Stephen P. Booth

October 2, 1995

Abstract

This thesis describes lattice gauge theories and discusses methods used to simulate them stochastically. The use of parallel computers for these simulations is discussed in depth.

Various pseudo-random number generator algorithms are reviewed and the implementation of these algorithms on parallel systems is investigated.

The strong-coupling phase transition of non-compact lattice QED is investigated. The phase diagram of strong-coupling non-compact lattice QED with an additional four-fermion interaction is deduced using a series of dynamical fermion simulations. The mass dependence of the system is investigated for non-compact QED and along the $\beta = 2.0$ axis, which is close to a system with only four-fermi interactions. These results are compared with solutions to the gap equation in order to determine if the data is consistent with a mean-field interpretation. An interpolation technique intended to improve the utilisation of the available data is investigated. The simulation program is also described in detail as a case study of a parallel implementation of a lattice gauge theory

The implementation of QCD on an i860 based parallel computer is described in depth. This includes a description of how code is optimised for the i860, an analysis of the time-critical portions of the code and a discussion of how these routines were implemented. Timings for these routines are given. Some results from these simulations are also presented.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Lattice gauge theories

1.1.1 Path Integrals

Quantum mechanics can be formulated in a variety of different ways. At first sight each formulation seems completely different from the others yet under closer examination they are all equivalent. One concept is common to all formulations, this is the idea of a complex probability amplitude.

In a classical system, probabilities are used to quantify our ignorance about the system. A classical probability is usually defined in terms of ratios of real numbers. The probability of a certain type of event is the ratio of the number of events of that type to the total number of events, in the limit where the number of events becomes infinite. These are intrinsically real quantities. A classical system is assumed to be in some specific though possibly unknown state. If we knew the state of the system exactly at any time it would be possible to calculate exactly all future states of the system. As we do not have complete information about a system, we have to assign probabilities to each of the possible resulting states. Now consider a system that passes through an intermediate state. If the probability of going from state A to state B is $P(A,B)$ then the probability of going from A to B via an intermediate state S is $P(A,S,B) = P(A,S)P(S,B)$. If there are a number of possible intermediate states then the total probability of going from A to B is the sum over the probabilities of going via each possible intermediate state,

$$P(A,B) = \sum_i P(A,S[i],B) = \sum_i P(A,S[i])P(S[i],B).$$

If the system passes through more than one intermediate state $S_1, S_2, \ldots S_N$ then $P(A,B)$ becomes;

$$P(A,B) = \sum_{i_1}\sum_{i_2}\ldots\sum_{i_N} P(A,S_1[i_1])[\prod_{\alpha=1}^{\alpha=N-1} P(S_\alpha[i_\alpha], S_{\alpha+1}[i_{\alpha+1}])]P(S_N[i_N],B). \tag{1.1}$$

That is, the probability $P(A,B)$ is the sum of the probabilities for each of the possible paths between A and B.

In a quantum mechanical system this kind of thinking breaks down. For example, if we think of photons passing through a two-slit apparatus, the probability of a photon arriving at a particular destination is no longer the simple sum of the probabilities from each of the slits by themselves, but is the squared modulus of the sum of the complex amplitudes from each of the slits. It is no longer reasonable to think of the system passing through a particular intermediate state. The real probabilities of the classical system must be replaced by complex amplitudes. Because these have a phase, it is possible for different alternatives to interfere with each other. The rules for combining complex amplitudes are the same as those for combining real probabilities.

The path integral formulation of quantum mechanics [1, 2] is a quantum generalisation of the classical path integral shown in equation 1.1. The complex amplitude going from an initial to a final state is given by the sum of the amplitudes of the possible intermediate paths. The amplitude of a path is given by

$$P_{path} = \kappa e^{\frac{i}{\hbar}S_{path}},$$

where S_{path} is the classical action of the path. The factor κ is chosen so that the sum of the probabilities over each of the possible outcomes is normalised to one. Lattice gauge theories are usually formulated using this path integral formulation of quantum mechanics.

Quantum field theories

The physics of high energy particles is currently thought to be best described using quantum field theories. As the name suggests these are quantum theories where the variables used to represent a system of particles form a field with values at all space-time points. This is in contrast to the original quantum mechanical description of particles where a particle is parametrised by its position or its momentum. The primary advantage of a field theory is that it is capable of describing many-particle systems where particles may be created and destroyed. It is also the only consistent way of constructing a relativistic quantum theory.

A field *configuration* is a possible state of the fields where the value of each field in the theory is defined at every point in space-time. In the path integral formulation, this is equivalent to a possible path of the system, so the integral over all possible paths becomes an integral over all possible field configurations. To calculate an observable quantity we must calculate the average of the observable over all possible field configurations with a complex weighting factor dependent on the action of the field configuration,

$$< O >= \frac{\sum_{config} O_{config} e^{\frac{i}{\hbar} S_{config}}}{\sum_{config} e^{\frac{i}{\hbar} S_{config}}}.$$

The complex weighting factor plays a similar role to the Boltzmann factor in statistical physics; it is much less convenient to work with though. The Boltzmann weight is a direct measure of how significant a particular configuration is. The quantum weights for different configurations only differ by a phase, so their relative contribution is a result of interference between the configurations and cannot be derived directly from the weighting factor. This causes several practical difficulties, especially when the path integral is approximated by a sum over a subset of configurations. The complex weight can be converted into the same decaying exponential form as the Boltzmann factor by formulating the theory in Euclidean space through performing the change of variables $t^{'} = it$.

1.1.2 Lattice field theories

In practice, it is usually not possible analytically to average over all possible field configurations. Because a field has an infinite number of degrees of freedom, this average involves an infinite number of integrations. It is therefore necessary to make a number of approximations. The first of these approximations is to replace spacetime by a finite box and second to replace the continuous spacetime with a lattice of discrete points. This reduces the problem to one with a finite number of degrees of freedom so the path integral is also reduced to a finite number of integrals. Any derivative of the fields that occurs in the action may be approximated using the difference of the field value at two lattice sites.

Gauge theories

A gauge theory is a field theory that contains a particular type of internal symmetry. A gauge theory is invariant under independent transformations of the fields at each point in space-time. This is very similar to the invariance of general relativity under transformations of the local coordinate system. In general relativity physics is independent of the choice of the local space-time coordinate system. In a gauge theory physics is independent of the choice of the local symmetry space coordinate system. This is very easy to arrange for local quantities but there is a difficulty in constructing a spatial derivative with this property, because a derivative is the limiting case of a difference between fields at different points in space-time:

$$\frac{d\psi(x)}{dx} = \lim_{\delta x \to 0} \frac{\psi(x+\delta x) - \psi(x-\delta x)}{2\delta x}.$$

If we consider gauge symmetries that are Lie groups, the ψ fields transform as

$$\psi'(x) = e^{igt^\alpha w^\alpha(x)}\psi(x),$$

where t^α is a generator of the group. The simple derivative will transform as

$$\frac{\partial\psi'(x)}{\partial x} = e^{igt^\alpha w^\alpha(x)}\frac{\partial\psi(x)}{\partial x} + igt^\alpha\frac{\partial w^\alpha(x)}{\partial x}\psi'(x).$$

We can construct a covariant derivative that respects the symmetry by introducing a gauge field $A^\alpha_\mu(x)$:

$$D_\mu = \partial_\mu - igt^\alpha A^\alpha_\mu(x).$$

Provided that $A^\alpha_\mu(x)$ transforms as

$$A^{\alpha'}_\mu(x) = A^\alpha_\mu(x) + \partial_\mu w^\alpha(x) - gf^{\alpha\beta\gamma}w^\beta(x)A^\gamma_\mu(x),$$

where f are the structure constants for the gauge group; the deviation in the simple derivative caused by a change in w is compensated by a change in the values of the gauge field A.

This is particularly obvious in theories defined on the lattice where derivatives remain as finite differences. On the lattice, a gauge field is represented by elements of the gauge group living on every link of the lattice. A link is a line that connects two neighbouring lattice sites. The group element of a link represents the transformation needed to convert between the parametrisations used at the two lattice sites. The lattice covariant derivative becomes:

$$D_\mu \psi(x) = \frac{U_\mu(x)\psi(x+a\hat{\mu}) - U_\mu^\dagger(x-a\hat{\mu})\psi(x-a\hat{\mu})}{2a},$$

where the U fields transform as;

$$U'_\mu(x) = (e^{igt^\alpha w^\alpha(x)})U_\mu(x)(e^{igt^\alpha w^\alpha(x+a\hat{\mu})})^{-1}.$$

The dynamics of the gauge field, $U_\mu(x)$, must also reflect the local symmetry; this can be achieved if the action for the gauge field is constructed out of quantities that are invariant under the symmetry, such as the trace of the product of link variables round a closed loop. Such an action was proposed by Wilson;

$$S_{Gauge} = \sum_{plaq} \beta[1 - (1/n)ReTrU_{plaq}],$$

where the sum is over all elementary squares of the lattice, '*plaquettes*', and U_{plaq} is the product of the link variables round the plaquette.

The symmetries of a gauge theory are particularly obvious when formulated on the lattice, but it is just as valid to formulate them as continuum field theories. The importance of the lattice is as a calculational tool. One of the original motivations for the development of lattice gauge theory was Quantum Chromo-Dynamics or QCD. This is the currently the best candidate for a theory of the strong nuclear force. It is a gauge theory based on the gauge group $SU(3)$ that couples several flavours of fermions called *Quarks*. One of the most interesting properties of QCD is asymptotic freedom; at high energies (or short distances) the effective coupling is very small. In this region it is possible to calculate observables using a power series in the effective coupling. This is perturbation theory. Each term in the power series refines the result. As the coupling constant is small in this region only a few terms of the series need to be calculated. The effective interaction becomes strong for low energy processes and an expansion in terms of the coupling constant is no longer a good approximation to the physics; there is therefore a limit to what can be calculated perturbatively. Lattice gauge theory was developed because it approximates the physics in a way that does not directly depend on the value of the coupling constant. It therefore stands a better chance of dealing with low energy QCD processes than perturbative methods.

1.1.3 Renormalisation

Quantum field theories are very prone to mathematical instabilities. Not all variables and parameters of a quantum field theory correspond to observable quantities. Observable quantities are constrained to remain finite by the physical properties of the theory, but there is no equivalent constraint for non-observable quantities. Treated naively, field theories will often produce divergent integrals. Renormalisation is the generic name given to the procedures used to control these mathematical instabilities. A renormalisation procedure starts by introducing a regularisation scheme that explicitly controls the mathematical divergences. For example, in perturbative expansions of QED the integral over the momenta of a fermion loop is divergent. This can be regulated by introducing an explicit momentum cut-off. Observable quantities are calculated with this regulator in place. The limit where the regulator is removed is then taken. In the case of the previous example, this is the limit where the momentum cut-off goes to infinity. In this limit, the intermediate non-physical quantities used in the calculation may become divergent but the physically observable quantities will remain finite and well defined. If this is not the case, then the theory is not renormalisable and cannot be used to describe physical processes. While the regulator is in place, the physically observable quantities are functions of the parameters of the field theory (bare couplings) and of the cut-off. If we hold the physical quantities fixed and vary the value of the cut-off then the couplings must vary in order to compensate. We can label each set of bare couplings using the corresponding set of observables at a particular value of the cut-off. A change in the value of the cut-off will produce a change in the position of these points in the space of the coupling constants. This

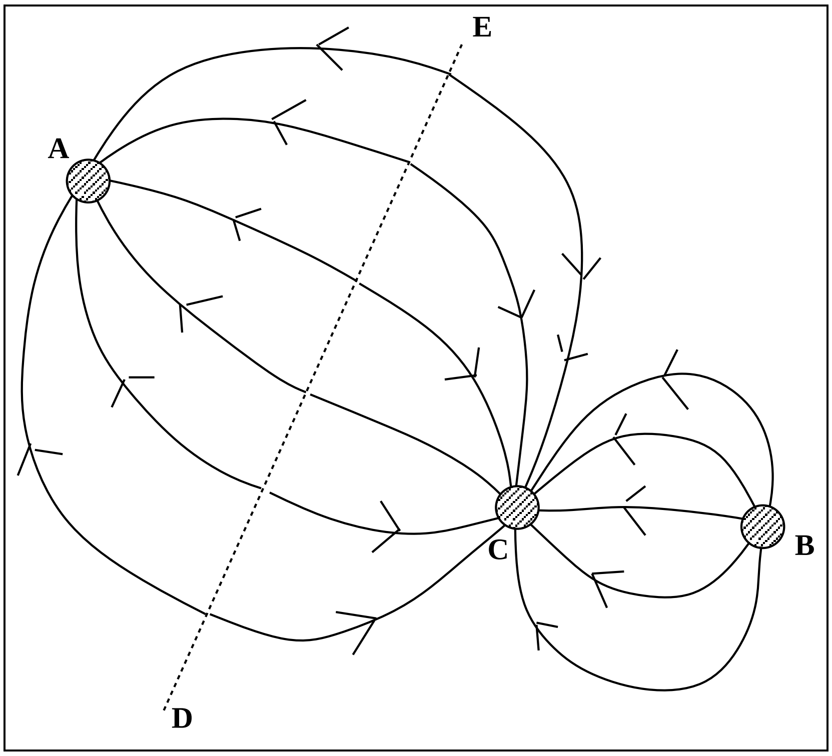


Figure 1.1: Renormalisation Group flow
Renormalisation trajectories must end on a fixed point. The points A and C represent ultra-violet stable fixed points, the point B and the line DE represent unstable fixed points.

movement is a renormalisation group transformation and the path of a single point in this space is a renormalisation trajectory. The values for the observables that are of physical interest are those where the cut-off has been totally removed. This occurs at the fixed points of the renormalisation group. A fixed point is a point in the space of coupling constants that maps to itself as the cut-off is changed (see figure 1.1).

In the lattice formulation, the regularisation is automatically provided by the lattice. A finite lattice of variables is incapable of representing a wave with a wavelength shorter than twice the spacing of the lattice. This introduces a momentum cut-off proportional to the inverse lattice spacing. In this case, there are actually two limits that have to be taken; the first is the thermodynamic limit, where the number of lattice points is taken to infinity and the second is the continuum limit where the regulator is removed and the lattice spacing is taken to zero. It is obviously not possible to perform a computer simulation in this limit so the best that we can hope for is to perform a number of simulations in a regime sufficiently close that we are able to extrapolate the results to the continuum limit. As the lattice spacing becomes zero in the continuum limit any physical process that propagates for a finite distance in the continuum must propagate for an infinite number of lattice spacings on the lattice. This means that the correlation length on the lattice must be infinite at the values of the bare coupling that correspond to the continuum result. An infinite correlation length is the indication of a second order phase transition. Therefore the fixed point corresponding to the continuum limit of a lattice theory must also correspond to a second order phase transition.

1.2 Stochastic simulation

By restricting the system to a finite box and approximating the fields by a lattice of points, a path integral is changed from an infinite to a finite but typically large number of integrations. In general, these integrals have to be calculated numerically. The number of integrals that need to be calculated is proportional to the number of lattice sites. It is impractical to perform all of these integrals for anything except a very small lattice. The alternative is to use some form of stochastic simulation. The integrals are a weighted average over all possible configurations of the system. A reasonable approximation to this average can be obtained by averaging over a representative sample of the possible configurations. When using this approximation the weighting factor for each configuration must be modified to account for the distribution used to choose the sample configurations. The most convenient and efficient approach is to choose configurations from a distribution proportional to the Boltzmann weight of the configurations, all the weighting factors become one and observables can be calculated as simple averages over the sample.

This leaves us with the problem of efficiently generating configurations from this particular probability distribution. One convenient method of doing this is a Markov chain. A Markov chain is an algorithm that generates a sequence of configurations by means of a transition probability from one configuration to the next which depends only on the previous configuration. Any configuration in the sequence is therefore correlated with the previous configuration. However, if the algorithm is constructed correctly, and the number of configurations is sufficiently large the sequence as a whole will have the statistics of the desired probability distribution. The procedure used to generate the next configuration in the Markov chain is called an *update* operation. Because of the correlations between successive configurations in the Markov chain it is usual to average over sets of configurations separated by a number of update steps. The desired probability distribution for configurations is

$$P(C) = \frac{1}{Z} e^{-S(C)}, \tag{1.2}$$

where $P(C)$ and $S(C)$ are the probability and action of a particular configuration C. The transition probability $W(C, C')$ is the probability that the update step will generate configuration C' from configuration C. The transition probability must map the distribution $P(C)$ onto itself:

$$P(C') = \sum_C P(C) W(C, C'), \tag{1.3}$$

for all configurations C'. If an update step satisfies equation 1.3 and it is guaranteed to eventually explore all possible configurations then it will generate configurations from the probability distribution $P(C)$. Most stochastic update schemes are based on the detailed balance condition:

$$P(C) W(C, C') = P(C') W(C', C). \tag{1.4}$$

This is a sufficient, though not necessary, condition for equation 1.3 to hold. Because the update step always results in a valid configuration,

$$\sum_B W(A, B) = 1.$$

If we sum equation 1.4 over all C we get,

$$\sum_C P(C') W(C', C) = P(C') = \sum_C P(C) W(C, C').$$

Therefore, providing the update procedure is capable of spanning the entire space of configurations, equation 1.4 is a sufficient condition for $W(C, C')$ to generate configurations from $P(C)$.

The detailed balance condition for equation 1.2 is;

$$W(C, C')e^{-S(C)} = W(C', C)e^{-S(C')}.$$

A good example of an update scheme based on this detailed balance condition is the Metropolis algorithm [3]. In its most general form, this algorithm consists of generating a trial configuration and then accepting or rejecting this change depending on the relative values of the action for the original and trial configurations. If the change is accepted the trial configuration becomes the next configuration in the sequence. If it is rejected the next configuration is the same as the previous one. If the probability of generating a particular trial configuration C_{trial} from a starting configuration C is $P_{trial}(C, C_{trial})$ and the acceptance probability is $P_{accept}(C, C_{trial})$, then the detailed balance condition will be obeyed if the following condition is enforced:

$$P_{trial}(A, B) = P_{trial}(B, A), \tag{1.5}$$

and the acceptance probability is

$$\begin{array}{ll} P_{accept}(A, B) = e^{S(A)-S(B)} & S(B) \geq S(A) \\ P_{accept}(A, B) = 1 & S(B) < S(A). \end{array}$$

Schemes of this type are very easy to implement. The efficiency of such a scheme is very dependent on the acceptance rate. If the trial solution is chosen completely at random, equation 1.5 will be satisfied and the sequence of configurations will eventually produce a good representative sample of the probability distribution. Very few of the trial configurations will be accepted so it will take a large number of update steps for this average to be achieved. It is important to use a selection method for the trial configurations that gives a high acceptance probability. If the trial configuration is only slightly different from the previous configuration, for example a single link reset randomly, then the difference in the action will be small and the probability of acceptance reasonably high. This will also mean that the successive configurations are very highly correlated and a large number of updates will be needed. An efficient stochastic algorithm will have to carefully balance the acceptance rate and the degree of correlations between successive configurations in order to maximise the rate at which the algorithm moves through the space of all possible configurations.

1.2.1 Fermions

Most of the physically important models in lattice gauge theory consist of fermionic fields interacting via gauge fields. The gauge fields are relatively straightforward to simulate. The fermionic fields introduce a large number of special problems. Fermions are best described using Grassmann variables. Grassmann variables are similar to normal real or complex variables except that they anti-commute: for any pair of Grassmann variables a and b, $ab = -ba$. This anti-commuting property automatically generates many of the properties of fermions such as the Pauli exclusion principle and anti-symmetry. Computers are not capable of using Grassmann variables directly, so it is necessary to use real variables and enforce anti-symmetry explicitly. The anti-symmetry condition for fermions is a global constraint on the fermion fields, so it is not generally possible to update part of the fermion field in isolation. All fermionic algorithms therefore involve global operations on the lattice variables. These usually take the form of inversions of the fermion matrix.

1.2.2 The fermion matrix

The path integrals that occur in lattice gauge theory are often of Gaussian form:

$$\int \mathcal{D}\psi \mathcal{D}\bar{\psi} e^{\sum_{x,y} \bar{\psi}(x) M_{x,y} \psi(y)}. \tag{1.6}$$

For example the lattice action for naive fermions is

$$S_{naive} = a^4 \sum_{x,y} \bar{\psi}(x) \left[\frac{1}{2a} \sum_{\mu} (\delta_{x+a\hat{\mu},y} U_\mu(x) - \delta_{x-a\hat{\mu},y} U^\dagger_\mu(y))\gamma_\mu + m\delta_{x,y} \right] \psi(y). \tag{1.7}$$

This is a direct translation to the lattice of the action for the Dirac equation. Naive fermions are rarely used in simulations because of fermion doubling. This is an unwanted result of using discrete spacetime inside a finite box. The dispersion relation for naive lattice fermions has more than one zero within the Brillouin zone so a single naive fermion field behaves like 16 flavours of fermions. There are other fermion lattice actions that reduce this problem, the staggered fermion action[4] and the Wilson fermion action[5]. The staggered fermion action is

$$S_{staggered} = a^4 \sum_{x,y} \bar{\chi}(x) \left[\frac{1}{2a} \sum_{\mu} \eta_\mu(x)(\delta_{x+a\hat{\mu},y} U_\mu(x) - \delta_{x-a\hat{\mu},y} U^\dagger_\mu(y))) + m\delta_{x,y} \right] \chi(y), \tag{1.8}$$

where χ and $\bar{\chi}$ are one component fermionic fields and,

$$\eta_\mu(x) = (-1)^{x_0 + \ldots + x_{\mu-1}}.$$

Naive lattice fermions have additional symmetries that are not found in the continuum. These additional symmetries allow the (free) action to be decoupled into $2^{\frac{d}{2}}$ parts. The staggered fermion action reduces the doubling problem by discarding all but one of these parts. The Wilson fermion action is

$$\begin{aligned} S_{Wilson} &= a^4 \sum_{x,y} \bar{\psi}(x) \left\{ \frac{1}{2a} \sum_{\mu} \left[(\gamma_\mu - r)\delta_{x+a\hat{\mu},y} U_\mu(x) - (\gamma_\mu + r)\delta_{x-a\hat{\mu},y} U^\dagger_\mu(y) \right] \right. \\ &\quad \left. + (m + \frac{4r}{a})\delta_{x,y} \right\} \psi(y). \end{aligned} \tag{1.9}$$

The Wilson action contains additional terms that increase the mass of the fermion doubles. This removes the fermion doubles but breaks chiral symmetry. All of these different actions have Gaussian form. If ψ is a complex variable then a Gaussian integral (equation 1.6) is proportional to the determinant of the matrix M^{-1}, whereas if ψ is a Grassmann variable the integral is proportional to the determinant of M. It is therefore possible to simulate fermions using real variables at the expense of calculating matrix inverses. Because the matrix is a function of the gauge fields, this inversion must in principle be carried out every time the gauge fields are updated.

1.2.3 Hybrid Monte Carlo

Hybrid Monte Carlo [6] is currently one of the most efficient algorithms for simulating dynamical fermions. Hybrid Monte Carlo is based on the Metropolis algorithm described earlier in this chapter; its efficiency lies in the method used to generate a new trial configuration. The field variables are treated as the coordinates of a classical mechanics system, a new trial configuration is generated by evolving this system in time; this is a new "computer" time τ, completely unrelated to the time dimension of the fields. A Hamiltonian is imposed on the system:

$$H'(\phi,\pi) \equiv \frac{1}{2}\pi^2 + S'(\phi),$$

where ϕ represents the field variables and S' is an action for the field variables. This action is usually chosen to be the same as the action being simulated. The initial values of the momenta π are chosen randomly from a Gaussian distribution,

$$P_G(\pi) \propto \exp[-\pi^2/2].$$

This system is now evolved a distance in τ to generate the trial configuration. This new configuration is accepted with a probability

$$P_{accept} = \min(1, \exp[\delta H]),$$

where

$$H(\phi,\pi) \equiv \frac{1}{2}\pi^2 + S(\phi),$$

and S is the action being simulated. As H' is the energy of the system it should be conserved if the time evolution is computed exactly. In the case where $H' \equiv H$, this would give us $P_{accept} \equiv 1$. This is the *hybrid* algorithm. It is only possible to integrate the equations of motion using discrete steps in τ. This gives rise to errors in the dynamics which means the hybrid algorithm is inexact. In the Hybrid Monte Carlo algorithm these error reduce the acceptance rate below 1 but the calculation remains exact. Reducing the size of these steps always reduces the discretisation errors and therefore increases the acceptance rate. For the detailed balance condition to hold the integration must be done in a reversible fashion: if the momenta π are reversed at the end of the evolution the same evolution procedure must return the system to its original state. A reversible integration of the equations of motion can be achieved using the leapfrog algorithm.

If the equations are to be evolved for a distance τ_{max}, the leapfrog algorithm starts with an initial half-step in the momenta:

$$\pi(\frac{\delta\tau}{2}) = \pi(0) - \frac{\partial S(0)}{\partial\phi}\frac{\delta\tau}{2},$$

this is followed by $n = \tau_{max}/\delta\tau$ steps in ϕ and $n-1$ steps in π, of the form

$$\phi(\tau+\delta\tau) \quad = \quad \phi(\tau) + \pi(\tau + \frac{\delta\tau}{2})\delta\tau$$

$$\pi(\tau + \frac{\delta\tau}{2}) \quad = \quad \pi(\tau - \frac{\delta\tau}{2}) - \frac{\partial S(\tau)}{\partial \phi}\delta\tau,$$

and finally a second half-step in π:

$$\pi(\tau_{max}) = \pi(\tau_{max} - \frac{\delta\tau}{2}) - \frac{\partial S(\tau_{max})}{\partial \phi}\frac{\delta\tau}{2}.$$

This algorithm can easily be extended to include fermionic fields. In a dynamical fermion simulation the probability distribution for the gauge fields becomes;

$$\begin{aligned} P(\phi) &= \frac{1}{Z}\int[d\bar{\psi}][d\psi]\exp[-S(\phi) - \bar{\psi}M\psi] \\ &= \frac{1}{Z'}\det(M)\exp[-S(\phi)] \\ &= \frac{1}{Z''}\int[d\chi^*][d\chi]\exp[-S(\phi) - \chi^*(M^\dagger M)^{-1}\chi]. \end{aligned}$$

The χ fields are bosonic pseudo-fermions that only exist on the even sites of the lattice. The Hamiltonian becomes

$$H(\phi,\pi) = \tfrac{1}{2}\pi^2 + S(\phi) + \chi^*(M^\dagger M)^{-1}\chi,$$

and the equations of motion are

$$\begin{aligned} \dot{\phi} &= \pi \\ \dot{\pi} &= -\frac{\partial S}{\partial \phi} - \chi^*\frac{\partial (M^\dagger M)^{-1}}{\partial \phi}\chi. \end{aligned}$$

The fermion contribution to the momentum can be simplified using the identity

$$AA^{-1} = 1$$

so that

$$\begin{aligned} \frac{\partial A}{\partial \phi}A^{-1} + A\frac{\partial A^{-1}}{\partial \phi} &= 0 \\ A\frac{\partial A^{-1}}{\partial \phi} &= -\frac{\partial A}{\partial \phi}A^{-1} \\ \frac{\partial A^{-1}}{\partial \phi} &= -A^{-1}\frac{\partial A}{\partial \phi}A^{-1}. \end{aligned}$$

The equations of motion can therefore be written as;

$$\begin{aligned}\dot{\phi} &= \pi \\ \dot{\pi} &= -\frac{\partial S}{\partial \phi} + \chi^*(M^\dagger M)^{-1}[M^\dagger \frac{\partial M}{\partial \phi} + (\frac{\partial M^\dagger}{\partial \phi})M](M^\dagger M)^{-1}\chi.\end{aligned}$$

The χ fields are held fixed during the integration and updated by an exact heat-bath, $\chi = M\eta$ where η is Gaussian noise, before the first half-step. Because the matrix M is a function of ϕ a matrix inversion step is needed every time the momenta are updated to calculate $(M^\dagger M)^{-1}\chi$. This procedure will remain reversible even if the matrix inversion is not exact provided that the inversion algorithm depends only on ϕ and χ. If the solution of the previous timestep is used as an initial guess then the matrix inversion must be exact to preserve reversibility.

It remains to be shown that reversibility is a sufficient condition for the detailed balance condition to hold. The probability distribution we wish to sample is

$$P_S = \frac{1}{Z}\exp[-S(\phi)].$$

The τ evolution of the fields is deterministic and defines a mapping on the phase space. This means that the probability Y of choosing the candidate phase space "configuration" (ϕ', π') is

$$Y((\phi, \pi) \mapsto (\phi', \pi')) = \delta[(\phi', \pi') - (\phi(\tau_{max}), \pi(\tau_{max}))].$$

The acceptance probability A is

$$A((\phi, \pi) \mapsto (\phi', \pi')) = \min(1, \exp(H(\phi', \pi') - H(\phi, \pi))).$$

The transition probabilities for the ϕ fields is given by

$$W(\phi, \phi') = \int [d\pi][d\pi']P_G(\pi)Y((\phi, \pi) \mapsto (\phi', \pi'))A((\phi, \pi) \mapsto (\phi', \pi')). \tag{1.10}$$

The dynamics are reversible so

$$Y((\phi, \pi) \mapsto (\phi', \pi')) = Y((\phi', -\pi') \mapsto (\phi, -\pi)). \tag{1.11}$$

The form of P_G is such that $P_S(\phi)P_G(\pi) \propto \exp[H(\phi, \pi)]$ and H is invariant under $\pi \mapsto -\pi$. By using the identity

$$\begin{aligned}\exp[-H(\phi, \pi)]\min(1, \exp[-\delta H]) &= \min(\exp[-H(\phi, \pi)], \exp[-H(\phi', \pi')]) \\ &= \exp[-H(\phi, \pi)]\min(\exp[\delta H], 1),\end{aligned}$$

we get

$$\begin{aligned} P_S(\phi)P_G(\pi)A((\phi,\pi)\mapsto(\phi',\pi')) &= P_S(\phi')P_G(\pi')A((\phi',\pi')\mapsto(\phi,\pi)) \\ &= P_S(\phi')P_G(-\pi')A((\phi',-\pi')\mapsto(\phi,-\pi)). \end{aligned}$$

Multiplying by Y, integrating over π and π' and using equation 1.11 we get

$$\int[d\pi][d\pi']P_S(\phi)P_G(\pi)Y((\phi,\pi)\mapsto(\phi',\pi'))A((\phi,\pi)\mapsto(\phi',\pi')) = \\ \int[d\pi][d\pi']P_S(\phi')P_G(-\pi')Y((\phi',-\pi')\mapsto(\phi,-\pi))A((\phi',-\pi')\mapsto(\phi,-\pi)).$$

Using equation 1.10 and the invariance of the measure $[d\pi][d\pi'] = [d(-\pi)][d(-\pi')]$ this becomes

$$P_S(\phi)W(\phi,\phi') = P_S(\phi')W(\phi',\phi),$$

which is the detailed balance condition.

1.3 Parallel computers

The main problem with lattice gauge theory is that it requires a great deal of computing power to perform realistic calculations. Further, one of the main sources of systematic error in these calculations is the lattice size. In order to control this systematic error it is necessary to perform simulations using a variety of different sized lattices and to compare the results. With current algorithms and computer power it is not possible to have a large enough range of lattice sizes for simulations involving dynamical fermions. Hence a further approximation is made known as the quenched approximation. This is where the contribution of fermion loops is ignored and is equivalent to making the fermions infinitely massive. The primary reason for making this approximation is that there is a very large saving in the requirement for computer time. Because the fermions have an infinite mass, the fermionic fields need not be updated during the simulation. This saves on all of the fermion matrix inversions except for those used to calculate observables.

Current calculations show little difference between the results of quenched and dynamical QCD simulations. It is therefore quite common to exploit the available computing power to simulate a larger lattice rather than to perform a dynamical simulation. However, this approximation is hard to justify physically, so the correspondence between quenched and dynamical simulations will need to be rechecked as the quality of simulations improves. Even in the quenched approximation, the majority of the available computer time is spent inverting the fermion matrix in order to calculate observables that involve the fermion fields.

The path integral has been approximated using Monte Carlo techniques. This has introduced a further source of error, the statistical error in the Monte Carlo sampling. As before, this error can only be reduced at the cost of more computer time, in this case to increase the sampling statistics.

The cost in computer time for a full dynamical simulation becomes quite staggering. Even with the most powerful computers available lattice simulations are nearly always limited by the available computer power. As computers become faster the demand for computer power has also increased as it becomes possible to reduce the systematic errors in the calculation. The lattice gauge theory community has therefore always had a vested interest in the progress of computer technology. In short, the lattice gauge calculations of QCD form one of the most computationally intensive problems currently being researched. The computational power needed is far beyond that normally available. It is therefore necessary to build very high performance computers dedicated to the problems of lattice gauge theory. Over the last few decades electronic computers have been steadily increasing in performance. A large part of this increase has been due to smaller and faster circuits that enable computers with faster cycle times to be built. Unfortunately this approach is starting to run into some fundamental physical limitations that will eventually place a lower limit on the cycle time of a computer. The cycle time of a processor is limited by the speed of propagation of electrical signals across the chip. This can only be reduced by shrinking the physical size of the circuit. At some stage the size of the individual components starts to approach the atomic scale. Currently chips are being constructed using ~ 1 um technology. Assuming that structures ~ 10 nm across cannot behave like a macroscopic crystal the speed of a single processor is unlikely to increase by much more than a factor of 100.

Parallel computation provides a method of circumventing these physical limitations and continuing to increase the available computer power. The basic approach used in all parallel computers is to divide problems into independent subproblems and to use separate hardware components to calculate a number of them simultaneously. The time taken to calculate each subproblem is still limited by the speed of the hardware components, but the total throughput of the computer is increased. An additional advantage of parallel computers is that they can be very cost effective. In principle, a large number of processors with a modest individual performance can be combined to produce a single computer with a very high total performance. To a first approximation, the price of such a machine is dominated by the cost of its component parts. These components can be mass-produced microprocessors using commonly available manufacturing techniques. On the other hand, an equivalent single processor would be built in small numbers and would have to use state-of-the-art techniques. Consider now how to double the power of each machine. The parallel computer is doubled in size by doubling the number of processors and approximately doubling the cost. The communication system is likely to cost more than double the original, depending on how it is implemented; many machines these days use some form of $n \log n$ interconnect network and the component cost of the network should scale in a similar fashion. The single processor would have to rely on more exotic technologies and extreme measures such as supercooling. This will increase the cost by far more than a factor of two, especially as this will require further research and development. Ignoring fault tolerance, a scaled-up parallel computer can be constructed simply by using a larger number of the original components. Once the power of a computer starts to strain the physical limitations on the speed of a single processor, equivalent parallel computers can be expected to be consistently cheaper.

There are several very different ways that parallelism can be attempted, each approach has different strengths and weaknesses. Some of these approaches have already achieved a wide degree of acceptance. For example, the vector supercomputer can be thought of as a type of parallel computer.

In order to implement a particular problem in parallel, it is first necessary to identify some inherent parallelism in the problem. Once this has been done, there are still various constraints on how this parallelism can be exploited. These constraints arise from the nature of the problem and the nature of the parallel hardware available. Some problems may only be able to run efficiently on partic-

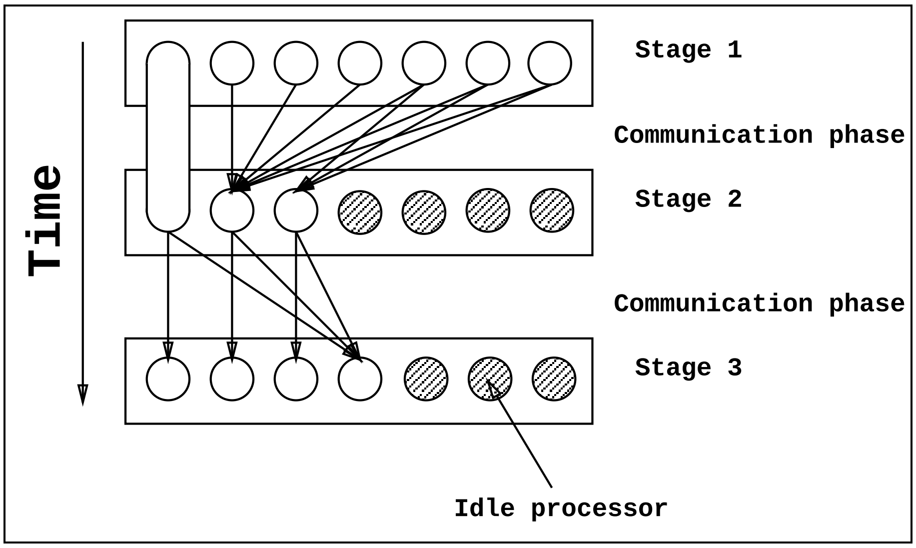


Figure 1.2: Decomposition of a program into parallel subprograms
A program may be divided into sequentially executed stages each consisting of a number of independent subprograms. Communications occur between these stages.

ular types of parallel computer. The majority of parallel computer architectures have local memory storage associated with each of the processors. In this case, it can be expensive to move data from one processor to another. In general, a program can be thought of as consisting of several sequentially executed stages, each of which may have some inherent parallelism and can be divided into independent subproblems. These subproblems can be distributed over the available processors and calculated in parallel. The results from one sequential stage may be used as input for some of the future sequential stages. In general, this requires a communication stage to redistribute the data. A subprogram need only synchronise itself with the rest of the program when it needs to communicate. If the result of a subprogram is not needed in the next stage it can in principle continue executing across a number of calculation stages; see figure 1.2. Some computers are capable of overlapping data communication with calculation. This model of parallel computation can still be applied though the communication stage is now used to initiate communications or to wait for their completion.

When considering how to divide up a problem into parallel subproblems (a decomposition scheme) there are two useful characterisations that can be made, complexity and data-coupling.

The complexity of the subprogram is whether the parallelism is at the level of individual arithmetic operations, simple procedures or entire subprograms. This is a measure of how often communication stages occur. A program that uses a parallel bubble sort has a low complexity. The subproblems only consists of swapping two numbers depending on their relative values. Other problems only have high complexity parallelism. For example, a parallel chess program can evaluate a number of possible board positions in parallel though there is little parallelism at any lower level. The more complex the subproblem is, the longer it will take to compute each subproblem and the less frequently the processors will have to communicate.

Data-coupling is a measure of how much data has to be moved in the communication stages. If most of the results of one stage are required by most of the subprograms in the next stage then the problem is strongly data-coupled. If most of the intermediate data can be left on its processor of origin then the problem is only loosely data-coupled. In shared memory machines, where all of the data is available to all of the processors this seems not to be a problem. However, unless some form of distributed memory system is used, a shared memory machine will be limited by the speed of its single memory bank; in a shared memory machine with multiple memory systems the problem of processor to processor communication is replaced by a memory to processor communication problem.

A typical loosely-coupled problem is a task farm. In this case, the problem consists of a number of completely independent problems; no data need be exchanged at all. The processors only need to communicate with the outside world when loading their initial data and when outputting their results. This is the simplest form of parallelism, but it is also the most efficient. There are many applications that can be successfully implemented as a task farm. One example is analysing the events produced by a HEP experiment. The analysis of a single event has virtually no inherent parallelism and cannot even be vectorised. However, because of the large number of events that need analysing it is possible to use a task farm [7, 8]. In lattice gauge theory, a task farm approach can be used to explore the phase diagram of a theory. Instead of attempting to implement the simulation code in parallel, each processor runs an independent simulation with different coupling values in order to span the phase diagram[9].

An example of a tightly coupled problem is a fast Fourier transform. In each of the n stages of a 2^n point fast Fourier transform, two numbers from the previous stage are combined to give a pair of numbers for the current stage. This has an obvious $n/2$ parallelism. Unfortunately, the pairs of numbers combined are never the same in two different stages, so at least half of the data has to be exchanged in every stage.

In lattice gauge theory, the complexity and the data coupling of the problem are connected. The basic form of parallelism in lattice gauge theory is a geometric parallelism. Most of the data is located at the points of the lattice; the operations that modify this data in general only depend on the values of data at each point, and at the neighbouring lattice points. There is an obvious way to implement this in parallel; regions of the lattice are allocated to separate processors. If a single lattice point is allocated to a single processor, then the complexity of the subproblem is quite low and the data coupling is quite quite high because all of the results from each step will have to be communicated to neighbouring processors. If a region containing a large number of lattice points is allocated to a processor then the complexity is quite high (proportional to the volume of the region), but the data coupling is increased by a much smaller amount (proportional to the surface area of the region). This means that lattice gauge theory can be performed on a wide variety of parallel machines provided the decomposition scheme is tuned to match the computation/communication abilities of the architecture.

There are two main limitations that must be considered. The first is a limitation on the minimum size of the lattice. It is obvious that a given lattice cannot be efficiently simulated on a parallel machine with more processors than the lattice has sites. Once this limit is reached, the size of the lattice must be increased along with the size of the machine otherwise efficiency will be lost. Unfortunately, the computer power needed to perform the simulation increases faster than the power of the machine. This is because the dominant mathematical operation is a matrix inversion, the difficulty of which does not scale linearly with the size of the system. Using the Hybrid Monte Carlo algorithm the number of floating point operations needed to generate independent configurations N_f has been estimated to scale as $N_f \sim V^{10.5}$ where V is the size of the system [10]. A parallel computer designed for lattice gauge theory is therefore likely to need the individual processors to be quite powerful. The second limitation comes from the ratio between the amount of communication and calculation that is needed. This is a simple surface-area-to-volume ratio that depends on the number of lattice sites assigned to each of the processors. As the size of the local lattice is decreased, the fraction of time spent performing communications increases. The maximum sustainable communication-to-calculation ratio varies greatly between different designs of parallel machines. On some machines there may be a minimum local lattice size such that additional communications will make the calculation go slower if more processors are added to the problem. On other machines it may be perfectly feasible to go all the way down to a single lattice site per processor. This limit also depends on which theory is to be simulated. QCD is based on $SU(3)$ multiplication which requires 198 floating point operations to multiply a single two-spinor by a gauge element. QED on the other hand, is based on the $U(1)$ gauge group and only requires 6 floating point operations to perform a similar operation. On the other hand, the required communication will only change by a factor of 3 going from QED to QCD. This means that QCD can run efficiently on computers with a lower communication to calculation ratio than is possible for QED. In general, the complexity and price of the communication system increases with the number of processors the system is able to support. So there are price benefits to making a machine with a smaller number of high performance processors.

1.3.1 Hardware

In general, a parallel computer can be thought of as a communication system connecting separate hardware components. Parallel computers have been constructed in an enormous variety of ways; any system invented to classify them will inevitably end up grouping some very different architectures together. For the purpose of this discussion, I intend to concentrate on few selected types of parallel computer and how their hardware impacts on the software environment.

Vector processors

Vector computers take advantage of the fact that most programs will repeat a set of simple operations on a large number of different pieces of data. They exploit parallelism at the lowest possible level, that of individual arithmetic operations (low complexity decomposition). A vector computer uses a number of processing elements connected in a pipeline to perform these simple operations in parallel. Though it exploits parallelism internally, a vector processor attempts to behave as much as possible like a very fast single processor. The vector pipelines can be thought of as a number of separate (though highly specialised) processing elements. Most vector machines have special vector registers to hold intermediate results and reduce the need to access the external memory system. If a vector machine is thought of as a parallel computer, the data pathways between the vector pipelines and these vector registers form the communication system. Because the parallelism is introduced at such a low level, a level that is already under the control of the compiler, it is possible to automate most of the effort involved in porting a program to a vector machine. Vector machines have the advantage that they are relatively easy to program and vectorisable programs are easily portable between different types of vector processor. In general, vector processors are programmed in conventional sequential programming languages. The compiler analyses this code and looks for loops where each iteration of the loop is independent of the results of all the other loop iterations. Each of these loops is suitable for vectorisation (vectorisable) and the compiler produces code to execute them in parallel. This means that the same programs that were written for vector machines can be run without modification on a sequential processor. Even though the same languages are used for sequential and vector computers, different programming styles are needed to get the maximum performance out of the two types of machine. In order to obtain the maximum performance from a vector processor, a program must have been written, or re-written, to contain as many vectorisable sections of code as possible. Otherwise the vector hardware will only be used for a small fraction of the code. This is often done at the expense of introducing large vectors of temporary variables. On sequential machines, vectorisable codes may run more slowly than a code written without vectorisation in mind. This is because of the vectors of temporary variables. On a vector machine, these variables can be easily implemented using the vector registers. On a sequential machine the vectors have to live in the main memory. There are two ways this can slow the program down. The first is that this may require more memory than is available. In this case the computer will have to continually swap data from the memory onto disk, a very wasteful and time consuming operation. The second potential slowdown is that the vectors will prevent caching from working properly. A cache is a region of particularly fast memory that automatically retains the values of recently accessed variables on the assumption that they are likely to be required again in the near future. Where these variables are reused a short time later the cache is able to provide the values much more rapidly than they could be fetched from main memory. As the temporary vectors are very large and each variable in them is only used in a single loop iteration, none of these values will be reused before the cache replaces them with more recent data. If you consider disk storage to just be a slower form of memory, excessive cache misses and excessive swapping are much the same thing. At the moment, all parallel computers share this problem in one form or another. Programs have to be modified in different ways to take full advantage of the particular parallel hardware being used. Of course, these disadvantages can be overcome by using a smarter compiler that is capable of re-writing vectorisable code to run efficiently on the scalar machine. Most types of parallel computer are usually more difficult to program than vector processors.

Data coupling is not an issue for vector machines, because the vector pipeline is thought of as a single fast processor and is attached to a single memory system. By definition, there is never any data dependency between the parallel sections of a vectorisable loop, so no communication is needed within a loop. At the end of each loop all the resulting data is written out to the main memory so there is never any danger of not having data accessible when needed. The price paid for this simplicity is the need to use a fast, sophisticated and expensive memory system. The maximum performance of a conventional vector processor is limited by the maximum performance of this memory system, though it is possible to use vector technology to construct high performance processing nodes for a distributed memory parallel computer.

SIMD processing

SIMD stands for Single Instruction stream Multiple Data stream. A SIMD computer consists of a large number of identical processing elements (PEs). All of the elements are constrained to perform the same operations at the same time as all of the other elements. Instructions are broadcast from a central controlling processor. An individual processing element is only capable of performing operations on data stored in its local data store. The data structures for a problem are distributed over the processing elements. If an operation requires data stored on a different PE then a communication must take place. Because all of the PEs perform the same operations at the same time, all of the PEs must simultaneously perform input and output operations when data is moved between PEs. The basic communication operation is therefore a shift, see figure 1.3.

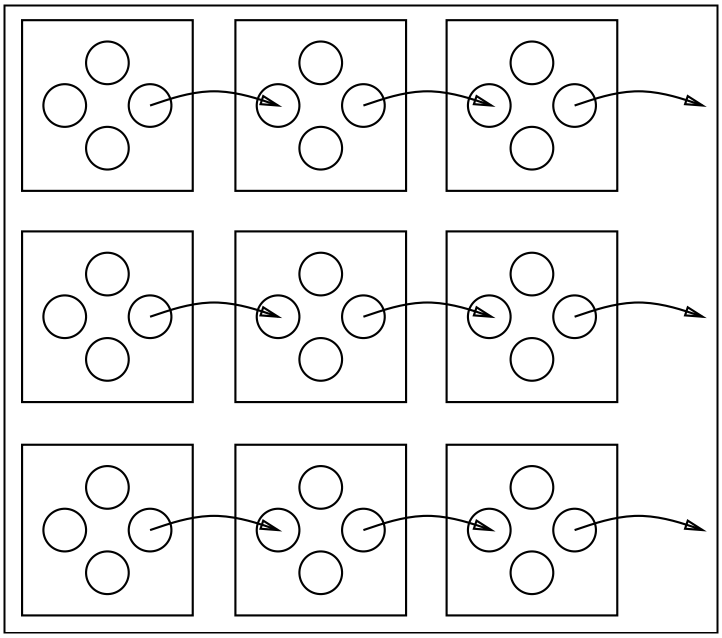


Figure 1.3: Dataflow in a SIMD architecture.
The communication pattern in a SIMD architecture is a data shift where every processors outputs some data at the same time as receiving some

SIMD computers have many similarities with vector processors. The main difference between vector processors and SIMD machines is that SIMD machines use distributed memory. This overcomes the memory bandwidth limitation of vector machines and enables SIMD machines to be scaled up to use very large numbers of processors. A good example of this is the Connection Machine [11]. The SIMD model is a direct hardware implementation of the Data-parallel programming model[1]. Much of the success of this type of computer is a reflection of the success of the data-parallel programming model.

[1] described in a later section

MIMD processing

MIMD stands for Multiple Instruction stream Multiple Data stream. This is a more flexible kind of parallel computer consisting of a number of independent processors that are capable of communicating with each other in some fashion. Each processor can be running different pieces of code, and even when there are multiple copies of the same piece of code, separate processors may be executing different sections of the code at any one time. Each processor has its own local data store that can only be accessed directly by that processor. MIMD computers can exploit parallelism at much higher level (more complex subproblems) than vector or SIMD machines. MIMD computers are therefore much more flexible, for example it is not possible to implement a simple task-farm on vector or SIMD machines. The parallelism is at the program rather than the instruction level. A MIMD computer can also be programmed using the data-parallel programming model. All that this requires a compiler that is capable of translating a single data-parallel program into a set of communicating subprograms. There are also computer languages such as OCCAM [12] that are specially designed for MIMD processing. These languages have inter-processor communications built into the language. It is also possible to program MIMD computers in conventional sequential languages. In this case the inter-processor communications must be invoked as procedure calls. A separate program could be provided for each of the processors. It is more usual to have have most of the processors running the same code but having responsibility for different regions of the data. If this code is written carefully, so that it is easy to change the number of processors, it is still possible to run the code on a sequential machine. This is just the special case of $n_{processors} = 1$.

The Transputer

Transputers are a family of microprocessors purpose built for MIMD parallel processing applications. Each Transputer has four bi-directional communication links that can be used to connect it to other Transputers (see figure 1.4). In addition they are specially designed to support multi-tasking. They have very few registers, so there is very little data to be saved/loaded when the processor switches between different tasks (a context switch). When one process is suspended, for example while it is waiting for a communication to finish, the decision about which of the other currently active processes is to be run is made by hardware on the chip. This is in contrast with multi-tasking systems on other microprocessors which usually have to run some supervisor program to make these decisions. This means that it is very simple and efficient to have several very simple sub-programs running on a single Transputer at the same time. On other microprocessors the added delays of context switching makes multi-tasking very inefficient, especially for small lightweight jobs. This is important for parallel processing because of the inter-processor communications. Processors have to cooperate with each other when moving data in order to ensure that the data is up to date and ready to be sent. It is very desirable for the processors to be able to switch quickly from their own calculations to handle data from other processors, so that the other processors will not be delayed.

All of the work presented in this thesis has been performed on MIMD computers constructed using T800 Transputers. Two different types of machine were used: a purely Transputer based machine that uses T800 transputers to perform all of the computation, and a combined i860/Transputer machine constructed out of composite nodes, each containing two Transputers and a single i860. In the i860 machine, all of the calculations are performed using the i860 processor and the Transputers are only responsible for inter-processor communication. These two machines are significantly different and need to be programmed in different ways. The T800 Transputer has a sustained floating-point performance of approximately 1 Mflops. Using hand coded assembly language, the i860 can sustain approximately 30 Mflops. The composite processing node is therefore 30 times faster than a single Transputer but has only twice the communication bandwidth. A Transputer is capable of using its communication links at the same time as it is performing calculations. The on-chip memory enables it to perform calculations without using all of its available memory bandwidth. It is therefore possible to perform a certain amount of inter-processor communications at the same time as calculations, without affecting the performance of the processor. These two operations use separate parts of the processor and only interfere when they both attempt to access memory at the same time. It is therefore highly desirable to attempt to overlap the communications with some other task that can usefully use the processor. The i860 on the other hand, usually saturates memory bandwidth only running calculations, so there are no memory cycles free for overlapped

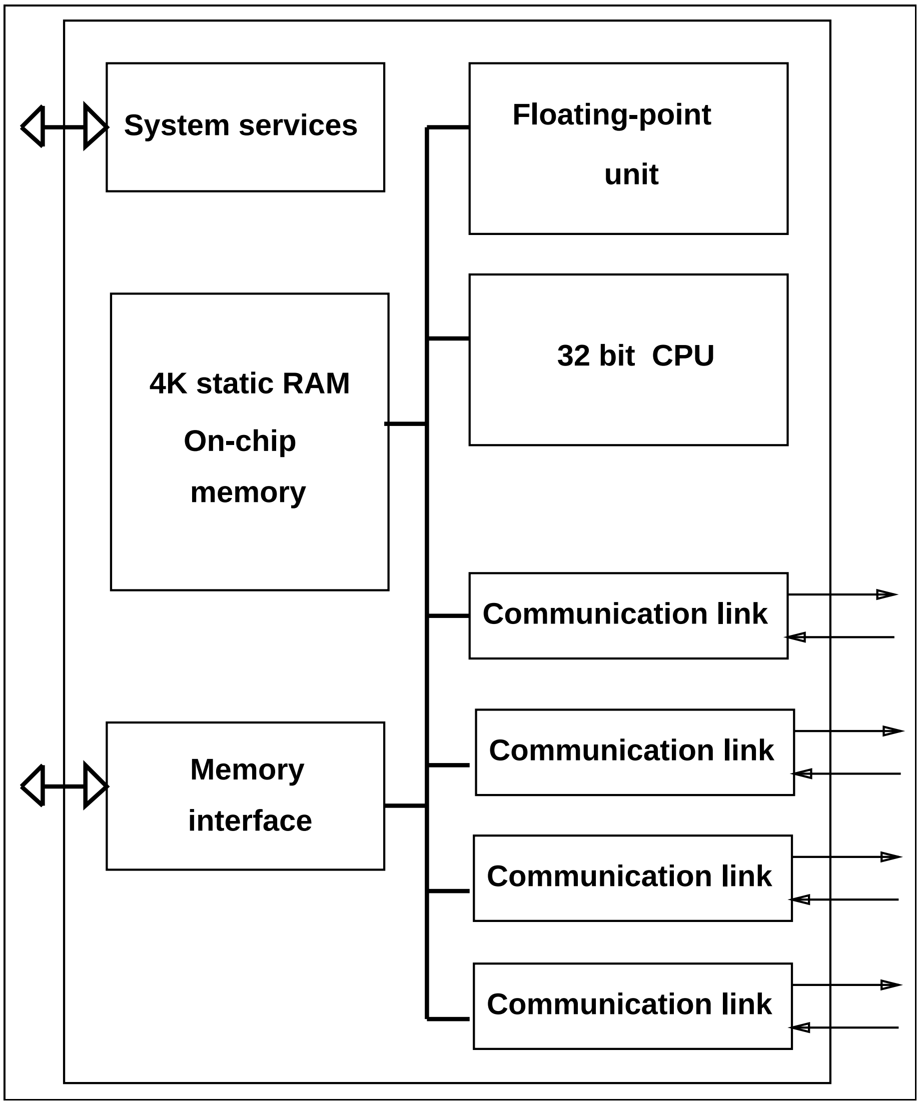


Figure 1.4: The Inmos T800 Transputer.

communications. In both cases, it is desirable to reduce the amount of communication as much as possible. The main disadvantage of constructing a MIMD computer out of T800 transputers is that the communication links are point to point connections and there are only four of them. In order to send a message to a processor that is not connected directly to the local processor the message must be forwarded by the intermediate processors. This makes T800 code very complex as it often has to include large amounts of code whose only purpose is to forward messages. This can also have an impact on the performance of the intermediate processors. The next generation of transputers is supposed to implement communication using a separate routing chip. These routing chips can be connected to processors and other routing chips, when a processor wishes to communicate with another processor is passes the message to the nearest routing chip. The routing chips then make all of the decisions about how to get the message to its destination and the message is passed between routing chips without having to pass through any processors other than the sender and receiver of the message.

1.3.2 Programming models

Sequential code

Most existing programs have been written in conventional sequential languages without any thought to parallelism. It would be very nice if these programs could be automatically ported to parallel computers using compiler technology. This is the approach used for vector machines, even though a different programming style is needed to get the most out of a vector processor. This is a very difficult trick to achieve. The task is much easier for a vector machine than for other types of parallel processor because at the end of a vectorised loop all the results are written out to a single memory system so the compiler does not have to try and match subprograms to the processors containing appropriate data, and each loop can therefore be compiled completely separately.

Data parallel

Data parallel programming languages are usually extensions of conventional programming languages. These extensions usually take the form of operations that operate on entire arrays rather than individual variables. Similar extensions have now been incorporated into the new Fortran-90 standard [13]. This makes the parallelism in the code much more explicit and has the beneficial side effect of making the programs clearer and easier to write. For example, the following fragment of sequential code:

```
DO I = 1,NX
  DO J = 1,NY
    DO K = 1,NZ
      A(I,J,K) = (14.0 * B(I,J,K)) + C(I,J,K)
    ENDDO
  ENDDO
ENDO
```

can be replaced by the single statement:

```
A = (14.0 * B) + C
```

The compiler now has to allocate elements of these arrays to different processors in such a way as to minimise the amount of communication needed. A reasonable heuristic for achieving this is to make sure that equivalent elements from arrays of the same shape are always allocated to the same processor. As with vector processors, the compiler can only work well if the program is written in a particular style. It is quite possible to automatically convert vectorisable programs into data parallel programs, though the differences in programming style mean that the results are liable to be less than perfect. Data parallel languages have usually been implemented on SIMD machines. It is much easier to compile code where the size of the arrays matches the number of processors exactly; there is then no need to generate different code to take account of array elements at processor boundaries. Some languages force the programmer to obey this restriction so the problem must be manually decomposed into arrays of the correct size, for example in DAP Fortran [14]. In other cases the computer emulates a computer of the required size by implementing a number of “virtual processors” on each real processor, for example CM Fortran[15]. Most existing data parallel compilers therefore tend to produce a low complexity highly data-coupled decomposition which requires a high performance communication system. In SIMD machines, it is common to assume that the most frequent form of communication is a data shift along array axes and to optimise the compiler and communication system for these operations. A more efficient use of the communication system can probably be achieved by allocating regions of the arrays to separate processors and treating the boundary points separately. This would require much more complex compilers, but may be necessary to implement data parallel languages on general purpose MIMD machines where the communication system is less specialised. Most of the complexity in designing a good data parallel compiler is due to the need to reduce the amount of communication; on a hardware platform with a high performance communication system the compilers can afford to be much simpler.

Explicit message passing

This is the programming environment most commonly found on distributed memory MIMD computers. A separate program is written for each processor and data is communicated between processors using explicit communication function calls or language constructs. This is a highly flexible programming model, but it can often be very difficult to port existing code to this kind of software environment. This difficulty can often be reduced by structuring the program, so that it behaves like one of the simpler programming models such as a task farm or a data parallel program.

1.4 Lattice gauge theories in parallel

Parallel computing can be used for lattice gauge theory. The parallelism in the problem is a direct consequence of the physics being simulated. The local nature of the interactions together with translational invariance means that any operations that acts on the lattice variables can be constructed out of local operations, and these local operations will be the same for all points on the lattice. The data-parallel programming model is therefore ideally suited to lattice gauge theory. They can also be simulated in parallel using a geometric decomposition and explicit message passing. The lattice is divided up into space-time regions and each region is assigned to a different processor. Apart from the lattice points on the boundaries between regions, each processor will be able independently to update its own part of the lattice. In order to update the boundary points, information must be communicated between the processors that share a common boundary. In an MIMD computer running an explicit message passing environment, the simulation code that runs on each of the processors is almost identical to a conventional sequential simulation of a smaller lattice. The only difference is in the boundary conditions for the lattice.

Gather-scatter

In order to update a single lattice point, data is required from neighbouring lattice sites. Lattice points on the boundary may have to request this data from some other processor. It is possible to separate lattice sites into internal sites and boundary sites and to write different code to handle each case. This makes the code very complex and difficult to write. It also means that there is twice as much code to optimise. It is much simpler to divide the data collection into separate routines. Outside of these routines all of the sites can then be treated in a similar fashion. On a single processor, data collection can be performed using gather-scatter operations, see figure 1.5. A gather operation has the form:

$$result[i] = source[table[i]]$$

A scatter operation has the form:

$$result[table[i]] = source[i]$$

If we desire to copy data from one array to another, shifting the position of the data by a single lattice site, four steps are needed.

- Internal data points must be copied between the arrays.
- Local points needed by other processors must be collected.
- Boundary points are exchanged between processors.

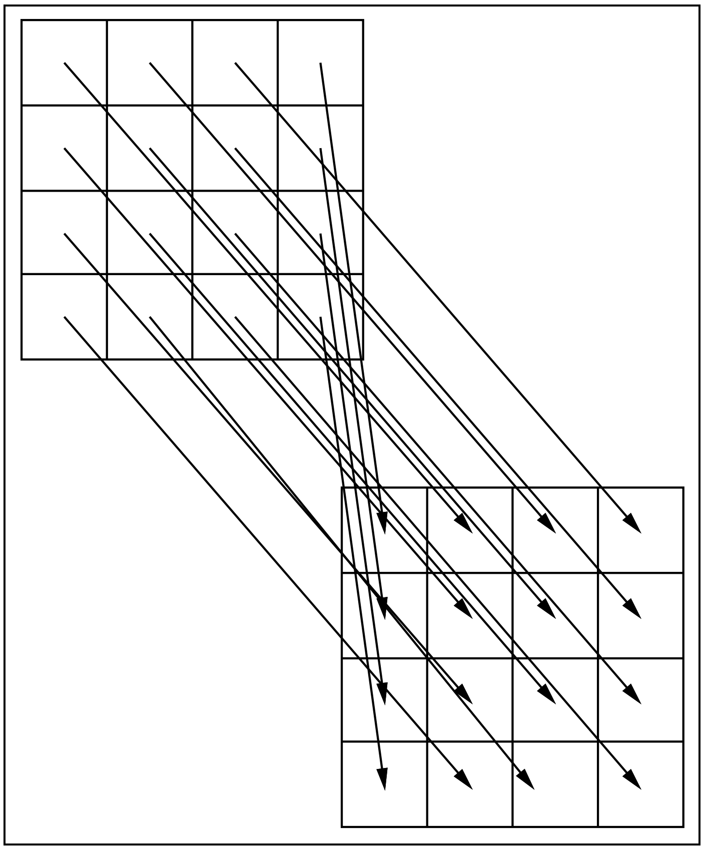


Figure 1.5: Data collection on a single processor.

On a single processor all of the data collection can be performed with a single gather-scatter operation.

- The points imported from other processors are copied into the final array.

The first two stages both involve a scatter operation. It is possible to combine these stages into a single operation at the expense of storage space. The destination array must be made a little larger, large enough to hold the local data and an additional set of boundary data. A single scatter operation can then be used to copy the internal lattice points, and to collect the boundary points into the extra "tail" at the end of the array; see figure 1.6. This "tail" can then be used as a communication buffer to transfer the boundary points to the appropriate processor. The use of gather-scatter operations in this manner simplifies the code to a great degree. The disadvantage of this approach is the extra time taken to copy the local data between the arrays. As the data was already accessible in the original array this data shuffling seems like a waste of time. In most cases it is possible to avoid this problem by combining the gather-scatter operation with part of the calculation. Instead of just being moved from one array to another the indirect addressing is inserted into one of the existing operations.

$$shift[i] = source1[table[i]]$$

$$result[i] = f(source2[i], shift[i])$$

becomes

$$result[i] = f(source2[i], source1[table[i]])$$

This removes all of the extra memory access cycles except those used to read the table. If the gather-scatter table is implementing a shift by one lattice site, as is usually the case for lattice gauge theory, it is possible to reduce these memory cycles as well. In this case, the table will contain large sections where the indices are in sequence. These sections need only be stored as a pair of numbers, the first number in the sequence and its length. This form of table is only applicable if the processor is capable of calculating the sequence faster than it can read a number in from the table.

Specific examples of how lattice gauge theories can be implemented on distributed memory parallel computers are given in later chapters. A dynamical fermion simulation of QED using a T800 based MIMD computer is described in chapter 3. The implementation of QCD codes on a composite i860/T800 computer is described in chapter 4.

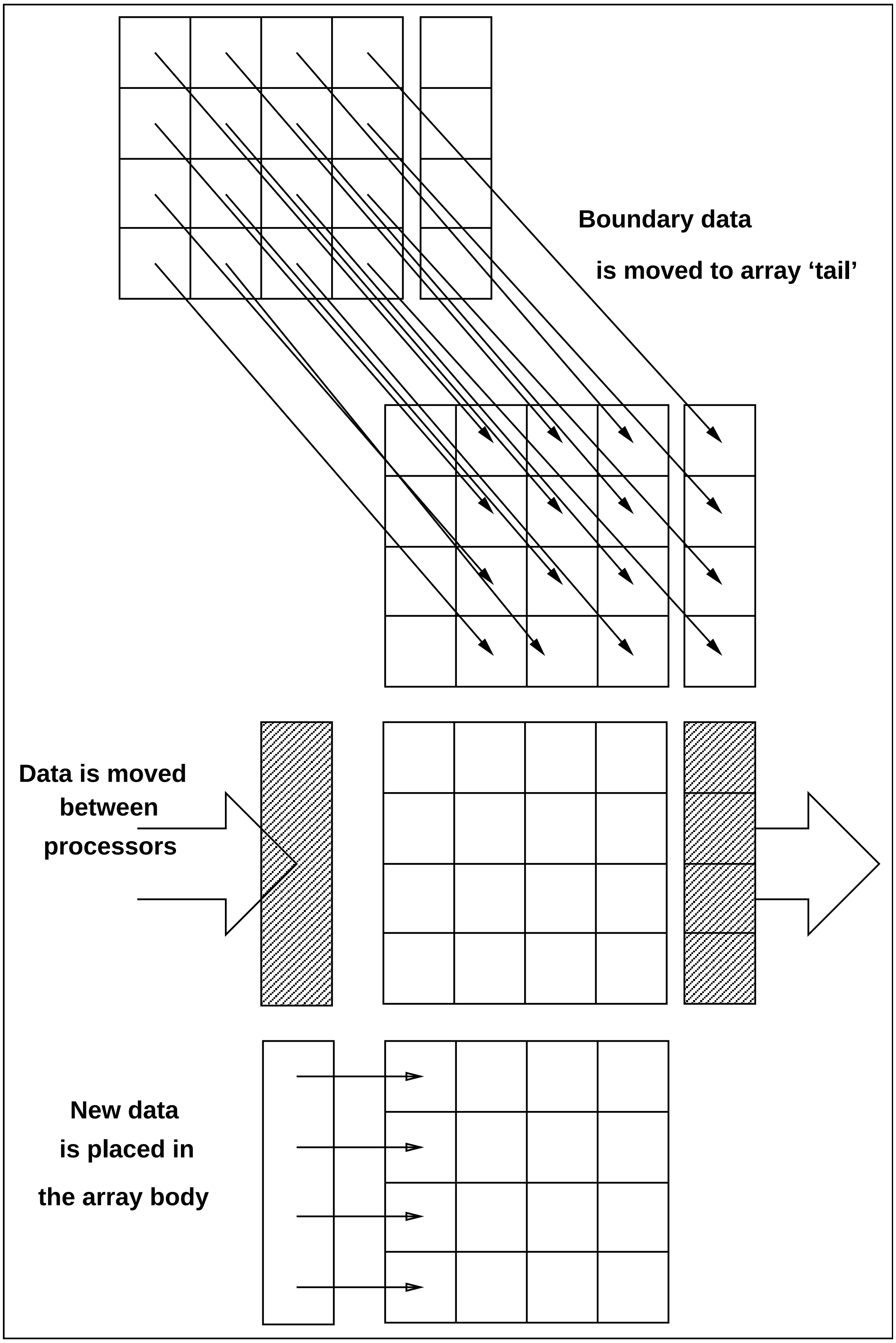


Figure 1.6: Data collection on a parallel processor.

On a parallel machine data collection requires two gather-scatter operations and an inter-processor communication.

Chapter 2

Random number Generators

All of computational particle physics, and especially lattice gauge theory, makes heavy use of Monte-Carlo algorithms. By definition a Monte-Carlo algorithm makes use of random numbers. Unfortunately true random numbers are hard to produce. It is possible to produce random numbers using specialised hardware. For example, amplifying the electronic noise of a reverse-biased diode. This is not really a practical approach. This kind of random number hardware is hard to build without any bias and true random numbers make programs very hard to debug because it is impossible to repeat exactly any single program run. The usual solution to this problem is to use a pseudo-random number generator. This is an iterative algorithm that generates a sequence of numbers that shares enough of the statistical properties of a true sequence of random numbers to be usable in a Monte-Carlo algorithm. There are of course always some correlations in the sequence, the iterative nature of the algorithm ensures this. However, if the algorithm is carefully chosen, these correlations are unlikely to effect the results of the simulation.

All pseudo-random number algorithms share a number of common features. The *state* of a random number generator is defined by the value of the set of variables, *state variables*, that are preserved between calls to the random number generator. These variables are often referred to as the random number seeds. It is less confusing if this term is reserved for the variables used to initialise the state of the random number generator at the start of the program. In a lot of cases the seed and the state variables will be the same. Each time the random number generator is called, the state variables are transformed using an *update transformation* that changes the state of the generator. The next number in the sequence is generated from the new state variables using an *output function*. There is not necessarily a one-to-one mapping between a number from the sequence and the corresponding state of the generator. Because there is only a finite number of possible states the sequence must eventually repeat itself. The number of iterations that make up one such repeat is the *period* of the generator. If T is the update transformation, $\tau = \{T^1, T^2, T^3, \ldots T^N\}$ forms a cyclic group, where N is the period of the generator, and the state variables form a representation of the group.

2.1 What is random ?

As was mentioned in the previous section a pseudo-random sequence is only an approximation to a true random sequence. It is therefore essential to be able to make some quantitative statements about how close to random a particular sequence is. Any potential random number generator should have its output tested using a series of statistical tests [16]. The choice of which statistical test to use is rather arbitrary. However a large body of experience has been built up in this area [16, 17]. In addition to empirical tests it is also possible to look at the theoretical properties of the random number sequence [16]. This analysis is usually done in terms of the frequency of occurrence of numbers, number-pairs, triples etc. A number sequence is *k-distributed* if all possible sequences of k numbers are equally likely in the number sequence.

For real numbers between 0 and 1 the definition of k-distribution is

$$Pr(\mu_1 \leq U_n < \nu_1, \ldots, \mu_k \leq U_{n+k-1} < \nu_k) = (\nu_1 - \mu_1) \ldots (\nu_k - \mu_k),$$

for all choices of real numbers μ_i, ν_i, with $0 \leq \mu_i < \nu_i \leq 1$ for $1 \leq i \leq k$.

For b-ary numbers, positive integers less than a maximum value b, the definition of k-distribution is

$$Pr(X_n X_{n+1} \ldots X_{n+k+1} = x_1 x_2 \ldots x_k) = 1/b^k,$$

for all b-ary numbers $x_1 x_2 \dots x_k$.

The distribution number of a sequence is the largest integer k for which the sequence is k-distributed. If a sequence of numbers is k-distributed then it will also be n-distributed for all n: $0 < n < k$. A reasonable definition of a truly random sequence is a sequence that is ∞-distributed. Different applications will require different statistical properties from a random number generator. However it is reasonable to assume that the statistics of short sequences of numbers are of particular importance. For example, pairs of numbers may be used to generate random numbers from a Gaussian distribution, or a series of numbers may be used in generating a single random gauge element. The information content of the state variables puts a limit on the distribution number of the generator. If $N(k)$ is the number of possible k-tuples, k_{max} is the distribution number of the sequence and M is the number of different states that map onto any given number then $M \geq N(k_{max} - 1)$. This is because there are $N(k-1)$ k-tuples beginning with the particular number. As each state only occurs once in a single period there must be $N(k-1)$ states that map onto that number for the sequence to be k-distributed. The value of M is therefore important as a limit on the distribution number of a generator. The output function is usually kept as simple as possible so that the program will run quickly. In most commonly used generators the state variables are either the same as the most recent number produced or are a history of the last p numbers produced. In this last case $p \geq k_{max}$. If the value of M cannot be directly calculated, an estimate of its value can be made using the period of the generator. If P is the period of the generator and μ is the number of possible output values, 2^{32} for a 32 bit number, then by assuming that all such values are equally likely, ie the generator is 1-distributed, we can show that $M \leq P/\mu$.

Some algorithms, for example linear congruential generators, [1] have better statistical properties in the higher order bits of the number sequence. If the low order bits are discarded to produce numbers with a lower resolution the distribution number can be increased. By discarding the low order bits, the output function of the algorithm has been changed, which has increased the value of M and therefore relaxed the limit on the distribution number.

[1] see later section

The lattice structure of a generator is one of the simplest empirical tests of a random number generator. Successive k-tuples from the number sequence are taken to be points in a k-dimensional space. A two dimensional cross section of this space is plotted and examined to see how well the plane is covered. A plot consisting of widely separated lines of points is an indication that the k-tuples are not random. There is a strong connection between the distribution number of a number sequence and its lattice structure. If k is less than the distribution number of the sequence the lattice plot is guaranteed to be evenly covered, because each of the possible k-tuples that make up the plot is equally likely.

2.2 Parallel random number generators

The problems associated with the design of good random number generators have been known for some time and a body of useful algorithms has slowly been built up. Most of this work has been for conventional single processors. The purpose of this chapter is to investigate the use of these algorithms in a parallel computing environment. Because of the difficulty of designing a high quality random number generator, I will restrict the discussion to parallel versions of well known single processor algorithms.

Parallel computation can introduce a number of complications to the design of a random number generator. In a parallel computer consisting of a number of separate processors, there will have to be a separate sequence of random numbers generated on each processor. These sequences will all be used for the same simulation. This means that correlations between different sequences will have to be prevented as well as correlations within each sequence. There are a number of approaches that can be used to get round this problem. In principle it is possible to use a separate algorithm, or variations of a single algorithm for each of the processors. The amount of work needed to test a new algorithm makes this option rather difficult to achieve in practice. This approach is also intrinsically dangerous. A problem with the random numbers on a small fraction of the processors in a simulation is enough to invalidate the results, but such a problem would be much harder to find than one where all the processors shared the same problem.

Parallel random number generators are usually modified versions of a single processor random number generator. This usually involves distributing the single processor number sequence over the processors. This can be done in two ways.

The first method is to put each successive number from the sequence on a separate processor, *successive distribution*. The potential problem in this case is that the sequence produced on each of the individual processors is of a much poorer quality than that produced by the original algorithm. If the original algorithm had a distribution number of k_{max} the local sequence produced on a n processor machine can only have a distribution number of k_{max}/n. This is only a problem if the statistics of the local sequence are more important than the correlations between the processors. As a rough rule of thumb, this depends on how fine grained the parallelism is. On a coarse-grained system where separate processors are responsible for relatively independent parts of the calculation then successive distribution is obviously a bad idea. For a fine-grained system, such as a vector processor or certain types of SIMD architecture, this is less of a problem. In these cases the random numbers are likely to be used in much the same way as they would have been in a single processor sequential machine. Even in this case it would still be necessary to check the application for any code that explicitly uses a set of random numbers at the same time, for example Gaussian random numbers or random gauge elements.

The remaining option is to use the same conventional algorithm on all of the processors and to select the starting state for each processor in such a way as to minimise the correlations between the processors, *block distribution*. On distributed memory MIMD machines it seems reasonable to assume that the statistics of the local sequence are far more important than the correlations between processors. If the system being simulated has local interactions then short distance correlations, between neighbouring points, are more important than long distance ones. This kind of problem is usually implemented on distributed memory MIMD machines using a geometric decomposition. In this case most neighbour-pairs will not cross processor boundaries, so correlations local to a processor are more importants than those between processors. A block distribution scheme is appropriate in this case, because the local sequence has the same statistical properties as the original sequential algorithm.

2.3 Linear Congruential Generators

Linear congruential generators are one of the most widely used forms of random number generator. They generate a sequence of integers, $x_0, x_1, x_2, x_3, \ldots$ using the transformation:

$$x_{i+1} \equiv (ax_i + b) \bmod m,\ 0 < x_i < m$$

Linear congruential generators have been around for a long time and are very easy to implement. This means that their behaviour is well understood. Most of the default generators provided with computer systems are of this type. The quality of the number sequence is very dependent on the value of the constants a, b and m. Luckily there is a large body of literature on this subject [16]. The small number of state variables, a single integer, severely limits the quality of the resulting number sequence. The integers x_i cannot be greater than 1-distributed. A good linear congruential generator is constructed by choosing the constants a, b and m so that the high order bits of the output have a high distribution number.

Any element of the sequence can be generated using:

$$x_n \equiv (a^n x_0 + \frac{a^n - 1}{a - 1} b) \bmod m$$

The relation:

$$x_{n+p} \equiv (a^p x_n + \frac{a^p - 1}{a - 1} b) \bmod m$$

shows that a parallel implementation using successive distribution is equivalent to changing the values of a and b. This will almost always have disastrous consequences for the number sequences produced by individual processors.

When using block distribution each copy of the generator must have a different value for its starting seed. A parallel implementation needs to provide some algorithm for selecting these starting seeds so as to reduce correlations between the processors. If a pair of generators have starting seeds, x_0 and y_0 such that $y_0 = x_0 + K$ then the two sequences will be related by:

$$y_n \equiv (x_n + a^n K) \bmod m$$

This shows one potential problem that should be avoided in a multi-processor calculation. The difference between the results of the two generators depends only on the initial difference between their seeds, and on the iteration number. Therefore if a set of generators have starting seeds with equal spaces between them then the sequences will remain equally spaced mod$m$. This suggests that in order to use a linear congruential generator in parallel it is necessary to ensure that the starting seeds for each processor do not have equal spaces between them.

2.4 Linear recurrence generators

Linear recurrence sequences are a central part of many random number generators. The general form of these sequences is:

$$X_n = (a_0 + a_1 X_{n-1} + \ldots + a_k X_{n-k}) \bmod p \tag{2.12}$$

The linear congruential generator described earlier is obviously a special case of this algorithm. We can set the constant $a_0 = 0$ without any loss of generality by noting that:

$$X_{n+1} = (a_0 + a_1 X_n + \ldots + a_k X_{n-(k-1)}) \bmod p,$$

substitute for a_0 using equation 2.12;

$$X_{n+1} = ((1 + a_1)X_n + \ldots + (a_k - a_{k-1})X_{n-(k-1)} - a_k X_{n-k}) \bmod p,$$

$$X_n = ((1 + a_1)X_{n-1} + \ldots + (a_k - a_{k-1})X_{n-k} - a_k X_{n-(k+1)}) \bmod p.$$

The general form of a linear recurrence relation is therefore:

$$X_n = (a_1 X_{n-1} + \ldots + a_k X_{n-k}) \bmod p. \tag{2.13}$$

The theory of finite fields [18] shows that if p is prime it is possible to find values for $a_1, a_2 \ldots a_k$ such that the period of the sequence is $p^k - 1$. If this is the case the sequence X_n is going to be a very good source of random integers in the range $[0, p)$. The state of the generator is equivalent to a k-tuple from the number sequence. With the exception of the zero tuple that maps to itself under the update transformation all possible k-tuples occur once during the period of the generator. This means that as near as to make no difference the sequence is k-distributed. The problem is that it is very difficult to find suitable values for the constants a. The constants a must be the coefficients of a primitive polynomial $\bmod p$. The most efficient known test to see if a particular polynomial is primitive involves finding the prime factors of $(p^k - 1)/(p - 1)$. For large values of p and k this becomes impractical [19]. When k is large and a large number of the constants a_i are non zero then this kind of generator will probably be too slow for Monte Carlo simulations.

A number of the algorithms discussed in the following sections are special cases of the one defined in equation 2.13, so it is worthwhile looking at some of the properties of this kind of sequence.

The state of the generator is most easily coded as a vector of k integers in the range $[0, p)$ that stores the last k numbers in the sequence $X_{n-1}, \ldots X_{n-k}$.

The zero tuple $X_{n-1} = \ldots = X_{n-k} = 0$ always maps onto itself regardless of the values of p or the values of the constants $\{a_0, \ldots, a_k\}$. The maximum possible period for the sequence is therefore $p^k - 1$. The maximum possible distribution number is $k - 1$, or k if we are prepared to ignore the non-occurrence of the zero k-tuple.

The update transformation T can be written as a matrix acting on the state vector. For example when $k = 5$.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ a_5 & a_4 & a_3 & a_2 & a_1 \end{pmatrix} \begin{pmatrix} X_{n-4} \\ X_{n-3} \\ X_{n-2} \\ X_{n-1} \\ X_n \end{pmatrix} \bmod p = \begin{pmatrix} X_{n-3} \\ X_{n-2} \\ X_{n-1} \\ X_n \\ X_{n+1} \end{pmatrix} \tag{2.14}$$

The transformation T^n is represented by the matrix T raised to the power n using arithmetic mod p. Compared to the simple update rule given in equation 2.13, performing the update using the matrix T is very expensive in terms of computer time and memory. This matrix form has the advantage if a large number of updates is to be performed in a single step, for example as a way of separating a number of parallel generators in a block distributed scheme. The size of the period for this kind of sequence can be so large that it can take years [2] for repeated application of equation 2.13 to traverse even a small fraction of the sequence. Once the number of iterations becomes large it is quicker to calculate the appropriate transformation matrix. Transformation matrices can be combined using matrix multiplication.

$$T^m \times T^n = T^{m+n}$$

This multiplication takes the same time regardless of the values of m and n . If x is a number in the range $[0, 2^{q+1} - 1)$ and

$$x = \sum_{i=0}^{q} s_i 2^i$$

$$s_i \epsilon \{0, 1\}$$

then the matrix T^x can be calculated using.

$$T^x = \prod_{i=0}^{q} (T^{2^i})^{s_i} \tag{2.15}$$

It takes q matrix multiplies to create the set of matrices T^{2^i} by repeated squaring. So the matrix for any value of x can be generated in at most $2q$ matrix operations. For really large powers of T the same approach can be used but decomposing x using a higher base than 2. This enables us to generate a matrix for any number of iterations within a reasonable length of time.

The update transformation is a *linear* transformation. Any linear combination of initial state vectors (seed vectors) will result in a state vector for the sequence that is the same linear combination of the sequences generated by the original state vectors. This gives us a constraint on the use of this kind of generator in parallel when using a block distribution scheme. If the state vectors on a number of processors are linearly dependent then the sequences produce will show the same dependence. It is therefore advantageous to choose the initial state vectors to be linearly independent. This can only be achieved if $n_{proc} \leq k$ where n_{proc} is the number of generators running in parallel and k is the number of values stored in the state vector. This is another reason to choose a generator with a large state vector. Though the ideal situation is to have all of the state vectors linearly independent this is probably a far stronger condition than is needed for most types of

[2] or multiples of the age of the universe

simulation. It will usually be sufficient to ensure that simple relations between two or three sequences are avoided. There is another possible correlation that is much harder to guard against. If we have to prevent correlations within short distances along each individual sequence and between numbers generated simultaneously by separate sequences we should also be concerned with correlations between the numbers from two sequences generated at slightly different times.

Algorithms of this kind are not suited to successive distribution on distributed memory parallel machines. Either the state of the generator will be distributed over a number of processors, requiring a communication step to perform an update, or the generator will have to generate the full sequence on all processors and select the appropriate elements from this sequence; thus preventing the speed of the generator from scaling with number of processors.

In order to use a block distribution we must find a way of reducing the correlations between processors as much as possible. We have to avoid having any two processors close together in the number sequence. In addition we have identified linear dependence between state vectors as an undesirable correlation. Because the space of valid state-vectors is very large, a set of randomly chosen state-vectors has a very high probability of being a good starting place for the generators in a parallel simulation. Ideally such a set of state-vectors should be checked to make sure they are all linearly independent of each other. An alternative approach is to generate a transformation matrix for a very large number of updates, choose the initial state for the first processor at random and generate the initial states for all the other processors by repeated applications of the transformation matrix. The transformation matrix can be precalculated to reduce the startup time for the generator. Any linear dependence between the processors is independent of the choice for the initial state of the first processor, so this can be checked when the matrix is calculated.

2.4.1 Shift register generators.

Linear recurrence generators with $p = 2$ are of particular interest. Primitive polynomials mod 2 are relatively easy to find because the value to be factored into primes is relatively small. In this case the algorithm becomes the shift register algorithm used for generating a sequence of random bits. The name comes from a simple method of implementing the algorithm in hardware using a shift register and xor gates (see figure 2.1). Addition mod2 is the same as the exclusive-or operation. The exclusive-or operation is also found in the instruction set of most

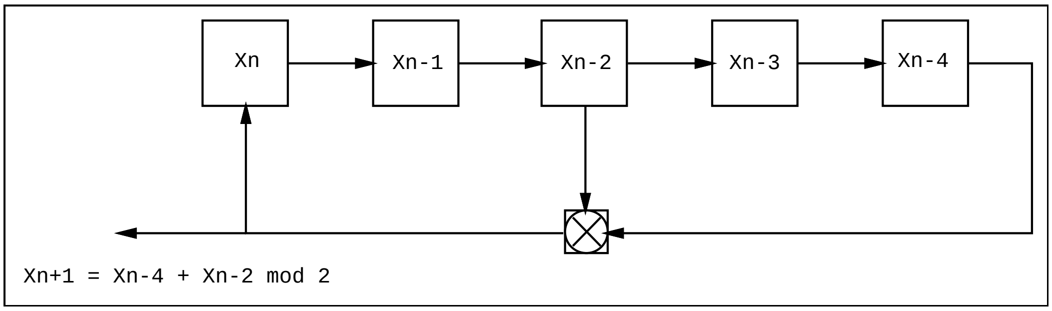


Figure 2.1: A shift register generator
The update transformation $X_n = X_{n-5} + X_{n-3}$ mod 2 can be implemented with a shift-register and an xor gate.

digital computers, so shift register generators are therefore also easy to implement in software. Shift register generators are used to generate random bits. It is very tempting to use the same algorithm to generate random words by using several bits from the random bitstream of a shift register generator. However it is worth noting that the distribution number of the resulting generator will be reduced by a factor of the wordsize. This should therefore only be contemplated if the size of the shift register is very much larger than the wordsize.

2.5 Lagged-Fibonacci generators.

Much recent work has been done on lagged-Fibonacci generators, much of it by G. Marsaglia [20]. Used properly they can produce very high quality random sequences with very long periods. Unlike the linear congruential generators each element is not generated from the previous one but from a pair of elements at fixed distances back in the sequence.

$$x_n \equiv f(x_{n-r}, x_{n-q}) \ : r < q$$

This means that the last q numbers produced by the generator must be preserved in order to continue to produce new numbers. These q numbers form the state-vector. There is often a strong relationship between lagged-Fibonacci generators and linear recurrence generators. When the binary function f and the set of possible values of x form a group isomorphic to modulo arithmetic then a lagged-Fibonacci generator is equivalent to a linear recurrence generator with $q \equiv k$ and $a_i = 0 \ : i \neq r, i \neq q$. As such a lagged-Fibonacci generator only requires a single binary operation to generate the next number in the sequence it can be considered the the optimal type of linear recurrence generator in terms of execution speed, while retaining the advantages of a large number of state variables.

The simplest form of lagged Fibonacci generator that generates n bit integers is one where:

$$f(x_{n-r}, x_{n-q}) = x_{n-r} \bigoplus x_{n-q}$$

Here $\bigoplus$ represents a bitwise exclusive-OR operation on the two n bit integers. This is sometimes called the generalised feedback shift register algorithm [21] . These algorithms have been criticised for having poor statistical properties [20]. Part of the reason for this is that the different bits of each word are generated completely independently of each other so the generator is in fact a number of shift register generators operating in parallel, one for each bit of the output word. The same kind of correlations that we have to avoid between generators operating on separate processors now occur between the different bits of the output word, where they will be a much greater problem. This algorithm will produce all possible n bit integers provided that the state-vector is a nonsingular matrix of bits, i.e. the n shift register state-vectors are linearly independent[22].

In the published literature the starting state-vectors are generated by choosing a bit pattern for the lowest order bits of the state-vector and then iterating the algorithm a large number of times in order to generate the bit values for the higher order bits. Typically the delays are of the order of $100q$ but may be as high as $20,000q$ [23]. However these delays are probably far too small. The matrix for a single iteration is very sparse and often very large, $q = O(100)$. If the transformation for $100q$ iterations is generated it remains a sparse matrix with visible structure. This matrix completely defines the connection between the different bits of the generator. A sparse matrix with obvious structure is an indication of simple correlations between the bits. It is therefore more desirable to use very long offsets where this obvious structure has disappeared.

I have implemented a parallel version of the gfsr generator using a block distribution. The update transformation was

$$X[n] = X[n-32] \bigoplus X[n-512],$$

giving a period of $2^{521}-1$. Each of the 32 bits in the output word is separated by an offset of $2^{401}-1$ and separate processors are separated by an offset of $2^{488}-1$. These offsets are generated by calculating the appropriate transformation matrix from powers of the update matrix T (see equation 2.15). This matrix is a constant transformation that connects the generators on each of the processors, but as it is a dense matrix acting on the last 521 values produced by the generator it is not expected to be a significant correlation for most applications.

The lagged Fibonacci generators are greatly improved when the $\bigoplus$ operator is replaced with one that does not preserve the independence of the different bits [20]. For example:

$$f(x_{n-r}, x_{n-q}) = (x_{n-r} + x_{n-q}) \bmod 2^n$$

$$f(x_{n-r}, x_{n-q}) = (x_{n-r} - x_{n-q}) \bmod 2^n$$

$$f(x_{n-r}, x_{n-q}) = (x_{n-r} \times x_{n-q}) \bmod 2^n \; x_l \, odd \, \forall l$$

where n is the number of bits in the integer. The shorthand designations for these generators are $F(r,q,+)$ $F(r,q,-)$ and $F(r,q,\times)$. The gfsr algorithm can be designated as $F(r,q,\oplus)$ using the same system. Provided that the values of r and q are chosen correctly these algorithms have a much larger period than the shift register generators. The $F(r,q,\pm \bmod 2^n)$ generators have a maximum period of $(2^q-1)2^{n-1}$ and the $F(r,q,\times \bmod 2^n)$ generators have a maximum period of $(2^q-1)2^{n-3}$.

A particularly elegant algorithm exists for generating real numbers uniformly distributed on $[0,1)$ using $F(r,q,\pm)$, also due to G. Marsaglia [24]. In this algorithm the state-vector is a vector of floating point numbers. Each element must be an exact binary fraction.

$$x_i \equiv \frac{I_i}{2^n}$$

where I_i is an integer $0 \leq I_i < 2^n$ and n is an integer small enough for all such fractions to be exactly represented by the floating-point format of the computer. This is usually taken to be the number of bits in the mantissa of the floating point representation. The generator is defined by:

$$f(x_{n-r}, x_{n-q}) = \begin{cases} x_{n-r} - x_{n-q} & \text{if } x_{n-r} \geq x_{n-q} \\ x_{n-r} - x_{n-q} + 1 & \text{otherwise} \end{cases}$$

This is equivalent to a $F(r, q, -)$ lagged Fibonacci generator acting on the integers I_i with the results divided by 2^n. This algorithm generates high quality random real numbers very quickly, and is therefore worth paying particular attention to.

The $F(r, q, \pm \bmod 2^n)$ generators are actually an extension of the shift register generators. The low-order bits of the result from a $(\pm \bmod 2^n)$ operation are independent of the value of any of the higher-order bits in the operands. This means that the lowest m bits of a $F(r, q, \pm \bmod 2^n)$ generator also form a generator of the same type, except that arithmetic is performed mod 2^m. The lowest-order bit $m = 1$ therefore forms a shift register generator with maximum period $2^q - 1$. The values of r and q must therefore be chosen using the same criterion as for a shift register, that is, they must be the non-zero components of a primitive polynomial mod 2. The higher-order bits have a very similar update rule except that in addition to the recurrence relation, the carry bits from the lower-orders are also added into the bitstream. The stream of carry bits feeding into the nth order bit is periodic with the period of the generator mod 2^{n-1}. Because of the linear nature of the recurrence relation, the resulting bitstream can be separated into a shift register sequence and contributions from each period of the carry sequence. The contributions from an even number of periods of the carry sequence will cancel so the period mod 2^n cannot be more than twice that mod 2^{n-1}. This argument shows that the maximum possible period for this kind of generator is $(2^q - 1)2^{n-1}$.

The transformation T for the $F(r, q, \pm \bmod 2^n)$ algorithm is a linear recurrence relation and can be represented as a matrix acting on the state-vector. It can be proved that a necessary and sufficient condition for a $F(r, q, \pm \bmod 2^n)$ generator to have maximal period is that T must have order $j = 2^k - l$ in the group of nonsingular matrices for mod 2 , order $2j$ for mod 4 and order $4j$ for mod 8. let P_n denote the period of the generator mod 2^n. If T has order $j = 2^k - 1$ in the group of nonsingular matrices for mod 2 then the generator mod 2 is a shift register generator with maximal period; $P_1 = 2^k - 1$. Let J be the matrix that advances the sequence by P_1;

$$J = T^{2^q - 1},$$

$$J \bmod 2 = I.$$

Assume the generator has maximal period mod 2^n;

$$\begin{aligned} P_n &= 2^{n-1} P_1 \\ T^{P_n} &= J^{2^{n-1}} \\ J^{2^{n-1}} &= I + 2^n X_n. \end{aligned} \tag{2.16}$$

As the period mod 2^{n+1} must be a multiple of the period mod 2^n, the generator will also have maximal period mod 2^{n+1} provided that $X_n \neq 0 \bmod 2$. If we square equation 2.16

$$\begin{aligned} J^{2^n} &= I + 2^{n+1}(X_n + 2^n X_n^2) \\ &= I + 2^{n+1} X_{n+1} \\ X_{n+1} &= X_n + 2^n X_n^2. \end{aligned} \tag{2.17}$$

So if $X_n \neq 0 \bmod 2$ then $X_{n+1} \neq 0 \bmod 2$. It follows that if the generator has maximal period for all n provided that T must have order $j = 2^k - l$ in the group of nonsingular matrices for mod 2 ,order $2j$ for mod 4 and order $4j$ for mod 8. This is a variation of the proof given by Marsaglia[20].

We can see that only the highest-order bits of the $F(p, q, \pm \bmod 2^n)$ generator can have the full period and that each successively lower-order bit cycles in half the time of the one above it. In principle we can consider consider $F(p, q, \pm \bmod r^n)$ generators for any prime number r. These would have a maximum period of $(r^q - 1)r^{n-1}$. The practical disadvantages of not using a binary number representation mean these algorithms are only of theoretical interest.

The $F(p, q, \times \bmod 2^n)$ generators can be reduced to $F(p, q, \pm \bmod 2^{n-1})$ generators by expressing the abelian group of residues as a direct product of cyclic groups and considering the update transformation of the generator exponents [20].

The only restriction on the state-vector of a $F(p, q, \pm \bmod 2^n)$ generator is that the elements of the state-vector cannot all be even. The number of valid state-vectors is therefore $(2^q-1)2^{q(n-1)}$. As the period of the generator is $(2^q-1)2^{n-1}$ there are $2^{(q-1)(n-1)}$ possible sequences that can be generated by the algorithm.

In a single-processor implementation of this kind of generator we are only concerned with correlations within a single sequence. Once we consider running a number of generators in parallel, the existence of separate number sequences is of interest. To implement this algorithm in parallel we want to understand how the independent cycles are related to each other. If there is a simple transformation that maps one cycle into another, then this is a potential correlation between processors that we have to avoid when initialising a set of parallel generators. We already have an efficient method of moving a generator to any point within its cycle. If we also have a method to transform cycles into each other we can consider putting a separate cycle on each processor of a parallel program.

A partial understanding of how separate cycles are related can be gained by considering sequences related by multiplying the state vector by a constant. This will only result in a valid state vector if the constant is an odd integer. It can be shown that this will never map the generator into a different part of the same sequence provided that

$$J = T^{2^q-1} \neq 3I \bmod 4. \tag{2.18}$$

The proof is as follows; assume there is no internal map mod 2^n,

$$T^i \neq kI \bmod 2^n, \tag{2.19}$$

for all i and all odd integers $1 < k < 2^n$. Consider arithmetic mod 2^{n+1}. If there exists an odd integer k such that

$$T^\alpha = kI \bmod 2^{n+1}, \tag{2.20}$$

with $k < 2^{n+1}$ and $\alpha < 2^n(2^q-1)$ then it follows that

$$T^\alpha = kI \bmod 2^n. \tag{2.21}$$

To be consistent with equation 2.19, we must have $k = 1 \bmod 2^n$, as we know k lies in the range $1 < k < 2^n$ this gives us $k = 2^n + 1$. From equation 2.21

$$T^\alpha = I \bmod 2^n,$$

so α is a multiple of P_n; $\alpha = 2^{n-1}(2^q - 1)$. Substituting the values of α and k into equation 2.20

$$J^{2^{n-1}} = (1 + 2^n)I \bmod 2^{n+1}$$

From equation 2.16 this gives us $X_n = I \bmod 2$ as a necessary condition for an internal map mod 2^{n+1} If $J \neq 3I \bmod 4$ then $X_1 \neq I \bmod 2$, from equation 2.17 $X_i \neq I \bmod 2 \forall i$. Hence there are no internal maps for all n.

This still does not take account of all of the separate cycles; there are 2^{n-1} odd integers less than 2^n. So if we group together cycles that are connected by a constant multiplier, we are still left with $2^{(q-2)(n-1)}$ independent groups. If the lowest n bits of 2 state vectors are from different groups of the generator mod 2^n then the state vectors must also be in different groups mod 2^m for all $m > n$. Each time n is increased by 1 the number of groups increases by $2^{q-2} = 2^q/4$. This shows us a way of constructing state vectors that belong to different groups. Start with a state vector S, $n-1$ bits wide, and generate a new state vector S' by setting the nth bits to random values,

$$S'_i = S_i + 2^{n-1}B_i$$

for all $i < q$ where the Bs are chosen randomly from 0 and 1. There are only 3 other possible choices of B that come from the same group of cycles. One occurs $2^{n-2}(2^q - 1)$ updates later in the same cycle as S'; the other two occur in the cycle obtained by multiplying S' by $2^{n-1} + 1$. These values of B can be easily calculated;

$$\begin{aligned} B' &= B + S \bmod 2 \\ B'' &= B + X.S \bmod 2 \\ B''' &= B + X.S + S \bmod 2 \end{aligned}$$

where $X = X_0 \bmod 2$ from equation 2.16. This enables us to keep choosing new values of B and easily reject state vectors that come from groups we have already selected. To minimise the relation between the state vectors this procedure should be carried out at $n = 2$ and the higher order bits filled in randomly. At this stage all the sequences will still be the same mod 2 so transformation matrices will have to be used to shift each sequence a different distance around its period.

This procedure seems unnecessarily complex; the one advantage it has over the other methods discussed previously is that it guarantees that there will never be a linear dependence between two of the sequences, including correlations between elements offset by an arbitrary distance.

All simulations presented in this thesis have used this kind of random number generator. In these cases the initial state for each processor was either chosen randomly or the states were distributed at constant offsets around a single cycle using a transformation matrix.

Chapter 3

QED and four-fermion interactions

3.1 QED

Quantum Electro-Dynamics or QED is a gauge theory based on the $U(1)$ gauge group. It represents a system of charged fermions coupled by electromagnetic interactions. The predictions made by QED are very accurate; perturbative calculations and experimental observations agree with each other to an extremely high accuracy. For example the experimental value and the theoretical prediction of the anomalous magnetic moment of the electron agree to 8 decimal places. QED is routinely used for a wide variety of calculations in atomic physics and is therefore of immense practical importance; in addition the simplicity of QED makes it an important test-bed for our understanding of field theories in general.

3.1.1 The triviality of QED

One of the oldest problems in four-dimensional quantum field theory is whether or not QED is interacting in the limit in which the cut-off is removed. Suppose the bare coupling is e_0, the bare electron mass is m_0, and that the theory is regularised with a momentum cut-off Λ. Assuming the theory does not confine charge, a renormalised charge, e_R, may be defined by the Thomson scattering cross-section and a renormalised electron mass, m_R, may be defined as the lowest energy in the charge-one sector. The relationship between the bare charge and the renormalised charge is

$$e_R^2 = Z_3 e_0^2 \tag{3.22}$$

where, for most regularisation schemes, including the lattice [25, 26],

$$0 \leq Z_3 \leq 1. \tag{3.23}$$

The renormalisability of QED guarantees that the results of renormalised perturbation theory (expressed in terms of e_R and m_R) are independent of Λ/m_R for sufficiently large Λ/m_R, but it does not imply that the theory has a non-zero renormalised charge in the infinite cut-off limit. Weak-coupling perturbation theory seems to suggest that the renormalised charge vanishes in this limit. The β function is a measure of how the coupling changes with the cut-off,

$$\beta = \Lambda \frac{\partial e_0^2}{\partial \Lambda}.$$

The leading terms of a weak-coupling perturbative expansion for β give a positive definite value. In the region where this expansion is valid an increase in the value of the cut-off will force a compensating increase in the value of the bare charge.

In renormalisation-group improved one-loop perturbation theory the bare charge is given by

$$e_0^2 = \frac{e_R^2}{1 - \frac{e_R^2}{6\pi^2} \log \frac{\Lambda}{m_R}}. \tag{3.24}$$

Keeping e_R fixed as the cut-off is removed, the bare charge has a pole at sufficiently large Λ/m_R, and becomes imaginary as Λ/m_R is increased further. In the real world, $e_R^2/4\pi = 1/137$, and the cut-off can be made larger than the Planck scale, the scale where quantum gravity effects should become significant, without the bare coupling becoming large. If this weak-coupling picture is qualitatively correct for all bare charges, then the cut-off cannot be completely removed at a fixed non-zero renormalised coupling, but can be taken larger and larger as the value of the renormalised charge is reduced. The physical process driving this behaviour is vacuum polarisation. Virtual electron-positron pairs in the vacuum can be separated by an electric field. The vacuum therefore behaves like a dielectric and shields the electric charge. As the cut-off is removed this effect becomes stronger and the bare electric charge must be increased to compensate. The exception to this is when the renormalised electric charge is zero. In this case the bare charge e_0 is zero for all values of the cut-off. The cut-off can therefore only be completely removed at zero renormalised charge, i.e. at the Gaussian fixed point. This would be the *triviality* of QED. Triviality could be avoided if new physics enters at large bare charge, where perturbation theory breaks down. The predictive power of perturbative QED remains unchallenged because other physical processes will have to be included in any real calculation long before the bare charge becomes large. The question about what happens to QED as the cut-off is removed is still an important one for our general understanding of field theories.

In terms of renormalisation group trajectories the perturbative picture is that all the possible trajectories except for the one at $e_0 = 0$ will travel out to $e_0^2 = \infty$ before the cut-off has been totally removed, so the only point possibly corresponding to a continuum theory is the infra-red stable fixed point (irsfp) at $e_0 = 0$ corresponding to non-interacting fermions. It is possible that some other physical process becomes significant at large values of the bare coupling. If such a process reduces the β function then a second fixed point ($\beta = 0$) could occur in the strong coupling region, see figure 3.1.

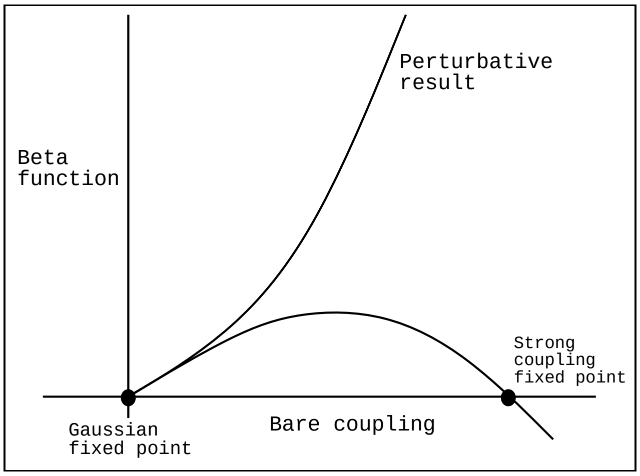


Figure 3.1: Possible evolution of the β function

Perturbative calculations give a positive definite value to the β function. A strong coupling fixed point is therefore only possible if some non-perturbative process reduces the value of the β function.

What is necessary for an interacting continuum limit is an ultra-violet stable fixed point (uvsfp) at some non-zero value of the bare coupling, $e_0 = e_{uv}$. Removal of the cut-off at fixed e_R would drive the bare coupling to the uvsfp. However, this is not sufficient for the continuum theory to be interacting; it is still possible for the renormalised charge to vanish. In contrast, it is now fairly clear that no such uvsfp occurs in the 4-dimensional ϕ^4 theory, so that the continuum theory must be built at the Gaussian fixed point and is free [27]. Apart from a calculation of the renormalised charge itself, which would obviously settle the matter, non-mean-field scaling exponents at the uvsfp would presumably indicate non-triviality. This is because an interacting continuum theory defined at a second order phase transition exhibits fluctuations on all length scales and is therefore unlikely to be well described using a mean field approximation.

A strong-coupling uvsfp also implies that a new phase of QED exists on the far side of this critical value. Unfortunately, strong-coupling QED can only be explored numerically using lattice gauge theory, or by means of *ad hoc* truncations of the Schwinger-Dyson equations. Since the theory would be scale-invariant and $\Lambda/m_R = \infty$ at the fixed point, there must exist a second (or higher) order phase transition at e_{uv}. The first evidence for such a transition was obtained several years ago [28, 29] by lattice Monte Carlo simulations using the non-compact form of the pure gauge action.

3.1.2 The Schwinger-Dyson equation

The Schwinger-Dyson equations are integral equations obeyed by the Greens functions of quantum field theories. The numerical lattice results have inspired a revival of analytic work based on solutions to truncated Schwinger-Dyson equations [30, 31, 32]. In the ladder approximation, where fermion loops and vertex corrections are ignored, there is indeed an uvsfp which separates the perturbative phase of QED from a phase where e^+e^- pairs condense into the vacuum and break chiral symmetry. The ladder approximation decouples the Schwinger-Dyson equation for the fermion self-energy, which may then be replaced by an ordinary differential equation and solved analytically at large momentum [30, 31, 32, 33]. A chiral-symmetry-breaking solution exists for $\alpha = e^2/4\pi > \alpha_c = \pi/3$. The phase transition is of infinite order (has an essential singularity in the order parameter) in the ladder approximation. The physics driving the transition, sometimes called 'collapse of the wavefunction', can be seen in the simpler case of a Dirac particle in an attractive α/r potential [31]. The wavefunction is singular for $\alpha > 1$. The singularity may be regularised by cutting off the potential at short distances, $r < a$. Then the requirement that the binding energy be independent of the cut-off, a, imposes a cut-off dependence of α, such that

$$\frac{d\alpha}{da} > 0, \qquad \alpha > 1; \tag{3.25}$$

α must be increased with a to compensate for the lost potential. This UV stability, reminiscent of QCD, is the opposite of screening due to vacuum polarisation and we must ask whether it survives the introduction of dynamical fermions?

In the ladder approximation only fermion mass renormalisation occurs and, notably, this results in a large anomalous dimension for $\bar{\psi}\psi$, corresponding to a scaling dimension of 2 at α_c [32]. As a consequence, four-fermi interactions acquire a scaling dimension of 4 at α_c in this approximation, indicating that a consistent treatment of QED at strong coupling should include them from the start.

In order to preserve the chiral symmetry of the action, the appropriate four-fermi interaction to include is that of the Nambu-Jona-Lasinio model[34]. The Lagrangian for the continuum theory is

$$\mathcal{L} = \mathcal{L}_{QED} + \frac{G}{2}[(\bar{\psi}\psi)^2 - (\bar{\psi}\gamma_5\psi)^2]. \tag{3.26}$$

The solution of the ladder approximation for this model was obtained in [32].

An interesting phase diagram for this model was conjectured on the basis of an analysis of the solutions of the approximate Schwinger-Dyson equation as a function of the two couplings α and $g = G\Lambda^2/4\pi^2$, where Λ is the momentum cut-off [35, 36, 37]. This is shown in figure 3.2. For large enough four-fermi coupling ($g > \frac{1}{4}\left(1 + \sqrt{1 - \frac{\alpha}{\alpha_c}}\right)^2$ for $\alpha < \alpha_c$, and $g > 0$ otherwise) a chiral-symmetry-breaking solution exists for all values of α. Within the symmetric phase, the anomalous dimension for $\bar{\psi}\psi$ [38] is the same as for $G = 0$:

$$\gamma_m = 1 - \sqrt{1 - \frac{\alpha}{\alpha_c}}, \tag{3.27}$$

whereas, in the broken phase, for $\alpha < \alpha_c$,

$$\gamma_m = 1 + \sqrt{1 - \frac{\alpha}{\alpha_c}}. \tag{3.28}$$

This is remarkable in that it seems to suggest that, close to $\alpha = 0$, for $g > 1$ the scaling dimension of the four-fermi interaction is 2, i.e. it is super-renormalisable!

Within the ladder approximation, the two phases are separated by a line of ultra-violet stable fixed points, with renormalisation-group flow only in g for $\alpha < \alpha_c$ [37], i.e. there is no single ultra-violet fixed point about which a continuum limit might be built. This is probably an artifact of the approximation.

Approximate Schwinger-Dyson calculations have been performed with a variable number of fermion 'flavours', N_f. The results indicate a conventional second-order transition with critical exponents which depend on N_f, rapidly approaching mean-field values for $N_f > 1$.

This hypothesis that there is an ultra-violet fixed point at strong coupling, which may be used to define a non-trivial four-dimensional field theory, has been used to construct technicolor models and also has intrinsic interest as a new type of continuum field theory. However it is built on rather drastic approximations. In particular, vacuum polarisation effects, which are responsible for the ultra-violet instability of the $e = 0$ fixed point in ordinary QED, are excluded but might be expected to be important.

For two reasons it is important to check the predictions of the truncated Schwinger-Dyson equations against numerical simulations. Firstly, to see whether any of the new strong-coupling dynamics persists in the full theory and secondly, to explore whether truncated Schwinger-Dyson equations can be a reliable guide to non-perturbative quantum field theory. The immediate questions to be asked of the numerical simulations are:

- how do physical quantities scale near the phase transition?
- is $e_R \neq 0$ possible?
- what is the dependence of the above on the number of fermion species?

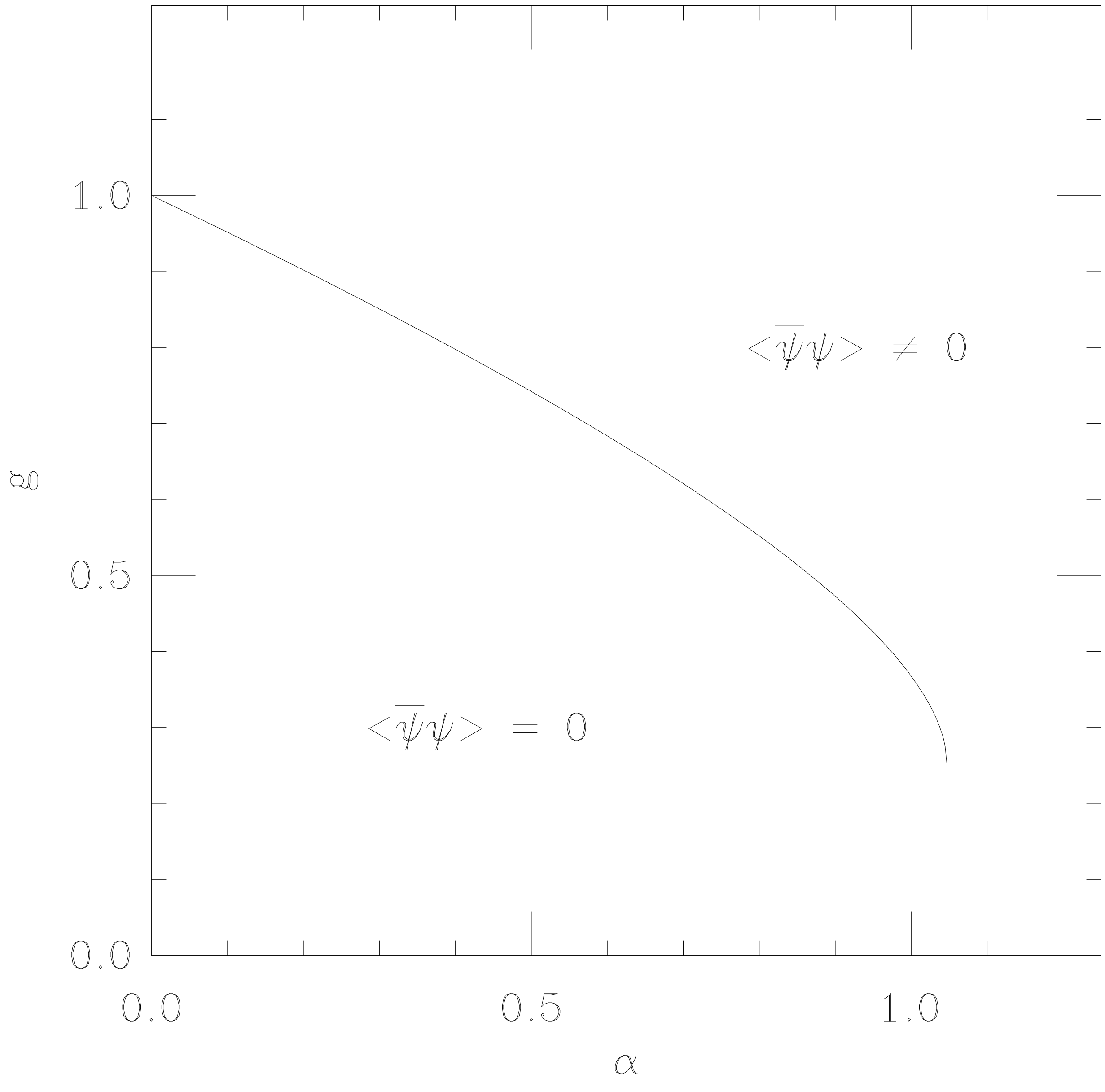


Figure 3.2: Phase diagram predicted using the Schwinger-Dyson equation Possible phase diagram of the $U(1)$-gauge-invariant Nambu-Jona-Lasinio model obtained using the ladder approximation to the Schwinger-Dyson equation for the fermion self-energy.

3.1.3 QED on the lattice

There are two possible ways of formulating QED on the lattice, compact and non-compact.

The compact form of lattice QED with massless staggered fermions, has the following lattice action:

$$\begin{aligned}
S &= \beta \sum_{x,\nu>\mu} (1 - \cos(\Delta_\mu \theta_\nu(x) - \Delta_\nu \theta_\mu(x))) \\
&\quad + \sum_{x,y} \bar{\chi}(x) D(\theta)_{xy} \chi(y) \\
D(\theta)_{xy} &= \frac{1}{2} \sum_\mu \eta_\mu(x) [e^{i\theta_\mu(x)} \delta_{x+\hat{\mu},y} - e^{-i\theta_\mu(y)} \delta_{x-\hat{\mu},y}] \\
\eta_\mu(x) &= (-1)^{x_0 + \ldots + x_{\mu-1}} \\
\beta &= \frac{1}{e^2},
\end{aligned} \tag{3.29}$$

where θ_μ takes values in the range $[0, 2\pi)$, x, with integer-valued components, labels the lattice sites, $\mu = 0, \ldots, 3$ labels the lattice directions and $\hat{\mu}$ is the corresponding unit vector.

The non-compact form of lattice QED with massless staggered fermions, has a very similar action:

$$\begin{aligned}
S &= \frac{\beta}{2} \sum_{x,\nu>\mu} (\Delta_\mu \theta_\nu(x) - \Delta_\nu \theta_\mu(x))^2 \\
&\quad + \sum_{x,y} \bar{\chi}(x) D(\theta)_{xy} \chi(y),
\end{aligned} \tag{3.30}$$

where θ_μ takes values on the real line.

Both of these actions are gauge invariant and in the limit where the lattice spacing is taken to 0 they both reduce to the continuum QED action. On the lattice however the two actions behave differently because of topological excitations. The compact form of QED only has a first order transition so it cannot correspond to an interacting theory [28, 29].

Non-compact QED appears to have a second order transition at strong coupling [28, 29]. The nature of this transition seems to depend on the number of fermion flavours. At large N_f, the transition appears to become first order [40]. For lower values of N_f the transition appears to be second order but non-mean-field critical exponents have only been claimed for small numbers of fermion flavours (less than about four) [28, 29, 41].

There is a certain practical difficulty in interpreting the data from this kind of simulation. This arises because we are interested in the behaviour of systems with zero fermion mass but we are only able to simulate systems with a non-zero fermion mass. Much of the published data is therefore based on polynomial extrapolation of the data for $\langle\bar{\chi}\chi\rangle$ at non-zero fermion mass on a finite lattice to zero fermion mass. Non mean-field behaviour has sometimes been claimed because of deviations from mean-field behaviour of this extrapolated curve. This procedure has obvious shortcomings, there is no way to distinguish between non mean-field behaviour and a breakdown in the extrapolation.

The Schwinger-Dyson equation calculations suggest that four fermion interactions could be significant in understanding the renormalisation behaviour of strong coupling QED . Consequently, we performed a numerical simulation of the fully-interacting $U(1)$-gauge-invariant Nambu-Jona-Lasinio model. We formulated the theory using staggered fermions, $\bar{\chi}$ and χ, [4] in order that the lattice theory possesses a continuous chiral symmetry:

$$\begin{aligned} \chi(x) &\rightarrow e^{i\lambda\epsilon(x)}\chi(x) \\ \bar{\chi}(x) &\rightarrow \bar{\chi}(x)e^{i\lambda\epsilon(x)} \\ \epsilon(x) &= (-1)^{x_0+x_1+x_2+x_3}, \end{aligned} \tag{3.31}$$

where λ is real. Then a gauge-invariant, chiral-invariant four-fermi interaction is

$$G\sum_{x,\mu} \bar{\chi}(x)\chi(x)\bar{\chi}(x+\hat{\mu})\chi(x+\hat{\mu}), \tag{3.32}$$

Our lattice action is that for noncompact QED plus the above four-fermi interaction. In order to perform the exact integration over the Grassmann variables, an auxiliary vector field is introduced, so that the lattice action is

$$S = \frac{\beta}{2}\sum_{x,\nu>\mu} \left(\Delta_\mu\theta_\nu(x) - \Delta_\nu\theta_\mu(x)\right)^2$$

$$
\begin{aligned}
& + \sum_{x,y} \bar{\chi}(x)[D(\theta) + 2\sqrt{G}D(\theta') + m]_{xy}\chi(y) && (3.33)\\
D(\theta)_{xy} &= \frac{1}{2}\sum_{\mu} \eta_\mu(x)[e^{i\theta_\mu(x)}\delta_{x+\hat{\mu},y} - e^{-i\theta_\mu(y)}\delta_{x-\hat{\mu},y}] && (3.34)\\
\eta_\mu(x) &= (-1)^{x_0+\ldots+x_{\mu-1}} && (3.35)\\
\beta &= \frac{1}{e^2}. && (3.36)
\end{aligned}
$$

Here, θ_μ takes values in the real line, whereas the auxiliary vector field $\theta'_\mu \in [0, 2\pi)$. The integration over θ'_μ can be done analytically and produces the chiral-invariant four-fermi interaction:

$$I = \int \mathcal{D}\theta' e^{2\sqrt{G}\sum_{x,y} \bar{\chi}(x)D(\theta')_{xy}\chi(y)},$$

$$I = \int \mathcal{D}\theta' e^{\sqrt{G}\sum_{x,y,\mu} \bar{\chi}(x)\,\eta_\mu(x)[V_\mu(x)\delta_{x+\hat{\mu},y} - V_\mu^\dagger(y)\delta_{x-\hat{\mu},y}]_{xy}\chi(y)},$$

where to $V_\mu = e^{i\theta'_\mu}$,

$$I = \prod_{x,\mu}\frac{1}{2\pi}\int d\theta' e^{\sqrt{G}[\bar{\chi}(x)\eta_\mu(x)V\chi(x+\hat{\mu}) - \bar{\chi}(x+\hat{\mu})\eta_\mu(x)V^\dagger\chi(x)]} = \prod_{x,\mu}\frac{1}{2\pi}\int d\theta' e^{A},$$

Now expand the exponential as a power series, there are no A^3 and higher terms because the χs are Grassmann variables.

$$I = \prod_{x,\mu}\frac{1}{2\pi}\int d\theta'[1 + A + \frac{A^2}{2}],$$

$$\int d\theta' V = \int d\theta' V^\dagger = 0,$$

therefore ,

$$\int d\theta' A = 0,$$

$$I = \prod_{x,\mu}\frac{1}{2\pi}\int d\theta'[1 + \frac{A^2}{2}].$$

$$I = \prod_{x,\mu}\frac{1}{2\pi}\int d\theta'[1 - G(\eta_\mu(x)^2\bar{\chi}(x)\chi(x+\hat{\mu})\bar{\chi}(x+\hat{\mu})\chi(x)V^\dagger V]$$

$$I = \prod_{x,\mu}[1 - G\bar{\chi}(x)\chi(x+\hat{\mu})\bar{\chi}(x+\hat{\mu})\chi(x)]$$

$$I = e^{-\sum_{x,\mu} G\bar{\chi}(x)\chi(x+\hat{\mu})\bar{\chi}(x+\hat{\mu})\chi(x)}$$

The effective action is therefore

$$\begin{aligned} S_{eff} &= \frac{\beta}{2}\sum_{x,\nu>\mu}(\Delta_\mu\theta_\nu(x)-\Delta_\nu\theta_\mu(x))^2+\sum_{x,y}\bar{\chi}(x)[D(\theta)+m]_{xy}\chi(y) \\ &\quad -G\sum_{x,\mu}\bar{\chi}(x)\chi(x)\bar{\chi}(x+\hat{\mu})\chi(x+\hat{\mu}). \end{aligned} \tag{3.37}$$

Note that a scalar auxiliary field produces the wrong sign of the four-fermi coupling. Then the chiral condensate is defined as

$$\langle\bar{\psi}\psi\rangle = \lim_{m\to 0}\lim_{V\to\infty}\langle\bar{\chi}\chi\rangle. \tag{3.38}$$

3.2 The phase diagram of QED with an additional four-fermi interaction

We investigated the phase diagram of QED with additional four-fermion interactions using a series of dynamical fermion simulations. Configurations on an 8^4 lattice were generated by the Hybrid Monte Carlo algorithm [6], with trajectories of unit length in molecular-dynamics time. The number of molecular dynamics steps used was chosen separately for each simulation in order to maximise the simulation speed while retaining an acceptance rate of approximately 80%. Our fermionic boundary conditions were periodic in space and anti-periodic in time. Our aim was to obtain evidence for the phase diagram, in figure 3.2, and check the results of [28, 29] for $G = 0$.

The results at $G = 0$ for $\langle\bar{\chi}\chi\rangle$ and $S_0/\beta = \langle\frac{1}{2}(\Delta_\mu\theta_\nu - \Delta_\nu\theta_\mu)^2\rangle$ for a range of β values, at fermion masses $m = 0.0125$, 0.025, 0.05 and 0.1, are in table 3.1 and table 3.2. The results are averages over a minimum of 200 trajectories, having discarded 100 trajectories for equilibration. The errors are estimates for the standard deviation in the mean obtained from binning the data. Our data is in good agreement with the data in reference [28, 29], except for the lightest fermion mass in the vicinity of the transition, in the broken phase, where our values for $\langle\bar{\chi}\chi\rangle$ are systematically larger by several standard deviations. This may be attributed to critical slowing down or, possibly, stepsize errors in the results of reference [28, 29], since these errors are expected to be largest in the broken phase at low mass.

Our results extend those of reference [28, 29] to higher masses. Graphs of the mass-dependence of $\langle\bar{\chi}\chi\rangle$ are shown in figure 3.3 Linear extrapolation to zero mass, using the data at $m = 0.0125$ and 0.025, is not supported by the higher-mass data for $\beta \leq 0.21$. Because of our disagreement with reference [28, 29] on some of the values of $\langle\bar{\chi}\chi\rangle$ at $m = 0.0125$ and the sensitivity of this extrapolation procedure to such discrepancies, the values we would obtain by extrapolation do not agree with [28, 29]. We deduce that current simulations of non-compact QED are finite-mass affected and that extrapolation to zero mass is unreliable. On the basis of figure 3.3, where there appears to be a qualitative difference between the curves for $\beta = 0.18$ and $\beta = 0.19$, in that the former seems more likely to support extrapolation to non-zero $\langle\bar{\chi}\chi\rangle$ on an infinite lattice, we conclude that the critical point is in the range $0.18 < \beta \leq 0.19$. $\beta = 0.19$ is also roughly the location of the points of inflexion in the plots of $\langle\bar{\chi}\chi\rangle$ vs. β, figure 3.4, which is sometimes taken as an indication of criticality.

Our data for the plaquette expectation value, S_0/β, at $G = 0$ exhibits the expected suppression due to dynamical fermions, relative to the quenched value of $1/4\beta$ (see figure 3.5). There is a slight steepening of the plaquette expectation value as a function of β and evidence of critical slowing down in the vicinity of the transition, but the effects are too small to provide a reliable independent estimate of the critical point.

In the region $0.18 \leq \beta \leq 0.215$ at $m = 0.0125$, we looked carefully at runs of 400-700 trajectories from different starts, but saw no evidence of metastability (see figure 3.6). This is in agreement with the conclusion of reference [28, 29] that the transition is not strongly first order.

If we assume that the transition is second order, then for infinite volume

$$\langle\bar{\chi}\chi\rangle|_{\beta=\beta_c} \sim m^{\frac{1}{\delta}}. \tag{3.39}$$

Mean-field theory gives the value $\delta \simeq 3$. We can compare this with our data by plotting $\langle\bar{\chi}\chi\rangle^3$ against mass, see figure 3.7. This plot is approximately linear in the region near the critical coupling. There is no compelling evidence for an essential singularity in $\langle\bar{\chi}\chi\rangle$ vs. β [28, 29], or for an interacting continuum limit, from such a crude analysis of our data.

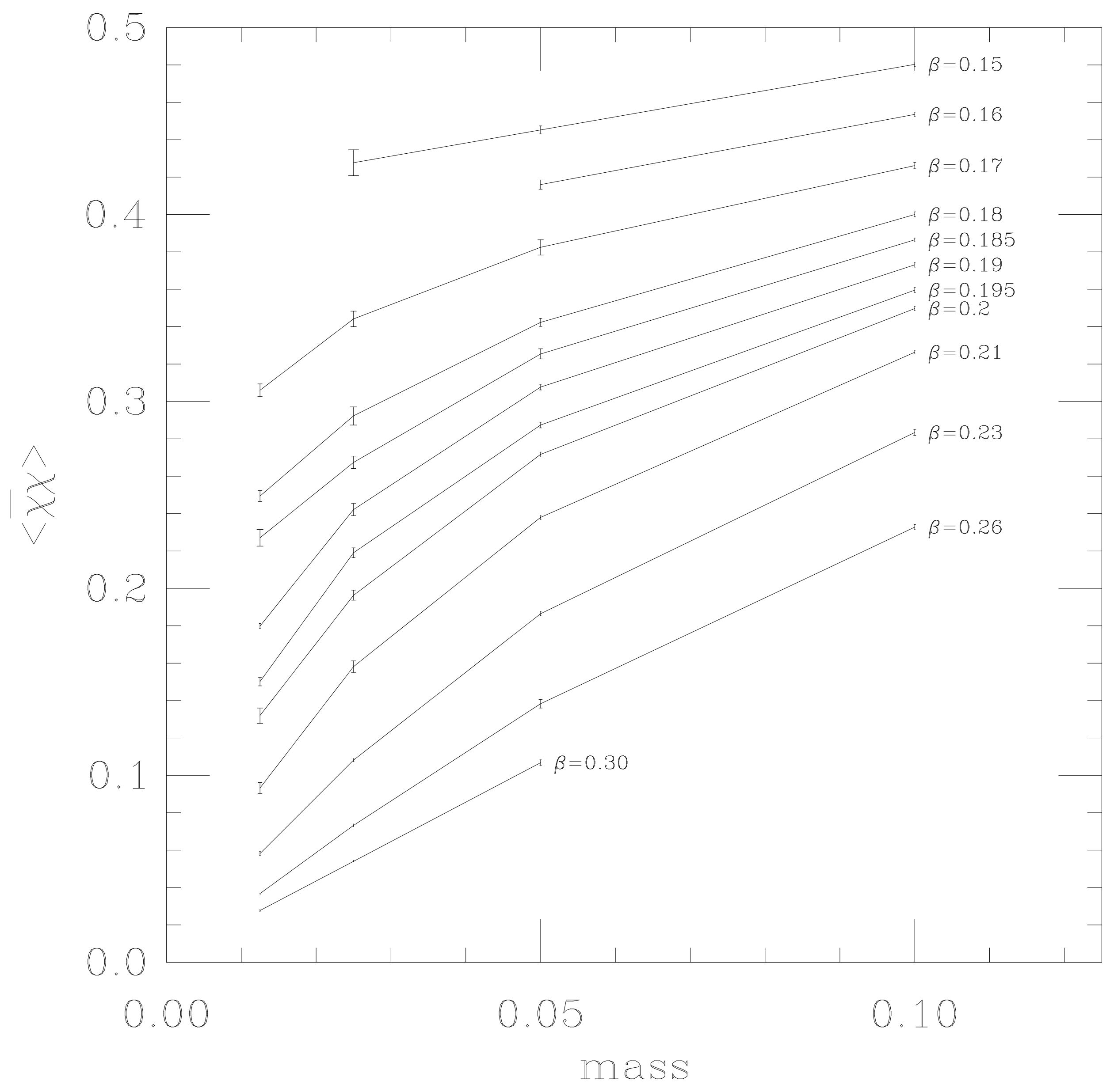


Figure 3.3: Fermion-mass dependence of $\langle\bar{\chi}\chi\rangle$ at $G = 0.0$
The points are joined by straight lines to guide the eye.

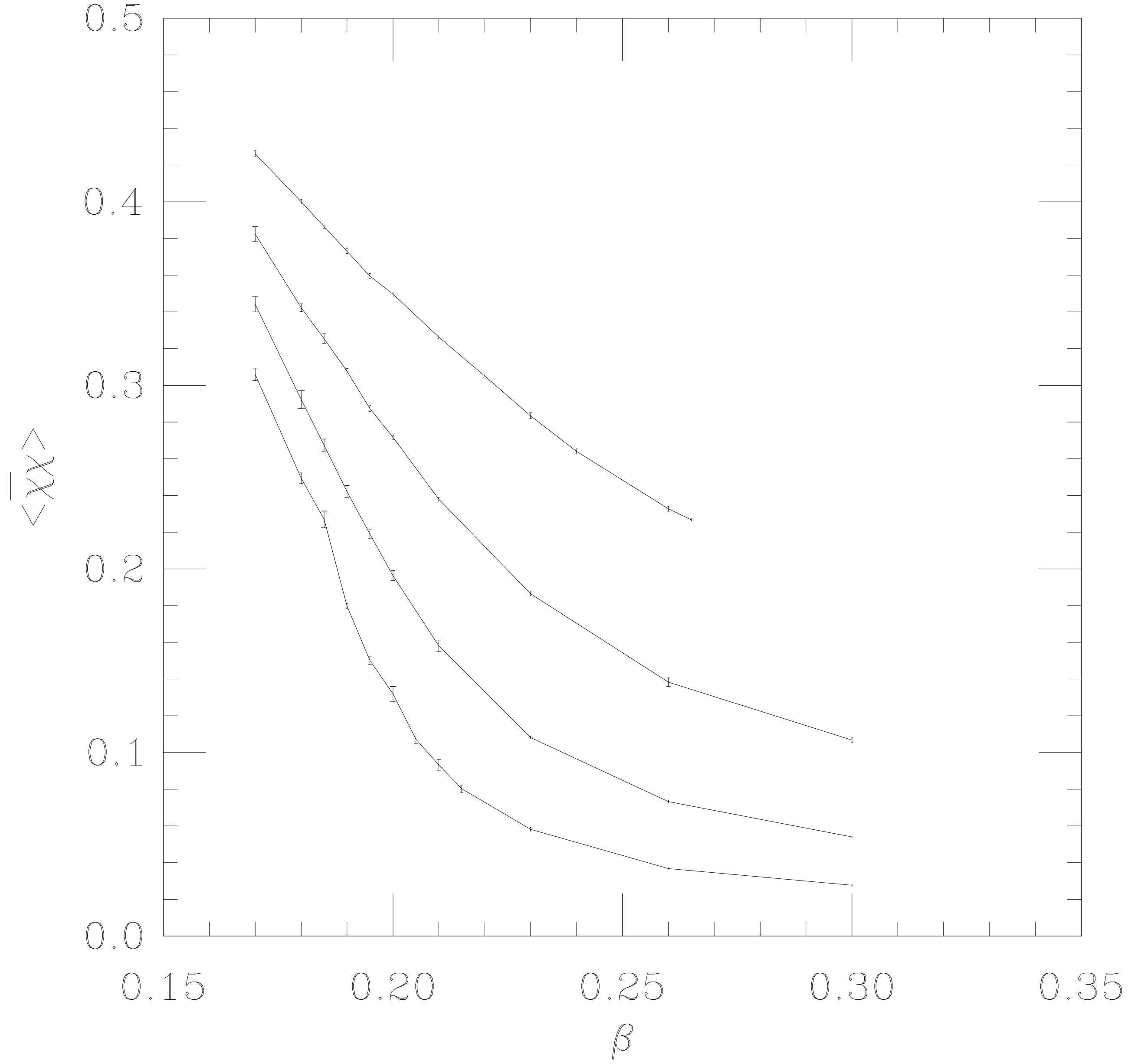


Figure 3.4: $\langle\bar{\chi}\chi\rangle$ against β at $G = 0.0$

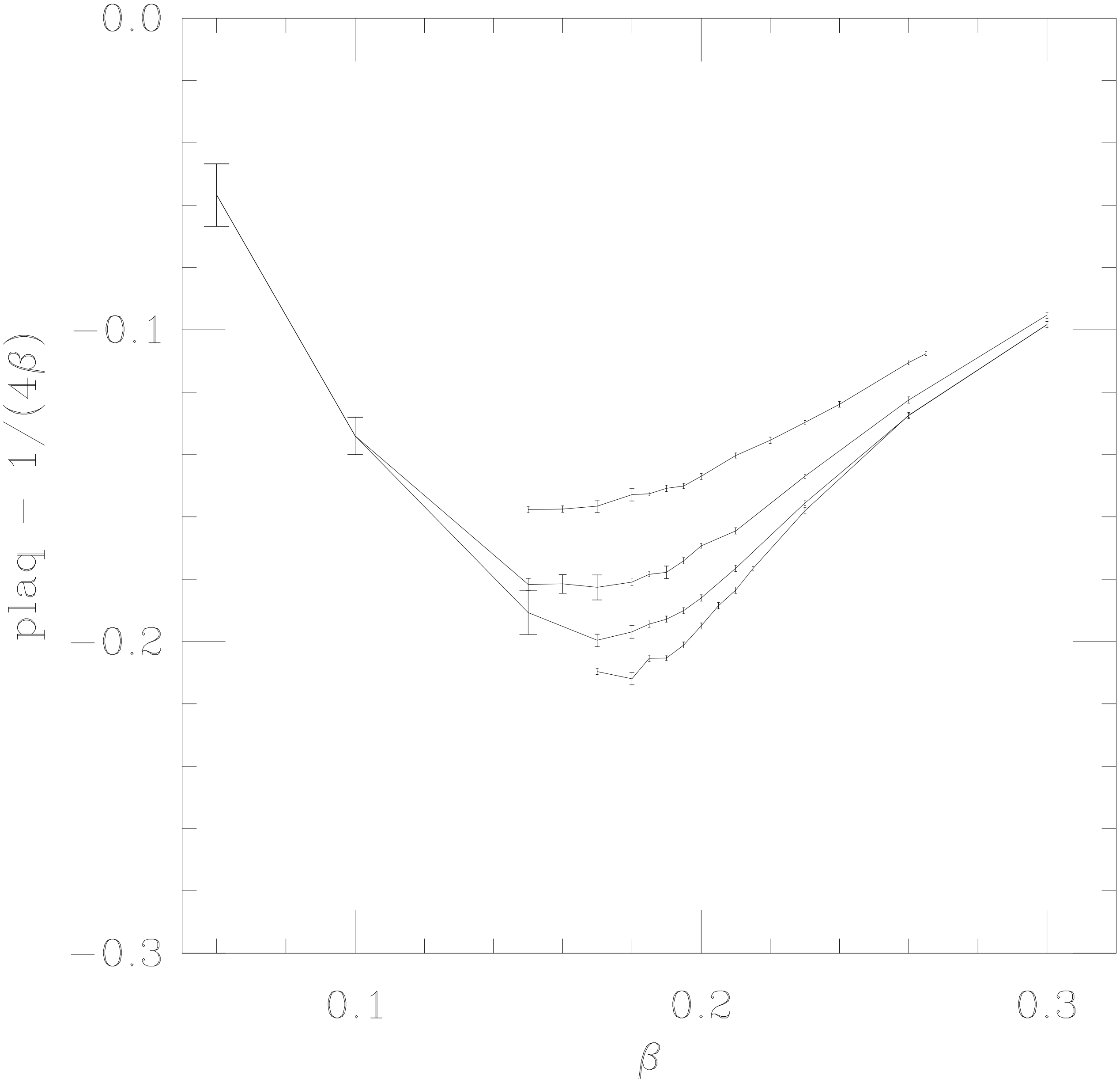


Figure 3.5: Suppression of the plaquette expectation value

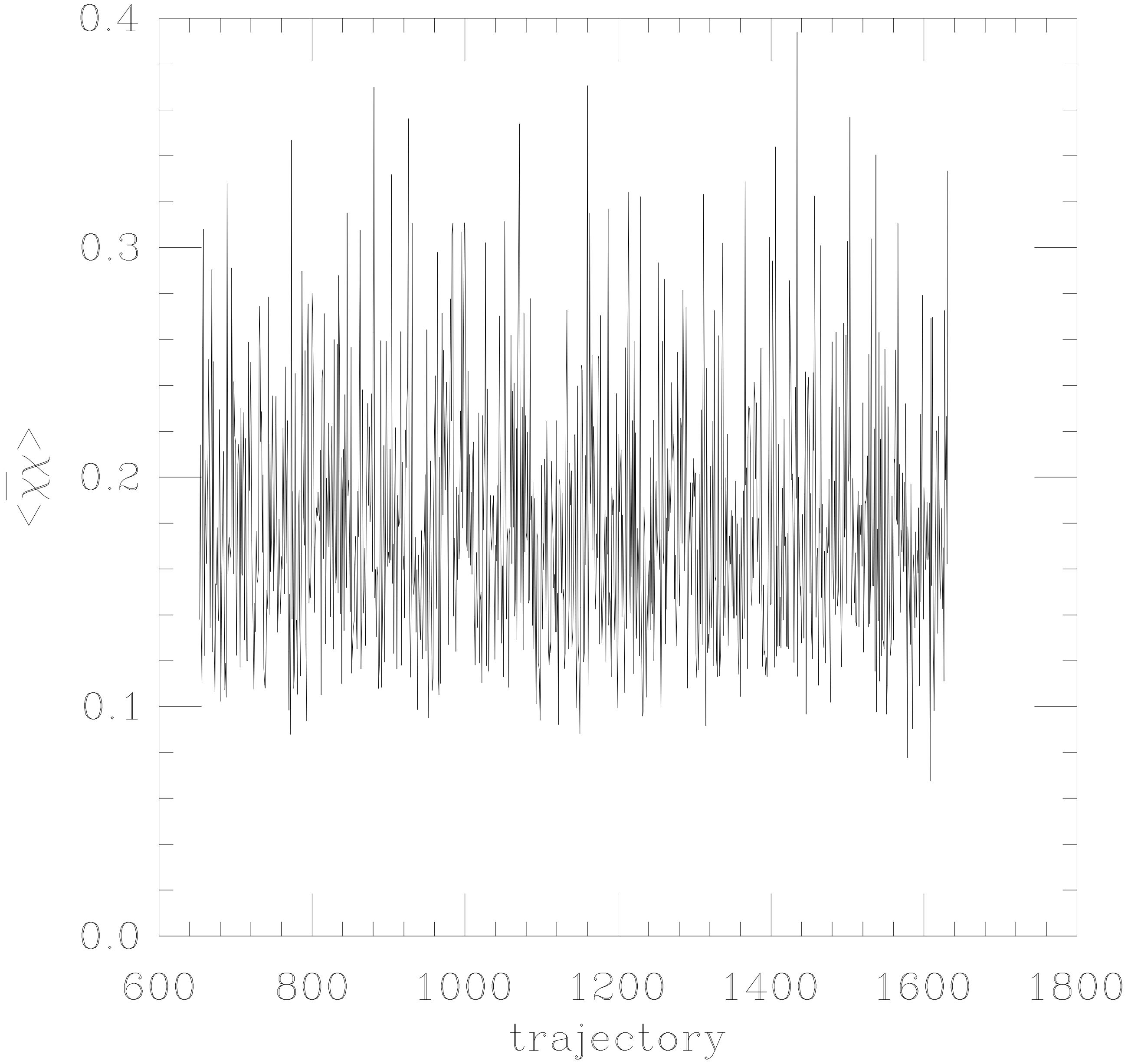


Figure 3.6: $\langle\bar{\chi}\chi\rangle$ against trajectory at $m = 0.0125$, $\beta = 0.19$

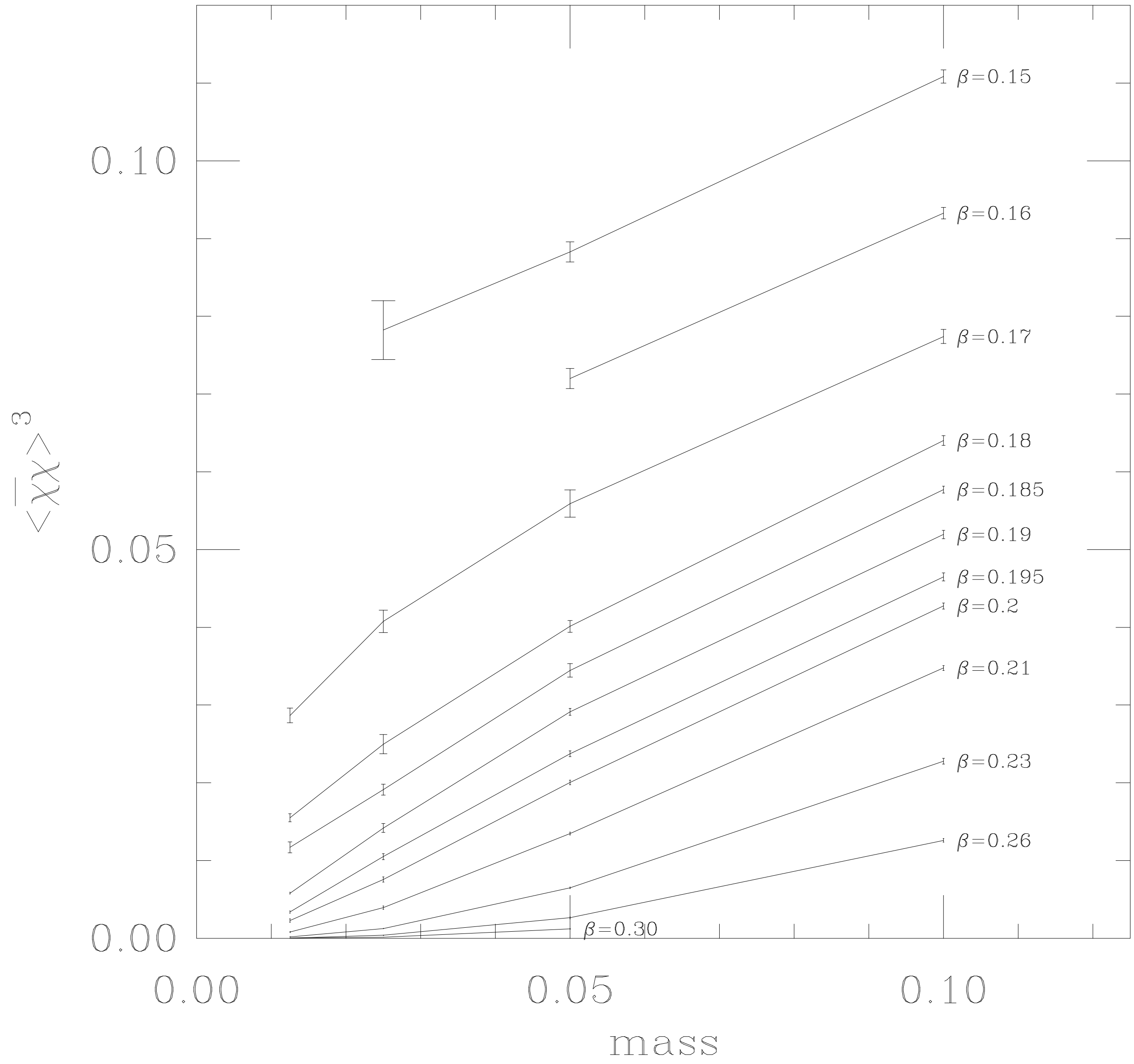


Figure 3.7: $\langle\bar{\chi}\chi\rangle^3$ against *mass* at $G = 0.0$

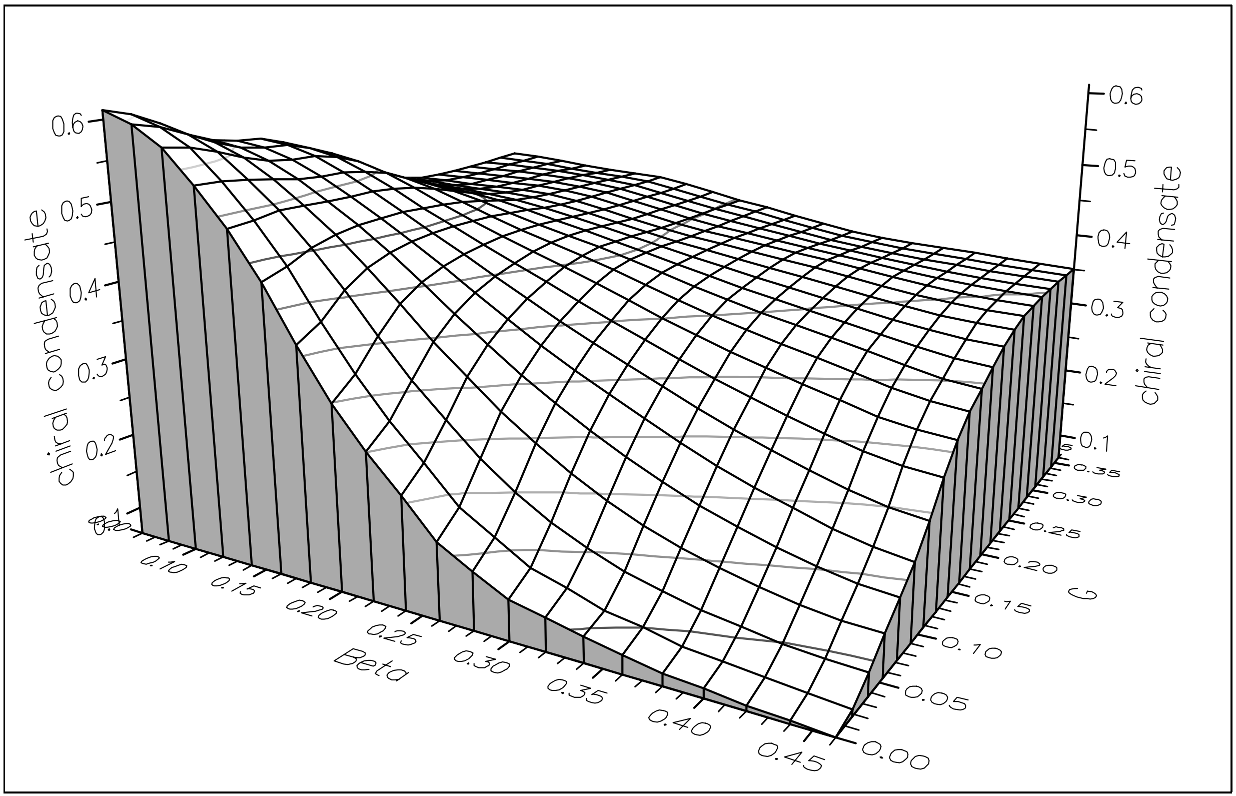


Figure 3.8: $\langle \bar{\chi}\chi \rangle$ at a fermion mass of 0.05 for different values of β and G.

We have attempted to map out the phase diagram for $G > 0$ at a fermion mass of 0.05 on an 8^4 lattice [42]. The results for $\langle \bar{\chi}\chi \rangle$ at a fermion mass of 0.05 are shown in figure 3.8. Here, we have fitted a smooth surface through the data points, without taking account of the sizes of the errors. We discarded 100 trajectories for equilibration, and then averaged over a minimum of a further 200 trajectories. The data used to generate this surface is concentrated in the foreground of the figure so the position of the surface in the high G low β region is suspect. Figure 3.8 indicates that the chiral-symmetry-breaking transition persists for $G > 0$. A line of transitions in the β-G plane is observed to connect the 'on-axis' transitions previously found in references [28, 29] and [43]. The signal for this transition in the plaquette expectation value, already small for $G = 0$, diminishes for $G > 0$ and is almost unobservable in our data.

We have analysed in more detail the transition for $G \geq 0$ at $\beta = 2.0$, a much larger value than for the data in figure 3.8. This corresponds to a very weak gauge coupling and should therefore approximate a pure four-fermi model and can be compared with the pure QED data from the $G = 0$ line. Our results

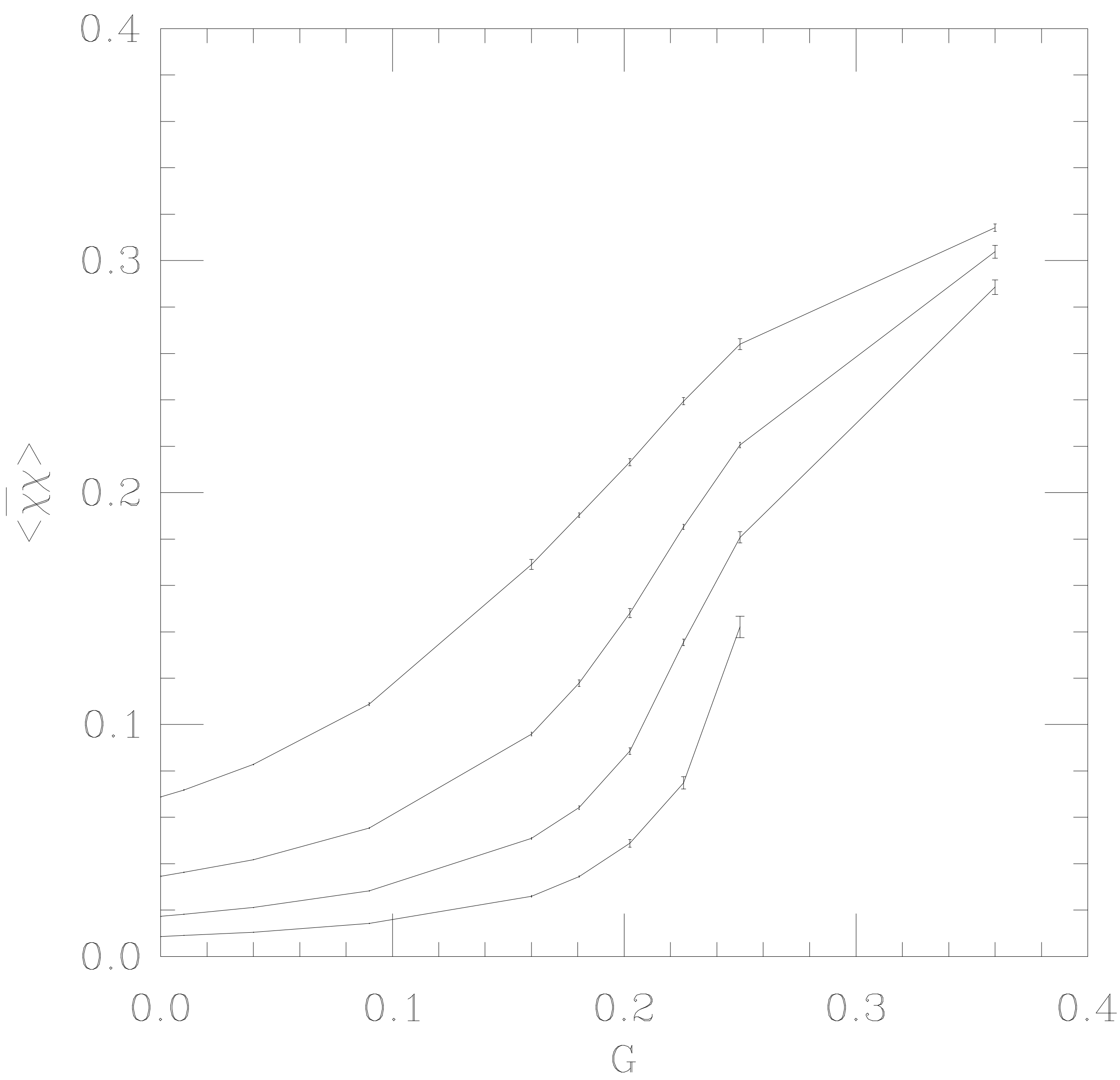


Figure 3.9: $\langle\bar{\chi}\chi\rangle$ against G at $\beta = 2.0$
The points are joined by straight lines to guide the eye.

for the chiral condensate and plaquette expectation values at four fermion mass values are given in table 3.3 and table 3.4. In a plot of $\langle\bar{\chi}\chi\rangle$ vs. G, the transition becomes sharper as the fermion mass is decreased, see figure 3.9.

The fermion-mass-dependence of $\langle\bar{\chi}\chi\rangle$ is plotted in figure 3.10. The data for

$G \leq 0.16$ appears to extrapolate linearly to zero at zero fermion mass, whereas for $G \geq 0.25$ the data indicates the possibility of a non-zero extrapolated value on an infinite lattice. We take this qualitative change in the mass-dependence of the chiral condensate as evidence for a transition between these G values. Our data at $G = 0.2025$ is, therefore, in the critical region. This conclusion is supported by the location of the point of inflexion in the curves of $\langle \bar{\chi}\chi \rangle$ vs. G. This critical value for the four-fermion coupling is slightly less than the value $G = 0.28$ obtained at $\beta = \infty$ in reference [43].

In conclusion, we have obtained indications of phase transitions at zero fermion mass in the lattice $U(1)$-gauge-invariant Nambu-Jona-Lasinio model at $\beta \simeq 0.19$, $G = 0$ and at $\beta = 2.0$, $G \simeq 0.2$ and along a line in the β-G plane connecting these points. This is qualitatively in agreement with analytic predictions for the critical line obtained from the ladder approximation to the Schwinger-Dyson equation for the fermion self-energy. However, the fermion-mass-dependence of the chiral condensate, for fermion masses $0.0125 \leq m \leq 0.1$, does not support a linear extrapolation to zero fermion mass in the broken phase; a crude analysis of our data at the $G = 0$ critical point is consistent with mean-field behaviour.

3.2.1 The gap equation

We have tested the conjecture that there is evidence for non-mean-field critical behaviour in our numerical data using data from two sections approximately transverse to the critical line: one coincides with pure non-compact QED, the other to a four-fermion theory with relatively weak gauge coupling (where large anomalous dimensions are predicted by the approximate Schwinger-Dyson analysis [38]). Because our analysis is at fixed lattice size, it is necessarily crude and we cannot reliably extract predictions for critical exponents. We reject any extrapolation of the data and, instead, study the dependence of the chiral condensate on fermion mass and couplings close to criticality.

We compare our results for $\langle \bar{\chi}\chi \rangle$ with the solution of the gap equation for the pure four-fermion system, with lattice action

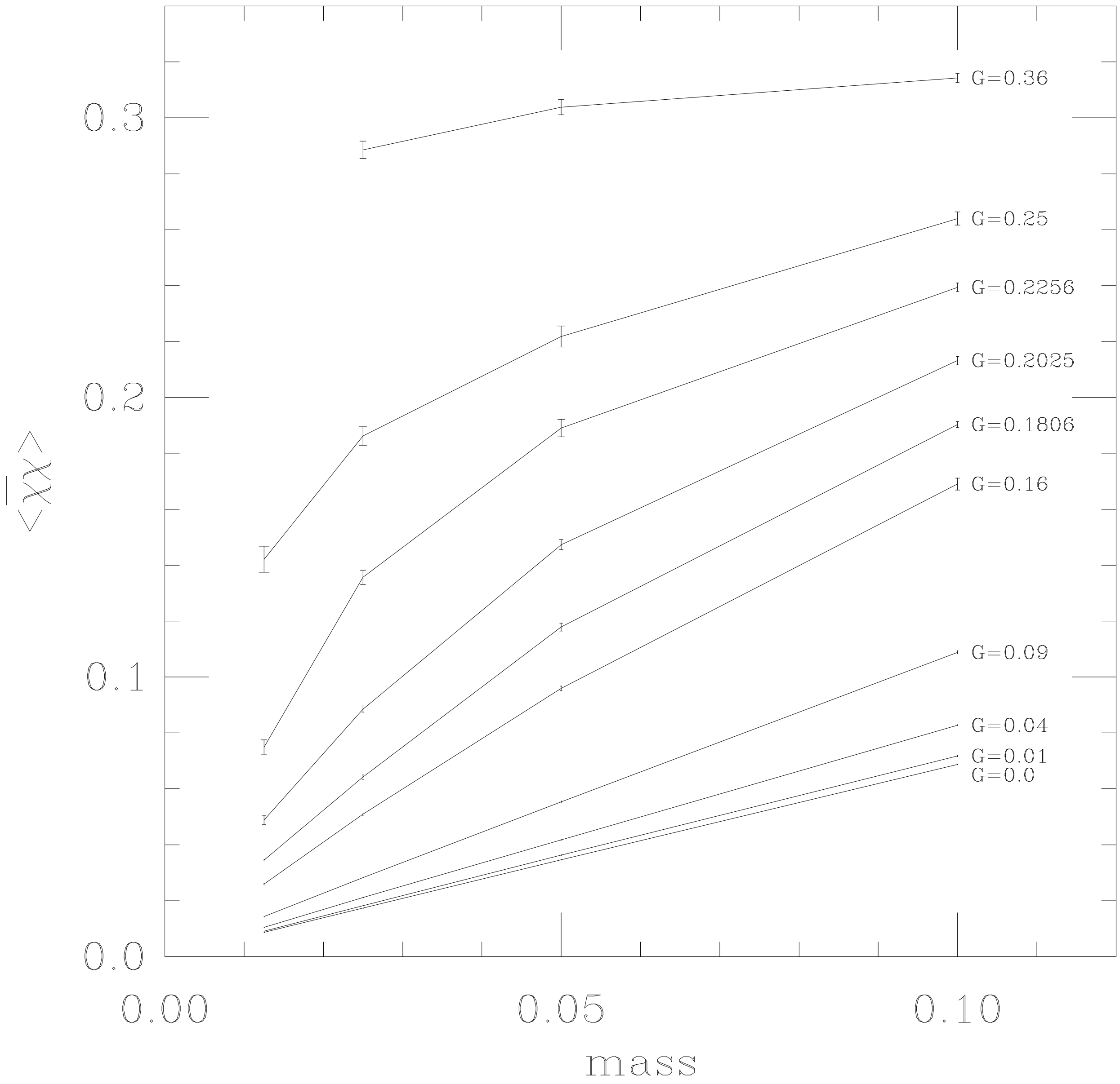


Figure 3.10: Fermion-mass dependence of $\langle\bar{\chi}\chi\rangle$ at $\beta = 2.0$
The points are joined by straight lines to guide the eye.

β	$\langle\bar{\chi}\chi\rangle$			
	$m = 0.0125$	$m = 0.025$	$m = 0.05$	$m = 0.1$
0.06		0.622(6)	0.631(6)	
0.10		0.580(6)	0.581(4)	
0.15		0.428(7)	0.445(2)	0.480(1)
0.16			0.416(2)	0.454(1)
0.17	0.306(3)	0.344(4)	0.382(3)	0.426(1)
0.18	0.249(2)	0.292(5)	0.340(4)	0.400(1)
0.185	0.227(4)	0.267(3)	0.325(3)	0.386(1)
0.19	0.180(1)	0.242(3)	0.308(2)	0.373(1)
0.195	0.150(2)	0.219(3)	0.287(2)	0.360(1)
0.20	0.132(4)	0.196(3)	0.272(1)	0.350(1)
0.205	0.107(2)			
0.21	0.093(3)	0.158(3)	0.238(1)	0.326(1)
0.215	0.080(2)			
0.22				0.305(1)
0.23	0.058(1)	0.1082(7)	0.187(1)	0.283(2)
0.24				0.264(1)
0.26	0.0367(3)	0.0733(7)	0.138(2)	0.233(1)
0.265				0.2267(6)
0.30	0.0277(5)	0.0541(4)	0.107(1)	
0.34			0.088(1)	
0.38			0.0752(5)	
0.42			0.0662(3)	
0.46			0.0613(3)	

Table 3.1: Chiral condensate expectation value at $G = 0$.

β	S_0/β			
	$m = 0.0125$	$m = 0.025$	$m = 0.05$	$m = 0.1$
0.06		4.11(1)	4.11(1)	
0.10		2.366(6)	2.366(6)	
0.15		1.476(7)	1.485(2)	1.509(1)
0.16			1.381(3)	1.405(1)
0.17	1.261(1)	1.271(2)	1.288(4)	1.314(2)
0.18	1.177(2)	1.192(2)	1.208(1)	1.236(2)
0.185	1.146(1)	1.157(1)	1.1730(8)	1.1988(6)
0.19	1.1105(7)	1.123(1)	1.138(2)	1.165(1)
0.195	1.081(1)	1.092(1)	1.108(1)	1.1320(8)
0.20	1.055(1)	1.064(1)	1.0807(7)	1.103(1)
0.205	1.031(1)			
0.21	1.007(1)	1.014(1)	1.026(1)	1.0502(8)
0.215	0.9862(7)			
0.22				1.001(1)
0.23	0.929(1)	0.9315(9)	0.9400(7)	0.9573(7)
0.24				0.9178(9)
0.26	0.8341(7)	0.834(1)	0.839(1)	0.8510(6)
0.265				0.8358(6)
0.30	0.735(1)	0.735(1)	0.738(1)	
0.34			0.6589(8)	
0.38			0.5964(6)	
0.42			0.5429(5)	
0.46			0.5001(7)	

Table 3.2: Plaquette expectation value at $G = 0$.

G	$\langle\bar{\chi}\chi\rangle$			
	$m = 0.0125$	$m = 0.025$	$m = 0.05$	$m = 0.1$
0.0	0.00865(2)	0.01731(5)	0.0346(1)	0.0688(1)
0.01	0.00910(2)	0.01825(6)	0.0363(1)	0.0718(2)
0.04	0.01049(3)	0.02111(8)	0.04177(9)	0.0828(2)
0.09	0.0143(1)	0.0283(1)	0.0554(3)	0.1088(6)
0.16	0.0259(3)	0.0509(4)	0.0959(8)	0.169(2)
0.1806	0.0345(3)	0.0642(8)	0.118(1)	0.190(1)
0.2025	0.049(2)	0.088(1)	0.148(2)	0.213(2)
0.2256	0.075(3)	0.135(1)	0.185(1)	0.239(2)
0.25	0.142(5)	0.181(2)	0.221(1)	0.264(2)
0.36		0.289(3)	0.304(3)	0.314(2)

Table 3.3: Chiral condensate expectation value at $\beta = 2.0$.

G	S_0/β			
	$m = 0.0125$	$m = 0.025$	$m = 0.05$	$m = 0.1$
0.0	0.1221(2)	0.1225(3)	0.1226(2)	0.1226(2)
0.01	0.1226(2)	0.1228(2)	0.1225(2)	0.1227(2)
0.04	0.1226(1)	0.1226(2)	0.1228(1)	0.1226(1)
0.09	0.1228(2)	0.1228(1)	0.1228(2)	0.1230(2)
0.16	0.1231(2)	0.1226(2)	0.1229(3)	0.1233(2)
0.1806	0.1229(2)	0.1229(3)	0.1226(6)	0.1231(2)
0.2025	0.1230(2)	0.1229(3)	0.1229(2)	0.1233(2)
0.2256	0.1236(4)	0.1230(3)	0.1233(2)	0.1236(2)
0.25	0.1237(2)	0.1230(1)	0.1236(1)	0.1238(3)
0.36		0.1240(4)	0.1237(2)	0.1247(3)

Table 3.4: Plaquette expectation value at $\beta = 2.0$.

$$
\begin{aligned}
S &= \sum_x \Big[\frac{1}{2}\sum_\mu \eta_\mu(x)\bar\chi(x)[\chi(x+\hat\mu)-\chi(x-\hat\mu)] \\
&\qquad\qquad\qquad\qquad\qquad\qquad + m_g\bar\chi(x)\chi(x)\Big] \\
&\quad - G_g \sum_{x,\mu} \bar\chi(x)\chi(x)\bar\chi(x+\hat\mu)\chi(x+\hat\mu). \qquad (3.40)
\end{aligned}
$$

The gap equation is a mean field solution to this model. The four-fermion interaction term is replaced by

$$
8G_g\langle\bar\chi\chi\rangle \sum_{x,\mu} \bar\chi(x)\chi(x).
$$

The partition function then becomes a Gaussian functional integral that can be solved to give a self-consistency equation for $\langle\bar\chi\chi\rangle$. The gap equation for this system on an L^4 lattice is [43, 44]

$$
\langle\bar\chi\chi\rangle_g = \frac{1}{L^4}\sum_{p_\mu} \frac{m_g + 8G_g\langle\bar\chi\chi\rangle_g}{(m_g + 8G_g\langle\bar\chi\chi\rangle_g)^2 + \sum_\mu \sin^2 p_\mu} \qquad (3.41)
$$

where $p_\mu = \frac{2\pi n_\mu}{L}$, or $\frac{2\pi(n_\mu+\frac{1}{2})}{L}$, $(n_\mu = 0, \ldots, L-1)$, depending on whether the fermionic boundary conditions in the μ direction are periodic, or anti-periodic, respectively.

The solution of the gap equation exhibits mean-field critical exponents [43, 44]. We regard agreement between the data and the gap equation as indicating that the data contains no evidence of non-mean-field critical behaviour.

In fitting the solution of eq.(3.41) to the numerical data we allow for the following four free parameters:

$$
\begin{aligned}
\langle\bar\chi\chi\rangle_g &= c_{\bar\chi\chi}\langle\bar\chi\chi\rangle \qquad (3.42) \\
G_g &= c_G G + c_0 \qquad (3.43) \\
m_g &= c_m m. \qquad (3.44)
\end{aligned}
$$

The freedom to vary these parameters does not alter the mean-field nature of eq.(3.41). What is at issue is how accurately eq.(3.41) fits the data in what appears to be the critical region.

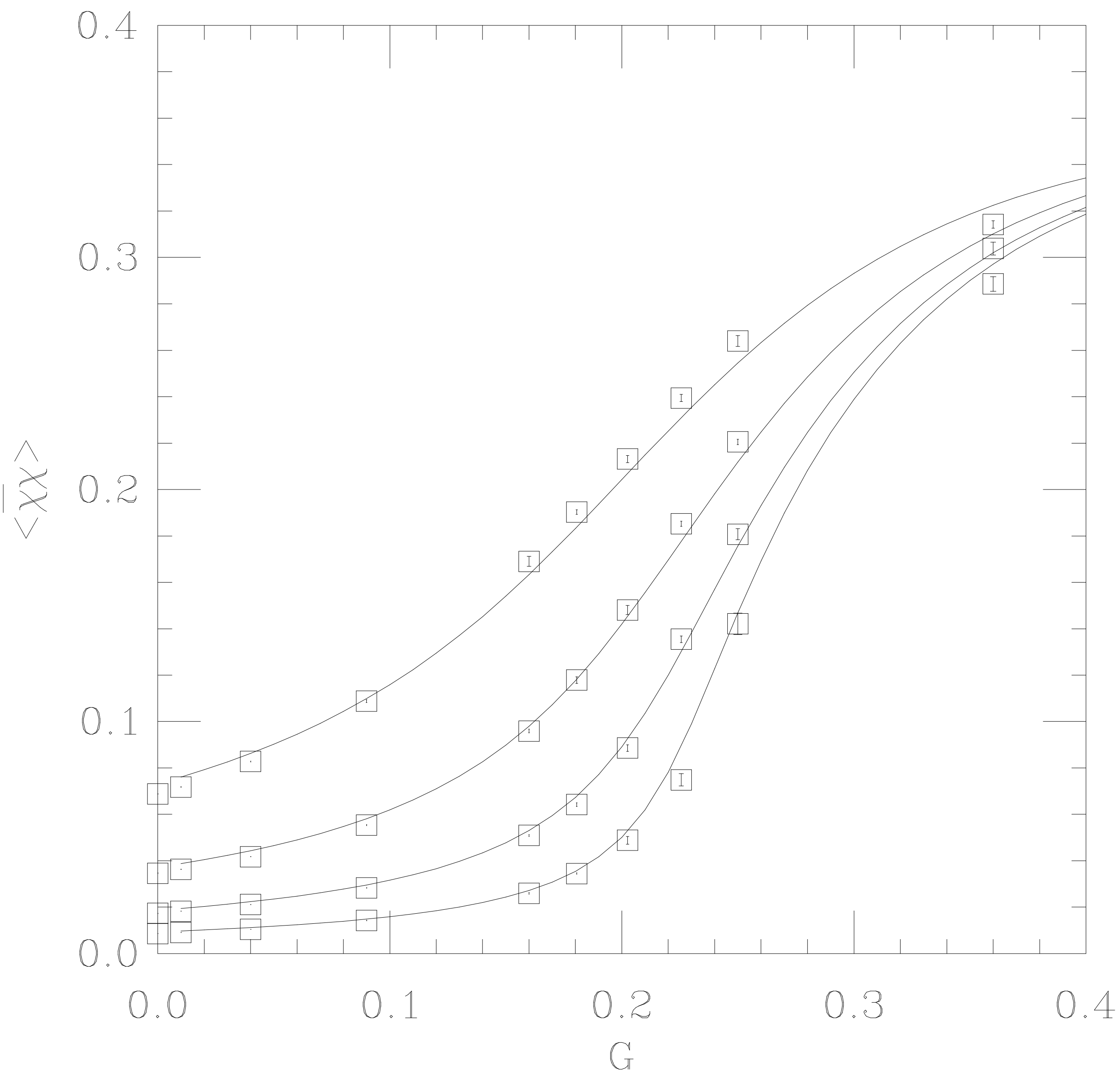


Figure 3.11: $\langle\bar{\chi}\chi\rangle$ against G at $\beta = 2.0$

Mass values shown are $m = 0.1$, 0.05, 0.025 and 0.0125 (from top to bottom). Superimposed is our best 3-parameter fit to the gap equation of a pure four-fermion model.

Both the $G = 0.0$ and the $\beta = 2.0$ data can be successfully fitted to the gap equation [45, 46]. Figure 3.11 shows the gap equation fit to our data at $\beta = 2.0$. Only three of the parameters are fitted, c_m is fixed to be one as it seems unnecessary to vary this parameter to obtain a good fit. The parameters used in the figure are

$$c_G = 0.66 \tag{3.45}$$

$$c_0 = -0.06 \tag{3.46}$$

$$c_{\bar{\chi}\chi} = 0.96 \tag{3.47}$$

.

3.3 The Swendsen Ferrenberg extrapolation

A Monte-Carlo simulation of a lattice gauge theory is an approximation to the full path-integral. A lattice path integral averages over all possible configurations of the lattice fields weighting each configuration by the exponential of its action. In contrast a Monte-Carlo simulation only averages over a representative subset of the possible configurations. These configurations are usually chosen from a distribution corresponding to the weighting factor of the path integral, so that explicit exponential factors need not occur in the average over configurations. Any configuration that occurs in a Monte-Carlo simulation is also likely to be significant for other nearby parameter values. It is therefore possible to use the configurations generated at a fixed set of parameter values to explore a local region of the phase diagram of the system. In particular, if the action is linear in a coupling,

$$S = S_a + \beta S_b \tag{3.48}$$

and we generate configurations using one value of the coupling $\beta = \beta_0$, the results can be extrapolated to $\beta = \beta_1$ by weighting each configuration by an appropriate exponential.

$$\langle A \rangle_{\beta=\beta_1} \approx \sum_{u \in U_{\beta_0}} A(u) e^{(\beta_0 - \beta_1) S_b(u)}, \tag{3.49}$$

where U_{β_0} is the set of configurations generated by a simulation at $\beta = \beta_0$. This extrapolation is obviously limited by the region of configuration space visited by the original simulation. As the extrapolation is taken further away from its starting point, the fraction of configurations in U_{β_0} that contribute significantly will inevitably be reduced. This can only be offset by increasing the statistics of the simulation. This kind of extrapolation has been extensively studied by A. Ferrenberg and R. Swendsen [47], who advocate a single high-statistics simulation at or near a critical point. The advantage of this approach is that each region of configuration space need only be explored once. If a series of separate simulations with similar parameter values are employed, the regions of the configuration space they explore will overlap. This produces redundant effort as each simulation has to explore the overlapping region independently.

A single high-statistics simulation should be located near a critical point, because the significant region of configuration space can be expected to influence the entire scaling region. In addition, the fluctuations on all length scales that occur at a second order phase transition may be an indication that the significant region is larger here than at an arbitrarily chosen set of parameter values. If this is the case then the high statistics needed because of critical slowing down is offset by an increased range for the extrapolation procedure.

The work presented in this section is an attempt to apply this type of technique to the dynamical fermion QED simulations described elsewhere in this chapter. Because of the inherent difficulties of a dynamical fermion simulation, it is impractical to attempt a high-statistics simulation. In fact, in places it is difficult to obtain even adequate statistics. This means that we will only be able to extrapolate short distances and we will still need a large number of simulations to span the entire parameter range of interest. The aim of this work is therefore not to extrapolate the phase diagram from a single simulation, but to improve the quality of the results by combining information from a number of nearby simulations. This is applying the original idea in reverse. As the statistics of an individual simulation are poor, it is no longer a disadvantage for a number of simulations to explore overlapping regions of configuration space; simulations with such an overlap will effectively increase each other's statistics and we get the greatest possible use out of each configuration we generate. If the previous conjecture about critical points is correct, this improvement should be greatest near the critical point, as a greater number of simulations are expected to make a significant contribution in this region. Because of the parallel nature of our simulations, it is easier for us to perform several independent simulations than to run a single simulation for a large number of updates. In principle, it would be possible to generate high statistics at a single point in the phase diagram by performing several simulations with the same parameters. This work was only started after the majority of our data had already been collected, but even if that had not been the case, distributing the simulations across the phase diagram saves us from having to rely totally on the extrapolation.

We therefore have to perform an interpolation based on a number of simulations at different points in the phase diagram. The technique for doing this is much less straightforward than for a single simulation and is again due to Swendsen and

Ferrenberg [48]. We assume an action of the form

$$S_{total}(u) = S_0(u) + \beta S(u), \tag{3.50}$$

where u represents the gauge configuration and β is the parameter we intend to vary in the extrapolation. The lattice QED action is of this form, where $S(u)$ is the gauge action and S_0 is the fermionic contribution. The partition function can be written as

$$Z(\beta) = \sum_u e^{S_{total}(u)} = \sum_S W(S) e^{\beta S}, \tag{3.51}$$

where all necessary constants are absorbed into β and $W(S)$ is the density of states. Consider R simulations at $\beta = \beta_1, \ldots, \beta_R$. Data from these simulations is stored in histograms $\{N_i(X)\}$ where $N_i(X)$ is the number of configurations from the simulation at $\beta = \beta_i$ with $S = X$. The total number of configurations in a simulation is given by

$$n_i = \sum_S N_i(S). \tag{3.52}$$

An approximate partition function can be calculated for each of the simulations:

$$z_i(\beta) = \sum_S N_i(S) e^{(\beta - \beta_i)S}. \tag{3.53}$$

The normalisation of z_i is obviously not the same as that for Z because z_i is roughly proportional to n_i We can relate this to the true partition function using

$$\overline{z_i(\beta)} \propto Z(\beta), \tag{3.54}$$

where the bar represents an average over all possible simulations with $\beta = \beta_i$ and $n = n_i$.

$$\overline{z_i(\beta)} = \sum_S \overline{N_i(S)} e^{(\beta - \beta_i)S}, \tag{3.55}$$

$$\overline{N_i(X)} = n_i \sum_u \frac{\delta(S(u), X) e^{S_0(u) + \beta_i S(u)}}{Z(\beta_i)}, \tag{3.56}$$

that is $N(X)$ is n_i multiplied by the fraction of configurations with $S = X$. This gives us a new expression for $\overline{z_i(\beta)}$,

$$\overline{z_i(\beta)} = \frac{n_i}{Z(\beta_i)} \sum_u \sum_X \delta(S(u), X) e^{S_0(u) + \beta S(u)}, \tag{3.57}$$

$$\overline{z_i(\beta)} = \frac{n_i}{Z(\beta_i)} \sum_u e^{S_0(u) + \beta S(u)} = n_i \frac{Z(\beta)}{Z(\beta_i)}. \tag{3.58}$$

We can use this to obtain an expression for $W(S)$:

$$W(X) = \sum_u \delta(S(u), X) e^{S_0(u)}, \tag{3.59}$$

$$W(S) = \frac{\overline{N_i(S)} Z(\beta_i)}{n_i e^{\beta_i S}}, \tag{3.60}$$

$$W(S) = \frac{\overline{N_i(S)}}{n_i} e^{f_i - \beta_i S}, \tag{3.61}$$

where $f_i = \ln Z(\beta_i)$ is the free energy at β_i.

We wish to produce an estimate of $W(S)$ using all R simulations:

$$W(S) \approx w(S) = \sum_{i=1}^{R} p_i(S) \frac{N_i(S)}{n_i} e^{(f_i - \beta_i S)}, \tag{3.62}$$

where $w(S)$ is our estimate of $W(S)$. The factor $p_i(S)$ is a weighting factor given to $N_i(S)$. For proper normalisation

$$\sum_{i=1}^{R} p_i(S) = 1. \tag{3.63}$$

This also gives us

$$\overline{w(S)} = \sum_{i=1}^{R} p_i(S) W(S) = W(S). \tag{3.64}$$

We wish to choose the values of $p_i(S)$ so as to minimise the error in $w(s)$. The error in $N_i(S)$ is expected to go as

$$\delta^2 N_i(S) = g_i \overline{N_i(S)} \tag{3.65}$$

because $N_i(S)$ has Poisson statistics. The constant g_i represents the correlations between configurations in the simulations. The simulations may have different correlations so g_i is necessary to correctly weight data from different simulations. This can be written as

$$\delta^2 N_i(S) = g_i n_i W(S) e^{(\beta_i S - f_i)}, \tag{3.66}$$

from equation 3.61 The error in $w(S)$ is therefore given by

$$\delta^2 w(S) = \sum_{i=1}^{R} \frac{p_i(S)^2}{(n_i e^{(\beta_i S - f_i)})^2} \delta^2 N_i(S), \tag{3.67}$$

$$\delta^2 w(S) = W(S) \sum_{i=1}^{R} \frac{p_i(S)^2}{\frac{n_i}{g_i} e^{(\beta_i S - f_i)}}. \tag{3.68}$$

This can be minimised subject to the normalisation constraint 3.63 using Lagrange multipliers:

$$\frac{2p_i(S)}{\frac{n_i}{g_i} e^{(\beta_i S - f_i)}} - \lambda = 0, \tag{3.69}$$

$$p_i(S) \propto \frac{n_i}{g_i} e^{(\beta_i S - f_i)}, \tag{3.70}$$

$$p_i(S) = \frac{\frac{n_i}{g_i} e^{(\beta_i S - f_i)}}{\sum_{j=1}^{R} \frac{n_j}{g_j} e^{(\beta_j S - f_j)}}. \tag{3.71}$$

If we now define

$$P(S, \beta) \equiv W(S) e^{\beta S} \tag{3.72}$$

then

$$e^{f_i} = \sum_S P(S, \beta_i). \tag{3.73}$$

Our best estimate for $P(S, \beta)$ is obtained by substituting $w(S)$ for $W(S)$:

$$P(S, \beta) \approx \frac{\sum_{i=1}^{R} g_i^{-1} N_i(S) e^{\beta S}}{\sum_{j=1}^{R} n_j g_j^{-1} e^{(\beta_j S - f_j)}} \tag{3.74}$$

and we can estimate $\{f_i\}$ by iterating the last two equations to a self consistent solution. The expectation value of an operator as a function of β can now be calculated using

$$\langle A(S) \rangle(\beta) = \frac{\sum_S A(S) P(S, \beta)}{\sum_S P(S, \beta)} \tag{3.75}$$

It is worth noting at this point that it is not actually necessary to place the data in histograms: as the weighting factor $P(S, \beta)$ occurs inside a sum over S, we can consider each gauge configuration as belonging to a separate bin of the histogram and sum over gauge configurations instead. The weighting factor for each configuration now becomes

$$P(u, \beta) = \frac{1}{\sum_{j=1}^{R} n_j \frac{g_u}{g_j} e^{((\beta_j - \beta) S(u) - f_j)}} \tag{3.76}$$

where g_u is the g factor from the simulation that generated the gauge configuration. Apart from g_u this weighting factor is independent of which simulation generated the gauge configuration: all the values of β_j are treated equally in the expression. This seems a little strange at first until we consider the relationship between this extrapolation method and the lattice path integral. In the path integral all possible configurations contribute at all values of β: all of the physics comes from the weighting factors. If the path integral is approximated using a finite number of configurations chosen from an arbitrary distribution, the physics still resides in the weighting factors, but the weights also have to compensate for the distribution the configurations were chosen from. In a normal Monte-Carlo simulation, the distribution is chosen so that the weighting factors are all equal. In this extrapolation procedure, we can treat all of the configurations as if they come from a single distribution that depends on the parameters $\{\beta_i\}$ $\{f_i\}$ $\{g_i\}$ and $\{n_i\}$.

It is important to calculate the error on the extrapolation. The variance can be calculated directly:

$$\sigma^2(A, \beta) = \langle (A - \langle A \rangle)^2 \rangle \tag{3.77}$$

The error in the mean can be calculated as follows

$$\langle A \rangle = \frac{\sum_u P(u, \beta) A_u}{\sum_u P(u, \beta)} \tag{3.78}$$

$$\delta^2 = \sum_u \left(\frac{\partial \langle A \rangle}{\partial A_u}\right)^2 \sigma^2 \tag{3.79}$$

$$\delta = \frac{\sigma}{\sqrt{n_{eff}}} \tag{3.80}$$

$$n_{eff}(\beta) = \frac{\left(\sum_u P(u, \beta)\right)^2}{\sum_u P(u, \beta)^2} \tag{3.81}$$

n_{eff} is a measure of the effective statistics at each value of β and is a good measure of where the predictions are valid and of how much the predictions from the original simulations have been improved. This calculation ignores any errors in the calculation of $\{f_i\}$, ignores rounding errors and relies on the g factors to account for any correlations between configurations. It will therefore tend to slightly underestimate the true error.

This technique was applied to our data for pure noncompact QED to produce extrapolated curves in β. In these plots, the inverse of the Hybrid Monte-Carlo acceptance rate was substituted for the g factor. As this procedure only depends on the ratios of the g factors and as all the simulations used a molecular dynamics trajectory of unit length, this was thought to be a reasonable estimate. In all cases,

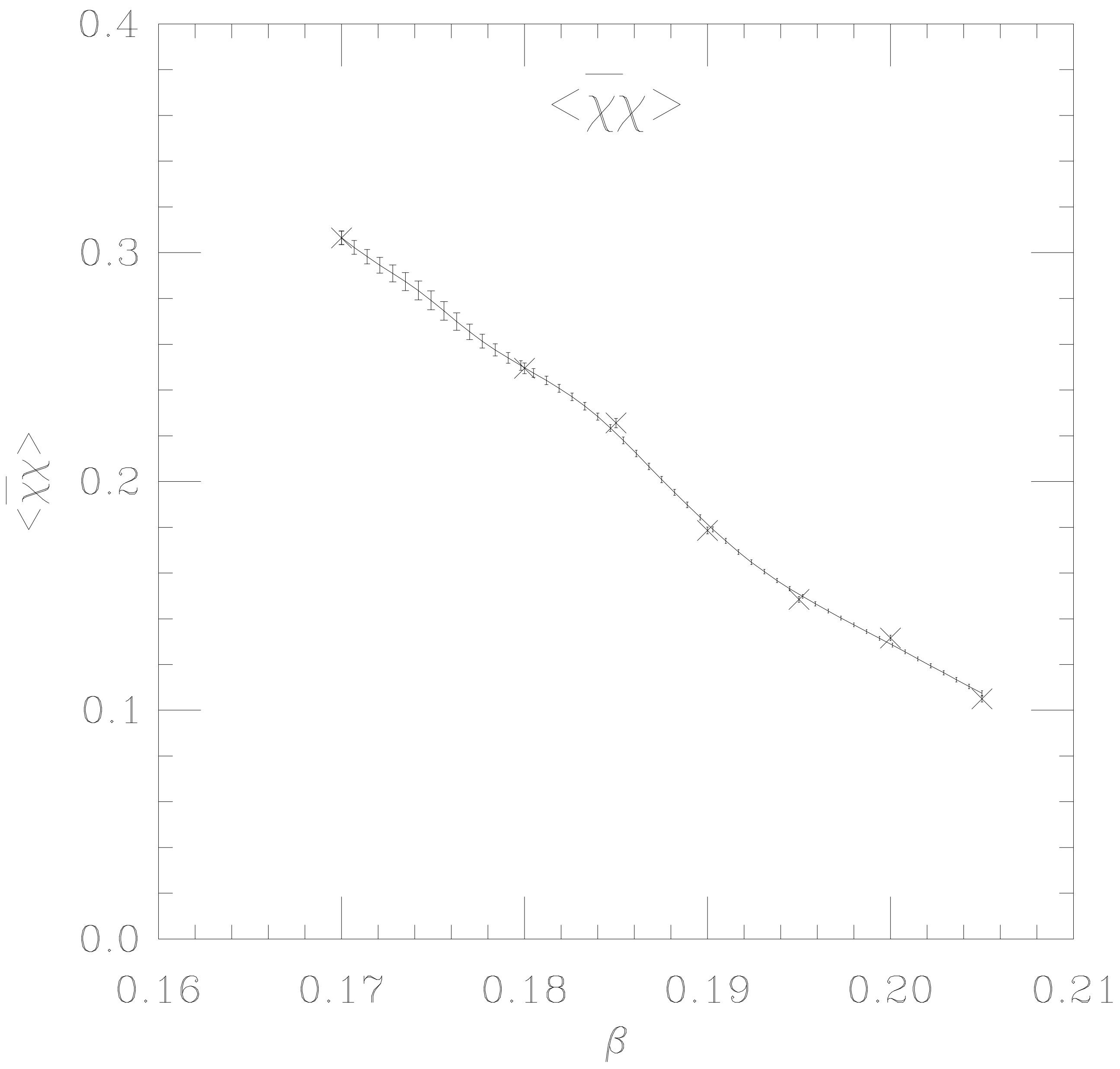


Figure 3.12: Extrapolated plot of $\langle\bar{\chi}\chi\rangle$ at $m = 0.0125$

This plot shows the extrapolated curve for $\langle\bar{\chi}\chi\rangle$ at $m = 0.0125$. The data and naive errors from the original simulations are also shown.

300 sweeps were discarded for equilibration. The calculations were performed on a Sun4 workstation using IEEE 64-bit arithmetic. An extrapolated curve for $\langle\bar{\chi}\chi\rangle$ was plotted which proved to be consistent with the original data; see figure 3.12.

A plot of n_{eff} see figure 3.13, showed that the resulting curve had approximately one and a half to twice the effective statistics of the original simulation in the range $\beta = 0.18$ to $\beta = 0.205$ but that the effective statistics are very low around $\beta = 0.175$ so the procedure cannot be trusted in this region.

It is possible to calculate the relative importance of each of the simulations by plotting the fraction of the total weight they provide at each value of β using

$$I_i(\beta) = \frac{\sum_{u \in \mathcal{U}_i} P(u, \beta)}{\sum_{u \in \mathcal{U}_T} P(u, \beta)} \tag{3.82}$$

where $\mathcal{U}_i$ is the set of configurations from the simulation at $\beta = \beta_i$ and $\mathcal{U}_T$ is the total set of configurations. These plots for our data are shown in figure 3.14. Note that this is only a measure of the quality of the simulation *relative* to the other simulations that contribute at that value of β.

We can see that this method can improve our utilisation of data. The effective statistics are increased above that obtained for each individual simulation and observables can be calculated for any value of the coupling provided that the simulations are spaced closely enough and have sufficient statistics. There are, however, several practical difficulties to be overcome. The main difficulty is that the technique requires a very high machine precision. The exponential form of the weights means that they can become very large and unless steps are taken to control them, they can overflow the floating-point format or introduce large rounding errors. As the action occurs in the exponent and is proportional to the lattice size, this problem would become greater if the lattice size is increased. The values of $\{f_i\}$ are not uniquely defined; it is possible to add a constant offset to each f_i and leave the results unchanged. This effectively rescales every $P(u, \beta)$ by the log of the offset but, as $\langle A \rangle$ is normalised by the sum over $P(u, \beta)$, this does not change the results. Once the separations between the values of $\{f_i\}$ have been calculated, a separate offset can be chosen for each value of β so as to control the exponential factors as much as possible. The optimal offset would be one that properly normalised $P(u, \beta)$ so that $\sum_u P(u, \beta) = 1$, $f(\beta) = \ln \sum_u P(u, \beta) = 0$ at the β being simulated. This could be achieved by first selecting an offset to control the largest $P(u, \beta)$ factor (the configuration with the extreme value of the action, that is the one closest to the classical solution) then iterating the calculation of the free energy a number of times, subtracting the previous value each time until the

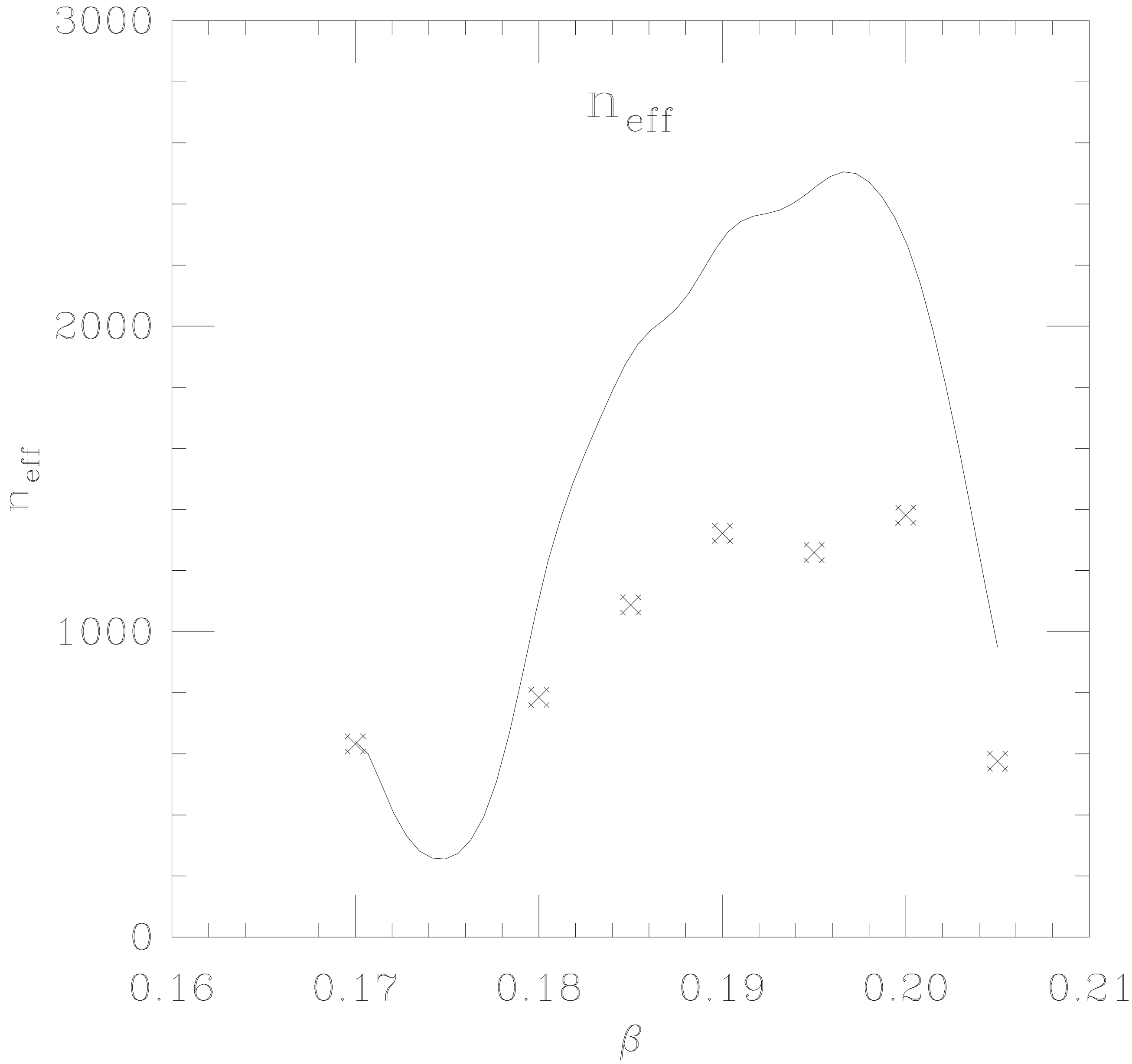


Figure 3.13: Extrapolated plot of n_{eff} at $m = 0.0125$

This plot shows the extrapolated curve for n_{eff} the effective number of configurations for $m = 0.0125$. The number of configurations from the original simulations are also shown.

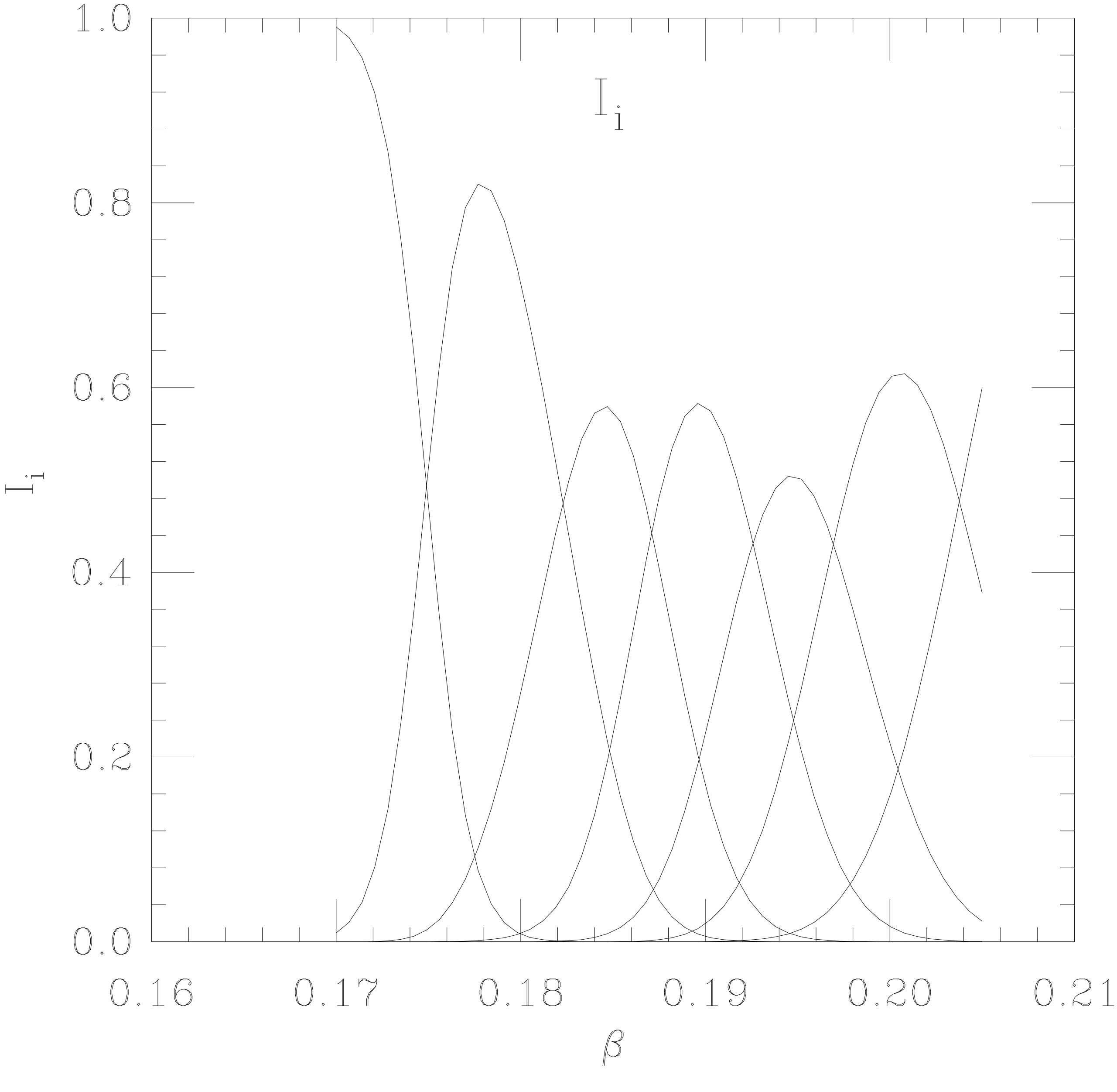


Figure 3.14: Extrapolated plots of I_i at $m = 0.0125$
This plot shows the the relative importance of each of the simulations for different values of β. Each curve has a peak at at the value of β where the corresponding simulation was performed. The simulations were performed at $\beta = 0.17$, 0.18 , 0.185 , 0.19 , 0.195 , 0.20 and 0.205 .

free energy becomes zero. A single iteration should be sufficient for all practical purposes. For the work presented here, a set of heuristics was good enough to calculate the offset except when calculating n_{eff}. This is because the heuristics were unable to regulate the $\sum_u P(u,\beta)^2$ term. The free energy was already known at this point and it was used to calculate the correct normalisation for $P(u,\beta)$.

The remaining awkwardness is that $P(u,\beta)$ consists of the inverse of a sum of exponentials. Potentially we can have configurations with $P(u,\beta) = 0$ to machine precision. The only way this can happen is for one of the exponentials to be infinite, again to machine precision. As the Sun4 floating point implementation produces NaN (Not a Number) when it calculates the reciprocal of Inf, the value of the exponents must be explicitly checked and $P(u,\beta)$ set to zero if any are greater than a cut-off value. This problem did not occur in our data provided the offset in f was chosen sensibly.

3.4 The simulation program

This work was carried out using the Edinburgh Concurrent Supercomputer (ECS), a large Meiko computing surface built out of T800 transputers. The ECS contains over 400 T800s divided into a number of domains (see figure 3.15). The simulation program used 17 processors to simulate an 8^4 lattice using the hybrid Monte-Carlo algorithm This code was written in Occam [12] and was developed from the program used to develop the hybrid Monte-Carlo algorithm [6]. A number of optimisations were introduced into this code as part of this project. Sixteen of the processors are wired as a four-dimensional binary hypercube see figure 3.16. Each of these processors is responsible for a 4^4 sublattice. The remaining processor is inserted into one of the links of this hypercube and acted as a controlling processor for the program. On domains larger than seventeen processors, several of these seventeen-processor building blocks are replicated to produce a program capable of simulating several lattices at once.

Each copy of the program reads a separate parameter file. Interactive commands such as those requesting program-shutdown, checkpoint, or a re-scan of the para-

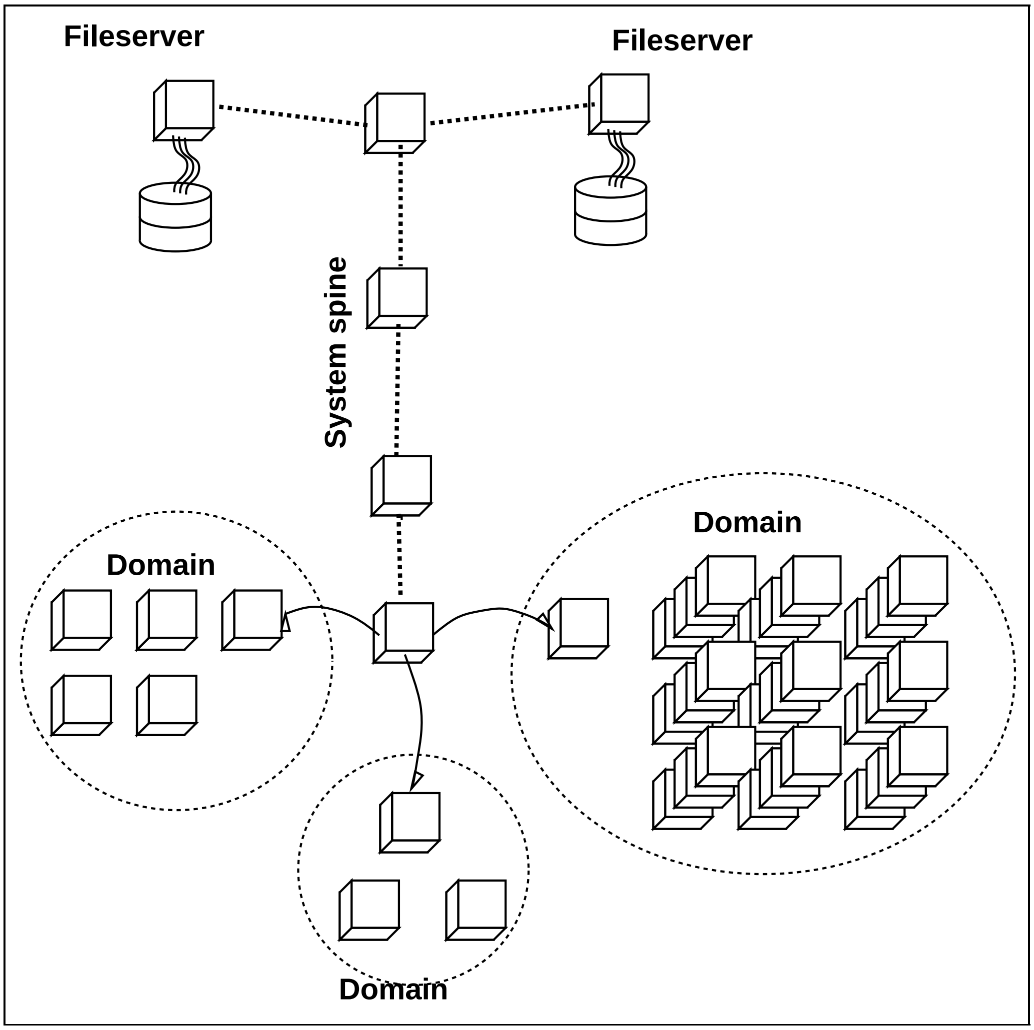


Figure 3.15: The Edinburgh Concurrent Supercomputer
The ECS is divided into a number of domains. Domains are only used by a single user at a time. System services are provided by the fileservers and a tree of system processors.

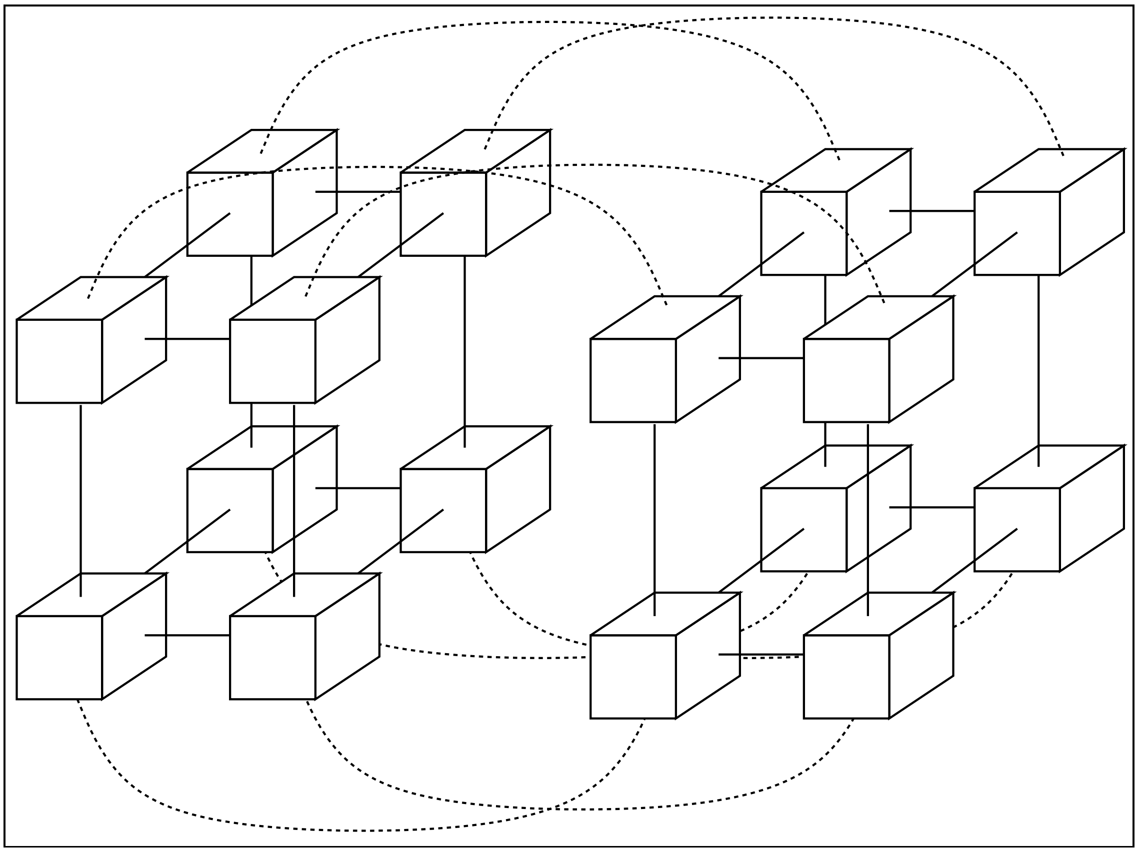


Figure 3.16: A binary hypercube of Transputers.

A binary hypercube can be constructed using 16 Transputers. This uses all four links on all of the processors so an additional processor must be inserted into one of the connections to provide free links to connect with the outside world.

meter files are broadcast to each copy of the program. The program therefore exhibits two separate levels of parallelism. Each simulation is distributed over seventeen processors using a geometric decomposition, and on larger domains several lattices could be simulated as independent tasks. The program was designed to simulate dynamical fermions where almost all of the time is spent inverting the fermion matrix. The difficulty of this operation increases very rapidly with the size of the lattice and had a major effect on the design of the program. The parallelisation scheme had to be as efficient as possible for small lattice sizes $\approx 8^4$ but the efficiency for a large lattice was of little importance.

Because the transputer only has four bi-directional communication links, the 16 processor hypercube is the largest fully connected four-dimensional grid of transputers that can be constructed. This represents a particularly good topology for a lattice simulation. The surface area to volume ratio for each processor is minimised so the ratio of communication to calculation is also minimised, and all of the inter-processor communication takes place between directly connected processors.

There are two basic approaches that could be used to increase the number of processors per simulation above 16. The first is to use a two-dimensional array of processors and to distribute only two of the lattice dimensions. This would produce a much larger surface area to volume ratio for each processor, so the relative communication requirement for the program will become much larger. For a 8^4 lattice, even 16 processors wired as a 4×4 processor array will have reached the situation where every lattice site is a boundary site for the two directions where communication takes place. In addition, the finite number of lattice sites places an upper limit on the number of processors that can be used. If we are only distributing two of the lattice dimensions, then we are limited by the number of sites in a two-dimensional plane of the lattice, so this limit will be reached much faster. The absolute maximum number of processors that could be used to simulate a 8^4 lattice using this scheme would be 64, and the surface area to volume effect would mean that such a program would run at much less than four times the speed of the 16-processor hypercube.

The other approach that could be used to increase the number of processors is to maintain a low surface area to volume ratio by using a 3 or 4 dimensional decomposition and relax the condition that processors controlling neighbouring regions of the lattice must be directly connected. The version of the Occam language available at the time this work was carried out, did not provide automatic message routing. Messages could only be communicated to directly connected processors if explicit message passing code was introduced into the program. On T800 Transputers this kind of message passing code always incurs some reduction in performance. Processor cycles must be used to make decisions about message routing and to copy transient messages to and from message buffers. The reduced message size due to the improved surface area to volume ratio is offset by the larger number of messages that would have to pass through each processor. If a parallel processing system has a general purpose message passing system, it is much easier to program, as the details of processor topology and connectivity need not be addressed by the application programmer. When this project was started, efficient message routing code was not widely available, so a message passing approach would have been much harder to implement. This is no longer the case, as efficient message routers such as CStools [49] and Tiny [50] have since become available. Even if such software had been available at the time, a message passing solution would have been less efficient, because the routing software would have to run on the same processor as the application code and compete with it for processor cycles. This would not be a problem on a hardware platform that implements the message passing using separate hardware. For example, the Meiko MK086 processing node uses an Intel i860 as the main processor with two T800 transputers dedicated to message routing.

Because we were investigating a phase diagram, it was necessary to perform a large number of independent simulations at different parameter values. It was therefore always possible to utilise large numbers of processors by replicating the basic 17-processor unit. It would have been possible to simulate a separate lattice on each processor. This would have been computationally efficient, but would not have been very practical. The largest domain in the ECS contains 131 processors; this is sufficient to run 7 simulations in parallel. Even if we ignore all unused and support processors this is equivalent to $7 * 16 = 112$ processors. As the phase diagram was not known when we started the simulations it was possible to use the results from previous simulations to refine our original guess about which regions of the phase diagram were of interest. It was therefore much more informative to perform 7 relatively fast simulations than 112 or 131 relatively slow ones. The fermion matrix inversion takes different numbers of iterations in different parts of the phase diagram. This means that the simulations did not all run at the same speed. With only a handful of simulations to look after, it was possible to re-allocate processors as various simulations reached an acceptable level of statistics. This would have been too time consuming a task if there had been a hundred simulations running concurrently.

3.4.1 Optimising the program

Transputers were designed to implement the Occam language efficiently. There is a very simple correspondence between an Occam program and the machine instructions it compiles to. There is little reason to program the transputer in assembly language; most optimisations can be implemented almost as efficiently in Occam as in assembly language. The current compilers do not make any attempt to optimise the code they produce. It is therefore possible to improve a program manually by changing the Occam code. In the QED simulation program, all of the inner loops of the low level routines were unrolled by a factor of 16. In a normal program loop, it is quite common to use as many instructions to perform the loop as are used to perform the calculation. By replicating the body of the loop a number of times the fraction of useful instructions can be increased. This is a very common form of code optimisation that is often performed automatically by optimising compilers. The factor of 16 was chosen because of a peculiarity of the Transputer instruction set. The basic Transputer instruction is only a byte long. Four bits of the instruction encodes one of 16 basic operations and the other four bits form an argument for the operation. One of these 16 basic operations is used to change the meaning of the next instruction so that a wider variety of operations becomes available. Another basic operation is the *prefix* operation. The prefix operation is used to increase the number of data bits available to the next instruction. If an instruction only needs 4 bits of data it can be encoded as a single byte; for each additional 4 bits it requires it must be prefixed by an additional prefix operation. The low level loops were unwound by a factor of 16 because any greater factor would have required prefix operations to perform vector indexing inside the loop. It was not possible to avoid this vector indexing because Occam does not have pointer data types. As can be seen from the first two columns of table 3.5, code without this modification takes approximately one and a half times as long to perform a CG iteration. The exception to this is for a 4^4 lattice size. In this case, unwinding the loops makes the code run slower. This is because each processor is only updating 16 lattice sites; the modifications are therefore introducing extra instructions for no useful purpose.

The Occam language provides intrinsic support for parallel execution. The *PAR* construct allows the programmer to specify a number of operations that are to execute concurrently. In an ideal world, any set of mutually independent operations could be specified as happening in parallel and the compiler would make the decision about the most efficient implementation. Unfortunately, the available Occam compilers are far from optimal. They perform a direct transliteration from Occam to Transputer instructions without making any modifications, so a program written in such a way would result in a large number of processes running on each processor. The transputer is designed to support large numbers of concurrent processes and will probably handle such a program better than any other processor would, but there is always some overhead when the processor switches between different tasks, so a single sequential piece of code is still much more efficient. It is therefore important to reduce the number of concurrent processes running on a single Transputer as much as possible. The multi-tasking capability of the Transputer is still important as it provides a way of making sure that the inter-processor communications are run efficiently. The Dslash procedure has to calculate the covariant derivative in all four directions and then combine the results. Each derivative is completely independent of the the others, so they can be calculated in serial or in parallel. The program timings (see table 3.5 Serial Dslash) show that there is a small improvement in performance $(5-10\%)$ if they are calculated in parallel. If the directions were calculated sequentially, only a single communication link would be in use at any one time. A Transputer memory cycle can be used directly by the current process, or it can be used by one of the links. When all four links are run simultaneously each memory cycle is more likely to be used usefully.

Size	Standard code	Un-optimised math	Serial Dslash	No on-chip mem
4^4	118.8 ± 2.1	110.9 ± 2.1	126.4 ± 2.1	171.9 ± 2.2
8^4	807.4 ± 2.3	1177.2 ± 3.0	864.7 ± 1.5	1028.8 ± 1.7
12^4	3747.3 ± 1.1	5548.3 ± 0.8	3937.4 ± 1.2	4759.2 ± 1.5
16^4	11326.3 ± 0.6	17003.1 ± 1.3	11774.3 ± 0.5	14325.4 ± 1.9
20^4	27079.2 ± 12.9	40728.7 ± 16.7	27951.0 ± 12.3	34251.09 ± 16.9

Table 3.5: Timings for a CG iteration of the QED program

All timings are given in units of 64 microseconds, this is the clockrate of the internal transputer timer. These are actually times for a full HMC update sweep divided by the total number of CG iterations performed as the HMC algorithm is totally dominated by the CG inversion this is a reasonable measure for a single CG step.

We can model these timing figures using the following equation,

$$T = Al^4 + Bl^3$$

where T is the time taken for a single CG iteration, and l is the linear dimension of the lattice. A represents the time taken to perform the calculation and B represents communication time. Lattice size independent overheads are assumed to be negligible. If this model holds then we should obtain a straight line graph if we plot T/l^3 against l, see figure 3.17.

Apart from the 4^4 lattice, this model seems to work quite well and gives values of $A \approx 0.15$ and $B \approx 0.4$. It is not so surprising that the model breaks down for the 4^4 lattice; as there are only 16 lattice sites per processor in this case, it is no longer sensible to neglect the size-independent overheads. This suggests that the 8^4 lattice is running at approximately 75% efficiency compared to an equivalent single processor program (assuming a single processor program has $B = 0.0$).

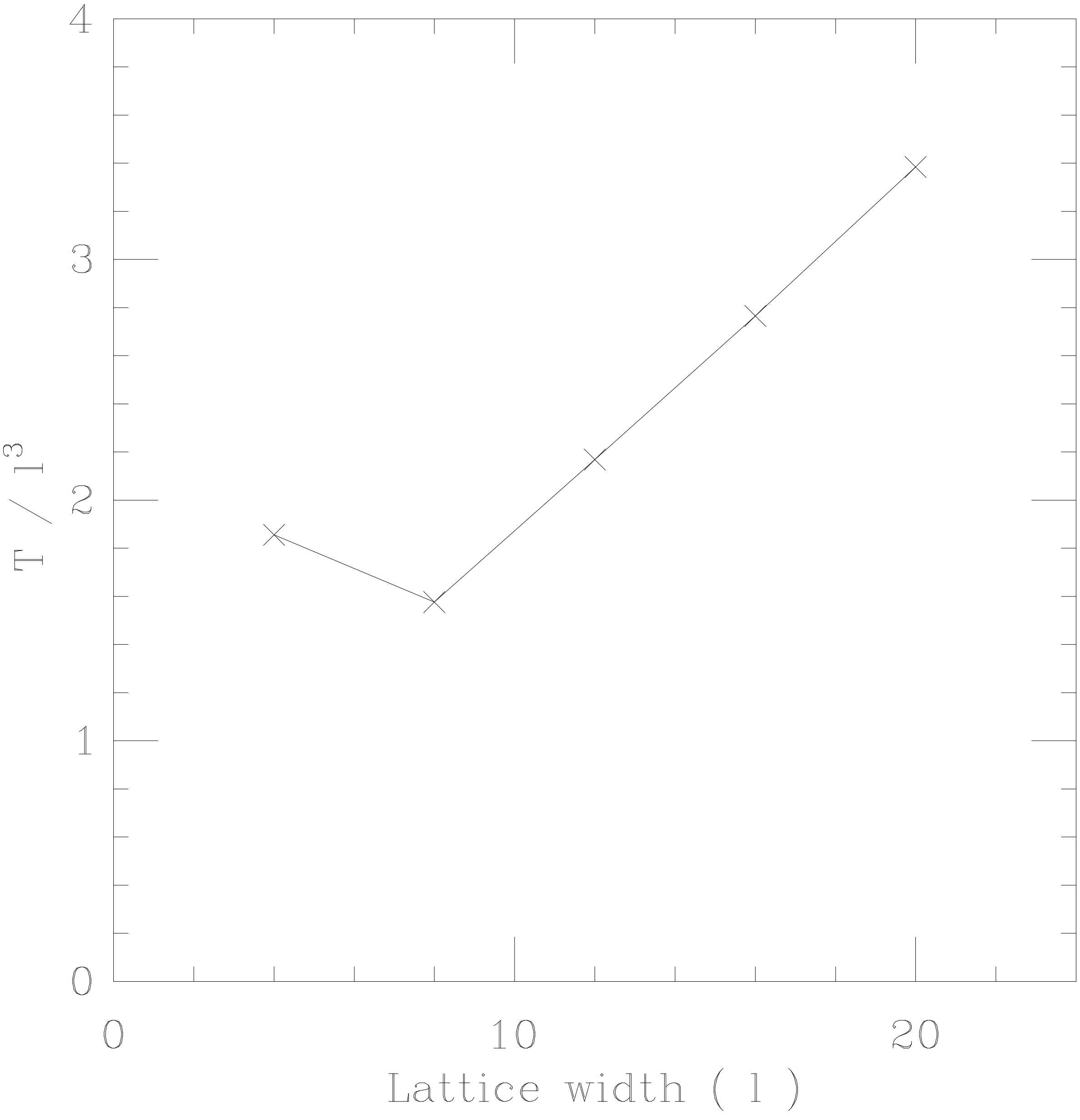


Figure 3.17: Problem size scaling of the QED program
The time for a CG iteration T is plotted as T/l^3 against l, the width of the lattice. If the program behaves as expected This should give a straight line showing the relative importance of communication and calculation.

Chapter 4

QCD on i860 based machines

4.1 The UKQCD collaboration

The computational needs of lattice gauge theory are now greater than can be supplied by conventional academic computing facilities. A collaborative project (UKQCD) has therefore been set up by six British universities centered round a high performance parallel computer based in Edinburgh. The six collaborating universities are Cambridge, Edinburgh, Glasgow, Liverpool, Oxford and Southampton. The computer is a 64 node parallel machine build by Meiko of Bristol. Each node consists of a 40MHz Intel i860 processor and two Inmos T800 transputers. The three processors on the node communicate by shared memory and the nodes communicate using the transputer links. Each node has 16 Mbytes of memory. The i860 is a commercially available microprocessor manufactured by the Intel corporation. This microprocessor is capable of high floating-point performance. The i860 has a peak performance of 80 Mflops, the peak performance of the machine is therefore 5.12 Gigaflops. Even though this peak speed will not be sustainable for normal operations this computer is a very significant resource. Even with such a powerful computer the computational requirements of lattice gauge theory are such that every effort has to be made to utilise the computer as effectively as possible. There are two types of basic optimisation that we can use in this case. The first is to distribute the problem over a number of processors. This kind of optimisation has been discussed in earlier chapters. The second type of optimisation are those specific to the i860 microprocessor. This is the subject of this chapter.

4.2 The i860 microprocessor.

The Intel i860 microprocessor is one of the new generation of high performance microprocessors. It incorporates design features from a number of different types of computer including vector supercomputers and reduced instruction set (RISC) designs. It is currently one of the most complex microprocessors on the market, incorporating over one million transistors in its design. It was designed to give high performance in a number of common application areas including floating-point numerical and graphical applications. The main features of the design are:

- Parallel execution of processor units
- Pipelined floating-point units
- Reduced instruction set core unit
- Large integer and floating-point register sets
- Data and instruction caches
- 64-bit external data path, 128-bit internal data path.
- Paged memory support

The processor has a number of separate processing units. The units of interest for QCD calculations are the floating-point adder, floating-point multiplier and the integer core unit. The core unit performs all integer arithmetic and logical operations, control transfer operations, such as jumps and procedure calls, all data transfers between memory and registers and also performs the system control functions such as cache flush operations. Floating-point arithmetic is handled by the two floating-point units. Each of these units is capable of being run simultaneously. This means that the chip is capable of simultaneously performing a floating-point multiplication, floating-point addition and an address calculation. Instructions where both floating-point units are in operation at the same time are called dual operations. When the core and floating-point units are operating together the chip is said to be operating in "Dual Instruction Mode" (DIM). The maximum performance of the chip can only be obtained when using dual operations in dual instruction mode.

Both of the floating-point units are pipelined. This means that floating-point operations are divided into a number of simpler stages. Each stage is implemented using separate hardware, the results from one stage being fed into the input of the next stage. The pipelines on the i860 are carefully designed so that each stage can complete a single-precision operation every clock cycle, even though a specific operation may have taken a number of cycles to complete. The peak performance of the i860 is therefore two floating-point operations per clock cycle, giving 80 Mflops peak performance for the commercially available 40 MHz chips. For single precision calculations both of the floating-point units have three-stage pipelines; so the result of any calculation emerges from the pipeline three instructions after the operation was started. When operating in double precision the multiplication pipeline has only two stages; however it takes two cycles to advance the multiplication pipeline when performing double precision calculations so the peak performance drops to 60 Mflops. The pipelines in the i860 are very flexible. Unlike some vector mainframes they are not restricted to repeating the same calculation a large numbers of times. Instead it is quite possible to have very different operations in the different stages of the pipeline. There are restrictions on how the floating-point registers are used; it is not possible to read from more than two of the main registers in a single cycle or to store more than one result per cycle. This in turn restricts the way that the floating-point pipelines can be used. In order to use both floating-point units at the same time (dual operation instructions) it is necessary to chain the units together, so that the output of one or both units form one of the inputs of the other unit. A number of special-purpose registers also exist to make dual operation instruction more flexible (see figure 4.1).

There are three special-purpose registers, which can store an operand from one dual operation instruction and then supply this value as an operand for later instructions. Registers Ki and Kr can be used as the first argument to the

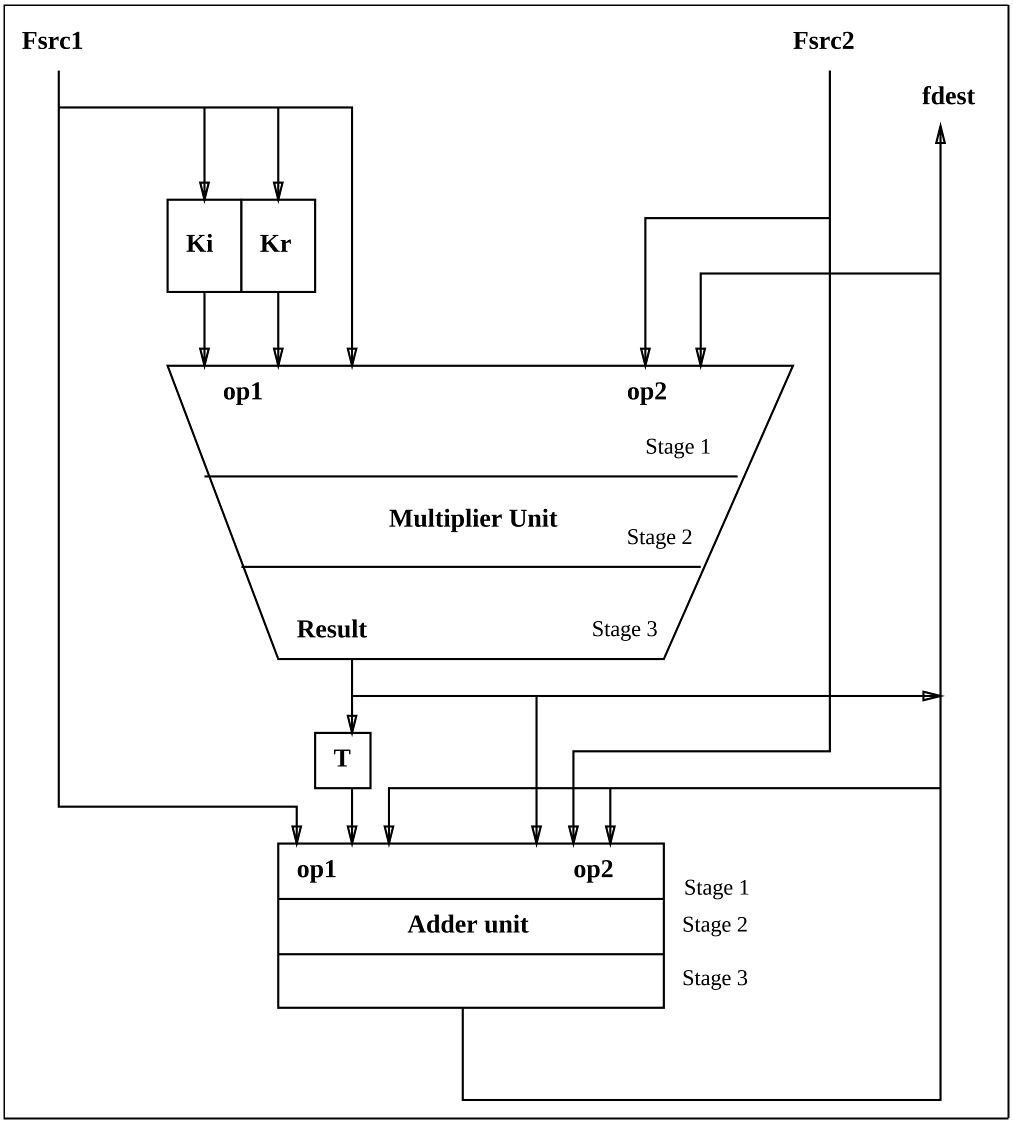


Figure 4.1: The i860 floating-point units

This diagram shows the data paths that connect the two floating-point units and shows the different ways in which they are capable of being chained together multiplier and the transfer register T can be used in a similar way as an argument for the adder. For example, the multiplier unit can be used to generate the product of two numbers and the result emerging from this pipeline can be fed into the adder unit, where it is added to the result emerging from the adder pipeline. This instruction is commonly used for the sum of a sequence of products and it forms the most common type of instruction used in complex arithmetic. The majority of the routines discussed later in this chapter use this configuration exclusively.

Other combinations of the two floating-point units are possible, for example, the multiplier unit can be used to multiply a main register by the contents of one of the K-registers. A second main register can then be added to the result of the multiplier pipe and the result of the adder pipe is stored into a third register. This is the *saxpy* operation.

In single precision it takes three cycles for an operation to progress through one of the pipelines. It is usually convenient to evaluate three similar but separate expressions at the same time. This allows machine instructions to be grouped into sets of three, one from each of the expressions. This ensures that results from the previous instruction of any expression reach the end of the pipeline at the same time as the next instruction of the same expression is executed. The advantage is that each of the expressions may be coded as if the pipelines only had a single stage. A single-stage pipeline is easier to use because the pipeline can be thought of as an extra register. For example, when calculating the sum of a large number of terms the running total can be kept in the adder pipeline, each new term being added to the result emerging from the pipeline.

Unlike the floating-point units, the core unit owes more to recent developments in microprocessor technology than to conventional mainframes. The core unit is a RISC (Reduced InStruCtion set) processor. It only has 41 separate types of instruction. The aim of the RISC style of architecture is to identify the handful of simple instructions that make up over 90% of most programs and to implement these instructions to execute as quickly as possible, at the expense of not implementing any more powerful but less frequently-used instructions. Code written for a RISC processor will in general be longer than that for a CISC (Complex InStruCtion set) processor but should execute more rapidly because each instruction only takes a single clock cycle. In addition the simpler structure often makes it easier for the chip to run at a higher clock speed. The other advantage of a RISC processor is that it is much easier to write code or compilers for it. Some CISC processors have instructions that are hardly ever used because most compilers never recognise the situations where they could be used. Though this advantage does not apply to floating-point-intensive application code it does mean that operating systems and support routines can be easily and efficiently ported to, or written for, the i860. In this respect the i860 is a rather strange hybrid. The floating-point processors are definitely not RISC processors; there are a very large number of floating-point instructions and it is very difficult to produce a compiler that uses the floating-point units efficiently.

Another characteristically RISC feature of the i860 is the set of delayed control transfer instructions. A delayed control transfer instruction is a kind of jump or branch instruction, where, instead of performing a jump when the instruction is executed, it causes a jump to occur a number of instructions later. The advantage of delayed control transfer is that the processor always has advance warning about the sequence of instructions it is going to have to execute later. It can therefore always pre-fetch and decode the correct instructions in advance. Without this delay the only option is to arbitrarily choose one of the possible branches to pre-fetch. If this guess turns out to be wrong the processor will have to halt execution until it has fetched and decoded the correct instruction. The i860 fetches its instructions two cycles before it has to execute them and then decodes the instruction in the cycle before execution. This is another example of pipelined operations in the chip design.

In principle it is possible to extend the RISC concept to cover the floating-point instructions as well. This would mean only implementing the simplest generic floating-point instructions, for example addition and multiplication. Unfortunately, floating-point operations are inherently complicated and cutting down the number of instructions therefore produces a relatively smaller simplification of the design and therefore a smaller increase in performance. Because of this, the potential gains are much less than for the pipeline approach used by the i860.

The i860 also has scalar floating-point instructions that run a single calculation through one of the pipelines as a single operation. These instructions take three cycles to perform a single floating-point operation but at 40 MHz this still produces a performance of 13 Mflops. These are the only floating-point instructions used by most generic compilers that have been modified to produce code for the i860. The i860 is still a fast floating-point chip even when only using these scalar instructions but for highly floating-point intensive applications it is worth finding a way to use the pipelines.

Both the floating-point and the integer units are capable of very high performance. This performance can only be sustained if they can be kept supplied with data at an equivalent rate. Unless a very expensive fast memory system is used, the external memory system is unlikely to be able to keep up with the processor when it is operating at maximum efficiency. This means that the register, cache and memory management systems on the chip are of vital importance to the performance of the processor.

On the i860 all data must be loaded into a register before it can be used and any result must be stored in a register before it can be written to main memory. This provides a simplification of the instruction set. This need not produce any degradation of performance. Registers are loaded and saved by the core unit so these operations may be overlapped with floating-point operations by using dual instruction mode. The i860 has two types of general purpose registers, integer registers and floating-point registers. Integer registers are used to store integer values or memory addresses. The floating-point registers are used to store floating-point data or data for the special graphics instructions. The graphics instructions are not relevant to QCD so they will not be discussed further. The core instruction unit is used to move data between main memory and both types of register.

There are 32 integer registers and 32 floating-point registers. All of these registers are 32 bits wide. The floating-point unit uses the IEEE floating-point standard. It supports both single (32 bit) precision and double (64 bit) precision representations. Double precision numbers are stored in a consecutive pair of floating-point registers. The first register of this pair must be an even-numbered register. The first integer register $r0$ and the first two floating-point registers $f0$ and $f1$ always contain the value zero. This enables the number of instructions to be further reduced; there are some instructions that can perform a number of different functions provided that there is always a register guaranteed to contain the value zero. For example, the addition instructions can be used to set a register to a constant value by adding a constant to the zero register. If the zero registers were not present this operation would need a special *load constant* instruction, or extra cycles would be needed at runtime to generate a zero by subtraction. In addition some instructions, particularly the dual operation instructions, can generate intermediate results that are not always needed; the zero registers are a convenient place to discard these values. Both $f0$ and $f1$ generate zero so that a double precision zero can be generated from the register pair.

Further important features of the chip are the on-chip caches. A cache is an area of faster-than-normal memory used to improve the performance of the memory system. It does this by assuming that memory locations that have been accessed recently are quite likely to be used again in the near future, so it keeps a record of recently-accessed values. There are two caches on the i860, a 4 Kbyte instruction cache and an 8 Kbyte data cache. When the processor attempts to read a memory location it first checks to see if the required data is already in the appropriate cache. If it is, the data will be read from the cache instead of the external memory. A read from the cache only takes a single clock cycle. If the required data is not in the cache the processor will load the appropriate 32 byte long "cache line" into the cache and then continue operation. If the data being loaded is destined for an integer register the core unit is capable of continuing with further instructions until that register is actually used. This means that if there are sufficient instructions between the register load instruction and the first instruction to use the value there will be no delay due to the cache-miss. If the value is destined for a floating-point register then the processor must halt until the desired value has been read from external memory. These cache-misses can be the main reason that a routine does not perform at the peak speed of the processor.

When all the data fits into the data cache the processor is capable of obtaining close to peak performance. Unfortunately in many applications, including lattice gauge theory, the data sets will be larger than the cache and delays due to cache misses will be introduced. In lattice gauge theory calculations the data cache will only be of limited use because the data sets are several times the size of the data cache. By the time the program gets round to reusing the data it will have been replaced by more recent values. The instruction cache on the other hand is extremely useful. Even the code for very complex operations should fit inside the instruction cache. Because of the large loop lengths the code for these operations will be repeatedly called a large number of times. The instruction cache will therefore be used very efficiently. If a floating-point value is only going to be used once, or if it will be a long time before it is used again, then it is possible to bypass the caching mechanism using a pipelined load operation. The pipelined load operations load floating-point values without placing them in the cache. They return the floating-point value that was asked for by the third previous pipelined floating-point load. The load pipeline operates in parallel with the other units of the processor. It performs the outstanding memory accesses during cycles when the external bus is not being used by the other parts of the processor. Up to three pipelined loads may be outstanding at any one time. If the requested values have not been fetched from memory by the time a fourth pipelined load is requested the processor must halt until the outstanding data has been read.

In most of the procedures that we have implemented for lattice gauge theory it is not necessary to use the data cache. There are usually enough registers that we do not need to re-load any element of data within a single procedure. This would suggest that all of the data should be loaded using pipelined loads. This makes things very difficult for the programmer. If a routine has to load data from two separate regions, for example two arrays of numbers that must be multiplied together, then the pipelining makes it very difficult to keep track of which value should be requested at what time. It is much simpler if one set of numbers is loaded using pipelined loads and the other set is loaded using caching loads. There may be some advantage to using a combination of caching and pipelined loads. If a series of memory locations have to be loaded in quick succession (enough to fill the load pipeline) then the exclusive use of either type of load will introduce a delay into the program. If some of these memory accesses are performed using caching loads, the remaining pipelined loads will not fill the load pipeline and the corresponding memory accesses can be widely spaced inside the loop and have a better chance of finding free memory cycles to use.

The i860 has a large number of different functional units on the same piece of silicon. One of the advantages of having everything on a single chip is that it allows a much greater communication rate between the different parts of the processor. There is a practical limit to the number of external connections that can be conveniently supported by a microprocessor. This limits the width of the external data bus. On the i860 the external data bus is 64 bits wide. There is less limitation on internal data paths. The communication bus that connects the data cache to the floating-point register set on the i860 is 128 bits wide (see figure 4.2).

The wide internal data bus enables the chip to load up to four registers in a single instruction. This can be used for complex numbers, double precision numbers, double precision complex numbers etc. The disadvantage of putting everything on a single chip is that there is only room for relatively small caches. This is less of a disadvantage for lattice gauge theory than for some other applications, because it is unlikely to benefit significantly from any kind of data cache.

4.3 Software development on the i860.

The i860 is a very new design of processor. This means that software development tools are still in the early stages of development. Of particular interest are the compilers. A number of compilers already exist for the i860. However most of these are general purpose compilers that have been modified to generate code for the i860, for example the "Green Hills" compilers [51]. Because these are generic compilers designed to produce code for a wide variety of processors they do not support the unique features of the i860 such as the dual instruction mode and the pipelined modes of the floating-point units. This is no problem for programs that use few or no floating-point operations but is inadequate for floating-point-intensive applications. A number of conventional vectorising compilers are also being ported to the i860. Most vector machines implement very similar sets of vector operations. These are usually low-level generic operations, for example, adding two vectors element by element or multiplying all the elements of a vector by a constant. Each of these simple operations is specially supported by the vector hardware. A conventional vectorising compiler identifies those parts of a program that are equivalent to these operations and replaces that part of the code

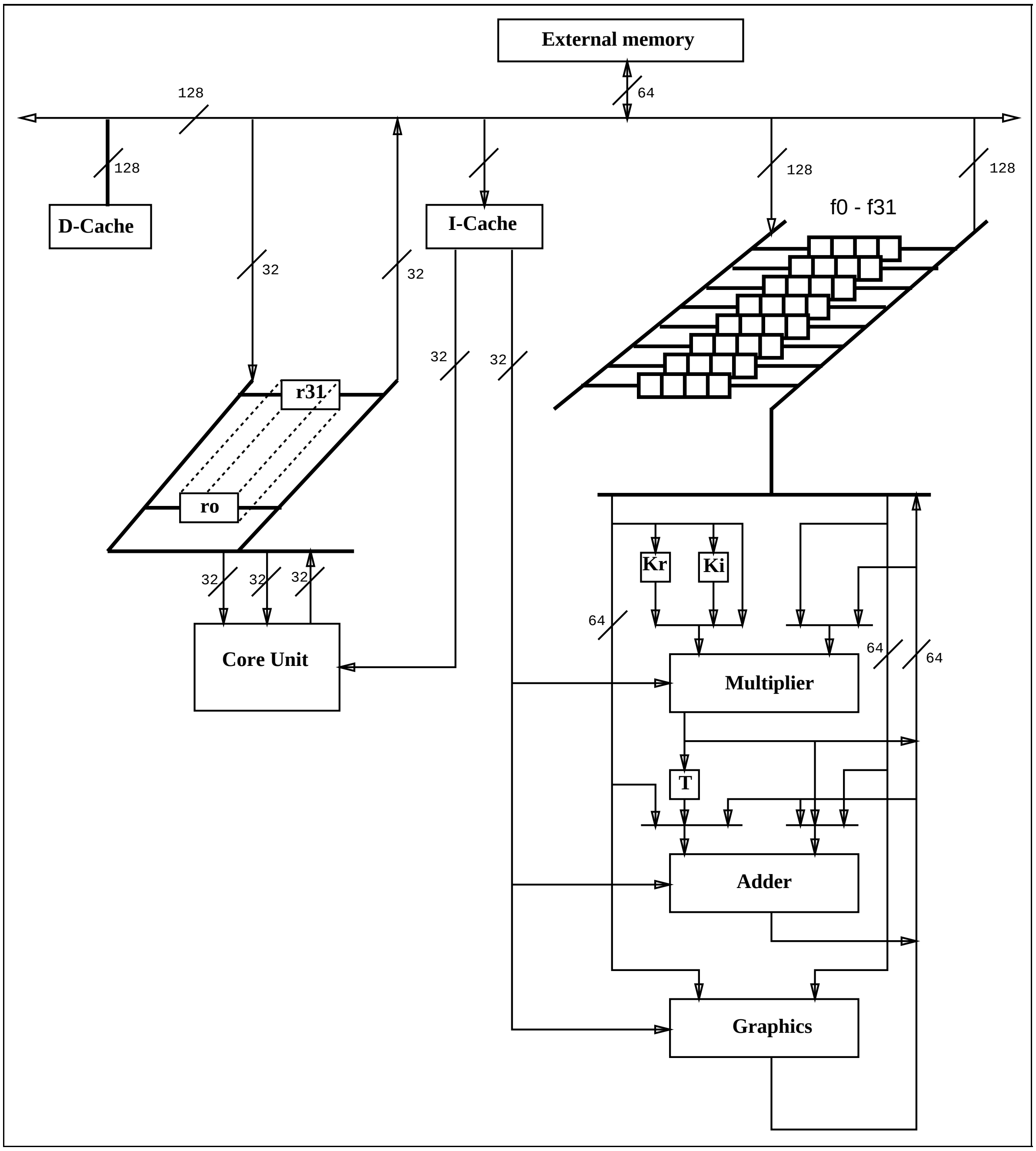


Figure 4.2: The i860 Internal datapaths

by the appropriate vector instruction. Strictly speaking the i860 is a *pipelined scalar* processor rather than a vector processor. It implements scalar operations using a pipeline rather than having single instructions that operate on vectors. When a vectorising compiler is ported to the i860 these standard operations must be implemented using hand-coded assembly language. The compiler then inserts these optimised routines into appropriate places in a program. For example the the code fragment

```
        DO I = 1,N
          X(I) = (A(I)*B(I))-(C(I)*D(I))+(E(I)*F(I))
        ENDO
```

may be converted into

```
c         vector multiply
        VVM(X,A,B,N)
c         vector multiply
        VVM(TMP,C,D,N)
c         vector subtract
        VVS(X,X,TMP,N)
c         vector multiply
        VVM(TMP,E,F,N)
c         vector add
        VVA(X,X,TMP)
```

Such a compiler can never achieve a greater efficiency than that achieved by the individual vector routines.

This kind of vectorising compiler will generate much more efficient code than the currently available generic compilers. The i860 floating-point units are in some ways more flexible than conventional vector architectures, being able to mix different operations in the pipeline. Vector mainframes are also usually connected to a much more sophisticated memory system than is usually used with a microprocessor, so the i860 will usually be limited by memory access speeds. This gives rise to a number of drawbacks to using the traditional vector processing model for the i860. When a program is decomposed into a large number of simple vector operations many of these operations will only use one of the two floating point units so the dual operation instructions will not be fully utilised. The program will also loop through the data a large number of times. This increases the number of memory accesses because intermediate results will be written back into memory and then retrieved in a later loop. If the vector length is long enough for the vectors to be larger than the size of the cache this will have a significant impact on the performance.

In order to get the maximum performance out of an i860 it is preferable to use a smaller number of loops that perform higher-level operations on the elements of the vectors. A high-level operation containing several multiplications and additions is preferable because it has a good chance of using both floating-point units simultaneously. This will not prevent the pipelined floating-point instructions from being used because the type of instruction being started at the head of a pipeline has no effect on the instructions that are already being processed. Different floating-point instructions can therefore be mixed freely within the inner loop of a vectorised procedure. There will also be a much lower demand on the memory system. Any temporary variables needed by the high-level operation can be implemented using registers or a few scratch variables that always remain in the cache. If the same operation was implemented as a number of less complex loops each temporary variable would have to be an entire vector in order to store one value from each of the iterations, for example the `TMP` vector introduced previously. If the vector length is large this can easily overflow the cache.

There is no reason why efficient compilers for the i860 cannot be developed. One possible solution is for a vectorising compiler to process long loops using a number of small sections, where each section is small enough for the vectors to remain in the cache between the low-level vector operations. At the time of writing the human assembly language programmer still has a significant advantage over the available compilers for the i860. A hand-coded routine takes longer to produce than a compiled one and cannot be ported to a different type of processor, but for programs that are expected to run for several months or years, such as lattice QCD simulations, this extra time is small compared to the time saved. The disadvantage of assembly language programming is that great care must be taken to ensure the routine does what is expected of it. Each routine must be thoughly debugged before it is installed in a full program. This is especially important when the program is intended to run for long periods of time. Only a limited number of key routines need be hand coded in assembly language. In lattice gauge simulations these will always be vector routines with a very long vector length. In the following text this kind of optimisation is discussed. In this discussion I will use "iteration" to refer to the operation applied to a single position of the input vectors, and "loop pass" to refer to the instructions that make up the body of the inner loop of the vectorised routine. More than one iteration may be contained in a loop pass.

4.3.1 Assembly language programming on the i860

The i860 has a very powerful assembly language [52]. The main difficulties when writing assembly code are keeping track of the various pipelines and identifying independent calculations that can be performed in parallel by the different units of the processor. The second problem is usually solved in one of two ways. The first way is to write the inner loop of a vectorised routine in such a way as to perform more than one iteration at a time. Initially the code for a single iteration is written assuming a single stage pipeline; this is relatively simple compared to code for a longer pipeline, see figure 4.3. This is then converted into code that calculates

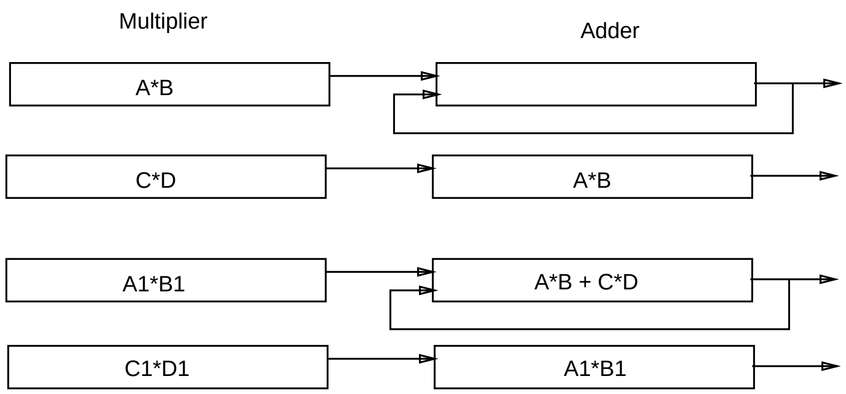


Figure 4.3: Code for single-stage pipelines.
The diagram shows how the function $(A * B) + (C * D)$ would be implemented using single-stage pipelines.

three iterations simultaneously by replicating each instruction three times. Each one of the three instructions belongs to a separate iteration. Because the number of instructions has tripled, the code now corresponds to a pipeline of length three, see figure 4.4. Because each iteration requires separate data, this approach may

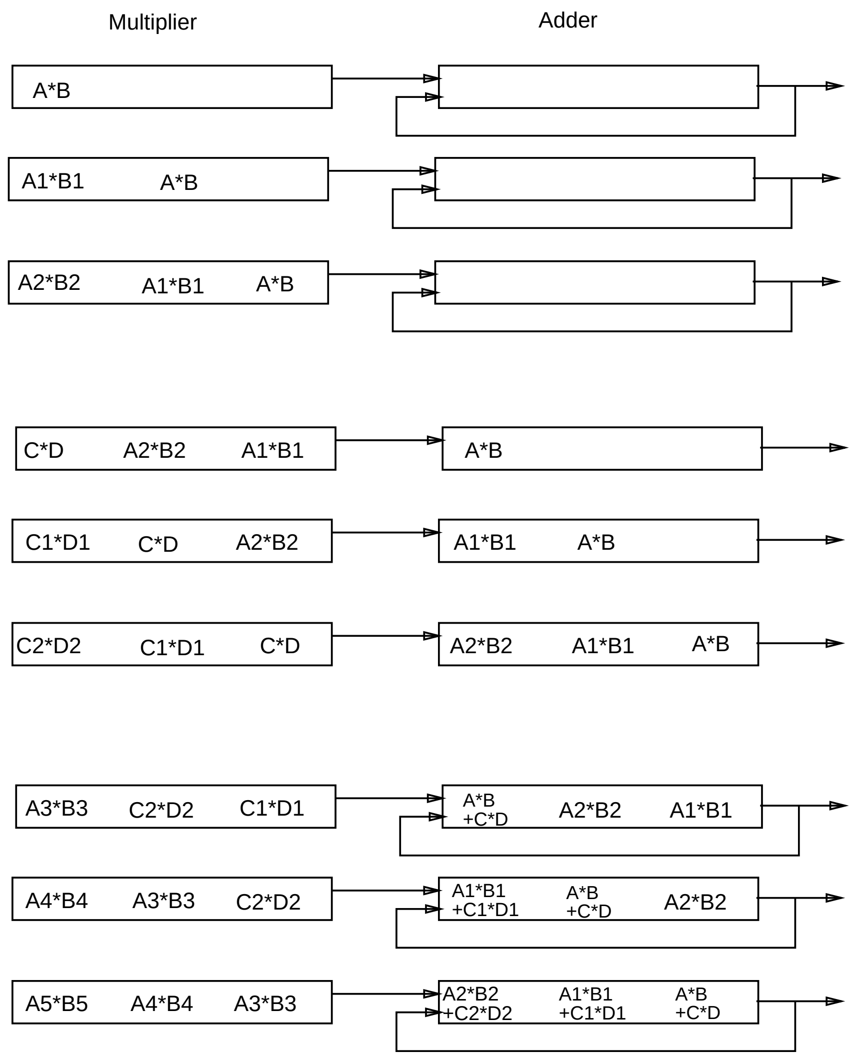


Figure 4.4: Code for three-stage pipelines.
The diagram shows how the function $(A * B) + (C * D)$ would be implemented using three-stage pipelines. Each set of three instructions corresponds to one of the instructions from the single-stage solution.

require three times as many registers as an unpipelined implementation. If the vector length cannot be guaranteed to be a multiple of three, a complicated section of code will also be needed at the end of the main loop in order to finish off any remaining iterations. This flushing phase can often be more complex than the main loop. The second possible approach is to identify some inherent parallelism in the iteration itself. The $SU(3)$ gauge group of QCD is especially convenient in this respect, because in matrix arithmetic the elements of the result are calculated independently. It is therefore always possible to split an iteration into a multiple of three independent parts, one for each row of the output. As these independent calculations share operands, the number of registers required is not increased. For less convenient routines it is still necessary to have several iterations performed in a loop-pass. The large requirement for registers can be reduced at the expense of clarity of code by overlapping sections from the same iteration where possible. For example, an $SU(2)$ routine has to perform three iterations for each loop pass. There are four independent sections to each iteration, corresponding to the four real parameters of an $SU(2)$ matrix. The most obvious way to construct the loop is to overlap the same calculation for three separate matrices; see Figure 4.5. This will require at least enough registers to store three sets of $SU(2)$ matrices simultaneously. In this scheme three new sets of $SU(2)$ matrices are required at a single point in the loop. This would introduce a severe performance overhead unless even more registers are used to allow the data for the next loop-pass to be read in without overwriting the current data. The better way is to overlap as much as possible from the same iteration; see figure 4.6. Because at most two iterations are proceeding at any one time this only requires enough registers for two sets of $SU(2)$ matrices. The register reloads are also spread more evenly over the loop-pass. Multiple iterations per loop-pass are still needed in order to make the loop-pass finish in the same state that it started in.

An example of this from the QCD code is the *psi2chi* procedure that was written to be part of a $r = 1$ Wilson fermion simulation. This procedure performs a set of projections on a vector of four spinors in order to produce eight sets of two spinors. The registers are divided into three blocks. At any one time, one of the blocks contains the four spinor being used by the present iteration. The next block is being loaded with the four spinor needed by the next iteration. The remaining block is used to save the results. Each block of registers is used for each of the different functions in turn. It therefore takes three iterations before the registers return to their original functions. The loop pass must therefore consist of a multiple of three iterations. It would have been possible to just swap the function of the first two blocks and reduce the loop pass to a pair of iterations but

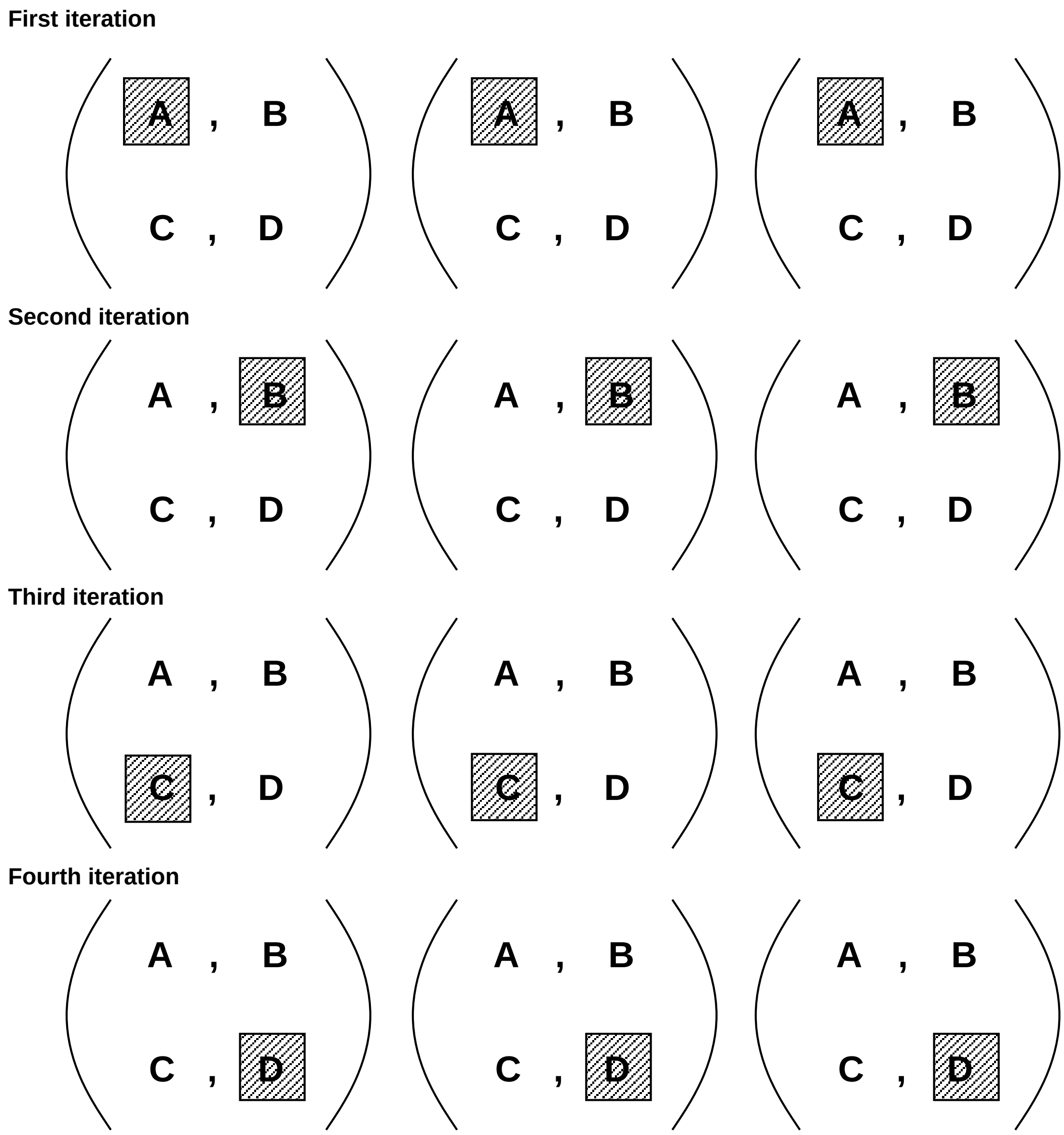


Figure 4.5: A simple implementation of $SU(2)$ multiplication

This figure shows a simple implementation of a $SU(2)$ multiplication. The shaded regions shows which of the four independent sections are active at any time. If the $SU(2)$ multiplies are implemented in this fashion all three sets of matrices will be required throughout the loop.

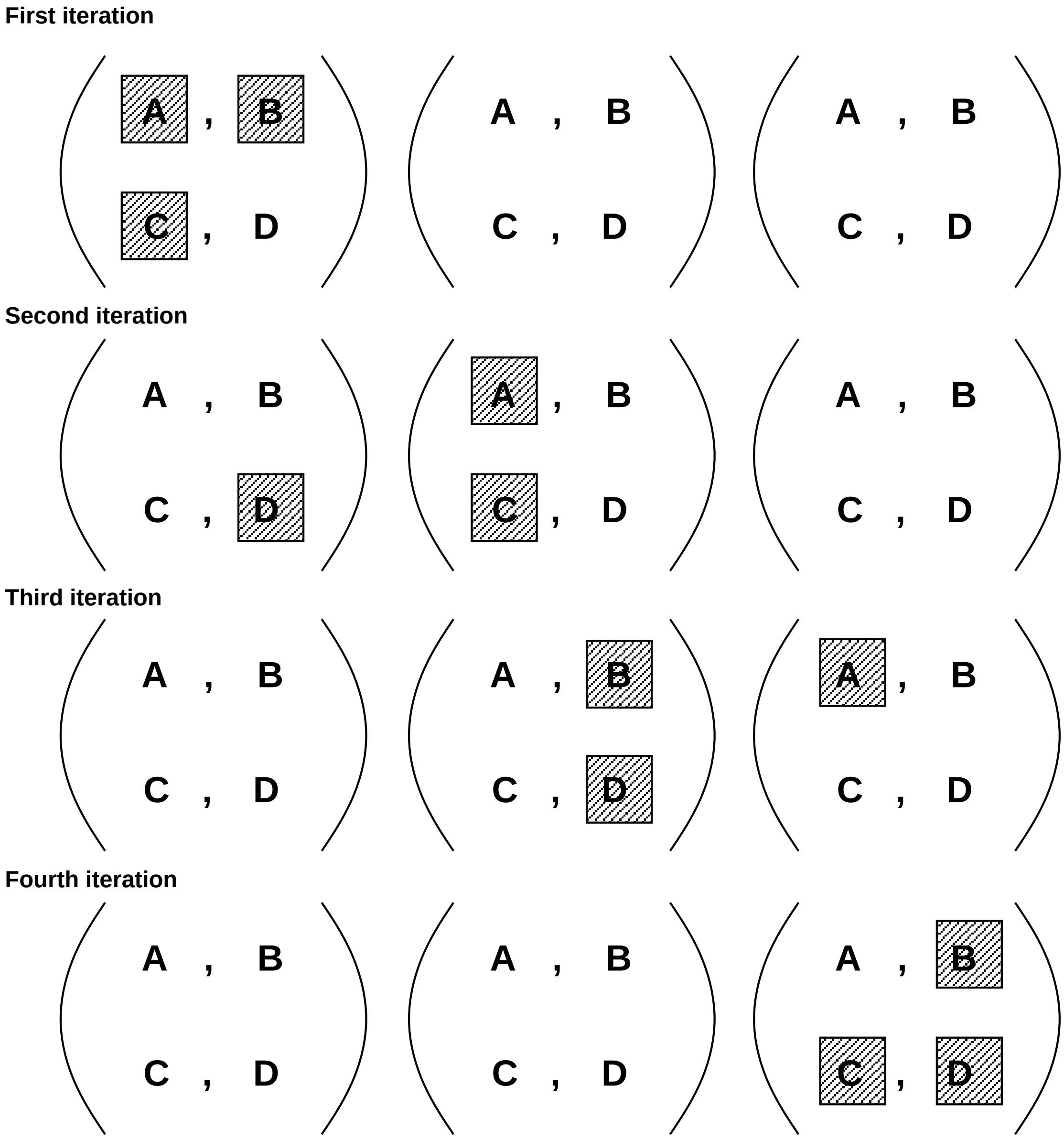


Figure 4.6: An improved implementation of $SU(2)$ multiplication

This figure shows an improved implementation of a $SU(2)$ multiplication. The shaded regions shows which of the four independent sections are active at any time. This implementation of the $SU(2)$ multiplication only requires enough registers for two sets of matrices. It is also much easier to keep the register load instructions spread out.

the vector length is guaranteed to be a multiple of three, because of the colour index, so a three iteration loop removes the need for a final flushing phase.

The second problem, keeping track of the pipelines, is nothing more than a complex book-keeping operation. When there is a natural three-way decomposition of the problem this is very easy to do. All of the instructions come in multiples of three and so a set of three results will emerge from the pipeline during the next set of three instructions. In more complex procedures such as the *psi2chi* procedure the book-keeping becomes very difficult. Luckily this is exactly the kind of book-keeping that computers are very good at doing. The *psi2chi* procedure was not written directly in assembly language. Instead a special program was written to write the inner loop of the code. Each one of the iterations was written by a single procedure that took a set of three parameters to tell it which register block was to be used for which purpose. This procedure was made up of other procedure calls each corresponding to a recognisable section of the iteration. At the lowest level was the procedure responsible for writing a single assembly language instruction. This procedure is passed the name of the register that is to receive the result of the current calculation. Because of the pipelining this result will not become available for a further three instructions so this procedure keeps a record of these destination registers and substitutes the correct name three instructions later. This approach is especially valuable because large sections of some procedures can occur several times with slight modifications. If a bug is found in one of these sections it usually exists in all of them and a large amount of editing is needed to correct the problem. When a simple code-writing program is used, only a single part of this program need be changed to correct the error. Even though this section of code is generated automatically it is still valuable to make it as readable as possible. The code writing program therefore also inserts appropriate comments into its output.

There is no easy way to verify that any code is correct. This is more of a problem with assembly language because there is much more detail in the code to hide mistakes. Assembly-language routines are much easier to verify if there is some standard that the code can be compared against. It is therefore essential to maintain a library of high-level language equivalents for all of the assembly-language routines. This high-level language implementation is essential in order to debug the test programs that are going to be used to verify the assembly-language routines. They can also serve as additional documentation and provide an easy way of porting the program to a different machine.

4.3.2 The inner loop.

All of the optimised routines considered here repeat their operations a large number of times, corresponding to the elements of the input vector. It is therefore necessary to pay particular attention to the implementation of loops on the i860 processor.

The i860 has a special instruction called the *bla* instruction that can be used to implement loops. It can only be used for the innermost loop of a set of nested loops. The outer loops must be written using conditional branching. In the case of the optimised routines considered here, short loops, such as the loop over colours, are usually unwound as part of the pipelining. This usually means that there are no nested loops and the *bla* instruction is used for the loop over sites. All of the operations needed to implement a loop are contained in this single instruction. This instruction is the most efficient way to implement a loop but it is not the simplest instruction to understand. The pseudo-code definition of the *bla* instruction is:

bla step, count, label

LCC_temp clear if count < comp2(step) (signed)
LCC_temp set if count ≥ comp2(step) (signed)
count ← step + count
Execute one more sequential instruction.
IF LCC
THEN LCC ← LCC_temp,
goto label
ELSE LCC ← LCC_temp
FI

LCC is a status flag that is only used by the *bla* instruction, its value is preserved between calls to this instruction. The function *comp2* denotes the "twos-compliment" of a number, this is equivalent to multiplying the number by -1. The source of a simple memory copy routine are shown here as an example of the use of the *bla* instruction.

```
//
//  scopy(a,b,len) - copies len words from a to b.
//
.string "$Id: scopy.s,v 1.3 1991/04/16 14:08:06 spb Exp $"
.align 4
      a = r16
      b = r17
      len = r18
      step = r19
      ftmp = f16

_scopy::                          // entry point
      adds -1, r0, step           // step = -1
      adds -1, len, len           // len = len-1
      addu -4, a, a               // allow for auto-increment, a=a-4
      bla  step, len, LOOP        // initialise bla state
      addu -4, b, b               // allow for auto-increment, b=b-4

LOOP:
      fld.l    4(b)++,  ftmp      // b = b+4 and load word.
      bla      step, len, LOOP
      fst.l    ftmp,  4(a)++      // a = a+4 and store word.

      bri        r1               // Return to calling procedure.
      nop                         // nop because of delayed branch.
```

The important points about this instruction are firstly that it uses a delayed branch; the jump does not take place until the instruction following the *bla* instruction has been executed. The *bla* instruction is therefore the next to last instruction in the loop. Secondly, the branch being taken or not is controlled by previously executed *bla* instructions. When a procedure first uses the *bla* instruction it is not known if the branch will be taken or not. Before using the *bla* instruction in a loop each procedure must use the instruction once to set the LCC flag to a known state. The jump address for this call must point the second instruction following itself. This will make sure that the initial value of the LCC flag makes no difference to the behaviour of the procedure.

The floating-point load instructions have an auto-incrementing mode. This is to enable loops to be coded efficiently. An auto-incrementing load is written as:
fld.l offset(pointer)++, destination
The variable in the memory location addressed by $pointer + offset$ is loaded into the floating point register "*destination*". The *pointer* register is then incremented by the value *offset*. This enables a large vector of values to be stepped through one at a time. Care must be taken to prevent the auto-increment instructions from skipping the first element of the vector.

When writing vectorised procedures it is often convenient to have a dummy loop pass immediately before the body of the main loop. The primary reason for this is because the results have to be written back into memory during the next iteration. The first iteration performed by a routine has no such values to store so it must be performed by a separate piece of code that is missing these instructions. If we have to have a dummy loop-pass, then a number of necessary initialisation steps can be absorbed into this dummy loop in order to reduce the set up time for the procedure. The *bla* instruction can be initialised by placing a *bla* instruction in the dummy loop. The first load instruction that uses each of the pointer registers should not be auto-incrementing: this can also be done inside the dummy loop. A similar problem exists for the final iteration. Some of the data needed by a loop iteration should be loaded while the previous iteration is being executed. This enables the iteration to start straightaway without waiting to load its data. Unless the final iteration is also performed by a dummy loop this means that extra values will be read from beyond the end of the input vectors. Because this data is effectively discarded when the procedure finishes this can usually be ignored. There are two potential problems that have to be avoided. The i860 supports paged memory management. This means that not all memory addresses necessarily correspond to a valid piece of memory. If an invalid memory address is read, an error is flagged. It is possible that the extra loads in the final loop pass may cause an illegal memory access. In a parallel computer it is sometimes possible to have communication happening simultaneously with numerical calculations. These communications will only be able to progress normally if the i860 does not attempt to access any of the memory locations being used in the communication. It is therefore also necessary to ensure that the extra loads do not access any communication buffers. The easiest way to get round both of these problems is to add some padding space to the end of all the data arrays. The extra loads will therefore always perform a safe read from this padding space.

4.3.3 Pipelined loads and managing the cache.

The main limitation to performance on an i860 is the memory system. Both floating-point units are capable of producing a result every clock cycle unless certain freeze conditions occur. Most of these conditions can be avoided by careful design of the code. The most important of these freeze conditions are cache misses, where a non-pipelined load attempts to read a value not stored in the cache, and an overfull load pipeline, where more than three pipelined loads are outstanding. For most lattice gauge simulations the data set can be expected to be much larger than the cache. Non-pipelined loads can therefore be expected to produce a large number of cache misses. When a floating-point load accesses outside of the cache, the processor will start to load the new cache-line containing the missing data into the cache. A cache line consists of 32 consecutive bytes of memory. The execution of the program will be delayed for one clock cycle plus the time to load the first value of the cache line. Providing that a reasonable number of clock cycles pass before the next value is needed, the rest of the cache line will have been loaded and further accesses can proceed without any delay. The alternative is to use pipelined loads. This pipeline can be processing up to three memory fetches at a time. If a fourth value is requested while there are still three outstanding accesses then there will be a freeze for one cycle plus the time to read the first outstanding value. There will also be a freeze if the value requested is actually in the cache. This will take two cycles plus the time to finish all of the outstanding accesses. An added complication is that many external memory systems use paged DRAM. In these systems a sequence of memory accesses from the same 1 Kbyte page of memory is more efficient than a sequence that hops from one region of memory to another. With non-pipelined loads we are guaranteed that reads will occur in blocks of 32 bytes and come from the same memory page. It is therefore very difficult to determine what mixture of cache and pipelined loads will be most efficient for a particular routine. The only safe way of doing this is to time various versions of procedures inside a real program. For example, a complex scalar product is about 10% faster if pipelined loads are used. However, if the source vector is already stored in the cache pipelined loads are almost certainly going to be slower.

4.4 QCD simulation programs.

The Wilson fermion action is:

$$S_{Wilson} = \sum_n \{\bar{\psi}_n \psi_n - K \sum_{\mu=1}^{4} [\bar{\psi}_n (rI - \gamma_\mu) U_{n,\mu} \psi_{n+\hat{\mu}} + \bar{\psi}_n (rI + \gamma_\mu) U^\dagger_{n-\hat{\mu},\mu} \psi_{n-\hat{\mu}}]\},$$

where K is the hopping parameter defined by:

$$K = \frac{1}{8r + 2ma}.$$

The fermion matrix can therefore be written as

$$A(m,n) = \delta_{m,n} - K \sum_{\mu=1}^{4} (rI - \gamma_\mu) U_{m,\mu} \delta_{n,m+\hat{\mu}} + (rI + \gamma_\mu) U^\dagger_{m-\hat{\mu},\mu} \delta_{n,m-\hat{\mu}}.$$

Thus inverting the Dirac equation is replaced by solving

$$A\phi = \mathcal{R},$$

where $\mathcal{R}$ is the source and K is the hopping parameter. This is equivalent to inverting the fermion matrix.

Any simulation of dynamical fermions will be dominated by the inversion of the fermion matrix. The fermion matrix is a function of the gauge fields and is therefore constantly changing throughout the simulation. This means that the matrix inversion will also have to be repeated throughout the simulation. Even if dynamical fermions are not used in a simulation it is still necessary to invert the fermion matrix in order to calculate propagators from the gauge configurations. It is therefore fairly obvious that the fermion matrix inversion procedures need optimising. Iterative algorithms such as conjugate gradient are used to invert the matrix. The majority of time in these algorithms is spent multiplying vectors by the matrix. So the greatest improvement can be obtained by optimising these routines.

4.4.1 Conjugate Gradient

Conjugate gradient is an iterative matrix inversion algorithm, commonly used in lattice gauge theory. It is a very robust algorithm and in exact arithmetic it is guaranteed to converge to a solution in n iterations where n is the dimensionality of the matrix. Unfortunately the algorithm is only applicable to positive definite hermitian matrices. If the Hybrid Monte-Carlo (HMC) algorithm is used to simulate a theory with dynamical fermions the fermion matrix equations that must be solved are all of the form

$$(M^\dagger M)\psi = \mathcal{R}.$$

The matrix to be inverted is always hermitian positive definite, so conjugate gradient is ideal for use with HMC. Propagator calculation need to solve equations of the form

$$M\psi = \mathcal{R}.$$

Conjugate gradient can still be used for this application by solving the equations

$$MM^\dagger \phi = \mathcal{R}$$

$$\psi = M^\dagger \phi.$$

In this case there are alternative algorithms such as *minimal residual* [53] that may be more appropriate. Minimal residual is sufficiently similar to conjugate gradient for the same optimisations to be applicable to both algorithms.

When solving the equation

$$A\psi = \mathcal{R}$$

starting from an initial guess of ψ_0, the generic form of the conjugate gradient matrix algorithm is:

$$p_0 = r_0 = \mathcal{R} - A\psi_0,$$

then repeat;

$$\begin{aligned}
\alpha_n &= \frac{|r_n|^2}{(p_n, Ap_n)} \\
\psi_{n+1} &= \psi_n + \alpha_n p_n \\
r_{n+1} &= r_n - \alpha_n A p_n \\
\beta_n &= \frac{|r_{n+1}|^2}{|r_n|^2} \\
p_{n+1} &= r_{n+1} + \beta_n p_n.
\end{aligned}$$

In the case that we are interested in, where $A = M^\dagger M$ this can be reformulated to be more efficient at the expense of creating an extra workspace $s_n \equiv Mp_n$. The conjugate gradient algorithm now becomes:

$$p_0 = r_0 = \mathcal{R} - M^\dagger M\psi_0,$$

then repeat;

$$\begin{aligned}
s_n &= Mp_n \\
\alpha_n &= \frac{|r_n|^2}{|s_n|^2} \\
\psi_{n+1} &= \psi_n + \alpha_n p_n \\
r_{n+1} &= r_n - \alpha_n M^\dagger s_n \\
\beta_n &= \frac{|r_{n+1}|^2}{|r_n|^2} \\
p_{n+1} &= r_{n+1} + \beta_n p_n
\end{aligned}$$

The conjugate gradient algorithm can therefore be divided into a number of component operations that can be optimised independently.

- Scalar products

$$a = |X|^2$$

- Saxby operations

$$R_n = X_n + a * Y_n$$

- Action of the fermion matrix on a vector

$$x = My$$

The multiplication by the fermion matrix is dominated by the $\not{D}$ operation that can in turn be divided into three sub-steps.

- Construction of two-component spinors.
- $SU(3)$ multiplication.
- Reconstruction of four-spinors.

The performance of each of these operations has a very significant effect on the time taken to invert the fermion matrix and hence on the total performance of the program. They are therefore the obvious candidates to be hand coded in assembly language.

The Dslash operation

Construction of two-component spinors For $r = 1$ Wilson fermions the action of the γ matrices on the spinors becomes a set of projection operators $(I \pm \gamma_\mu)$. This means that the $SU(3)$ multiplications need only be applied to two of the components of the resulting 4-component spinors; the missing two components of the result can always be constructed from the two known components. In the CG algorithm, $\not{D}$ is applied to s and p vectors. Here these vectors are generically denoted by ψ. These are 4-component spinors $(\psi_1, \psi_2, \psi_3, \psi_4)$. The 2-component spinors obtained from $(1 \pm \gamma_\mu)\psi$ are denoted by $\chi_\mu^\pm$ and are defined as follows.

$$\chi_{1,1}^+ = \psi_1 + i\psi_4 \qquad \chi_{1,1}^- = i\psi_2 + \psi_3$$

$$\chi^+_{1,2} = \psi_2 + i\psi_3 \qquad \chi^-_{1,2} = i\psi_1 + \psi_4$$

$$\begin{aligned} \chi^+_{2,1} &= \psi_1 + \psi_4 \qquad & \chi^-_{2,1} &= \psi_2 + \psi_3 \\ \chi^+_{2,2} &= \psi_2 - \psi_3 \qquad & \chi^-_{2,2} &= -\psi_1 + \psi_4 \end{aligned}$$

$$\begin{aligned} \chi^+_{3,1} &= \psi_1 + i\psi_3 \qquad & \chi^-_{3,1} &= i\psi_1 + \psi_3 \\ \chi^+_{3,2} &= \psi_2 - i\psi_4 \qquad & \chi^-_{3,2} &= -i\psi_2 + \psi_4 \end{aligned}$$

$$\begin{aligned} \chi^+_{4,1} &= 2\psi_1 \qquad & \chi^-_{4,1} &= 2\psi_3 \\ \chi^+_{4,2} &= 2\psi_2 \qquad & \chi^-_{4,2} &= 2\psi_4 \end{aligned}$$

The implementation of this procedure *psi2chi* has been discussed previously. It is not possible to use dual operation instructions in this procedure. The maximum possible performance will therefore only be half of the peak performance of the processor because only one of the floating-point units, the adder, is being used. This puts an upper bound of 40 Mflops on the performance of this code. The performance will be further reduced because of the the large number of memory accesses needed by each iteration.

SU(3) multiplication The fermion matrix is

$$A(m,n) = \delta_{m,n} - K \sum_{\mu=1}^{4} (rI - \gamma_\mu) U_{m,\mu} \delta_{n,m+\hat{\mu}} + (rI + \gamma_\mu) U^\dagger_{m-\hat{\mu},\mu} \delta_{n,m-\hat{\mu}}$$

Evaluating this function requires two sets of $SU(3)$ multiplications and a number of data shifts. There are four steps involved in computing the two terms:

- The $U^\dagger$ multiplication at site $x - \hat{\mu}$ with local operands,
- Shift result of first stage to site x;
- Shift $\psi(x + \hat{\mu})$ to site x;
- The U multiplication at site x

In terms of the χ fields this procedure becomes:

$$\begin{aligned}
W_\mu(x) &= U_\mu^\dagger(x)\chi_\mu^+(x) \\
\chi_\mu^{'+}(x) &= W_\mu(x-\hat{\mu}) \\
X_\mu(x) &= \chi_\mu^-(x+\hat{\mu}) \\
\chi^{'-} &= U_\mu(x)X_\mu(x)
\end{aligned}$$

The most straightforward way of doing this is to perform the $U^\dagger$ multiplication while transferring the χ needed for the U multiplication, and then perform the U multiplication while transferring the result of the $U^\dagger$ multiplication. This is the most time-critical portion of the entire program. Two very similar routines are needed, one to perform the U multiplication and one to perform the $U^\dagger$ multiplication. Because these two routines are so similar, the following comments apply equally well to both of them. Matrix operations are well balanced between addition and multiplication operations, so there will be no difficulty in using both of the floating point units effectively. The i860 has 30 general purpose registers. This enables us to hold an entire $SU(3)$ matrix in registers and still leave room for six more complex variables to form the source and result of the matrix multiplication. Each $SU(3)$ matrix will be used twice before being discarded, once for each of the two spin components.

These routines will always come paired with either a gather or scatter operation that performs a shift in the $\pm\mu$ direction. Normally any such operation will be highly inefficient because there is a large number of memory accesses and the floating-point units will remain idle throughout this process. One obvious optimisation is to combine gather scatter operations with the $U/U^\dagger$ products. This not only reduces the total number of memory accesses but also allows the core unit to be doing useful work in parallel with the floating-point units. The data shifts we wish to implement have a very regular pattern; variables that are stored in adjacent memory locations are very likely to still be adjacent after the data shift. This enables us to move large sections of the lattice together as a block. The number of memory accesses needed by the gather/scatter can be drastically reduced if the table used to control the operation specifies blocks of data to be moved rather than individual variables. In our i860 assembly-language routines, the gather/scatter tables are implemented as pairs of values. The first number represents the start of a contiguous block of variables encoded as a byte offset from the start of the array. The second number encodes the end of the block in a similar fashion. At the start of a block of data the procedure converts these numbers into absolute memory addresses and sets a data pointer to the beginning of the block. The data pointer is incremented by a constant value each loop pass until its value equals the finish address, at which point a new pair of numbers is read from the gather/scatter table and the process is repeated. This requires a conditional branch in the main body of the loop, one half of which increments the input pointer and the other half starts a new data block. Only the core unit plays any part in the gather/scatter process. As these routines operate in dual-instruction we can hide the branch from the floating-point units by putting identical floating-point instructions in each half of the branch. These routines are part of a large family of $SU(3)$ multiplication routines. These routines fall into three main classes. Within each class there are several options that give rise to separate routines

Two-spinor These reuse the $SU(3)$ matrix twice, once for each spin component. Possible options are: gather/scatter the source/result, multiply by the hermitian conjugate of the matrix.

Matrix-matrix In these routines the left-matrix is reused three times, once for each column of the right-matrix. Possible options are: gather/scatter the right-matrix/result-matrix, use hermitian conjugate of either/both source matrices.

Four-spinor These reuse the $SU(3)$ matrix four times, once for each spin component. Possible options are: gather/scatter the source/result, multiply by the hermitian conjugate of the matrix.

The changes needed to implement the various options are reasonably simple. Each class is implemented as a single source file with compile time definitions to select the different options.

Reconstruction of 4-component spinors

Following the operation of $SU(3)$ multiplication routines we have 8 2-component spinors, $\chi_\mu^{'\pm}$. These must be used to reconstruct the 4-component spinors ψ. This is done by the procedure *reconstruct*. Generically, writing ψ' for the result of the $\not{D}$, this operation is:

$$\begin{aligned}
\psi_1' &= \psi_1 - \kappa\{\chi_{1,1}^{'+} - i\chi_{1,2}^{'-} + \chi_{2,1}^{'+} - \chi_{2,2}^{'-} + \chi_{3,1}^{'+} - i\chi_{3,1}^{'-} + \chi_{4,1}^{'+}\} \\
\psi_2' &= \psi_2 - \kappa\{\chi_{1,2}^{'+} - i\chi_{1,1}^{'-} + \chi_{2,2}^{'+} + \chi_{2,1}^{'-} + \chi_{3,2}^{'+} + i\chi_{3,2}^{'-} + \chi_{4,2}^{'+}\} \\
\psi_3' &= \psi_3 - \kappa\{\chi_{1,1}^{'-} - i\chi_{1,2}^{'+} + \chi_{2,1}^{'-} - \chi_{2,2}^{'+} + \chi_{3,1}^{'-} - i\chi_{3,1}^{'+} + \chi_{4,1}^{'-}\} \\
\psi_4' &= \psi_4 - \kappa\{\chi_{1,2}^{'-} - i\chi_{1,1}^{'+} + \chi_{2,2}^{'-} + \chi_{2,1}^{'+} + \chi_{3,2}^{'-} + i\chi_{3,2}^{'+} + \chi_{4,2}^{'-}\}
\end{aligned}$$

There are not enough registers to store all the components of the χ spinors simultaneously. It is therefore necessary to use non-pipelined loads and rely on the cache. Like the *psi2chi* function, this operation is dominated by addition operations and needs a large number of memory accesses.

4.4.2 The scalar product

The scalar product is the simplest of the CG procedures to implement because it only consists of a sum of products. This is easily achieved by chaining the adder and multiplier units together. A generic scalar product has the form:

$$s = \sum_{i=0}^{n} X_i Y_i$$

For conjugate gradient we only need the norm of a single vector rather than the scalar product of a pair of vectors. It is worth implementing the vector norm procedure explicitly because it requires only half the memory access of a general scalar product. For maximum performance the pipelines require three summations to be carried out simultaneously. When all three summations have finished, their results are added together to give the final scalar product. Because of the colour index the vector length can be guaranteed to be a multiple of three, so the scalar product routine will not need a final flushing phase.

4.4.3 The saxpy operation

The saxpy operation

$$R_n = X_n + a * Y_n$$

is also relatively straightforward to implement on the i860. The adder and multiplier pipelines can be chained together and the constant a can be stored in one of the constant registers. The timings for this procedure are totally dominated by the time taken to load and store the data. For every pair of floating-point operations two operands must be loaded and a single result stored.

4.5 Benchmarking the i860.

All of the routines mentioned in the previous section have been implemented in i860 assembly language and the performance measured. All of the timings in this section are for a single i860. In each case the vector-length was chosen to be representative of the real production code. In normal production the local lattice size is $12^3 \times 6$; because we split the data into even and odd parity sites, this corresponds to a vector length of $12^3 \times 3 = 5184$. In each case the input arrays were initialised randomly and the time taken to execute the routine 500 times was measured. In the $SU(3)$ routines with built-in gather/scatter the gather-scatter table was constructed to access the input vector as 8 separate blocks. Timings are presented for one of the 16Mbyte MK086 boards (see table 4.1), and for a prototype MK096 board with 4Mbytes of memory (see table 4.2). The MK096 board is similar to the MK086 but is a much later design of board with an improved memory system. A set of timings for equivalent routines written in the C language are shown in table 4.3. These routines were compiled using the "Greenhills" compiler. The C versions of the code were written as part of the validation procedure for the assembly language routines and no attempt has been made to optimise them, other than the optimisations provided by the compiler.

The values presented in the tables are as follows:

routine The name of the routine being benchmarked.

length The vector length used in the benchmark.

oper The number of floating-point operations per iteration.

mem The number of bytes of memory accessed in each iteration.

repeat The number of times the routine is called during the benchmark.

time The number of microseconds taken to execute the benchmark.

Mflops The average floating-point performance in units of 10^6 floating-point operations per second.

Mbyte/sec The average memory bandwidth achieved during the benchmark.

The routines presented presented in the tables are as follows:

psi2chi Apply the gamma-matrix projection operators to a four-spinor to generate 8 sets of two-spinors. This forms the initial stage of the $\not{D}$ operation.

reconstruct Reconstruct a four-spinor from 8 sets of two-spinors. This forms the final stage of the $\not{D}$ operation.

saxpy Scale a vector by a constant and add the result to a second vector. The vector length has been chosen to match that for vectors of four-spinors with a colour index.

sdot Calculate the norm of a complex vector. The vector length is the number of complex numbers in the vector.

su3_xx Matrix-matrix multiply.

su3_xx_g Matrix-matrix multiply. Gathering source.

su3_xx_s Matrix-matrix multiply. Scattering result.

su3_mv4_xm_n Matrix-four-spinor multiply.

su3_mv4_xm_g Matrix-four-spinor multiply. Gathering source.

su3_mv4_xm_s Matrix-four-spinor multiply. Scattering result.

su3x Matrix-two-spinor multiply.

su3xg Matrix-two-spinor multiply. Gathering source.

su3xs Matrix-two-spinor multiply. Scattering result.

where the letter x stands for either h or m denoting if the hermitian conjugate of the $SU(3)$ matrix is used or not.

As can be seen from the timings for the different routines, the performance of an individual routine is very strongly dependent on the way it accesses the external memory system. There is a consistent trend within the $SU(3)$ routines; four-spinor routines are faster than the matrix-matrix routines that are in turn faster than the two-spinor routines. This corresponds to the different ratios of floating-point operations to memory accesses. The improved memory system of the MK096 board gives rise to a dramatic increase in performance. The *sdot* procedure has a very high performance compared to the other routines. This can be explained by noting that it has a much simpler memory access pattern than the other procedures; it only accesses a single vector.

routine	length	oper	mem	repeat	time μs	Mflops	Mbyte/sec
psi2chi	5184	96	480	500	25784128	9.65	48.25
reconstruct	5184	144	480	500	27395584	13.62	45.41
saxpy	124416	2	12	500	12879936	9.66	57.96
sdot	62208	4	8	500	3214464	38.71	77.41
su3_hh	5184	198	216	500	13887488	36.96	40.31
su3_hh_g	5184	198	216	500	13895360	36.93	40.29
su3_hh_s	5184	198	216	500	13894848	36.94	40.29
su3_hm	5184	198	216	500	13895232	36.93	40.29
su3_hm_g	5184	198	216	500	13900864	36.92	40.28
su3_hm_s	5184	198	216	500	13901632	36.92	40.27
su3_mh	5184	198	216	500	13720320	37.41	40.81
su3_mh_g	5184	198	216	500	13729472	37.38	40.78
su3_mh_s	5184	198	216	500	13728192	37.38	40.78
su3_mm	5184	198	216	500	13724608	37.39	40.79
su3_mm_g	5184	198	216	500	13732544	37.37	40.77
su3_mm_s	5184	198	216	500	13731264	37.38	40.77
su3_mv4_hm_g	5184	264	264	500	17590208	38.90	38.90
su3_mv4_hm_n	5184	264	264	500	17585408	38.91	38.91
su3_mv4_hm_s	5184	264	264	500	17590208	38.90	38.90
su3_mv4_mm_g	5184	264	264	500	17398016	39.33	39.33
su3_mv4_mm_n	5184	264	264	500	17385664	39.36	39.36
su3_mv4_mm_s	5184	264	264	500	17400128	39.33	39.33
su3h	5184	132	168	500	10216128	33.49	42.62
su3hg	5184	132	168	500	10219200	33.48	42.61
su3hs	5184	132	168	500	10220480	33.48	42.61
su3s	5184	132	168	500	10054656	34.03	43.31
su3sg	5184	132	168	500	10059648	34.01	43.29
su3ss	5184	132	168	500	10061568	34.01	43.28

Table 4.1: Timings of i860 assembly language routines on an MK086.

routine	length	oper	mem	repeat	time μs	Mflops	Mbyte/sec
psi2chi	5184	96	480	500	16066752	15.49	77.44
reconstruct	5184	144	480	500	20126144	18.55	61.82
saxpy	124416	2	12	500	8721664	14.27	85.59
sdot	62208	4	8	500	1627776	76.43	152.87
su3_hh	5184	198	216	500	9113536	56.31	61.43
su3_hh_g	5184	198	216	500	9203968	55.76	60.83
su3_hh_s	5184	198	216	500	9153728	56.07	61.16
su3_hm	5184	198	216	500	9113536	56.31	61.43
su3_hm_g	5184	198	216	500	9203968	55.76	60.83
su3_hm_s	5184	198	216	500	9163776	56.00	61.10
su3_mh	5184	198	216	500	9103488	56.38	61.50
su3_mh_g	5184	198	216	500	9163776	56.00	61.10
su3_mh_s	5184	198	216	500	9133632	56.19	61.30
su3_mm	5184	198	216	500	9103488	56.38	61.50
su3_mm_g	5184	198	216	500	9163776	56.00	61.10
su3_mm_s	5184	198	216	500	9143680	56.13	61.23
su3_mv4_hm_g	5184	264	264	500	11675776	58.61	58.61
su3_mv4_hm_n	5184	264	264	500	11766208	58.16	58.16
su3_mv4_hm_s	5184	264	264	500	11675776	58.61	58.61
su3_mv4_mm_g	5184	264	264	500	11635584	58.81	58.81
su3_mv4_mm_n	5184	264	264	500	11726016	58.36	58.36
su3_mv4_mm_s	5184	264	264	500	11635584	58.81	58.81
su3h	5184	132	168	500	6752256	50.67	64.49
su3hg	5184	132	168	500	6782400	50.45	64.20
su3hs	5184	132	168	500	6782400	50.45	64.20
su3s	5184	132	168	500	6712064	50.97	64.88
su3sg	5184	132	168	500	6742208	50.75	64.59
su3ss	5184	132	168	500	6742208	50.75	64.59

Table 4.2: Timings of i860 assembly language routines on an MK096.

routine	length	oper	mem	repeat	time μs	Mflops	Mbyte/sec
psi2chi	5184	96	480	500	64163136	3.88	19.39
reconstruct	5184	144	480	500	54897536	6.80	22.66
saxpy	124416	2	12	500	33473472	3.72	22.30
sdot	62208	4	8	500	12583680	9.89	19.77
su3_hh	5184	198	216	500	109559488	4.68	5.11
su3_hh_g	5184	198	216	500	111122624	4.62	5.04
su3_hh_s	5184	198	216	500	111028928	4.62	5.04
su3_hm	5184	198	216	500	101674304	5.05	5.51
su3_hm_g	5184	198	216	500	103154816	4.98	5.43
su3_hm_s	5184	198	216	500	103181952	4.97	5.43
su3_mh	5184	198	216	500	102217600	5.02	5.48
su3_mh_g	5184	198	216	500	103800320	4.94	5.39
su3_mh_s	5184	198	216	500	103758656	4.95	5.40
su3_mm	5184	198	216	500	94416896	5.44	5.93
su3_mm_g	5184	198	216	500	96060992	5.34	5.83
su3_mm_s	5184	198	216	500	96051136	5.34	5.83
su3_mv4_hm_n	5184	264	264	500	80228736	8.53	8.53
su3_mv4_mm_n	5184	264	264	500	79619648	8.59	8.59
su3h	5184	132	168	500	83012032	4.12	5.25
su3hg	5184	132	168	500	84539456	4.05	5.15
su3hs	5184	132	168	500	43795776	7.81	9.94
su3s	5184	132	168	500	83088896	4.12	5.24
su3sg	5184	132	168	500	43713216	7.83	9.96
su3ss	5184	132	168	500	84819904	4.03	5.13

Table 4.3: Timings of i860 C language routines on an MK086.

Chapter 5

Conclusion

5.1 The use of parallel computers in LGT

Immense computer resources are required to perform meaningful lattice gauge theory simulations. All of the important data-sets are four dimensional so even with modest lattice dimensions a large amount of memory and disk space is required. If any fermionic quantities are to be calculated the need to invert the fermion matrix requires a prodigious amount of raw computer power. With current computer technology any attempt to perform lattice gauge simulations are always constrained by the available computer resources.

Much of this thesis is concerned with the attempts to reduce these limitations. The primary method of doing this is by utilising parallel computers. As explained in the introduction parallel computation offers a significant increase in available computational power at the cost of making program writing more difficult. In common with many physical systems lattice gauge theory calculations can be distributed over several processing elements using a spacial decomposition. This is possible because of the local nature of the physical interactions. The difficulties often arise because of features of the algorithm that do not share this local property. The random number generators discussed in chapter 2 are an example of this. Monte-Carlo algorithms require a source of effectively random numbers. This is a feature of the algorithm rather than the physical processes so this part of the algorithm does not exhibit locality and cannot be distributed using a geometric decomposition. In this case the solution I adopt is to use a separate random number generator on each processor of the parallel machine and to attempt to reduce the correlations between processors as much as possible. Provided the number of lattice sites controlled by each processor is not too small the quality of the individual random number generators is more important than the correlation between processors. There is much more work that can be done in this field and I hope to return to it at some later date.

It is not sufficient to just implement the simulation code in parallel. The current generation of parallel computers provide a significant increase to the computer power available to the individual researcher but the increase is not enough to remove the constraints on lattice simulations. Further programming effort is required to extract the maximum possible performance out of the available hardware and to match the simulation to the particular strengths and weaknesses of the parallel system being used. The form of these optimisations depend on the parallel system and the problem being simulated. Two different cases have been discussed in this thesis; the simulation of QED using a transputer based parallel computer and the simulation of quenched QCD on a hybrid i860/transputer system. It is possible to use this experience to make some general conclusions about lattice gauge theory simulations on distributed memory MIMD computers.

Parallel computers have become more common over the last few years. Even though they are not yet part of the computing mainstream they are now routinely used in some fields, including lattice gauge theory. Parallel computers vary greatly in the way they operate and in the way that they are programmed. This is because large scale parallel computing is still a relatively young subject and various new ideas are still being tried out. All of the work presented in this thesis has been performed on parallel computers constructed using T800 transputers. The T800 communicates using point to point communication links. Before running a program the processors must be configured (wired up) in a topology appropriate to the application. A processor is then only capable of sending messages to the processors it is directly connected to. Messages for remote processors must be forwarded in software. This approach has serious drawbacks. It is impossible to write general purpose code, the application is always strongly effected by this underlying hardware. The QED program only uses replicated sets of 16 processors because of the T800 only has 4 links. The QCD machine represents an evolution away from this approach. It is still uses T800s but these now play the role of dedicated communication chips. The application code runs on the i860 and messages are passed to the T800s for delivery to any processor in the program. Message routing is still performed in software but this is now system software running on dedicated processors rather than being explicitly included into the application code. Even so a dynamically reconfigurable network still leaves us with a non trivial optimisation problem; how best to wire up the network. The wiring files for the QED program took about half an hour each to generate. On the QCD machine wirings for small programs are automatically generated at run time, but the necessary placement for all 64 processors pushes the capabilities of the machine to the limit and had to be generated by hand. As all of the programs use the same wiring this only had to done once, but this process took several hours to do. More recent parallel computers have dispensed with reconfigurable arrays entirely and provide a fixed topology general purpose communication network where all message routing is done in hardware[54].

Both of the simulation programs were written in a very similar style. Each processor in the program runs the same piece of code, this code approximates a conventional single processor program except for the boundary conditions. Instead of implementing the usual periodic or anti-periodic boundary conditions, boundary values are transferred to and from neighbouring processors. This is a relatively small change to the program as a whole and it involves very little effort over that needed for an equivalent single processor program. In both cases the main difficulty encountered was parallel file access. As with the random number generators these difficulties arose because this is a non-local operation. The QED program was written in OCCAM[12], which provided no intrinsic support for parallel file-access. Only the processor directly connected to the host was able to read or write files. All data needed to be sent to this processor before it could be written out to a file. This process was simplified by using a file format that stored the data for each processor in contiguous blocks. This made it very difficult for a sequential program to use these files. The CSTools[49] environment used by the QCD program does support a restricted form parallel file access, each processor is capable of reading and writing its own files. As it is very important that we should be able to access these data files using a variety of programs and machines, we needed to write the data in a standard format that did not depend on a particular hardware configuration. Again this requires a single processor to distribute/collect the data and read/write the file. We need to access several different types of file in this fashion so a library of low-level parallel file access routines have been developed and the routines needed to read and write each type of file have been constructed using this library. Hopefully similar libraries will be provided by the manufactures of future parallel systems. As the QCD simulation utilised the quenched approximation and a much larger lattice than the QED it produces a greater volume of data. This gave rise to additional problems because the host processor is essentially a workstation and has a limited IO capability. Ideally a parallel computer requires high-performance IO systems to match their computational speed. If this performance is to scale with the size of the computer the system must be able to support parallel data access to multiple storage devices. Unless the hardware and system software is carefully designed this will conflict with our desire to store data in a format that is independent of the hardware configuration.

The programming style described here makes no attempt to automate the distribution of the problem. The decisions about how the arrays are distributed and about how the boundary values are communicated are all made by the applications programmer. It would be perfectly possible to automate much of this work, either at the compiler stage, for example by using the data-parallel programming model, or by providing libraries of specialised routines to support regular domain decomposition. Neither type of package is available at the moment on the machines that were used for this work. Even if such packages had been available it is probable that they would require a greater amount of memory than the programs we actually use. By using our knowledge of the application we are able to reduce the requirement for communication buffers to a bare minimum. A compiler or library package has to address a more general case and cannot make as many savings. Nevertheless I expect future parallel lattice gauge simulations to pay this price in exchange for much simpler program development.

5.2 QED

The phase diagram of strong-coupling non-compact lattice QED with an additional four-fermion interaction has been deduced using a series of dynamical fermion simulations. The mass dependence of the system has been investigated for non-compact QED and along the $\beta = 2.0$ axis which is close to a system with only four-fermi interactions. There seems to be a line of chiral-symmetry-breaking transitions in the β-G plane connecting the strong-coupling QED transition the four-fermi phase transition. There is no evidence that the strong-coupling phase transition can be used to construct a continuum theory. Our analysis suggests that the transition has mean-field scaling exponents. This is supported by fitting the data to the gap-equation. The gap-equation is derived using a mean-field approximation and the results of our calculations appear consistent with a solution of the gap-equation.

The Swendsen Ferrenberg extrapolation technique has been applied to our data on the $G = 0$ axis. This technique was originally developed for spin-models, where a few high-statistics simulations can be used to generate results for a range of coupling constant values. Because our simulation used dynamical fermions there was difficulty obtaining high statistics. By utilising data from several simulations we were able to calculate results with an effective statistics higher than from the individual simulations taken in isolation. In addition results can be calculated for any value of the coupling that lies between the simulated values. By calculating the effective statistics as a function of the coupling we are also able to obtain a quantitative understanding of how the spacing of our simulations across the phase diagram effects our knowledge of the system.

5.3 QCD

The QCD simulation discussed in chapter 4is the result of an ongoing collaborative project (UKQCD). The work presented in this thesis is mostly concerned with the implementation of the necessary programs as this reflects my own contribution to the project. Most of this work has been the development of i860 assembly language routines. It also includes the development of libraries of parallel file access functions and inter-processor communication functions. These routines have been used in the "pure gauge" simulation code, the quark propagator codes and in an $SU(2)$ simulation code that also forms part of the UKQCD program[55]. We have attempted to restrict all of the Meiko/cstools specific code to these libraries so that these programs will be easy to port to other i860 based MIMD computers such as the Intel ipsc/860 or the Intel "paragon".

This work has been directed towards an ongoing program of science, centered around large lattice QCD simulations. This work is a collaborative effort involving a large number of people. As my contribution to this work has mostly been in developing and maintaining the enabling technology described in chapter 4, only a brief overview of the results are given here. This work is presented in more detail in the collaboration publications [55, 56, 57]. We simulate quenched QCD on a $24^3 \times 48$ lattice, at a $\beta = 6.2$. The gauge fields are updated using a cycle of three-subgroup Cabibbo-Marinari [58] heat-bath sweep followed by 5 over-relaxed sweeps. Configurations are sampled every 400 of these composite update steps. For each of these sampled configurations, quark propagators are calculated at a number different quark masses. Propagators are calculated using both the standard Wilson fermion action and a nearest-neighbour $O(a)$ improved or "clover" fermion action [59, 60], using the same set of gauge configurations for both actions. The lattice-action for the gauge sector is correct up to $O(a^2)$ terms. The Wilson fermion action is only correct to $O(a)$. The "clover" action is similar to the Wilson action but contains $O(a)$ corrections. By using this improved action we hope to reduce the systematic errors due to the finite lattice spacing, without having to increase the lattice size any further. Propagators are calculated for $r = 1$ at $\kappa = 0.1510$, 0.1520, 0.1523, 0.1526 and 0.1529 for the Wilson action, and at $\kappa = 0.14144$, 0.14226, 0.14244, 0.14262 and 0.14280 for the clover action; the latter values were chosen to match roughly the pion masses computed in the Wilson case. We use an over-relaxed minimal residual algorithm with red-black preconditioning for propagator calculations These quark propagators have been used to calculate hadron masses. Edinburgh plots for the two actions are given in fig. 5.1. This data was generated using 18 configurations. The plots are broadly consistent, showing a trend towards the physical value for m_N/m_ρ with decreasing pion mass.

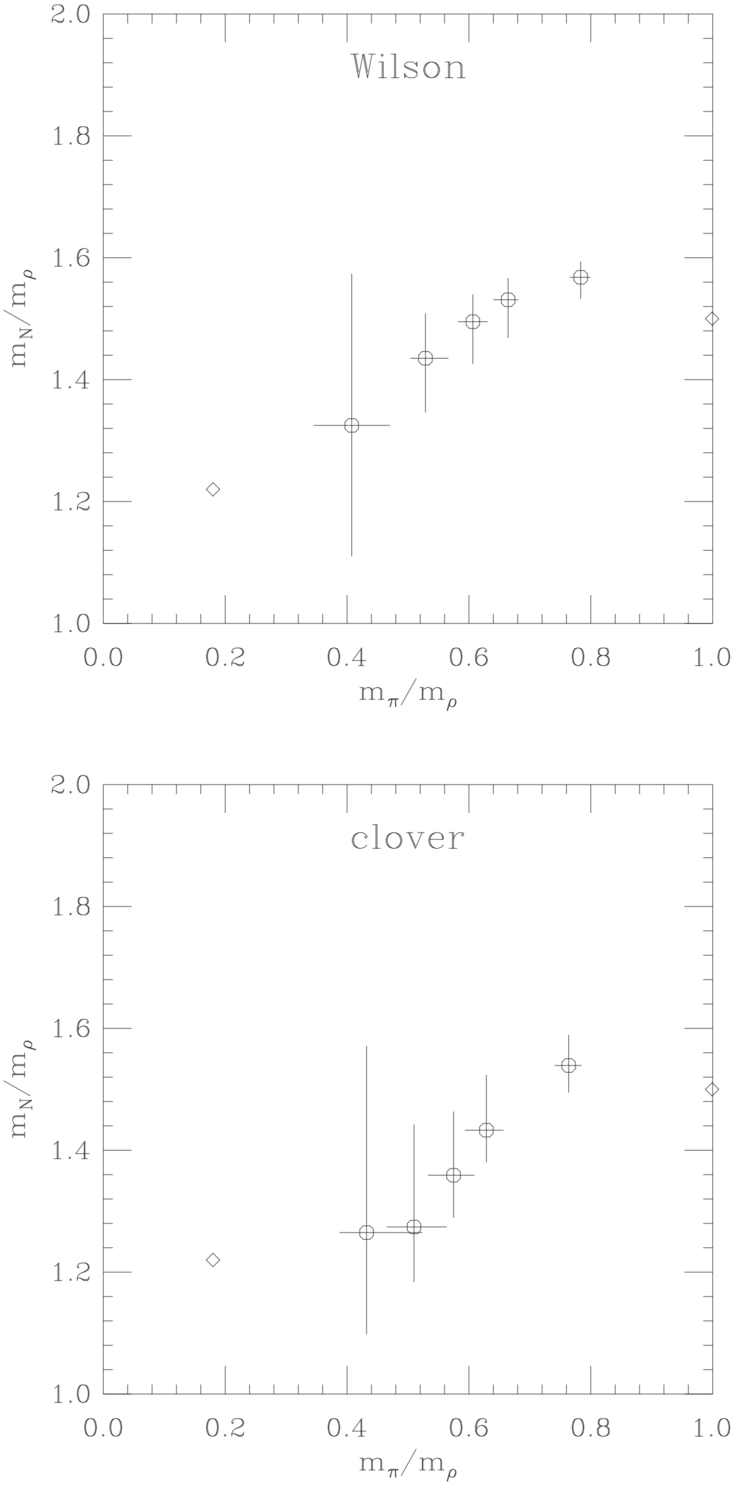
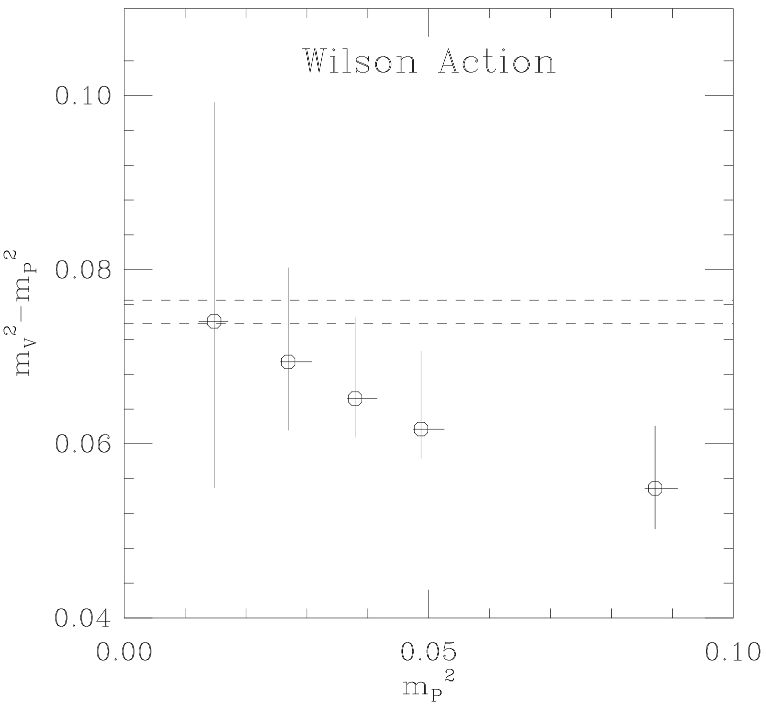


Figure 5.1: Edinburgh plots.

We see no statistically significant difference between the two actions at light quark masses, though there is some preliminary evidence for improvement in the heavy quark regime. This comes from the splitting between the squares of the vector and pseudoscalar meson masses; experimentally, this quantity is very nearly the same for the $\rho - \pi$ system and for corresponding mesons containing a strange, charm or bottom quark. This is a quantity which may be sensitive to the different discretisation errors in the two formulations. Our lattice results are shown in Figure 5.2. Neither result is in agreement with the experimental values for heavy quark masses. The "clover" action seems to be better than the Wilson action in this regime. This is to be expected; because heavy quarks have a shorter wavelength, O(a) effects will be more significant for heavy quarks.

This work is still in progress. In addition to increasing the statistics we are also intending to calculate matrix-elements and to perform further heavy-quark calculations using the "clover" action.



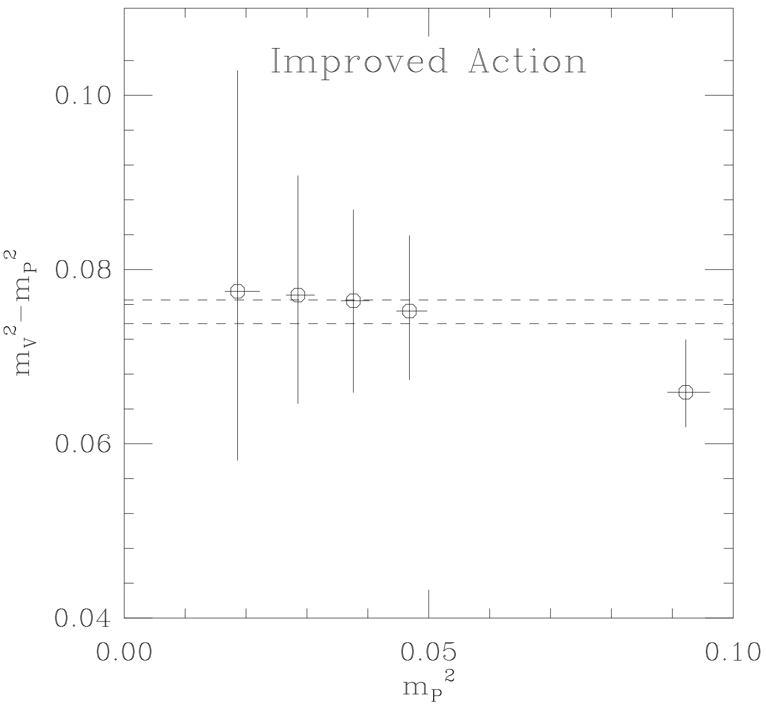


Figure 5.2: $m_V^2 - m_P^2$ versus m_P^2.

The dashed lines correspond to the experimental range. The highest mass point corresponds to the physical quark mass.

Bibliography

[1] R. Feynman and A. Hibbs
Quantum mechanics and path integrals
McGraw-Hill (New York 1965)

[2] D. Bailin and A. Love
Introduction to Gauge Field Theory
Adam Hilger (Bristol and Boston 1986)
ISBN $0 - 85274 - 818 - 3$.

[3] N. Metropolis, A. Rosenbluth, M. Rosenbluth,
A. Teller and E. Teller
J. Chem Phys. 21(6):1087-1092, June 1953

[4] L. Susskind,
Phys. Rev. D16 (1977) 3031

[5] K. Wilson
Adv. Math. 16 (1975) 176

[6] S. Duane, A. Kennedy, B. Pendleton and D. Roweth
Phys. Lett. 195B (1987) 216.

[7] S. Booth R. Dobinson D. Jeffery W. Lu
K. Storr and A. Thornton
Comput. Phys. Commun. 57 (1989) 486.

[8] S. Booth, K. Bowler, D. Candlin, R. Kenway,
B. Pendleton, A. Thornton and D. Wallace.
Comput. Phys. Commun. 57 (1989) 101.

[9] D. Stephenson and A. Thornton
Phys. Lett. B 212 (1988) 479.

[10] S. Gupta, A. Irbäck, F. Karsch and B. Petersson
Phys. Lett. B242 (1990) 437-443.

[11] Thinking Machines Corporation
Connection Machine CM-200 series technical summary.

[12] K. Bowler, R. Kenway, G. Pawley and D. Roweth
An introduction to Occam 2 programming,
Student-litterature. Chartwell-Bratt, 1987.

[13] M. Metcalf and J. Reid
Fortran 90 *explained*
Oxford science publications (1990)
ISBN $0-19-853772-7$.

[14] AMT Ltd.
DAP Series FORTRAN PLUS Language, 1990

[15] Thinking Machines Corporation
CM Fortran Reference Manual.

[16] D. Knuth
The art of computer programming, Vol 2 Ch 3
Addison Wesley (Reading, Mass 1973)
ISBN $0-201-03822-6$

[17] G. Marsaglia
Computer Science and Statistics 16th Symposium on the interface,
Atlanta, March 1984

[18] R. McEliece
Finite fields for computer scientists and engineers
Kluwer (Boston 1987)

[19] D. Knuth
The art of computer programming, Vol 2 Ch 4

Addison Wesley (Reading, Mass 1973)
ISBN $0-201-03822-6$

[20] G. Marsaglia and L. Tsay,
Linear Algebra and its applications, 67, 147-156, 1985.

[21] T. Lewis and W. Payne,
J. Assoc. Comput. March. 20, 456-468 (1973).

[22] B. Ripley
Stochastic simulation
Wiley (New York Chichester 1987)

[23] M. Fushimi and S.Tezuka,
Comm. ACM 26, 516-523 (1983).

[24] G. Marsaglia and A. Zaman
Towards a universal random number generator,
Florida State preprint, 1987.

[25] J.D. Bjorken and S.D. Drell,
Relativistic Quantum Fields
(McGraw-Hill) 166-170

[26] M. Lüscher,
Nucl. Phys. B341 (1990) 341-357

[27] M. Lüscher and P. Weisz,
Nucl. Phys. B290[FS20] (1987) 25

[28] J.B. Kogut, E. Dagotto and A. Kocic,
Phys. Rev. Lett. 60 (1988) 772.

[29] J.B. Kogut, E. Dagotto and A. Kocic,
Nucl. Phys. B317 (1989) 271

[30] R. Fukuda and T. Kugo,
Nucl. Phys. B117 (1976) 250

[31] V.A. Miransky,
Nuovo Cim. 90A (1985) 149

[32] C.N. Leung, S.T. Love and W.A. Bardeen,
Nucl. Phys. B273 (1986) 649

[33] A. Cohen and H. Georgi,
Nucl. Phys. B314 (1989) 7

[34] Y. Nambu and G. Jona-Lasinio,
Phys. Rev. 122 (1961) 345

[35] T. Appelquist, M. Soldate, T. Takeuchi and L.C.R. Wijewardhana,
Effective four-fermion interactions and chiral symmetry breaking,
Proceedings of the 12th Johns Hopkins Workshop on Current Problems in Particle Theory, Baltimore 1988

[36] K. Kondo, H. Mino and K. Yamawaki,
Phys. Rev. D39 (1989)

[37] T. Nonoyama, T.B. Suzuki and K. Yamawaki,
Prog. Theor. Phys. 81 (1989) 1238

[38] V.A. Miransky and K. Yamawaki,
Mod. Phys. Lett. A4 (1989) 129.

[39] J. Oliensis and P.W. Johnson,
Dynamical chiral symmetry breaking in strong coupling unquenched QED_4 ,
Argonne preprint ANL-HEP-PR-88-45 (Aug 1988)

[40] E. Dagotto, A. Kocic and J.B. Kogut,
Phys. Lett. B232 (1989) 235.

[41] E. Dagotto, A. Kocic and J.B. Kogut,
Finite Size, Fermion Mass and N_f *Systematics in Computer Simulations of Quantum Electrodynamics*,
Illinois preprint ILL-(TH)-89-#34, July 1989.

[42] S. Booth, R. Kenway and B. Pendleton
Phys. Lett. B228 (1989) 115

[43] A.M. Horowitz,
Phys. Lett. 219B (1989) 329.

[44] I-H. Lee and R.E. Shrock,
Phys. Rev. Lett. 59 (1987) 14.

[45] A. Horowitz
Nucl. Phys. B(Proc. Suppl.) 17 (1990) 694-698.

[46] S. Booth, R. Kenway, B. Pendleton and A. Horowitz
Nucl. Phys. B(Proc. Suppl.) 17 (1990) 691-693

[47] A. Ferrenberg and R. Swendsen
Phys. Rev. Lett. 61 (1988) 2635

[48] A. Ferrenberg and R. Swendsen
Optimized Monte-Carlo data analysis,
Caltech preprint.

[49] Meiko Ltd
CSTools for SunOS

[50] L. Clarke and G. Wilson
Tiny: An efficient routing harness for the Inmos Transputer
Concurrency practice and experience (1990)

[51] Green Hills Software, Inc.
C-860 User's Guide; Fortran-860 User's Guide.

[52] Intel corporation
i860 64-*bit microprocessor programmer's reference manual*

[53] Y. Oyanagi
Comp. Phys. Comm. 42 (1986) 333

[54] Thinking Machines Corporation
Connection Machine CM-5 technical summary.

[55] UKQCD collaboration
Phys. Lett. B275 (1992) 424-428.

[56] UKQCD collaboration
Nucl. Phys. B (Proc. Suppl.) XXX (1992) 1-6.

[57] UKQCD collaboration
Quenched Hadrons using Wilson and $O(a)$-Improved Fermion Actions at $\beta = 6.2$,
Edinburgh Preprint: 92/506,
Southampton Preprint: SHEP 91/92–15

[58] N. Cabibbo & E. Marinari,
Phys. Lett. 119B (1982) 387.

[59] B. Sheikholeslami & R. Wohlert,
Nucl. Phys. B259 (1985) 572.

[60] G. Heatlie et al.,
Nucl. Phys. B352 (1991) 266.