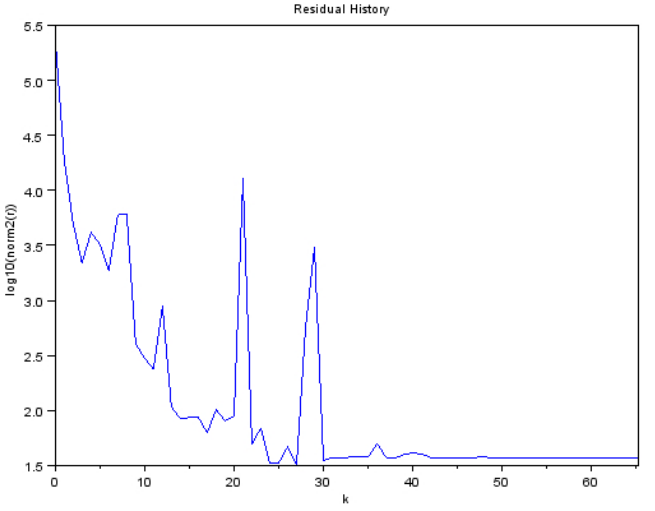


FIGURE 5.31: PBiCGStab convergence N=10 and m=120.

5.7 Hybrid Preconditioned BiCGStab

The effectiveness of the hybrid SSOR-GCR preconditioner can also be demonstrated for the BiCGStab method as shown in table 5.14. The results are comparable to the effect the hybrid preconditioner has on GCR itself, but in the case of BiCGStab one now has convergence for all masses, including the critical mass, where there was no convergence before.

As with PGCR for heavy fermion masses all the convergence was achieved within a single application of the hybrid preconditioner, requiring little work done in the outer Krylov method. The configuration of the hybrid preconditioner is again 100 inner iterations with a single SSOR with relaxation parameter set at $\omega = 1.1$. The time taken for heavy fermion masses is longer: this is due to the BiCGStab algorithm using the preconditioner twice per iteration, as opposed to GCR only using preconditioning once per iteration.

TABLE 5.14: Hybrid PBiCGStab performance and mass.

mass	Iterations	Time / s
0	1	34.4
10	1	35.4
20	1	37.2
30	1	40.5
40	1	43.2
50	1	48.2
60	1	51.2
70	1	62.1
80	1	71.7
90	1	81.9
100	1	111
110	1	133
120	1	176
130	1	213
140	2	357
150	1	319
160	2	506
170	2	469
180	3	744
190	6	1459
200	30	9307

As with PGCR, the value of the hybrid preconditioner becomes apparent when the 100 SSOR-GCR iterations in the preconditioner are insufficient to achieve full convergence alone, and hybrid preconditioning is effectively used as a restart mechanism. As one would expect, this occurs for both hybrid variants at the same mass value of 140, since the hybrid SSOR-GCR preconditioner is the same in both cases. A notable difference however is the outer GCR loop requires 3 iterations to solve m=140 but BiCGStab only requires 2 iterations for that m=140 matrix, but that would equate to 4 calls to the hybrid preconditioner in total, so the PBiCGStab version still takes slightly longer at 357 seconds compared to 313 seconds for PGCR.

As with PGCR and the hybrid preconditioner, graph 5.32 plotting solver time against mass for PBiCGStab with hybrid preconditioner, shows that the critical slowing down has been delayed until closer to the critical mass compared to other solvers. The critical slowing down is even slightly better, with the PBiCGStab having a faster solver time over the light fermion mass range of m=150 to a critical m=190. The solver time for PBiCGStab with hybrid SSOR-GCR did perform worse though at the most critical value of m=200. At this value both outer Krylov solvers required a similar number of iterations, but PBiCGStab took twice as long with it requiring two calls to the preconditioner per outer iteration.

Adjusting the preconditioner between the two calls within a PBiCGStab iteration was attempted but it was found that the full hybrid SSOR-GCR preconditioner was required at both points within the algorithm.

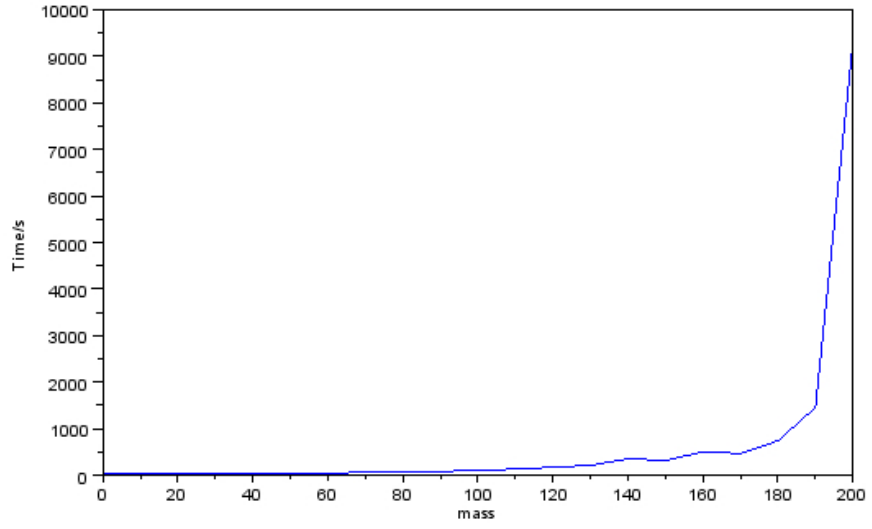


FIGURE 5.32: Hybrid PBiCGStab time taken against mass.

For example not calling the preconditioner at its first point of usage to save time there, resulted in a strong divergence on the critical mass matrices from which the second preconditioning call could not recover. Both calls to preconditioning with the BiCGStab method required the full hybrid SSOR-GCR to be applied.

Figure 5.33 shows the convergence of PBiCGStab with hybrid SSOR-CGR preconditioner for the light fermion case of m=190: the convergence is smooth even for this light mass value. For the most critical mass value of m=200 the convergence becomes less smooth as shown in figure 5.34, but the progress is still very good, and the hybrid preconditioner has successfully induced convergence in the worse case matrix and smoothed out the dramatic fluctuations that can often be seen in BiCGStab's progress for these ill-conditioned matrices.

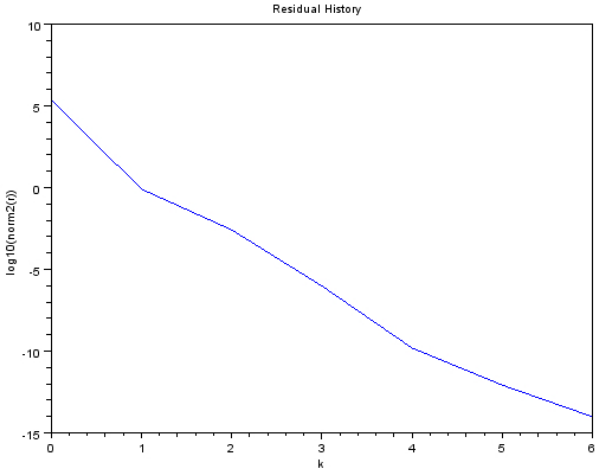


FIGURE 5.33: Hybrid PBiCGStab convergence N=10 and m=190.

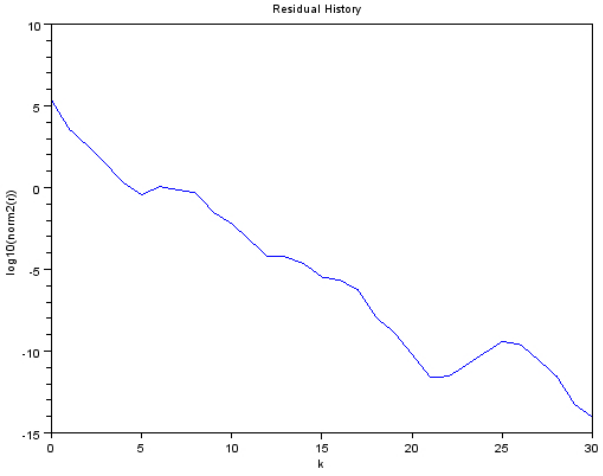


FIGURE 5.34: Hybrid PBiCGStab convergence N=10 and m=200.

```
-sh-3.2$ time ./test1 --loops 10000 --uncertainty 1e-14 --arg1 10
--solver PBiCGStab --precon Monogrid --smoother K3 --restarts 100
--omega 1.1 --depth 1 --logtype +PBiCGStab_SSOR_N10_m180 --arg2 180

MPI node 0 of 1. Processor: hpc24.its.uow.edu.au
Max Level: 64 Max Iterations: 10000
Tolerance: 1e-14 Omega: 1.1 Depth: 1
Dirac.rank() = 240000
A.non_zeros  = 23280000
A.per_row_av = 97
-----------------------------------------------
                 Run Werner
-----------------------------------------------
Loop: 0  dot(r,r) = 240000
K3(0) norm(r) = 194730
K3(10) norm(r) = 126681
K3(20) norm(r) = 80971
K3(30) norm(r) = 31868.2
K3(40) norm(r) = 10244.6
K3(50) norm(r) = 2711.88
K3(60) norm(r) = 470.595
K3(70) norm(r) = 118.843
K3(80) norm(r) = 28.9607
K3(90) norm(r) = 5.12984
K3(100) norm(r) = 1.18198

K3(0) norm(r) = 194735
K3(10) norm(r) = 126805
K3(20) norm(r) = 81081.7
K3(30) norm(r) = 31882
K3(40) norm(r) = 10264.1
K3(50) norm(r) = 2716.68
K3(60) norm(r) = 471.737
K3(70) norm(r) = 97.7741
K3(80) norm(r) = 27.4897
K3(90) norm(r) = 4.156
K3(100) norm(r) = 1.17467
Loop: 1  dot(r,r) = 0.00294765

K3(0) norm(r) = 4.31749
K3(10) norm(r) = 2.53644
K3(20) norm(r) = 1.16508
K3(30) norm(r) = 0.394658
K3(40) norm(r) = 0.122786
K3(50) norm(r) = 0.027957
K3(60) norm(r) = 0.00444067
K3(70) norm(r) = 0.00109122
K3(80) norm(r) = 0.00029351
K3(90) norm(r) = 5.01558e-05
K3(100) norm(r) = 1.13787e-05

K3(0) norm(r) = 0.000830432
K3(10) norm(r) = 0.00010504
K3(20) norm(r) = 4.49241e-05
K3(30) norm(r) = 2.15646e-05
K3(40) norm(r) = 7.47085e-06
K3(50) norm(r) = 1.55354e-06
K3(60) norm(r) = 2.65896e-07
K3(70) norm(r) = 6.33869e-08
K3(80) norm(r) = 1.52176e-08
K3(90) norm(r) = 3.86421e-09
K3(100) norm(r) = 5.22292e-10

Loop: 2  dot(r,r) = 5.2209e-10
```

FIGURE 5.35: Hybrid PBiCGStab command line output.

```
K3(0) norm(r) = 0.000721007
K3(10) norm(r) = 4.06285e-05
K3(20) norm(r) = 4.37671e-06
K3(30) norm(r) = 6.03433e-07
K3(40) norm(r) = 1.58155e-07
K3(50) norm(r) = 3.30297e-08
K3(60) norm(r) = 4.82257e-09
K3(70) norm(r) = 1.14914e-09
K3(80) norm(r) = 2.41053e-10
K3(90) norm(r) = 5.1649e-11
K3(100) norm(r) = 7.211e-12

K3(0) norm(r) = 8.56602e-10
K3(10) norm(r) = 4.8914e-10
K3(20) norm(r) = 2.56257e-10
K3(30) norm(r) = 7.88031e-11
K3(40) norm(r) = 2.27277e-11
K3(50) norm(r) = 4.78458e-12
K3(60) norm(r) = 8.46593e-13
K3(70) norm(r) = 2.00805e-13
K3(80) norm(r) = 5.29788e-14
K3(90) norm(r) = 1.03102e-14
K3(92) norm(r) = 8.4403e-15
Loop: 3  dot(r,r) = 8.43911e-15

OUTPUT RESIDUAL-NORM: 9.18646e-08


 x = [ 1.09604; 1.36379; -1.30755; 1.35647; 0.288112; 1.09658; -0.648108;
       0.287746; 0.29291; 0.298302; -0.665889; ]

 b = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]
Ax = [ 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ]
```

FIGURE 5.36: Hybrid PBiCGStab command line output.

The command line and output for the N=10 and m=180 test run is shown between the figure 5.35 and figure 5.36. The relationship between the progress of the two calls to the hybrid SSOR-GCR preconditioning stages within a single outer Krylov iteration can be seen. The fact that both preconditioning calls are required is shown with 'Loop: 2' and 'Loop: 3', where the convergence of the second preconditioning call is following on from the convergence of the first preconditioning call.

5.7.1 Matrix Size Performance

Table 5.15 and figure 5.37 show the performance of hybrid SSOR-GCR preconditioned BiCGStab against matrix size for the case of fermion mass being 180 and restarts set at 100. The time to solve per iteration per million rows is uniform at a value of about 1000 seconds, but the crucial value is the number of iterations required to reach the required tolerance is dramatically increasing causing a large increase in overall solution time.

With two preconditioner calls per outer Krylov loop the time per iteration per million rows has doubled, but the time per iteration is still growing linearly with respect to matrix size. The major issue is the growth in the total number of outer Krylov loops required to reach solution accuracy contributing to the over-linear increase in solution times.

TABLE 5.15: Hybrid PBiCGStab performance with 100 restarts for mass 180.

Lattice Width	Matrix Rank	Iterations	Time / s	Time per Iteration/s	Time/iter/million(s)
8	98,304	5	567	113	1154
9	157,464	3	465	155	984
10	240,000	3	744	248	1033
11	351,384	4	1334	334	949
12	497,664	5	2602	520	1046
13	685,464	6	4235	706	1030
14	921,984	8	7464	933	1012
15	1,215,000	11	14291	1299	1069
16	1,572,864	27	50648	1876	1193

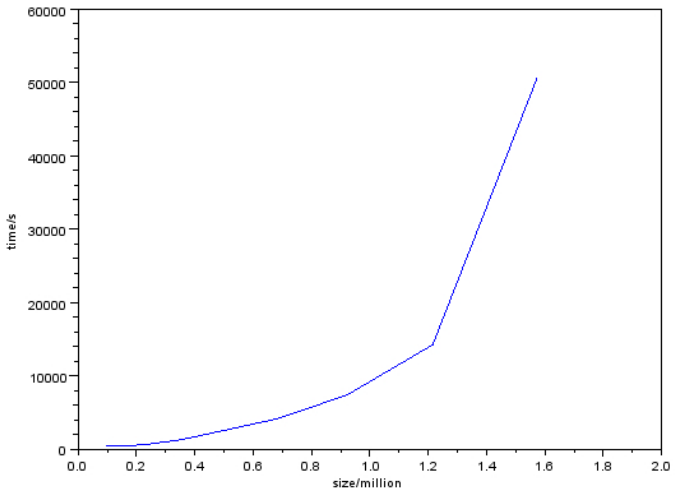


FIGURE 5.37: Hybrid PBiCGStab time taken against matrix size, r=100, m=180.

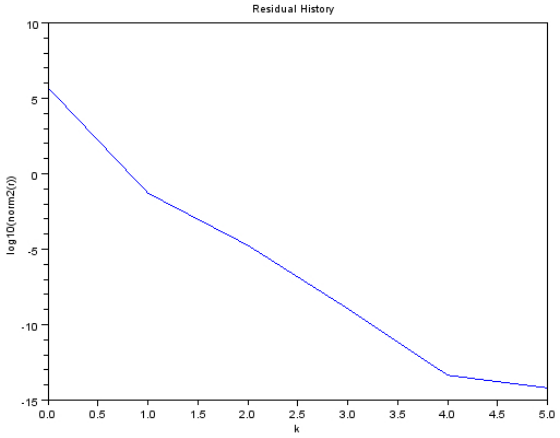


FIGURE 5.38: Hybrid PBiCGStab convergence N=12, m=180, restarts=100.

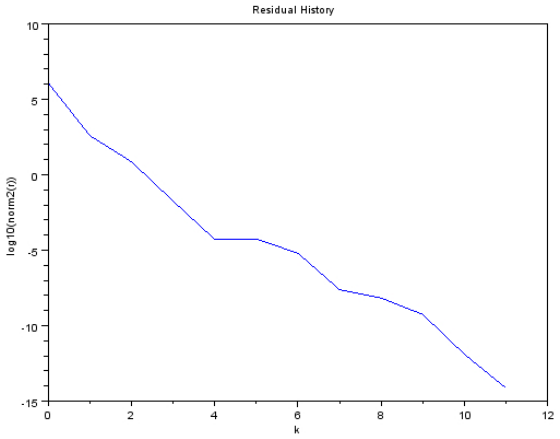


FIGURE 5.39: Hybrid PBiCGStab convergence N=15, m=180, restarts=100.

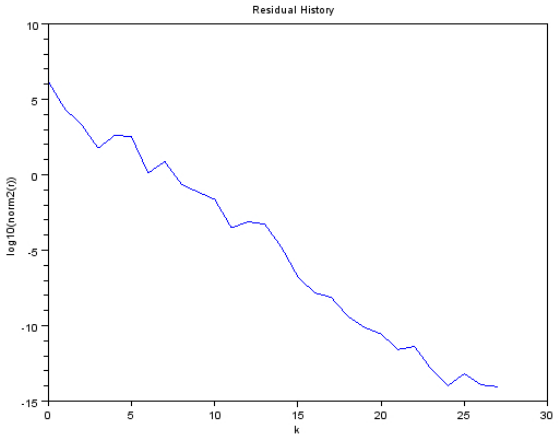


FIGURE 5.40: Hybrid PBiCGStab convergence N=16, m=180, restarts=100.

Figures 5.38, 5.39, and 5.40 show the effect that increasing matrix size is having on the convergence profile with fermion mass at m=180 and restarts set to 100. As the size of the matrix increases the irregular convergence is returning to the BiCGStab algorithm. When N=16 the matrix is more ill-conditioned than usual anyway, as has been seen for PGCR with hybrid preconditioner in figure 5.25, but here for PBiCGStab the consequences are more pronounced.

5.7.2 Preconditioner Restarts

To improve the convergence of PBiCGStab one can increase the length of the hybrid SSOR-GCR preconditioner stage in order to smooth out the convergence of the outer PBiCGStab iterations. Table 5.16 shows the behaviour of Hybrid preconditioned BiCGStab against hybrid preconditioner restarts. As one would expect the greater the length of the inner SSOR-GCR stage before restarting, the fewer outer Krylov iterations that are required. The data in table5.16 was collected with mass m=140 on a lattice width of N=10 and is also plotted in figure 5.41. There is an optimum restart length at about 70 restarts which manages a turnover of 20 solves per hour. For these matrices, once the restart reaches 110 all the convergence is achieved by the preconditioning anyway and the outer iteration becomes immaterial so the time settles down to a value of 13 solves per hour.

TABLE 5.16: Hybrid PBiCGStab performance and restarts.

restarts	Iterations	Time / s	Solves/hr
10	32	845	4.26
20	16	721	4.99
30	6	422	8.53
40	4	384	9.38
50	2	248	14.5
60	2	254	14.2
70	1	179	20.1
80	2	294	12.2
90	2	316	11.4
100	2	275	13.1
110	1	272	13.2
120	1	283	12.7
130	1	277	13.0
140	1	277	13.0
150	1	282	12.8

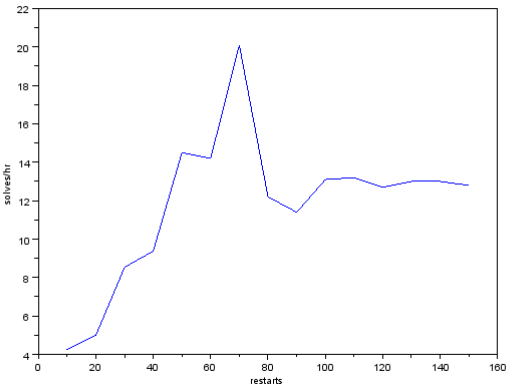


FIGURE 5.41: Hybrid PBiCGStab time taken against restarts.

Figure 5.42 shows the outer iteration convergence for the case of restarts every 10 inner SSOR-GCR repetitions, which can be compared to the smoother convergence progress when restarts are trebled to 30 as shown in figure 5.43.

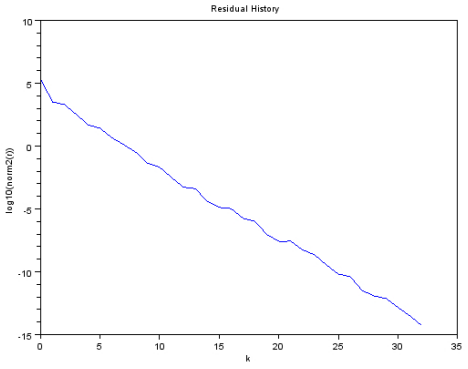


FIGURE 5.42: Hybrid PBiCGStab convergence N=10, m=140, restarts=10.

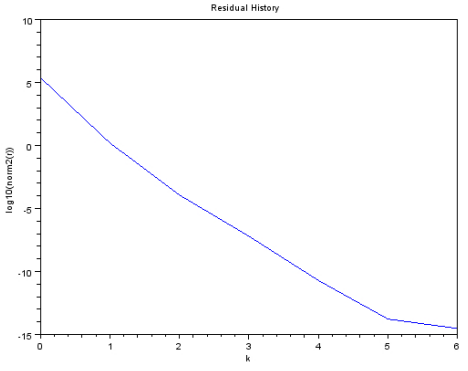


FIGURE 5.43: Hybrid PBiCGStab convergence N=10, m=140, restarts=30.

5.8 Hybrid Flexible BiCGStab

The hybrid SSOR-GCR preconditioner acting on the BiCGStab algorithm vastly improves the convergence of the outer BiCGStab convergence for the critical LQCD matrices but some irregular convergence is still present. In particular as the residual approaches the required stopping criterion tolerance, to have a small divergence at that point can cause a significant delay if the inner Krylov iteration of the hybrid preconditioner is again invoked. Whilst convergence has been shown to continue, this irregularity and potential slowdown can be removed by a minor reconfiguration of the hybrid preconditioner.

The data presented for the hybrid preconditioner so far is based on the preconditioner applying a pre-specified number of inner Krylov iterations using the Werner command line parameters *–restarts*. As shown in section 5.7.2 on preconditioner restarts, the larger the number of inner loop iterations the faster the outer method convergences. In order to achieve a better convergence for the outer Krylov method one can specify a multiplicative improvement factor instead of a fixed number of inner iterations. This has the effect of smoothing out the remaining irregularity with the progress of the outer BiCGStab method, resulting in a monotonic convergence pattern even for the most critical LQCD matrices.

Figure 5.44 shows the convergence of the hybrid SSOR-GCR preconditioner for lattice width $N = 10$ and critical mass $m = 200$ for the setup that the hybrid preconditioner will exit its solve when the residual has improved by a specified factor of one hundred.

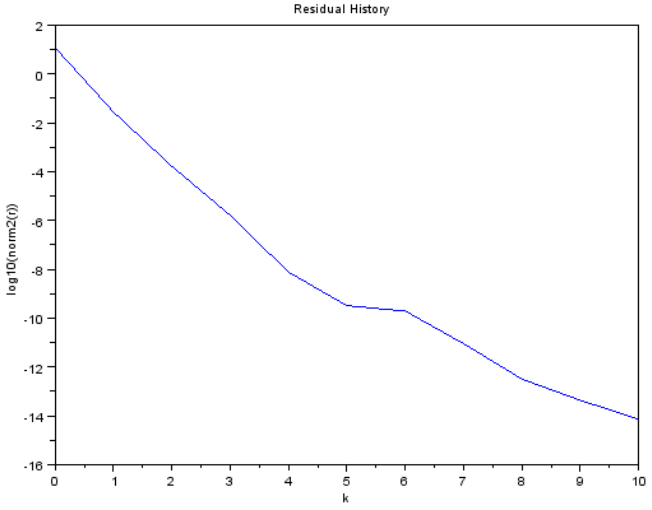


FIGURE 5.44: Hybrid PBiCGStab performance for critical LQCD matrix.

The graph in figure 5.44 can be compared to the graph in figure 5.34 to see that the small amount of irregular convergence that was still present has now been completely removed. The convergence is now completely monotonic and steady with the only exception being between the fifth and sixth steps when convergence temporarily declined, but still decreased albeit by a small amount.

Table 5.17 and the plot in figure 5.45 show the scalability of this hybrid SSOR-GCR preconditioner variant. The additional number of inner iterations required to perform the preconditioning steps to the necessary residual improvement causes a distinct non-linearity with respect the time complexity performance once the matrix size exceeds a million for this configuration of hybrid solver.

TABLE 5.17: Hybrid PBiCGStab performance and matrix size.

Lattice Width	Matrix Rank	Iterations	Time / s	Time per Iteration/s	Time/iter/million(s)
8	98,304	10	4803	480	4886
9	157,464	11	3914	356	2260
10	240,000	9	4652	517	2154
11	351,384	13	10898	838	2386
12	497,664	9	11760	1307	2626
13	685,464	8	17719	2215	3231
14	921,984	8	27496	3437	3728
15	1,215,000	8	53868	6734	5542
16	1,572,864	8	143292	17912	11388
17	2,004,504	8	146110	18264	9111

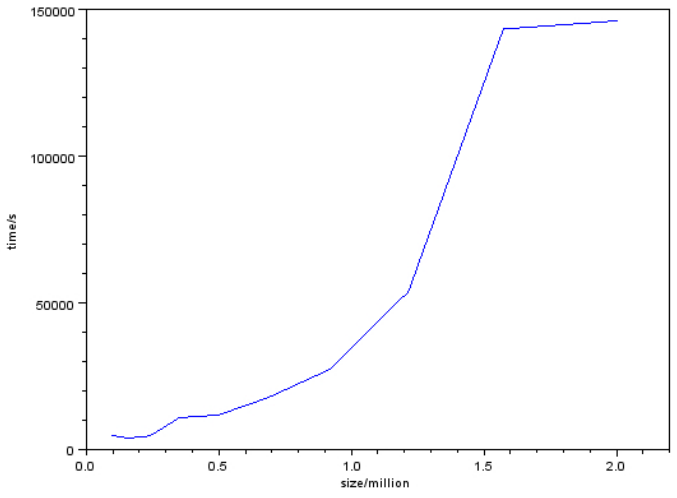


FIGURE 5.45: Hybrid PBiCGStab performance and matrix size.

5.9 Multi Processing

The hybrid SSOR-GCR preconditioner presented has been investigated with requirement of high scalability predominant. The sparse nature of the Wilson-Dirac matrix allows one to take advantage of nearest neighbour only communication resulting in fixed time complexity scalability of vector element updates with respect to matrix size. The main impediment to scalability, being the global scalar reduce operation of the inner product calculations, are still present but can be organised to overlap with computations to minimise the delay. The communication delay caused by the scalar reduce can be made to have time complexity of $O[logN]$ by arranging the communication in a tree formation [76].

Table 5.18 demonstrates the potential scalability of the solver when executed as a multi-threaded process using POSIX-threads, Pthreads; the graph is plotted in figure 5.46. The loop per second value is the time taken for a single top-level PBiCGStab iteration

TABLE 5.18: Hybrid performance against Pthreads.

Threads	Loops/s
1	0.054
2	0.105
3	0.155
4	0.197
5	0.230
6	0.260

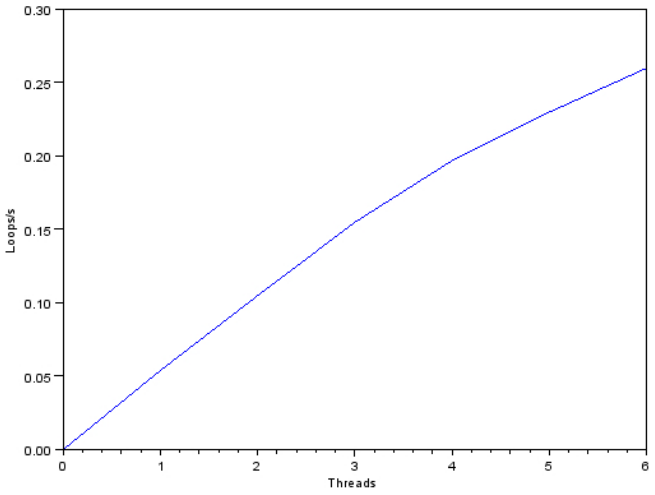


FIGURE 5.46: Hybrid performance against Pthreads.

with hybrid SSOR-GCR preconditioning. The lattice width is N=10 and the mass value is the heavy quark mass with m=40. This test was executed on a single cluster box, so a maximum of eight cores where available if not in use by other users. As the number of cores used increased the number of loops per second executed increase almost in direct proportion until the sixth thread was added.

The exercise was repeated with MPI processes instead and the results are given in table 5.19 and plotted in figure 5.47. Similiar results are shown when the algorithm is spread across CPUs using MPI instead of Pthreads. The data presented here is very small scale and needs to be expanded to hundreds and preferably thousands of cores and hence is just at the stage of initial proof of concept that the hybrid preconditioned Krylov solver can be effectively invoked over a distributed system. The fact that the results are very similar for both Pthreads and MPI is encouraging and demonstrates valid partitioning of the matrix data and algorithm.

TABLE 5.19: Hybrid performance against MPI nodes.

Nodes	Loops/s
1	0.050
2	0.103
3	0.236
4	0.254
5	0.263
6	0.312
8	0.341

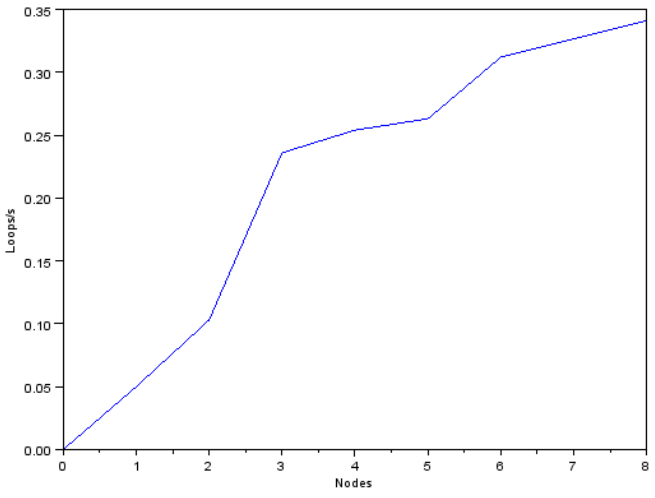


FIGURE 5.47: Hybrid performance against MPI nodes.

Chapter 6

Conclusions and Further Work

"Just like a computer, we must remember things in the order in which entropy increases. This makes the second law of thermodynamics almost trivial. Disorder increases with time because we measure time in the direction in which disorder increases. You can't have a safer bet than that!"

Stephen Hawking

By developing the highly flexible object orientated numerical framework Werner, a numerical investigation was performed to profile the performance of iterative solvers used with the very large matrices of Lattice QCD. The emphasis has been on discovering a hybrid solver combination that can be configured to scale well with matrix size over massively parallel architectures built from modestly priced commodity components. This has lead to a nested Krylov approach with two layers of preconditioning in an attempt to combine the advantages of fast BiCGStab convergence, with the guaranteed residual minimisation of GCR, and with the high frequency smoothing benefits of stationary Jacobi variants. To address the critical slowing down problem inherent in the lattice modeling of continuum quantum dynamics it was discovered that the hybrid preconditioner created could also be used as a variable preconditioner. The configuration with the preconditioner changing between iterations affects the Krylov subspace generation and the algorithm has turned into a flexible Krylov subspace method. A lack of provable convergence for BiCGStab combined with this variable preconditioning means flexible BiCGStab is an area with little prior literature within the field of LQCD.

6.1 Hybrid Preconditioning

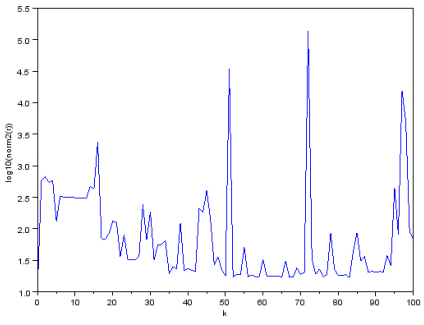
The size of Wilson-Dirac matrices will soon exceed a billion rows as the demand to investigate the hadron spectrum of lighter quarks becomes feasible with the resources of ever larger supercomputing clusters. Making full use of these massively parallel architectures requires algorithms with an extremely high degree of concurrency, and this now means algorithms with low communication overhead between processor nodes. The algorithms investigated here can take advantage of the fixed time complexity of nearest neighbour communication with the primary data bandwidth bottleneck being the $O(logN)$ scalar products.

The algorithm found to be most effective involved using GCR as a preconditioner, itself preconditioned with SSOR, to give a doubly preconditioned algorithm. The GCR as a preconditioner on its own was usable but its effect was greatly enhanced by the use of SSOR as an extra smoothing preconditioner on the Krylov preconditioner itself.

This hybrid GCR-SSOR preconditioner can be used to accelerate the convergence of both an outer Krylov method, such as another GCR solver or BiCGStab method. When using an outer GCR solver the hybrid preconditioner provides an effective restart mechanism. This is useful because SSOR preconditioned GCR suffered from stalling on regular restarting, and would take a long time to recover convergence. With highly ill-conditioned matrices this was a severe limitation, which can now be worked around with the hybrid preconditioner mechanism.

GCR outer with SSOR-GCR hybrid inner was the most effect solver for the worst case matrices where it provided an effective restart mechanism. The SSOR-GCR hybrid had an even more pronounced effect on actually getting BiCGStab to converge.

SSOR preconditioning



Hybrid SSOR-GCR preconditioning

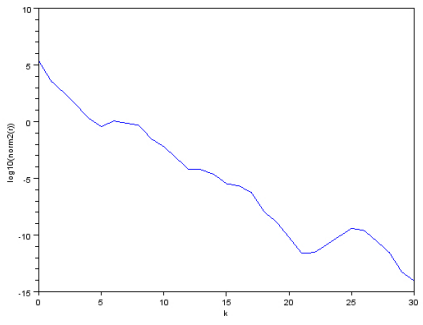


FIGURE 6.1: The effect of SSOR-GCR hybrid preconditioning with low mass matrices.

For the worst case matrices the simply preconditioned BiCGStab was highly erratic and failed to converge anywhere near a solution, but with the action of the hybrid preconditioner convergence was induced even for the worst case light mass Wilson-Dirac matrix. Figure 6.1 shows how convergence changes from simple SSOR preconditioned BiCGStab to BiCGStab with the hybrid preconditioner tuned to 300 GCR preconditioner repeats each with a single SSOR smoother.

The slight irregularity with the hybrid preconditioner can be further smoothed out by allowing the preconditioner to be variable, giving rise to a flexible preconditioned BiCGStab variant. The flexible variant works by executing the inner GCR preconditioner iterations until the preconditioner solution has converged by a preset factor, which was the *–gamma* factor on the Werner command line. Figure 6.2 shows the monotonic convergence of the flexible hybrid preconditioned BiCGStab with the irregularity completely removed.

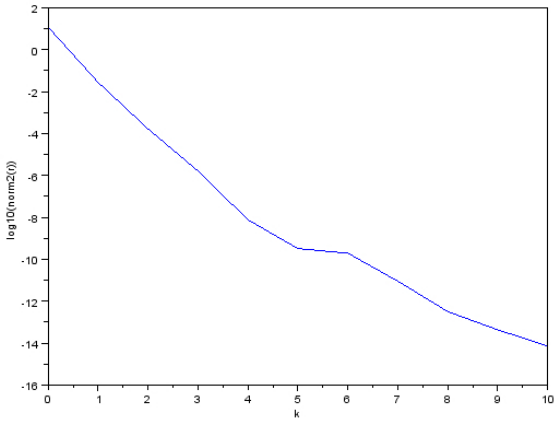


FIGURE 6.2: Flexible BiCGStab monotonic convergence.

The combination of variable length preconditioner GCR iterations and the controllable SSOR lower level preconditioning provide possible runtime adaptability. Matrices at the critical point will diverge if more than one SSOR iteration is applied, so a single iteration is optimal for the most critical matrices. For the cases that the matrices requiring solution are not quite so ill-conditioned it may be beneficial to use more repetitions of the SSOR component of the hybrid preconditioner. This adaptive algorithm would be straight forward to code by testing the effect of increasing or decreasing the *–depth* parameter within Werner at runtime and settling on the one that provides the best rate of convergence.

Flexible BiCGStab is not an algorithm that receives much attention in the literature but was recently considered for nonsymmetric matrices derived from two dimensional partial differential equations [77]. Flexible BiCGStab was reported as no faster than

preconditioned BiCGStab when both converged but was noted to reach a solution in the cases regular preconditioned BiCGStab failed. The same author has also done some work on flexible Quasi Minimal Residual [78] which has better convergence properties than BiCG variants, more similar to GMRES, but is a more complicated algorithm to scale for parallel architectures. QMR has been discussed for parallel architectures and would be an area to investigate further [79].

GCR is not perfect for use with these very large matrices over massively parallel architectures due to the high number of scalar reduces required. The bandwidth can be kept under control by grouping together the large number of scalar reduces in the final for-loop that implements the orthogonalisation process since each vector update in the for-loop is independent of all others. GMRES is a popular method for non-symmetric matrices, with similar performance and issues as GCR, but has been investigated to see how possible it is to largely eliminate communication overhead [80]. This research into communication avoiding sparse matrix operations [81] is a key area of future work to integrate into Werner when further scalability tests are undertaken on larger numbers of processing units. The communication avoiding techniques for GMRES should carry across to GCR to ensure the theoretical $O(logP)$ scalability, where P is the number of processors.

Some theoretic analysis of flexible BiCGStab has been attempted [82] but the conclusions are still unclear; although practical utility was again demonstrated for non-LQCD matrices. In the context of LQCD, flexible BiCGStab has been investigated using BiCGStab itself as an inner Krylov preconditioner using single precision on GPUs with double precision outer iterations [83]. This is a popular configuration since the preconditioner is only an approximation so one can take advantage by using the faster single precision arithmetic for the inner preconditioner solve. This technique can also be used on PCs with C/C++ by choosing the $float$ type instead of the *double* type [84].

With BiCGStab not necessarily minimising the residual, and proofs not otherwise being available for convergence of BiCGStab anyway, the success of the hybrid GCR-SSOR algorithm as a method to induce convergence on BiCGStab is simply established via empirical means. The convergence patterns presented in this thesis being the actual residual $A\vec{x} - \vec{b}$ explicitly calculated and the solution double checked by back substitution to verify the vector $\vec{b}$ is correctly retrieved from $A\vec{x}_k$. The convergence of the flexible variant is likely due to the fact that the variations of each preconditioner on each difference loop are very small such that the perturbation on the Krylov subspace compared to a standard fixed preconditioner is also very small. Since the outer BiCGStab loops converge in a very small number of iterations, about a dozen, compared to the total size of the Krylov subspace being in the millions (that is the matrix size), these

small perturbations in the Krylov subspace never have the chance to add up and become sufficient to destroy the Krylov method convergence.

GCR is known to be amenable to flexible preconditioning [85] but when convergence is slow, such as the case for critical mass Wilson-Dirac matrices, GCR is considered significantly less stable and less accurate than GMRES [86]. By using the hybrid SSOR-GCR preconditioner on GCR itself as the main outer solver the preconditioning was found to be just as effective as in the BiCGStab case and the over algorithm was about 50% faster due to GCR only having one preconditioned step whereas BiCGStab has two preconditioning steps. BiCGStab wasn't twice as slow since the second call to the preconditioner would see the residual continue its convergence from somewhere close to the residual of the call to the first preconditioning step.

The hybrid preconditioner is not a typical preconditioner in the sense that the preconditioner is usually a quick and easy solver that approximates the inverse matrix with favourable eigenvalue clusterings. The hybrid SSOR-GCR preconditioner in fact does most of the solving: the vast majority of the critical matrix-vector multiplications and bandwidth communication critical scalar products. The primary effect of this hybrid algorithm is to have the SSOR preconditioned GCR inner Krylov iterations do the bulk of the work with an effective restart that works around the terminal stalling that SSOR preconditioned with restarts GCR(L) had on its own for the worst case matrices, as shown in figure 5.16 .

The primary benefits of the hybrid SSOR-GCR preconditioner for the critical LQCD matrices are to induce convergence in BiCGStab and also to allow SSOR-GCR to converge in the presence of restarts. The performance has been studied on a single processor and scaling has been investigated for multi-threaded and MPI implementations on several CPU cores. With some further tuning using the ideas of communication avoiding techniques this combination of solvers should scale to massive parallel CPU and GPU clusters, and possibly FPGA implementations if development tools and cost of the required larger FPGAs allows. Scaling the Werner architecture to massively parallel systems with GPU support would be the next task in any future work on the Krylov solver method part of the project.

6.2 Graphene Physics

The first quantisation of Hamiltonian mechanics is today by far the most common approach to the calculation of observable properties for systems subject to the limitations imposed by Heisenberg's uncertainty principle. By constructing the Hamiltonian from differential operators for continuous variables, or matrices for discrete measurables such as spin, one can calculate the energy eigenvalue spectra. The energy distribution in solid state physics is typically expressed as the number of available states per energy unit, called the density of states.

From the density of states many physical properties can be deduced such as the energy band structure of a solid that determines if a solid can conduct. If there are easily accessible higher energy states available then an electric field can accelerate electrons into those higher kinetic energy states to cause a net electron flow, else if no states are available electrons remain in their original insulating ground state with no net movement. If there is a small gap between a band of higher empty conduction states and the top of a lower insulator band, a small electric field, light source or thermal energy, can cause electrons to quantum leap between bands to allow, and control, conduction for a material described as a semi-conductor.

Electron transport in solids can also be affected by the interaction of intrinsic spin with angular momentum, coupled via magnetic moments and thus altering energy levels. In two dimensional electron gas materials this Rashba effect can be used to store and transmit information signals with greater speed and efficiency than charge based electrons, and has developed into the field of spintronics. By solving for the eigenvalues and eigenvectors of Hamiltonian systems containing the Rashba spin-orbit interaction terms one can calculate physical properties such as electrical resistivity or light source absorption of spintronic systems [1].

The thermodynamics properties such as available free energy, entropy and heat capacity are also readily calculable from the energy eigenvalues of a given Hamiltonian by integrating over the density of states weighted by an appropriate particle distribution; that being the Fermi-Dirac distribution for spin-half electrons [2]. All thermodynamic properties follow from the partition function Q, which requires only knowledge of the Hamiltonian's energy eigenvalues.

$$Q = \sum_i e^{-E_i/kT} \tag{6.1}$$

All measurable quantities can be calculated from the complete knowledge of both eigenvalues and eigenvectors, and has been described by Feynman as the pinnacle law of statistical mechanics [87].

$$\langle A \rangle = \frac{1}{Q} \sum_i \langle i \,|A|\, i \rangle \, e^{-E_i/kT} \tag{6.2}$$

Particular examples of two dimensional electron and spin systems are those constructed from a layer of carbon atoms. A single flat layer of carbon atoms arranged in an hexagonal honeycomb lattice is as known as graphene, but there are endless possible structures including the globe shaped Buckminsterfullerene C60, or the fascinating Möbius strips with a twist. Basic graphene itself is an insulator, but via interactions with a mounting substrate or through impurity doping, band gaps can be created and semi-conductor properties emerge which can be studied using the characteristic modes technique of first quantisation [3].

Layers of individual carbon atoms were first theoretically analysed over sixty years ago [88] and even at that time it was noted that the band structure would contain a very interesting feature: massless Dirac quasi-particles. The top and bottom of the electronic energy bands are generally curved which leads to a Taylor expansion about those points with a second order quadratic term which can be compared to the standard classical formula for kinetic energy $E = p^2/2m$, which results in an effective mass m^* for the mobility of conduction charge carriers.

In graphene there exist corners of the band structure which form cones instead of concave maxima and minima. These cones have straight edges and in insulating graphene they met at the tips. This linear relationship results in the kinetic energy of the conduction charge carriers being directly proportional to momentum p instead of the classical quadratic.

$$E = \left(\frac{c}{300}\right) p \tag{6.3}$$

This linear relationship is similar to the linear relationship between kinetic energy and momentum for the massless photon $E = cp$ but with the velocity of propagation being three hundred times smaller. Conversely the quantum electrodynamic coupling constant α_g between these quasi-particles and the electromagnetic field is three hundred times bigger than the fine structure constant $\alpha = \frac{1}{137}$ of standard QED.

$$\alpha_g = 300 \times \alpha \approx 2 \tag{6.4}$$

With the charge carriers in graphene being massless Dirac particles, second quantisation within the framework of quantum electrodynamics becomes a natural choice. With the coupling constant being greater than one, perturbative techniques can run into the same problems as that with the strong force of QCD.

Lattice field theory simulations of graphene where first attempted [89] on cubic lattices of width about $N = 20$, using direct solvers for the lower dimensional matrices of only two spatial plus time; for increasing lattice width iterative solvers became necessary. A more natural lattice for the numerical simulation of graphene is a hexagonal lattice. Simulations have been performed on very small spatial width lattices, but up to a large temporal width of 256 in the time dimension, and have produced results very close to the analytic estimates available for these special small models [90].

The electronic properties of graphene can be studied using first quantisation techniques applied to the single particle Dirac equation within tight binding models of electronic structure [91] but the second quantisation techniques are also producing results [92]. In the second quantisation of field theory the partition function is again the key formula from which all properties of the quantum system may be calculated, but is now expressed as variations over all possible fields rather than the simpler sum over exponential functions of eigenvalues. Since the quantum states are implicitly embedded in the variation over all possibilities the expectation values of measurable observables can likewise be extracted using correlation functions built from derivatives of the partition function [93].

The second quantisation approach of quantum field theory, and its lattice variant, are the more detailed alternatives to the averaging over all possible single particle eigenstates of first quantisation. The infinite dimensional analysis of all possible field variables, both matter and force, naturally require a far greater computational cost even with Monte Carlo sampling to make the situation tractable. Matrix inversion to generate the Green's functions is one of the major costs when currently dealing with fermions such as electrons and other Dirac spin half fields.

6.3 Further Work

Build a GPU stack and add CUDA bindings to the Werner library. This will provide a solver platform on which to perform the LQED calculations useful to investigate nanostructures such as graphene and other possible configurations for new quantum physics based materials. One such recent material is graphyne which contains non-hexagonal structures that cause variations in band structure and associated properties [94]. The accurate modeling of band gap effects is crucial for the design for next generation electronic transistors and then future spintronic devices for use in trying to build possible quantum computers.

The hybrid SSOR-GCR preconditioner presented here is a new combination of existing algorithms that facilitates the solution of critical lattice quantum field matrices. The composition is highly configurable and is easily tunable at runtime by monitoring the convergence rate: hence can be dynamically adapted to the matrix that requires solution. The component algorithms are amenable to communication avoiding sparse matrix techniques that result in only $O(logN)$ communication growth, and the combination with SSOR-GCR hybrid preconditioner presented has been kept within that framework of communication reduction.

The issue of critical slowing down has been addressed to a degree by the SSOR-GCR preconditioner, although the blowout in time caused by low mass fermions has been suppressed rather than removed. This allows a wider range of matrices to be solved before needing more advanced techniques. The divide and conquer approach of domain decomposition [95] combined with eigenvalue deflation [96] looks the most promising approach to dealing with the ill-conditioned nature of low mass fermions. These approaches need to be considered in greater detail and incorporated into Werner for use on the very worst case matrices. Red-black (even-odd) preconditioning and the classical Schwarz alternating procedure have also been used to increase the performance of solver algorithms [97] and can be added on top of the main solver for additional performance.

Another area worth investigating would be in relation to the domain decomposition technique being used as the preconditioner. The fast monotonic convergence induced on BiCGStab was due to the guaranteed residual minimisation returned by the SSOR-GCR hybrid preconditioner. With the domain decomposition naturally distributable across a massively parallel cluster, it should be worth investigating whether small domains can be used as an effective preconditioner to a global BiCGStab algorithm. If the domains are small enough direct solvers can be used, such as the parallel PARDISO solver [98] for large non-symmetric matrices. Using direct solvers has the benefit that the ill-conditioned nature of the low mass fermion matrices would not affect the performance

of direct solvers, so direct solvers would not suffer from the critical slowing down in the preconditioning stage, but could still induce the smooth convergence of the fast outer BiCGStab method.

The hybrid direct-iterative approach to scaling into exascale computational performance is being researched to provide implementations of sub-domain based solvers over large multicore systems [99]. The SSOR-GCR hybrid preconditioner presented here is not exactly in the spirit of using a cheap preconditioner, but direct solvers on small sub-domains not affected by ill-conditioning could be a very viable option given the desirable monotonic convergence of BiCGStab is induced by returning a guaranteed improved preconditioned result. This would be a step in the direction of have a localised preconditioner improving the performance of a global overall solver.

The global nature of matrix inversion is fundamental to the large time complexity of matrix inversion. This mirrors the quantum reality that bothered Einstein so much: spooky action at a distance and the EPR experiment [100]. The Wilson-Dirac matrix models the fact that the quantum wavefunction at any point of spacetime is correlated to the wavefunction at any other point of spacetime irrespective of the distance and time between them; even when beyond the causal connection suggested by the limit of the speed of light. To make the matrix solver completely local with unconnected sub-domains would would break this relationship to quantum reality and hence limit the accuracy of the algorithm. Using local sub-domain solvers to generate approximate solutions as preconditioners in order to accelerate an outer global solver such as BiCGStab is an appealing balance between speed and physical accuracy.

To actually perform physics calculations requires more than just the matrix solvers but rather an entire molecular dynamics and gauge field software package such as Chroma [101] or MILC [102]. Chroma is a large software package used for production runs on large UNIX clusters and is written primarily in C++. It took several weeks to gain sufficient understanding in order to build the code on a Windows environment, but then the complete visual debugging environment of Visual Studio provided excellent means to trace through the details of the code. The code was almost all standard C++ that was straight forward to port between Linux and Windows but a few minor issues absorbed the time due to the shear combined size of Chroma and its dependency libraries.

Chroma is a large fully featured library but one drawback is an extensive use of the C++ template mechanism which made debug tracing difficult, and quite opaque in places, without detailed documentation of the software design structure. MILC is an alternative C coded library that should provide better access with respect to building executable programs for smaller clusters with bindings into Werner and GPU support.

The testing of Werner incorporated into real lattice field code is an important next step to validate the SSOR-GCR hybrid algorithm over a wider set of gauge configurations and the corresponding Wilson-Dirac matrices.

Feynman stated he was convinced that fermion operators on a lattice, and the bosonic wave interactions that they cause, could be modelled by classical universal computers to form a general quantum mechanical simulator [103]. He remarked on not knowing how to simulate the fermions but had previously suggested that the Dirac equation can be completely derived from a cellular automaton implementation of a path integral variant in a famous "exercise for the student" in the book on path integrals [104]. The three dimensional Dirac equation was finally derived from this Feynman checkerboard idea [105] as a Poisson process in imaginary time for relativistic electrons.

The evolution of this Poisson Dirac model would not require matrix solvers, but to investigate real physical systems would require very large lattices with combinatoric evaluations that become geometrically large very rapidly. Even in one dimension the Feynman checkerboard exhibits the phenomenon of zitterbewegung first spotted by Schrödinger as evidence that something is wrong with the Dirac equation since the motion results in a faster than light average speed for the electron [106]. With the discovery of Dirac quasi particles in graphene the possibility of observing this phenomena of zitterbewegung has become a topic of experimental interest [107] instead of just being a theoretical paradox [108].

So how did the particle move from A to B? The energy-momenta quantum of excitation in the field propagates as a wave through the field, interacting along the way with an infinite sea of virtual particles. Some of those virtual particles will be identical and indistinguishable from the particle originally located at event A, rendering any question about a specific particle unanswerable: including the question itself about "a" particular particle in the first place! The original particle may or may not have actually travelled to B, but rather what can be certain, is the calculation of the probability with which the energy-momenta will arrive.

Appendix A

Feynman Path Lattice QFT

The Wilson-Dirac matrix used in LQCD simulations is derived from the first principles of the Feynman path integral formulation of quantum mechanics, then followed by its generalisation to the second quantization of quantum fields. The material presented here is a summary of information found in the books by Feynman and Hibbs[104], Creutz[8], Rothe[9], and Montvay and Munster[10].

A.1 The Feynman Path Integral

The transition amplitude for a particle to transfer from event $E(x,t)$ with quantum state $|x\rangle$ to event $E'(x',t')$ with state $|x'\rangle$ is given by

$$\langle x'|\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H(t'-t)}|x\rangle \tag{A.1}$$

The exponentiation of the quantum Hamiltonian operator provides a probability conserving unitary operator $U\left(x',t';x,t\right) = \mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H(t'-t)}$ that evolves the operand $|x\rangle$ over time. When projected into the state $|x'\rangle$ at the later time t' the probability amplitude of the transition results.

The key idea behind the Feynman Path Integral is that in moving from $E(x,t)$ to $E'(x',t')$ the particle takes all possible routes: a generalisation of the Young's 2-slit wave interference experiment [104]. The transition from $E(x,t)$ to $E'(x',t')$ can be viewed as taking place via a half-way point $E_1(x_1,t_1)$ such that

$$\langle x'|U\left(x',t';x,t\right)|x\rangle = \langle x'|U\left(x',t';x_1,t_1\right)|x_1\rangle\langle x_1|U\left(x_1,t_1;x,t\right)|x\rangle \tag{A.2}$$

But not just a single half-way point but all possible half-way points such that those half-way points form a complete set $\int dx_1|x_1\rangle\langle x_1| = 1$

$$\langle x'|U\left(x',t';x,t\right)|x\rangle = \int dx_1\langle x'|U\left(x',t';x_1,t_1\right)|x_1\rangle\langle x_1|U\left(x_1,t_1;x,t\right)|x\rangle \tag{A.3}$$

This process can then be recursively repeated for half-way points between half-way points until the set of all possible paths has been constructed. The unitary evolution operator $U\left(x',t';x,t\right)$ can also be divided into time-steps $\epsilon = (t'-t)/N$ so

$$\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H(t'-t)} = \lim_{N\to\infty}\left[\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H\left(\frac{t'-t}{N}\right)}\right]^N \tag{A.4}$$

The resulting transition amplitude evaluated over all possible paths is given by

$$\langle x'|\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H(t'-t)}|x\rangle = \lim_{N\to\infty}\int dx_1\cdots dx_{N-1}\langle x'|\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H\epsilon}|x_1\rangle\cdots\langle x_{N-1}|\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H\epsilon}|x\rangle \tag{A.5}$$

In the continuum case the number of mid-points is increased without limit, whilst for lattice implementation N will be limited to a finite number, as constrained by memory and runtime limits. By assuming the Hamiltonian is formed from a quadratic kinetic energy term and a time independent potential, $H = \frac{p^2}{2m} + V(x)$, using the Baker-Campbell-Haussdorff formula, and neglecting terms of order ϵ^2, the expression for the time evolution between locations x_i and x_{i+1} can be written

$$\langle x_{i+1}|\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H(t'_{i+1}-t_i)}|x_i\rangle = \langle x_{i+1}|\mathrm{e}^{-\frac{\mathrm{i}\epsilon p^2}{2m\hbar}}\mathrm{e}^{-\frac{\mathrm{i}\epsilon}{\hbar}V(x)}|x_i\rangle \tag{A.6}$$

Inserting a complete set of momentum basis states allows the kinetic energy term to be integrated (using analytic continuation into the complex plane):

$$\langle x_{i+1}|\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H(t'_{i+1}-t_i)}|x_i\rangle = \frac{1}{2\pi}\int dp\langle x_{i+1}|\mathrm{e}^{-\frac{\mathrm{i}\epsilon p^2}{2m\hbar}}|p\rangle\langle p|\mathrm{e}^{-\frac{\mathrm{i}\epsilon}{\hbar}V(x)}|x_i\rangle \tag{A.7}$$

$$= \frac{1}{2\pi}\int dp\mathrm{e}^{-\frac{\mathrm{i}\epsilon p^2}{2m\hbar}}\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}p(x_{i+1}-x_i)}\mathrm{e}^{-\frac{\mathrm{i}\epsilon}{\hbar}V(x_i)} \tag{A.8}$$

$$= \sqrt{\frac{m}{2\pi\mathrm{i}\hbar\epsilon}}\mathrm{e}^{-\frac{\mathrm{i}\epsilon}{\hbar}[\frac{m}{2}(\frac{x_{i+1}-x_i}{\epsilon})^2-V(x_i)]} \tag{A.9}$$

Placing all N expressions for $\langle x_{i+1}|\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H(t'_{i+1}-t_i)}|x_i\rangle$ back into the formula for transition amplitude A.5 gives the Feynman Path Integral for the propagation of a particle from x to x' over the time interval $t'-t$. The integration constant $\sqrt{\frac{m}{2\pi\mathrm{i}\hbar\epsilon}}$ is also dropped since observable values are calculated from the ratio of two path integrals and hence the constant is cancelled out in actual calculations.

$$\langle x'|\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H(t'-t)}|x\rangle = \lim_{\epsilon\to 0}\int dx_1\cdots\int dx_{N-1}\mathrm{e}^{\frac{\mathrm{i}}{\hbar}\lim\limits_{\epsilon\to 0}\sum\limits_{k=1}^{N}\epsilon\left[\frac{m}{2}\left(\frac{x_{i+1}-x_i}{\epsilon}\right)^2-V(x_i)\right]} \tag{A.10}$$

In lattice calculations over N sized grid intervals ϵ is the lattice spacing a. The above formula is thus already the lattice path interval without taking the limit $\epsilon\to 0$, whilst the continuum version takes the limit. The Feynman Path Integral identifies the sum in the exponent as the action integral of classical mechanics,

$$S[x] = \frac{\mathrm{i}}{\hbar}\sum_{k=1}^{N}\epsilon\left[\frac{m}{2}\left(\frac{x_{i+1-x_i}}{\epsilon}\right)^2 - V(x_i)\right] \tag{A.11}$$

whilst the N-dimensional integral is abbreviated to

$$\int\mathcal{D}x = \int dx_1\cdots\int dx_{N-1} \tag{A.12}$$

Either taking the continuum limit $\epsilon\to 0$, or keeping a finite lattice $\epsilon = a$, the Feynman Path Integral for transition amplitudes is given its canonical form by

$$\langle x'|\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}H(t'-t)}|x\rangle = \int\mathcal{D}\ \mathrm{e}^{-\frac{\mathrm{i}}{\hbar}S[x]} \tag{A.13}$$

A.2 Lattice Path Integration

The action terms of the path integral in the continuum limit are functionals: numbers that depend on the path taken between x and x'. In classical physics the Lagrangian is the function integrated to evaluate the classical action, and the path taken by classical particles is specifically the path that produces an extremum in the action (the principle of least action). Feynman's path integral reproduces the classical path in the limit $\hbar \to 0$ whilst quantum mechanics can be seen to introduce fluctuations around the classical path, consistent with the principles behind Heisenberg uncertainty. In the classical limit the quantum wavefunctions away from the classical extremum have large oscillating phases which cancel to leave little probability in those paths away from the classical path, whilst those paths very close to the extremum have little phase difference and as such constructively interfere to make the classical path overwhelming most probable.

The large oscillations in the quantum phase make evaluating the path integral highly numerically unstable due to all the subtractions involved and those small differences contributing large percentage errors in any simulation. The procedure used is to perform a Wick rotation $t \to -it$ which turns the oscillations in numerically more stable decaying exponentials. For the relativistic path integral required for QCD this also turns the indefinite Minkowski metric into a positive definite Euclidean metric. Planck's constant and the speed of light are also normally rescaled to 1.

The lattice action after a Wick rotation also turns the $-V(x)$ term in the Lagrangian into $+V(x)$; note this will also turn $-m$ in the Dirac equation for QCD into $+m$ for LQCD. The lattice action over a specific path x_c on a lattice with grid spacing a is given by

$$S_{LAT}[x_c] = \sum_{i=0}^{N-1} \left[\frac{m}{2a} \left(x_{i+1} - x_i \right)^2 + aV(x_i) \right] \tag{A.14}$$

The specific path for a given functional evaluation of $S_{LAT}[x_c]$ is known as a configuration

$$x_c = \{x_0, x_1, \cdots, x_N\} \; \{x_i\}_{i=0 \to N} \tag{A.15}$$

When the path integral is evaluated over a discrete configuration on the lattice, the path integral becomes a regular multi-dimensional integral where each of the individual $\{x_j\}$ can be integrated separately.

$$\langle x'|\mathrm{e}^{-H\Delta t}|x\rangle = \int \mathcal{D}x \ \mathrm{e}^{-S_{LAT}[x_c]} \tag{A.16}$$

$$= \int_{x_1=-\infty}^{x_1=+\infty} dx_1 \cdots \int_{x_{N-1}=-\infty}^{x_{N-1}=+\infty} dx_{N-1} \ \mathrm{e}^{\sum_{i=0}^{N-1}\left[\frac{m}{2a}(x_{i+1}-x_i)^2+aV(x_i)\right]} \tag{A.17}$$

Thus quantum mechanics has been reduced to a problem in numerical integration!

The Euclidean path integral allows us to evaluate observable expectation values. Consider an operator $O\hat{(}x)$ that is diagonal in the position state basis and insert this operator into the path integral to compute its expectation value.

$$\langle \mathcal{O}(x)\rangle = \langle x|\hat{\mathcal{O}}|x\rangle = \frac{1}{Z}\int \mathcal{D}x \ \mathcal{O} \ \mathrm{e}^{-S[x_c]} \tag{A.18}$$

This expression allows observable values to be calculated and also provides the link between the path integral formalism and canonical operator quantum mechanics. The partition function Z normalizes the result similar to the usual normalization procedure in quantum wave mechanics.

$$Z = \int \mathcal{D}x \ \mathrm{e}^{-S[x_c]} \tag{A.19}$$

In the limit of large time $\Delta t \to \infty$ only the quantum ground state $|0\rangle$ (the vacuum) contributes to the partition function $Z \sim e^{-E_0\Delta t}$ and this describes the vacuum zero-point value of the operator. To extract higher eigenvalues 2-point correlation functions can be used

$$\langle \mathcal{O}(0)\mathcal{O}(x)\rangle = \frac{1}{Z}\int \mathcal{D}x \ \mathcal{O}(x(0))\mathcal{O}(x(t)) \ \mathrm{e}^{-S[x]} \tag{A.20}$$

$$\lim_{t\to\infty}\langle \mathcal{O}(0)\mathcal{O}(x)\rangle - |\langle \mathcal{O}\rangle|^2 = |\langle 0|(x)|1\rangle|^2 e^{-(E_1-E_0)} \tag{A.21}$$

Here $|1\rangle$ is the first excited state of the quantum system with an energy E_1. This provides a method to calculate the energy of the first excited state above the vacuum. In quantum field theory this is the lightest particle caused by excitations of the field.

A.3 The Wilson-Dirac Matrix

Quantum electrodynamics of charged particles and the quantum chromodyamics of hadronic matter uses special relativistic kinematics and for spin-$\frac{1}{2}$ electrons and quarks this means the Dirac equation

$$(\mathrm{i}\gamma^{\mu}\partial_{\mu} - m)\,\psi(x) = 0 \tag{A.22}$$

where $\{\gamma^{\mu}\}$ are dirac matrices obeying the Clifford algebra $\{\gamma^{\mu}, \gamma^{\nu}\} = 2g^{\mu\nu}$ and the fermion field $\psi(x)$ is a $4-$component spinor field. In the following description the spinor indices will be suppressed, but run $\alpha = 0 \to 3$ otherwise. The equation of motion for the fermion field ψ and $\bar{\psi} \equiv \psi^{\dagger}\gamma^{0}$ follows from the action

$$S_F\left[\psi, \bar{\psi}\right] = \int d^4x\ \bar{\psi}\,(\mathrm{i}\gamma^{\mu}\partial_{\mu} - m)\,\psi(x) \tag{A.23}$$

Applying a Wick rotation to get the Euclidean formulation for use on a lattice

$$t \to -\mathrm{i}x_4,\ \gamma^0 \to \gamma_4,\ \gamma^i \to \mathrm{i}\gamma_i \tag{A.24}$$

$$S_F \to \int d^4x\bar{\psi}\,(\gamma_{\mu}\partial_{\mu} + m)\,\psi(x) \tag{A.25}$$

The gamma matrices now satisfy the algebra $\{\gamma_{\mu}, \gamma_{\nu}\} = 2\delta_{\mu\nu}$ with $\mu = 1 \to 4$ and now $\bar{\psi} = \psi^{\dagger}\gamma_4$. A convenient representation for the gamma matrices using the usual Pauli spin matrices $\{\sigma_i\}$ is given by

$$\gamma_i = \begin{pmatrix} 0 & \sigma_i \\ \sigma_i & 0 \end{pmatrix} \qquad i = 1, 2, 3 \tag{A.26}$$

$$\gamma_4 = \begin{pmatrix} \mathbb{I} & 0 \\ 0 & -\mathbb{I} \end{pmatrix} \tag{A.27}$$

$$\gamma_5 = \begin{pmatrix} 0 & -\mathrm{i} \\ \mathrm{i} & 0 \end{pmatrix} = \gamma_1\gamma_2\gamma_3\gamma_4 = \gamma_5^{\dagger} \tag{A.28}$$

Each lattice site n has a spinor field element $\psi(n)$ and the path integration is now over a configuration of field spinors $\{\psi(1), \psi(1), \cdots, \psi(N-1)\}$ instead of the position configuration $\{x_i\}$ before. The path integral over a configuration of coordinates quantizes those coordinates and maps them to operators providing a first quantization procedure equivalent to single particle quantum mechanics. By replacing position coordinates in the path integral with field values, those fields are mapped into operators providing a second quantization where the fields are now operators obeying Heisenberg uncertainty relationships.

The next step, before constructing the Wilson-Dirac matrix from discretized differentials, is to rescale m and ψ by a into dimensionless lattice variables.

$$m \to \frac{1}{a}\hat{m}, \qquad \psi(x) \to \frac{1}{a^{\frac{3}{2}}}\hat{\psi}(n), \qquad \partial_\mu \psi(x) \to \frac{1}{a^{\frac{3}{2}}}\hat{\partial}_\mu \hat{\psi}(n) \tag{A.29}$$

The discrete lattice derivative $\hat{\partial}_\mu$ is given by

$$\hat{\partial}_\mu \hat{\psi}(n) = \frac{1}{2}\left(\hat{\psi}_{n+\hat{\mu}} - \hat{\psi}_{n-\hat{\mu}}\right) \tag{A.30}$$

This difference operator gives the μ direction differential of $\hat{\psi}$ at lattice site n as the difference between $\hat{\psi}$ in the $+\mu$ direction from n minus the value of $\hat{\psi}$ in the $-\mu$ direction from n. This discrete difference operator is anti-hermitian $(\hat{\partial}_\mu \hat{\psi}(n))^\dagger = -\hat{\partial}_\mu \hat{\psi}(n)$, respecting the anti-hermitian nature of the continuous Dirac equation, and this is the root cause of the failure of the convenient conjugate gradient method and hence the need for more general non-symmetric matrix solvers. The basic discrete version of the Dirac matrix is therefore

$$S_F = \sum_{n,m} \bar{\hat{\psi}} \left[\sum_\mu \frac{1}{2}\gamma_\mu \left(\delta_{m,n+\hat{\mu}} - \delta_{m,n-\hat{\mu}}\right) + \hat{m}\delta_{mn} \right] \hat{\psi} \tag{A.31}$$

This matrix has the mass term along the diagonal and connections to eight nearest lattice neighbour sites. In QCD each site has a 4-component spinor field and each spinor component carries red-green-blue color charge triplet. This gives a 4-spin $\times$ 3-color $=$ 12-component matrix at each lattice site forming the field configuration over which the path integral needs evaluating, hence the matrix formed to model QCD dynamics has 8 $\times$ 12 + (1 mass on the diagonal) = 97 non-zeroes per row.

This simplest discretized lattice version of the Dirac Lagrangian is known as "naive" since it leads to unwanted extra fermions being simulated. When the propagators are generated under a Fourier transform to momentum space the 16 corners of the first Brillouin zone each cause a massive fermion to be propagated. This is known as "doubling" since each of the 8 directions $\pm\hat{\mu}$ models two fermions during simulation. Using left or right discrete lattice differences removes this doubling at the expense of destroying the anti-hermitian symmetry of the original Dirac operator.

Wilson discovered an alternative option was to introduce a discrete Laplacian term proportional to the lattice spacing a, hence disappears as $a \to 0$, but has the effect of suppressing 15 of the 16 doublers. Those 15 extra doublers are suppressed dynamically by giving them infinite mass at their corners of the Brillouin zone, so they never propagate and hence contribute a fixed amount that is normalised away without contributing to dynamic observables.

$$\Box\psi(x) = \sum_{\mu=1}^{4} \partial_\mu \partial_\mu \psi(x) \to \frac{1}{a^2}\hat{\Box}\hat{\psi}(n) \tag{A.32}$$

$$\hat{\Box}\hat{\psi}(n) = \sum_{\mu} \left[\hat{\psi}_{n+\hat{\mu}} - 2\hat{\mu} + \hat{\psi}_{n-\hat{\mu}}\right] \tag{A.33}$$

This extra term is summed over all lattice sites, and is known as the Wilson term S_W

$$S_W\left[\bar{\hat{\psi}}, \hat{\psi}\right] = -\frac{r}{2}\sum_{n} \hat{\psi}\hat{\Box}\hat{\psi} \tag{A.34}$$

The parameter r is the Wilson parameter $0 < r \leq 1$ and is typically taken to be $r = 1$. The total action for the Dirac fermion with the additional Wilson term is then

$$S_F^{(W)} = S_F\left[\bar{\hat{\psi}}, \hat{\psi}\right] + S_W\left[\bar{\hat{\psi}}, \hat{\psi}\right] \tag{A.35}$$

$$S_F^{(W)} = \sum_{n,m} \bar{\hat{\psi}} \left[\sum_{\mu=1}^{4} \frac{\gamma_\mu}{2}\left(\delta_{m,n+\hat{\mu}} - \delta_{m,n-\hat{\mu}}\right) + \hat{m}\delta_{m,n} - \frac{r}{2}\sum_{\mu=1}^{4}\left(\delta_{m,n+\hat{\mu}} - 2\delta_{m,n} + \delta_{m,n-\hat{\mu}}\right)\right]\hat{\psi} \tag{A.36}$$

$$S_F^{(W)} = \sum_{n,m} \bar{\hat{\psi}} \left[(\hat{m} + 4r)\, \delta_{nm} - \frac{r}{2} \sum_{\mu=1}^{4} \left([r - \gamma_\mu]\, \delta_{m,n+\hat{\mu}} + [r + \gamma_\mu]\, \delta_{m,n-\hat{\mu}} \right) \right] \hat{\psi} \qquad \text{(A.37)}$$

The factors $P_{\pm\mu} = [1 \pm \gamma_\mu]$ are described as projectors and they have the effect of causing half the states to propagate only in the positive direction whilst the other half only propagate in the negative direction. Propagation in both directions is suppressed since $(1 - \gamma_\mu)]\,(1 + \gamma_\mu) \equiv 0$. This technique is known as Wilson's propagator method and is an alternative insight into how doublers are removed. The original naive fermions have $P_{\pm\mu} = \pm\frac{1}{2}\gamma_\mu$ which preserves chiral symmetry but with all the doublers.

The final step before reaching the actual matrix used in simulations, and the actual matrix required for inversion, is to introduce the hopping parameter κ (and use $r = 1$)

$$S_F^{(W)} = \sum_{n,m} (\hat{m} + 4)\, \bar{\hat{\psi}} \left[\delta_{nm} - \frac{1}{2\,(\hat{m} + 4)} \sum_{\mu=1}^{4} \left([r - \gamma_\mu]\, \delta_{m,n+\hat{\mu}} + [r + \gamma_\mu]\, \delta_{m,n-\hat{\mu}} \right) \right] \hat{\psi} \qquad \text{(A.38)}$$

$$S_F^{(W)} = \sum_{n,m} \left(\frac{\bar{\hat{\psi}}}{\sqrt{2\kappa}} \right) K_{nm} \left(\frac{\hat{\psi}}{\sqrt{2\kappa}} \right) \qquad \text{(A.39)}$$

where

$$K_{nm} = \delta_{nm} - \kappa M_{nm} \qquad \text{(A.40)}$$

$$\kappa = \frac{1}{2\,(\hat{m} + 4)} = \frac{1}{2\,(ma + 4)} \qquad \text{(A.41)}$$

The fields are rescaled $\hat{\psi} \to \hat{\psi}/\sqrt{2\kappa}$ and the matrix M_{nm} holds all the off-diagonal terms. In the limit $a \to 0$ the hopping parameter tends to a critical value $\kappa \to \frac{1}{8}$ where the matrices become extremely ill-conditioned and simulations suffer from critical slowing down. The hopping parameter corresponds to the projected propagators moving quarks from site-to-site, and is useful in perturbative approximations expanded around the hopping parameter $\kappa < 1$.

There are several regularly used alternatives to the Wilson-Dirac fermions, and each has particular properties suited to specific investigations. Staggered Kogut-Susskind fermions are a common alternative that distributes the spinor components over interleaved lattice sites to get exactly 4 doublers and is better for studying spontaneous chiral symmetry breaking. This restriction to four flavors of fermion is less flexible than Wilson fermions which can be studied with an arbitrary number of flavors. Ginsparg-Wilson fermions have made a comeback recently that keep exact global chirality at the expense of being slightly delocalised, but are numerically far more costly that the original Wilson fermions.

A.4 Gauge Fields

Quantum observables are invariant under global phase shifts

$$\psi' \to \mathrm{e}^{+\mathrm{i}g\Lambda}\ \psi \tag{A.42}$$

$$\psi'^{*}\ \psi' = \psi^{*}\mathrm{e}^{+\mathrm{i}g\Lambda}\mathrm{e}^{-\mathrm{i}g\Lambda}\psi = \psi^{*}\ \psi \tag{A.43}$$

In the fermionic action $S_F\left[\psi,\bar{\psi}\right] = \int d^4x\ \bar{\psi}\left(\mathrm{i}\gamma^{\mu}\partial_{\mu} - m\right)\psi(x)$ the mass term is also invariant when the phase shift is promoted to a local phase shift with Λ no longer globally the same everywhere but varies from point to point as a function of x, $\Lambda \to \Lambda(x)$. The kinetic term involving the derivative is not invariant under a local phase shift, but instead picks up an extra contribution.

$$\partial_{\mu}\psi' = \partial_{\mu}\left(\mathrm{e}^{-\mathrm{i}g\Lambda(x)}\psi(x)\right) = \mathrm{e}^{-\mathrm{i}g\Lambda(x)}\partial_{\mu}\psi - \mathrm{i}g\partial_{\mu}\Lambda(x)\mathrm{e}^{-\mathrm{i}g\Lambda(x)}\psi \tag{A.44}$$

$$\partial_{\mu}\psi' = \left[\partial_{\mu} - \mathrm{i}g\partial_{\mu}\Lambda(x)\right]\psi(x) \tag{A.45}$$

The extra $-\mathrm{i}g\partial_{\mu}\Lambda(x)$ breaks the symmetry and observables are no longer invariant. Invariance can be restored by replacing the ordinary partial derivative with a covariant derivative $D_{\mu} = \partial_{\mu} + \mathrm{i}gA_{\mu}$ such that the vector field $A_{\mu}(x)$ introduced also transforms, according to $A'_{\mu} \to A_{\mu} + \partial_{\mu}\Lambda$.

$$\bar{\psi}'(x)D'_\mu\psi'(x) = \bar{\psi}(x)\mathrm{e}^{+\mathrm{i}g\Lambda(x)}\left[\mathrm{e}^{-\mathrm{i}g\Lambda(x)} + \mathrm{i}g\left(A_\mu + \partial_\mu\Lambda(x)\right)\right]\mathrm{e}^{-\mathrm{i}g\Lambda(x)}\psi(x) \tag{A.46}$$

$$= \bar{\psi}(x)\mathrm{e}^{+\mathrm{i}g\Lambda(x)}\mathrm{e}^{-\mathrm{i}g\Lambda(x)}\left(\partial_\mu + \mathrm{i}gA_\mu + \mathrm{i}g\partial_\mu\Lambda(x) - \mathrm{i}g\partial_\mu\Lambda(x)\right)\psi(x) \tag{A.47}$$

$$= \bar{\psi}(x)\left(\partial_\mu + \mathrm{i}gA_\mu\right)\psi(x) \tag{A.48}$$

$$\bar{\psi}'(x)D'_\mu\psi'(x) = \bar{\psi}(x)D_\mu\psi(x) \tag{A.49}$$

The invariance of observables evaluated using the Dirac equation is preserved if the local gauge change $\mathrm{e}^{+\mathrm{i}g\Lambda(x)}$ is accompanied by a vector field $A_\mu + \partial_\mu\Lambda(x)$. This relationship manifests itself in classical electrodynamics as the principle of minimal substitution, which would again be recovered from Feynman path QED in the limit $\hbar \to 0$. Locality of gauge invariance in quantum physics thus requires the presence of a vector field whose variations counter balances the spinor phase variations when evaluating sums over all possible spinor fields and their phase configuration.

The gauge vector fields also require Feynman path evaluation over all their possible field configurations: this process of second quantization resulting in spin-1 vector bosons. For QED these vector field excitations result in long range Coulomb photons. For QCD the field excitations result in color charge carrying gluons which interact with each other (unlike photons which are chargeless and obey linear superposition) and thus lead to the non-linear confinement of themselves and the colored quarks. The force mediating gauge vector bosons could be seen as emerging from the effects of relativity being imposed upon Feynman path quantization. The vector bosons of Weak interactions are also based upon gauge symmetry but the situation is complicated by the presence of non-zero vacuum expectation value of the field, whose spontaneous symmetry breaking causes fermions to have mass via the Higgs mechanism.

When a charged fermion is evaluated over a path in spacetime its wavefunction acquires a factor of U from interactions with a gauge field $A(x)$, known as a Schwinger line integral

$$\psi'(x) \to \psi\mathrm{e}^{\mathrm{i}g\int A_\mu dx_\mu} \equiv \psi U \tag{A.50}$$

In the case of a particle at rest $dx_\mu = dt$ so $U = \mathrm{e}^{\mathrm{i}gAt}$: the particle has acquired a time oscillation ($E = \hbar\omega$) at a rate proportional to its charge g and scalar potential A_0.

The curl of A provides a field tensor $F_{\mu\nu}$ and a conserved current j_μ.

$$F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu - ig\left[A_\mu, A_\nu\right] \tag{A.51}$$

$$j_\nu = D_\mu F_{\mu\nu} \tag{A.52}$$

The Lagrangian for the vector field is

$$\mathcal{L} = \frac{1}{4} F^{\mu\nu} F_{\mu\nu} \tag{A.53}$$

In the case of QED this is the Maxwell Lagrangian and reproduces Maxwell's electromagnetic field equations under classical variation to find the configuration of least action. The coupling constant g is identified as the unit electric charge, and the commutator in A.51 is zero so the gauge field excitations have no interaction between themselves. For QCD the vector fields have a matrix representation so the gauge field excitations do interact due to the non-commuting matrices representing those gluon fields.

This Lagrangian can be rewritten in terms of Schwinger line integrals around closed loops to provide a convenient form for lattice simulation, and these are know as Wilson loops or plaquettes. The sum of all plaquettes around a lattice is then the Wilson gauge action $S_G[U]$ used in simulations. Whilst String theory generalizes Feynman path integrals over point particle configurations to path integrals over string configurations, quantum loop gravity generalizes Wilson loops as an alternative route to a quantum field theory of gravitational interactions.

The Wilson action of a gauge field is given by multiplying the gauge links U around a closed loop, typically formed from the smallest square in the lattice with sides of length a. A single Wilson loop plaquette associated with lattice site n is given by the expression

$$S_P = \beta\left[1 - \frac{1}{N}\mathbb{R}\ Tr\left(U(n)\ U(n+\hat{\mu})\ U(n+\hat{\mu}+\hat{\nu})\ U(n+\hat{\nu})\right)\right] \tag{A.54}$$

$$S_P = \beta\left[1 - \frac{1}{2N}\left(U_P - U_P^\dagger\right)\right] \tag{A.55}$$

The constant $\beta = 2N/g^2$ chosen so that the lattice action matches the continuum action in the limit $a \to 0$; N being the dimension of the gauge, that is $N = 1$ for the $U(1)$ symmetry group of QED, $SU(2)$ for the Weak force, or $N = 3$ for the $SU(3)$ for QCD. The constant unit term 1 in the plaquette action causes no observable effects but is used by convention so that $S_P \to 0$ as $U \to 1$: no phase change around a Wilson loop results in no action.

Assuming A_μ is slowing varying so that $A_\mu(n) \approx$ constant in the neighbourhood of lattice site n, then a Taylor expansion of $S_P \approx \mathrm{e}^{-igA_\mu dx_\mu}$ results in

$$S_P = \frac{\beta g^2}{2N} \int \frac{1}{4} F^{\mu\nu} F_{\mu\nu} d^4x \; + \; O(a^6) \tag{A.56}$$

which correctly reproduces the required field action A.53 in the limit $a \to 0$.

The total action of the gauge field over the entire lattice is then given by the sum of individual site plaquettes, and is known as the Wilson gauge action.

$$S_G^{(W)} = \sum_P \left[1 - \frac{1}{2N} \left(U_P - U_P^\dagger \right) \right] \tag{A.57}$$

A.5 Lattice Simulations

The Wilson-Dirac action simulating freely propagating fermions (electrons, quarks, $\cdots$) is given by A.38

$$S_F^{(W)} = \sum_{n,m} (\hat{m} + 4)\, \bar{\hat{\psi}} \left[\delta_{nm} - \frac{1}{2\,(\hat{m} + 4)} \sum_{\mu=1}^{4} \left([1 - \gamma_\mu]\, \delta_{m,n+\hat{\mu}} + [1 + \gamma_\mu]\, \delta_{m,n-\hat{\mu}} \right) \right] \hat{\psi}$$

The Wilson-Gauge action to simulate gauge fields (photons, gluons, $\cdots$) is given by A.57

$$S_G^{(W)} = \sum_P \left[1 - \frac{1}{2N} \left(U_P - U_P^\dagger \right) \right]$$

Simulating pure gauge fields using A.57 alone is the least computationally intensive task and the focus of the earliest efforts. Incorporating fully dynamic and fully interacting fermions has only recently become feasible with improved algorithms and petaflop supercomputers. Previously the approach to fermions was to make the quenched approximation that averages out dynamic fermion motion and achieved success to within about 15%. With current technology fully dynamic fermions can be simulated for 2 or 3 dynamical quarks to accurately predict rest masses for many subatomic particles, although the next area of interest with the next generation of exascale supercomputing, is to simulate multi-quark atomic nuclei.

Given the actions for free fermions and pure gauge fields the combined dynamics is given by the total action of the individual actions summed together, but with interaction between the two following the lines of minimal substitution used in classical mechanics (to which quantum field theory should tend in the appropriate limit). For small enough lattice spacing the Schwinger line integral tends to a lattice covariant derivative:

$$U_\mu(n) \;=\; \mathrm{e}^{\mathrm{i}g\int A_\mu(n)\times a} \;\approx\; \delta_{n,n+\hat{\mu}} + \mathrm{i}gA_\mu(n) \tag{A.58}$$

This parallel transport corresponds to the covariant minimal substitution given by $\partial_\mu \to D_\mu = \partial_\mu + \mathrm{i}g\int A_\mu$ so that on the lattice the terms in the Wilson-Dirac matrix A.38, with $\delta_{n,n\pm\hat{\mu}}$ originating from the derivatives, should be minimally substituted on the lattice via

$$\delta_{n,n\pm\hat{\mu}} \;\to\; U_\mu(n) = \delta_{n,n+\hat{\mu}} + \mathrm{i}gA_\mu(n) \tag{A.59}$$

That is the projectors in the Wilson-Dirac matrix,

$$[1-\gamma_\mu]\,\delta_{m,n+\hat{\mu}} \;\to\; [1-\gamma_\mu]\,U_\mu(n) \tag{A.60}$$

and

$$[1+\gamma_\mu]\,\delta_{m,n-\hat{\mu}} \;\to\; [1+\gamma_\mu]\,U^\dagger_\mu(n) \tag{A.61}$$

As the fermions hop from site to site under the path integral simulation they interact with the gauge field by acquiring a phase shift $U_\mu(n)$, instead of the $\delta_{m,n-\hat{\mu}}$ with no interactions in the free fermion case.

The Wilson-Dirac matrix for fully dynamic simulations is therefore given by A.62

$$S_F^{(W)} = \sum_{n,m} (\hat{m}+4)\,\bar{\hat{\psi}} \left[\delta_{nm} - \frac{1}{2\,(\hat{m}+4)} \sum_{\mu=1,\hat{\mu}}^{4} \Big([1-\gamma_\mu]\,U_\mu(n) + [1+\gamma_\mu]\,U_\mu^\dagger(n) \Big) \right] \hat{\psi} \tag{A.62}$$

The hadronic masses can then be calculated from first principles using simulations with

$$\int \mathcal{D}U\mathcal{D}\bar{\psi}\mathcal{D}\psi\ \mathrm{e}^{-S_{QCD}} \tag{A.63}$$

$$S_{QCD} = S_F^{(W)}[U,\bar{\psi},\psi] + S_G^{(W)}[U] \tag{A.64}$$

Appendix B

Werner Software Library

A selection of Werner source code is listed to give an overview of the software used to perform the investigations presented in this thesis. The listings here were chosen to show how the main algorithms were implemented, including the hybrid preconditioner presented as the thesis result. The fifteen files given in this appendix are about ten percent of the total Werner project, whilst the dirac matrix source code has also been omitted. Section B.1 shows the Werner project structure for the C++ source code, which is also mirrored by the C++ declaration headers. The solvers subdirectory is expanded on the second page for clarity. Section B.2 shows the main application function which makes call to the dirac matrix library followed by the call into the Werner library to then invoke the solver for an initial condition, also specified here.

Section B.3 is the wrapper class that provides the external interface to the solver's implementation. The distribution over a cluster is also invoked from the constructor here, as requested. Section B.4 shows the matrix inverter interface header from which all the solver implementations inherit their callable interface. Implementation classes are created by the Werner wrapper as specified from the executable's command line arguments, and some examples are given: section B.5 gives the canonical preconditioned conjugate algorithm used within Werner. Section B.6 gives the normal equation version of the conjugate gradient showing how the matrix transpose is called. Section B.7 shows the BiCGStablised version for non-symmetric matrices, and section B.8 shows the general conjugate residual algorthim. Sections B.9 and B.10 show the MPI-pthread versions of BiCGStab and GCR for comparison respectively.

Section B.11 shows the base class for stationary solvers, whilst the smoother version B.12 is set up for preconditioning and multigrd. SSOR B.13 and multigrid B.14 are given as concrete examples. Finally, thesis hybrid preconditioner implementations are given in sections B.15 for the single-threaded version and B.16 for the MPI-pthread version.

B.1 Werner Directory Structure

The overall structure of the source code is shown in figure B.1. The matrix solvers subdirectories are expanded in detail in figure B.2.

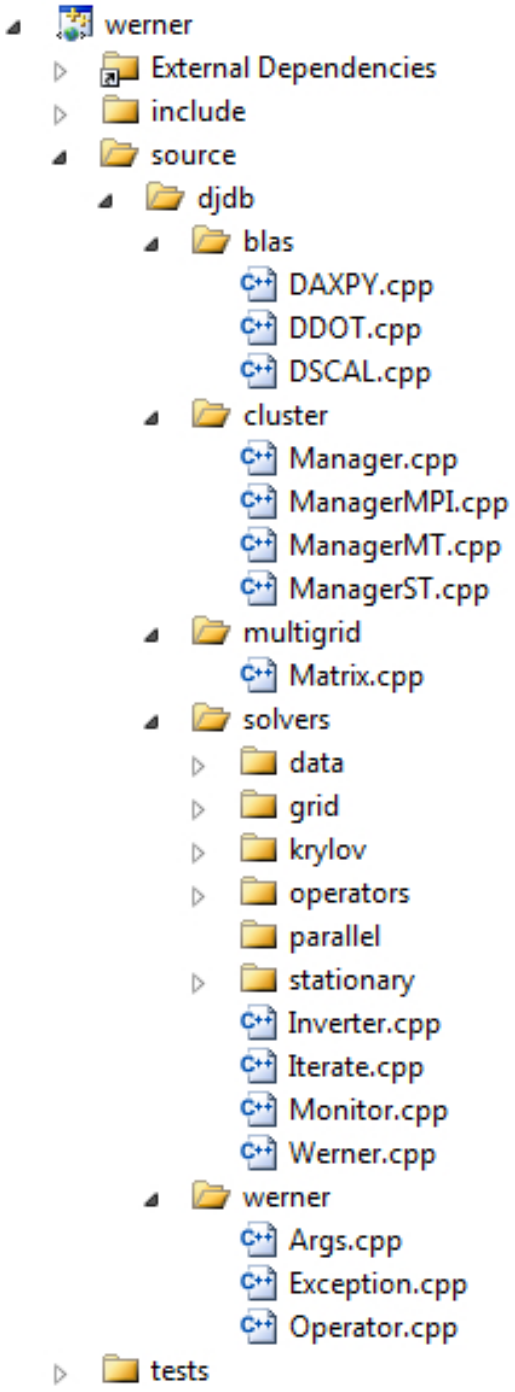


FIGURE B.1: Werner directory source structure.

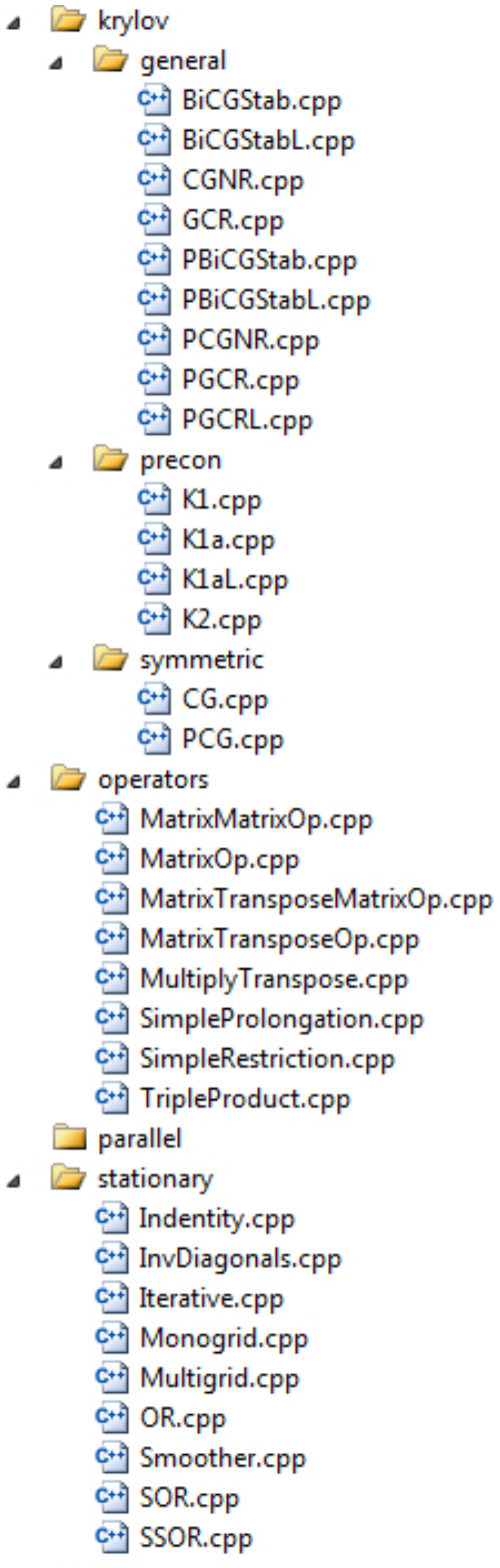


FIGURE B.2: Werner directory source structure.

B.2 Main Application Example

```
1  #include "djdb/werner/package.hpp"
2  #include "djdb/lattice/dslash/Dirac3.hpp"
3  #include "djdb/scilab/CppSciLab.hpp"
4  #include <iostream>
5  #include <fstream>
6  #include <math.h>
7
8
9  namespace dmg = djdb::multigrid;
10 namespace qcd = djdb::lattice::dslash;
11 typedef qcd::Dirac3 Dirac;
12
13
14 int buildA( dmg::Matrix& A, djdb::werner::Args& ARGS );
15 void make_b( dmg::Vector& b, const int SIZE );
16 void init_x( dmg::Vector& x, const int SIZE );
17 void test_x( djdb::werner::Operator& S );
18 void plot_graph( const djdb::werner::Args& ARGS );
19
20
21 int main( int argc, char * argv[] )
22 {
23   djdb::werner::Args args;
24
25   try
26   {
27     djdb::werner::Operator solver( args.load( argc, argv ) );
28
29     const int RANK = buildA( solver.A(), args );
30     std::cout << "Dirac.rank() = " << args.arg3() << std::endl;
31
32     make_b( solver.b(), RANK );
33     init_x( solver.x(), RANK );
34
35     solver.solve();
36     test_x( solver );
37
38     std::cout << "Residual = " << solver.residual() << std::endl;
39   }
40   catch ( const std::exception& EX )
41   {
42     std::cout << "Exception = " << EX.what();
43   }
44
45   if ( args.logtype() == "file" || args.logtype() == "all" )
46   {
47     plot_graph( args );
48   }
49
50   std::cout << std::endl << std::endl;
51
52   return 0;
53 }
```

```
54
55
56  void test_x( djdb::werner::Operator& S )
57  {
58    const int FIRST_FEW = 21;
59    const dmg::Vector& x = S.x();
60    const dmg::Vector& b = S.b();
61
62    std::cout << "x = [ ";
63    for ( int i = 0; i < FIRST_FEW; i+=2 )
64    {
65      std::cout << x[ i ] << "; ";
66    }
67    std::cout << "]" << std::endl;
68
69    std::cout << " b = [ ";
70    for ( int i = 0; i < FIRST_FEW; i+=2 )
71    {
72      std::cout << b[ i ] << "; ";
73    }
74    std::cout << "]" << std::endl;
75
76    dmg::Vector y( x.size() );
77    y = S.test( y, x );
78
79    std::cout << "Ax = [ ";
80    for ( int i = 0; i < FIRST_FEW; i+=2 )
81    {
82      std::cout << y[ i ] << "; ";
83    }
84    std::cout << "]" << std::endl;
85
86  }
87
88
89  void make_b( dmg::Vector& b, const int SIZE )
90  {
91    b.resize( SIZE );
92
93    for ( int i = 0; i < SIZE; ++i )
94    {
95      b[ i ] = 0.0;
96    }
97
98    for ( int i = 0; i < 12; ++i )
99    {
100     b[ i ] = 1.0;
101   }
102 }
103
104
105 void init_x( djdb::multigrid::Vector& x, const int SIZE )
106 {
107   x.resize( SIZE );
108
```

```
109   for ( int i = 0; i < SIZE; ++i )
110   {
111     x[ i ] = 0.0;
112   }
113 }
114
115
116 int buildA( djdb::multigrid::Matrix& A, djdb::werner::Args& args )
117 {
118   const int LATTICE_SIZE = args.arg1();
119   const double DIRAC_MASS = args.arg2() / -100.0;
120
121   Dirac dirac;
122   dirac.lattice_length( LATTICE_SIZE ).logging( true );
123   dirac.mass( DIRAC_MASS );
124
125   dirac.create();
126
127   dmg::Element element;
128
129   for ( int i = 0; i < dirac.rank(); ++i )
130   {
131     const Dirac::elements_type& ROW = dirac[ i ];
132     element.row( i );
133
134     for ( unsigned int j = 0; j < ROW.size(); ++j )
135     {
136       A.buffer( element.column( ROW[ j ].column() ).value( ROW[ j ].value() ) );
137     }
138   }
139
140   A.update();
141
142   args.arg3( dirac.rank() );
143
144   return dirac.rank();
145 }
146
147
148 void plot_graph( const djdb::werner::Args& ARGS )
149 {
150   djdb::scilab::CppSciLab scilab;
151   scilab.loadXY( ARGS.logname() );
152   scilab.plotXY( "-" );
153   scilab.savePlot( ARGS.logname() );
154   scilab.displayPlot();
155 }
156
```

B.3 Werner Solver

```
1   #include "djdb/solvers/Werner.hpp"
2   #include "djdb/solvers/operators/MatrixOp.hpp"
3   #include "djdb/solvers/Inverter.hpp"
4   #include "djdb/solvers/Iterate.hpp"
5   #include "djdb/cluster/Manager.hpp"
6   #include "djdb/werner/Args.hpp"
7   #include <iostream>
8   #include <math.h>
9
10
11  namespace djdb { namespace solvers
12  {
13
14
15  Werner::Werner( werner::Args& ARGS )
16    : params_( ARGS ),
17      matrix_( A_.crs() )
18  {
19    djdb::cluster::Manager::initialize( ARGS );
20    this->params_.display();
21  }
22
23
24  Werner::~Werner()
25  {
26    djdb::cluster::Manager::finalize();
27  }
28
29
30  double Werner::solve()
31  {
32    Inverter::smart_ptr solver = Inverter::factory::instance().create(
33      this->params_.solver() );
34
35    if ( this->params_.logging() )
36    {
37      std::cout << "------------------------------------------------" << std::endl;
38      std::cout << "                  Run Werner                    " << std::endl;
39      std::cout << "------------------------------------------------" << std::endl;
40      std::cout << "INPUT RESIDUAL-NORM: " << this->residual() << std::endl;
41    }
42
43    solver->init( this->matrix_, this->params_ );
44
45    try
46    {
47      solver->solve( this->x_, this->b_ );
48    }
49    catch ( const std::exception& e )
50    {
51      std::cerr << "Exception occurred: " << e.what() << std::endl;
52    }
53
```

```
54    if ( this->params_.logging() )
55    {
56      std::cout << "OUTPUT RESIDUAL-NORM: " << this->residual() << std::endl;
57    }
58
59    return this->residual();
60  }
61
62
63  double Werner::residual() const
64  {
65    return ::sqrt( operators::MatrixOp( this->matrix_ ).norm2r( this->x_, this->b_ ) );
66  }
67
68
69  blas::Vector& Werner::test( blas::Vector& y, const blas::Vector& X ) const
70  {
71    return operators::MatrixOp( this->matrix_ ).multiplyA( y, X );
72  }
73
74
75  multigrid::Matrix& Werner::A()
76  {
77    return this->A_;
78  }
79
80
81  blas::Vector& Werner::x()
82  {
83    return this->x_;
84  }
85
86
87  blas::Vector& Werner::b()
88  {
89    return this->b_;
90  }
91
92
93  }  } // End of namespace djdb::solvers
94
95
```

B.4 Inverter Interface Base Class

```
1  #pragma once
2
3
4  #ifndef _DJDB_SOLVERS_INVERTER_2011_12_22_
5  #define _DJDB_SOLVERS_INVERTER_2011_12_22_
6
7
8  #include <djdb/factory/smart_pointer_registry.hpp>
9
10
11 namespace djdb {
12
13
14 namespace werner { class Args; }
15 namespace blas { class Vector; }
16
17
18 namespace solvers {
19
20
21 namespace data { class CRSMatrix; }
22
23
24 class Inverter
25 {
26   public:
27     typedef factory::SmartPointerRegistry< Inverter > factory;
28     typedef factory::smart_ptr smart_ptr;
29
30   public:
31     Inverter& init( const data::CRSMatrix& A, const werner::Args& PARAMS );
32     int solve( blas::Vector& x, const blas::Vector& b );
33
34   private:
35     virtual void initImpl( const data::CRSMatrix&, const werner::Args& ) = 0;
36     virtual int solveImpl( blas::Vector& x, const blas::Vector& b ) = 0;
37
38   protected:
39     Inverter();
40
41   public:
42     virtual ~Inverter();
43 };
44
45
46 } } // End of namespace djdb::solvers
47
48
49 #endif
50
51
```

B.5 Preconditioned Conjugate Gradient

```
1  #include "djdb/solvers/Inverter.hpp"
2  #include "djdb/solvers/Iterate.hpp"
3  #include "djdb/solvers/operators/MatrixOp.hpp"
4  #include "djdb/blas/package.hpp"
5  #include "djdb/werner/Args.hpp"
6  #include <math.h>
7
8
9  namespace djdb { namespace solvers { namespace {
10
11
12 class PCG : public Inverter
13 {
14   private:
15     virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
16     virtual int solveImpl( blas::Vector& x, const blas::Vector& b );
17
18   private:
19     boost::shared_ptr< operators::MatrixOp > A_;
20     Iterate::factory::smart_ptr iterate_;
21     Inverter::factory::smart_ptr M_;
22 };
23
24
25 Inverter::factory& factory = Inverter::factory::instance().add< PCG >( "PCG" );
26
27
28 void PCG::initImpl( const data::CRSMatrix& A, const werner::Args& ARGS )
29 {
30   this->M_ = Inverter::factory::instance().create( ARGS.precon() );
31   this->M_->init( A, ARGS );
32
33   std::string s = std::string( Iterate::CLASS_NAME ).append( ARGS.logtype() );
34   this->iterate_ = Iterate::factory::instance().create( s );
35   this->iterate_->init( ARGS );
36
37   this->A_.reset( new operators::MatrixOp( A ) );
38 }
39
40
41 int PCG::solveImpl( blas::Vector& x, const blas::Vector& b )
42 {
43   double alpha = 1.0;
44   double beta  = 0.0;
45
46   blas::Vector z( x.size() );
47   blas::Vector mr( b.size() );
48   blas::Vector r = b;
49   blas::Vector p = x;
50
51   this->A_->multiplyA( z, x );
52   blas::daxpy( r, -alpha, z );
53   this->iterate_->test( blas::ddot( r, r ) );
```

```
54
55    this->M_->solve( mr, r );
56    blas::daxpy( blas::dscal( p, beta ), 1, mr );
57
58    double rho1 = blas::ddot( mr, r );
59    double rho0 = rho1;
60
61    while ( this->iterate_->next() )
62    {
63      this->A_->multiplyA( z, p );
64      alpha = rho0 / blas::ddot( z, p );
65
66      blas::daxpy( x,  alpha, p );
67      blas::daxpy( r, -alpha, z );
68      this->iterate_->test( blas::ddot( r, r ) );
69
70      this->M_->solve( mr, r );
71      rho1 = blas::ddot( mr, r );
72
73      beta = rho1 / rho0;
74      blas::daxpy( blas::dscal( p, beta ), 1, mr );
75
76      rho0 = rho1;
77    }
78
79    return this->iterate_->index();
80  }
81
82
83  }  }  } // End of namespace djdb::solvers::krylov::symmetric
84
85
```

B.6 Conjugate Gradient with Normal Residual

```
1   #include "djdb/solvers/Inverter.hpp"
2   #include "djdb/solvers/Iterate.hpp"
3   #include "djdb/solvers/operators/MatrixTransposeMatrixOp.hpp"
4   #include "djdb/blas/package.hpp"
5   #include "djdb/werner/Args.hpp"
6   #include <math.h>
7
8
9   namespace djdb { namespace solvers { namespace {
10
11
12  class PCGNR : public Inverter
13  {
14    private:
15      virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
16      virtual int solveImpl( blas::Vector& x, const blas::Vector& b );
17
18    private:
```

```
19      boost::shared_ptr< operators::MatrixTransposeMatrixOp > AtA_;
20      Iterate::factory::smart_ptr iterate_;
21      Inverter::factory::smart_ptr M_;
22  };
23
24
25  Inverter::factory& factory = Inverter::factory::instance().add< PCGNR >( "PCGNR" );
26
27
28  void PCGNR::initImpl( const data::CRSMatrix& A, const werner::Args& ARGS )
29  {
30    this->M_ = Inverter::factory::instance().create( ARGS.precon() );
31    this->M_->init( A, ARGS );
32
33    std::string s = std::string( Iterate::CLASS_NAME ).append( ARGS.logtype() );
34    this->iterate_ = Iterate::factory::instance().create( s );
35    this->iterate_->init( ARGS );
36
37    this->AtA_.reset( new operators::MatrixTransposeMatrixOp( A ) );
38  }
39
40
41  int PCGNR::solveImpl( blas::Vector& x, const blas::Vector& b )
42  {
43    blas::Vector f( b );
44    this->AtA_->multiplyAt( f, b );      // f = transpose(A) * b
45
46    double alpha = 1.0;
47    double beta  = 0.0;
48    blas::Vector z( x.size() );
49    blas::Vector mr( b.size() );
50    blas::Vector r = f;
51    blas::Vector p = x;
52
53    this->AtA_->multiplyAtA( z, x );
54    blas::daxpy( r, -alpha, z );
55
56    this->M_->solve( mr, r );
57    blas::daxpy( blas::dscal( p, beta ), 1, mr );
58
59    double rho1 = this->iterate_->test( blas::ddot( mr, r ) );
60    double rho0 = rho1;
61
62    while ( this->iterate_->next() )
63    {
64      this->AtA_->multiplyAtA( z, p );
65      alpha = rho0 / blas::ddot( z, p );
66
67      blas::daxpy( x,  alpha, p );
68      blas::daxpy( r, -alpha, z );
69      this->iterate_->test( blas::ddot( r, r ) );
70
71      this->M_->solve( mr, r );
72      rho1 = blas::ddot( mr, r );
73
```

```
74      beta = rho1 / rho0;
75      blas::daxpy( blas::dscal( p, beta ), 1, mr );
76
77      rho0 = rho1;
78  }
79
80    return this->iterate_->index();
81  }
82
83
84  }  }  } // End of namespace djdb::solvers::anonymous
```

B.7 Biorthogonal Conjugate Gradient Stablised

```
1   #include "djdb/solvers/Inverter.hpp"
2   #include "djdb/solvers/Iterate.hpp"
3   #include "djdb/solvers/operators/MatrixOp.hpp"
4   #include "djdb/blas/package.hpp"
5   #include "djdb/werner/Args.hpp"
6   #include <math.h>
7
8
9   namespace djdb { namespace solvers { namespace {
10
11
12  class PBiCGStab : public Inverter
13  {
14    private:
15      virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
16      virtual int solveImpl( blas::Vector& x, const blas::Vector& b );
17
18    private:
19      boost::shared_ptr< operators::MatrixOp > A_;
20      Iterate::factory::smart_ptr iterate_;
21      Inverter::factory::smart_ptr M_;
22  };
23
24
25  Inverter::factory& factory = Inverter::factory::instance().add< PBiCGStab >( "PBiCGStab" );
26
27
28  void PBiCGStab::initImpl( const data::CRSMatrix& A, const werner::Args& ARGS )
29  {
30    this->M_ = Inverter::factory::instance().create( ARGS.precon() );
31    this->M_->init( A, ARGS );
32
33    std::string s = std::string( Iterate::CLASS_NAME ).append( ARGS.logtype() );
34    this->iterate_ = Iterate::factory::instance().create( s );
35    this->iterate_->init( ARGS );
36    this->A_.reset( new operators::MatrixOp( A ) );
37  }
38
39
```

```
41 int PBiCGStab::solveImpl( djdb::blas::Vector& x, const djdb::blas::Vector& b )
42 {
43   double rho1 = 0.0;
44   double rho2 = 1.0;
45   double alpha = 1.0;
46   double omega = 1.0;
47   double beta  = 0.0;
48
49   blas::Vector s( x.size() );
50   blas::Vector t( x.size() );
51   blas::Vector z( x.size() );
52   blas::Vector v( x.size() );
53   blas::Vector ph( x.size() );
54   blas::Vector sh( x.size() );
55   blas::Vector r( b );
56
57   this->iterate_->test( this->A_->norm2r( x, b ) );
58
59   this->A_->multiplyA( z, x );
60   blas::daxpy( r, -alpha, z );
61   blas::Vector rt = r;
62   blas::Vector p = r;
63
64   while ( this->iterate_->next() )
65   {
66     rho1 = blas::ddot( rt, r );
67     this->iterate_->test_breakdown( rho1 );
68
69     if ( 1 == this->iterate_->index() )
70     {
71       p = r;
72     }
73     else
74     {
75       beta = ( rho1 / rho2 ) * ( alpha / omega );
76       z = p;
77       blas::daxpy( z, -omega, v );       // z = p(k) - omega(k-1) * v(k-1)
78       p = r;
79       blas::daxpy( p, beta, z );         // Update directions p(k+1) = r(k) + beta * z(k)
80     }
81
82     this->M_->solve( ph, p );
83     this->A_->multiplyA( v, ph );              // v = A(ph)
84     alpha = rho1 / blas::ddot( rt, v );   // alpha = rho / <rt|v>;  MPI_AllReduce()
85
86     s = r;
87     blas::daxpy( s, -alpha, v );
88
89     /*  ss = ::fabs( blas::ddot( s, s ) );
91     if ( ss < this->TOLERANCE_ )
92     {
93       std::cout << "ss = " << ss << ", TOL = " << this->TOLERANCE_ << std::endl;
94       blas::daxpy( x, alpha, p );
95       return this->iterate_->index();
96     } */
```

```
98
99      this->M_->solve( sh, s );
100     this->A_->multiplyA( t, sh );
101     omega = blas::ddot( t, s ) / blas::ddot( t, t );
102
103     blas::daxpy( x, alpha, ph );  // Update solution vector X(k+1) = X(k) + alpha * ph(k)
104     blas::daxpy( x, omega, sh );  // Update solution vector X(k+1) = X(k) + alpha * sh(k)
105
106     r = s;
107     blas::daxpy( r, -omega, t );  // Update residuals       r(k+1) = s(k) - omega * t
108
109     this->iterate_->test( this->A_->norm2r( x, b ) );
110     rho2 = rho1;
111 }
112
113   return this->iterate_->index();
114 }
115
116
117 }  }  }  // End of namespace djdb::solvers::cg
118
```

B.8 General Conjugate Resdiual (with Restarts)

```
1   #include "djdb/solvers/Inverter.hpp"
2   #include "djdb/solvers/Iterate.hpp"
3   #include "djdb/solvers/operators/MatrixOp.hpp"
4   #include "djdb/solvers/krylov/Kspace.hpp"
5   #include "djdb/blas/package.hpp"
6   #include "djdb/werner/Args.hpp"
7   #include <math.h>
8
9
10  namespace djdb { namespace solvers { namespace {
11
12
13  class PGCRL : public Inverter
14  {
15    private:
16      virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
17      virtual int solveImpl( blas::Vector& x, const blas::Vector& b );
18
19    private:
20      boost::shared_ptr< operators::MatrixOp > A_;
21      Iterate::factory::smart_ptr iterate_;
22      Inverter::factory::smart_ptr M_;
23      int restarts_;
24      int loops_;
25  };
26
27
28  Inverter::factory& factory = Inverter::factory::instance().add< PGCRL >( "PGCRL" );
29
```

```
31 void PGCRL::initImpl( const data::CRSMatrix& A, const werner::Args& ARGS )
32 {
33   this->M_ = Inverter::factory::instance().create( ARGS.precon() );
34   this->M_->init( A, ARGS );
35
36   std::string s = std::string( Iterate::CLASS_NAME ).append( ARGS.logtype() );
37   this->iterate_ = Iterate::factory::instance().create( s );
38   this->iterate_->init( ARGS );
39
40   this->A_.reset( new operators::MatrixOp( A ) );
41   this->restarts_ = ARGS.restarts();
42   this->loops_ = ARGS.max_iterations();
43 }
44
45
46 int PGCRL::solveImpl( blas::Vector& x, const blas::Vector& b )
47 {
48   const int L = this->restarts_;
49   this->iterate_->test( this->A_->norm2r( x, b ) );
50
51   double alpha = 1.0;
52   blas::Vector r = b;
53   blas::Vector z( x.size(), 0.0 );
54   krylov::Kspace::smart_ptr ks = krylov::Kspace::create( L, z );
55
56   this->A_->multiplyA( z, x );
57   blas::daxpy( r, -1.0, z );
58
59   this->M_->solve( ks->p( 0 ), r );
60   this->A_->multiplyA( ks->q( 0 ), ks->p( 0 ) );
61
62   while ( this->iterate_->index() < this->loops_ )
63   {
64     int k = 0;
65     while ( k < L && this->iterate_->next() )
66     {
67       ks->sigma( k ) = blas::ddot( ks->q( k ), ks->q( k ) );
68       this->iterate_->test_breakdown( ks->sigma( k ) );
69
70       alpha = blas::ddot( r, ks->q( k ) ) / ks->sigma( k );  // alpha = <r|q> / <q|q>
71       blas::daxpy( x,  alpha, ks->p( k ) );                  // x = x + alpha * p
72       blas::daxpy( r, -alpha, ks->q( k ) );                  // r = r - alpha * q
73
74       this->iterate_->test( this->A_->norm2r( x, b ) );
75
76       this->M_->solve( ks->p( k + 1 ), r );
77       this->A_->multiplyA( ks->q( k + 1 ), ks->p( k + 1 ) );
78
79       for ( int i = 0; i <= k; ++i )
80       {
81         ks->beta( i ) = -1.0 * blas::ddot( ks->q( k + 1 ), ks->q( i ) ) / ks->sigma( i );
82         blas::daxpy( ks->p( k + 1 ), ks->beta( i ), ks->p( i ) );
83         blas::daxpy( ks->q( k + 1 ), ks->beta( i ), ks->q( i ) );
84       }
85
```

```
86        ++k;
87      }
88    }
89
90    return this->iterate_->index();
91  }
92
93
94  }  }  } // End of namespace djdb::solvers::anonymous
95
96
```

B.9 MPI-pthread PBiCG Stablised

```
1   #include "djdb/solvers/Inverter.hpp"
2   #include "djdb/solvers/Iterate.hpp"
3   #include "djdb/solvers/operators/mt/MatrixOpMT.hpp"
4   #include "djdb/solvers/parallel/PreconMT.hpp"
5   #include "djdb/solvers/data/CRSMatrix.hpp"
6   #include "djdb/cluster/Manager.hpp"
7   #include "djdb/blas/mt/package.hpp"
8   #include "djdb/blas/package.hpp"
9   #include "djdb/werner/Args.hpp"
10  #include <boost/thread.hpp>
11  #include <math.h>
12
13
14  namespace djdb { namespace solvers { namespace {
15
16
17  int solverTC();
18
19
20  class PBiCGStabmt : public Inverter
21  {
22    private:
23      virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
24      virtual int solveImpl( blas::Vector& x, const blas::Vector& b );
25
26    public:
27      static const werner::Args * ARGS_;
28      static const data::CRSMatrix * A_;
29      static blas::VectorMT b;
30      static blas::VectorMT x;
31  };
32
33
34  const werner::Args * PBiCGStabmt::ARGS_( 0 );
35  const data::CRSMatrix * PBiCGStabmt::A_;
36  blas::VectorMT PBiCGStabmt::b;
37  blas::VectorMT PBiCGStabmt::x;
38
39
```

```
40
41  Inverter::factory& factory = Inverter::factory::instance()
              .add< PBiCGStabmt >( "PBiCGStabmt" );
42
43
44  void PBiCGStabmt::initImpl( const data::CRSMatrix& A, const werner::Args& ARGS )
45  {
46    PBiCGStabmt::ARGS_ = &ARGS;
47  }
48
49
50  int PBiCGStabmt::solveImpl( blas::Vector& x, const blas::Vector& b )
51  {
52    PBiCGStabmt::x = blas::VectorMT( x );
53    PBiCGStabmt::b = blas::VectorMT( b );
54
55    boost::thread_group threads;
56    for ( int i = 0; i < PBiCGStabmt::ARGS_->threads(); ++i )
57    {
58      threads.create_thread( &solverTC );
59    }
60    threads.join_all();
61
62    std::copy( PBiCGStabmt::x.begin(), PBiCGStabmt::x.end(), x.begin() );
63    return 0;
64  }
65
66
67  int solver()
68  {
69    double rho1 = 0.0;
70    double rho2 = 1.0;
71    double alpha = 1.0;
72    double omega = 1.0;
73    double beta  = 0.0;
74
75    operators::MatrixOpMT A( (*PBiCGStabmt::ARGS_).datafile(), (*PBiCGStabmt::ARGS_).arg3() );
76
77    PreconMT::factory::smart_ptr M = PreconMT::factory::instance().create( "Monogridmt" );
78    M->init( A, (*PBiCGStabmt::ARGS_) );
79
80    Iterate::factory::smart_ptr iterate = Iterate::factory::instance().create(
81      A.getIterateName( PBiCGStabmt::ARGS_->logtype() ) );
82    iterate->init( *(PBiCGStabmt::ARGS_) );
83
84    blas::Vector x( A.active_rows(), 0.0 );
85    blas::Vector b( A.active_rows(), 0.0 );
86
87    A.scatter( x, PBiCGStabmt::x );
88    A.scatter( b, PBiCGStabmt::b );
89
90    blas::Vector r = b;
91    blas::Vector r1 = b;
92
93    blas::Vector t( A.active_rows(), 0.0 );
```

```
94    blas::Vector z( A.active_rows(), 0.0 );
95    blas::Vector v( A.active_rows(), 0.0 );
96    blas::Vector s( A.active_rows(), 0.0 );
97    blas::Vector p( A.active_rows(), 0.0 );
98    blas::Vector sh( A.active_rows(), 0.0 );
99    blas::Vector ph( A.active_rows(), 0.0 );
100
101   A.multiplyA( z, PBiCGStabmt::x );
102   A.daxpy( r, -1.0, z );
103   iterate->test( A.ddot( r, r ) );
104   const blas::Vector RT = r;
105
106   // MT
107   blas::VectorMT p1( A.rows(), 0.0 );
108   blas::VectorMT s1( A.rows(), 0.0 );
109
110
111   while ( iterate->next() )
112   {
113     rho1 = A.ddot( RT, r );
114     iterate->test_breakdown( rho1 );
115
116     if ( 1 == iterate->index() )
117     {
118       p = r;
119     }
120     else
121     {
122       beta = ( rho1 / rho2 ) * ( alpha / omega );
123
124       z = p;
125       A.daxpy( z, -omega, v );       // z = p(k) - omega(k-1) * v(k-1)
126
127       p = r;
128       A.daxpy( p, beta, z );         // Update directions p(k+1) = r(k) + beta * z(k)
129     }
130
131     M->solve( ph, p );
132     A.allgather( p1, ph );
133
134     A.multiplyA( v, p1 );              // v = A(ph)
135     alpha = rho1 / A.ddot( RT, v );   // alpha = rho / <rt|v>;  MPI_AllReduce()
136
137     s = r;
138     A.daxpy( s, -alpha, v );
139
140     M->solve( sh, s );
141     A.allgather( s1, sh );
142
143     A.multiplyA( t, s1 );
144     omega = A.ddot( t, s ) / A.ddot( t, t );
145
146     A.daxpy( x, alpha, ph );  // Update vector X(k+1) = X(k) + alpha * ph(k)
147     A.daxpy( x, omega, sh );  // Update vector X(k+1) = X(k) + alpha * sh(k)
148
```

```
149     r = s;
150     A.daxpy( r, -omega, t );  // Update residuals       r(k+1) = s(k) - omega * t
151
152     A.gather( PBiCGStabmt::x, x );
153     A.multiplyA( z, PBiCGStabmt::x );
154
155     r1 = b;
156     A.daxpy( r1, -1.0, z );
157
158     iterate->test( A.ddot( r1, r1 ) );
159     rho2 = rho1;
160 }
161
162   return iterate->index();
163 }
164
165
166 int solverTC()
167 {
168   try
169   {
170     solver();
171   }
172   catch ( ... )
173   {
174   }
175
176   return 0;
177 }
178
179 }  }  }  // End of namespace djdb::solvers::cg
180
181
```

B.10 MPI-pthread GCR(restarts)

```
1  #include "djdb/solvers/Inverter.hpp"
2  #include "djdb/solvers/Iterate.hpp"
3  #include "djdb/solvers/operators/MatrixOp.hpp"
4  #include "djdb/solvers/krylov/Kspace.hpp"
5  #include "djdb/blas/package.hpp"
6  #include "djdb/werner/Args.hpp"
7  #include <math.h>
8
9
10 namespace djdb { namespace solvers { namespace {
11
12
13 class Kspace
14 {
15   public:
16     Kspace( const int RESTARTS, blas::Vector& z )
17       : SIZE_( RESTARTS + 1 ),
18         p_( SIZE_, z ),
19         q_( SIZE_, z ),
20         sigma_( SIZE_, 1.0 ),
21         beta_( SIZE_, 1.0 )
22     {
23       /* NOOP */
24     }
25
26   public:
27     djdb::blas::Vector& p( const int INDEX ) { return this->p_[ INDEX ]; }
28     djdb::blas::Vector& q( const int INDEX ) { return this->q_[ INDEX ]; }
29     double& sigma( const int INDEX ) { return this->sigma_[ INDEX ]; }
30     double& beta( const int INDEX ) { return this->beta_[ INDEX ]; }
31
32   private:
33     const int SIZE_;
34     std::vector< djdb::blas::Vector > p_;
35     std::vector< djdb::blas::Vector > q_;
36     blas::Vector sigma_;
37     blas::Vector beta_;
38 };
39
40
41 class PGCRL : public Inverter
42 {
43   private:
44     virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
45     virtual int solveImpl( blas::Vector& x, const blas::Vector& b );
46
47   private:
48     boost::shared_ptr< operators::MatrixOp > A_;
49     Iterate::factory::smart_ptr iterate_;
50     Inverter::factory::smart_ptr M_;
51     int restarts_;
52     int loops_;
53 };
```

```
54
55
56  Inverter::factory& factory = Inverter::factory::instance().add< PGCRL >( "PGCRL" );
57
58
59  void PGCRL::initImpl( const data::CRSMatrix& A, const werner::Args& ARGS )
60  {
61    this->M_ = Inverter::factory::instance().create( ARGS.precon() );
62    this->M_->init( A, ARGS );
63
64    std::string s = std::string( Iterate::CLASS_NAME ).append( ARGS.logtype() );
65    this->iterate_ = Iterate::factory::instance().create( s );
66    this->iterate_->init( ARGS );
67
68    this->A_.reset( new operators::MatrixOp( A ) );
69    this->restarts_ = ARGS.restarts();
70    this->loops_ = ARGS.max_iterations();
71  }
72
73
74  int PGCRL::solveImpl( blas::Vector& x, const blas::Vector& b )
75  {
76    const int L = this->restarts_;
77    this->iterate_->test( this->A_->norm2r( x, b ) );
78
79    double alpha = 1.0;
80    blas::Vector r = b;
81    blas::Vector z( x.size(), 0.0 );
82    krylov::Kspace::smart_ptr ks = krylov::Kspace::create( L, z );
83
84    this->A_->multiplyA( z, x );
85    blas::daxpy( r, -1.0, z );
86
87    this->M_->solve( ks->p( 0 ), r );
88    this->A_->multiplyA( ks->q( 0 ), ks->p( 0 ) );
89
90    while ( this->iterate_->index() < this->loops_ )
91    {
92      int k = 0;
93      while ( k < L && this->iterate_->next() )
94      {
95        ks->sigma( k ) = blas::ddot( ks->q( k ), ks->q( k ) );
96        this->iterate_->test_breakdown( ks->sigma( k ) );
97
98        alpha = blas::ddot( r, ks->q( k ) ) / ks->sigma( k );  // alpha = <r|q> / <q|q>
99        blas::daxpy( x,  alpha, ks->p( k ) );                  // x = x + alpha * p
100       blas::daxpy( r, -alpha, ks->q( k ) );                  // r = r - alpha * q
101
102       this->iterate_->test( this->A_->norm2r( x, b ) );
103
104       this->M_->solve( ks->p( k + 1 ), r );
105       this->A_->multiplyA( ks->q( k + 1 ), ks->p( k + 1 ) );
106
```

```
107       for ( int i = 0; i <= k; ++i )
108       {
109         ks->beta( i ) = -1.0 * blas::ddot( ks->q( k + 1 ), ks->q( i ) ) / ks->sigma( i );
110         blas::daxpy( ks->p( k + 1 ), ks->beta( i ), ks->p( i ) );
111         blas::daxpy( ks->q( k + 1 ), ks->beta( i ), ks->q( i ) );
112       }
113
114       ++k;
115     }
116   }
117
118   return this->iterate_->index();
119 }
120
121
122 }  }  } // End of namespace djdb::solvers::anonymous
123
124
```

B.11 Stationary Solver Template Class

```
1   #include "djdb/solvers/Inverter.hpp"
2   #include "djdb/solvers/Iterate.hpp"
3   #include "djdb/solvers/operators/MatrixOp.hpp"
4   #include "djdb/solvers/stationary/Smoother.hpp"
5   #include "djdb/blas/package.hpp"
6   #include "djdb/werner/Args.hpp"
7   #include <math.h>
8
9
10  namespace djdb { namespace solvers { namespace stationary {
11
12
13  class Stationary : public Inverter
14  {
15    private:
16      virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
17      virtual int solveImpl( blas::Vector& x, const blas::Vector& b );
18
19    private:
20      virtual const std::string& smoother() const = 0;
21
22    private:
23      boost::shared_ptr< operators::MatrixOp > A_;
24      stationary::Smoother::smart_ptr smoother_;
25      Iterate::factory::smart_ptr iterate_;
26  };
27
28
29  class Jacobi : public Stationary
30  {
31    private:
32      virtual const std::string& smoother() const { return SMOOTHER; }
```

```
33
34    public:
35      static const std::string SMOOTHER;
36  };
37
38
39  class GaussSeidel : public Stationary
40  {
41    private:
42      virtual const std::string& smoother() const { return SMOOTHER; }
43
44    public:
45      static const std::string SMOOTHER;
46  };
47
48
49  class SSOR : public Stationary
50  {
51    private:
52      virtual const std::string& smoother() const { return SMOOTHER; }
53
54    public:
55      static const std::string SMOOTHER;
56  };
57
58
59  const std::string SSOR::SMOOTHER( "SSOR" );
60  const std::string Jacobi::SMOOTHER( "OR" );
61  const std::string GaussSeidel::SMOOTHER( "SOR" );
62
63
64  namespace {
65
66
67  Inverter::factory& factory = Inverter::factory::instance()
68    .add< SSOR >( "SSOR" )
69    .add< Jacobi >( "Jacobi" )
70    .add< GaussSeidel >( "GaussSeidel" )
71  ;
72
73
74  }
75
76
77  void Stationary::initImpl( const data::CRSMatrix& A, const werner::Args& ARGS )
78  {
79    this->A_.reset( new operators::MatrixOp( A ) );
80
81    std::string s = std::string( Iterate::CLASS_NAME ).append( ARGS.logtype() );
82    this->iterate_ = Iterate::factory::instance().create( s );
83    this->iterate_->init( ARGS );
84
85    this->smoother_ = Smoother::factory::instance().create( this->smoother() );
86    this->smoother_->init( A, ARGS );
87  }
```

```
88
89
90  int Stationary::solveImpl( blas::Vector& x, const blas::Vector& b )
91  {
92    this->iterate_->test( this->A_->norm2r( x, b ) );
93    while ( this->iterate_->next() )
94    {
95      this->smoother_->solve( x, b );
96      this->iterate_->test( this->A_->norm2r( x, b ) );
97  }
98
99    return this->iterate_->index();
100 }
101
102
103 }  }  }  // End of namespace djdb::solvers::stationary
104
105
```

B.12 Smoother Base Class

```
1   #include "djdb/solvers/stationary/Smoother.hpp"
2
3
4   namespace djdb { namespace solvers { namespace stationary {
5
6
7   Smoother::Smoother()
8   {
9     /* NOOP */
10  }
11
12
13  Smoother::~Smoother()
14  {
15    /* NOOP */
16  }
17
18
19  Smoother& Smoother::init( const data::CRSMatrix& M, const werner::Args& A )
20  {
21    this->initImpl( M, A );
22    return *this;
23  }
24
25
26  Smoother& Smoother::init( const data::CRSMatrix& M, const blas::Vector& D,
27    const werner::Args& A )
28  {
29    this->initImpl( M, D, A );
30    return *this;
31  }
32
```

```
33
34 blas::Vector& Smoother::solve( blas::Vector& x, const blas::Vector& b ) const
35 {
36   return this->solveImpl( x, b );
37 }
38
39
40 blas::Vector& Smoother::update( blas::Vector& x, const blas::Vector& b ) const
41 {
42   return this->updateImpl( x, b );
43 }
44
45
46 blas::Vector& Smoother::update( data::CRSMatrix& r, blas::Vector& x,
47   const blas::Vector& b ) const
48 {
49   return this->updateImpl( r, x, b );
50 }
51
52
53 } } } // End of namespace djdb::solvers::preconditioning
54
55
```

B.13 Symmetric Gauss-Seidel Relaxed

```
1  #include "djdb/solvers/stationary/Smoother.hpp"
2  #include "djdb/solvers/data/CRSMatrix.hpp"
3  #include "djdb/blas/Vector.hpp"
4  #include "djdb/werner/Args.hpp"
5
6
7  namespace djdb { namespace solvers { namespace stationary {
8
9
10 class SSORSmoother : public Smoother
11 {
12   private:
13     virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
14     virtual void initImpl( const data::CRSMatrix&, const blas::Vector&, const werner::Args& );
15     virtual blas::Vector& solveImpl( blas::Vector& x, const blas::Vector& b ) const;
16     virtual blas::Vector& updateImpl( blas::Vector& x, const blas::Vector& b ) const;
17     virtual blas::Vector& updateImpl( data::CRSMatrix& r, blas::Vector& x,
18       const blas::Vector& b ) const;
19
20   private:
21      const data::CRSMatrix& A() const { return *(this->A_); }
22      const double D( const int I) const { return (*(this->D_))[ I ]; }
23
24   private:
25     blas::Vector store_D_;
26     const data::CRSMatrix * A_;
27     const blas::Vector * D_;
```

```
28      double omega_;
29      int depth_;
30      int rows_;
31  };
32
33
34  static Smoother::factory& factory = Smoother::factory::instance()
35    .add< SSORSmoother >( "SSOR" );
36
37
38  void SSORSmoother::initImpl( const data::CRSMatrix& M,
39    const werner::Args& ARGS )
40  {
41    M.get_diagonals( this->store_D_ );
42    for ( int i = 0; i < M.rows(); ++i )
43    {
44      this->store_D_[ i ] = 1.0 / this->store_D_[ i ];
45    }
46
47    this->omega_ = ARGS.omega();
48    this->depth_ = ARGS.depth();
49    this->rows_ = M.rows();
50    this->D_ = &(this->store_D_);
51    this->A_ = &M;
52  }
53
54
55  void SSORSmoother::initImpl( const data::CRSMatrix& M,
56    const blas::Vector& DIAGONAL, const werner::Args& ARGS )
57  {
58    this->omega_ = ARGS.omega();
59    this->depth_ = ARGS.depth();
60    this->rows_ = M.rows();
61    this->D_ = &DIAGONAL;
62    this->A_ = &M;
63  }
64
65
66  blas::Vector& SSORSmoother::solveImpl( blas::Vector& x,
67    const blas::Vector& b ) const
68  {
69    const int REPEATS = ( this->depth_ + 1 ) / 2;
70    for ( int k = 0; k < REPEATS; ++k )
71    {
72      int idx = 0;
73      for( int row = 0; row < this->rows_; ++row )
74      {
75        double tmp = b[ row ];
76
77        for( int j = 0; j < this->A()[ row ].non_zeros(); ++j )
78        {
79          const data::CRSMatrix::element_type& ELEMENT = this->A().element(idx);
80          tmp -= ELEMENT.value() * x[ ELEMENT.column() ];
81          ++idx;
82        }
```

```
83
84        x[ row ] += this->omega_ * tmp * this->D( row );
85      }
86
87      idx = this->A().non_zeros() - 1;
88      for( int row = this->rows_ - 1; row >= 0; --row )
89      {
90        double tmp = b[ row ];
91
92        for( int j = 0; j < this->A()[ row ].non_zeros(); ++j )
93        {
94          const data::CRSMatrix::element_type& ELEMENT = this->A().element(idx);
95          tmp -= ELEMENT.value() * x[ ELEMENT.column() ];
96          --idx;
97        }
98
99        x[ row ] += this->omega_ * tmp * this->D( row );
100     }
101   }
102
103   return x;
104 }
105
106
107 blas::Vector& SSORSmoother::updateImpl( data::CRSMatrix& rcrs,
108   blas::Vector& x, const blas::Vector& b ) const
109 {
110   for( int row = 0; row < this->rows_; ++row )
111 {
112     x[ row ] = this->omega_ * b[ row ] * this->D( row );
113 }
114
115   int idx = 0;
116   for( int row = 0; row < this->rows_; ++row )
117 {
118     double acc = b[ row ];
119
120     for( int j = 0; j < this->A()[ row ].non_zeros(); ++j )
121     {
122       const data::CRSMatrix::element_type& ELEMENT = this->A().element(idx);
123       acc -= ELEMENT.value() * x[ ELEMENT.column() ];
124       ++idx;
125     }
126
127     rcrs.updateElementValue( row, acc );
128 }
129
130   return x;
131 }
132
133
134 blas::Vector& SSORSmoother::updateImpl( blas::Vector& x,
135   const blas::Vector& b ) const
136 {
137   for( int row = 0; row < this->rows_; ++row )
```

```
138 {
139     x[ row ] = this->omega_ * b[ row ] * this->D( row );
140 }
141
142   return x;
143 }
144
145
146 }  }  } // End of namespace djdb::solvers::stationary
147
148
```

B.14 Multigrid Solver

```
1   #include "djdb/solvers/Inverter.hpp"
2   #include "djdb/solvers/operators/SimpleProlongation.hpp"
3   #include "djdb/solvers/operators/SimpleRestriction.hpp"
4   #include "djdb/solvers/stationary/Smoother.hpp"
5   #include "djdb/solvers/grid/Operators.hpp"
6   #include "djdb/solvers/grid/Matrices.hpp"
7   #include "djdb/solvers/grid/Vectors.hpp"
8   #include "djdb/solvers/grid/Levels.hpp"
9   #include "djdb/solvers/data/CRSMatrix.hpp"
10  #include "djdb/werner/Args.hpp"
11  #include "djdb/blas/DAXPY.hpp"
12  #include <iostream>
13  #include <vector>
14
15
16  namespace djdb { namespace solvers { namespace stationary {
17
18
19  class Multigrid : public Inverter
20  {
21    private:
22      virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
23      virtual int solveImpl( blas::Vector& x, const blas::Vector& b );
24
25    private:
26      blas::Vector& update( const int LEVEL ) const;
27      blas::Vector& getx0( blas::Vector& x );
28      void setb0( const blas::Vector& b );
29
30    private:
31      boost::shared_ptr< grid::Levels > level_;
32  };
33
34
35  static Inverter::factory& factory = Inverter::factory::instance()
36    .add< Multigrid >( "Multigrid" );
37
```

```
38 void Multigrid::initImpl( const data::CRSMatrix& M0, const werner::Args& ARG )
39 {
40   this->level_.reset( new grid::Levels( ARG, M0 ) );
41 }
42
43
44 int Multigrid::solveImpl( blas::Vector& x, const blas::Vector& b )
45 {
46   this->setb0( b );
47   this->update( 0 );
48   this->getx0( x );
49   return 0;
50 }
51
52
53 void Multigrid::setb0( const blas::Vector& b )
54 {
55   ::memcpy( this->level_->vector( 0 ).b().data(), b.data(),
56     b.size() * sizeof( double ) );
57 }
58
59
60 blas::Vector& Multigrid::getx0( blas::Vector& x )
61 {
62   const blas::Vector& x0 = this->level_->vector( 0 ).x();
63   ::memcpy( x.data(), x0.data(), x0.size() * sizeof( double ) );
64   return x;
65 }
66
67
68 djdb::blas::Vector& Multigrid::update( const int L ) const
69 {
70   if ( L == this->level_->depth() )
71 {
72     return this->level_->vector( L ).x();
73 }
74
75   stationary::Smoother& smoother = this->level_->function( L ).smoother();
76   const blas::Vector& b = this->level_->vector( L ).b();
77   blas::Vector& x = this->level_->vector( L ).x();
78   blas::Vector& s = this->level_->vector( L ).s();
79
80   smoother.update( this->level_->matrix( L ).rcrs(), x, b );
81   this->level_->function( L ).restrict( this->level_->vector( L + 1 ).b() );
82
83   this->update( L + 1 );
84
85   this->level_->function( L ).prolong( this->level_->vector( L + 1 ).x(), s );
86   djdb::blas::DAXPY()( x, 1.0, s );
87
88   return smoother.solve( x, b );
89 }
90
91
92 }  }  } // End of namespace djdb::solvers::stationary
```

B.15 GCR-SSOR Hybrid Preconditioner

```
1  #include "djdb/solvers/stationary/Smoother.hpp"
2  #include "djdb/solvers/operators/MatrixOp.hpp"
3  #include "djdb/solvers/data/CRSMatrix.hpp"
4  #include "djdb/solvers/krylov/Kspace.hpp"
5  #include "djdb/solvers/Inverter.hpp"
6  #include "djdb/blas/package.hpp"
7  #include "djdb/blas/Vector.hpp"
8  #include "djdb/werner/Args.hpp"
9  #include <iostream>
10
11
12 namespace djdb { namespace solvers { namespace stationary {
13
14
15 class K1 : public Smoother
16 {
17   private:
18     virtual void initImpl( const data::CRSMatrix&, const werner::Args& );
19     virtual void initImpl( const data::CRSMatrix&, const blas::Vector&,
20       const werner::Args& );
21     virtual blas::Vector& solveImpl( blas::Vector& x, const blas::Vector& b ) const;
22     virtual blas::Vector& updateImpl( blas::Vector& x, const blas::Vector& b ) const;
23     virtual blas::Vector& updateImpl( data::CRSMatrix& r, blas::Vector& x,
24       const blas::Vector& b ) const;
25
26   private:
27      const data::CRSMatrix& A() const { return *(this->crsA_); }
28
29   private:
30     krylov::Kspace::smart_ptr ks_;
31     boost::shared_ptr< operators::MatrixOp > A_;
32     const data::CRSMatrix * crsA_;
33     Inverter::factory::smart_ptr M_;
34     double gamma_;
35     int restarts_;
36     bool log_;
37     int rows_;
38 };
39
40
41 static Smoother::factory& factory = Smoother::factory::instance().add< K1 >( "K1" );
42
43
44 void K1::initImpl( const data::CRSMatrix& M, const werner::Args& ARGS )
45 {
46   this->M_ = Inverter::factory::instance().create( "Monogrid" );
47   this->M_->init( M, werner::Args( ARGS ).smoother( "SSOR" ) );
48
49   this->A_.reset( new operators::MatrixOp( M ) );
50   this->restarts_ = ARGS.restarts();
51   this->gamma_ = ARGS.gamma();
52   this->log_ = ARGS.logging();
53
```

```
54    blas::Vector z( M.rows(), 0.0 );
55    this->ks_ = krylov::Kspace::create( this->restarts_, z );
56
57    this->rows_ = M.rows();
58    this->crsA_ = &M;
59  }
60
61
62  void K1::initImpl( const data::CRSMatrix& M, const blas::Vector& DIAGONAL,
63    const werner::Args& ARGS )
64  {
65    this->initImpl( M, ARGS );
66  }
67
68
69  blas::Vector& K1::solveImpl( blas::Vector& x, const blas::Vector& b ) const
70  {
71    double rho = this->A_->norm2r( x, b );
72    const double RR0 = rho;
73    const double FACTOR = this->gamma_;
74    const int L = this->restarts_;
75
76    int k = 0;
77    double alpha = 1.0;
78    blas::Vector r = b;
79    blas::Vector z( x.size(), 0.0 );
80
81    this->A_->multiplyA( z, x );
82    blas::daxpy( r, -1.0, z );
83
84    this->M_->solve( this->ks_->p( 0 ), r );
85    this->A_->multiplyA( this->ks_->q( 0 ), this->ks_->p( 0 ) );
86
87    while ( RR0 < FACTOR * rho )
88    {
89      for ( k = 0; k < L && (RR0 < FACTOR * rho); ++k )
90      {
91        this->ks_->sigma( k ) = blas::ddot( this->ks_->q( k ), this->ks_->q( k ) );
92
93        alpha = blas::ddot( r, this->ks_->q( k ) ) / this->ks_->sigma( k );
94        blas::daxpy( x,  alpha, this->ks_->p( k ) );
95        blas::daxpy( r, -alpha, this->ks_->q( k ) );
96
97        rho = this->A_->norm2r( x, b );
98
99        if ( this->log_ && !(k%10) )
100       {
101         std::cout << "K1(" << k << ") norm(r) = " << rho << std::endl;
102       }
103
104       this->M_->solve( this->ks_->p( k + 1 ), r );
105       this->A_->multiplyA( this->ks_->q( k + 1 ), this->ks_->p( k + 1 ) );
106
```

```
107       for ( int i = 0; i <= k; ++i )
108       {
109         this->ks_->beta( i ) = -1.0 * blas::ddot( this->ks_->q( k + 1 ),
            this->ks_->q( i ) ) / this->ks_->sigma( i );
110         blas::daxpy( this->ks_->p( k + 1 ), this->ks_->beta( i ), this->ks_->p( i ) );
111         blas::daxpy( this->ks_->q( k + 1 ), this->ks_->beta( i ), this->ks_->q( i ) );
112       }
113     }
114   }
115
116   if ( this->log_ )
117   {
118     std::cout << "K1(" << k << ") norm(r) = " << rho << std::endl;
119   }
120
121   return x;
122 }
123
124
125 blas::Vector& K1::updateImpl( blas::Vector& x,
126   const blas::Vector& b ) const
127 {
128   return x;
129 }
130
131
132 blas::Vector& K1::updateImpl( data::CRSMatrix& rcrs, blas::Vector& x,
    const blas::Vector& b ) const
133 {
134   int index = 0;
135   for( int row = 0; row < this->rows_; ++row )
136 {
137     double acc = b[ row ];
138
139     for( int j = 0; j < this->A()[ row ].non_zeros(); ++j )
140     {
141       const data::CRSMatrix::element_type& ELEMENT = this->crsA_->element( index );
142       acc -= ELEMENT.value() * x[ ELEMENT.column() ];
143       ++index;
144     }
145
146     rcrs.updateElementValue( row, acc );
147 }
148
149   return x;
150 }
151
152
153 }  }  }  // End of namespace djdb::solvers::stationary
154
155
```

B.16 MPI-pthread GCR-SSOR Hybrid Preconditioner

```
1  #include "djdb/solvers/parallel/PreconMT.hpp"
2  #include "djdb/solvers/operators/MatrixOp.hpp"
3  #include "djdb/solvers/data/CRSMatrix.hpp"
4  #include "djdb/solvers/krylov/Kspace.hpp"
5  #include "djdb/solvers/Inverter.hpp"
6  #include "djdb/solvers/Iterate.hpp"
7  #include "djdb/blas/package.hpp"
8  #include "djdb/blas/Vector.hpp"
9  #include "djdb/werner/Args.hpp"
10 #include <iostream>
11
12
13 #include "djdb/blas/mt/VectorMT.hpp"
14 #include "djdb/solvers/operators/mt/MatrixOpMT.hpp"
15
16
17 namespace djdb { namespace solvers { namespace stationary {
18
19
20 class GCSmt
21 {
22   public:
23     void init( operators::MatrixOpMT&, const werner::Args& );
24     int solve( blas::Vector& x, const blas::Vector& b );
25
26   private:
27     const werner::Args& args() const { return *(this->ARGS_); }
28     operators::MatrixOpMT& A() { return *(this->A_); }
29
30   private:
31     std::vector< double > sigma_;
32     blas::Vector r_;
33     blas::Vector z_;
34     std::vector< djdb::blas::Vector > q_;
35     std::vector< djdb::blas::Vector > p_;
36     blas::VectorMT pt_;
37     blas::VectorMT xh_;
38
39   private:
40     const werner::Args * ARGS_;
41     operators::MatrixOpMT * A_;
42     static const int MAX_RESTARTS = 3;
43 };
44
45
46 void GCSmt::init( operators::MatrixOpMT& M, const werner::Args& ARGS )
47 {
48   this->A_ = &M;
49   this->ARGS_ = &ARGS;
50
51   const int L = ARGS.restarts();
52   const double ZERO = 0.0;
53
```

```
54    this->sigma_.resize( L, ZERO );
55    this->r_.resize( M.active_rows(), ZERO );
56    this->z_.resize( M.active_rows(), ZERO );
57    this->q_.resize( L, this->z_ );
58    this->p_.resize( L, this->z_ );
59    this->pt_.resize( M.rows(), ZERO );
60    this->xh_.resize( M.rows(), ZERO );
61  }
62
63
64  int GCSmt::solve( blas::Vector& x, const blas::Vector& b )
65  {
66    double beta = 1.0;
67    double alpha = beta;
68    const int L = this->args().restarts();
69
70    PreconMT::factory::smart_ptr M = PreconMT::factory::instance().create( "Monogridmt" );
71    M->init( this->A(), this->args() );
72
73    this->r_ = b;
74    this->A().allgather( this->xh_, x );
75    this->A().multiplyA( this->z_, this->xh_ );
76    this->A().daxpy( this->r_, -1.0, this->z_ );
77
78    double rho = this->A().ddot( this->r_, this->r_ );
79    const double TARGET = rho / this->args().gamma();
80
81    M->solve( this->p_[ 0 ], this->r_ );
82    this->A().allgather( this->pt_, this->p_[ 0 ] );
83    this->A().multiplyA( this->q_[ 0 ], this->pt_ );
84
85    for ( int rpts = 0; rpts < MAX_RESTARTS; ++rpts )
86    {
87      for ( int k = 0; k < L; ++k )
88      {
89        this->sigma_[ k ] = this->A().ddot( this->q_[ k ], this->q_[ k ] );
90
91        alpha = this->A().ddot( this->r_, this->q_[ k ] ) / this->sigma_[ k ];
92        this->A().daxpy( x,  alpha, this->p_[ k ] );
93        this->A().daxpy( this->r_, -alpha, this->q_[ k ] );
94
95        rho = this->A().ddot( this->r_, this->r_ );
96        if ( rho < TARGET )
97        {
98          return k;
99        }
100
101       M->solve( this->p_[ k + 1 ], this->r_ );
102       this->A().allgather( this->pt_, this->p_[ k + 1 ] );
103       this->A().multiplyA( this->q_[ k + 1 ], this->pt_ );
104
105       for ( int i = 0; i <= k; ++i )
106       {
107         beta = this->A().ddot( this->q_[ k + 1 ], this->q_[ i ] )
                    / ( -1.0 * this->sigma_[ i ] );
```

```
108         this->A().daxpy( this->p_[ k + 1 ], beta, this->p_[ i ] );
109         this->A().daxpy( this->q_[ k + 1 ], beta, this->q_[ i ] );
110       }
111     }
112
113     this->A().allgather( this->xh_, x );
114     this->A().multiplyA( this->z_, this->xh_ );
115     this->A().daxpy( this->r_, -1.0, this->z_ );
116
117     M->solve( this->p_[ 0 ], this->r_ );
118     this->A().allgather( this->pt_, this->p_[ 0 ] );
119     this->A().multiplyA( this->q_[ 0 ], this->pt_ );
120   }
121
122   return -1;
123 }
124
125
126
127
128 class K1amt : public PreconMT
129 {
130   private:
131     virtual void initImpl( operators::MatrixOpMT&, const werner::Args& );
132     virtual int solveImpl( blas::Vector& x, const blas::Vector& b );
133
134   private:
135     boost::shared_ptr< GCSmt > smoother_;
136
137 };
138
139
140 static PreconMT::factory& factory = PreconMT::factory::instance().add< K1amt >( "K1amt" );
141
142
143 void K1amt::initImpl( operators::MatrixOpMT& M, const werner::Args& ARG )
144 {
145   this->smoother_.reset( new GCSmt );
146   this->smoother_->init( M, ARG );
147 }
148
149
150 int K1amt::solveImpl( blas::Vector& x, const blas::Vector& b )
151 {
152   this->smoother_->solve( x, b );
153   return 0;
154 }
155
156
157 }  }  }  // End of namespace djdb::solvers::parallel
158
159
```

Bibliography

[1] D. J. D. Beaven, J. Fulcher, C. H. Yang, Z. Zeng, W. Xu, and C. Zhang. Photo absorption in spintronic multilayer systems. *Physica E: Low-Dimensional Systems and Nanostructures*, 40(6):2138–2140, 2008.

[2] F. Gao, D. J. D. Beaven, J. Fulcher, C. H. Yang, Z. Zeng, W. Xu, and C. Zhang. Thermodynamic properties of two-dimensional semiconductors with spin-orbit coupling. *Physica E: Low-Dimensional Systems and Nanostructures*, 40(5):1454–1456, 2008.

[3] A. R. Wright, T. E. O'Brien, D. J. D. Beaven, and C. Zhang. Gapless insulator and a band gap scaling law in semihydrogenated graphene. *Applied Physics Letters*, 97(4), 2010.

[4] D. J. D. Beaven, J. Fulcher, and C. Zhang. A linear time complexity solver for lattice quantum field theory computations. In *Lecture Notes in Engineering and Computer Science*, volume 2, pages 1641–1646, 2012.

[5] P. A. M. Dirac. The lagrangian in quantum mechanics. *Physikalische Zeitschrift der Sowjetunion*, 3:6472, 1933.

[6] R. P. Feynman. Space-time approach to non-relativistic quantum mechanics. *Rev. Mod. Phys.*, 20:367–387, Apr 1948.

[7] K. G. Wilson. Confinement of quarks. *Physical Review D*, 10(8):2445–2459, 1974.

[8] Michael Creutz. *Quarks, gluons and lattices.* Cambridge monographs on mathematical physics. Cambridge Univ. Press, Cambridge, 1983.

[9] Heinz J Rothe. *Lattice Gauge Theories: An Introduction.* World Scientific Lecture Notes in Physics. World Scientific, Singapore, 3rd edition, 2005.

[10] I. Montvay and G. Munster. *Quantum fields on a lattice.* Cambridge Univ. Press, 1994.

[11] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354-356, 1969.

[12] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Computation*, 9(3):251-280, 1990, 1990.

[13] V. Vassilevska-Williams. Breaking the coppersmith-winograd barrier. *ACM Proceedings STOC*, 2012.

[14] Clay Mathematics Institute. Yang-mills and mass gap, 2000. URL `http://www.claymath.org/millennium/Yang-Mills_Theory`.

[15] Peter J. Moran, Derek B. Leinweber, and Jianbo Zhang. Wilson mass dependence of the overlap topological charge density. *Phys.Lett.B695:337-342*, 2011.

[16] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[17] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Standards*, 49:409–435, 1952.

[18] Michael A. Clark, Ron Babich, Kipton Barros, Richard C. Brower, and Claudio Rebbi. Solving lattice qcd systems of equations using mixed precision solvers on gpus. *Computer Physics Communications*, 181(9):1517–1528, 2010.

[19] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bureau Standards*, 49:33–53, 1952.

[20] H. A. Van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM J. Sci. and Stat. Comput.*, 13(2):631644, 1992.

[21] Artan Borici and Philippe de Forcrand. Fast krylov space methods for calculation of quark propagator. *IPS research report*, 94(3), 1994.

[22] Y. Saad and M. Schultz. Gmres: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7(3):856–869, 1986.

[23] Vance Faber and Thomas Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM Journal on Numerical Analysis*, 21(2):352–11, 1984.

[24] S. C. Eisenstat, H. C. Elman, and M. H. Schultz. Variational methods for nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.*, 20(2):345–357, 1983.

[25] R. Babich, J. Brannick, R.C. Brower, M.A. Clark, T.A. Manteuffel, et al. Adaptive multigrid algorithm for the lattice wilson-dirac operator. *Phys.Rev.Lett.*, 105: 201602, 2010.

[26] D. C. Sorensen and C. Yang. Accelerating the lanczos algorithm via polynomial spectral transformations. Technical report, Rice University, 1997.

[27] Yong Dou, Stamatis Vassiliadis, Georgi Kuzmanov, and Georgi Gaydadjiev. 64-bit floating-point fpga matrix multiplication. In *FPGA*, pages 86–95, 2005.

[28] Walter B. Ligon III, Scott McMillan, Greg Monn, Kevin Schoonover, Fred Stivers, and Keith D. Underwood. A re-evaluation of the practicality of floating-point operations on fpgas. In *FCCM*, pages 206–215, 1998.

[29] Keith Underwood. FPGAs vs. CPUs: trends in peak floating-point performance. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 171–180, New York, NY, USA, 2004. ACM.

[30] Ling Zhuo and Viktor K. Prasanna. High performance linear algebra operations on reconfigurable systems. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, 2005.

[31] Owen Callanan, David Gregg, Andy Nisbet, and Mike Peardon. High performance scientific computing using fpgas with ieee floating point and logarithmic arithmetic for lattice qcd. In *FPL*, pages 1–6, 2006.

[32] Ali Irturk, Bridget Benson, Shahnam Mirzaei, and Ryan Kastner. An fpga design space exploration tool for matrix inversion architectures. In *SASP*, pages 42–47, 2008.

[33] Jason D. Bakos and Krishna K. Nagar. Exploiting matrix symmetry to improve fpga-accelerated conjugate gradient. In *FCCM*, pages 223–226, 2009.

[34] P. Bellows and B. Hutchings. Jhdl - an hdl for reconfigurable systems. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, FCCM '98, pages 175–185. IEEE Computer Society, 1998.

[35] Yongfeng Gu and M.C. Herbordt. Fpga-based multigrid computation for molecular dynamics simulations. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 117–126, 2007.

[36] Zhi Guo, Walid Najjar, and Frank Vahid. A quantitative analysis of the speedup factors of fpgas over processors. In *Processors, Int. Symp. Field-Programmable gate Arrays (FPGA*, pages 162–170. ACM Press, 2004.

[37] Keith D Underwood and K Scott Hemmert. Closing the gap: Cpu and fpga trends in sustainable floating-point blas performance. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 219–228. IEEE, 2004.

[38] B.R. Lee and N. Burgess. Improved small multiplier based multiplication, squaring and division. In *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pages 91–97, 2003.

[39] K.K. Liu, C.B. Cameron, and A.A. Sarkady. Using mitrion-c to implement floating-point arithmetic on a cray xd1. In *DoD HPCMP Users Group Conference, 2008. DOD HPCMP UGC*, pages 391–395, 2008.

[40] J. Wawrzynek, D. Patterson, M. Oskin, Shih-Lien Lu, C. Kozyrakis, J.C. Hoe, D. Chiou, and K. Asanovic. Ramp: Research accelerator for multiple processors. *Micro, IEEE*, 27(2):46–57, 2007.

[41] Ling Zhuo and V.K. Prasanna. Design tradeoffs for blas operations on reconfigurable hardware. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 78–86, 2005.

[42] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, FPGA '05, pages 63–74. ACM, 2005.

[43] Ulrich W. Eisenecker. Generative programming (gp) with c++. In *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, JMLC '97, pages 351–365. Springer-Verlag, 1997.

[44] C. Cote and Z. Zilic. Automated systemc to vhdl translation in hardware/software codesign. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, volume 2, pages 717–720 vol.2, 2002.

[45] L. Charest and E. Aboulhamid. A vhdl/systemc comparison in handling design reuse. In *System-on-Chip for Real-Time Applications*, volume 711 of *The Kluwer International Series in Engineering and Computer Science*, pages 41–50. Springer US, 2003.

[46] D. Lettnin, A. Braun, M. Bodgan, J. Gerlach, and W. Rosenstiel. Synthesis of embedded systemc design: a case study of digital neural networks. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 3, pages 248–253 Vol.3, 2004.

[47] M. Bombana and F. Bruschi. Systemc-vhdl co-simulation and synthesis in the hw domain. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 101–105 suppl., 2003.

[48] R. Scrofano, Seonil Choi, and V.K. Prasanna. Energy efficiency of fpgas and programmable processors for matrix multiplication. In *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pages 422–425, 2002.

[49] C. Cote and Z. Zilic. Automated systemc to vhdl translation in hardware/software codesign. In *Electronics, Circuits and Systems, 2002. 9th International Conference on*, volume 2, pages 717–720 vol.2, 2002.

[50] Jimmy Xu, Nikhil Subramanian, Adam Alessio, and Scott Hauck. Impulse c vs. vhdl for accelerating tomographic reconstruction. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 171–174. IEEE Computer Society, 2010.

[51] Zbigniew Koza, Maciej Matyka, Sebastian Szkoda, and Lukasz Miroslaw. Compressed multiple-row storage format. *CoRR*, abs/1203.2946, 2012.

[52] Y. Notay. An aggregation-based algebraic multigrid method. *Electronic Transactions on Numerical Analysis*, 37:123–146, 2010.

[53] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2000.

[54] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[55] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[56] Mark D. Hill and Michael R. Marty. Amdahls law in the multicore era. *IEEE COMPUTER*, 2008.

[57] D. Luebke. Cuda: Scalable parallel programming for high-performance scientific computing. In *Biomedical Imaging: From Nano to Macro, 2008. ISBI 2008. 5th IEEE International Symposium on*, pages 836–838, 2008.

[58] Gyozo I. Egri, Zoltan Fodor, Christian Hoelbling, Sandor D. Katz, Daniel Nogradi, et al. Lattice qcd as a video game. *Comput.Phys.Commun.*, 177:631–639, 2007.

[59] Jianbin Fang, A.L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 216–225, 2011.

[60] R. Babich, M.A. Clark, and B. Joo. Parallelizing the quda library for multi-gpu calculations in lattice quantum chromodynamics. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, pages 1–11, 2010.

[61] D. Holmgren, N. Seenu, J. Simone, and A. Singh. Fermilab multicore and gpu-accelerated clusters for lattice qcd. *J.Phys.Conf.Ser.*, 396:042029, 2012.

[62] R. Babich, M. A. Clark, B. Joó, G. Shi, R. C. Brower, and S. Gottlieb. Scaling lattice qcd beyond 100 gpus. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 70:1–70:11. ACM, 2011.

[63] P. Sedcole, B. Blodget, T. Becker, J. Anderson, and P. Lysaght. Modular dynamic reconfiguration in virtex fpgas. *Computers and Digital Techniques, IEE Proceedings* -, 153(3):157–164, 2006.

[64] E.J. McDonald. Runtime fpga partial reconfiguration. In *Aerospace Conference, 2008 IEEE*, pages 1–7, 2008.

[65] Roman Lysecky and Frank Vahid. Design and implementation of a microblaze-based warp processor. *ACM Trans. Embed. Comput. Syst.*, 8(3):22:1–22:22, April 2009.

[66] J. W. Ruge and K. Stben. Multigrid methods. frontiers in applied mathematics. *SIAM Frontiers in Applied Mathematics*, 3:73130, 1987.

[67] C.C. Douglas. Multigrid methods in science and engineering. *Computational Science Engineering, IEEE*, 3(4):55–68, Winter.

[68] Bálint Joó and Mike A. Clark. Lattice qcd on gpu clusters, using the quda library and the chroma software system. *International Journal of High Performance Computing Applications*, 26(4):386–398, 2012.

[69] Martin Lüscher. Local coherence and deflation of the low quark modes in lattice qcd. *Journal of High Energy Physics*, 2007(07), 2007.

[70] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *IEEE International Parallel and Distributed Processing Symposium*, April 2008.

[71] J. Brannick, R.C. Brower, M.A. Clark, J.C. Osborn, and C. Rebbi. Adaptive multigrid algorithm for lattice qcd. *Phys.Rev.Lett.*, 100:041601, 2008.

[72] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[73] Martin Lücher. Deflation acceleration of lattice qcd simulations. *Journal of High Energy Physics*, 2007(12):011, 2007.

[74] Ronald Babich, James Brannick, Richard C. Brower, Michael A. Clark, Saul D. Cohen, et al. The role of multigrid algorithms for lqcd. *PoS*, LAT2009:031, 2009.

[75] Rüdiger Weiss. Minimization properties and short recurrences for krylov subspace methods. *Electronic Transactions on Numerical Analysis*, 2:57–75, 1994.

[76] Pedro V. Silva and João Gabriel Silva. Implementing mpi-2 extended collective operations. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 125–132. Springer-Verlag, 1999.

[77] Judith A. Vogel. Flexible bicg and flexible bi-cgstab for nonsymmetric linear systems. *Applied Mathematics and Computation*, 188(1):226 – 233, 2007.

[78] Daniel B. Szyld, Judith A. Vogel, Judith, and A. Vogel. Fqmr: A flexible quasi-minimal residual method with inexact preconditioning. *SIAM J. Sci. Comput*, 23: 363–380, 2001.

[79] H.Martin Bcker and Manfred Sauren. A parallel version of the quasi-minimal residual method based on coupled two-term recurrences. In Jerzy Waniewski, Jack Dongarra, Kaj Madsen, and Dorte Olesen, editors, *Applied Parallel Computing Industrial Computation and Optimization*, volume 1184 of *Lecture Notes in Computer Science*, pages 157–165. Springer Berlin Heidelberg, 1996.

[80] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Katherine Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 36:1–36:12. ACM, 2009.

[81] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *In Proceedings of IPDPS*, 2008.

[82] J. Chen, L. C. McInnes, and H. Zhang. Analysis and practical use of flexible bicgstab. *Preprint ANL/MCS-P3039-0912*, 2012.

[83] Yusuke Osaki and Ken-Ichi Ishikawa. Domain decomposition method on gpu cluster. *PoS*, LAT2010, 2010.

[84] Tsubasa Saito, Emiko Ishiwata, and Hidehiko Hasegawa. Analysis of the gcr method with mixed precision arithmetic using qupat. *Journal of Computational Science*, 3(3):87 – 91, 2012.

[85] Daisuke Aoto, Emiko Ishiwata, and Kuniyoshi Abe. A variable preconditioned gcr() method using the gsor method for singular and rectangular linear systems. *Journal of Computational and Applied Mathematics*, 234(3):703 – 712, 2010.

[86] P. Jiranek, M. Rozloznik, and M. H. Gutknecht. How to make simpler gmres and gcr more stable. Technical Report 2008-10, Seminar for Applied Mathematics, ETH Zürich, 2008.

[87] Richard P. Feynman. *Statistical mechanics: a set of lectures.* Frontiers in physics. Westview Press, 1972.

[88] P. R. Wallace. The band theory of graphite. *Phys. Rev.*, 71:622–634, May 1947.

[89] Joaquín E. Drut and Timo A. Lähde. Lattice field theory simulations of graphene. *Phys. Rev. B*, 79:165425, Apr 2009.

[90] Richard Brower, Claudio Rebbi, and David Schaich. Hybrid monte carlo simulation on the graphene hexagonal lattice. *PoS*, LATTICE2011:056, 2011.

[91] A. H. Castro Neto, F. Guinea, N. M. R. Peres, K. S. Novoselov, and A. K. Geim. The electronic properties of graphene. *Rev. Mod. Phys.*, 81:109–162, Jan 2009.

[92] P. V. Buividovich, E. V. Luschevskaya, O. V. Pavlovsky, M. I. Polikarpov, and M. V. Ulybyshev. Numerical study of the conductivity of graphene monolayer within the effective field theory approach. *Phys. Rev. B*, 86:045107, Jul 2012.

[93] Hagen Kleinert. *Path integrals in quantum mechanics, statistics, polymer physics, and financial markets; 5th ed.* World Scientific, 2009.

[94] Daniel Malko, Christian Neiss, and Andreas Görling. Two-dimensional materials with dirac cones: Graphynes containing heteroatoms. *Phys. Rev. B*, 86:045443, Jul 2012.

[95] Martin Lüscher. Solution of the dirac equation in lattice qcd using a domain decomposition method. *Computer Physics Communications*, 156v, number = 3, pages = v209 - 220, 2004.

[96] Walter Wilcox. Deflation methods in fermion inverters. *PoS*, LAT2007, 2007.

[97] Martin Lüscher. Lattice qcd and the schwarz alternating procedure. *Journal of High Energy Physics*, 2003(05):052, 2003.

[98] Klaus Grtner. Solving unsymmetric sparse systems of linear equations with pardiso. *Journal of Future Generation Computer Systems*, 20:475–487, 2004.

[99] Sivasankaran Rajamanickam, Erik G. Boman, and Michael A. Heroux. Shylu: A hybridhybrid solver for multicore platforms. In *Proc. of 26th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS12)*, 2012.

[100] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Physical Review*, 47(10):777–780, May 1935.

[101] Robert G. Edwards and Balint Joo. The chroma software system for lattice qcd. *Nucl.Phys.Proc.Suppl.*, 140:832, 2005.

[102] Guochun Shi, S. Gottlieb, A. Torok, and V. Kindratenko. Design of milc lattice qcd application for gpu clusters. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 363–371, 2011.

[103] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7), 1982.

[104] Richard P. Feynman and Albert R. Hibbs. *Quantum mechanics and path integrals.* International series in pure and applied physics. McGraw-Hill, 1965.

[105] B. Gaveau, T. Jacobson, M. Kac, and L. S. Schulman. Relativistic extension of the analogy between quantum mechanics and brownian motion. *Phys. Rev. Lett.*, 53:419–422, Jul 1984.

[106] Erwin Schrödinger. Uber die kraftefreie bewegung in der relativistischen quantenmechanik. *Sitz. Preuss. Akad. Wiss. Phys.-Math. KL*, 24:418–428, 1930.

[107] József Cserti and Gyula Dávid. Unified description of zitterbewegung for spintronic, graphene, and superconducting systems. *Phys. Rev. B*, 74:172305, Nov 2006.

[108] R. Gerritsma, G. Kirchmair, F. Zahringer, E. Solano, R. Blatt, and C. F. Roos. Quantum simulation of the dirac equation. *Nature*, 463(7277):68–71, Jan 2010.