Portable lattice QCD software for massively parallel processor systems

Nicholas Paul Stanford

Doctor of Philosophy

University of Edinburgh

1994

Abstract

Quantum Chromodynamics (QCD), which models the interactions of quarks and gluons, forms part of the standard model, currently the best theoretical framework of unified particle interactions. Lattice QCD is a method of simulating the theory of QCD in a discretised form on computers. This approach to particle physics is vitally important for providing a comparison with experimental measurements and predicting new particle properties. To implement lattice QCD we require very high performance computers, the latest generation of which are known as Massively Parallel Processors (MPPs). These are available in two main distinct architectures, Multiple Instruction Multiple Data (MIMD) and Single Instruction Multiple Data (SIMD).

We present a suite of lattice QCD software intended to be portable across all currently available MPP platforms. This is achieved by utilising emerging standards in parallel programming languages. We use subset High Performance Fortran for SIMD machines and the PVM message passing package, with provision for the forthcoming Message Passing Interface (MPI) standard, for MIMD machines. Software engineering techniques are used to design and document a package which delivers a high output of physics results without a large investment in optimisation for new platforms. This is achieved while still preserving the major requirements of reducing memory demands and increasing speed and user understanding. Detailed procedures for testing the package and validating results are presented, without which there could be little confidence in the physics generated.

To evaluate the efficiency of the software suite we present timings for important code sections generated on a range of MPP platforms.

Declaration

The work discussed in this thesis was undertaken in collaboration with Stephen Booth at Edinburgh University under the management of Richard Kenway. I made a substantial contribution to the work; my responsibility was the introduction and application of software engineering techniques, drafting of most documentation, co-designing and implementing the top-level message-passing software, designing and implementing the user interface, and the design and implementation of all data-parallel software. I was not heavily involved in the low-level message-passing design and implementation. All test data and performance data was generated by me, except where stated in the text. This thesis has been wholly composed by me.

N.P. Stanford

Acknowledgements

I would like to thank the EPSRC for funding for the duration of my PhD; Richard Kenway and Stephen Booth for their expertise and hard work which went into the creation of the MPP software suite; Ken Bowler for his advice on my thesis content and structure; Hugh, Henning, Nick and John for their sharing of the PhD experience; Mike Peardon for illuminating unwanted ‘features’ in my code; Edinburgh Parallel Computing Centre for the use of a Connection Machine CM200 and Cray T3D; Peer Ueberholtz at Wuppertal University for the use of a Connection Machine CM5 and advice on its use; Andersen Consulting for providing me with post-PhD employment and the light at the end of the tunnel. And finally, my wife Alison for keeping me in high spirits throughout the final year.

Nick Stanford

Contents

List of Abbreviations

ABRC	The Advisory Board to the Research Councils.
ANSI	The American National Standards Institution.
BCs	Boundary conditions.
CM	Connection Machine or Cabibbo-Marinari update depending on context.
DP	Data-parallel.
DR	Data repository. A large central data structure used in the ECU application.
DMA	Direct Memory Access. A method of inserting/removing data directly to/from memory without the central processor being involved.
ECU	Executive Control Utility. The user interface to the MPP software suite.
F77	Fortran 77.
F90	Fortran 90.
FE	Front end. The host computer or processor to an MPP system.
HMC	Hybrid Monte Carlo. The algorithm used to create unquenched gauge configurations.
HPF	High Performance Fortran.
$\subset$**HPF**	Subset High Performance Fortran.
I/O	Input/Output.
MC	Monte-Carlo.
MIMD	Multiple instruction stream, multiple data stream.
MP	Message-passing.
MPI	Message-passing interface. A standard message-passing package.
MPP	Massively-parallel processor.
OR	Over-relaxed update.
PVM	Parallel virtual machine. A standard message-passing package.
QCD	Quantum Chromodynamics.
RNG	Random-number generator.
SIMD	Single instruction stream, multiple data stream.
SPMD	Single program, multiple data stream.
SW	Sheikholeslami-Wohlert, or Clover, action.

Chapter 1

Lattice QCD for MPP systems

In this chapter we discuss the physics on which the MPP project is based. Our starting point is continuum QCD. We then proceed to place the theory on the lattice and examine the Monte Carlo techniques required to simulate such a theory. Sections 1.3 and 1.4 look at the mechanisms for generating gauge configurations in both the quenched and unquenched regimes. The generation of quark propagators in the background gauge configurations is presented in section 1.5, with the methods for generating the quark sources discussed in section 1.8.

In sections 1.6, 1.7 and 1.9 we take a cursory look at the other elements required to generate useful physics. These elements are not all implemented on MPP machines, and are therefore outwith the subject of this thesis. However, provision must be made for them to interface with the existing applications in the suite.

Once the physics has been discussed, we examine the motivation for the creation of a new software suite, the Cray T3D purchased by the ABRC and the need to run on other platforms, then the two machine architectures on which the project is implemented, MIMD and SIMD, and the programming environments available. We finally present the principles guiding the creation of the new suite of software and the base for its development.

1.1 From continuum to lattice

Several excellent references for a general introduction to lattice QCD are [1, 2, 3, 4].

1.1.1 Continuum QCD

Quantum Chromodynamics (QCD), part of the standard model of particle interactions, models the interaction between quarks and gluons. The theory can be expressed in terms of the QCD Lagrangian $\mathcal{L}_{\mathrm{QCD}}$ [5] , the fields A_μ represent gluons and $q_k, \overline{q}_k$ represent quarks of flavour k.

$$\begin{aligned} \mathcal{L}_{\mathrm{QCD}} &= -\frac{1}{2}\mathrm{Tr}\ F_{\mu\nu}F^{\mu\nu} + \sum_k^{n_f} \overline{q}_k(i\gamma^\mu D_\mu - m_k)q_k \\ F_{\mu\nu} &= \partial_\mu A_\nu - \partial_\nu A_\mu - ig\left[A_\mu, A_\nu\right] \\ D_\mu q_k &= (\partial_\mu - igA_\mu)q_k \\ A_\mu &= \sum_{a=1}^{8} \frac{A_\mu^a \lambda^a}{2} \end{aligned} \tag{1.1}$$

where the λ^as are Gell-Mann matrices satisfying the SU(3) commutation relation

$$\left[\frac{\lambda_a}{2}, \frac{\lambda_b}{2}\right] = if^{abc}\frac{\lambda^c}{2} \tag{1.2}$$

and normalisation condition

$$\mathrm{Tr}\ \left(\lambda^a\lambda^b\right) = 2\delta^{ab} \tag{1.3}$$

The quantum mechanical expectation value of an operator $\mathcal{O}(\overline{q}, q, A)$ may be written as a functional integral in Euclidean spacetime

$$\langle\mathcal{O}\rangle = \frac{1}{Z}\int d[\overline{q}]d[q]d[A]\mathcal{O}(\overline{q}, q, A)e^{-S(\overline{q},q,A)} \tag{1.4}$$

where Z is the partition function, defined by the condition $\langle 1\rangle = 1$,

$$Z = \int d[\overline{q}]d[q]d[A]e^{-S(\overline{q},q,A)} \tag{1.5}$$

and S is the action, $S = \int d^4x\mathcal{L}_{\mathrm{QCD}}$.

Since numerical/computational treatment of the integrals over the grassmannian variables $\overline{q}$ and q is not efficient, we integrate them out. If the form of the action is

$$S = S_G(A) + \overline{q}M(A)q, \tag{1.6}$$

where S_G is the part of the action depending only on the gauge fields and M is the *fermion matrix*, the integrated result is

$$Z = \int d[A] \det M(A) e^{-S_G(A)} \tag{1.7}$$

The problem with simulating this on a computer is that $\det M$ is highly nonlocal and requires enormous amounts of computer time to calculate. We shall see how to perform this simulation in section 1.4, however a simpler solution exists. The *quenched approximation* consists of setting

$$\det M = 1 \tag{1.8}$$

which corresponds physically to the removal of virtual quark loops in the background gluon fields, or letting the masses of the virtual quarks tend to infinity. Part of the work of lattice QCD research is to evaluate the effects and significance of the quenched approximation.

1.1.2 Lattice QCD

Computers cannot deal with continuous variables so some form of discretisation needs to be effected in order to extract numerical results. Wilson [6] discovered how to do this by defining a Euclidean 4-D hypercubic lattice *i.e.* space and time are treated equally. A natural regularisation is then introduced which ensures convergence of integrals when calculating physical quantities and preserves gauge invariance.

The gluons are formulated by elements of the gauge group SU(3). One associates a gauge variable $U_\mu(x)$ with every *link* in the lattice connecting the site x to site $x+\hat{\mu}$. Path ordering of the path integral requires that travelling along the link in the opposite direction gives

$$U_{-\mu}(x+\hat{\mu}) = U_\mu^\dagger(x). \tag{1.9}$$

Fermion fields, situated at the sites of the lattice, carry both colour and spinor indices; they are represented by 3×4 dimensional complex matrices.

We can now approximate the functional integral by multiple integrals over the group-valued link elements

$$Z = \int \prod_i dU_i e^{-S_g[U]} \tag{1.10}$$

1.1.3 Gauge invariance and the lattice action

1.1.3.1 Gauge invariance

As in the continuum theory gauge invariance dictates the form of the lattice action. On the lattice the effect of an independent gauge rotation at each site is

$$\begin{aligned} q(x) &\rightarrow V(x)q(x) \\ \overline{q}(x) &\rightarrow \overline{q}(x)V^\dagger(x) \\ U_\mu(x) &\rightarrow V(x)U_\mu(x)V^\dagger(x+\hat{\mu}) = U'_\mu(x), \end{aligned} \tag{1.11}$$

where $V(x)$ is a gauge rotation, $V(x) \in SU(3)$, in the same representation as $U_\mu(x)$.

The only gauge-invariant quantities which can be constructed from *gauge fields alone* are *Wilson Loops*, and are calculated by taking the trace over colour indices of the product of gauge fields around a closed loop *e.g.* the 1×1 Wilson

Loop or *plaquette*

$$\mathrm{Tr}\ U_\mu(x)U_\nu(x+\hat{\mu})U_\mu^\dagger(x+\hat{\nu})U_\nu^\dagger(x) = \mathrm{Tr}\ U_{\Box\mu\nu}(x) \tag{1.12}$$

To verify the gauge invariance of this and similar objects simply substitute equation 1.11 into equation 1.12. The comparison of gauge invariant quantities before and after a gauge rotation provides us with a valuable method for testing code, as we shall see later.

1.1.3.2 Lattice action

The choice of action is crucial, bringing in such topics as fermion doubling and the problems associated with it. We shall not discuss them here and merely present the chosen action.

The action S is composed of gauge and fermion parts

$$S = S_G + S_F. \tag{1.13}$$

The gauge action S_G is given by

$$S_G = \beta \sum_x \sum_{\mu<\nu} \left(1 - \frac{1}{N}\mathrm{ReTr}\ U_{\Box\mu\nu}(x)\right) \tag{1.14}$$

where

$$\beta = \frac{2N}{g^2} = \frac{6}{g^2} \text{ for SU(3).} \tag{1.15}$$

The gauge action is equivalent to

$$S_G = \int d^4x \left\{\frac{1}{4}F_{\mu\nu}^2 + O(a^2)\right\} \tag{1.16}$$

where

$$F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu + ig\left[A_\mu, A_\nu\right] \tag{1.17}$$

i.e. the continuum Yang-Mills action with a discretisation error of order a^2, where a is the lattice spacing. The β–value (see equation 1.15) is important in lattice gauge theory as it specifies the strength of the coupling in the theory and hence the strength of the colour force.

The lattice fermion action S_F is constructed from the Wilson term [6] and and additional term to remove discretisation errors of $\mathcal{O}(a)$, the Sheikholeslami-Wohlert (SW) term [7, 8, 9]. The action can be written in the bilinear form

$$S_F = \overline{q}(x) M q(x). \tag{1.18}$$

where M is the *fermion matrix*.

The fermion matrix can be written as

$$M = A - \kappa\Delta \tag{1.19}$$

where A is the SW term, and $-\kappa\Delta$ is the Wilson term. The lattice hopping parameter κ is related to the quark mass m_q by

$$\kappa = \frac{1}{2m_q + 8}. \tag{1.20}$$

The *hopping term* Δ is defined by

$$(\Delta q)(x) = \sum_{\mu} (1 - \gamma_\mu) U_\mu(x) q(x + \hat{\mu}) + (1 + \gamma_\mu) U_\mu^\dagger(x - \hat{\mu}) q(x - \hat{\mu}), \tag{1.21}$$

and is related to the amplitude that a quark will hop between neighbouring lattice sites.

The SW term A is defined by

$$A = 1 - \kappa C \sigma_{\mu\nu} F_{\mu\nu} \tag{1.22}$$

where the coefficient C allows variation of the action from Wilson ($C = 0$) to SW ($C = 1$). Other values can be used, such as 1.4 from mean field improvement to remove tadpole diagrams in [10], but will not be considered here. The field strength $F_{\mu\nu}$ can be written

$$F_{\mu\nu}(x) = \frac{Q_{\mu\nu}(x) - {Q_{\mu\nu}}^\dagger(x)}{2i} \tag{1.23}$$

where the $Q_{\mu\nu}$ (see figure 1.1), defined as

$$Q_{\mu\nu}(x) = \frac{1}{4}\sum_{i=1,4} U^{i}_{\Box\mu\nu}(x), \tag{1.24}$$

give this action the name 'Clover'.

To calculate matrix elements under the $\mathcal{O}(a)$-improvement scheme we must use rotated quark fields. These are obtained by applying the following transformation [1]

$$\begin{aligned} \overline{q}(x) &\longrightarrow \overline{q}(x)(1+\frac{1}{2}\overleftarrow{\not{D}}) \\ q(x) &\longrightarrow (1-\frac{1}{2}\overrightarrow{\not{D}})q(x) \end{aligned} \tag{1.25}$$

for $\mathcal{O}(a)$ improvement, where

$$\left(\overrightarrow{\not{D}}q\right)(x) = \frac{1}{2}\sum_{\mu}\gamma_\mu U_\mu(x)q(x+\hat{\mu}) - \gamma_\mu U^{\dagger}_{\mu}(x-\hat{\mu})q(x-\hat{\mu}) \tag{1.26}$$

This only involves next-to-nearest-neighbour communication and is therefore relatively easy to implement on a parallel machine. The $\sigma_{\mu\nu}$ and γ matrices used are defined in Appendix A.

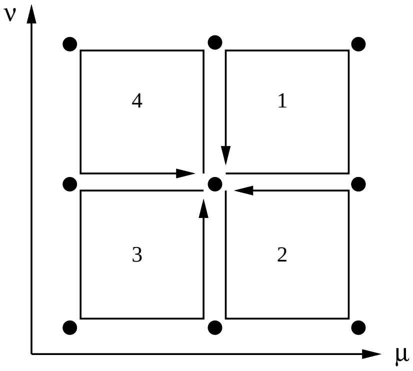


Figure 1.1: Definition of $Q_{\mu\nu}$. Point x is at the centre of the Clover-leaf. The sense of orientation of the plaquettes used for calculating $Q_{\mu\nu}$ is taken such that the first gauge field in the plaquette product is leaving point x.

---

[1] The implementation of the rotations is discussed in more detail in Appendix C.

1.1.4 Monte Carlo methods

To calculate the integral in equation 1.10 we need to integrate over several million variables. This is not possible to do exactly, so we are forced to approximate the integral by a sum over a sample of points in variable space. Further information about use of Monte Carlo methods in statistical physics can be found in [11, 12].

If we generate some configurations of the gauge links U_i with probability distribution $\propto \exp(-S(U_i))$, a method known as *importance sampling*, then we can calculate the expectation value of a suitable observable $\mathcal{O}$ by averaging over the configurations generated *i.e.*

$$\langle \mathcal{O} \rangle = \frac{1}{N} \sum_{i=1}^{N} \mathcal{O}(U_i). \tag{1.27}$$

This is explained further in section 1.3.

1.1.4.1 Gauge configurations

Computers cannot simulate a continuous object such as a link, so gauge variables must be represented at points. A *gauge configuration* then, is an array of SU(3) matrices, one for each direction at each site of the lattice.

1.2 Overview of physics elements

The logical order in which the different lattice QCD physics elements in the MPP software suite fit together is shown in figure 1.2. Only GAUGE, SOLVER and SOURCE will be discussed in detail as part of this thesis.

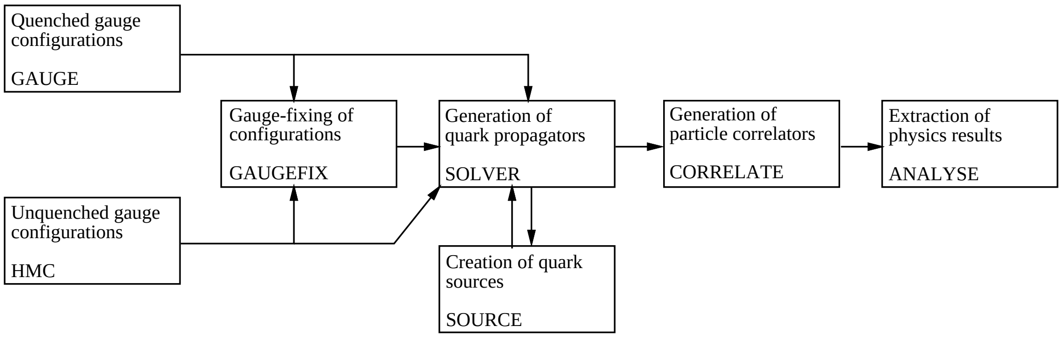


Figure 1.2: The interconnection of the various physics elements in the MPP software suite.

1.3 Generating quenched gauge configurations

In section 1.1.1 we saw that by setting $\det M = 1$, the quenched approximation, the amount of computer time required to calculate expectation values is reduced.

In order to obtain a meaningful average using Monte Carlo methods we need to sample statistically uncorrelated configurations. This is done by creating a Markov chain of configurations. The probability distribution generated tends to that required if the *Metropolis algorithm* is used [13, 14]:

1. Propose a random update and evaluate action change δS.
2. Accept updated configuration with probability

$$p_{\mathrm{acc}} = \min(1, \exp(-\delta S)) \tag{1.28}$$

We are left with choosing the random update method in such a way that the acceptance rate is sufficiently large. The method used is a combination of heatbath and over-relaxed updates and will be described in the following sections.

1.3.1 Heatbath update

We successively place each link of the lattice in contact with a 'heat-bath' which selects a new link variable stochastically with Boltzmann probability $P(U) \sim \exp(-S_G(U))$. This is explored fully in Appendix B.

1.3.2 Over-relaxed update

The heatbath update explores the group manifold at a relatively slow rate if the configurations are to be accepted with any reasonable probability. The aim of over-relaxation is to maximise the change in the gauge links while minimising the change in the action. In practice the action stays constant, removing the need for an accept/reject stage. This is explained fully in Appendix B.

1.3.3 Lattice decomposition

To update more than one link at a time we use the fact that the gauge action couples only next-nearest neighbours and can be written as

$$S_G = -\frac{\beta}{N}\mathrm{ReTr}\ U_\mu R_\mu \tag{1.29}$$

where R_μ is the sum of staples around U_μ (see figure 1.3). For details of how to calculate the staple sum see section B.3.

Since the staple links have to be held constant while the chosen link is updated, the number of links that can be *simultaneously* updated is restricted to half of the links in a certain direction (see figure 1.4).

This is a form of red-black (or odd-even) decomposition. In practice this is achieved by doubling up the lattice in the x-direction[2]. The two sub-lattices

---

[2] The choice of direction is arbitrary, although the t-direction is avoided to aid time-slicing.

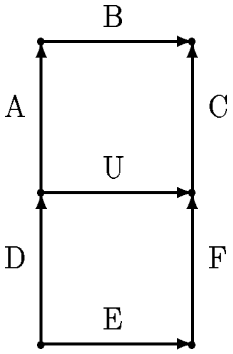


Figure 1.3: Two staples ABC and DEF in the plane around link U to be updated.

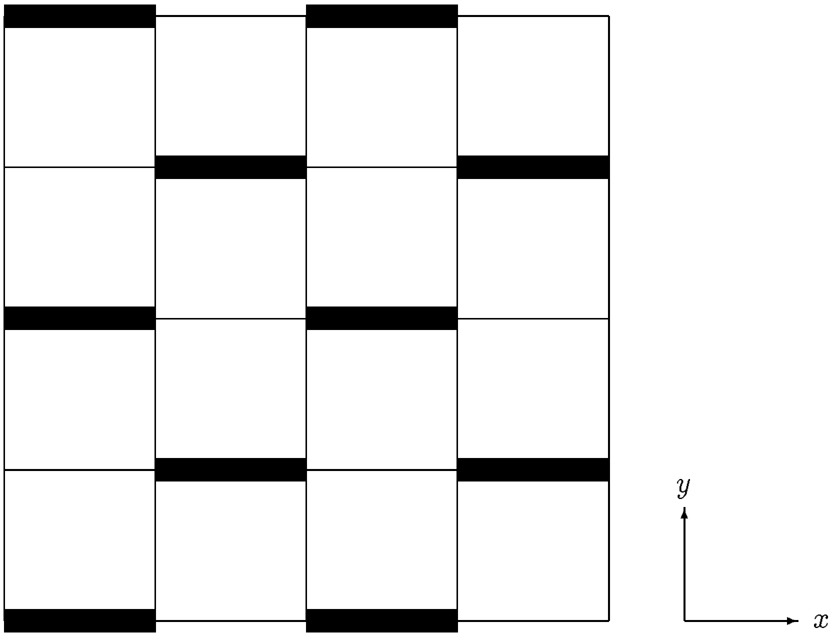


Figure 1.4: Only the thick links in the x-direction may be updated at the *same instant*. This leads to the division of the lattice into two sub-lattices of ODD and EVEN parity.

thus generated are labelled by *parity*[3].

So to update all links the procedure is as follows

```
loop over parity (odd/even)
    loop over direction (x,y,z,t)
        calculate staple sum $R_\mu(x)$ for particular $U_\mu(x)$
        update $U_\mu(x)$
```

1.3.4 Hybrid update algorithm

The Markov chain of configurations is generated by successive sweeps through the lattice, each sweep is referred to as a 'hybrid' or 'compound' update[4]. The compound update consists of the following

- ☐ A local gauge transformation.
- ☐ One or more update stages, each consisting of ...
    - ◇ Heatbath (Cabibbo-Marinari) updates.
    - ◇ Over-relaxed updates.
- ☐ A unitarisation of the gauge matrices.

The gauge transform is a new element to the update proposed by S. Booth at Edinburgh. The idea is to remove any bias in the way the unitarisation is performed.

The unitarisation is performed in order to correct any numerical rounding errors which creep in while performing the updates. The matrices are forced back onto the SU(3) manifold. Full details are given in section B.5.

---

[3] Parity is always defined in these codes by $x + y + z + t$ MOD 2. Since coordinates start from zero, the origin is of *even* parity.

[4] This update algorithm is similar to that in [15].

Since we do not want to correct for numerical errors too often, as a unitarisation takes time to perform, the Cabibbo-Marinari and over-relaxed update stages may be performed more than once per hybrid update.

1.3.5 Initialising, thermalising and selecting configurations

We must start off this Markov chain of configurations in some way. There are two methods supported by our package.

1. Ordered start. Set all gauge matrices to the unit matrix.
2. Disordered start. Set all matrices to independent random values such that $U \in \mathrm{SU}(3)$.

Before we select any configurations we must allow the chain to become thermalised, *i.e.* the configurations are in equilibrium with the heatbath. Once thermalised we must select statistically uncorrelated configurations by evolving the hybrid update for $\mathcal{O}(100)$ iterations. The criterion for this interval between samples is determined by auto-correlations of an observable [16], has a complicated dependence on lattice size, β and the size of the operator and is left to the user to establish.

1.4 Generating unquenched gauge configurations

As explained in section 1.1.1 the quenched approximation is used to eliminate the extensive time required to calculate $\det M$. Although unquenched simulations consume more computer time, they are still performed as they give us an idea of the effect of quenching on the physics generated. As quenching is an uncontrolled approximation we must at some point revert to the full theory to extract realistic physics. The most commonly used simulation technique is 'Hybrid Monte Carlo' (HMC) [17] and will be described briefly below.

The HMC algorithm uses molecular dynamics to evolve the system through a fictitious time variable 'τ', the *simulation time.* The molecular dynamics update introduces systematic errors through the integration of the equations of motion by finite timesteps. This error is removed by a Metropolis accept/reject decision based on the change in the Hamiltonian for the configuration.

1.5 Generation of quark propagators: the solver

When considering an operator of the form $\mathcal{O} = \overline{q}\Gamma(U)q$, the lattice equivalent of equation 1.4 can be written as

$$\langle \mathcal{O} \rangle = \int \prod_i dU_i M^{-1} \Gamma(U) e^{-S_G(U)} \tag{1.30}$$

With a quark source η we need to solve the equation

$$\psi^j_\beta = \left(M^{ij}_{\alpha\beta} \right)^{-1} \eta^i_\alpha \tag{1.31}$$

for ψ^i_β, a single column of the full quark propagator matrix; we do not usually have sufficient memory space to solve for the full quark propagator. The indices i, α are the source colour/spin and j, β the sink colour/spin. The solution of this large sparse system of linear equations must be performed using an iterative scheme as discussed in [16, chapter 2].

1.5.1 Preconditioning

Experience within UKQCD has shown that two types of preconditioning improve convergence of the linear equation solver; our package has both built in as standard. The first is due to [18] and reduces the off-diagonal elements of the fermion matrix in the following way.

We define

$$\begin{aligned} M' &= (A - \kappa^2 \Delta A^{-1} \Delta) \\ \eta' &= (1 + \kappa \Delta A^{-1}) \eta \end{aligned} \qquad (1.32)$$

such that $M'\psi = \eta'$. The hopping term Δ is defined in equation 1.21.

The second method is that of red-black preconditioning. As in section 1.3.3 we split the lattice into 2 sub-lattices labelled by parity p and $\overline{p}$. The Clover term A connects sites of equal parity and Δ connects sites of opposite parity. We can therefore solve

$$\begin{aligned} M'_{pp} \psi_p &= \eta_p + \kappa \Delta_{p\overline{p}} A^{-1}_{\overline{p}\overline{p}} \eta_{\overline{p}} \\ &= \eta'_p \end{aligned} \qquad (1.33)$$

and reconstruct the opposite parity solution from

$$\psi_{\overline{p}} = A^{-1}_{\overline{p}\overline{p}} (\eta_{\overline{p}} + \kappa \Delta_{\overline{p}p} \psi_p) \qquad (1.34)$$

This means we can save time and storage space when solving the system. For more details of the derivation of the above preconditioning see [16].

1.6 Gauge fixing

The action defined in section 1.1.3 contains an inherent degeneracy arising from its gauge invariant nature. Any term in the action containing a total derivative of the gauge fields will vanish in the equations of motion. *Gauge fixing* introduces a term in the action which breaks this invariance, satisfying a new gauge fixing condition *e.g.*

$$f(A) = (\partial_\mu A^\mu)^2 = 0 \qquad \text{(Lorentz gauge)} \qquad (1.35)$$

This condition is implemented in practice by iteratively minimising a known function of the gauge fields.

The gauge fixing procedure is not needed for measuring gauge-invariant quantities, *e.g.* particle correlators, but is essential for measurement of *e.g.* gluon propagators [19]. An excellent review of lattice gauge fixing can be found in [20].

1.7 Correlators

A quark propagator is the correlation function

$$\psi^{ij}_{\alpha\beta}(\vec{x},t;0) = \left\langle 0 | q^i_\alpha(\vec{x},t)\overline{q}^j_\beta(0) | 0 \right\rangle \tag{1.36}$$

of the quark fields q, where i, j are colour indices and α, β are spin indices. A propagator for a particle such as a meson is given by

$$C(\vec{x},t) = \left\langle 0 | \Pi(\vec{x},t)\Pi^\dagger(0) | 0 \right\rangle \tag{1.37}$$

where $\Pi(x) = \overline{q}(x)\Gamma q(x)$ and Γ is one of the 16 linearly independent γ-matrix combinations 1, γ_5, γ_μ, $\gamma_5\gamma_\mu$, $\sigma_{\mu\nu}$ giving the required quantum numbers under charge conjugation and parity.

It can be shown [16, chapter 4] that inserting a complete set of spatial momenta and particle states, transforming to momentum space and summing over the spatial volume gives

$$\begin{aligned} C(t) &= \sum_{\vec{x}} C(\vec{x},t) \\ &= \sum_n \left(F_n \mathrm{e}^{-m_n t} + B_n \mathrm{e}^{-m_n (T-t)} \right) \end{aligned} \tag{1.38}$$

for $t < T/2$ where F_n and B_n are the amplitudes of the forward and backward propagating particles. As $t \to \infty$ and $T - t \to \infty$ we are left with the lightest state with a non-zero overlap with the operator Π. The particle mass can then be extracted.

We see therefore that the *correlator*, $C(t)$, is vital for later analysis. Baryon correlators are calculated in a similar way using three valence quarks

$$\Pi_B(x) \sim \varepsilon_{ijk}(q_i(x)\Gamma q_j(x))q_k(x) \tag{1.39}$$

1.7.1 Smearing

To improve the overlap of the lightest state, or an excited state, with the operator we *smear* the quark source and/or sink [21, 22, 23, 24]. The smearing procedure effectively extends the quark source or sink over a finite spatial volume. Many different techniques can be used; see the references for further details.

1.8 Quark sources

To create quark propagators we solve the equation

$$M\psi = \eta \tag{1.40}$$

for the propagator ψ. The simplest source, η, used in practice is the 'point' source, a local spin/colour source of strength 2κ placed at a single site on the lattice (usually the origin). We then use the quark propagators generated to make correlation functions as discussed in section 1.7.

Three-point functions are quantities calculated to determine the matrix elements of flavour changing currents. In this section we will only present the algorithmic elements needed to create such functions, for full details of the technique refer to [25, Chapter 1]. To calculate a three-point function we need to use an 'extended propagator' and a normal propagator as described in the reference. An extended propagator is calculated by applying the following steps.

1. Calculate a quark propagator for hopping parameter value κ_1.
2. Multiply the propagator by a plane-wave momentum factor $e^{i\vec{p}.\vec{x}}$.
3. Multiply the propagator by a Γ matrix factor defining the interaction.
4. Use a single time-slice of the above propagator as the source for a new propagator with hopping parameter κ_2. This is now an extended propagator.

1.9 Analysis

Analysis is a generic umbrella for all processing of correlators needed to generate physics results *e.g.* masses, decay constants and form factors. The theory of these diverse areas will not be discussed here, for further information see [1, 2, 3, 4].

1.10 The targeted system: the need for portable lattice QCD software

The Advisory Board to the Research Councils (ABRC) began a procurement for an MPP system in September 1992 to be used for Grand Challenge projects. The system, a Cray T3D[5], was delivered in April 1994 and UKQCD aimed to be in a position to have codes ready to run as soon as service began.

As this was such a major purchase a benchmarking exercise was performed involving real application codes running on as many platforms as possible. In order to take part in the exercise and to have the ability to run codes on other available platforms, UKQCD decided to design a new suite of lattice QCD software.

---

[5] This is not surprising as 41.3% of the supercomputers in the world are manufactured by Cray Research Inc. (Source: The World's Most Powerful Supercomputers 6/6/94, newsgroup comp.sys.super on usenet)

There is currently a wide variety of both dedicated QCD computers, those designed specifically for QCD computation, and commercial parallel computers used for QCD. Dedicated QCD computers usually gain high performance through finely tuned assembly level code, with little investment in high level compiler technology, largely precluding portability of software. Commercial machines, however, must adhere to at least minimal standards in their programming environments in order to attract a wide customer base. Although portability has not been emphasised in past years, emerging standards are beginning to form a wide portability base for applications.

1.11 MPP architectures

There are two main architectures, identified by Flynn's taxonomy [26], used for MPP systems: SIMD, single instruction stream multiple data stream, and MIMD, multiple instruction stream multiple data stream.

1.11.1 SIMD: single instruction multiple data

SIMD machines are characterised by a large number (typically up to 64K) of processing nodes receiving common instructions broadcast from a central host processor. Synchronism is achieved by utilising a common clocking signal, thus removing the need for synchronization in the software layer and simplifying user application software. Examples of SIMD machines are GF11 [27] and APE [28] (dedicated) or Thinking Machines CM-200 [29] (commercial).

1.11.2 MIMD: multiple instruction multiple data

MIMD machines typically have more powerful node processors with greater memory capacity than the corresponding SIMD elements. Nodes can run independent instructions with synchronism implemented through 'message passing' between them. Memory access is usually private, off-processor data is sent or received by passing messages. Examples of private memory MIMD machines are Columbia [30, 31], CP-PACS [32], 0.5 Teraflops [33], Teraflops [34], QCD-PAX [35, 36] and ACPMAPS [37] (dedicated) or Thinking Machines CM-5 [38], Cray T3D [39], Intel Paragon [40], Fujitsu [41, 42] and Meiko CS-2 (commercial). Shared memory architectures also exist, *e.g.* KSR [43], but are not common due to the reduction of speed from memory access conflicts and the difficulty of designing a communications network to overcome them. They do however obviate the need for message passing resulting in simpler user software.

Our use of MIMD machines is to run the same program on all processors, although not all of them will execute the same conditional branches. This programming model is known as SPMD (single program, multiple data).

1.11.3 Convergence of architectures

The two architectures discussed above are currently distinct, though becoming less so. Machines like the CM-5 and T3D, although MIMD, support SIMD-style operation implemented through the improved communications networks used for synchronisation. This trend is likely to increase as the vendors attempt to capture the markets for both message passing and data parallel programming environments as discussed below. Reviews of QCD machines can be found in [44, 45, 46, 47, 48, 49, 50].

1.12 Programming environments

For the applications software engineer, programming environments are of far greater importance than architecture. The environments supplied again fall into two groups reflecting the underlying architecture: Data Parallel implemented on SIMD machines and Message Passing on MIMD machines.

Most MPP platforms support both C and Fortran compilers, but as it is sensible to write codes in only one language we must select either C or Fortran. There is little published data on the relative performance of C/Fortran generated code, so our choice must be motivated by other criteria. A look at published benchmark data [51, 52, 53, 54] shows that nearly all floating point benchmarks are written in Fortran, while C is primarily reserved for string and integer applications[6]. Vectorising compilers, as discussed in [50, section 6.2] and [56], have traditionally been for Fortran because of its prolific use in numerical simulation and easy analysis of loops compared with C. For these reasons dialects of Fortran for data parallel and message passing are used.

There will of course be some platforms to which it will be impossible to port code. For example the APESE environment on the APE100 processor, an object-oriented programming environment [57].

1.12.1 Data parallel: High Performance Fortran

The data parallel paradigm has been implemented in several dialects, for example CM Fortran [58, 59], Fortran D and Vienna Fortran, based on the array handling constructs of Fortran 90 [60] with added directives for distributing data objects across processor topologies.

As discussed in [61] a standard, High Performance Fortran (HPF), has emerged with major vendors pledging support:

- ☐ **Announced product**: Applied Parallel Research, Kuck and Associates, PGI, Intel, Meiko, Digital.
- ☐ **Announced effort**: TMC, IBM, nCube, NEC, PSR, NASoftware, ACE,

---

[6] An exception to this is the NAS parallel benchmarks [55] which are specified in a language independent way.

Lahey, MasPar, Archipel, Convex.

☐ **Announced interest**: Cray, Hewlett-Packard, Fujitsu, Silicon Graphics, Hitachi, SUN.

Copies of the specification can be obtained by anonymous ftp [62], in published literature [63, 64, 65] or through the World Wide Web [66].

A subset of the language specification [62, section 8] ($\subset$HPF) has been identified as 'being capable of being implemented more rapidly than the full HPF... [and] is intended to be a minimal requirement'. The elements of full HPF omitted from $\subset$HPF do not impose any restrictions on lattice QCD, resulting in the adoption of $\subset$HPF as our data parallel standard. All mention of 'HPF' from this point refers to the subset unless 'full' is stated explicitly.

1.12.2 Message passing: PARMACS, PVM and MPI

Where the implicit synchronism of SIMD machines leads to only a few Fortran dialects (all based on Fortran 90) and a standard language, the wide variety in the design of MIMD machines has resulted in a plethora of message-passing systems for different platforms; Express, PVM, NX/2, Vertex, PARMACS, P4, CHIMP, Zipcode, IBM EUI, CS-tools, LINDA, Canopy and CMMD. A comparison of some of these packages is presented in [67]. Of these packages the most highly standardised are PARMACS [68, 69, 70] and PVM [71, 72], both of which have been ported to a wide range of platforms.

Because of this variety, standardisation has not been as fast as for data parallel programming. However a standard has emerged, MPI [73, 74, 75], along the same lines as HPF and has been implemented by IBM, Argonne National Laboratory (on top of Chameleon, P4 and PVM) [76], Edinburgh Parallel Computing Centre (on top of CHIMP) [77] and Ohio Supercomputer Centre (on top of LAM, a UNIX cluster package) [78]. A book, advertising clear examples on the use of MPI, is in the process of being written [79].

Since MPI has not yet been implemented widely, PARMACS was chosen as the standard for our MPP message-passing layer with Fortran 77 for computation[7]. However problems have been found porting PARMACS because of licencing restrictions and differing implementation features. PVM has since become far more widespread as a standard so we have adopted it also. These changes illustrate that the message-passing code in the MPP package must be kept as simple, flexible, contained and low-level as possible to facilitate change. As we see in Appendix E, which explains the message-passing features used in the MPP codes in terms of PVM, this containment is relatively easy to achieve.

1.12.3 Shared memory/single processor

Shared memory computers are often programmed in normal Fortran 77. No message passing is needed as all memory is accessible from all processors. A 'single processor' version of the message-passing codes can therefore be used. This requires no code other than that used for testing: the single-processor code is vital as a first stage in debugging message-passing code.

1.12.4 Convergence of programming models...the future

As mentioned earlier, SIMD and MIMD designs are converging in the Cray T3D and Thinking Machines CM-5. This convergence is mirrored in the programming environments; CRAFT [81] (Cray Research Adaptive Fortran) currently supports message passing only but advertises data parallel extensions in the Autumn of 1994, the CM-5 can be programmed in either data parallel (CM Fortran [58, 59]) or message passing (CMMD [82]) modes. The HPF specification

---

[7] PARMACS has been proposed as a porting standard by other computational physics researchers, *e.g.*[80] in meteorology.

[62] states its aim as being '[to] provide support for high performance programming on a wide variety of machines, including massively parallel SIMD and MIMD systems and vector processors', so we can once again see convergence.

But what of the relative merits of data parallel and message passing codes? In data parallel programming, once the arrays have been distributed actual computation is straightforward, requiring a minimum of code. Communications are also easily implemented on the periodic lattices required for QCD through the Fortran 90 '`CSHIFT`' command, as demonstrated in Appendix D. Both of these features are in marked contrast to the complicated mechanisms required for message passing and multiple nested loops of Fortran 77.

HPF is therefore most useful for code development and testing; its simpler coding style introduces fewer bugs. Message passing is required for the fastest machines to get high performance; the accuracy of the code can be validated in a single processor mode and against HPF code.

1.12.5 Parallel I/O

The main feature of MPP programming which is not discussed in any standards is that of parallel input and output (I/O). Desirable features and benchmarking of I/O have been discussed in [83], but we conclude that parallel file access must of necessity be machine specific because of the varying topologies of MPP platforms and designs of I/O systems.

1.13 Development base

The UKQCD collaboration has been running lattice QCD codes for several years now; it would be foolish to ignore the codes and algorithms developed and implemented. We present a brief summary of this material in table 1.1.

As can be seen, the Maxwell codes are far more efficient than those on the Connection Machine. This is a necessity; Maxwell is used for production, whereas the CM is primarily for development work.

	Platform		
	Maxwell	**Euclid**	**non-MPP**
Features			
Vendor	Meiko	Thinking Machines	Varied
Model	CS-1/860	CM-200	Varied
No. Nodes	64	512	1
Node type	i860+2×T800	1×32-bit custom+ 1×Weitek 3132	Varied
Topology	4-D hypercube	9-D hypercube	Single proc.
Architecture	MIMD	SIMD	Single proc.
Memory (GByte)	1	0.5	Varied
Speed (peak/ sustained) Gflop/s	5/2	8/1	Varied
Language	C, Ass., CS-tools	CM Fortran	C/Fortran
Application			
GAUGE	$\surd$	$\surd$	×
SOLVER	$\surd$	$\surd$	×
HMC	$\surd$	×	×
GAUGEFIX	×	$\surd$	$\surd$
SMEAR	$\surd$	$\surd$	$\surd$
SOURCE	$\surd$	$\surd$	×
CORRELATE	×	×	$\surd$
ANALYSE	×	×	$\surd$

Table 1.1: Existing UKQCD codes.

As the codes have been written by many people there is much duplication, with diverse styles of design and coding used. As algorithms have progressed, codes have evolved to form a sprawling, tangled maze. Formal documentation has been extremely limited [84] and is in part responsible for the duplication and diversity.

1.14 Conclusions: the new software suite

Given the need to run lattice QCD codes on the T3D and other platforms, the creation of a *new* set of codes is clearly required. These have to adhere to the following principles:

- The code should be modular in order to minimise redundancy between applications and incorporate new physics in as short a time as possible.
- The requirement for both message-passing and data-parallel programming models implies that the design should be kept independent of programming language as far as possible.
- To improve the learning curve for new personnel all codes should be accompanied by complete and clearly-written documentation. The codes should have a common user interface requiring *no* specialist programming knowledge.
- The languages to be used for the MPP codes are
    - **Data parallel:** $\subset$HPF with testing performed in CM Fortran on a TMC CM-200.
    - **Message passing:** Fortran 77 with PARMACS. The PARMACS layer should be isolated for easy conversion to PVM, MPI *etc.*
    - **Shared memory/single processor :** Fortran 77. This is the same as above with PARMACS calls replaced by local boundary processing.
- The kernel of codes for benchmarking should be able to be completed in nine months (assuming two people working full-time) including documentation, testing and operating procedures.

These criteria are by no means impossible to fulfil... as long as a sensible method of software engineering commensurate with available resources[8] is used. As Loken says [85]

> ‘The real need in software engineering is not for a set of tools or languages. It is rather for an approach to understanding the software problem and developing the optimum solution based on the best available technology.’

In the next chapter we expound a practical solution to this problem.

---

[8] Human, time, financial and equipment.

Chapter 2

Design and implementation of the MPP software suite: global issues

In the previous chapter we introduced the need for a new set of lattice QCD codes. The 'waterfall' method of software engineering, as described briefly in section 2.1, is the canonical software development model for medium to large scale projects and is used to design the MPP codes.

We consider the global issues, affecting the whole MPP project, in section 2.2 which provides an example of the waterfall method in use. The user interface to the suite is discussed in section 2.3, of vital importance to the successful operation of the suite to produce physics results.

2.1 The waterfall method of software engineering

2.1.1 Introduction

> *'Software engineering is the science and art of specifying, designing, implementing and evolving — with economy, timeliness and elegance — programs, documentation and operating procedures whereby computers can be made useful to man.'*
>
> John A. McDermid
> Software Engineer's Reference Book

There are many different software development models described in standard texts [86, 87, 88]; *e.g.* prototyping, waterfall, exploration, formal transformations and reassembly. Of these we will discuss only prototyping and the waterfall method; the others are either irrelevant or inappropriate.

'Prototyping' is mainly used to establish the requirements of a system through construction of a prototype code skeleton. The requirements identified by this process can then be used as the starting point for a more structured method *e.g.* waterfall. The use of prototypes in physics packages is common for new physics, where different algorithms need to be evaluated, but of little relevance to the MPP codes.

The 'waterfall' method of software engineering [89], the canonical method, is highly structured while at the same time retaining sufficient flexibility to be useful for all sizes of project and design team. Documentation is intrinsic to the method, providing information sharing between team members and a permanent record of both concepts and design details. There are several distinct phases of engineering, as shown in figure 2.1, which can be considered separately even though they normally overlap in practice. This method is well suited to academic use as it can be implemented without any special tools[1]: a

---

[1] CASE (computer aided software engineering) packages can be used to increase productivity

drawing package and word-processor are sufficient for the design phases.

As software engineering methods are uncommon in computational physics research we present a brief outline of the waterfall method in the following sections. For more details and background refer to the references mentioned above.

2.1.2 Requirements analysis and specification

The first stage in a project is to ascertain the users' requirements, often through a set of brainstorming sessions followed by a rationalisation to formulate achievable elements. The rationale behind concepts should be included; it is particularly useful in later stages of design and implementation and should be documented with the requirements in the 'Requirements specification document'. This record contains all functionality provided by the system and any constraints thereon.

2.1.3 Standards

As more than one person will be working on the MPP project we need a consistent procedure for writing code and documentation. The standards document covers everything from identifier naming to versions of compilers to be used, but should be kept as clear and simple as possible.

2.1.4 Design principles

Design is performed in a top-down fashion beginning with the splitting of the system into major sub-systems, the physics application codes. Once major applications have been identified they can be further divided into functional modules and finally units. If the design is documented at all stages with the

---

but are expensive and non-essential.

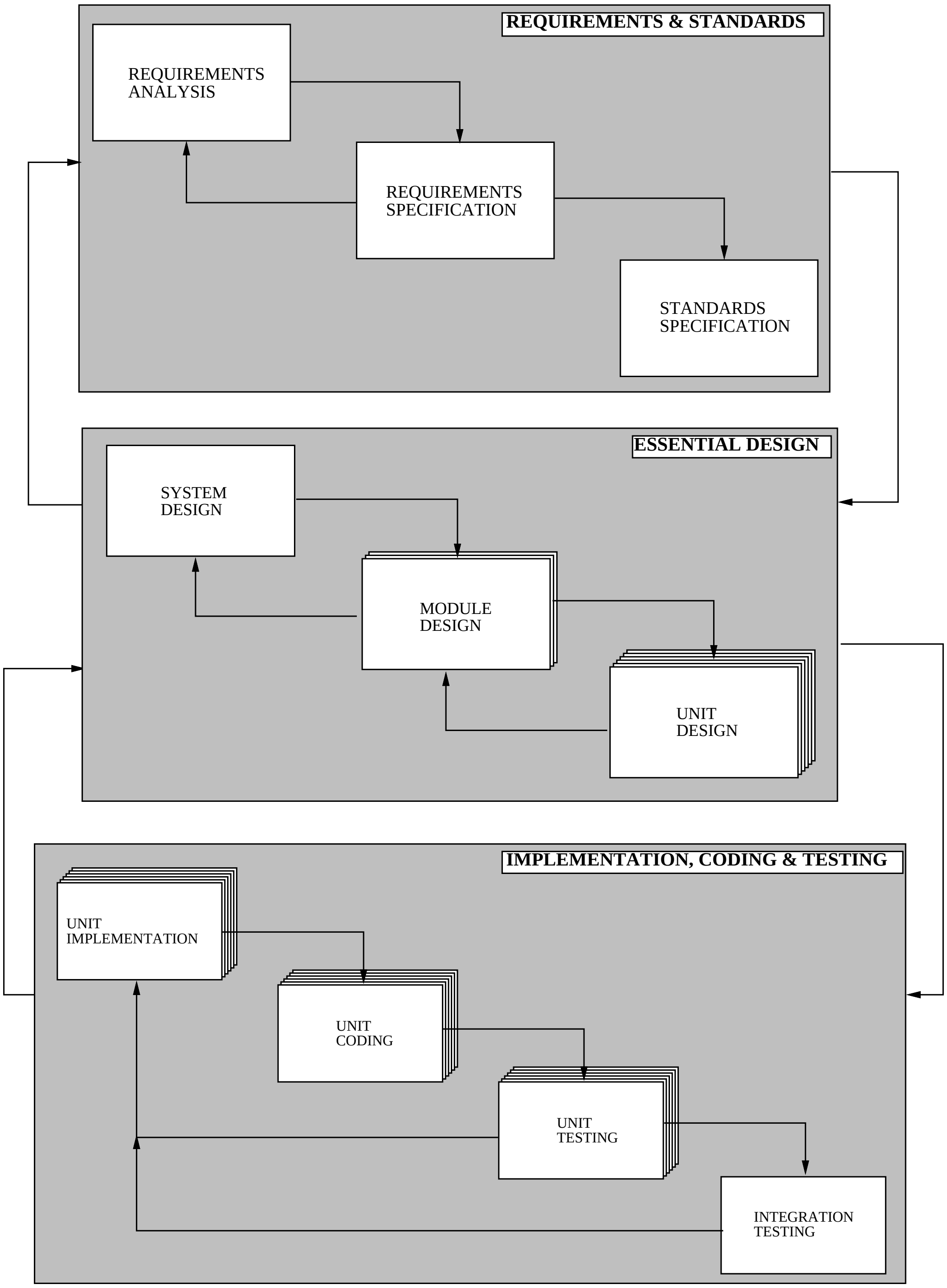


Figure 2.1: The waterfall method of software engineering.

interfaces between applications and modules specified in detail, people can work on different areas at the same time.

Documentation of the design is achieved through layered dataflow diagrams, as described in [88], with textual comments where needed. Examples are given in section 2.2.4. As time is limited, documentation of the lower levels of design, units within a module, should be kept as brief as possible. An example is shown in section 4.2.

Since the MPP codes will be implemented in two versions, data-parallel and message-passing, the design is to be kept free of implementation details as far as possible, concentrating on the essential physics and algorithms.

2.1.5 Implementation, coding and testing the design

To obtain a working package from our design we must implement it in both data-parallel and message-passing models. It is vital to document problems with different versions of the supplied compilers and libraries and iterate design stages if necessary to cater for these differences.

The testing of the code should focus on the physics required. Lower level tests, such as validating the operation of the maths routines, are useful before attempting to test a large complicated module, *e.g.* the solver.

2.1.6 Operational issues

In order for the MPP codes to be useful, they must be easy to operate in the real environment of the targeted MPP machine. This environment incorporates the data storage, batch queuing system and user interface.

Although the data storage and queuing systems will vary across platforms, the user interface can, and should, be kept simple and consistent for all of the MPP applications. This interface includes file formats for long-term storage, which should be rigidly defined and documented.

2.2 Global project engineering issues

2.2.1 Introduction

In this section we consider engineering issues carried through the entire MPP project. Standards used to record, implement and test all designs are detailed in section 2.2.2 followed by requirements on the functionality of the communication and file systems with necessary constraints in section 2.2.3.

The MPP system is split into major subsystems in section 2.2.4 and the principles guiding the design and implementation of all subsystems are brought out in section 2.2.5.

The full design and implementation documentation set, [90, 91, 92, 93], is far too large (over 1000 pages) to be discussed in detail in this thesis. We therefore present only the important features of the design and implementation.

2.2.2 Standards[2]

We have two main types of code: those running on an MPP machine, the production physics applications, and those running on a workstation, the analysis package and the user interface. These will be considered separately.

2.2.2.1 MPP codes

The codes running on the MPP platforms are all written in a Fortran dialect, either Fortran 77 with PARMACS for message-passing or $\subset$HPF for data-parallel. The motivation for using these languages is discussed in section 1.12.

---

[2] The format of the actual standards document is more formal and subdivided than that given here, but less appropriate for a PhD thesis: the full standards document is [90, MPP-GEN-0003].

2.2.2.2 Workstation codes

The codes running on the workstations are all written in ANSI C. The departure from Fortran is taken because these codes will need to handle files, strings and memory more often than numerical data. The 'C' language is well suited to these tasks and offers a seamless interface with the UNIX operating system where necessary.

The use of *YACC* and *LEX*[3] when parsing files is not allowed. Although these extensions to C enable shorter file handling code to be written, therefore less prone to errors, they reduce the possibility that the codes can be extended easily, one of the major requirements of the workstation codes.

2.2.2.3 Fortran 77 standard and pre-processing

The message passing MPP codes are implemented in Fortran 77 with PARMACS. This precludes the use of any of the standard military extensions to the language [94], such as 'ENDDO' and long identifiers. Since short identifiers lead to incomprehensible code we make use of a pre-processor (written in C by Stephen Booth) to compress long identifiers to the standard length. Any compilers which allow long identifiers will not need this stage. The C pre-processor 'CPP' is used for all source files, and any of its features may be used. Header files should only include other header files if absolutely necessary in order to keep the structure straightforward.

Implicit typing for variables should not be used. All variables should be declared explicitly, with 'IMPLICIT NONE' or its analogue used. As not all compilers support the use of 'IMPLICIT NONE', a header file should be included in all routines to mimic this operation as closely as possible.

---

[3] LEX is a lexical analyser and YACC a grammar parser. They are standard packages, often used to parse text files.

2.2.2.4 Identifiers

Identifiers can be up to 31 characters long, and this available length should be used to construct meaningful names. The standard extended character set (alphanumeric plus underscore) is used with the following restrictions.

- ☐ Constants, as defined by the Fortran 'PARAMETER' statement, have a leading capital letter. All other characters are lower case *e.g.* 'Pi_by_2'.
- ☐ Variable and subprogram names are entirely lower case *e.g.* 'write_2column_format'.
- ☐ Language elements are written in capitals *e.g.* 'SUBROUTINE'.

As Fortran is not case sensitive, care must be taken not to use the same name for a variable and constant in the same routine *e.g.* 'Colour' and 'colour'. The compiler should pick up most instances of this, as their usage would be different, and multiple definitions are not allowed.

2.2.2.5 Comments and revision control

All source files should have a comment header block at the beginning containing the name of the source file, its purpose, the author(s), documentation references and revision information. All subprograms should have a comment header block summarising the algorithm and parameters, and if possible a reference to the documentation for further information. The comments within the code should refer only to algorithm stages defined in the subprogram header and any subtle implementation points, care should be taken not to use unnecessary comments which would confuse the code.

2.2.2.6 Numerical precision

Real numbers should be defined as being of either 'Fpoint' (single-precision) or 'Dpoint' (double-precision); two macros defined in a header file. This enables us to switch precision simply by redefining the macros. Macros are also provided to convert to required types, ‘Ftype()’ and ‘Dtype()’, and evaluate their storage requirements in bytes, ‘Fsize’ and ‘Dsize’.

Fortran 77 does not allow double-precision complex numbers, so all complex numbers in the message-passing codes must use two-component ‘Fpoint’ or ‘Dpoint’ arrays. This restriction does not exist in HPF, and the macros ‘Cfpoint’, ‘Cdpoint’, ‘Cftype’, ‘Cdtype’, ‘Cfsize’ and ‘Cdsize’ can be used.

2.2.2.7 Array indexing

Although Fortran 77 arrays are indexed from $1 \dots N$ by default, this can be changed. C does not allow this flexibility; arrays must be indexed from 0 to $N-1$. For this reason we also force Fortran to adopt this rule: all array indexing starts from zero.

2.2.3 Requirements[4]

There are several requirements on all of the MPP codes which must be incorporated from the beginning. These can be split into distinct areas.

1. **Accommodation of lattice in memory**

   We want to be able to accommodate the whole target lattice size in memory at the same time. This approach simplifies the code design and implementation, reducing the need for highly efficient I/O routines to swap lattice segments.

2. **File system**

   As discussed in section 1.12.5, parallel file access will be platform specific. However we can identify a number of common requirements for a file system.

---

[4] As with the standards section, these requirements are in a slightly less formal style than would be used in practice. The actual requirements document is [90, MPP-GEN-0002].

Only the large datasets, *e.g.* gauge configurations and quark propagators, will be stored in a binary format to save space. All other files will use a text format which, although less economical on space, is easy to read without using special utilities and avoids problems with floating point format and byte ordering. The text files can provide information to enable the physics application to sort out the byte order and float format of the large binary datasets[5].

Each binary dataset will have an accompanying textual information file containing all information relevant to the history, creation and validation of the dataset. These files will be mainly read and written by the user interface and must therefore be in a standardised form. The storage of data in the binary datasets must be arranged so that the data can be read in both 4-D and 3-D formats for timeslicing. Any data files ported between platforms must be 'flattened', converted from a format containing machine-specific parallel distribution information to a standard serial format accessible from any processor topology.

The physics application codes must be kept as small as possible in order to work on large datasets. One way of achieving this is to ensure that all files read/written by an application are in a fixed format, requiring no validation. The user interface can again be used to automate this process.

3. **Communications**

   Communications are only of relevance in the message passing model. We must test for failures whenever possible; parallel machines are complex and cannot be assumed to operate without error. Since we cannot recover from communications errors in most circumstances, an error message should be generated and program flow aborted.

---

[5] An example of this could be to use the value of the gauge configuration checksum to establish the byte order, and the plaquette average to establish the floating point format.

4. **Timing and other measurement**

   Timing should be provided in the form of a single routine 'TIMER' conforming to that used in the GENESIS benchmarking suite [95]. Top-level components, *e.g.* a compound update or single solver inversion, should have built-in timing; all other timing should be switchable.

   Floating point operation counts (flopcounts) should be built into all code, with the same metric used as [96] and shown in table 2.1. From these numbers a performance rating can be measured for the timed sections in units of Gflop/s.

Operation	**Count (flop)**
Real add, subtract, multiply	1
Real divide, square root	4
Exp, Sine *etc.*	8
Complex add, subtract	2
Complex multiply	6

Table 2.1: Floating point operation metric.

   It is useful to have a means for measuring a high-water mark for the memory usage. This only needs to be calculated once for each of the data-parallel and message-passing versions and does not always need to be included in the code; some compilers supply this capability. If we do need to perform this operation in the code it is straightforward.

5. **Lattice size**

   The lattice size must be a factor of two in all dimensions in order to implement red-black precondititioning.

6. **Random number generators**

   We require a system for generating pseudorandom numbers uniformly distributed between 0 and 1 for both lattice arrays and scalar variables. The actual random number generator (RNG) used should be easily changeable as different RNGs are suitable for different applications. The RNG used in production should satisfy all of the standard tests discussed in [97, 98, 99, 100]. The design of random number generators is considered in section 2.2.4.

7. **Random gauge transformation**

   An excellent method for testing code is to construct gauge invariant quantities, *e.g.* the plaquette average, and then perform a gauge transformation on the fields with a set of random SU(3) matrices. If the same answer is obtained before and after the transformation, we can be fairly confident that that section of the code is working. We therefore need a module capable of implementing both local and global gauge transformations.

8. **Error logging**

   Error messages are to be written to a logfile with as much information as possible present to facilitate recovery and debugging. No other messages are to be written to this logfile so that errors can be detected quickly.

   The system must indicate that it has terminated with or without an error condition being generated, *e.g.* by a non-zero exit code, so that the parent process can act accordingly.

2.2.4 System context and design

Before discussing the design of the system, we must consider its boundaries. These are shown in figure 2.2. As can be seen, the codes only need to interact with the user and data stores.

The physics application codes are identified by main functional blocks. They interact as shown in figure 2.3. This is an example of a 'dataflow diagram' such as is used later to represent the essential design. Ovals are the main processes, parallel lines represent datastores and arrows show data flowing between data stores and processes.

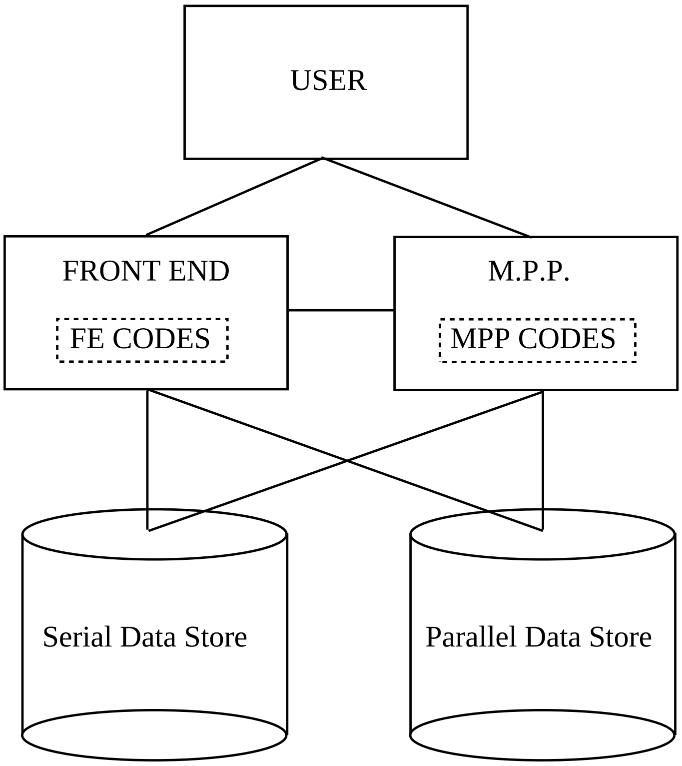


Figure 2.2: View of the MPP codes system boundaries.

Note that code to analyse gauge configurations, *e.g.* to measure glueball masses, is expected to constitute a separate application or applications. This may not be the case in reality if relative computational and I/O speeds promote in-line computation requiring GAUGE or HMC to be modified.

2.2.5 Design and implementation issues

The main features in the design of the MPP codes are the need for high speed and as large a lattice as possible. These principles require us to keep memory usage to a minimum, to maximise the available lattice size, and keep the number of floating point operations to a minimum, maximising the speed. Of course, life is rarely simple: in order to reduce computation we often need more workspace; compromise is clearly required.

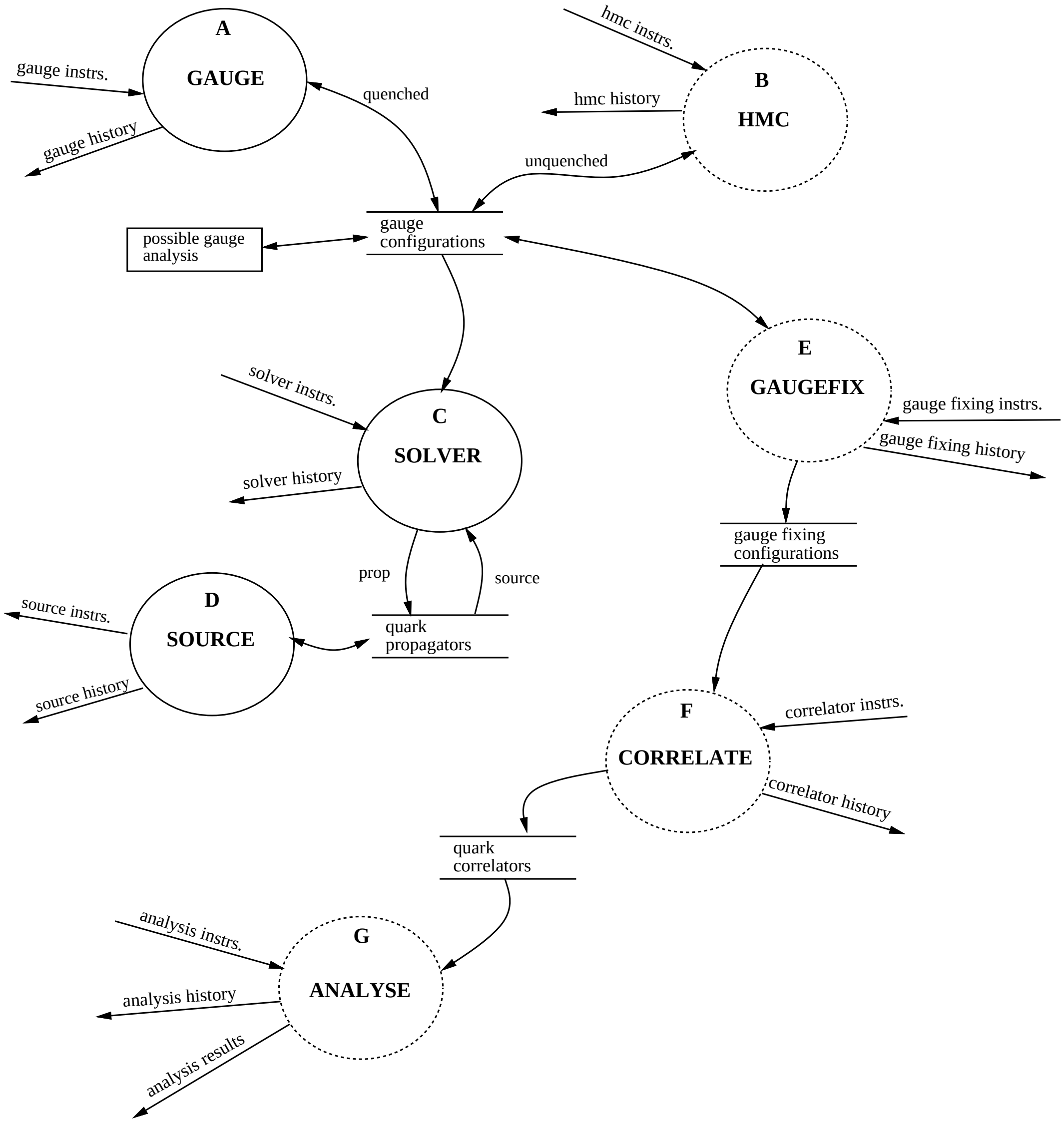


Figure 2.3: The physics applications comprising the MPP system and their interrelations; dotted applications are not considered in this thesis. The user interface ECU and treatment of messages to the user have been omitted.

2.2.5.1 Minimising memory usage and controlling allocation

There are several ways in which memory usage can be minimised.

- By grouping together into one unit only those operations which absolutely need to be together we keep the unit size small. If a single unit is then implemented in a single file, only that code needs to be incorporated into the final application; no wastage occurs.
- By delegating all complicated file handling to the ECU application, the control utility, we keep the code space required for file handling to a minimum.
- If we can modify an algorithm to only require temporary workspace for a single matrix element or single row, we keep heap[6] memory usage, the most crucial area, to a minimum. We must, however, balance this against the requirement for high speed.
- Stack[7] space is not particularly important in an MPP platform as it is a tiny portion of the available memory. However, we wish to minimise the depth to which subroutines are nested in order to reduce the overhead from calling and returning from subroutines; this requirement must be balanced against the need for a large number of small units to provide flexibility and a reusable toolkit of operations. If in doubt, keep the unit size small, code can be optimised for a particular platform later.

Since we have several people working on the codes at the same time, usually on separate modules, we need to be careful how workspace memory is allocated. A simple heap-based model is assumed; each routine reserves heap memory as required and frees it on exiting the routine. All of the compilers we have

---

[6] 'Heap' memory is that used for allocation of temporary (or automatic) variables within a routine. The memory is usually freed when the routine terminates.

[7] 'Stack' memory is that used for passing parameters between subroutines and the return values from functions.

encountered have this model of memory management, which makes it easy for modules to be implemented independently.

An alternative model which might possibly arise is that of static memory allocation. In this model, the memory is not freed when a routine finishes resulting in the program running out of memory and crashing. Provision has been made for this outcome: at the start of the program execution a large block of memory is reserved and made common to all routines. An internal heap is then set up to allocate and free the space as required. This method requires a significant amount of work to implement as allocation/deallocation calls need to be made explicitly. For this reason, and also because we hope never to encounter this type of compiler, the memory management calls have not been inserted although the routines to implement them have been supplied. This model of memory management cannot be used in ⊂HPF as we need to know information about the distribution of arrays over the platform.

Common blocks and global variables

Use of globally-accessed memory space can be useful in reducing the amount of memory used for both heap and stack variables. However, the code then becomes far more inter-dependent and less modular. As we want to construct a range of applications from a common library of modules we need a modular design and therefore avoid common blocks and global variables as far as possible outside a particular module.

2.2.5.2 Platform specific variations

For several areas of the MPP codes, the implementation will depend on the particular platform; these areas can be identified as parallel file I/O, timing, communications and random number generation. In order to maximise portability we have to isolate the platform dependent sections. This can be easily achieved as shown in chapter 3.

2.2.5.3 Data distribution in HPF

In $\subset$HPF data distribution is straightforward. The lattice indices are distributed over the abstract processors of the platform, while all matrix indices *e.g.* spin or colour are local to a processor. An example is shown in section 2.2.5.5 where the distribution of gauge and fermion fields is specified.

2.2.5.4 Data distribution in Message passing

For the message-passing mode of operations we assume a regular 4-D problem distributed over a 4-D grid of processors, each processor being responsible for a 4-D sub-lattice. We make the following restrictions on the communications and distribution system:

- ☐ Only nearest-neighbour and global-sum communications are assumed. These are all we need to implement a local grid-based problem.
- ☐ The lattice is assumed to be larger than the grid so that we have no idle processors. The sub-lattices may be different sizes on different processors as long as neighbouring processors have the same size for their common boundaries.
- ☐ To implement red-black preconditioning we need the local sub-lattice to be at least two sites wide in the fastest changing index. We address the local sites using a parity-site scheme.
- ☐ All processors are assumed to be executing the same program with different data, the SPMD model. This allows the communications routines to be called by all processors in the same order at approximately the same time.
- ☐ To simplify the number of parameters characterising the distribution we require that at least two of the local boundary dimensions are even, so that both parities of the local lattice have equal lengths. This is a much simpler situation to handle than if the two parities had different lengths;

we would require separate parameters to describe each parity instead of one for both parities.

☐ One of the local even dimensions should be the fastest changing index, *e.g.* the x-direction. This simplifies parallel file access.

The MPP codes should arrange the distribution automatically to satisfy these requirements.

Communications

To shift arrays one lattice site in any direction we use *neighbour tables*, *gather-scatter tables* and *boundary tables.*

A neighbour table on a particular processor contains the processor number of adjoining processors so we know where to send or receive data. An example 2-D array of processors, with periodic boundary conditions, is shown in figure 2.4.

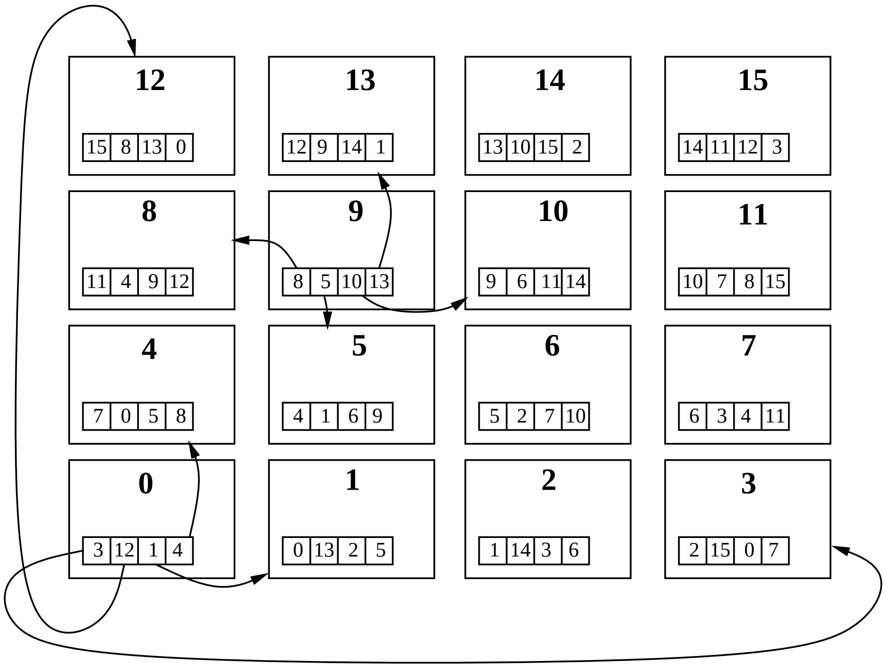


Figure 2.4: An example of neighbour tables in a 2-D problem. Periodic boundary conditions are implemented here.

When shifting data distributed across processors the boundary sites need to be moved to an adjoining processor. This can be implemented by 'scattering' the local data into workspace with a 'tail' region, communicating the tail to the neighbouring processor in direction μ (and receiving the tail from the neighbouring processor in direction $-\mu$), then copying the data from the received boundary into the body of the workspace region. The workspace area now contains the shifted data. The scatter into the workspace moves data which remains on the processor to the correct array locations, and data to be moved off-processor into the tail region using the gather-scatter tables. Elements composing the boundary can be found using the boundary tables. An example of these tables is shown for a single processor of a 2-D problem in figure 2.5.

Compressed gather-scatter tables

Conventional gather-scatter tables will contain long sequences of consecutive numbers which can be encoded as start and stop indices. The gather-scatter operations can then be implemented as a pair of nested loops; the outer loop reads the table to find the loop limits used by the inner loops. Both methods need to be provided for optimal performance on different architectures and shift directions.

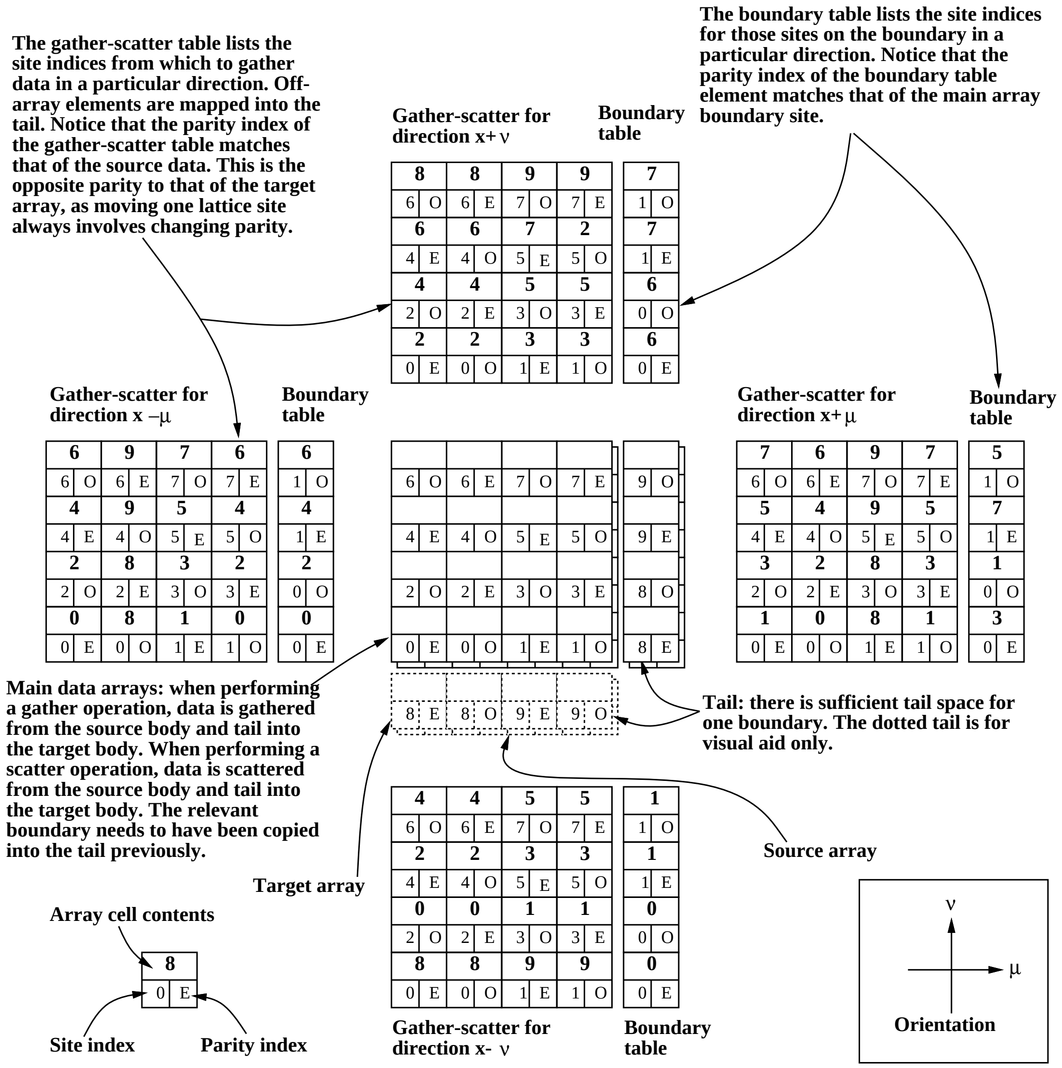
Single processor code

When the processor grid is only one processor wide in any direction we need the ability to ignore the tail and implement local periodic boundary conditions directly using the gather-scatter tables.

2.2.5.5 Data types

The main data types used in the codes are gauge fields and fermion (4-spinor) fields. To see how to implement them we need to consider all indices required.

Gauge fields



The gather-scatter table lists the site indices from which to gather data in a particular direction. Off-array elements are mapped into the tail. Notice that the parity index of the gather-scatter table matches that of the source data. This is the opposite parity to that of the target array, as moving one lattice site always involves changing parity.

The boundary table lists the site indices for those sites on the boundary in a particular direction. Notice that the parity index of the boundary table element matches that of the main array boundary site.

Main data arrays: when performing a gather operation, data is gathered from the source body and tail into the target body. When performing a scatter operation, data is scattered from the source body and tail into the target body. The relevant boundary needs to have been copied into the tail previously.

Tail: there is sufficient tail space for one boundary. The dotted tail is for visual aid only.

Figure 2.5: Example gather-scatter and boundary tables in two dimensions.

Gauge fields can be written with all indices explicit as $U_{\mu ij}(x)$, and are complex. Since the staple sum and other operations need to be red-black decomposed we need to split the fields into two parities, even and odd. The gauge configuration can then be written in HPF as

```
      Cfpoint, DIMENSION (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1) ::
     $  gauge_xevn, gauge_yevn, gauge_zevn, gauge_tevn,
     $  gauge_xodd, gauge_yodd, gauge_zodd, gauge_todd
!HPF$ DISTRIBUTE (*,*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS ::
!HPF$$  gauge_xevn, gauge_yevn, gauge_zevn, gauge_tevn,
!HPF$$  gauge_xodd, gauge_yodd, gauge_zodd, gauge_todd
```

where `Ncolour` is the number of colour components, *i.e.* 3, and `Nxby2`, `Ny`, `Nz` and `Nt` specify the size of the lattice. The mapping, `QCDPROCS`, onto the physical processors must be made for all arrays; for this reason it is stored in a header file. An example of such a mapping is

```
!HPF$ PROCESSORS QCDPROCS (4,4,4,8)
```

which distributes the lattice sizes over the $4 \times 4 \times 4 \times 8 = 256$ processors. We use a different array for each direction and parity because, as discussed in section D.2.2, Connection Machine Fortran does not let us pass a portion of an array as an argument to a subroutine.

The message-passing version of the gauge field declaration is in many ways simpler. The entire set of gauge fields is declared as

```
CFTRANS gauge :site :I :I :I : :
      Fpoint gauge(0:Max_array-1,
     $  0:Ncomplex-1,0:Ncolour-1,0:Ncolour-1,0:Npar-1, 0:Ndim-1)
```

where the indices are in order; site (the x-index moves fastest, followed by y, z then finally t), complex (real then imaginary), colour-row, colour-column, parity (even then odd) and direction. The `CFTRANS` line allows our pre-processor, FTRANS which is also used to compress long identifier names, to rearrange the indices as needed for optimisation. By default, the site index is always taken to be fastest changing for efficient vectorisation. On cache-based machines it can be more efficient to make the site index move slowest. The indices are divided into three types

1. The site index, as indicated by '`:site`'.
2. Indices that may be re-ordered, as indicated by '`:I`'. These are always moved together so that the site index is either faster or slower moving than all of them.
3. Indices that may not be re-ordered, as indicated by '`:`'. These are usually used for passing sections of arrays to subroutines, *e.g.* a single parity or direction of a gauge field.

The site index for the gauge fields incorporates a tail for communications.

Notice that in HPF we must hard-wire the lattice size into all of the code and re-build the executable images whenever the size changes. In message passing we can be more flexible, we specify a maximum size, `Max_array`, at build-time and as long as that accommodates the local run-time lattice size we have no problems. The reason for this inflexibility in HPF is the `CSHIFT` operation, which shifts array elements cyclically. If we declared our arrays to be larger than needed, junk at the unused end of the array would be shifted into the space we do use during a `CSHIFT`. The use of gather-scatter and boundary tables in message passing obviates this problem.

Spinor fields

The fermion 4-spinor fields are declared in much the same way. In HPF we have

```
      Cfpoint, DIMENSION (0:Ncolour-1,0:Nspin4-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1) ::
     $  psi_evn, psi_odd
!HPF$ DISTRIBUTE (*,*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS ::
!HPF$$  psi_evn, psi_odd
```

where `Nspin4` is defined to be '4' as expected. Because the main 4-spinors are never used for communication in the message passing codes we do not need a tail space. When we do need to communicate spinors we either use 2-spinors with tails (for the `delta_term` routine) or temporary 4-spinors with tails (for the `dslash` routine as discussed in section 5.2). The main 4-spinors are therefore

```
CFTRANS psi :site :I :I :I :
      Fpoint psi (0:Max_body-1,
     $  0:Ncomplex-1,0:Ncolour-1,0:Nspin4-1,0:Npar-1)
```

All other data types are introduced as needed in the remainder of this chapter.

2.3 The user interface: the ECU application

2.3.1 Introduction

The user interface is one of the most important from the design point of view because often novice users need to use, extend and modify the package in as short a time as possible. The design therefore needs to be kept simple while at the same time remaining flexible. The complexity of the system introduces a large number of different types of file used for information storage; some for communicating with the user, some for communicating with the applications and others for long-term data storage. There will be a large number of data elements common to several file types; this data needs to be easily accessible in a variety of formats with strict validation built in for the package to operate successfully.

2.3.2 Design and implementation

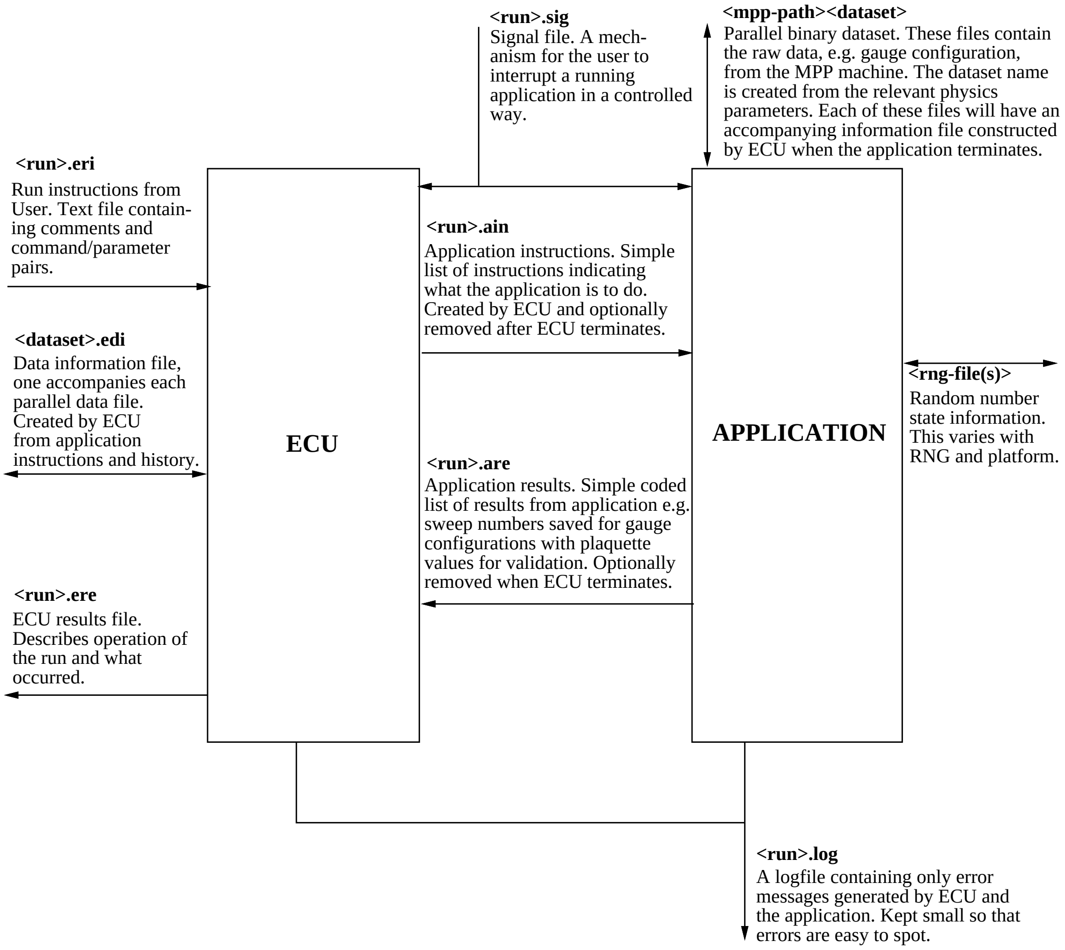
The ECU application is written in ANSI C for portability and strong memory, string and file handling abilities. As inexperienced programmers need to work on the code we avoid any complicated language extensions such as LEX, YACC and C++.

File system: system context

The whole file system is shown in figure 2.6. The detailed file formats are defined in [90, 93] if required.

The files requiring most parsing are those read/written by the user. To keep the general format under control we apply the following restrictions.

- ☐ The files are textual with comments (lines beginning with '`#`') and command/parameter pairs.



<run>.sig
Signal file. A mechanism for the user to interrupt a running application in a controlled way.

<mpp-path><dataset>
Parallel binary dataset. These files contain the raw data, e.g. gauge configuration, from the MPP machine. The dataset name is created from the relevant physics parameters. Each of these files will have an accompanying information file constructed by ECU when the application terminates.

<run>.eri
Run instructions from User. Text file containing comments and command/parameter pairs.

<run>.ain
Application instructions. Simple list of instructions indicating what the application is to do. Created by ECU and optionally removed after ECU terminates.

<dataset>.edi
Data information file, one accompanies each parallel data file. Created by ECU from application instructions and history.

<rng-file(s)>
Random number state information. This varies with RNG and platform.

<run>.are
Application results. Simple coded list of results from application e.g. sweep numbers saved for gauge configurations with plaquette values for validation. Optionally removed when ECU terminates.

<run>.ere
ECU results file. Describes operation of the run and what occurred.

<run>.log
A logfile containing only error messages generated by ECU and the application. Kept small so that errors are easy to spot.

Figure 2.6: The MPP file system.

- ☐ There can only be one command with following parameter per line. The command must be separated from the parameter by one or more tabs or spaces. Tabs and spaces leading the command or trailing the parameter are ignored.
- ☐ Blank lines are legal and are ignored.
- ☐ Upper and lower cases are considered identical except within quoted strings or paths.
- ☐ The maximum length of a command is 24 characters. The maximum length of a line is 80 characters.
- ☐ Information from other applications can be included, *e.g.* a propagator file will include information about the gauge configuration used to generate it and the source, and is represented as

  ```
  <included-app> {
  <included-command-1> <included-parameter-1>
  <included-command-2> <included-parameter-2>
  .
  .
  }
  ```

  These included blocks are parsed to check that `<included-app>` is legal for the file type and that the commands and parameters are valid. These blocks cannot be nested.

Overall design of ECU

The design of the ECU application is shown in figure 2.7. Notice that ECU runs in two modes, before and after the physics application has run.

Routines for reading user text files, one for each application, all make use of a common routine to parse the files and process the data. The advantages of this are that the files are constrained to a common format —entirely non-overlapping routines always result in subtle differences in format— and by having a common routine we are forced to use some sort of list of legal elements

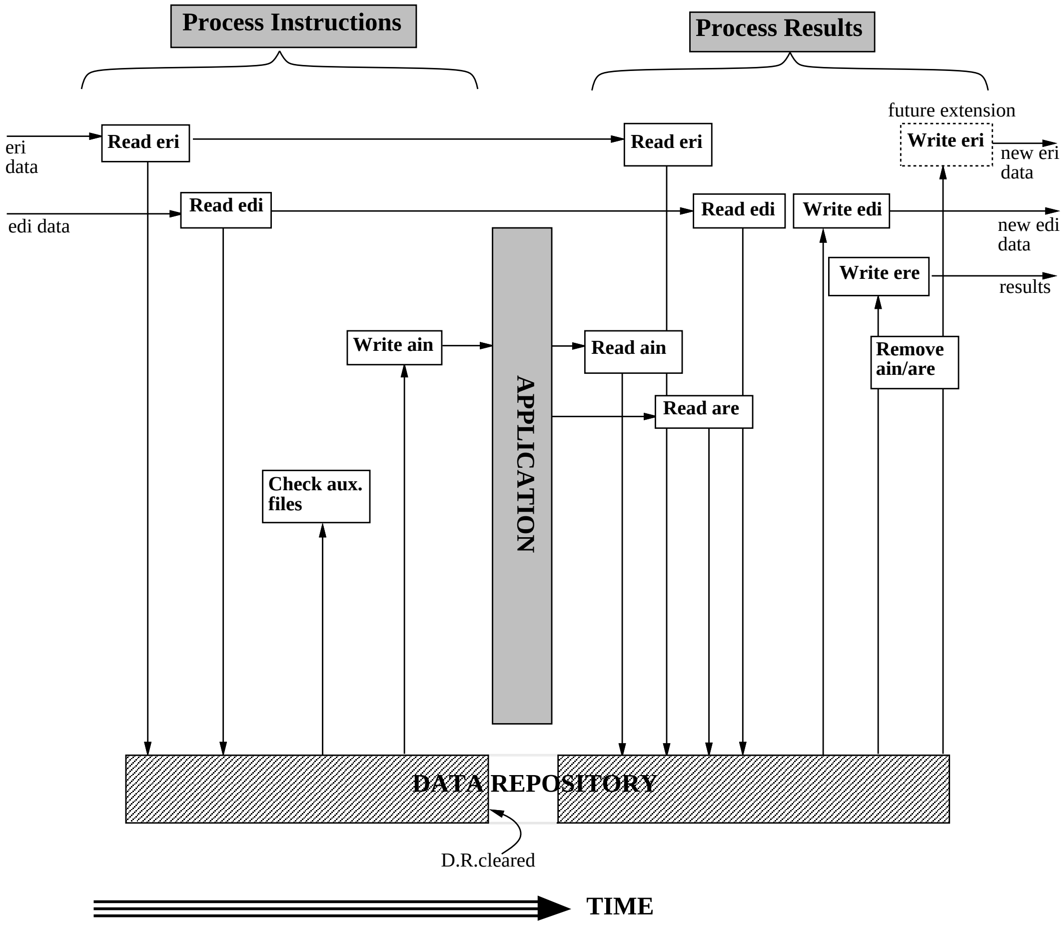


Figure 2.7: Overall design of ECU.

in the files, easy to update and understand. Even when we do not have a common routine for file I/O with different applications we still use lists of items to reduce code complexity and improve understanding making extensions easier to implement.

The data repository

At the heart of ECU is the data repository. This is a single module which provides a central storage area for all data read from or written to files. Limit checking, format conversion, default selection and optional arguments are all implemented for scalar or vector quantities in a highly consistent way. In addition, automatic consistency checking is built in for when the data structure is extended or modified.

The data is stored in an array of records, each with the following structure.

`name`	A lowercase word (optional '_'s) describing the data element, *e.g.* `lattice_x` the lattice extent in the x-direction.
`type`	An enumerated constant defining the data type. Legal types are `string`, `path` (a string with enforced trailing `/`), `int`, `float`, `choice` (different choices *e.g.* `point` and `loaded` for `src_type`), `tsliced_int` and `tsliced_float` (time-sliced vector quantities), `sctsliced_int` and `sctsliced_float` (spin, colour and time-sliced indexed vector quantities).
`index`	An enumerated type used for all references outside of this module. By including this number in the list we can check that all elements are in the correct order on first using the DR.
`min`	
`max`	String representations of the lower and upper limits on a quantity.
`deflt`	The default value to use if we want to read the value of an element without first having inserted some data.

`choices`	If `type` is set to `choice` then this element of the record points to a list of strings that may be used for the field. For example, boolean variables are represented as a choice between '`no`' and '`yes`'.
`depend`	
`depval`	Some `choice` elements will require different data for the various choices, *e.g.* source type for SOLVER: for a point source we want to know the location (`psrc_x`, `psrc_y`, `psrc_z` and `psrc_t`), whereas for a loaded source we want to know the name (`lsrc_name`) and the time-slices to use (between `tslice_max` and `tslice_min`). So in this example, for `psrc_x` we would have `depend` and `depval` as `src_type` and `point` respectively. This facility saves having unnecessary elements in files read/written by the user, improving clarity.
`format`	A normal C format string specifying precision used and type of output, *e.g.* pion propagator values have a `format` of '`%13.10f`'.
`action`	An enumerated type defining the action to be taken when a value to be inserted into the DR disagrees (or agrees in a few cases) with a previous value. Either an error or warning condition is raised.
`result`	The value of the data element stored as a string. By defining this record element as a '`void *`' we can easily implement scalar or vector quantities.

A single header file containing a list of enumerated indices and prototypes for functions intended for external use provide the only access to this module. The use of enumerated types is ideal for validation as most compilers check these types strictly.

Routines exist for the following

insert We can insert a data value by name or index. Insertion can be forced (even if the values disagree) and specify whether data is coming from the application or user to get the correct format.

read We can optionally clear the result after reading a data element and specify whether the data is going to the application or user.

display It is useful to be able to print out all details about an item in the DR to save time finding the documentation. Higher level routines can print out all options for a file type or application.

reset If needed, all data elements can be reset to empty.

Vector type elements can only be accessed after the relevant DR items `source_spin`, `source_colour` and `time_slice` have been set to the correct index values.

2.3.3 Operation

Use of the ECU application is usually wrapped up in a shell script as there are three main stages.

1. Run ECU in 'before' mode to convert user-style run instructions and previously written data information files to a form easily understood by the application; the application instructions file. Full validation takes place during the conversion. Auxiliary files, binary datasets and RNG state information, are checked for existence if possible; the ability to check for their existence depends on the parallel file system in question.
2. Run the application to generate physics.
3. Run ECU in 'after' mode to convert the application results log into user-readable form and write the necessary data information files to accompany the parallel datasets.

Each production run of an application should be uniquely labelled, this run name is given to ECU and the application as sole input; all filenames are derived from this or the relevant physics parameters.

2.3.4 Operation example: generating a gauge configuration

As an example of the input and output files used, consider the following input file used to create a gauge configuration.

```
# Example run instruction file ‘example.eri’ for GAUGE

# Set the lattice size to be 16^4
lattice_x 16
lattice_y 16
lattice_z 16
lattice_t 16

# Specify the physics parameters used.
beta 6.0

# Specify the starting point and duration of simulation.
# We use a disordered/random start and generate 5 compound
# sweeps through the lattice.
start_type hot
compound_sweeps 5

# Specify algorithmic parameters for a single sweep.
# We use (in order) :-
# Random gauge transform.
# 3 update sweeps, each consisting of ...
#        2 Cabibbo-Marinari updates and
#        2 Over-Relaxed updates
# A reunitarisation.
gauge_transform yes
update_sweeps 3
cm_sweeps 2
or_sweeps 2
reunitarise yes

# Save the plaquette average on every update sweep.
plaquette_interval 1
```

```
# Save the configuration every 5th compound sweep, i.e.
# at the end of the simulation for this run.
checkpoint_interval 5

# Seed for initialisation of the random number generator
# for the hot-start and updates.
rng_seed 38234765

# Specify where to put the parallel data files.
gauge_mpp_path /scratch1/qcd/npstan/

# Specify where to put the data information files.
gauge_fe_path /home2/npstan/example/data/
```

To run the application we must first convert the run instructions by issuing the command

```
ecu -agauge -rexample -mbefore
```

which produces the file '`example.ain`' for the GAUGE application. If we then run GAUGE giving it the run name '`example`' the gauge configurations will be generated. We expect the configuration number generated to be $5 \times 3 \times 2 \times 2 = 60$ from the algorithmic parameters and number of compound sweeps generated. At the end of the run the parallel data files are left in directory '`/scratch1/qcd/npstan/`' as required. A file is generated for each time-slice, i.e.

```
Q60U000060T00  Q60U000060T04  Q60U000060T08  Q60U000060T12
Q60U000060T01  Q60U000060T05  Q60U000060T09  Q60U000060T13
Q60U000060T02  Q60U000060T06  Q60U000060T10  Q60U000060T14
Q60U000060T03  Q60U000060T07  Q60U000060T11  Q60U000060T15
```

and possible the lattice RNG data if using the data-parallel code. To tidy up after the application we issue the command

```
ecu -agauge -rexample -mafter
```

which analyses the application results file '`example.are`' and produces data information files in directory '`/home2/npstan/example/data/`' as requested. Again there is one file for each time-sliced data file and a random number state file if using the message-passing codes:-

```
Q60U000060.rng     Q60U000060T05.edi  Q60U000060T11.edi
Q60U000060T00.edi  Q60U000060T06.edi  Q60U000060T12.edi
Q60U000060T01.edi  Q60U000060T07.edi  Q60U000060T13.edi
Q60U000060T02.edi  Q60U000060T08.edi  Q60U000060T14.edi
Q60U000060T03.edi  Q60U000060T09.edi  Q60U000060T15.edi
Q60U000060T04.edi  Q60U000060T10.edi
```

The data information files will be of the form

```
lattice_x           16
lattice_y           16
lattice_z           16
lattice_t           16
beta                6.0
update_sweeps       3
cm_sweeps           2
or_sweeps           2
reunitarise         yes
gauge_transform     yes
rng_seed            38234765
swap_row_col        no
gauge_mpp_path      /scratch1/qcd/npstan/
gauge_fe_path       /home2/npstan/example/data/
sweep_number        60
plaquette_real      0.5917322655
plaquette_imag      0.0000462752
gauge_version       1
ecu_version         1
time_slice          0
tplaquette_real     0.5876517477
tplaquette_imag     0.0003444789
gtcsum              42556
```

i.e. a record of the algorithmic and physics parameters with the average plaquette value for the configuration, versions of applications, time-slice data and checksum. A simple record of what has been written is left in file '`example.ere`'

```
Written gauge configuration Q60U000060
```

This could be expanded in the future to include other useful information.

If we then wanted to use this configuration to generate others, the next run would have to have a modified run instructions file, ‘example1.eri’

```
# Example run instruction file ‘example1.eri’ for GAUGE

# Set the lattice size to be 16^4
lattice_x 16
lattice_y 16
lattice_z 16
lattice_t 16

# Specify the physics parameters used.
beta 6.0

# Specify the starting point and duration of simulation.
# We use the previously generated configuration, number 60,
# and perform 100 sweeps through the lattice.
start_type old
start_sweep 60
compound_sweeps 100

# Specify algorithmic parameters for a single sweep.
# We use (in order) :-
# Random gauge transform.
# 3 update sweeps, each consisting of ...
#         2 Cabibbo-Marinari updates and
#         2 Over-Relaxed updates
# A reunitarisation.
gauge_transform yes
update_sweeps 3
cm_sweeps 2
or_sweeps 2
reunitarise yes

# Save the plaquette average on every update sweep.
plaquette_interval 1

# Save the configuration every 20th compound sweep.
# This run should therefore generate 100/20=5 configurations.
checkpoint_interval 20

# Seed for initialisation of the random number generator
```

```
# for the hot-start and updates. The zero means that we
# want to use the saved RNG state information.
rng_seed 0

# Specify where to read/write parallel data files
gauge_mpp_path /scratch1/qcd/npstan/

# Specify where to read/write data information files.
gauge_fe_path /home2/npstan/example/data/

# Indicate that we want full validation of input data
# files. We check the plaquette average for all time-slices,
# and the whole configuration. A checksum for the data
# files is calculated and verified against that written in the
# 'edi' files shown above.
validate_plaquettes yes
validate_tplaquettes yes
validate_gtcsum yes
```

2.3.5 Future extensions

There are several features which could be added to this application to save space and labour. For example ...

- ☐ The ability to create the next instruction file for the next run, taking into account anomalous exit via user signals (implemented through the 'sig' file). This is particularly useful for running GAUGE where endless configurations need to be generated. Automating this stage reduces the load on the user when starting new runs.
- ☐ The ability to compress/decompress data files as needed for a run and convert formats if necessary.

Chapter 3

Design and implementation: modules common to multiple applications

To make the MPP applications easy to implement we use a library of common routines divided into the following areas

☐ Communications

☐ Parallel I/O

☐ Maths

☐ Random numbers

☐ Timing

3.1 Communications

There are two main types of communication used by the MPP codes: local, shifting whole arrays one lattice site in a particular direction, and global, summing a quantity over all lattice sites and processors to give a vector or scalar result. Point-to-point communications are not needed in lattice QCD and will not be considered.

HPF implementation

The HPF implementations of both communication types are relatively straightforward: the global sum is provided by the intrinsic routine '`SUM`', *e.g.*

```
      INTEGER t
      Cfpoint psi (0:Ncolour-1,0:Nspin4-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)
!HPF$ DISTRIBUTE psi (*,*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS
      Cfpoint, DIMENSION tslice_average (0:Nt-1)
!HPF$ ALIGN tslice_average(t) WITH psi(0,0,0,0,0,t)
```

```
C  sum over a time-slice.

      FORALL (t=0:Nt-1)
     $  tslice_average(t) = SUM(psi(:,:,:,:,:,t))
```

calculates a simple time-sliced sum over all other indices.

The local communications in HPF are simple in all directions except the x-direction because of our parity assignments. Consider figure 3.1 showing both the 4-parity used to split the lattice and a 3-parity MOD$(y+z+t)$. Notice that to go from global coordinates $(1,0)$ to $(0,0)$ does not require any communications since the two points are on different parity sub-lattices with the same sub-lattice coordinates. We can extract a general rule from this diagram if we calculate the logical quantity 'decision' where

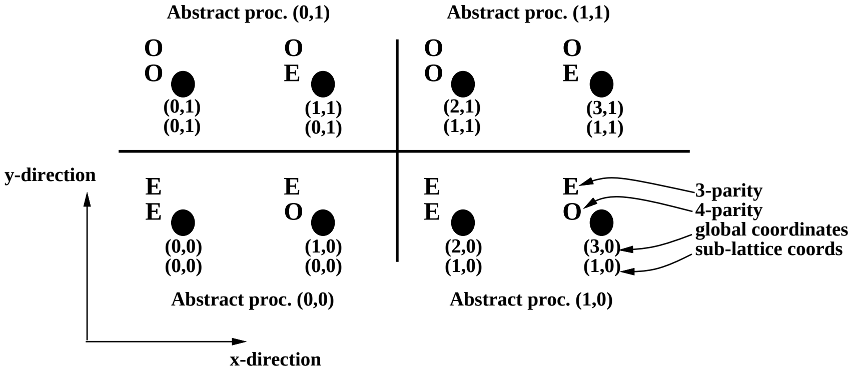


Figure 3.1: Parity assignments for an xy slice through the origin. The grid lines show the abstract processor boundaries.

```
decision = (par .EQ. Even .AND. updown .EQ. Negative)
        .OR.(par .EQ. Odd  .AND. updown .EQ. Positive)
          = D1 .OR. D2
```

where 'par' is the parity of the source array and 'updown' indicates the direction of shifting. The possibilities can be tabulated

par	updown	D1	D2	decision
Even	Positive	0	0	0, Odd
Odd	Positive	0	1	1, Even
Even	Negative	1	0	1, Even
Odd	Negative	0	0	0, Odd

From the 3-parity equivalents of '`decision`' given in this table we can see from the first row that if we wish to shift an even 4-parity array in the positive x-direction we only wish to communicate those elements where the 3-parity is `Odd` (*i.e.* global coordinates $(1, 1)$ or $(3, 1)$).

To implement these 3-parities we use masks set to '`.TRUE.`' on even 3-parity (an arbitrary but crucial convention). Because of a restriction in the CM compiler we use a different mask for each associated data type. For example, with gauge fields we have

```
      LOGICAL gauge_mask (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)
!HPF$ DISTRIBUTE gauge_mask (*,*,BLOCK,BLOCK,BLOCK,BLOCK)
!HPF$$ ONTO QCDPROCS

      CALL setup_gauge_mask (gauge_mask)
```

where the routine '`setup_gauge_mask`' must be called before any communications take place.

To hide all details of the x-direction communications from the user we provide a set of routines, one for each data type, to perform all shifting operations. For example, with gauge fields we would use subroutine '`shift_3by3`' defined as

```
      SUBROUTINE shift_3by3 (gauge_mask,
     $  source, par, dir, updown, dest)

#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "processors.h"
#include "precision.h"
#include "shift.h"

      Cfpoint, DIMENSION (0:Ncolour-1,0:Ncolour-1,
```

```
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)  ::
     $  source, dest
      INTEGER par, dir, updown, shift_dir
      LOGICAL decision
      LOGICAL gauge_mask (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)
!HPF$ DISTRIBUTE (*,*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS ::
!HPF$$ source, dest, gauge_mask

C Convert x,y,z,t into correct indices for shifting arrays.
C X_shift is defined in shift.h to be 3
      shift_dir = dir - X_index + X_shift

C move the gauge fields, no matter which direction.
      dest = CSHIFT (source, SHIFT=updown, DIM=shift_dir)

C check whether to set gauge fields back to what they were
C on certain parities for the x-direction.
      IF (shift_dir .EQ. X_shift) THEN
        decision = (par.EQ.Even_parity .AND. updown.EQ.Negative)
     $    .OR.     (par.EQ.Odd_parity  .AND. updown.EQ.Positive)
        dest = MERGE (source, dest, gauge_mask .NEQV. decision)
      END IF

      RETURN
      END
```

As an example of use consider the following equation fragment

$$\mathrm{res}_o(r) = \mathrm{src}_e(r + \hat{x})$$

which would be implemented as

```
      CALL shift_3by3 (gauge_mask, src, Even_parity,
     $  X_index, Negative, res)
```

where all constants are defined in header files for ease of use. Note that for any other direction than x the parity is ignored as you always change parity.

MP implementation: initialisation

The communications details for MP with PVM are dealt with in Appendix E. In this section we present a portable interface to the MP communications.

Before we can perform any actual communications calls we need to initialise all data structures and establish which processor is running which segment of the lattice. A portable interface is provided to implement this

```
      SUBROUTINE init_comms (grid_size, grid_pos,
     $                        proc_id, boss_proc)

      INTEGER grid_size(0:Ndim-1), grid_pos(0:Ndim-1),
     $        proc_id, boss_proc
```

where the parameters are shown in table 3.1.

`grid_size`	The size of the processor grid in each direction. A size of 8 processors is returned as '8', not '7'.
`grid_pos`	The position of the local processor in the grid. Positions run from '0'.
`proc_id`	A unique number labelling the local processor.
`boss_proc`	A nominated boss processor. Any operations which should only be performed by one processor are performed by the boss processor.

Table 3.1: Parameters for the `init_comms` routine.

Each process finds the above information from the boss processor. A loader program, *e.g.* `pvmgrid` for PVM as described in Appendix E, spawns the processes and sends the necessary information to them.

Once we know the local processor position and ID, the boss processor reads in the parameters for the application run; F77 file handling is performed by one processor only. As we have been told the boss processor and local processor numbers it is trivial to tell whether the local processor is the boss or not. Parameters are set in other processors using the routines shown in table 3.2

Routine	Data type
`ig_set`	integer
`ivg_set`	vector of integers
`lg_set`	logical
`lvg_set`	vector of logicals
`g_set`	single precision real
`vg_set`	vector of single prec. reals
`dg_set`	double precision real
`dvg_set`	vector of double prec. reals

Table 3.2: Routines to set data elements on all processors.

The interfaces to these routines are as straightforward as you would expect, *e.g.*

```
      Fpoint kappa, v_real(0:9)

      CALL g_set (kappa)
      CALL vg_set (10, v_real)
```

Before we make any references to lattice coordinates we need to initialise the common block containing all information about the decomposition. This is done by calling routine `grid_start` with the following interface

```
      SUBROUTINE grid_start (grid_size, grid_pos,
     $  global_latt, proc_id, boss_proc)

C extents of the global lattice.
      INTEGER global_latt(0:Ndim-1)
```

all other parameters are as specified above for `init_comms`. The `grid_start` routine sets up the parameters defined in table 3.3.

Dimension `Ndim`=4	
`local_latt`	The local lattice size in each direction.
`is_local`	Set to `TRUE` if the grid is only one processor wide in a particular dimension.
`local_start`	Global coordinates of the first local lattice point.
`local_end`	Global coordinates of the last local lattice point.
`n_bound`	Number of sites in each boundary.
`np_bound`	Number of boundary sites of each parity.
`np_comm`	As `np_bound` except if `is_local` is `TRUE` there is no communications in this direction so `np_comm` equals zero.

Dimension `Ndim*Max_width`	
`grid_latts`	All local sizes.
`gr_starts`	All local starts.
`gr_ends`	All local ends.

Scalars	
`base_parity`	Parity of the first local site.
`n_sites`	Number of local sites.
`np_sites`	Number of local sites of each parity.
`g_sites`	Number of global sites.
`gp_sites`	Number of global sites of each parity.

Table 3.3: Parameters initialised by `grid_start`.

Now we know the characteristics of the local lattice we can define the shift (gather-scatter) and boundary tables as introduced in section 2.2.5.4. As before, we have a routine to do this, *i.e.*

```
      CALL make_table (is_local, shift_table,
     $                 boundary_table)
      CALL conv_shift_tables ()
```

acting on the common block variables defined in a header file for ease of use. Routine `conv_shift_tables` only implements compressed gather-scatter tables if desired at build-time. Everything is now set up ready for the application.

MP implementation: global sums

Global sums are calculated by passing local values up to the boss processor creating a cumulative sum. The final value is then broadcast back to the local processors, either using the `_set` routines discussed earlier or a package specific method. We use a different routine for each data type as we did with the global setting routines, *e.g.*

```
      Fpoint r, vr(0:9)
      INTEGER i, vi(0:12)

      CALL g_sum (r)
      CALL vg_sum (10, vr)
      CALL ig_sum (i)
      CALL ivg_sum (13, vi)
```

MP implementation: local shift

All processors are running the same program so boundary transfers always come in pairs; an outgoing send and incoming receive in opposite directions. For each communication we make two subroutine calls; one to initialise the communication, `fstart_com` for reals or `istart_com` for integers, and one to end it, `fend_com` or `iend_com`. Between these calls the processor is free to perform any other work provided the buffers are not disturbed. The call to end the communication will not return until all data is safely received.

The syntax for these calls is

```
      SUBROUTINE fstart_com (len, idir, icmp, ocmp, ilen,
     $                       finput, olen, foutput)
      SUBROUTINE fend_com   (len, idir, icmp, ocmp, ilen,
     $                       finput, olen, foutput)
      SUBROUTINE istart_com (len, idir, icmp, ocmp, ilen,
     $                       iinput, olen, ioutput)
      SUBROUTINE iend_com   (len, idir, icmp, ocmp, ilen,
     $                       iinput, olen, ioutput)

      INTEGER len, idir, icmp, ocmp, ilen, olen,
     $        iinput (0:icmp-1,0:ilen-1,0:ocmp-1),
     $        ioutput(0:icmp-1,0:olen-1,0:ocmp-1)
      Fpoint  finput (0:icmp-1,0:ilen-1,0:ocmp-1),
     $        foutput(0:icmp-1,0:olen-1,0:ocmp-1)
```

where the parameters are explained in table 3.4.

`len`	The length of the communication will be `len`×`icmp`.
`idir`	Direction of the communication. Values 0, 1, 2, 3 represent a shift in the positive x, y, z and t directions; values 4, 5, 6, 7 represent negative shifts.
`icmp`	Number of components inside the length index.
`ocmp`	Number of components outside the length index.
`olen`	Length of the output array.
`<f|i>output`	Output array for the send.
`ilen`	Length of the input array.
`<f|i>input`	Input array for the receive.

Table 3.4: Parameters for the communications routines.

An example of use is

```
      INTEGER length
      PARAMETER (length=10)

      Fpoint fin_buff(0:length-1), fout_buff(0:length-1)
      INTEGER iin_buff(0:length-1), iout_buff(0:length-1)

C start communication in positive Z-direction
      CALL fstart_com (length,2,1,1,length,fin_buff,
     $                 length,fout_buff)

C start communication in negative T-direction
      CALL istart_com (length,7,1,1,length,iin_buff,
     $                 length,iout_buff)
```

```
C wait for both comms to end.
      CALL fend_com   (length,2,1,1,length,fin_buff,
     $                 length,fout_buff)
      CALL iend_com   (length,7,1,1,length,iin_buff,
     $                 length,iout_buff)
```

Further examples are shown in the later sections of this chapter.

MP implementation: tidying up

When the application has finished we need to provide a tidy way to terminate the communications package. The routine provided is `finish_comms`, *i.e.*

```
      CALL finish_comms ()
```

3.2 Parallel I/O

As HPF does not define a standard for parallel I/O we cannot discuss portability; there is none. All of the platform-specific routines for handling parallel I/O operations are therefore concentrated into a few isolated routines as discussed for the GAUGE application in figure 4.3. We keep the format for the large scale data files used on the MPP machines free so that the platform can use the fastest I/O possible. For example, on the Connection Machine we use the 'fixed machine size' I/O commands which do not pad out the data files (size is at a premium on most mass-data stores) and allow fast I/O access from a DataVault. When the configurations are used on another machine for post-processing we must convert the format into the standard flat format defined for all applications in the MPP suite.

We do not have a generic layer of I/O routines in HPF for any data type because of the strong type checking; it is easier to have a different routine for each data type. An example of the call structure is given in figure 4.3.

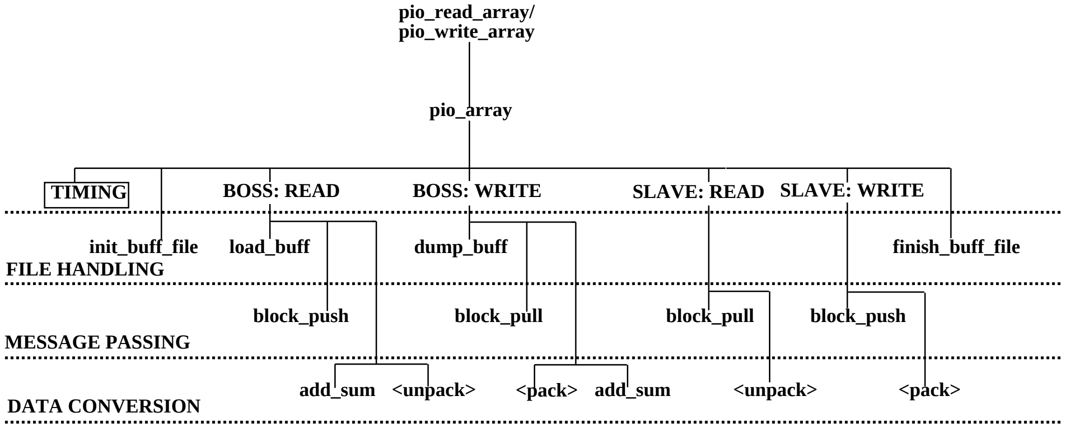


Figure 3.2: Call structure of the generic message passing I/O routines.

As the message passing codes use F77 as the Fortran layer, we can use generic routines for all data types. The structure is shown in figure 3.2. These routines use normal Fortran I/O on a single nominated node, the boss processor. This approach is portable, and in use on the Cray T3D, but not necessarily efficient for all machines, especially if they supply routines to perform I/O from multiple processors simultaneously. Each call to a 'write' routine creates a new file specified by name. We do not support appending to existing files as this would again reduce portability. The I/O routines assume that all data associated with a single lattice site is located in a contiguous section of the data file: an 'atom'. To pull data from memory into this atomic form, or vice-versa, we use a packing or unpacking routine which is passed down to '`pio_array`'. These (un)packing routines allow for the index swapping needed to optimise code for vectorisation and are specific to a particular data type. For example, '`pack_gauge`' and '`unpack_gauge`' are used to implement the two-row format needed for a gauge configuration.

The file is opened by the boss processor using routine '`init_buff_file`', a normal F77 '`OPEN`' statement together with any platform-specific qualifiers. Operation then depends on whether the file is being read or written. For reading files the boss processor loops over the data file indices, reads in a block of data contiguous to a certain processor using '`load_buff`', adds the contents to the checksum and then either unpacks it (local memory) or sends the block to a remote processor using '`block_push`'. The message passing layer is implemented directly in terms of the message passing primitives as shown in Appendix E. The remote processors receive the sent block and then unpack the contents to local memory. Writing a file is the opposite sequence of events.

The file formats used are straightforward: for gauge configurations the loop ordering is: Real part of complex (fastest moving), Imaginary part of complex, Colour row (0–1), Colour column (0–2), Direction μ, X, Y, Z (slowest moving). And for quark propagators: Real part of complex (fastest moving), Imaginary part of complex, Colour (0–2), Spin (0–3), X, Y, Z (slowest moving).

3.3 Parallel I/O performance

We have measured the speed of the I/O systems on various platforms as shown in table 3.5[1].

				12^4		16^4	
App.	Platform	Size	Precision	Speed MB/s	Efficiency % peak	Speed MB/s	Efficiency % peak
G	CM200	8K	D	0.140	0.56	0.127	0.51
G	CM200	8K	S	0.095	0.38		
G	CM5	16	D	0.216	0.68	1.144	3.58
G	CM5	32	D	0.277	0.87	0.561	1.75
G	T3D	8	D	1.62	5.1	2.29	7.2
G	T3D	16	D	1.56	4.9	1.97	6.2
S	CM200	8K	S	0.197	0.79	0.162	0.65
S	CM200	8K	D	0.181	0.73		
S	CM5	16	S	0.355	1.11		
S	CM5	32	S	0.168	0.53		
S	CM5	32	D	0.322	1.01		
S	T3D	8	D	1.05	3.28		
S	T3D	16	D	0.910	2.84		

Table 3.5: Performance data for parallel I/O on various platforms. The top part of the table shows data from the GAUGE application, the bottom section shows data from the SOLVER application.

It can immediately be seen that the efficiency of these operations is extremely low, at most 7% of the peak rate. The main reason for this poor performance is that the fields are saved in time-sliced form. This increases the number of files which need to be opened and close, with the opening and closing operations validated. Our timings are for the whole configuration, not a single read/write operation so these extra elements become important. The need to increase the length of single data transfers is shown by the increase in performance for larger lattices and the reduction in speed for more nodes.

The CM200 (peak I/O rate of 25 MB/s) and CM5 (peak I/O rate of 32 MB/s) timings use the 'fixed memory size' I/O routines which do not pad out files to huge lengths, a useful saving in disk space. They can only be read back

---

[1] The platforms are discussed in section 3.6.

into a machine of the same size as they were written from, usually not too big a restriction as lattice QCD tends to need the whole machine to get high performance for computation.

The T3D (peak I/O rate of 32 MB/s) is an factor of 10 better in performance, although still only 7% of peak at best. This poor performance is due to the communication between the T3D and its YMP front-end. Test code on the YMP can achieve almost peak I/O transfers [101]. The I/O is performed in an asynchronous way, the I/O takes place in the background once started so that communications with other processors can occur concurrently. This I/O method has yet to be optimised; applications programmers at Edinburgh Parallel Computing Centre expect a large improvement in the near future.

3.4 Maths

Throughout the MPP codes we need to perform matrix operations, mainly multiplication and addition, on a variety of data types. By channeling such operations through subroutines, rather than performing them in-line, we incur a slight slowing down through the routine call and return but gain by reducing code space and providing an easy target area for platform-specific optimisations. We form the routines into a library as they are not likely to change in the future, except for optimisations, and should be isolated from the higher-level routines.

SU(3) matrices

The largest number of routines are associated with the gauge fields used throughout all codes. In HPF we always operate on a single parity sub-lattice, the smallest unit passed around the codes, and in MP we use a variable length vector of SU(3) matrices adaptable to all situations.

There are only two unary operations on gauge fields, taking the trace over colour indices and daggering (taking the hermitian conjugate). The implementation and use of these routines is straightforward in either HPF or MP model. For example in HPF

```
      Cfpoint, DIMENSION (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1) ::
     $  a, b
!HPF$ DISTRIBUTE (*,*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS ::
!HPF$$ a, b
      Cfpoint, DIMENSION  t (
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)
!HPF$ DISTRIBUTE t (BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS

C t = Trace (a)
      CALL trace_3by3 (a, t)

C b = hermitian conjugate of 'a'
```

```
      CALL dagger_3by3 (a, b)

C a = hermitian conjugate of 'a'
      CALL r_dagger_3by3 (a)
```

Note that we use a convention that the rightmost parameter(s) contain the results of the operation. The '`r_`' prefix denotes a form of in-place operation, this notation is expanded for binary operations as shown below.

Binary operations on gauge fields are more complicated. Matrix multiplication can be used to combine daggered or undaggered matrices and the result can be placed in a new matrix, the left operand or right operand. The various possibilities are tabulated below.

`hh_3by3`	$a = b^\dagger * c^\dagger$	`lr_hh_3by3`	$b = b^\dagger * c^\dagger$	`rr_hh_3by3`	$c = b^\dagger * c^\dagger$
`hm_3by3`	$a = b^\dagger * c$	`lr_hm_3by3`	$b = b^\dagger * c$	`rr_hm_3by3`	$c = b^\dagger * c$
`mh_3by3`	$a = b * c^\dagger$	`lr_mh_3by3`	$b = b * c^\dagger$	`rr_mh_3by3`	$c = b * c^\dagger$
`mm_3by3`	$a = b * c$	`lr_mm_3by3`	$b = b * c$	`rr_mm_3by3`	$c = b * c$

In the MP approach we have a further complication in the addressing mode. Routine names are written as

$$[\mathit{dest}][\mathit{operation}][\mathit{addr\ mode}]\_3\mathrm{by}3\ (n,\ \{\mathcal{T}\},\ l_b,\ b,\ l_c,\ c,\ \{l_a\},\ \{a\})$$

Optional parameters are shown in '$\{\ldots\}$'. The 'n' parameter is the number of SU(3) matrices to operate on, '$\mathcal{T}$' is a gather-scatter table and 'l_x' is the length of vector x. The destination for the result, '[*dest*]' can be any of {'', '`lr_`', '`rr_`'}, the multiplication operation can be any of {'`hh`', '`hm`', '`mh`', '`mm`'} and the addressing mode '[*addr mode*]' can take any of the following values

`'_s'`	The result is scattered using table $\mathcal{T}$.
`'_lg'`	Matrix b is gathered using table $\mathcal{T}$. We cannot use this in conjunction with destination '`lr_`'.
`'_rg'`	Matrix c is gathered using table $\mathcal{T}$. We cannot use this in conjunction with destination '`rr_`'.
`'_lc'`	Matrix b is a single matrix rather than a vector and is used for all

elements of c. We cannot use this with the destination '`lr_`'.

'`_rc`' Matrix c is a single matrix rather than a vector and is used for all elements of c. We cannot use this with the destination '`rr_`'.

'' No indirect addressing.

The are so many possibilities that we only implement those needed according to the naming scheme above.

In HPF, addition of matrices is trivial since the code

```
      Cfpoint, DIMENSION (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1) :: a, b, c
!HPF$ DISTRIBUTE (*,*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS ::
!HPF$$ a, b, c

     a = b + c
```

operates on *all* elements of `a`, `b` and `c`. It is more tedious to do this with the MP code as we have four indices to loop over and the possibilities of gathering/scattering arrays. We therefore use subroutines, extending the previous naming scheme to incorporate the '`add`' operation. An example of such a routine is '`lr_add_rg_3by3`'.

SU(2) matrices

We perform gauge update algorithms using SU(2) subgroups as described in Appendix B. To reduce space needed and speed up computation we represent these complex 2×2 matrices as four real Pauli parameters *i.e.*

$$\mathcal{M}_{2\times 2} = m_0 \mathbf{1} + i\mathbf{m}.\sigma$$

e.g. in HPF

```
C  Npauli is defined to be 4
      Fpoint, DIMENSION (0:Npauli-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1) :: a
!HPF$ DISTRIBUTE a (*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS
```

We only need to perform a few operations on these data objects and they are all *local*, *i.e.* no gather-scatter needed. The routines supplied are

`mm_su2`	$a = b * c$
`dagger_su2`	$a = b^\dagger$
`square_su2`	$a = b * b$

Spinors

The routines which act on 4-spinors form the basic toolkit for constructing different solver algorithms and, as such, are discussed in section 5.2.

Mixed data-type operations

All operations performed on mixed types are forced to be local, simplifying their interfaces. We summarise the required operations below; ψ is a 4-spinor, χ a 2-spinor and U an SU(3) matrix.

`su3_hvv`	$\chi_a = U_b^\dagger * \chi_c$
`su3_mvv`	$\chi_a = U_b * \chi_c$
`su3_hv4`	$\psi_a = U_b^\dagger * \psi_c$
`su3_mv4`	$\psi_a = U_b * \psi_c$
`xpgammay`	$\psi_x = \psi_x + \gamma_\mu \psi_y$
`xmgammay`	$\psi_x = \psi_x - \gamma_\mu \psi_y$
`gammax`	$\psi_x = \gamma_\mu \psi_x$
`mm_su2_su3`	$U = m_{2\times 2} U$

3.5 Testing the maths routines

When testing maths routines we need known results generated in as independent a way as possible from the production codes. In practice we use the serial 'gcc' C compiler on a SUN workstation.

3.5.1 SU(3) matrix testing

The first routines to test are those which operate on SU(3) matrices. If these do not work properly, nothing else will. The two matrices

$$
\begin{aligned}
A= & \left(\begin{array}{ccc}
\begin{bmatrix}1.2\\3.4i\end{bmatrix} & \begin{bmatrix}5.6\\7.8i\end{bmatrix} & \begin{bmatrix}9.10\\11.12i\end{bmatrix}\\
\begin{bmatrix}13.14\\15.16i\end{bmatrix} & \begin{bmatrix}17.18\\19.20i\end{bmatrix} & \begin{bmatrix}21.22\\23.24i\end{bmatrix}\\
\begin{bmatrix}25.26\\27.28i\end{bmatrix} & \begin{bmatrix}29.30\\31.32i\end{bmatrix} & \begin{bmatrix}33.34\\35.36i\end{bmatrix}
\end{array}\right)\\
B= & \left(\begin{array}{ccc}
\begin{bmatrix}2.3\\4.5i\end{bmatrix} & \begin{bmatrix}6.7\\8.9i\end{bmatrix} & \begin{bmatrix}10.11\\12.13i\end{bmatrix}\\
\begin{bmatrix}14.15\\16.17i\end{bmatrix} & \begin{bmatrix}18.19\\20.21i\end{bmatrix} & \begin{bmatrix}22.23\\24.25i\end{bmatrix}\\
\begin{bmatrix}26.27\\28.29i\end{bmatrix} & \begin{bmatrix}30.31\\32.33i\end{bmatrix} & \begin{bmatrix}34.35\\36.37i\end{bmatrix}
\end{array}\right)
\end{aligned}
\tag{3.1}
$$

give the following results when the code works correctly.

$$
\begin{aligned}
A^{\dagger}= & \left(\begin{array}{ccc}
\begin{bmatrix}1.2\\-3.4i\end{bmatrix} & \begin{bmatrix}13.14\\-15.16i\end{bmatrix} & \begin{bmatrix}25.26\\-27.28i\end{bmatrix}\\
\begin{bmatrix}5.6\\-7.8i\end{bmatrix} & \begin{bmatrix}17.18\\-19.20i\end{bmatrix} & \begin{bmatrix}29.30\\-31.32i\end{bmatrix}\\
\begin{bmatrix}9.10\\-11.12i\end{bmatrix} & \begin{bmatrix}21.22\\-23.24i\end{bmatrix} & \begin{bmatrix}33.34\\-35.36i\end{bmatrix}
\end{array}\right)\\
A_{SU(3)}= & \left(\begin{array}{ccc}
\begin{bmatrix}0.067973\\0.192591i\end{bmatrix} & \begin{bmatrix}0.317208\\0.441826i\end{bmatrix} & \begin{bmatrix}0.515463\\0.629885i\end{bmatrix}\\
\begin{bmatrix}0.737817\\0.561411i\end{bmatrix} & \begin{bmatrix}0.118322\\0.114478i\end{bmatrix} & \begin{bmatrix}-0.305720\\-0.140976i\end{bmatrix}\\
\begin{bmatrix}-0.023573\\0.313332i\end{bmatrix} & \begin{bmatrix}0.020323\\-0.822588i\end{bmatrix} & \begin{bmatrix}-0.000000\\0.473502i\end{bmatrix}
\end{array}\right)\\
\mathrm{Tr}\ A= & \begin{bmatrix}51.72\\57.96i\end{bmatrix}\\
B^{\dagger}= & \left(\begin{array}{ccc}
\begin{bmatrix}2.3\\-4.5i\end{bmatrix} & \begin{bmatrix}14.15\\-16.17i\end{bmatrix} & \begin{bmatrix}26.27\\-28.29i\end{bmatrix}\\
\begin{bmatrix}6.7\\-8.9i\end{bmatrix} & \begin{bmatrix}18.19\\-20.21i\end{bmatrix} & \begin{bmatrix}30.31\\-32.33i\end{bmatrix}\\
\begin{bmatrix}10.11\\-12.13i\end{bmatrix} & \begin{bmatrix}22.23\\-24.25i\end{bmatrix} & \begin{bmatrix}34.35\\-36.37i\end{bmatrix}
\end{array}\right)
\end{aligned}
$$

$$
\begin{aligned}
A^{\dagger} * B^{\dagger} = & \left(\begin{array}{ccc}
\begin{bmatrix} -134.954 \\ -813.943i \end{bmatrix} & \begin{bmatrix} -205.375 \\ -1827.823i \end{bmatrix} & \begin{bmatrix} -281.004 \\ -2863.356i \end{bmatrix} \\
\begin{bmatrix} -161.683 \\ -996.736i \end{bmatrix} & \begin{bmatrix} -230.585 \\ -2304.147i \end{bmatrix} & \begin{bmatrix} -306.214 \\ -3642.194i \end{bmatrix} \\
\begin{bmatrix} -185.621 \\ -1172.996i \end{bmatrix} & \begin{bmatrix} -251.066 \\ -2750.635i \end{bmatrix} & \begin{bmatrix} -324.513 \\ -4367.200i \end{bmatrix}
\end{array} \right) \\
A^{\dagger} * B = & \left(\begin{array}{ccc}
\begin{bmatrix} 1884.460 \\ -6.500i \end{bmatrix} & \begin{bmatrix} 2231.293 \\ -32.502i \end{bmatrix} & \begin{bmatrix} 2572.961 \\ -56.542i \end{bmatrix} \\
\begin{bmatrix} 2257.295 \\ 19.501i \end{bmatrix} & \begin{bmatrix} 2708.135 \\ -6.500i \end{bmatrix} & \begin{bmatrix} 3144.305 \\ -31.332i \end{bmatrix} \\
\begin{bmatrix} 2623.200 \\ 43.937i \end{bmatrix} & \begin{bmatrix} 3169.334 \\ 18.727i \end{bmatrix} & \begin{bmatrix} 3693.449 \\ -6.121i \end{bmatrix}
\end{array} \right) \\
A * B^{\dagger} = & \left(\begin{array}{ccc}
\begin{bmatrix} 351.887 \\ 6.880i \end{bmatrix} & \begin{bmatrix} 803.413 \\ 83.935i \end{bmatrix} & \begin{bmatrix} 1266.639 \\ 161.745i \end{bmatrix} \\
\begin{bmatrix} 880.863 \\ -70.966i \end{bmatrix} & \begin{bmatrix} 2166.895 \\ 6.121i \end{bmatrix} & \begin{bmatrix} 3489.672 \\ 79.568i \end{bmatrix} \\
\begin{bmatrix} 1421.900 \\ -148.777i \end{bmatrix} & \begin{bmatrix} 3563.119 \\ -67.327i \end{bmatrix} & \begin{bmatrix} 5767.262 \\ 6.121i \end{bmatrix}
\end{array} \right) \\
A * B = & \left(\begin{array}{ccc}
\begin{bmatrix} -134.954 \\ 763.703i \end{bmatrix} & \begin{bmatrix} -161.683 \\ 919.768i \end{bmatrix} & \begin{bmatrix} -185.621 \\ 1071.063i \end{bmatrix} \\
\begin{bmatrix} -205.375 \\ 1854.307i \end{bmatrix} & \begin{bmatrix} -230.585 \\ 2305.421i \end{bmatrix} & \begin{bmatrix} -251.066 \\ 2726.152i \end{bmatrix} \\
\begin{bmatrix} -281.004 \\ 2965.469i \end{bmatrix} & \begin{bmatrix} -306.214 \\ 3719.098i \end{bmatrix} & \begin{bmatrix} -324.513 \\ 4416.165i \end{bmatrix}
\end{array} \right) \\
A_{\mathrm{herm}} = & \left(\begin{array}{ccc}
\begin{bmatrix} 3.4 \\ 0.0i \end{bmatrix} & \begin{bmatrix} 11.48 \\ 3.77i \end{bmatrix} & \begin{bmatrix} 19.20 \\ 8.08i \end{bmatrix} \\
\begin{bmatrix} 11.48 \\ 0.0i \end{bmatrix} & \begin{bmatrix} 19.20 \\ 0.0i \end{bmatrix} & \begin{bmatrix} 27.28 \\ 4.04i \end{bmatrix} \\
\begin{bmatrix} 19.20 \\ -8.08i \end{bmatrix} & \begin{bmatrix} 27.28 \\ -4.04i \end{bmatrix} & \begin{bmatrix} 35.36 \\ 0.0i \end{bmatrix}
\end{array} \right)
\end{aligned} \tag{3.2}
$$

3.5.2 Spinor testing

The routines at the core of the solver perform the matrix operations

$$
\begin{aligned}
A' &= A * D \\
A'' &= A^{\dagger} * D
\end{aligned} \tag{3.3}
$$

where A is an SU(3) matrix and D is a 2-spinor. Only the colour indices of D are involved so if we use our previous definition of A, and define the colour indices of D to be

$$D = \begin{pmatrix} \begin{bmatrix} \text{37.38} \\ \text{39.40i} \end{bmatrix} \\ \begin{bmatrix} \text{41.42} \\ \text{43.44i} \end{bmatrix} \\ \begin{bmatrix} \text{45.46} \\ \text{47.48i} \end{bmatrix} \end{pmatrix} \tag{3.4}$$

then we obtain the results

$$\begin{aligned} A * D &= \begin{pmatrix} \begin{bmatrix} \text{-310.2756} \\ \text{1678.2952i} \end{bmatrix} \\ \begin{bmatrix} \text{-367.353} \\ \text{4689.9756i} \end{bmatrix} \\ \begin{bmatrix} \text{-440.8045} \\ \text{7775.4854i} \end{bmatrix} \end{pmatrix} \\ A^{\dagger} * D &= \begin{pmatrix} \begin{bmatrix} \text{3825.1992} \\ \text{-177.7416i} \end{bmatrix} \\ \begin{bmatrix} \text{4881.3428} \\ \text{-152.5320i} \end{bmatrix} \\ \begin{bmatrix} \text{5861.2930} \\ \text{-122.4120i} \end{bmatrix} \end{pmatrix} \end{aligned} \tag{3.5}$$

3.5.3 Gauge update matrix testing

When performing gauge updates we use SU(2) subgroups. To test the SU(2) maths routines we use the known SU(2) matrices parametrised in terms of Pauli matrices

$$\begin{aligned} E &= (1, 4, 3, 2) \\ F &= (5, 8, 7, 6) \end{aligned} \tag{3.6}$$

which multiply to give

$$E * F = (-60, -24, 30, 12) \tag{3.7}$$

We can then use the previous definition of A to test the multiplication by subgroups to get SU(3) matrices:

$$E * A_{01}=\begin{pmatrix} \begin{bmatrix} \text{-26.82} \\ \text{103.84i} \end{bmatrix} & \begin{bmatrix} \text{-35.26} \\ \text{145.32i} \end{bmatrix} & \begin{bmatrix} \text{-42.44} \\ \text{183.92i} \end{bmatrix} \\ \begin{bmatrix} \text{26.26} \\ \text{-16.52i} \end{bmatrix} & \begin{bmatrix} \text{7.58} \\ \text{-16.16i} \end{bmatrix} & \begin{bmatrix} \text{-4.08} \\ \text{-16.16i} \end{bmatrix} \\ \begin{bmatrix} \text{25.26} \\ \text{27.28i} \end{bmatrix} & \begin{bmatrix} \text{29.30} \\ \text{31.32i} \end{bmatrix} & \begin{bmatrix} \text{33.34} \\ \text{35.36i} \end{bmatrix} \end{pmatrix}$$

$$E * A_{12}=\begin{pmatrix} \begin{bmatrix} \text{1.2} \\ \text{3.4i} \end{bmatrix} & \begin{bmatrix} \text{5.6} \\ \text{7.8i} \end{bmatrix} & \begin{bmatrix} \text{9.10} \\ \text{11.12i} \end{bmatrix} \\ \begin{bmatrix} \text{-50.52} \\ \text{224.32i} \end{bmatrix} & \begin{bmatrix} \text{-58.60} \\ \text{264.72i} \end{bmatrix} & \begin{bmatrix} \text{-66.68} \\ \text{305.12i} \end{bmatrix} \\ \begin{bmatrix} \text{-20.24} \\ \text{-16.16i} \end{bmatrix} & \begin{bmatrix} \text{-36.40} \\ \text{-16.16i} \end{bmatrix} & \begin{bmatrix} \text{-52.56} \\ \text{-16.16i} \end{bmatrix} \end{pmatrix}$$

$$E * A_{02}=\begin{pmatrix} \begin{bmatrix} \text{-38.94} \\ \text{188.68i} \end{bmatrix} & \begin{bmatrix} \text{-47.38} \\ \text{230.16i} \end{bmatrix} & \begin{bmatrix} \text{-54.56} \\ \text{268.76i} \end{bmatrix} \\ \begin{bmatrix} \text{13.14} \\ \text{15.16i} \end{bmatrix} & \begin{bmatrix} \text{17.18} \\ \text{19.20i} \end{bmatrix} & \begin{bmatrix} \text{21.22} \\ \text{23.24i} \end{bmatrix} \\ \begin{bmatrix} \text{62.62} \\ \text{-28.64i} \end{bmatrix} & \begin{bmatrix} \text{43.94} \\ \text{-28.28i} \end{bmatrix} & \begin{bmatrix} \text{32.28} \\ \text{-28.28i} \end{bmatrix} \end{pmatrix} \tag{3.8}$$

The final test of the gauge update maths routines is to ensure that k and u_1 required in section B.1.1 are calculated correctly. If we use a complex 2×2 matrix

$$G = \begin{pmatrix} \begin{bmatrix} \text{0.104820} \\ \text{0.209657i} \end{bmatrix} & \begin{bmatrix} \text{0.314485} \\ \text{0.419314i} \end{bmatrix} \\ \begin{bmatrix} \text{0.684738} \\ \text{0.556348i} \end{bmatrix} & \begin{bmatrix} \text{0.171185} \\ \text{0.171185i} \end{bmatrix} \end{pmatrix} \tag{3.9}$$

we should get a resultant SU(2) matrix (Pauli parametrised)

$$u_1 = (0.2555314, -0.9032891, 0.3427882, -0.0356182) \tag{3.10}$$

and a k of $2.7774645 \times \beta^{-1}$.

3.6 Maths routine performance

The performance of the maths routines is easily compared on different MPP platforms by timing the '`mm_3by3`' and '`su3_hvv`' routines. The first of these, '`mm_3by3`', is heavily used in the GAUGE application code, and multiplies together two SU(3) matrix arrays over a single-parity sub-lattice, taking 198 floating point operations (flops) per lattice site. The second routine, '`su3_hvv`', forms the core of the SOLVER application, multiplying together a daggered SU(3) matrix and a two-spinor. This routine takes 132 flops per lattice site and, again, operates on a single-parity sub-lattice.

Connection Machine CM-200 performance

The Thinking Machines Connection Machine CM-200 is a data-parallel machine running CM Fortran, similar to HPF as explained in Appendix D. The CM-200 used has 16384 (16K) single-bit processors which are grouped into groups of 32-bit compound processors. Each of these 32-bit processors has a double-precision Weitek floating-point-accelerator processor connected to it. The result is, in effect, a 512 processor SIMD computer with a peak speed of 8 Gflop/s. The CM-200 can be operated as a single partition of 512 processors or two partitions of 256 processors (8K single-bit processors) each with a peak speed of 4 Gflop/s. The software used was CM Fortran Slicewise version 2.1.1-2 (SPARC).

Size	Prec.	12^4 lattice			16^4 lattice		
		Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
8K	D	0.852e-2	0.24	6.0	0.161e-1	0.40	10.1
8K	S	0.674e-2	0.30	7.6			
16K	S	0.586e-2	0.35	4.4	0.686e-2	0.946	11.8

Table 3.6: Performance data for the '`mm_3by3`' routine on a Thinking Machines CM-200 computer. The CM-200 used to generate this data was clocked at 8MHz for a peak speed (16K processors) of 8Gflop/s.

In tables 3.6 and 3.7 we present data for the performance of the '`mm_3by3`' and '`su3_hvv`' routines on the CM-200 for differing machine sizes and precision.

Size	Prec.	12^4 lattice			16^4 lattice		
		Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
8K	S	0.331e-2	0.42	10.4	0.555e-2	0.78	19.5
8K	D	0.519e-2	0.26	6.6			
16K*	S	0.303e-2	0.45	5.6	0.409e-2	1.06	13.2
8K*	S				0.657e-2	0.66	16.5

Table 3.7: Performance data for the '`su3_hvv`' routine on a CM-200 computer. Entries with a '*' are calculated from code with unrolled serial-index loops.

We can see that the 12^4 lattice does not give as high performance as the 16^4 lattice. The CM-200 distributes the arrays across its processors so that each dimension is a power of two. This is straightforward in the 16^4 case, the decomposition is shown in table 3.8. Note that we are using single-parity sub-lattices, so the x direction has half the extent of the other directions. The 12^4 lattice however is padded in the z- and t-directions. This padding results in a direct reduction of performance as the padded elements must be avoided during computation. The same problem applies to the case of 16K processors.

	12^4 lattice			16^4 lattice		
Direction	Physical	Local	Phys.*Local	Physical	Local	Phys.*Local
x	1	6	6	2	4	8
y	4	3	12	4	4	16
z	8	2	16	4	4	16
t	8	2	16	8	2	16

Table 3.8: Decomposition of the lattice onto the CM-200 processor array (8K processors, or 256 compound processing elements). The physical extents are forced to be powers of two; the product of these must equal the number of compound processing elements ($1 \times 4 \times 8 \times 8 = 256$).

The fairly low performance on the CM-200 (only 20% of peak at best) is partially due to not being able to perform both an addition and multiplication on each cycle. There are no communications in these two routines, so that cannot be reducing performance. Another contributing factor is load on the front-end from other users. Since the front-end is responsible for broadcasting instructions to the processor array performance will be degraded if the front-end is required to perform other operations. The CM-200 used is extremely heavily used resulting in some loss of performance.

The increase in performance for the 'su3_hvv' routine over 'mm_3by3' is probably due to the layout of the serial indices in memory. The gauge fields are declared as

```
      Cfpoint gauge (0:2,0:2,
     $ 0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)
```

and addressed in the order row, column. The two-spinor fields are declared as

```
      Cfpoint chi (0:2,0:1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)
```

where the first index is the colour index and the second is the two-spinor index. The 'su3_hvv' routine only operates on the colour indices. To obtain a single element in a gauge field from multiplying two gauge fields together we loop over the columns in the first field and the rows in the second field as shown below.

$$\begin{pmatrix} X & . & . \\ . & . & . \\ . & . & . \end{pmatrix} = \begin{pmatrix} X & X & X \\ . & . & . \\ . & . & . \end{pmatrix} \begin{pmatrix} X & . & . \\ X & . & . \\ X & . & . \end{pmatrix}$$

CM Fortran uses the C convention for fastest-moving index; the column index moves fastest. When we use the daggered matrix for the 'su3_hvv' operation we are in effect doing

$$\begin{pmatrix} X & . \\ . & . \\ . & . \end{pmatrix} = \begin{pmatrix} X & . & . \\ X & . & . \\ X & . & . \end{pmatrix}^* \begin{pmatrix} X & . \\ X & . \\ X & . \end{pmatrix}$$

so the memory is accessed with the same stride for all arrays, a more efficient operation.

Connection Machine CM-5 performance

The Connection Machine CM-5 is intended for MIMD programming, although it can run the same data-parallel CM Fortran code as the CM-200; no alterations are necessary. It is in this SIMD mode that we use the platform. Each node consists of a SPARC processor and 4 vector processors for floating-point arithmetic. The peak speed of a node is 160 Mflop/s (for a clock speed of

8MHz) resulting in a peak speed for the machine used of 5.12 Gflop/s (32 nodes). The machine can be operated as a single partition of 32 nodes, or two partitions of 16 nodes each. The software used to create the following data was CMOST version 7.3 Final 1 Rev 3 and CM Fortran version 2.1.1-2 (CM5 VecUnit).

In tables 3.9 and 3.10 we present data for the two routines under consideration. We can see that good efficiency is obtained from the vector processors without any optimisation. The 12^4 lattice is not as big a problem as it was on the CM-200, the arrays are distributed as shown in table 3.11. There is no padding, and therefore no empty elements to worry about.

Size	Prec.	12^4 lattice			16^4 lattice		
		Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
16	D	0.224e-2	0.92	35.7	0.678e-2	0.96	37.4
16	S				0.496e-2	1.31	51.1
32	D	0.123e-2	1.67	32.6	0.342e-2	1.90	37.1

Table 3.9: Performance data for the '`mm_3by3`' routine on a Thinking Machines CM-5 computer. The CM-5 used to generate this data was clocked at 8MHz for a peak speed (32 nodes) of 5.12Gflop/s.

Size	Prec.	12^4 lattice			16^4 lattice		
		Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
32	S	0.815e-3	1.68	32.8	0.203e-2	2.13	41.6
32	D	0.100e-2	1.37	26.7			
16	S	0.138e-2	0.99	38.7			
16*	S	0.123e-2	1.11	43.5	0.354e-2	1.22	47.7

Table 3.10: Performance data for the '`su3_hvv`' routine on a CM-5.

Changing from single-precision to double-precision does not halve performance as the floating-point vector units operate on 64-bit data. The reduction in performance comes from the increased memory-access time required.

A notable improvement in performance comes from unrolling serial-index loops in the code, as indicated by the 16* entry in table 3.10. The '`DO`' loops over colour and two-spinor indices are removed completely increasing the code space

	12^4 lattice			16^4 lattice		
Direction	Physical	Local	Phys.*Local	Physical	Local	Phys.*Local
x	1	6	6	2	4	8
y	4	3	12	2	8	16
z	4	3	12	4	4	16
t	4	3	12	4	4	16

Table 3.11: Decomposition of the lattice onto the CM-5 processor array (16 nodes). The physical extents are forced to be powers of two; the product of these must equal the number of vector processors, 4 per node ($1 \times 4 \times 4 \times 4 = 64$).

required considerably, but at the same time increasing performance by a few percent. The figures for the 'mm_3by3' code are all calculated from unrolled code.

Some our our figures are higher than those obtained by the Wuppertal group using vector unit assembler code on the same CM-5 computer [102]. They achieve speeds of 1.8 Gflop/s on a single-precision $24^3 \times 48$ lattice. One reason for this is that the compilers have improved over the last few years to the point where it is as good to program the CM-5 in CM Fortran as it is in assembler. This is good news for our code portability; we do not have to sacrifice performance to gain portability if the compiler technology is as good as this.

Cray T3D performance

The Cray T3D is a MIMD computer, running our message-passing codes under PVM. Each node consists of two DEC-ALPHA processors running at 150 MHz with other hardware for communications. The peak speed of a node is 300 Mflop/s giving a peak speed for the platform of 38.4 Gflop/s (128 nodes or 256 processors). The machine can be operated in a large number of partition sizes. We use only two sizes, 8 processors and 16 processors. The software used to generate the following data was UNICOS version 8.0.2.1, UNICOS MAX version 1.1.0.2 and CF77 version 6.1. The hardware used was a T3D/MC256-8 with a Y-MP4E/264 front-end.

In tables 3.12 and 3.13 we present data for the two routines under consideration. The T3D uses 64-bit words throughout so there is no advantage in using single-precision; only double-precision data is presented.

Size	Prec.	12^4 lattice			16^4 lattice		
		Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
8	D	0.942e-2	0.22	18.2	0.297e-1	0.22	18.3
16	D	0.468e-2	0.44	18.3	0.148e-1	0.44	18.3

Table 3.12: Performance data for the '`mm_3by3`' routine on a Cray T3D computer. The T3D used to generate this data has a peak speed (128 nodes) of 38.4Gflop/s.

Size	Prec.	12^4 lattice			16^4 lattice		
		Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
8	D	0.642e-2	0.22	18.3	0.195e-1	0.22	18.5
16	D	0.318e-2	0.43	17.9			

Table 3.13: Performance data for the '`su3_hvv`' routine on a Cray T3D computer. The T3D used to generate this data has a peak speed (128 nodes) of 38.4Gflop/s.

Overall, efficiency of the code is low at 18% of peak. This is because of the slow memory access of the ALPHA processor; memory access takes 24 chip cycles as there is no pipe-lining. This is a feature that Cray aim to improve on with the T3E computer, but there is little that can be done at present. Because there is no padding of the arrays for the 12^4 lattice, performance is extremely steady. There is no improvement obtained from the differing memory access described above for '`su3_hvv`' and '`mm_3by3`' as the memory-access redundancy dominates timing.

3.7 Random numbers

One of the most controversial questions in lattice simulations is: 'which random number generator should I be using?' Monte Carlo simulations require extremely long period generators with low bit-level and lattice correlations. In this section we do not attempt to answer the question, but instead accept that the RNG used by the MPP codes must be easily changeable. In order to keep the code usable, however, we must maintain a standard interface to the RNGs in some way.

There are two types of RNG potentially used in the codes; lattice and single generators. When creating, for example, a local random gauge transformation we need a different random SU(3) matrix at each lattice site, hence the need for a lattice generator. If we wanted a global gauge transformation however, we only need a single SU(3) matrix which is then communicated to all points on the lattice for use. The distinction is not so great for the message-passing approach, where we can have the same random number generator running on all processors with different seeds.

In the HPF implementation we have several problems to contend with. Firstly, we do not want to have to run a single generator on the host processor and then loop over all lattice sites broadcasting numbers as this would be painfully slow. We do not have a portable access to the physical processors however, so we cannot run a single generator on each processor as is possible in message passing. This leaves us two options: we either run a single RNG per lattice site or use a machine-specific generator and sacrifice portability. The first option must be exercise with caution, the RANMAR [103] modified lagged-fibonacci generator used by previous UKQCD message-passing code requires a state table of 97 words. Since a quark propagator only requires 24 words per lattice site we can see that using RANMAR in this way is not always possible and certainly not practical. There are some generators which can be used in this way, for example a simple linear congruential multiplier generator [104, p.284] which only requires 1 word per lattice site, but they are not as likely to have a long period.

A further problem arises in both programming models: how do we initialise the generator over the lattice to reduce correlations? Obviously we need to give the generator a different seed on each processor or lattice site to extract distinct number sequences, but how do we guarantee that the sequences are not correlated in some way? This problem is more relevant in HPF when using a different generator on each lattice site. The essence of data-parallel programming is that all processors execute the same code at the same time; any correlations in the random number sequences at the beginning of the simulation will remain for the entirety of the simulation.

We do not have any hard and fast answers to these questions as they vary for different generators and become an in-depth research subject themselves. Instead we present a brief survey of the available literature for more information.

The theory of pseudorandom number generators is best obtained from historical work by Knuth [105] and Marsaglia [106, 97] or a more recent review by Vattulainen *et. al.* [98]. The various portable RNGs in use are discussed by James [107], Marsaglia [103] (one of the few generators with tests included for accuracy of implementation), Lüscher [108, 109] (implemented on the APE100 but available in portable Fortran), Cray Research [110] (their RANF generator relies on bit-level operations for portability, fine for HPF but not necessarily for all F77 implementations) and Vattulainen *et. al.* (a good review of popular generators). Tests which can be performed on RNGs are discussed by Vattulainen *et. al.* [111, 112, 100, 98] (a comprehensive selection from bit-level to Monte Carlo tests), Marsaglia [97], Coddington [99] and Ferrenberg [113].

Random number generators for parallel machines are discussed by Anderson [114], Deak [115] and Aluru *et. al.* [116].

Implementation of the module. Since we can pass array sections to subroutines in F77, the RNGs are easy to implement for message-passing systems. A single routine is needed which selects the required generator, either at build-time or run-time, and fills the offered array with random numbers. Because we cannot pass such array sections in HPF (due to restrictions from some compilers) we need a separate routine for each type of object we are filling with random numbers. Seeding the generators is best done through a single common block variable, used for all generators, so that the driver routine for the application does not need to be changed for each application. All of the RNG modules can be made to be self-initialising through use of '`SAVE`'ed variables.

3.8 Timing

We use only one timing routine, '`timer`', which conforms with the interface used for GENESIS [95]. The interface is

```
SUBROUTINE timer (seconds)
Dpoint seconds
```

i.e. the current time in seconds is returned. We only ever use the timer to measure time differences so the absolute value is never needed. The insides of this routine will need changing for different platforms.

Chapter 4

Generating quenched gauge configurations: the GAUGE application

4.1 Requirements

The pure gauge application, GAUGE, must be able to provide the following functionality (see figure 4.1)

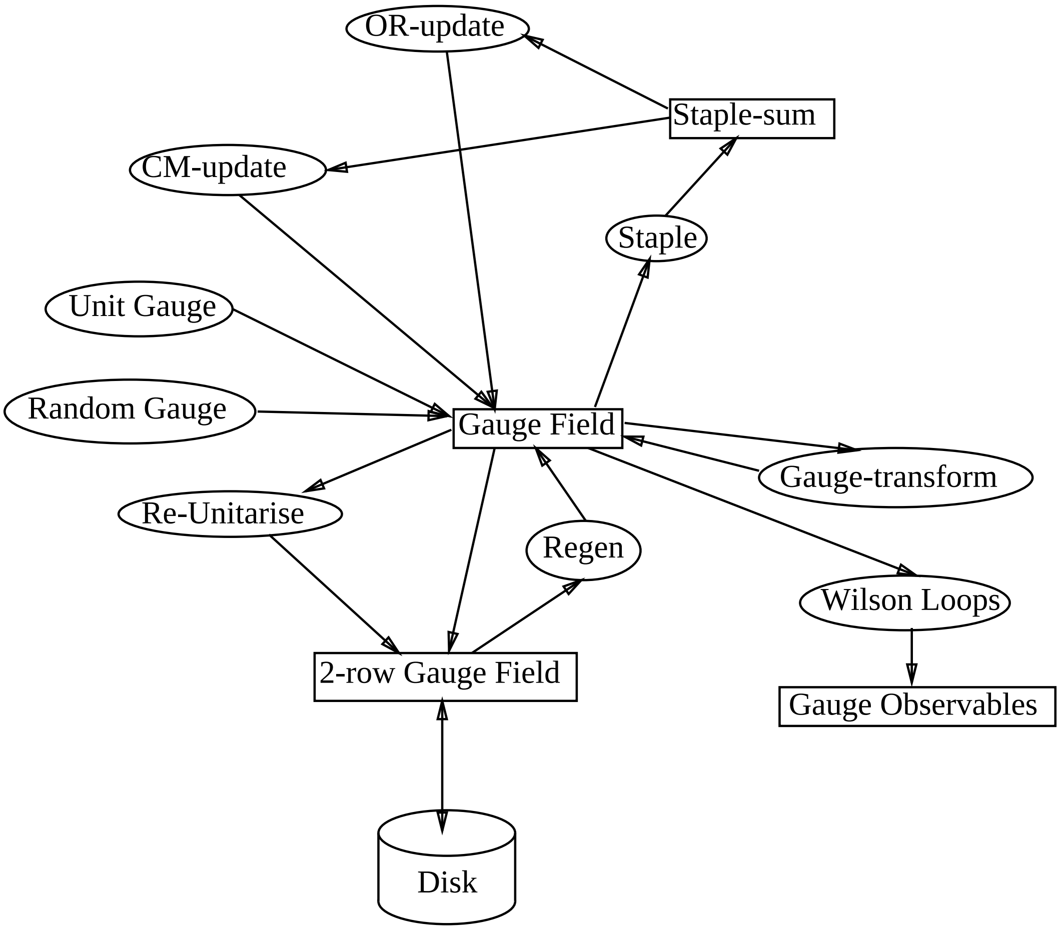


Figure 4.1: Functionality required from the GAUGE application.

1. **Initialisation of Markov chain**. As discussed in the theory section 1.3.5 we need to be able to initialise the gauge fields with either an ordered start (unit matrices), disordered start (random SU(3) matrices) or load a previously saved configuration.
2. **Update algorithm**. As specified in section 1.3.4. We require a local gauge transformation, heatbath and over-relaxed updates, and unitarisation of gauge fields (also useful for creating random SU(3) matrices in a disordered start).
3. **Input and output**. We need the ability to load and save gauge configurations in time-sliced form and the accompanying random number state information. To save space we store gauge fields in a two-row format (the first two rows) and reconstruct the third row on loading.

   There are three possible mechanisms for validating gauge configurations:

   (a) Calculating a 16-bit checksum on the binary data file. This provides information on the byte-ordering of the data file if required. It is not possible to calculate such a checksum on the Thinking Machines CM-200 or CM-5, but it is possible on the Cray T3D and workstations. The checksum is calculated to agree with the UNIX '`sum`' command.

   (b) Calculating a time-sliced plaquette average. This quantity can be used to detect the floating-point format of the gauge configuration (although most machines conform to IEEE standards now). We need a time-sliced average in case any analysis routines need to read in a single timeslice only, *e.g.* when smearing.

   (c) Calculating a plaquette average on the whole configuration. This is useful as it is only a single complex number and therefore easy to compare with a previously calculated value. The average on the whole configuration also tells us whether the time-slices have been read in the correct order.

There is no easy way to validate random number state information in general. Different random number generators have completely different sizes and type of state information, and it is not always accessible *e.g.* the `FAST_RNG` generator on the Connection Machine. The only way to check that the I/O works is to periodically check that restarting from a loaded configuration yields an identical plaquette average to what would have been obtained if updating had not been interrupted.

4. **Plaquette value saving**. We must be able to save plaquette values by plane to a separate file. The plaquette is a statistical quantity and can only be verified using a separate package; to extract them from the logfile in the correct format would be tedious. Information is given in section 4.3 about testing the plaquette values.
5. **Emergency termination**. We require a mechanism to cause the GAUGE application to terminate operation after the current compound update, having saved the configuration and random number information. The need for this function comes from running batch jobs: if we normally save every 200 compound sweeps for example, and the batch job is only going to generate 198 sweeps for some unusual reason, then we do not want to lose the hours of computer time used.

4.2 Design and implementation

The design of the GAUGE application can be easily seen to break into modules on top of underlying common layers (or libraries) as shown in figure 4.2. The important features of the design are discussed in order of the modules shown in the figure.

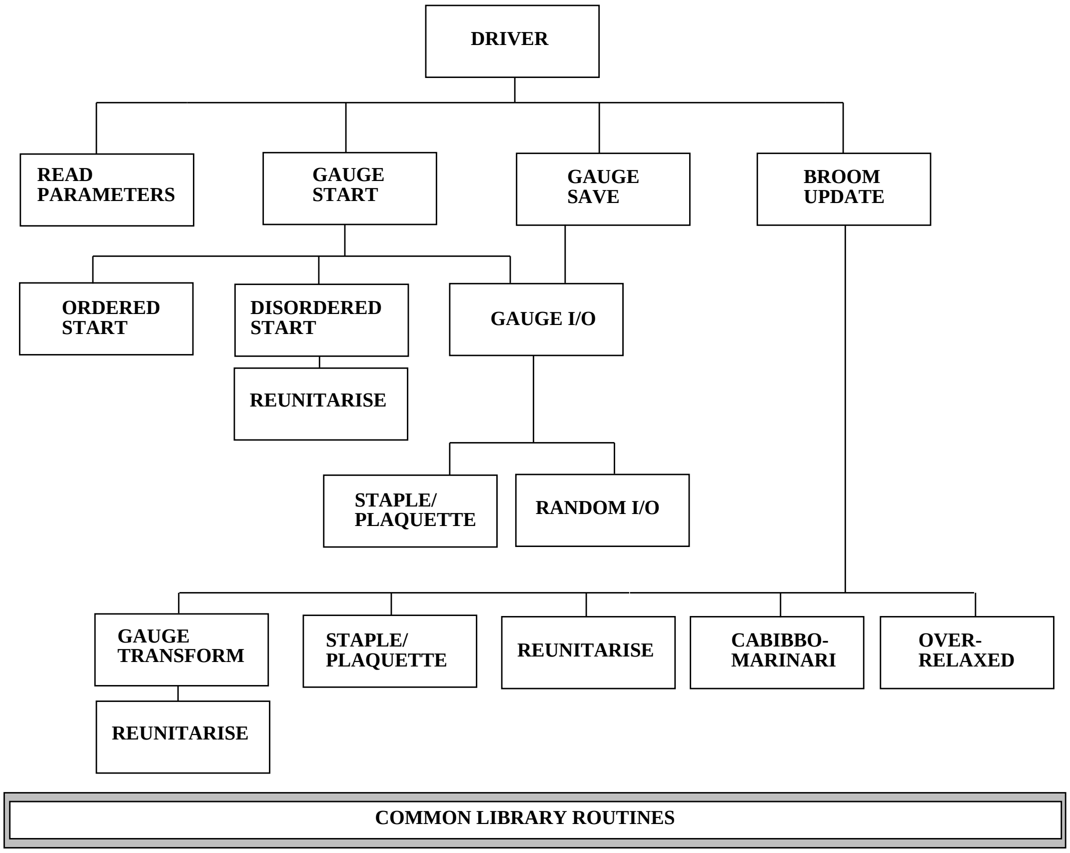


Figure 4.2: Overview of GAUGE design structure.

4.2.1 Read parameters

This section of the design is intended to be as sparse and simple as possible as it is only intended to convert parameters to a useful form, *e.g.* integer 0/1 to boolean `.TRUE.`/`.FALSE.`. Only vital parameters such as the lattice size or beta value are to be validated and displayed. Because Fortran, unlike C, does not provide a standard mechanism for accessing command line arguments, we use the shell to provide the 'run-name' on the standard input channel from which all filenames for I/O are derived.

A common block is used to implement the storage of all parameters. We do this as there are so many parameters in the list a function declaration would become impossibly long. These common block variables are not guaranteed to exist outside of 'DRIVER' and 'READ-PARAMETERS' in order to keep the design localised.

The I/O can be implemented in standard Fortran 77 using '`READ`' commands for both DP and MP approaches. The parameters read in by this module are identical for both programming environments since the processor decomposition, of relevance only for MP, is specified prior to execution of the application.

4.2.2 Disordered start

The easiest implementation of this function is to fill the first two rows of all gauge matrices with random numbers distributed uniformly in $[0, 1]$, then pass the result to 'REUNITARISE' to convert to SU(3).

4.2.3 Gauge I/O

We need to be able to load and save gauge fields in two-row time-sliced form with validation as described in section 4.1 using checksums and/or plaquette averages. All primitive I/O operations should be validated since high bandwidth data stores can be extremely unreliable. We require timing of the data rate for loading and saving of gauge fields as this can become an important statistic if checkpointing is performed often.

When saving a gauge configuration we also write an entry into the gauge 'application results file', which records the progress of GAUGE. Information written includes the version of GAUGE, sweep number and configuration validation data. This can be written in Fortran 77 and used for both MP and DP approaches without alteration.

Because there is stronger type checking available in HPF than F77 we write more subroutines, one for each data type. This implies that there is no generic-type I/O layer and a large GAUGE I/O layer for HPF, and *vice-versa* for MP. The implementation outline for both of the programming environments is shown in figure 4.3.

An operational issue raised by this module is: when is it safe to save a gauge configuration and guarantee reproducible results? Because we save in two-row form and reconstruct the third row using the 'reunitarise' module it is sensible only to save when we have performed a reunitarisation on the whole configuration. This is guaranteed by the 'driver' and 'broom update' modules which ensure that the last element of a compound update is a reunitarisation, and the configuration may only be saved at the end of such a compound update.

We use a machine-specific file format for parallel data storage as this can usually be implemented with a far higher data bandwidth. We require a separate utility, easily written in terms of the available library, to transfer files from this machine-specific format to a portable flat format.

Message-passing features. Because of the requirement for optimisation for vectorising compilers we need to be able to change the order of the indices for the gauge fields and hence the internal storage format. Since the external storage format needs to be held constant we have a set of packing/unpacking routines to perform conversion. These are discussed in more detail in section 3.2

HPF features. As can be seen in figure 4.3 we have isolated a set of routines underneath the heading '`gauge_parallel_save`'. This is done so that the HMC application can avoid duplicating code to save its gauge configurations. All

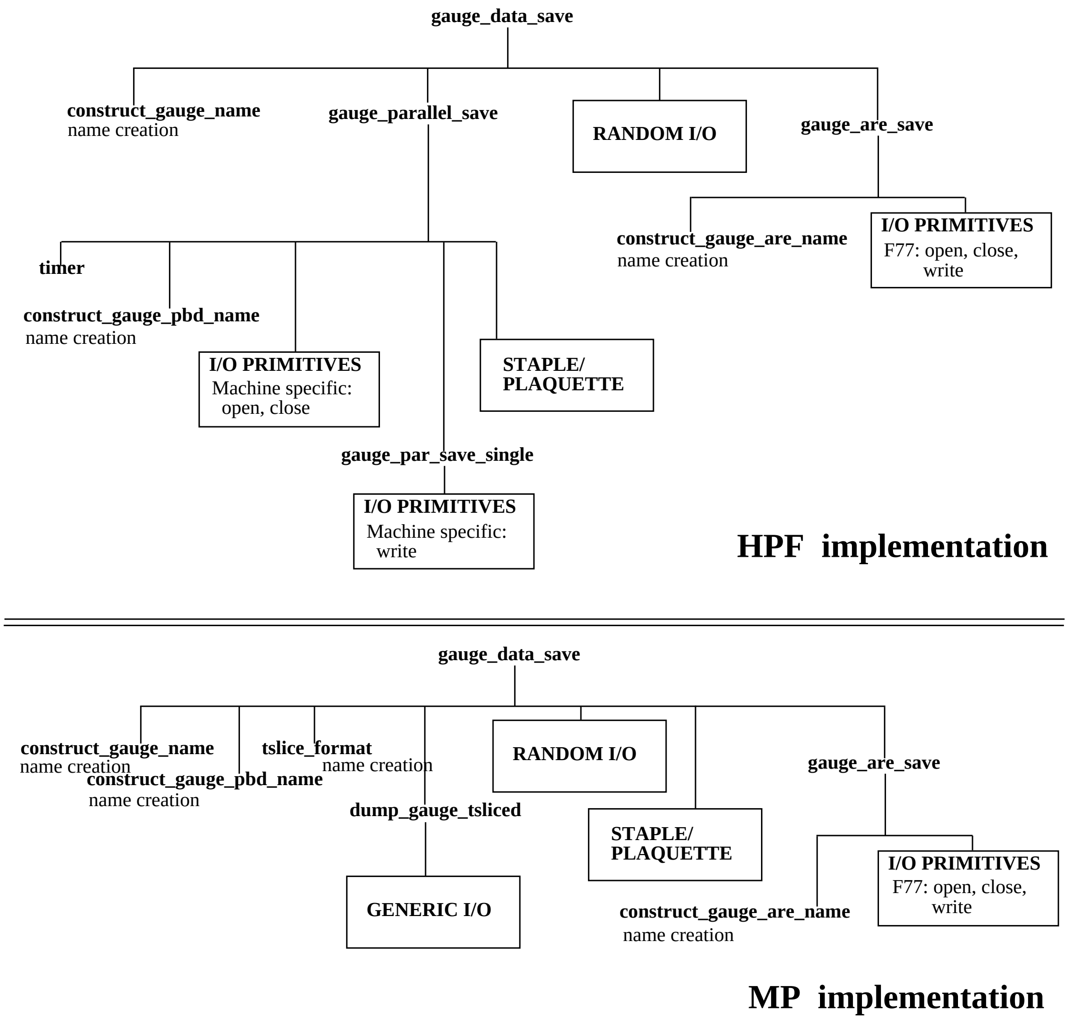


Figure 4.3: The structure of the gauge I/O routines. Routines labelled 'name creation' are implemented in F77 and used in identical form for both HPF and MP systems.

application specific code, *i.e.* filenames and ‘are’ file format, is executed above this heading. There is an option to swap row and column indices when loading and saving the gauge fields in HPF for backwards compatibility with earlier Connection Machine code. This is not required for MP as there are no flat configurations in that format.

Dataset names are kept as short as possible and reflect the important physics content. The gauge dataset name is of the format

$$\overbrace{\texttt{QbbUuuuuuu}}^{\text{name root}}\texttt{Ttt}$$

where ‘`Q`’ represents quenched, ‘`bb`’ is INT($\beta \times 10$), ‘Uuuuuuu’ is the elemental update number and ‘`Ttt`’ is the time-slice number. All numeric fields are zero padded.

4.2.4 Reunitarise

The theory of this module is discussed in section B.5 and the structure shown in figure 4.4. The structure shown is duplicated for routines to handle a single SU(3) matrix, rather than a lattice full, as is needed by the global gauge transform. The module is designed as a set of operations on row vectors so they can be re-used as necessary. Regeneration of gauge fields is performed by using ‘`cross_3vec`’ after loading in the two-row formatted fields.

4.2.5 Staple/plaquette

The staple and plaquette calculation, as discussed in section B.3, forms the core of the GAUGE update mechanism and contains all of the local communications used in the application. Although there are several stages to the algorithm, it is easily built in terms of the maths and communications layers defined previously. The call structure of the module is shown in figure 4.5. Note that

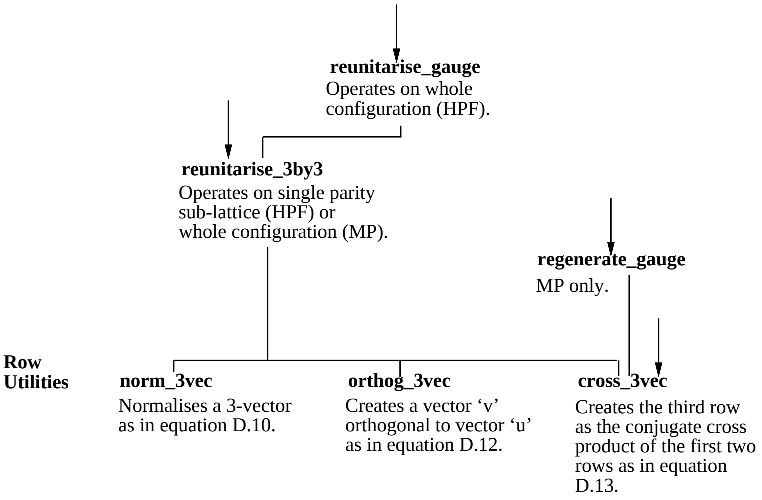


Figure 4.4: Structure of the reunitarise module.

the message-passing version is far more complex due to the need to explicitly start, wait for, and stop communications. This introduces the need for a wider range of maths routines to perform in-line gathers of communicated arrays. The implementation in both DP and MP models is shown below.

One feature of the design of the staple sum is that the plaquette can be calculated from a single parity in-line with the staples. The plaquette value thus obtained is therefore that at the *start* of the update, rather than at the end as would normally be calculated. However if the user does not mind this unorthodox method of presentation a large amount of time is saved from having to recalculate the staple explicitly for the plaquette.

HPF implementation. In HPF this algorithm is easily implemented as follows (routine `any_staple` in file `staple_gun.HPF`). We are working in the μ, ν plane; refer to figure 4.6 for labels used in the following discussion.

Top staple

1. Move 3 to A.

```
      CALL shift_3by3 (gauge_mask, u_mu_notpar,
     $  notpar, nu, Negative, temp1)
```

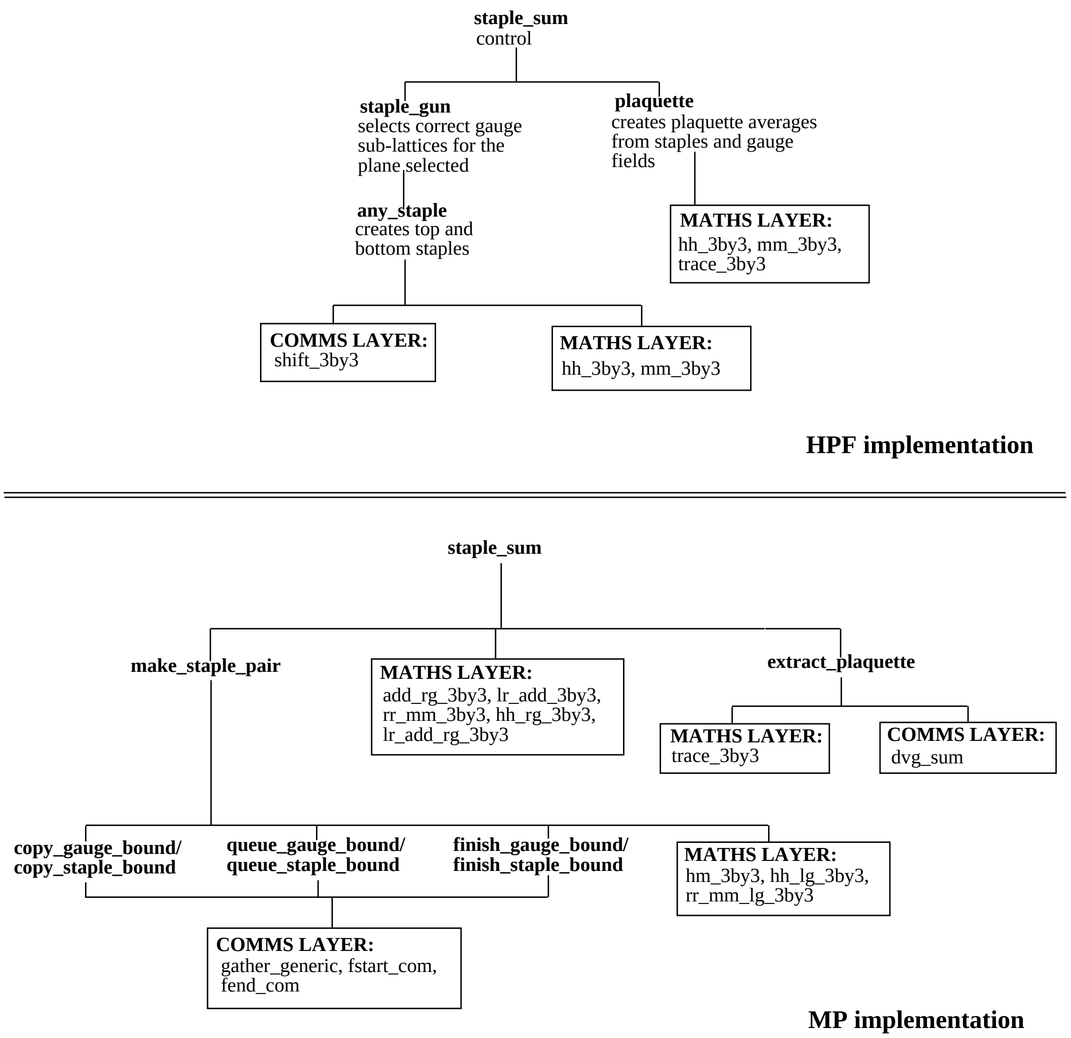


Figure 4.5: Call structure of the staple sum and plaquette module for both HPF and Message Passing implementations.

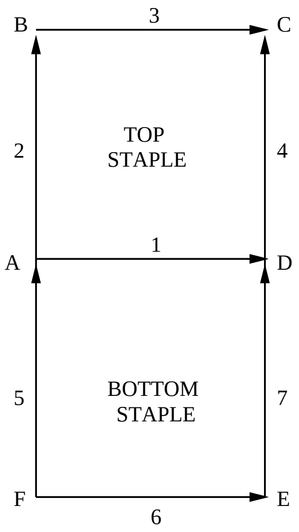


Figure 4.6: Labelling of links as used in the algorithmic description of the creation of the staples.

2. Multiply $3^\dagger \times 2^\dagger$.

    ```
         CALL hh_3by3 (temp1, u_nu_par, temp2)
    ```

3. Move 4 to A.

    ```
         CALL shift_3by3 (gauge_mask, u_nu_notpar,
        $  notpar, mu, Negative, staple_bottom)
    ```

4. Multiply $4 \times (3^\dagger \times 2^\dagger)$.

    ```
         CALL mm_3by3 (staple_bottom, temp2, staple_top)
    ```

Bottom staple

1. Move 7 to F.

    ```
         CALL shift_3by3 (gauge_mask,  u_nu_par,
        $  par, mu, Negative, temp2)
    ```

2. Multiply $7^\dagger \times 6^\dagger$.

    ```
         CALL hh_3by3 (temp2, u_mu_notpar, temp1)
    ```

3. Multiply $(7^\dagger \times 6^\dagger) \times 5$.

    ```
         CALL mm_3by3 (temp1, u_nu_notpar, temp2)
    ```

4. Move bottom staple to A.

```
      CALL shift_3by3 (gauge_mask, temp2, notpar,
     $  nu, Positive, staple_bottom)
```

MP implementation. Using the message-passing model we can overlap communications on one staple with calculation on the other as follows (routine `make_staple_pair` in file `staple_sum.F`).

1. Send 3 to A.

   ```
   C set up communications direction (-ve nu)
         comdir = nu + Ndim
   C perp is a perpendicular direction to the plaquette
   C plane, whose tail is used as workspace.
         CALL copy_gauge_bound(perp,mu,notpar,np_comm(nu),
        $          boundary_table(0,comdir,notpar),u)
         CALL queue_gauge_bound(comdir,perp,mu,notpar,n
        $          p_comm(nu),u)
   ```

2. Meanwhile, calculate $6^\dagger \times 5$.

   ```
         CALL hm_3by3(np_sites,
        $         Max_array, u(0,0,0,0,notpar,mu),
        $         Max_array, u(0,0,0,0,notpar,nu),
        $         Max_array, down_staple)
   ```

3. Wait for 3 to finish sending, store the communication number for the multiply.

   ```
         CALL finish_gauge_bound(comdir,perp,mu,
        $     notpar,np_comm(nu),u)
         lastcom = comdir
   ```

4. Send 7 to F.

   ```
         comdir = mu + Ndim
         CALL copy_gauge_bound(perp,nu,par,np_comm(mu),
        $          boundary_table(0,comdir,par),u)
         CALL queue_gauge_bound(comdir,perp,nu,par,
        $          np_comm(mu),u)
   ```

5. Meanwhile calculate $3^\dagger \times 2^\dagger$, gathering 3 as needed.

   ```
         CALL hh_lg_3by3(np_sites,
        $         shift_table(0,lastcom,notpar),
        $         Max_array, u(0,0,0,0,notpar,mu),
        $         Max_array, u(0,0,0,0,par,nu),
   ```

```
     $          Max_body, up_staple)
```

6. Wait for 7 to finish sending, store the communication number for next multiply.

```
      CALL finish_gauge_bound(comdir,perp,nu,
     $     par,np_comm(mu),u)
      lastcom = comdir
```

7. Send 4 to A.

```
      comdir = mu + Ndim
      CALL copy_gauge_bound(perp,nu,notpar,np_comm(mu),
     $           boundary_table(0,comdir,notpar),u)
      CALL queue_gauge_bound(comdir,perp,nu,notpar,
     $           np_comm(mu),u)
```

8. Meanwhile multiply $7^\dagger \times (6^\dagger \times 5)$, gathering 7 as needed.

```
      CALL rr_hm_lg_3by3(np_sites,
     $          shift_table(0,lastcom,par),
     $          Max_array, u(0,0,0,0,par,nu),
     $          Max_array, down_staple)
```

9. Wait for 4 to finish sending.

```
      CALL finish_gauge_bound(comdir,perp,nu,
     $     notpar,np_comm(mu),u)
      lastcom = comdir
```

10. Send lower staple to A.

```
      comdir = nu
      CALL copy_staple_bound(np_comm(nu),
     $   boundary_table(0,comdir,notpar),
     $   u(0,0,0,0,notpar,perp), down_staple)
      CALL queue_staple_bound(np_comm(nu),comdir,
     $   u(0,0,0,0,notpar,perp),down_staple)
```

11. Meanwhile calculate upper staple $= 4 \times (3^\dagger \times 2^\dagger)$, gathering 4 as needed.

```
      CALL rr_mm_lg_3by3(np_sites,
     $          shift_table(0,lastcom,notpar),
     $          Max_array, u(0,0,0,0,notpar,nu),
     $          Max_body, up_staple)
```

12. Wait for lower staple to finish sending. Leave the routine which uses the staples to gather in.

```
           CALL finish_staple_bound(np_comm(nu),comdir,
          $   u(0,0,0,0,notpar,perp),down_staple)
```

4.2.6 Random I/O

The random number I/O is relegated to a separate module from gauge I/O as there could be several optional RNGs requiring different I/O handling. If we are using a machine-specific RNG supplied through a library, *e.g.* 'FAST_RNG' on the Connection Machine, we may have very little control over the I/O, having to use a couple of supplied functions. If we are using a portable RNG, the state information data structures still vary enormously. For this reason we provide a separate implementation of the I/O for each random number generator used.

HPF implementation note. The 'FAST_RNG' generator mentioned above introduces further subtle problems. In our testing of the HPF codes on the Connection Machine this has been our chosen generator, for reasons discussed in section 3.7, but using a larger state table than the default for better performance. As we are not using the default sizes we must initialise the generator *before* loading the old state tables so the sizes are set correctly as shown in the following code fragment, otherwise the library assumes you want the default sizes and gives non-reproducible results. This subtle bug took a long time to track down! We advise use of portable generators where possible to avoid such problems.

```
C initialise the generator with our state table size
      CALL cmf_lattice_init_rng ()

C 'path' contains the full path to the saved RNG
C state information. Open the file.
      CALL CMF_FILE_OPEN (in_unit, path, ios)
      IF (ios .LT. 0) THEN
        WRITE (mess_buff,20) in_unit,ios, path
20      FORMAT ('Unit : ',I2,' ios : ',I3,' Path : ',A)
        CALL status_message (mess_buff, 'cmf_rng_load')
```

```
        CALL error_message ('Error opening file.',
     $    'cmf_rng_load', Err_file_error)
      END IF

C seek to the beginning of the file and read in the
C data.
      CALL CMF_FILE_REWIND (in_unit,ios)
      CALL RESTORE_FAST_RNG_TEMPS (in_unit,ios,ier)
      IF (ier .NE. 0) THEN
        WRITE (mess_buff,30) ios, ier, path
30      FORMAT ('ios : ',I3,' ier : ',I3,' path : ',A)
        CALL status_message (mess_buff, 'cmf_rng_load')
        CALL error_message ('Error opening/reading file.',
     $    'cmf_rng_load', Err_file_error)
      ELSE
        WRITE (mess_buff,462) ios
462     FORMAT ('RESTORE_FAST_RNG_TEMPS read ',I10,' bytes')
        CALL status_message (mess_buff, 'cmf_rng_load')
      END IF

C close the file.
      CALL CMF_FILE_CLOSE (in_unit, ios)
      IF (ios .LT. 0) THEN
        WRITE (mess_buff,20) in_unit,ios,path
        CALL status_message (mess_buff, 'cmf_rng_load')
        CALL error_message ('Error closing file.',
     $    'cmf_rng_load', Err_file_error)
      END IF
```

The initialisation is performed by

```
C   Need to set up the weedy random number generator first to
C   put values in the state tables for fast_rng. Important
C   that the seed
C   for weedy random number generator is reproducible.
        dummy = RAND (rng_seed)
        CALL CMF_RANDOMIZE(rng_seed)

C   Now set up the fast rng.
        CALL INITIALIZE_FAST_RNG(
     $    cmf_rng_table_lag, cmf_rng_short_lag,
     $    cmf_rng_width, error_code)
        init_cmf_rng = .TRUE.
```

4.2.7 Gauge transform: $U_\mu(x) \rightarrow V(x)U_\mu(x)V^\dagger(x+\hat{\mu})$

As with the staple sum, this is easy to implement in terms of the maths and communications layers. In HPF this would be written

```
C Even parity sub-lattice
        CALL shift_3by3 (gauge_mask, trans_odd, Odd_parity,
     $    mu, Negative, temp1)
        CALL mh_3by3 (U_mu_evn, temp1, temp)
        CALL mm_3by3 (V_evn, temp, U_mu_evn)

C Odd parity sub-lattice
        CALL shift_3by3 (gauge_mask, V_evn, Even_parity,
     $    mu, Negative, temp1)
        CALL mh_3by3 (U_mu_odd, temp1, temp)
        CALL mm_3by3 (V_odd, temp, U_mu_odd)
```

while in MP we would write

```
C copy low bound of v(notpar) to v(par) tail.
C 'dir' is the correct communications direction.
      CALL copy_t_bound(par, np_comm(mu),
     $  boundary_table(0,dir,notpar) , v)

C queue send in -ve mu dir  v(par) tail -> v(notpar) tail
      CALL queue_t_bound(dir, par, np_comm(mu), v)

C u(x) = v(x) u(x)
      CALL rr_mm_3by3(np_sites,
     $  Max_array, v(0,0,0,0,par), Max_array,
     $  u(0,0,0,0,par,mu))

C finish send
      CALL finish_t_bound(dir, par, np_comm(mu), v)

C u(x) = u(x) v (x+mu)
      CALL lr_mh_rg_3by3(np_sites,
     $  shift_table(0,dir,notpar), Max_array,
     $  u(0,0,0,0,par,mu), Max_array,
     $  v(0,0,0,0,notpar))
```

4.2.8 Cabibbo-Marinari update

There are two points of interest in this module.

1. The creation of the random a_0 lattice, see theory in section B.1.2, can be performed using only two temporary vectors as shown in the following HPF code.

```
C generate R
      CALL lattice_fpoint_rng (rng_get_routine,
     $  epsilon, temp1)
C generate R'
      CALL lattice_fpoint_rng (rng_get_routine,
     $  epsilon, temp2)
C X=-ln(R)*alphainv
      temp1=-LOG(temp1)*alphainv
C C = cos^2 (2*PI*R')
      temp2=COS(two_pi*temp2)
      temp2=temp2*temp2
C A=X*C
      temp1 = temp1 * temp2
C generate R'
      CALL lattice_fpoint_rng (rng_get_routine,
     $  epsilon, temp2)
C X'=-ln(R')*alphainv
      temp2 =-LOG(temp2)*alphainv
C delta = X'+A
      temp1 = temp1 + temp2
C generate R''^2
      CALL lattice_fpoint_rng (rng_get_routine,
     $  epsilon, temp2)
      temp2 = temp2 * temp2
C T = 1-delta/2
      temp1 = 1.0-(temp1*0.5)
```

2. When implementing the local Metropolis accept/reject stage in HPF it is inefficient to loop over the lattice sites as this would happen on the front-end processor causing a bottleneck to occur. Instead we create whole lattices of random numbers at a time and insert new numbers to pass the accept/reject where failures occur using the '`ANY`' and '`WHERE`' intrinsics.

```
C  Generate the first arrays of random numbers in
C  ‘dest’ and ‘compare’.
      CALL test_a0 (rng_get_routine,
     $  alphainv, dest, compare)

C While any sites fail the test generate fresh test
C numbers.
      DO WHILE (ANY (compare .GT. dest))

C We need to setup array everywhere for reject because WHERE
C can only have assignment operations inside clause, no
C function calls.
        CALL test_a0 (rng_get_routine, alphainv,
     $    temp_a0, temp_compare)

C Replace the failing numbers
        WHERE (compare .GT. dest)
          compare = temp_compare
          dest = temp_a0
        END WHERE

C Keep going until all pass.
      END DO
```

4.3 Gauge testing

4.3.1 Over-relaxation testing

The over-relaxation algorithm, as described in sections 1.3.2 and B.2, should preserve the value of $\left\langle \frac{1}{3}\mathrm{ReTr}\ U_{\Box\mu\nu} \right\rangle$ (averaged over all planes μ, ν). This can be seen clearly in real plaquette data as shown in figure 4.7. Of course, if the over-relaxed algorithm does nothing at all the same output will be obtained, so we must examine other quantities as well: the imaginary part of the plaquette should be randomly distributed around zero as shown in figure 4.8 and the values of the real part of the plaquette for single planes should fluctuate as shown in figure 4.9.

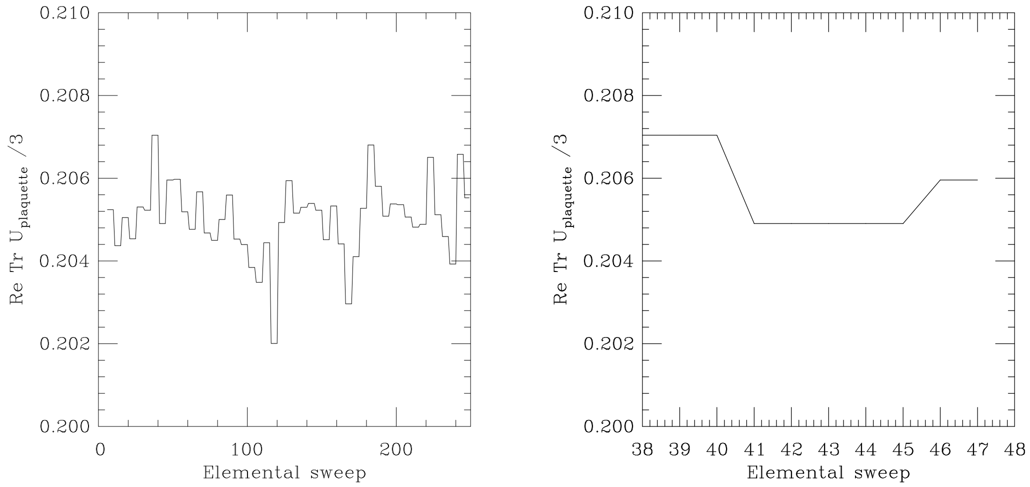


Figure 4.7: Real part of plaquette averaged over all planes. This data was obtained from a Thinking Machines CM-5 using one Cabibbo-Marinari (heatbath) elemental sweep and four over-relaxed elemental sweeps per compound sweep with $\beta = 3.0$. The plaquette is conserved between over-relaxed sweeps as can be seen in the expanded graph on the right hand side.

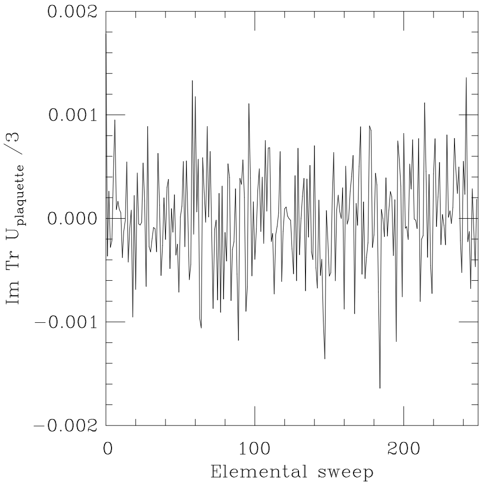


Figure 4.8: Imaginary part of plaquette averaged over all planes. This data is from the same run as that in figure 4.7. The plaquette is *not* conserved between over-relaxed sweeps.

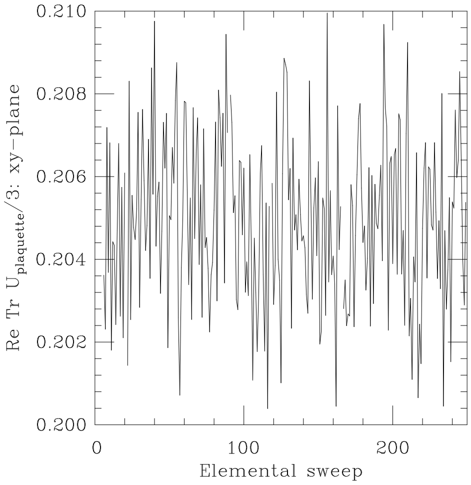


Figure 4.9: Real part of plaquette for the xy-plane only. This data is from the same run as that in figure 4.7. Note that the plaquette for the single plane is *not* conserved between over-relaxed sweeps.

4.3.2 Heatbath testing

The Cabibbo-Marinari heatbath updates are intrinsically difficult to test as they use random numbers to create the required distribution $P(a_0)da_0$ as shown in sections B.1 and B.1.2. We can however easily test that the distribution generated is correct for a range of parameters. This test is shown in figure 4.10.

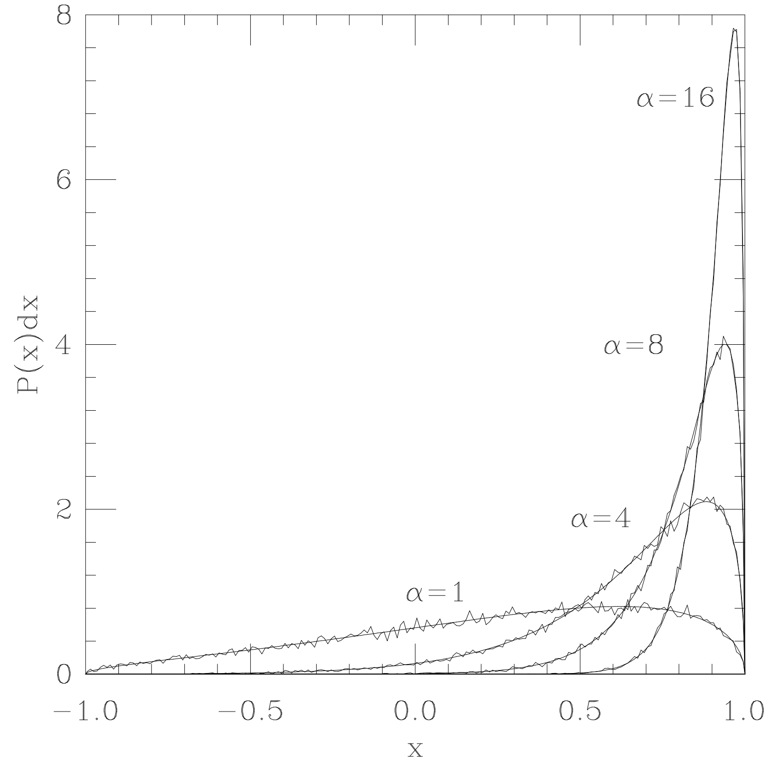


Figure 4.10: Random number distribution for Cabibbo-Marinari heatbath update. The distribution shown is $P(x) = N^{-1}\sqrt{1-x^2}e^{\alpha x}$, where the normalisation factor $N = \frac{\pi}{\alpha}I_1(\alpha)$. Both the ideal distribution and one generated by the MPP codes are shown.

To test the rest of the heatbath algorithm, we need to examine the plaquette values as a function of β. To find data to compare with we examine the strong coupling expansion, for low β, and data produced by Lepage and Mackenzie [10] for high β. Note that we do not expect to agree exactly with the figures quoted because of differences in lattice size and algorithmic parameters, the plaquette is a statistical quantity which will fluctuate. However the plaquette values should agree within errors over the whole range of test data from $\beta = 3.0$ to $\beta = 9.0$. If we get good agreement over this wide range we can be certain that the code is working.

4.3.3 Strong coupling expansion: validation at low β

The strong coupling expansion is a power series in β valid for high values of the coupling g, hence low values of $\beta = \frac{6}{g^2}$. The coefficients in the series for the free energy have been presented by Edgar et. al. [117] ($\mathcal{O}(\beta^6)$) and Balian et. al. [118, 119] ($\mathcal{O}(\beta^{16})$) from summation of graphs as

$$\begin{aligned}
\frac{F}{d(d-1)} &= = \frac{1}{2}\overline{\beta}^2 + \frac{1}{6}\overline{\beta}^3 - \frac{1}{24}\overline{\beta}^5 + (\frac{1}{243}d - \frac{113}{3888})\overline{\beta}^6 + (\frac{1}{81}d - \frac{133}{6480})\overline{\beta}^7 \\
&+ (\frac{5}{324}d - \frac{1069}{51840})\overline{\beta}^8 + (\frac{5}{972}d - \frac{509}{77760})\overline{\beta}^9 \\
&+ (\frac{2}{6561}d^2 - \frac{157}{11664}d + \frac{490757}{20995200})\overline{\beta}^{10} \\
&+ (\frac{4}{2187}d^2 - \frac{59}{2160}d + \frac{435299}{9797760})\overline{\beta}^{11} \\
&+ (\frac{1775}{354294}d^2 - \frac{218824907}{7255941120}d + \frac{1682010779}{42326323200})\overline{\beta}^{12} \\
&+ (\frac{440}{59049}d^2 - \frac{13919677}{604661760}d + \frac{7603159}{440899200})\overline{\beta}^{13} \\
&+ (\frac{20}{531441}d^3 + \frac{8377}{2125764}d^2 - \frac{12469727}{5441955840}d - \frac{14239256399}{1333279180800})\overline{\beta}^{14} \\
&+ (\frac{544}{1594323}d^3 - \frac{69331}{7971615}d^2 + \frac{106962409}{2821754880}d - \frac{3474317893}{79361856000})\overline{\beta}^{15} \\
&+ (\frac{2323}{1594323}d^3 - \frac{5838272899}{220399211520}d^2 + \frac{10597782658021}{123423558451200}d \\
&- \frac{6402970751747}{82282372300800})\overline{\beta}^{16} + O(\overline{\beta}^{17}) \qquad (4.1)
\end{aligned}$$

in d dimensions, where $\overline{\beta} = \frac{1}{g^2}$. By making the substitution $\beta = 6\overline{\beta}$, we can calculate the average value of the plaquette in 4 dimensions to be

$$\left\langle \frac{1}{3}\mathrm{ReTr}\ U_{\Box\mu\nu} \right\rangle = \frac{1}{6}\frac{dF(\beta)}{d\beta}$$

$$\begin{aligned}
&= \frac{1}{18}\beta + \frac{1}{216}\beta^2 - \frac{5}{93312}\beta^4 - \frac{49}{15116544}\beta^5 \\
&+ \frac{1309}{906992640}\beta^6 + \frac{2131}{5441955840}\beta^7 + \frac{1091}{43535646720}\beta^8 \\
&- \frac{179081}{21158324305920}\beta^9 - \frac{1277749}{592433080565760}\beta^{10} \\
&- \frac{93151153}{115168990861983744 00}\beta^{11} + \frac{3052831769}{34550697258595123200}\beta^{12} \\
&+ \frac{6757393949}{41460836710314147840 0}\beta^{13} \\
&- \frac{1932793007}{3198407403367091404800}\beta^{14} \\
&- \frac{16029793987553}{217619639725096899182592 00}\beta^{15} + O(\beta^{16}) \qquad (4.2)
\end{aligned}$$

This expansion has been plotted in figure 4.11 for various orders in order to establish a threshold for the accuracy of the expansion. By comparing measured plaquette values against those obtained from the expansion, we can validate the software at low values of β. Suggested β values and plaquette averages are shown in table 4.1.

β	$\left\langle \frac{1}{3}\mathrm{ReTr}\ U_{\square_{\mu\nu}} \right\rangle$
2.0	0.12881138(1)
2.5	0.1659980(4)
3.0	0.205047(5)
3.5	0.24629(5)
4.0	0.2907(4)

Table 4.1: Plaquette averages from the strong coupling expansion for several β values. The values presented are calculated from the average of the 14th and 15th order expansions.

Plaquette data measured on different platforms with various algorithms are shown in figure 4.12. We plot both symmetric and squashed lattice data for comparison. The squashing should raise the effective β value and hence the plaquette average over the squashed planes. This gives us information on whether the individual directions are treated correctly in the code. The x-axis values have the following meanings

A: Reference data from the strong coupling expansion.

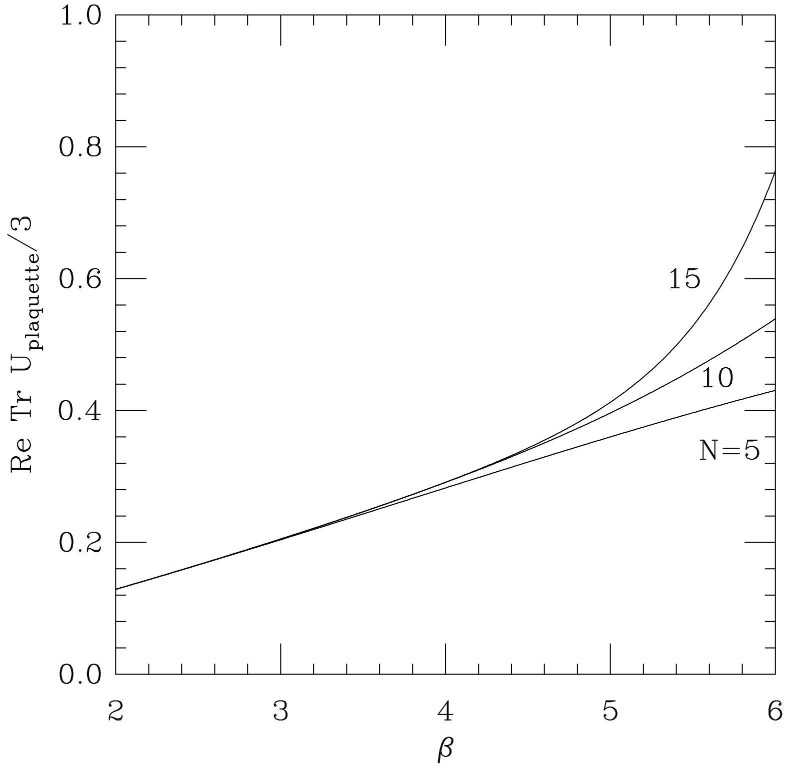


Figure 4.11: The strong coupling expansion of the average plaquette plotted against $\beta = \frac{6}{g^2}$, where g^2 is the coupling constant for QCD. Plots are shown for the expansions up to order β^{N-1}. It can be seen that the 10th and 15th order expansions are only valid up to $\beta \simeq 4.0$.

B: Data from the MPP codes with a symmetric lattice.

C-F: Data from the MPP codes on various platforms (both message-passing and data-parallel) squashed in the x, y, z and t directions respectively.

The crosses show data on a $16^3 \times 2$ lattice averaged over those planes containing the squashed axis. The diamonds are data on the same lattice averaged over planes not containing the squashed axis. The bursts show data on a $16^3 \times 4$ lattice averaged over squashed planes, and the squares show the unsquashed planes for the same lattice.

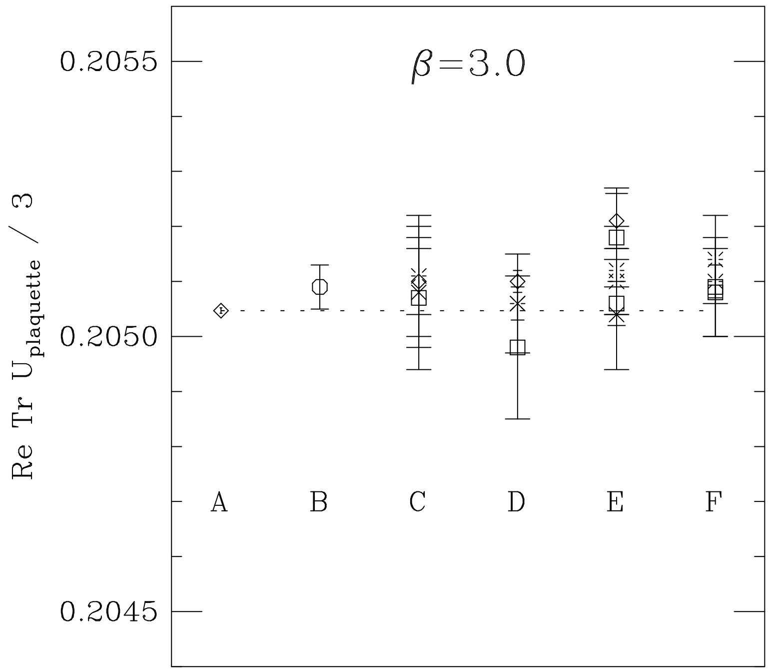
The data for $\beta = 3.0$ is not affected by the squashed lattices as would be expected, whereas for $\beta = 4.0$ the squashing is changing the effective β value. The deviation of the MPP data for $\beta = 4.0$ from the strong coupling expansion could be caused by the breakdown of the expansion at that β value. The 15th order expansion shown in figure 4.11 is higher than the 10th order so this could explain why the theoretical value shown is higher than the Monte Carlo data.

4.3.4 Lepage and Mackenzie data

In [10] Lepage and Mackenzie discuss lattice perturbation theory and present Monte Carlo plaquette data to compare with their coupling constants. Their data is presented in table 4.2.

The data from the MPP codes running both message-passing and data-parallel on different platforms is presented in figures 4.13 and 4.14. The x-axis is explained in the previous section.

We can see that the symmetric lattice data from the MPP codes agrees with the data from Lepage and Mackenzie (both were measured on 16^4 lattices). On asymmetric lattices the plaquette values for squashed planes are higher than the unsquashed planes. The size of this difference increases with β and the



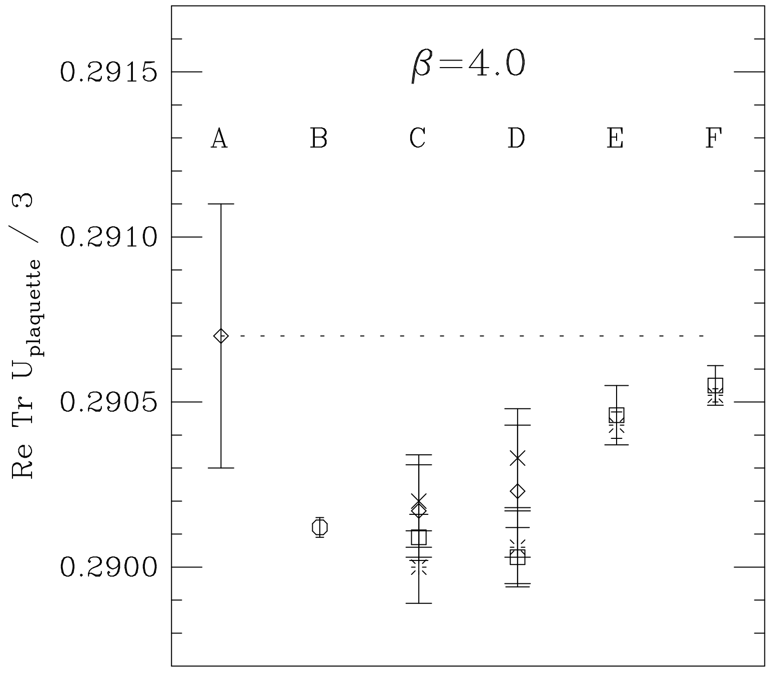
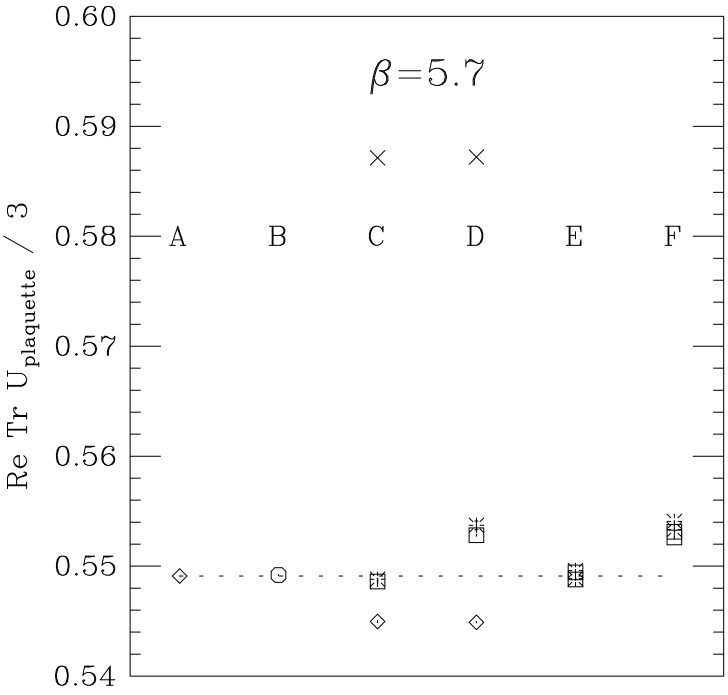


Figure 4.12: Plaquette data for $\beta = 3.0$ and $\beta = 4.0$. See text for explanation. The data on these graphs was generated with differing update parameters on different platforms; the results are consistent within errors indicating that the code is working properly.



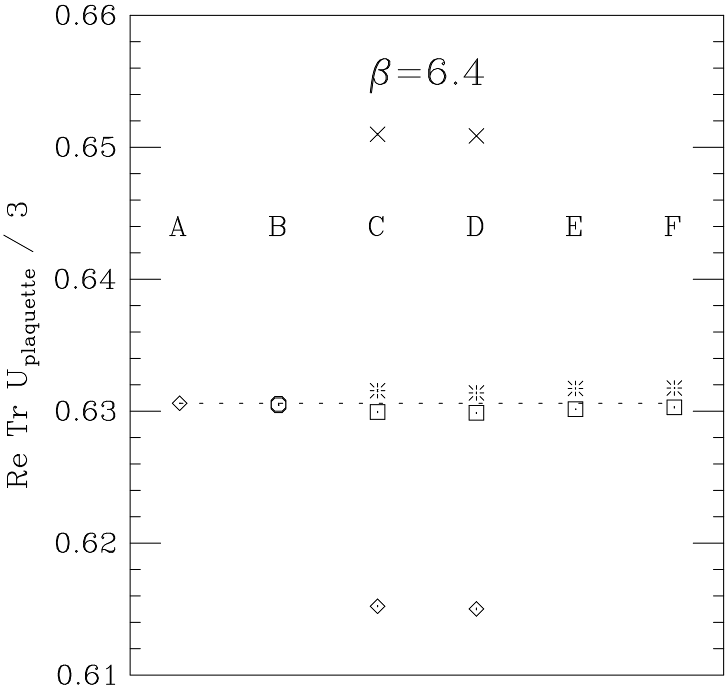


Figure 4.13: Plaquette data for β of 5.7 and 6.4. See text for explanation.

β	$\ln \frac{1}{3} \langle \mathrm{ReTr}\ U_{\square\mu\nu} \rangle$	$\frac{1}{3} \langle \mathrm{ReTr}\ U_{\square\mu\nu} \rangle$
5.7	0.5995	0.5491
6.0	0.5214	0.5937
6.1	0.5025	0.6050
6.2	0.4884	0.6136
6.3	0.4740	0.6225
6.4	0.4610	0.6306
9.0	0.2795	0.7562
12	0.1954	0.8225
18	0.1224	0.8848

Table 4.2: Lepage and Mackenzie plaquette data. This was calculated on a 16^4 lattice.

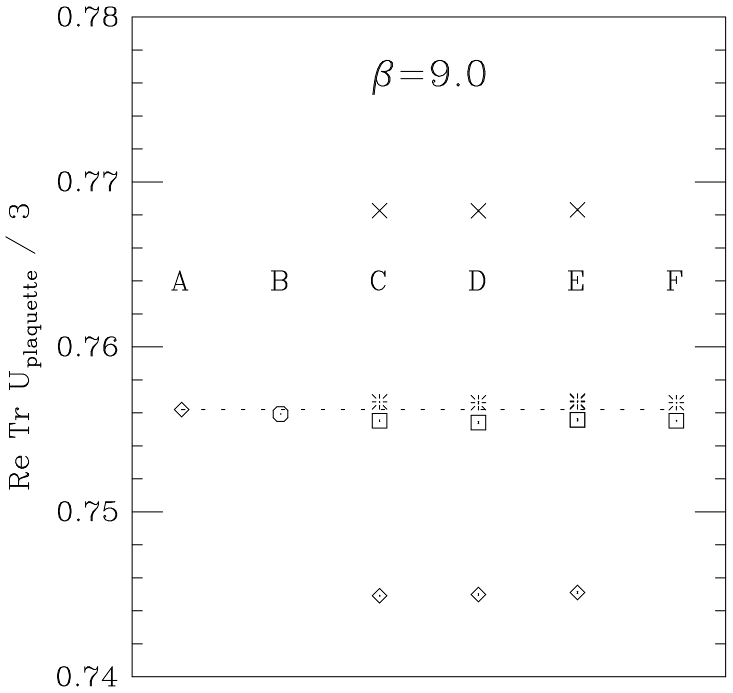


Figure 4.14: Plaquette data for β of 9.0. See text for explanation.

extent of the squashing as expected.

Our simulation data agrees well with Lepage and Mackenzie's numbers for the unsquashed lattices at all β values, indicating that the code is working properly. The squashed lattice data shows that the expected planes receive the higher plaquette average so we know that the different directions are treated correctly.

4.4 Gauge performance

The performance of the GAUGE application is of great interest as it is used for long periods of time to generate gauge configurations. Slight increases in the speed of code can result in a few more configurations being generated. In this section we present data for the timing and efficiency of different elements of the update algorithm on the three platforms discussed in section 3.6; the Cray T3D, Connection Machine CM200 and Connection Machine CM5. We generated the data at a β value of 5.9 with a disordered start on 12^4 and 16^4 lattices.

Random Local Gauge Transform								
			12^4 lattice			16^4 lattice		
Platform	Size	Precision	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
CM200	8K	D	0.44	0.084	2.1	0.405	0.289	7.3
CM200	8K	S	0.25	0.150	3.8			
CM5	16	D	0.079	0.466	18.2	0.229	0.511	20.0
CM5	16	S				0.163	0.718	28.0
CM5	32	D	0.050	0.741	14.5	0.122	0.957	18.7
T3D	8	D	0.639	0.058	4.8	1.05	0.111	9.3
T3D	16	D	0.545	0.068	2.8	0.752	0.156	6.5

Table 4.3:

Cabibbo-Marinari update								
			12^4 lattice			16^4 lattice		
Platform	Size	Precision	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
CM200	8K	D	5.38	0.0691	1.8	5.64	0.209	5.3
CM200	8K	S	3.26	0.114	2.9			
CM5	16	D	1.01	0.368	14.4	2.97	0.396	15.5
CM5	16	S				2.19	0.537	21
CM5	32	D	0.64	0.584	11.4	1.61	0.729	14.2
T3D	8	D	3.77	0.099	8.3	10.1	0.116	9.7
T3D	16	D	2.32	0.160	6.7	5.52	0.213	8.9

Table 4.4:

Observations about scaling performance with lattice size, machine size and precision have already been made in section 3.6, they hold for these results as well.

Over-relaxed update								
			12^4 lattice			16^4 lattice		
Platform	Size	Precision	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
CM200	8K	D	4.53	0.0686	1.7	3.89	0.252	6.3
CM200	8K	S	2.47	0.126	3.2			
CM5	16	D	0.667	0.466	18.2	2.03	0.483	18.9
CM5	16	S				1.36	0.720	28.1
CM5	32	D	0.413	0.753	14.7	1.09	0.898	17.5
T3D	8	D	2.17	0.143	11.9	6.74	0.146	12.2
T3D	16	D	1.13	0.275	11.5	3.43	0.286	11.9

Table 4.5:

Reunitarise								
			12^4 lattice			16^4 lattice		
Platform	Size	Precision	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
CM200	8K	D	0.0639	0.203	5.1	0.108	0.370	9.3
CM200	8K	S	0.0472	0.275	6.9			
CM5	16	D	0.0224	0.580	22.7	0.0651	0.614	24.0
CM5	16	S				0.0627	0.638	24.9
CM5	32	D	0.0123	1.057	20.6	0.0352	1.136	22.2
T3D	8	D	0.159	0.0818	6.8	0.502	0.080	6.7
T3D	16	D	0.0796	0.163	6.8	0.251	0.159	6.6

Table 4.6:

Compound update (GT+2CM+2OR+RE)								
			12^4 lattice			16^4 lattice		
Platform	Size	Precision	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
CM200	8K	D	21.4	0.0660	1.7	20.2	0.221	5.6
CM200	8K	S	12.18	0.115	2.9			
CM5	16	D	3.86	0.366	14.3	10.78	0.414	16.2
CM5	16	S				7.81	0.572	22.3
CM5	32	D	2.73	0.524	10.2	5.99	0.745	14.6
T3D	8	D	13.1	0.0992	8.3	35.7	0.115	9.6
T3D	16	D	7.92	0.164	6.8	19.3	0.213	8.9

Table 4.7:

There are several notable features in the performance data shown. The Cabibbo-Marinari update is slower than either the gauge transform or over-relaxed update on the CM200 and CM5. The reason for this is the need to generate whole lattices of random numbers to implement the accept/reject stage as discussed earlier in this chapter. On the T3D the message-passing code need only generate single random numbers; a far more efficient process. The over-relaxed update stands out as the most efficient routine for the T3D. The most likely reason for this is that it is the only routine which does not involve maths functions (*e.g.* SIN, COS, LOG, SQRT) to a high degree.

Note that the Cabibbo-Marinari update performance dominates that of the compound update. This is because over 50% of the time is spent performing the Cabibbo-Marinari update. This will not necessarily be the case in production, *e.g.* a compound update consisting of 1 Cabibbo-Marinari update and 5 over-relaxed updates has been used over the last few years by UKQCD.

Chapter 5

Generation of quark propagators: the SOLVER application

5.1 Requirements

The solver application, SOLVER, must be able to provide the following functionality (see figure 5.1).

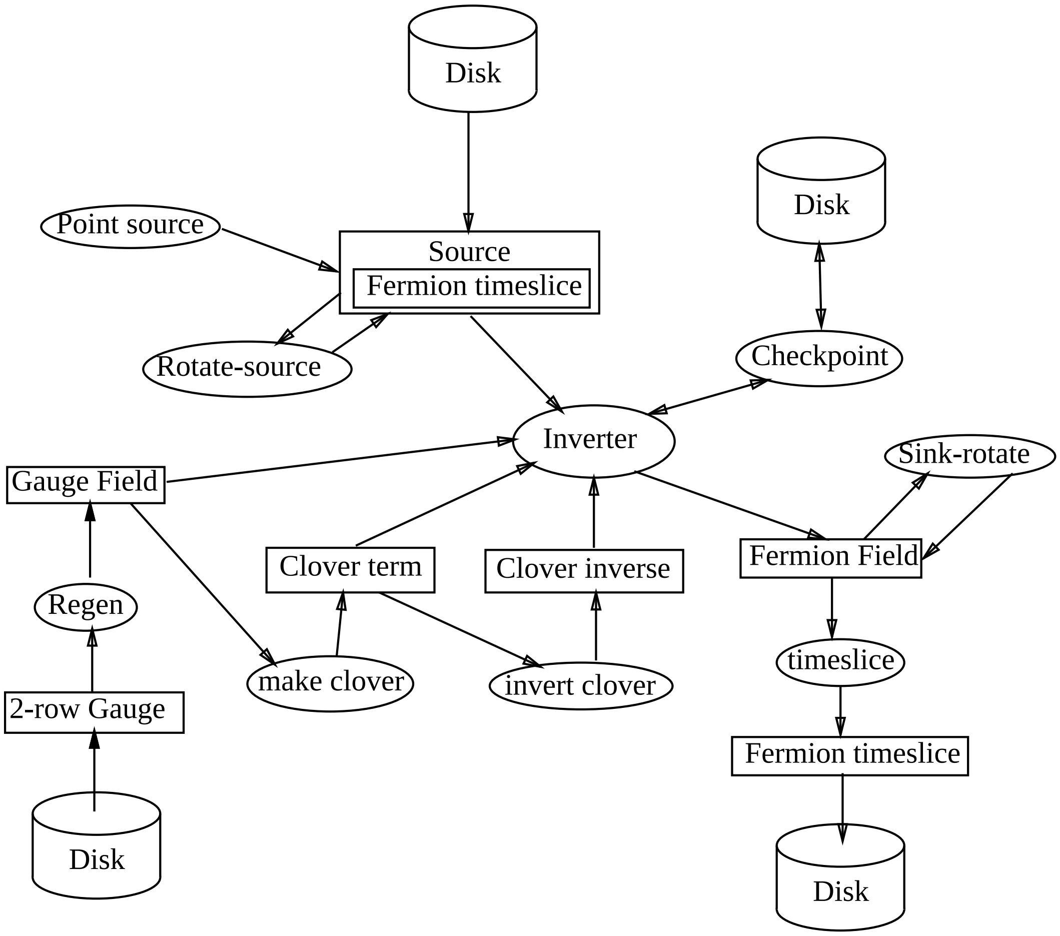


Figure 5.1: Functionality required from the SOLVER application.

1. **Gauge field initialisation**. Although we only need to initialise the gauge fields by loading an old configuration (not the RNG information) in production, for testing we also require the ability to use an ordered, disordered or crossed start[1]. These extra starts should be disabled by a build-time flag when production code is built to minimise code space.
2. **Quark propagator initialisation**. The quark propagators are to be initialised by setting them equal to a point source. This is used as an initial guess for the solver and is as good as most other initial guesses without taking any time to implement and execute.
3. **Linear equation solver**. The package must be capable of supporting more than one linear solver algorithm so that we can re-start a propagator calculation if convergence fails. The solvers should all be red-black preconditioned and have a common interface for simplicity. We also require the ability to swap from red-black to black-red, *i.e.* solve on either even or odd parity and regenerate the opposite, in order to validate code sections as we described fully in section 5.3.1.
4. **Clover terms**. The Clover term 'C' is to be adjustable at run-time, although if a value of 0 ± 0.01[2] is requested the Clover terms should not be calculated in order to save time. There should also be a build-time option to disable the Clover terms (and force C to be zero) in case we want to use the Wilson action and save space usually used for the Clover terms.

   Only one parity of each term is to be stored, as this is all that is required for the solvers. The residue calculation (see below) will require both parities, but time is non-critical here and we can calculate each parity as needed. The terms are stored in the block form defined in section C.2

---

[1] The crossed configuration is used for testing Clover code and is described in section 5.3

[2] This value is arbitrary.

as this takes less memory. We must be must be careful where in the call-tree we create the Clover terms; their creation takes a large amount of temporary work space.

5. **Source creation**. The SOLVER application should only be able to create point sources; any other type should be read in from disk and created by the SOURCE application (see section 5.5) so that we again reduce code space required in SOLVER.
6. **Residue calculation**. Once the propagator has been calculated, the package should calculate an algorithmically independent measure of the accuracy of the solution, the *residue*, which should also be independent of the source normalisation. We define the residue to be

$$|r| = \frac{|M\psi - \eta|}{|\eta|} \tag{5.1}$$

7. **Pion propagator**. We require the pion propagator to be calculated for each source spin/colour of the inversion and also a cumulative sum over all source spin/colour components. This is used for validating the results of the solver and quark propagator files. The algorithm to implement this is given in section C.3.
8. **Input and output**. A separate file is to be used for each source spin and colour component and time-slice of the propagator. When gauge fields are loaded from disk, we do not need to read the random number information.

   The quark propagator files loaded/saved are to be validated by two possible methods:

   (a) Calculating a 16-bit checksum on the files. This is to be done in the same ways as for the GAUGE application and is subject to the same restrictions.

(b) Calculating the time-sliced pion propagator summed over space, sink spin and spin colour, *i.e.* a value for each timeslice, source spin and source colour.

5.2 Design and implementation

The design of the SOLVER application can be easily seen to break into modules on top of the underlying libraries as shown in figure 5.2. The important features of the design are discussed below.

5.2.1 Driver

The necessary gauge and quark propagator (psi) fields are declared here. It is worth noting that if the Clover action is used, 2.8 times more memory than the GAUGE application uses is needed. The gauge fields require 72 words per lattice site, the quark propagator fields are 96 words p.l.s. and the Clover terms 108 words p.l.s.. For this reason the workspace used matters far more than it did for the GAUGE application.

5.2.2 Gauge start

For production purposes we restrict the types of start to loaded gauge configurations. For testing and debugging purposes the ordered, disordered and crossed starts can be used. A build-time flag is the simplest way to implement these two possibilities.

5.2.3 Implement Boundary Conditions

Initially we only intend to implement periodic and anti-periodic boundary conditions (BCs). The periodicity is implemented through the communications library in the construction of the gather-scatter and neighbour tables or use of `CSHIFT` as described previously. Anti-periodic boundary conditions require the positive boundary in the direction chosen to bring a factor of -1 into calculations with the fermion fields. Since these fields always occur multiplied by a

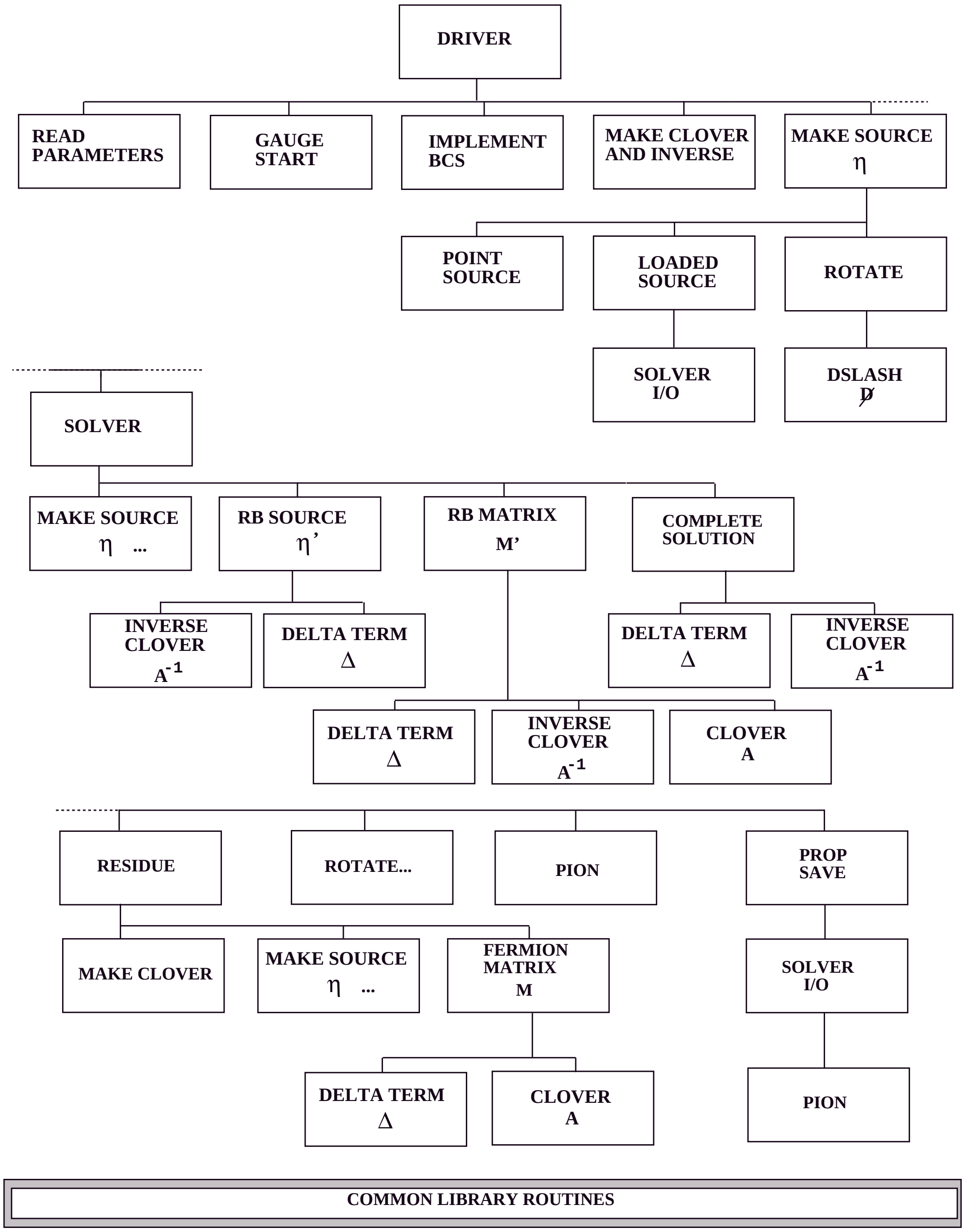


Figure 5.2: Overview of SOLVER design structure.

gauge field it is easier to throw this factor of -1 onto the gauge configuration; in practice we do this once after the gauge fields have been initialised. Note that this does not affect the Clover term constructed from the gauge fields since they consist of closed plaquette sums; any plaquette extending over the lattice boundary containing the anti-periodic BCs will have two factors of -1 incorporated, hence no net effect.

In the future, other types of boundary condition may need to be implemented, *e.g.* Dirichlet which require no periodicity and off-lattice values of the fermion fields set to zero. These require such a different implementation that they need to be applied far lower down the call tree where the fermion fields are evaluated. For this reason the variable(s) identifying the BCs should be passed down the tree whether currently used or not.

5.2.4 Clover term construction and application

As we do not need the inverse Clover term in the residue calculation we must have the ability to create the Clover term separately to save time and workspace.

We implement the data structures W_μ, L and D defined in section C.2 since they require less storage space and time to calculate than a more naïve implementation. The relevant signs, hermicities and μ-values needed by equation C.16 to create $\sigma_{\mu\nu}F_{\mu\nu}$ are most easily implemented through three lookup tables. The plaquettes needed to make up the Clover leaves are calculated using the library routines discussed in section 4.2 for the staple sum. The most important point to remember when constructing the plaquettes is the order of multiplication of the gauge fields, defined in figure 1.1.

The Clover term and its inverse are stored in a common block; specifying them as explicit parameters passed down through the call tree would mean we *must* allocate space for them, but we want to avoid this for the Wilson action (no Clover terms).

It might appear that we need both parities of the Clover term since we use A_p and $A_{\overline{p}}^{-1}$ in the solver. We can however create $A_{\overline{p}}$, then $A_{\overline{p}}^{-1}$ from it and finally A_p. This takes longer but since we only create the Clover terms in non time-critical sections the memory saving is more important.

5.2.5 Make source

We have two possibilities for the quark source

1. **Point source**

   We are trying to solve the equation

   $$\frac{1}{2\kappa}(A - \kappa \not{D})\psi = \delta_{xy}$$

   for the point source. This is easier to calculate if the normalisation factor 2κ is transferred to the right hand side of the equation. Therefore a single spin/colour element of the fermion field is set to 2κ, the rest of the fermion field is set to zero.

2. **Loaded source**

   The loaded sources are implemented with the same routines as are used to load and save quark propagators for simplicity. This implies a need for a time-slice range in the I/O routines and a general method for passing in the source name.

After creating the source we have the possibility to rotate it as defined in sections 1.1.3.2 and C.4.

5.2.6 Rotate and Dslash ($\not{D}$)

The main problem associated with the rotation is the workspace needed by both it and DSLASH, where 4-spinors are communicated (we cannot perform a decomposition to 2-spinors such as is possible for the hopping/delta term Δ). This is most apparent in the SOLVER module where the solver workspace also has to be accommodated. Luckily the source does not need to be created on every solver iteration, only on entry and when creating the missing parity of the solution. Since time is not critical in the calls we could sacrifice speed to save memory in the $\not{D}$. This possibility must be balanced against over complicating the $\not{D}$ to save a few bytes and not giving up too much speed.

The $\not{D}$ operation is very easily implemented in terms of the maths and communication layers discussed already. For example, in HPF the y-direction calculation

$$2(\not{D}_y\psi)(x) = \gamma_y\left\{U_y(x)\psi(x+\hat{y}) - U_y^\dagger(x-\hat{y})\psi(x-\hat{y})\right\}$$

is implemented as

```
      CALL shift_4spin (spin4_mask, src, p, Y_index,
     $ Negative, tmp1)
      CALL su3_mv4 (gauge_ynp, tmp1, tmp2)
      CALL xpgammay (Y_index, tmp2, res)

      CALL su3_hv4 (gauge_yp, src, tmp1)
      CALL shift_4spin (spin4_mask, tmp1, p, Y_index,
     $ Positive, tmp2)
      CALL xmgammay (Y_index, tmp2, res)
```

The message passing version is similar

```
C copy the source into the workspace vector
        CALL copy_generic(np_sites,(Ncomplex*Ncolour*Nspin4),
     $           0,Max_body,psi, 0,Max_array,phi1)

C copy the fermion vector from x+mu to phi2
```

```
        CALL grab_generic(Ncmp, (mu+Ndim), par, phi1, phi2)

C perform the U^dagger multiplication.
          CALL su3_r_hv4(np_sites,
     $             Max_array, gauge(0,0,0,0,par,mu),
     $             Max_array, phi1)

C perform the multiplication of phi2 by U on site notpar
          CALL su3_r_mv4(np_sites,
     $             Max_array, gauge(0,0,0,0,notpar,mu),
     $             Max_array, phi2)

C copy phi1 from par to notpar
        CALL grab_generic(Ncmp, mu, par, phi1, phi3)

C contruct the result from phi2, phi3 with the gamma algebra
        CALL construct_res(mu, left, phi2, phi3, res)
```

5.2.7 Solver

Since we need to be able to implement *any* solver satisfying the requirements in section 5.1 we cannot be specific about the design or implementation. All of the solvers considered are constructed from a number of common routines.

1. The red-black quark source η', defined by

   $$\eta' = (1 + \kappa\Delta A^{-1})\eta$$

   To generate this we need both parities of the non-red-black source since

   $$\eta'_p = \eta_p + \kappa\Delta_{p\overline{p}}A^{-1}_{\overline{pp}}\eta_{\overline{p}}$$

2. The red-black fermion matrix M', defined by

   $$M'_p = (A_{pp} - \kappa^2\Delta_{p\overline{p}}A^{-1}_{\overline{pp}}\Delta_{\overline{p}p})$$

   which does not mix parities. This operation contains the major computational effect in SOLVER as it needs to be performed at least once per iteration. For some solvers, *e.g.* Conjugate gradient, we need to act with $M'^{\dagger}$ on a 4-spinor. Since A is hermitian we write this as

$$M'^{\dagger}_{pp} = (A_{pp} - \kappa^2 \Delta^{\dagger}_{p\overline{p}} A^{-1}_{\overline{p}\overline{p}} \Delta^{\dagger}_{\overline{p}p})$$

3. Once we have solved for a single red-black parity we need to construct the complete solution by the equation

$$\psi_{\overline{p}} = A^{-1}_{\overline{p}\overline{p}}(\eta_{\overline{p}} + \kappa \Delta_{\overline{p}p}\psi_p)$$

4. Linear algebra routines, *e.g.*

`faxpy` :	$y = ax + y$	`fcaxpy` :	$y = \alpha x + y$
`faxpz` :	$y = ax + z$	`fcaxpz` :	$y = \alpha x + z$
`faypx` :	$y = ay + x$	`fcaypx` :	$y = \alpha y + x$
`fysx` :	$y = y - x$	`fzero` :	$y = 0.0 \ \forall$ sites
`fmod2` :	$\sum_{i,\beta,r} \lvert y^i_\beta(r)\rvert^2$		
`fcdot` :	$\sum_{i,\beta,r} y^{i*}_\beta(r).z^i_\beta(r)$		

where $a \in \mathcal{R}$, $\alpha \in \mathcal{C}$ and x, y and z are 4-spinors. Some of these routines are used in other parts of the solver code and therefore live in the maths library. They closely mimic the standard BLAS routines so that the names can be easily changed to use an optimised BLAS package for a particular platform.

In case the red-black decomposition introduces a small numerical error through the Clover term or modified source, we allow the solver to *restart* itself. Most solver algorithms involve two main steps; an initialisation and then iteration. The minimal residual algorithm for example

initialisation

$$r_0 = \eta' - M'\psi_0$$

iteration: repeat the following until convergence

$$s = M' r_i$$

$$\begin{aligned}\alpha &= \omega\frac{(s,r_i)}{(s,s)}, \ \omega \text{ is the over-relaxation parameters}\\ \psi_{i+1} &= \psi_i+\alpha r_i\\ r_{i+1} &= r_i-\alpha s\end{aligned}$$

The source is only involved once, in the initialisation. Restarting the solver with ψ_0 as the tentative solution forces a 'realignment' of the solution with the source. This normally only needs to be done once but we allow a maximum of 4 restarts for safety.

When we implement the solver algorithms, workspace is one of the major issues. The minimal residual solver described above requires two workspace vectors r and s plus space for the red-black source. This is a small memory requirement; other solvers such as conjugate gradient least norm and biconjugate gradient require far more workspace. We reduce this demand in two ways. Firstly the source storage is written over by the red-black source. Since I/O is assumed to be fast, loaded sources can be reconstructed quickly when needed. If I/O is particularly fast in comparison with the calculation it might be better to save the rotated source the first time it is used (outside solver in the call tree) and then read it in from disk on successive uses. This removes all of the workspace associated with source creation and rotation from the solver call tree thereby reducing overall memory requirements. This method has not yet been tried as we have not been that short on memory. The second way in which we reduce workspace is to use the opposite parity of ψ, *i.e.* $\psi_{\overline{p}}$, for calculations, *e.g.* as s in the minimal residual algorithm.

5.2.8 Hopping or Delta term (Δ)

The hopping, or delta, term defined by equation (1.21)

$$(\Delta q)(x)=\sum_{\mu}(1-\gamma_\mu)U_\mu(x)q(x+\hat{\mu})+(1+\gamma_\mu)U_\mu^\dagger(x-\hat{\mu})q(x-\hat{\mu})$$

forms the core of the solver as it is the only unit requiring local communications. The decomposition into 2-spinors is well described in section C.1. The construction of the term is extremely similar to that of the $\not{D}$ described previously in terms of the communications and maths libraries. The subroutines most in need of optimisation if possible are `su3_mvv` and `su3_hvv` which multiply a quark field by U or $U^\dagger$.

The message-passing implementation allows more possibilities than HPF. If we have enough memory we can overlap communications in all four directions and hopefully speed up the solver.

5.2.9 Residue

Because the residue is independent of the solver algorithm we calculate it outside the solver module. The creation of the Clover term and source requires a large amount of workspace which, when added to that needed for the fermion matrix M and the residue itself, becomes the maximal memory usage in the SOLVER application.

This can be avoided by introducing a second layer to the residue call structure '`residue_par`' as shown below in figure 5.3. The workspace for making the Clover term is now subtracted from the maximum needed.

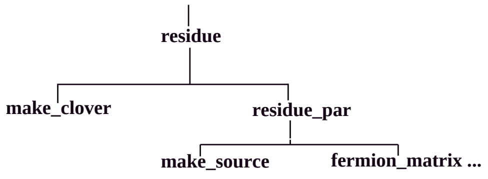


Figure 5.3: Call structure of the residue module.

5.2.10 Solver I/O

All of the points raised for the gauge field I/O apply to this module. An added feature is to use limits on the time-slices loaded and saved so that quark sources can be implemented easily. The dataset names for the propagators reflect the different sources which may be used and the mode in which the gauge field was generated. The format

`<gauge><source>kkkk<action>scTtt`

is used. The gauge name `<gauge>` is simply passed in to the SOLVER application without processing, to simplify the interface. The other elements in the name are the kappa value `kkkk` equal to $\mathrm{INT}(\kappa \times 10^5) - 10^4$, the action type `<action>` which is usually equal to '`W`' for the Wilson action or '`C`' for the Clover action, the source spin '`s`', the source colour '`c`' and the time-slice '`tt`'.

5.3 Solver testing

5.3.1 General tests

There are several general tests (applying to any input parameters) which can be applied to the SOLVER application.

1. The pion propagator can be compared with values generated by an analytic algorithm as shown in section 5.3.2.
2. The pion propagator should be independent of the solver convergence parameters as long as sufficient precision is used to compare results.
3. The pion propagator should be invariant under colour conjugation

$$U \to U^*$$

4. Having a non-zero Clover parameter should not affect the results for a unit gauge configuration or transformed unit gauge configuration. This is because the plaquettes will all be equal to the identity matrix; when the trace is subtracted off as is needed in the Clover algorithm we are left with zero in all elements of the $\sigma_{\mu\nu}F_{\mu\nu}$.
5. Solving on the even-parity sub-lattice should give identical results for the pion propagator as solving on the odd-parity sub-lattice.
6. The quark propagator should be satisfy the following symmetry condition: $\psi^\dagger = \gamma_5 \psi \gamma_5$.

Tests of more specific input parameters are discussed in section 5.3.3.

5.3.2 Analytic pion

The theory in this section below is taken from the paper by Carpenter and Baillie [120], with the notation slightly altered.

The Euclidean lattice fermion propagator can be written

$$\psi(x) = \frac{1}{L_x}\frac{1}{L_y}\frac{1}{L_z}\frac{1}{L_t}\sum_k \mathrm{e}^{ikx}\psi(k) \tag{5.2}$$

where

$$\psi(k) = \frac{m_q + \sum_\mu -i\gamma_\mu \sin k_\mu + (1-\cos k_\mu)}{\sum_\mu \sin^2 k_\mu + \left[m_q + \sum_\mu (1-\cos k_\mu)\right]^2} \tag{5.3}$$

The γ_μ matrices are taken to be hermitian (see equation A.1)

and the momentum sum is over

$$k_\mu = \frac{2\pi(n_\mu + \delta_\mu)}{L_\mu}, \qquad n_\mu = 0, 1, \ldots L_\mu - 1 \tag{5.4}$$

where $\delta_\mu = 0$ for periodic boundary conditions and $\delta_\mu = \frac{1}{2}$ for antiperiodic boundary conditions in the μ-direction.

Once the quark propagator has been calculated, equation (C.23) is used to calculate the pion propagator.

5.3.3 Crossed configurations

Following [120], the quark propagator with the Clover action can be written in terms of a Fourier series [121]

$$\psi(t) = \sum_{k_t} \mathcal{R}\left(\vec{M} + P_- e^{ik_t} + P_+ e^{-ik_t}\right)^{-1} \mathcal{R}\frac{e^{ik_t t}}{N_t} \tag{5.5}$$

where

$$\begin{aligned}
\vec{M} &= M_{em} + \sum_{\vec{\mu}} P_- e^{i\vec{k}} + P_+ e^{-i\vec{k}} \\
\mathcal{R} &= 1 + \sum_{\vec{\mu}} -\frac{\Gamma_{\vec{\mu}} e^{i\vec{k}}}{4} + \frac{\Gamma^\dagger_{\vec{\mu}} e^{-i\vec{k}}}{4} \\
&= 1 - \frac{1}{2}\vec{\not{D}}(\vec{k}) \\
\vec{k} &= \frac{2\pi\left(\vec{x} + \vec{\delta}\right)}{\vec{N}}; \quad \begin{cases} \delta = \frac{1}{2} & \text{antiperiodic b.c.} \\ \delta = 0 & \text{periodic b.c.} \end{cases}
\end{aligned}$$

$$\begin{aligned}
M_{em} &= \frac{1}{2\kappa} + CF_{\mu\nu}\sigma_{\mu\nu} \\
\Gamma_\mu &= U_\mu \gamma_\mu \\
P_{+\mu} &= -\frac{1}{2}U_\mu^\dagger(1+\gamma_\mu) \\
P_{-\mu} &= -\frac{1}{2}U_\mu(1-\gamma_\mu).
\end{aligned}$$

Using the same method as section 5.3.2 we can perform the Fourier sum to obtain an independent measure of the time-sliced pion propagator which can be compared with values obtained from the MPP SOLVER application.

The crossed configuration, *i.e.* a constant background chromo-magnetic field, which we use is defined by the gauge algebra

$$\begin{aligned}
A_x &= (1,3,0,7,0,2,2,0) \\
A_y &= (0,0,0,0,0,0,0,0) \\
A_z &= (0,2,0,0,4,0,5,0) \\
A_t &= (0,0,0,0,0,0,0,0) \qquad (5.6)
\end{aligned}$$

or the group elements

$$U_\mu = e^{iA_\mu^a \lambda^a} \qquad (5.7)$$

where λ^a are the Gell-Mann matrices specified in Appendix A and the A_μ^a are those in equation 5.6. The y and t directions yield the unit matrix, while the x and z gauge fields are shown in table 5.1.

The time-sliced pion propagator values obtained using this initial configuration are shown in tables 5.2 (4^4 lattice) and 5.3 (12^4 lattice) and figure 5.4.

Row,Col	Real part	Imag. part
	x-direction	
0,0	-0.15014863014	0.22101625075
0,1	0.25635275245	-0.30382781143
0,2	0.25006130338	0.84145614691
1,0	-0.55907303095	0.29258449693
1,1	0.40063261986	0.53186789570
1,2	0.33964949846	-0.20755708838
2,0	0.37136927247	0.62651712892
2,1	-0.38087511063	0.50350385177
2,2	-0.12328124791	0.23617974562
	z-direction	
0,0	0.96045821905	-0.00000002002
0,1	0.08339362592	0.00000003994
0,2	0.26564168930	-0.00000001131
1,0	-0.16247718036	0.00000003301
1,1	0.94266444445	-0.00000000361
1,2	0.29152178764	0.00000002325
2,0	-0.22609992325	-0.00000002516
2,1	-0.32315522432	0.00000000593
2,2	0.91893935204	0.00000001828

Table 5.1: Gauge elements for the crossed configuration used for the analytic pion propagator with the Clover action. Values are given for the x and z directions only, the y and t directions are unit matrices.

Timeslice	Wilson	Clover Unrotated	Clover Rotated
0	0.710589902901e 0	0.711207983145e 0	0.119130162633e 1
1	0.363996054550e-1	0.367609728443e-1	0.254744013994e 0
2	0.105703566630e-1	0.109063074722e-1	0.222617573491e-1
3	0.363996054550e-1	0.367609728443e-1	0.254744013994e 0

Table 5.2: Time-sliced pion propagator summed over all spin/colour indices for 4^4 lattice, $\kappa = 0.113636$. Boundary conditions are periodic for the spatial directions and antiperiodic for the t-direction. The target residue was 1×10^{-16} and the solver was started three times for each spin/colour index. These values were first compared with those from the analytic code of [121] to 4 significant figures. Once the code was validated, these figures were generated.

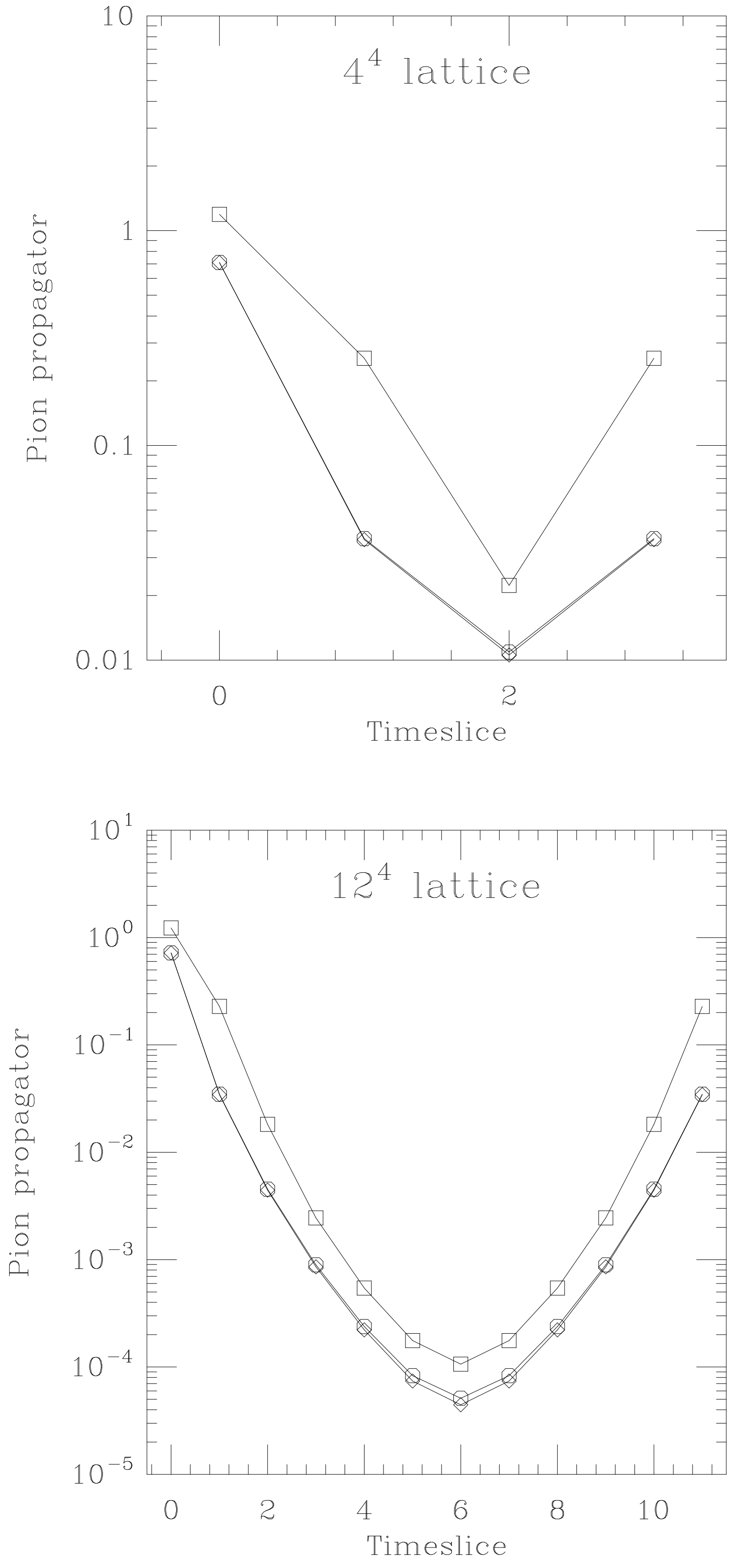


Figure 5.4: The pion propagator for the crossed configuration. Diamonds mark the Wilson action, octagons the unrotated Clover action and squares the rotated Clover action.

Timeslice	Wilson	Clover Unrotated	Clover Rotated
0	0.718904188921e 0	0.719738715575e 0	0.122801635116e 1
1	0.344682975303e-1	0.347212612766e-1	0.228273178229e 0
2	0.444216279213e-2	0.453409840230e-2	0.182428204944e-1
3	0.858280502130e-3	0.895657737452e-3	0.245341606706e-2
4	0.221687794206e-3	0.238508627968e-3	0.544681966919e-3
5	0.741372603559e-4	0.830073090671e-4	0.176177532160e-3
6	0.444284442156e-4	0.511879553088e-4	0.106104264043e-3
7	0.741372603559e-4	0.830073090671e-4	0.176177532160e-3
8	0.221687794206e-3	0.238508627968e-3	0.544681966919e-3
9	0.858280502130e-3	0.895657737452e-3	0.245341606706e-2
10	0.444216279213e-2	0.453409840230e-2	0.182428204944e-1
11	0.344682975303e-1	0.347212612766e-1	0.228273178229e 0

Table 5.3: Time-sliced pion propagator summed over all spin/colour indices for 12^4 lattice, $\kappa = 0.113636$. Initial conditions were as for table 5.2. These values were obtained from the MPP codes after they had been validated against analytic values from the 4^4 lattice.

5.4 Solver performance

The best measure of the performance of the SOLVER application is the time taken per iteration as these iterations dominate all runs of the application. Timings on the CM200, CM5 and T3D are presented in table 5.4.

				12^4 lattice			16^4 lattice		
Platform	Size	Precision	C	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)	Time (secs)	Speed (Gflop/s)	Efficiency (% peak)
CM200	8K	S	0.0	0.315	0.146	3.7	0.315	0.465	11.7
CM200	8K	D	0.0	0.542	0.085	2.1			
CM200	8K	S	1.0	0.380	0.184	4.6	0.416*	0.518	13.0
CM200	16K	S	1.0	0.363*	0.188	2.4	0.207*	1.04	13.0
CM5	16	S	0.0	0.083	0.538	21.0			
CM5	16	S	1.0	0.121*	0.562	22.0	0.264*	0.815	31.8
CM5	32	S	0.0	0.056	0.796	15.5	0.110	1.28	25.0
CM5	32	D	0.0	0.070	0.638	12.5			
CM5	32	S	1.0	0.067	1.01	19.7			
T3D	8	D	0.0	0.323	0.097	8.1	0.977	0.102	8.5
T3D	16	D	0.0	0.173	0.181	7.5			

Table 5.4: Timing data for the minimal residual solver on various platforms for Wilson ($C = 0.0$) and Clover ($C = 1.0$) actions. The times shown are for a single iteration of the solver. Entries with a '*' were generated using code with serial loops unrolled.

From the table we can see that the Clover action is more efficient than Wilson since the application of the Clover term does not involve any communications. Efficiency in general is better than that of the GAUGE code for the Connection Machines because there are no trigonometric maths functions used at all. The CM5 performance is increased by a large degree by unrolling serial loops as explained in section 3.6; an efficiency of 31% for a real application is extremely good for a parallel processor.

As the solvers are built from a generic toolbox of operations we supply timings for these lower-level functions in tables 5.5, 5.6 and 5.7. They should be of some use in predicting the run-time of new solvers and guidance for optimisation.

Operation	12^4 lattice		16^4 lattice	
	Time (secs)	Efficiency (% peak)	Time (secs)	Efficiency (% peak)
8K processors, single precision, $C = 0.0$				
rb_source	0.669	0.7	0.435	3.7
rb_matrix	0.273	3.7	0.261	12.0
fermion_matrix	0.394	1.3	0.293	10.9
fmod2	0.616e-2	4.1	0.104e-1	7.6
fysx	0.183e-2	3.4	0.303e-2	6.5
faxpy	0.199e-2	6.3	0.320e-2	12.3
fcdot	0.106e-1	2.4	0.130e-1	6.1
fcaxpy	0.226e-2	11	0.375e-2	21.0
faypx	0.195e-2	6.4	0.113e-1	3.5
faxpz	0.202e-2	6.2	0.336e-2	11.7
8K processors, single precision, $C = 1.0$				
rb_source	0.254	3.1	0.337*	7.3
rb_matrix	0.338	4.7	0.360*	13.9
fermion_matrix	0.268	3.1	0.172*	15.0
make A,A^{-1}	4.45	2.4	4.05*	8.3
make A	1.63	3.0	2.04*	7.6
make A^{-1}	0.112	7.7	0.190*	14.3
16K processors, single precision, $C = 1.0$, unrolled loops				
rb_source	1.44	0.3	0.422	2.9
rb_matrix	0.177	4.5	0.193	12.9
fermion_matrix	0.083	4.9	0.091	14.1
make A, A^{-1}	3.27	1.6	3.96	4.2
make A	1.20	2.0	2.07	3.7
make A^{-1}	0.057	7.6	0.097	14.0
fmod2	0.247e-2	5.0	0.288e-2	17.2
fysx	0.283e-2	1.1	0.251e-2	3.9
faxpy	0.527e-1	0.1	0.230e-1	0.9
fcdot	0.492e-2	2.5	0.635e-2	6.2
fcaxpy	0.261e-2	4.8	0.310e-2	12.7
faypx	0.113e-2	5.5	0.175e-2	11.2

Table 5.5: CM200 solver toolkit performance.

Operation	12^4 lattice		16^4 lattice	
	Time (secs)	Efficiency (% peak)	Time (secs)	Efficiency (% peak)
16 nodes, single precision, $C = 1.0$, unrolled loops				
rb_source	0.168	7.2	0.249	15.4
rb_matrix	0.102	24.2	0.251	31.1
fermion_matrix	0.040	31.5	0.116	34.7
make A, A^{-1}	0.862	19.3	2.02	26.0
make A	0.390	19.6	0.913	26.4
make A^{-1}	0.101	13.3	0.186	22.8
fmod2	0.862e-3	45.1	0.222e-2	55.4
fysx	0.689e-3	14.1	0.181e-2	17.0
faxpy	0.809e-3	24.1	0.220e-2	27.9
fcdot	0.213e-2	18.3	0.474e-2	25.9
fcaxpy	0.100e-2	38.9	0.279e-2	44.0
faypx	0.951e-3	20.5	0.231e-2	26.6
32 nodes, single precision, $C = 0.0$				
rb_source	0.136	2.9	0.063	19.7
rb_matrix	0.359e-1	21.7	0.182	13.5
fermion_matrix	0.438e-1	9.0	0.730e-1	17.0
fmod2	0.123e-1	1.6	0.233e-2	26.4
fysx	0.693e-3	7.0	0.115e-2	13.4
faxpy	0.727e-3	13.4	0.134e-2	22.9
fcdot	0.234e-2	8.3	0.370e-2	16.6
fcaxpy	0.885e-3	22	0.167e-2	36.8
faypx	0.733e-3	13.3	0.1380e-2	23.6
faxpz	0.818e-3	11.9	0.138e-2	22.3

Table 5.6: CM5 solver toolkit performance.

Operation	12^4 lattice		16^4 lattice	
	Time (secs)	Efficiency (% peak)	Time (secs)	Efficiency (% peak)
8 processors, double precision, $C = 0.0$				
rb_source	0.182	9.2	0.468	11.3
rb_matrix	0.283	11.7	0.853	12.3
fermion_matrix	0.157	10.7	0.466	11.4
fmod2	0.157e-1	5.3	0.316e-1	8.3
fysx	0.160e-1	1.3	0.505e-1	1.3
faxpy	0.179e-1	2.3	0.563e-1	2.3
fcdot	0.111e-1	7.5	0.335e-1	7.8
fcaxpy	0.121e-1	6.9	0.377e-1	7.0
faypx	0.192e-1	2.2	0.607e-1	2.2
faxpz	0.207e-1	2.0	0.655e-1	2.0
16 processors, double precision, $C = 0.0$				
rb_source	0.117	7.2		
rb_matrix	0.150	11.1		
fermion_matrix	0.851e-1	9.9		
fmod2	0.123e-1	3.4		
fysx	0.810e-2	1.3		
faxpy	0.888e-2	2.3		
fcdot	0.613e-2	6.8		
fcaxpy	0.602e-2	6.9		
faxpz	0.104e-1	2.0		

Table 5.7: T3D solver toolkit performance.

5.5 Quark sources: the SOURCE application

The source creation application, SOURCE must be able to provide the functionality shown in figure 5.5.

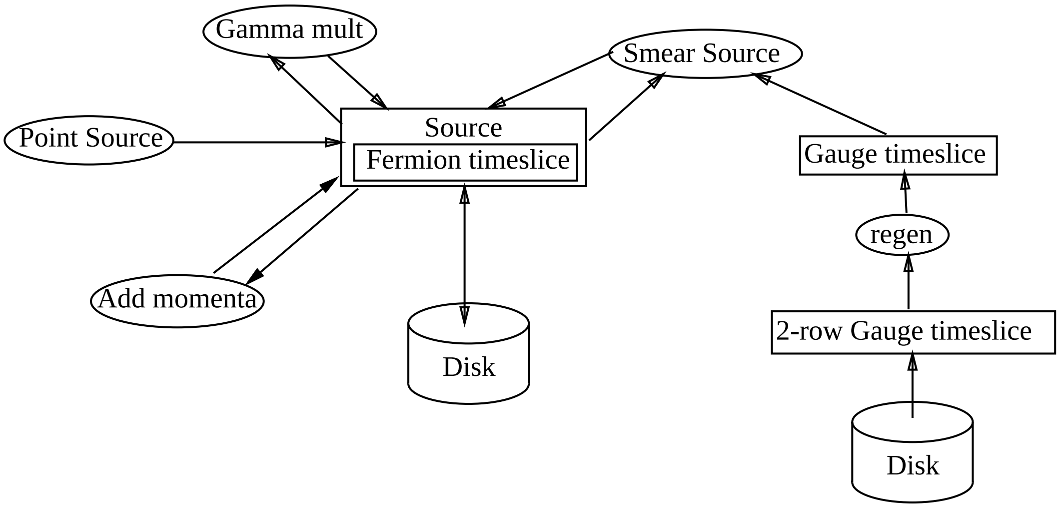


Figure 5.5: Functionality required from the SOURCE application.

1. **Momentum injection**. We must be able to inject momentum into the propagator through the inclusion of a plane-wave factor, $e^{i\vec{p}.\vec{x}}$.
2. **Gamma matrices**. We require the ability to multiply the propagator by a generic Γ matrix (one of 1, γ_5, γ_μ, $\gamma_5\gamma_\mu$, $\sigma_{\mu\nu}$).
3. **Smearing**. We require the ability to smear a propagator over a time-slice as described in section 1.7.1.

The design of this application draws on several of the modules discussed previously; no new features require discussion.

Chapter 6

Conclusions

In this thesis we have shown that it is possible to design and implement a suite of lattice QCD software for message-passing and data-parallel massively parallel processors using software engineering methods. The codes that we have produced will run on any platform supporting PVM or $\subset$HPF with little or no alteration and in some cases, *e.g.* 50% of peak speed on the Connection Machine CM5, produce extremely high performance without optimisation as shown in chapters 4 and 5. The SOLVER kernel has been recently incorporated into the PARKBENCH [122] benchmarking suite, a suite of parallel application kernels from various disciplines requiring high performance computing. As the kernel has only recently been included, no results are available yet. However, the codes in this suite will be tested on all parallel platforms supporting PVM and HPF, the current standards for portability.

The software engineering methods used are not at all common in most physics research establishments, but have been of enormous aid in structuring and documenting the design of the software, sharing the work amongst project members, and detailing the interfaces to the implementation for other members of the group. Mike Peardon, a research student at Edinburgh, has made use of the documentation set and software library described in this thesis to write a Hybrid Monte Carlo simulation code in High Performance/CM Fortran in a short time, less than two months. His project proved valuable for both field-testing our software and documentation and demonstrating the need for a library of portable lattice QCD software and relevant background manuals which are easy to use by people with no previous knowledge of the project.

Software engineering does have its negative aspects. Iteration in the design and implementation phases of the project require the documentation to be kept up to date so that everyone knows what the others are doing. This requires a high degree of organisation and discipline which is not usually present in a physics research department; people usually focus on a narrow area at a time, produce a minimal amount of documentation in order to extract the maximum yield of results in the given time and then move on to another area. Researchers on large software projects need to mainly consider how to make their work easily accessible to people with little specialised knowledge. In our experience however, the extra time and effort required to keep the documentation up to date is easily balanced by the time saved by having all relevant information at hand and the legacy of an easily accessible package.

Some of the portability of the package remains to be tested. The lack of availability of an HPF compiler means that our code has yet to be testing with that standard. HPF compilers are only now emerging on a few platforms with several more in development. MPI is another such problem. Again, a few implementations have been developed, as discussed in chapter 1, but these are mainly built on top of other message-passing systems. The addition of this further layer is good for distributing the standard as widely as possible, but sacrifices some performance through an extra set of function calls. Only once MPI has been implemented as the native message-passing package for a platform will the desired performance be achieved; until that time it is better to stick with established packages, *e.g.* PVM, whose functionality can be easily converted to MPI at a later date.

The future

There are elements to the MPP codes which have yet to be designed and implemented; gauge-fixing in both message-passing and data-parallel and hybrid Monte Carlo in message-passing. These elements should be as easy to construct from the available library as Mike Peardon's data-parallel hybrid Monte Carlo codes.

Extensions can always be made to existing software to enhance performance on particular platforms; particularly the Cray T3D, UKQCD's main production platform for the next few years. Most of these optimisations will be possible without any re-designing of the software due to the modular construction and isolation of critical sections of code as described in this thesis.

It would be most useful to port the message-passing layer to MPI when it becomes widely available. This should only be a matter of a few weeks work for someone familiar with message-passing systems. So many people use PVM at present that a guide to conversion from PVM to MPI will no doubt appear in the near future. New platforms will most likely undertake to implement MPI or HPF as the standard package as they have become so widely accepted among the high performance computing community.

Appendix A

Mathematical conventions

A.1 γ-matrix definitions

The γ-matrices used are

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

$$\gamma^2 = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad \gamma^3 = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}$$

$$\gamma^5 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \tag{A.1}$$

so that $\sigma_{\mu\nu} = \frac{i}{2}\left[\gamma_\mu, \gamma_\nu\right]$ is

$$\sigma_{00} = \sigma_{11} = \sigma_{22} = \sigma_{33} = 0$$

$$\sigma_{01} = -\sigma_{10} = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} \quad \sigma_{02} = -\sigma_{20} = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & -i & 0 \\ 0 & i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}$$

$$\sigma_{03} = -\sigma_{30} = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad \sigma_{12} = -\sigma_{21} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\sigma_{13} = -\sigma_{31} = \begin{pmatrix} 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{pmatrix} \quad \sigma_{23} = -\sigma_{32} = \begin{pmatrix} 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{A.2}$$

A.2 Gell-Mann matrix definitions

We follow the definitions in [123, Appendix F]. The Gell-Mann matrices λ_a satisfy the relation

$$\mathrm{Tr}\ \lambda_a \lambda_b = 2\delta_{ab} \tag{A.3}$$

The definitions are

$$\begin{aligned}
\lambda_1 &= \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \lambda_2 = \begin{pmatrix} 0 & -i & 0 \\ i & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \lambda_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
\lambda_4 &= \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad \lambda_5 = \begin{pmatrix} 0 & 0 & -i \\ 0 & 0 & 0 \\ i & 0 & 0 \end{pmatrix} \quad \lambda_6 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \\
\lambda_7 &= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -i \\ 0 & i & 0 \end{pmatrix} \quad \lambda_8 = \frac{1}{\sqrt{3}} \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -2 \end{pmatrix}
\end{aligned} \tag{A.4}$$

Appendix B

Generating quenched gauge configurations: technical details

B.1 Heatbath update

In the heatbath algorithm (see section 1.3.1) the new values of the link variables are independent of the old ones. The quasi-heatbath method consists of performing heatbath updates on a sequence (we use 3) of $SU(2)$ subgroups of the group $SU(3)$. Reference [124] is more general, describing $SU(N)$. The main reason for using subgroups is that while the sum of $SU(2)$ matrices is proportional to an $SU(2)$ matrix this does not hold for $SU(3)$ matrices.

We write the Wilson pure gauge action in the form

$$S = constant - \frac{\beta}{N}\mathrm{ReTr}\ \ U.R$$

where U is the matrix of the link to be updated and R is the sum over staples (see figure 1.3).

In the following, 3×3 matrices are denoted by capital letters (*e.g.* U, X), and 2×2 matrices by lowercase letters (*e.g.* u, x). So we have

$$\begin{aligned} X &= U.R \\ x &= (U.R)_{2\times 2} \ \text{ a submatrix} \end{aligned}$$

One can parametrise the 2×2 complex matrix x as (see section B.1.1)

$$x = ku_1 + ik'u_2 \tag{B.1}$$

where $k, k' \in \mathcal{R}$ and u_1, u_2 are SU(2) matrices. We then use the property that for any SU(2) matrix h one can write

$$\mathrm{ReTr}\ \ (h.x) = k\mathrm{Tr}\ \ (h.u_1)$$

Remember also that an SU(2) matrix h can be parametrised in terms of Pauli matrices

$$\sigma_0 = 1_{2\times 2} \quad \sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{B.2}$$

$$h = h_0.1_{2\times 2} + ih_i\sigma_i \tag{B.3}$$

with

$$h_0^2 + h_i^2 = 1 \text{ and } h_0, h_{i=1,2,3} \in \mathcal{R} \tag{B.4}$$

Now suppose we choose

$$h = au_1^\dagger$$

with $u_1^\dagger$ defined via equation B.1 and a an SU(2) matrix so that h is itself an SU(2) matrix. Then the configuration probability

$$\begin{aligned} dP(hu) &\propto e^{\frac{\beta}{N}\mathrm{Re}\mathrm{Tr}\;(hx)}dh \quad = e^{\frac{\beta}{N}k\mathrm{Tr}\;(hu_1)}dh \quad = e^{\frac{\beta}{N}k\mathrm{Tr}\;(a)}dh \\ &= e^{\frac{\beta}{N}k.2a_0}da \end{aligned} \tag{B.5}$$

since a is SU(2), and we have used the invariance of the SU(2) Haar measure. The problem is now reduced to generating a_0 with the distribution

$$P(a_0)da_0 \propto e^{\frac{2\beta k}{N}a_0}da_0(1-a_0^2)^{\frac{1}{2}} \tag{B.6}$$

where we used

$$da = da_0 d^3a_i\delta(1-a_0^2-a_i^2)$$

which in polar coordinates is

$$\begin{aligned} & da_0 dr d\theta d\phi \frac{\partial(a_1,a_2,a_3)}{\partial(r,\theta,\phi)}\delta(1-a_0^2-\vec{r}^2) \\ = \;& \frac{1}{2}(1-a_0^2)^{\frac{1}{2}}da_0 dr d\theta d\phi \sin\theta\,\delta(r-(1-a_0^2)^{\frac{1}{2}}) \end{aligned}$$

The a_i are then generated uniformly on a 2-sphere of radius $(1-a_0^2)^{1/2}$. The procedure for generating these random matrices is described in section B.1.2.

We perform *3 hits* of this kind taking 3 different 2×2 submatrices of $X^i = U^i R$, $i = 1, 2, 3$

$$X^1 = \begin{pmatrix} x^1_{00} & x^1_{01} & 0 \\ x^1_{10} & x^1_{11} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad X^2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & x^2_{00} & x^2_{01} \\ 0 & x^2_{10} & x^2_{11} \end{pmatrix} \quad X^3 = \begin{pmatrix} x^3_{00} & 0 & x^3_{01} \\ 0 & 1 & 0 \\ x^3_{10} & 0 & x^3_{11} \end{pmatrix} \tag{B.7}$$

and U^i denotes

$$U^0 = U, \;\; U^1 = h_1 U^0, \;\; U^2 = h_2 U^1, \;\; U' = U^3 = h_3 U^2 \tag{B.8}$$

The

$$h_i = (a u_1^\dagger)_i$$

are enlarged to SU(3) matrices by putting a 1 in the diagonal element and $0's$ in the off-diagonal elements. The staple sum R_μ remains *unchanged* throughout.

Summary

The Cabibbo-Marinari update of $U_\mu(x)$ can be summarised as follows

```
loop over hits $i$ from 1 to 3 step +1
    let $X = U^i.R$
    choose $2 \times 2$ complex submatrix to be $x_1, x_2$ or $x_3$
    parametrise $x_i$ to get $k$ and $u_1^\dagger$
    generate $a_0$ and $a_i$ according to distribution (B.6)
    let $U^i = (a u_1^\dagger)_i U^{i-1}$
```

B.1.1 Parametrisation of a complex 2×2 ($\mathcal{C}_{2\times 2}$) matrix.

Given

$$x = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathcal{C}_{2\times 2}$$

we want to write

$$x = k u_1 + i k' u_2$$

with $k \in \mathcal{R}$ and $u_1 \in \mathrm{SU}(2)$. We only need to calculate k and u_1. So

$$x = ku_1 + ik'u_2 = k(e_0 1_{2\times 2} + i\vec{e}.\vec{\sigma}) + ik'(f_0 1_{2\times 2} + i\vec{f}.\vec{\sigma})$$

with $e_0^2 + \vec{e}^2 = 1$ and $f_0^2 + \vec{f}^2 = 1$. Then

$$\begin{aligned} x &= (ke_0 + ik'f_0)1_{2\times 2} + i(k\vec{e} + ik'\vec{f}).\vec{\sigma} \\ &= x_0.1_{2\times 2} + i\vec{x}.\vec{\sigma} = x_0.1_{2\times 2} + \tilde{\vec{x}}.\vec{\sigma} \end{aligned}$$

where $x_0, x_j \in \mathcal{C}$ for $j = 1, 2, 3$. We then have

$$\tilde{x}_\mu = \frac{1}{2}\mathrm{Tr}\ \ (x.\sigma_\mu)$$

for $\mu = 0, 1, 2, 3$. This then gives

$$\begin{aligned} x_0 &= \frac{1}{2}(a + d) = ke_0 + ik'f_0 \\ x_j &= -\frac{i}{2}\mathrm{Tr}\ \ (x.\sigma_\mu) = ke_j + ik'f_j \end{aligned}$$

so that

$$\begin{aligned} ke_0 &= \frac{1}{2}\mathrm{Re}\ (a + d),\ \ ke_1 = \frac{1}{2}\mathrm{Im}\ (b + c) \\ ke_2 &= \frac{1}{2}\mathrm{Re}\ (b - c),\ \ ke_3 = \frac{1}{2}\mathrm{Im}\ (a - d) \end{aligned} \tag{B.9}$$

Use $e_0^2 + \vec{e}^2 = 1$ to get

$$k = \frac{1}{2}\sqrt{\left\{\mathrm{Re}^2(a + d) + \mathrm{Re}^2(b - c) + \mathrm{Im}^2(b + c) + \mathrm{Im}^2(a - d)\right\}} \tag{B.10}$$

Finally,

$$u_1 = \begin{pmatrix} e_0 + ie_3 & e_2 + ie_1 \\ -e_2 + ie_1 & e_0 - ie_3 \end{pmatrix} \tag{B.11}$$

B.1.2 Generating the pseudo-random numbers a_0

The full working for this algorithm is shown in [125]. To generate a random number a_0 with the distribution

$$P(a_0) = N^{-1}\sqrt{1 - {a_0}^2}e^{\alpha a_0},\ \ (-1 \leq a_0 \leq 1)$$

1. Generate two uniformly distributed pseudo-random numbers R and R' in the unit interval.
2. Set $X = -\frac{\ln R}{\alpha}$, $X' = -\frac{\ln R'}{\alpha}$.
3. Set $C = \cos^2(2\pi R'')$, with R'' another uniform random number in $(0, 1]$.
4. Let $A = XC$.
5. Let $\overline{\delta} = X' + A$.
6. If $R'''^2 > 1 - \frac{\overline{\delta}}{2}$, for R''' pseudo-random and uniform in $(0, 1]$, go back to step 1.
7. Set $a_0 = 1 - \overline{\delta}$.

Note that in step 4 using $B = X - A$ will generate an independent result for a_0.

B.1.3 Generating the random numbers a_1, a_2, a_3

Now that the a_0 have been generated we still need to calculate the a_1, a_2 and a_3 on the SU(2) manifold. We require that

$$a_1^2 + a_2^2 + a_3^2 = 1 - a_0^2$$

The procedure is as follows

1. Let r_1 be $\sqrt{1 - a_0^2}$
2. Let r'' be uniformly distributed random number in $[-1, 1]$
3. $a_3 = r'' r_1$
4. Let r_2 be $\sqrt{1 - r''^2}$
5. Let r' be uniformly distributed random number in $(0, 1]$
6. $a_1 = r_1 r_2 \cos(2\pi r')$
7. $a_2 = r_1 r_2 \sin(2\pi r')$

B.2 Over-relaxed update

The rationale for using over-relaxed techniques is explained in section 1.3.2. We shall first explain the method for SU(2). In practice, as with Cabibbo-Marinari, the hits are performed on SU(2) subgroups of SU(3).

B.2.1 Over-relaxation with SU(2).

We want to update the gauge field $U \in SU(2)$. The action is

$$S_U = -\frac{\beta}{N}\mathrm{ReTr}\ (U.R)$$

In SU(2) we can write the sum over staples as a multiple of another SU(2) matrix

$$\begin{aligned} R &= \sum^{6}(staples) \\ &= k.\tilde{U} \end{aligned}$$

where $k \in \mathcal{R}$, and $\tilde{U}$ is SU(2).

We then define U_0 to be the SU(2) matrix which minimises the action

$$\begin{aligned} S_{U_0} &= -\frac{\beta k}{2}\mathrm{ReTr}\ (U_0\tilde{U}) \\ &= -\frac{\beta k}{2}\mathrm{Tr}\ (U_0\tilde{U}) \end{aligned}$$

So clearly letting

$$U_0 = \tilde{U}^{-1}$$

gives

$$S_{U_0} = -\frac{\beta k}{2}2 = -\beta k$$

Then let the new link be

$$\begin{aligned} U' &= U_0 U^{-1} U_0 \\ &= \tilde{U}^{-1} U^{-1} \tilde{U}^{-1} \end{aligned}$$

The new element U' now lies on the opposite side of the group manifold from element U_0. Note that U_0, whilst minimising the action, does not depend on U. With this choice

$$\begin{aligned}
S_{U'} &= -\frac{\beta k}{2}\mathrm{Tr}\ (U'\tilde{U}) \\
&= -\frac{\beta k}{2}\mathrm{Tr}\ (\tilde{U}^{-1}U^{-1}) \\
&= -\frac{\beta k}{2}\mathrm{Tr}\ (\tilde{U}^{\dagger}U^{\dagger}) \\
&= -\frac{\beta k}{2}\mathrm{Tr}\ (U\tilde{U})^{\dagger} \\
&= -\frac{\beta k}{2}\mathrm{Tr}\ (U\tilde{U}) \\
&= S_U
\end{aligned} \tag{B.12}$$

i.e. the action remains unchanged and the update is always accepted.

B.2.2 Over-relaxation with SU(3).

As in the Cabibbo-Marinari method we perform 3 hits on SU(2) subgroups. We have

$$S_U = -\frac{\beta}{3}\mathrm{ReTr}\ \ (U.R)$$

and we need to find an element U' such that

$$S_U = S_{U'}$$

where U and U' differ by a multiple of an SU(2) subgroup.

With

$$X = U.R$$

and

$$\begin{aligned}
x &= (U.R)_{2\times 2}\ \text{submatrix} \in \mathcal{C} \\
&= ku_1 + ik'u_2
\end{aligned}$$

Let

$$U_0 = V.U$$

where

$$V = \begin{pmatrix} v_{00} & v_{01} & 0 \\ v_{10} & v_{11} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and $v \in \mathrm{SU}(2)$. We choose v to minimise the action S_U

$$\begin{aligned} S_{min} &= -\frac{\beta}{3}\mathrm{ReTr}\ \begin{pmatrix} v_{00} & v_{01} & 0 \\ v_{10} & v_{11} & 0 \\ 0 & 0 & 1 \end{pmatrix} U.R \\ &= -\frac{\beta}{3}\mathrm{ReTr}\ (v.x) + constant \end{aligned}$$

Choosing $v = u_1^\dagger = u_1^{-1}$ minimises the action

$$\begin{aligned} S_{min} &= = -\frac{\beta}{3}\mathrm{ReTr}\ (u_1^{-1}(ku_1 + ik'u_2)) \\ &= -\frac{\beta}{3}2.k \end{aligned}$$

So let the new link be

$$\begin{aligned} U' &= U_0 U^{-1} U_0 \\ &= VUU^{-1}VU \\ &= V^2 U \\ &= \begin{pmatrix} (u_1^\dagger)^2_{00} & (u_1^\dagger)^2_{01} & 0 \\ (u_1^\dagger)^2_{10} & (u_1^\dagger)^2_{11} & 0 \\ 0 & 0 & 1 \end{pmatrix} U \end{aligned} \tag{B.13}$$

This choice leaves the action unchanged and reduces to the SU(2) result if U is an SU(2) matrix.

The 3 hits are performed with the same subgroups as for the Cabibbo-Marinari update. And, as can easily be seen, the computation is almost identical as for the Cabibbo-Marinari update.

B.3 Calculation of the staple sum

The requirement for the staple sum is explained in section 1.3.3.

B.3.1 Theory

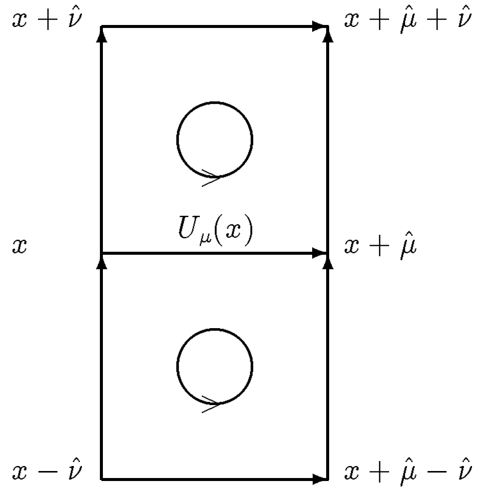


Figure B.1: The two plaquettes containing the link to be updated. The direction of evaluation of the links is also shown.

The plaquette action in the μ,ν plane is (see figure B.1)

$$\mathrm{ReTr}\ \left\{U_\mu(x)U_\nu(x+\hat{\mu})U_\mu^\dagger(x+\hat{\nu})U_\nu^\dagger(x)\right\} \tag{B.14}$$

for the top plaquette and

$$\mathrm{ReTr}\ \left\{U_\mu^\dagger(x)U_\nu^\dagger(x-\hat{\nu})U_\mu(x-\hat{\nu})U_\nu(x-\hat{\nu}+\hat{\mu})\right\} \tag{B.15}$$

for the bottom plaquette. So summing top and bottom plaquettes over all planes ($\mu\nu$, $\mu\rho$, $\mu\tau$) yields

$$\mathrm{ReTr}\ \left\{U_\mu(x)\left[\sum_{i=\nu,\rho,\tau} U_i(x+\hat{\mu})U_\mu^\dagger(x+\hat{\imath})U_i^\dagger(x)\right]\right.$$

$$+ U_\mu^\dagger(x) \left[\sum_{i=\nu,\rho,\tau} U_i^\dagger(x-\hat\imath)U_\mu(x-\hat\imath)U_i(x-\hat\imath+\hat\mu) \right] \Bigg\} \tag{B.16}$$

But for any 3 by 3 complex matrix

$$\mathrm{ReTr}\;\; U = \mathrm{ReTr}\;\; U^\dagger \tag{B.17}$$

so that defining the staple sum $R_\mu(x)$ as

$$R_\mu(x) = \sum_{i=\nu,\rho,\tau} U_i(x+\hat\mu)U_\mu^\dagger(x+\hat\imath)U_i^\dagger(x) + U_i^\dagger(x-\hat\imath+\hat\mu)U_\mu^\dagger(x-\hat\imath)U_i(x-\hat\imath) \tag{B.18}$$

the sum of plaquette actions around $U_\mu(x)$ is

$$\mathrm{ReTr}\;\; \{U_\mu(x)R_\mu(x)\} \tag{B.19}$$

B.3.2 Algorithm

The algorithm for creating the top and bottom staples is as shown below (see figure B.2 for labelling of links and points used).

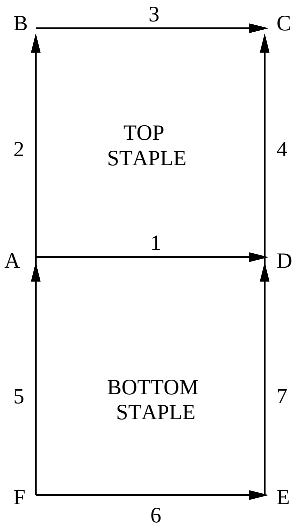


Figure B.2: Labelling of links as used in the algorithmic description of the creation of the staples.

Top Staple

1. Move 3 to A.
2. Multiply $3^\dagger \times 2^\dagger$.
3. Move 4 to A.
4. Multiply $4 \times (3^\dagger \times 2^\dagger)$.

Bottom Staple

1. Move 7 to F.
2. Multiply $7^\dagger \times 6^\dagger$.
3. Multiply $(7^\dagger \times 6^\dagger) \times 5$.
4. Move $(7^\dagger \times 6^\dagger \times 5)$ to A.

B.4 Calculation of the plaquettes

Plaquettes can be calculated in-line with a staple sum or on a stand-alone basis. The number returned is obtained in the following way

1. Multiply gauge link by the staple sum.
2. Calculate trace of plaquette product at all sites.
3. Sum over the plane of interest.
4. Divide by the lattice volume and no. of colours

resulting in a number in the interval $[-1, 1]$.

B.5 Reunitarisation

The requirement for reunitarisation is given in section 1.3.4.

SU(3) matrices can be written in the form

$$\begin{pmatrix} \vec{u} \\ \vec{v} \\ (\vec{u} \times \vec{v})^* \end{pmatrix} \tag{B.20}$$

where $\vec{u}$,$\vec{v}$ are three-vectors of complex numbers. They obey the constraints

$$\begin{aligned} \vec{u}^*.\vec{u} = \vec{v}^*.\vec{v} &= 1 \\ \vec{u}^*.\vec{v} &= 0 \end{aligned} \tag{B.21}$$

Therefore the method used to reunitarise

$$\begin{pmatrix} \vec{u} \\ \vec{v} \\ \vec{w} \end{pmatrix} \rightarrow \begin{pmatrix} \vec{u}' \\ \vec{v}' \\ \vec{w}' \end{pmatrix}$$

is

1. Normalise $\vec{u} \rightarrow \vec{u}'$ s.t.

   $$\vec{u}'^*.\vec{u}' = 1 \tag{B.22}$$

   i.e.

   $$\vec{u}' = \frac{\vec{u}}{|\vec{u}|} \tag{B.23}$$

   where

   $$|\vec{u}| = \sqrt{u_1^* u_1 + u_2^* u_2 + u_3^* u_3} \tag{B.24}$$

2. Use the Gram-Schmidt orthogonalisation process to construct a vector $\tilde{\vec{v}}$ orthogonal to $\vec{u}'$ *i.e.*

   $$\tilde{\vec{v}} = \vec{v} - (\vec{v}.\vec{u}'^*)\vec{u}' \tag{B.25}$$

3. Normalise $\tilde{\vec{v}} \rightarrow \vec{v}'$ as in 1 above.

4. Construct

   $$\vec{w}' = (\vec{u}' \times \vec{v}')^* \tag{B.26}$$

Appendix C

Generation of quark propagators: technical details

C.1 Hopping term algorithm

C.1.1 Introduction

The hopping, or delta, term arises in the fermion matrix as shown in section 1.1.3. The fermion matrix is defined as

$$M = A - \kappa\Delta \tag{C.1}$$

where A is the Clover term (see Appendix C) and Δ is the hopping term.

C.1.2 Non-daggered

The delta term is the most compute-intensive part of the whole propagator code and is therefore worth spelling out in detail. The operation to be performed is

$$(\Delta\psi)(x) = \sum_{\mu}(1-\gamma_\mu)U_\mu(x)\psi(x+\mu) + (1+\gamma_\mu)U_\mu^\dagger(x-\mu)\psi(x-\mu) \tag{C.2}$$

For gamma matrix definitions refer to equation A.1. This is the lattice difference operator which corresponds to $\not{D}$ in the continuum limit.

It is possible to perform the gamma matrix algebra and shifts as given *i.e.* with ψ as a four-spinor. However this is slow and a faster method has been found (and used with great effect in the Maxwell code).

Since the γ matrices act only on the spin indices we can commute them through the gauge fields. With the substitutions

$$\chi_\mu(x+\mu) \quad = \quad (1-\gamma_\mu)\psi(x+\mu)$$

$$\chi'_\mu(x-\mu) \;=\; (1+\gamma_\mu)\psi(x-\mu) \tag{C.3}$$

we get the four-spinors χ_μ, χ'_μ which can be written as

$$\begin{aligned}
\chi_0 &= \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix} = \begin{pmatrix} 2\psi_0 \\ 2\psi_1 \\ 0 \\ 0 \end{pmatrix} \\
\chi_1 &= \begin{pmatrix} 1 & 0 & 0 & i \\ 0 & 1 & i & 0 \\ 0 & -i & 1 & 0 \\ -i & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix} = \begin{pmatrix} \psi_0 + i\psi_3 \\ \psi_1 + i\psi_2 \\ -i(\psi_1 + i\psi_2) \\ -i(\psi_0 + i\psi_3) \end{pmatrix} \\
\chi_2 &= \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ 0 & -1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix} = \begin{pmatrix} \psi_0 + \psi_3 \\ \psi_1 - \psi_2 \\ -(\psi_1 - \psi_2) \\ \psi_0 + \psi_3 \end{pmatrix} \\
\chi_3 &= \begin{pmatrix} 1 & 0 & i & 0 \\ 0 & 1 & 0 & -i \\ -i & 0 & 1 & 0 \\ 0 & i & 0 & 1 \end{pmatrix} \begin{pmatrix} \psi_0 \\ \psi_1 \\ \psi_2 \\ \psi_3 \end{pmatrix} = \begin{pmatrix} \psi_0 + i\psi_2 \\ \psi_1 - i\psi_3 \\ -i(\psi_0 + i\psi_2) \\ i(\psi_1 - i\psi_3) \end{pmatrix}
\end{aligned} \tag{C.4}$$

These four spinors only have two independent components each so therefore can be written as two-spinors without any loss of information. The two-spinors can then be sent or received by processors with half of the communication time (ignoring overheads) and recombined. Since the gamma matrix operations are simply permuting indices and multiplying by factors of 'i' there is no great overhead in computation time.

If we write

$$\begin{aligned}
\Pi_{\mu\alpha}(x) &= U_\mu(x)\chi_{\mu\alpha}(x+\mu) \\
\Pi'_{\mu\alpha}(x) &= U^\dagger_\mu(x-\mu)\chi'_{\mu\alpha}(x-\mu)
\end{aligned} \tag{C.5}$$

where $\mu \in 0..3$ is the direction index and $\alpha = 0..1$ is the 2-spinor index, then the delta term can be written as

$$\Delta\psi(x) = \begin{pmatrix} -i\Pi_{x1} + \Pi'_{x0} & -\Pi_{y1} + \Pi'_{y0} & -i\Pi_{z0} + \Pi'_{z0} & +\Pi'_{t0} \\ -i\Pi_{x0} + \Pi'_{x1} & +\Pi_{y0} + \Pi'_{y1} & +i\Pi_{z1} + \Pi'_{z1} & +\Pi'_{t1} \\ \Pi_{x0} - i\Pi'_{x1} & +\Pi_{y0} - \Pi'_{y1} & +\Pi_{z0} - i\Pi'_{z0} & +\Pi_{t0} \\ \Pi_{x1} - i\Pi'_{x0} & +\Pi_{y1} + \Pi'_{y0} & +\Pi_{z1} + i\Pi'_{z1} & +\Pi_{t1} \end{pmatrix} \tag{C.6}$$

C.1.3 Daggered

The daggered equation is needed for some solvers. It can be written as

$$\begin{aligned}\left(\Delta^\dagger\psi\right)(x) &= \sum_\mu (1+\gamma_\mu)U_\mu(x)\psi(x+\mu) + (1-\gamma_\mu)U^\dagger_\mu(x-\mu)\psi(x-\mu) \qquad (C.7)\\ &= \sum_\mu \Lambda_{\mu\alpha} + \Lambda'_{\mu\alpha} \qquad (C.8)\end{aligned}$$

where

$$\begin{aligned}\Lambda_{\mu\alpha} &= U_\mu(x)\chi'_{\mu\alpha}(x+\mu) \qquad (C.9)\\ \Lambda'_{\mu\alpha} &= U^\dagger_\mu(x-\mu)\chi_{\mu\alpha}(x-\mu)\end{aligned}$$

The daggered delta term can then be written as

$$\Delta^\dagger\psi(x) = \begin{pmatrix} \Lambda_{x0} - i\Lambda'_{x1} & +\Lambda_{y0} - \Lambda'_{y1} & +\Lambda_{z0} - i\Lambda'_{z1} & +\Lambda_{t0} \\ \Lambda_{x1} - i\Lambda'_{x0} & +\Lambda_{y1} + \Lambda'_{y0} & +\Lambda_{z1} - i\Lambda'_{z0} & +\Lambda_{t1} \\ -i\Lambda_{x1} + \Lambda'_{x0} & -\Lambda_{y1} + \Lambda'_{y0} & -i\Lambda_{z0} + \Lambda'_{z0} & +\Lambda'_{t0} \\ -i\Lambda_{x0} + \Lambda'_{x1} & +\Lambda_{y0} + \Lambda'_{y1} & +i\Lambda_{z1} + \Lambda'_{z1} & +\Lambda'_{t1} \end{pmatrix} \tag{C.10}$$

C.2 Clover term implementation

C.2.1 Introduction

The Clover term in the action is defined in section 1.1.3 to be

$$A = 1 - \kappa C \sigma_{\mu\nu} F_{\mu\nu} \tag{C.11}$$

where κ is the quark hopping parameter, C is the Clover coefficient for adjusting the action, $\sigma_{\mu\nu}$ are defined in Appendix A and $F_{\mu\nu}$ is the field strength, defined by

$$\begin{aligned}F_{\mu\nu}(x) &= \frac{Q_{\mu\nu}(x) - Q_{\mu\nu}\dagger(x)}{2i}\\ Q_{\mu\nu}(x) &= \frac{1}{4}\sum_{i=1,4} U^i_{\Box\mu\nu}(x) \qquad (C.12)\end{aligned}$$

with the $Q_{\mu\nu}$ shown in figure 1.1.

C.2.2 Matrix definitions

Because all of the plaquettes are calculated in the same direction (see figure 1.1), $F_{\mu\nu}$ has the following anti-symmetry

$$F_{\mu\nu} = -F_{\nu\mu} \tag{C.13}$$

With these definitions we can see that the matrix $\sigma_{\mu\nu}F_{\mu\nu}$ can be written as

$$\sigma_{\mu\nu}F_{\mu\nu} = \begin{bmatrix} -F_{12} & -F_{23} - iF_{13} & -F_{03} & -F_{01} + iF_{02} \\ -F_{23} + iF_{13} & F_{12} & -F_{01} - iF_{02} & F_{03} \\ -F_{03} & -F_{01} + iF_{02} & -F_{12} & -F_{23} - iF_{13} \\ -F_{01} - iF_{02} & F_{03} & -F_{23} + iF_{13} & F_{12} \end{bmatrix} \tag{C.14}$$

Note that we have only summed over $\mu < \nu$ so that planes are not double counted. By defining the following quantities

$$\begin{aligned} W_0 &= F_{21} \\ W_1 &= F_{32} + iF_{31} \\ W_2 &= F_{30} \\ W_3 &= F_{10} + iF_{02} \end{aligned} \tag{C.15}$$

we see that

$$\sigma_{\mu\nu}F_{\mu\nu} = \begin{bmatrix} W_0 & W_1 & W_2 & W_3 \\ W_1^\dagger & -W_0 & W_3^\dagger & -W_2 \\ W_2 & W_3 & W_0 & W_1 \\ W_3^\dagger & -W_2 & W_1^\dagger & -W_0 \end{bmatrix} \tag{C.16}$$

It is the elements of W^μ which are stored by the program, this form requires a quarter of the memory of storing the full Clover term.

C.2.3 The decomposition of A^{-1}

A decomposition of A is performed to make the multiplication by A^{-1} easier (and indeed to *find* A^{-1} more easily) [126]. The following equation is being solved

$$Ax = y$$

so that by setting

$$A = L^\dagger D L$$

we ensure that

$$L^\dagger D L.x = y$$

or

$$x = (L)^{-1}.(D)^{-1}.(L^\dagger)^{-1}.y = A^{-1}.y$$

where L is a lower triangular matrix and D is a diagonal matrix. In fact the matrices L^{-1} and D^{-1} are stored. This is simply the action of Clover inverse on a quark propagator field as required. This decomposition can be performed because of the hermicity properties of A.

C.2.4 Decomposition algorithm

The decomposition is performed by the following algorithm (in the following we let i, j stand for both the spin indices α, β and the colour indices a, b, so that Ndiag is equal to 12).

```
DO i = Ndiag-1 TO 0 STEP -1
  DO j = i TO 0 STEP -1

    LET t = $A_{ij}$
    DO k = i+1 TO Ndiag-1 (not for $i$ =Ndiag-1)
      LET t = t - $L^*_{ki}.D_k.L_{kj}$
    ENDDO
    IF ( i = j ) THEN
      LET $D_i$ = t
    ELSE
      LET $L_{ij}=t/D_i$
    ENDIF

  ENDDO
  LET $D_i = 1/D_i$ (gives us $D^{-1}$)
ENDDO
```

C.2.5 Multiplying by A^{-1}

To multiply a quark propagator field by A^{-1} the following algorithm is used

First multiply by $(L^\dagger)^{-1}$
On entry: x is the vector to be multiplied, y is the result.

```
LET $y = x$ (Do not do in-place calculation.)
DO i = Ndiag-1 TO 0 STEP -1
  DO j = Ndiag-1 TO i+1 STEP -1 (not for $i$ =Ndiag-1)
    LET $y_i = y_i - L^*_{ji}.y_j$
  ENDDO
ENDDO
```

Now multiply by $(DL)^{-1}$...

```
DO i = 0 TO Ndiag-1 STEP 1
  LET $y_i = y_i * D_i$
  DO j = 0 TO i-1 STEP 1 (Do nothing when $i = 0$)
    LET $y_i = y_i - L_{ij} * y_j$
  ENDDO
ENDDO
```

C.3 The in-line pion propagator

The in-line pion propagator is calculated after the solver has calculated the quark propagator as a check on a physical quantity. This should be gauge invariant (a useful check when random gauge transform of unity is applied).

The quark propagator can be written as

$$\psi^{ij}_{\alpha\beta}(\vec{x}, t; 0) = \langle 0| q^i_\alpha(\vec{x}, t) \overline{q}^j_\beta(0) |0\rangle \tag{C.17}$$

where i, j are colour indices and α, β are spin indices. The quark propagator is evaluated for different values of the quark mass m_q, which is related to the hopping parameter κ by

$$\kappa = \frac{1}{2m_q + 8} \tag{C.18}$$

The pion correlation function is the quantity which we wish to measure and is denoted by

$$C(\vec{x}, t) = \langle 0|\pi(\vec{x}, t)\pi^\dagger(0)|0\rangle \tag{C.19}$$

where

$$\pi^\dagger = ((q^\dagger\gamma_0)\gamma_5 q)^\dagger = q^\dagger\gamma_5\gamma_0 q = -\overline{q}\gamma_5 q$$

so that

$$\begin{aligned} C(x) &= -\langle 0|\overline{q}^i_\alpha(x)(\gamma_5)_{\alpha\beta}q^i_\beta(x)\overline{q}^j_\gamma(0)(\gamma_5)_{\gamma\delta}q^j_\delta(0)|0\rangle \\ &= -(\gamma_5)_{\alpha\beta}(\gamma_5)_{\gamma\delta}\langle 0|q^i_\beta(x)\overline{q}^j_\gamma(0)|0\rangle\langle 0|\overline{q}^i_\alpha(x)q^j_\delta(0)|0\rangle \\ &= -(\gamma_5)_{\alpha\beta}(\gamma_5)_{\gamma\delta}\psi^{ij}_{\beta\gamma}(x;0)\psi^{ji}_{\delta\alpha}(0;x) \end{aligned} \tag{C.20}$$

The lattice Dirac equation gives

$$\psi^{ji}_{\delta\alpha}(0;x) = (\gamma_5)_{\delta\rho}\psi^{\dagger ji}_{\delta\sigma}(0;x)(\gamma_5)_{\sigma\alpha} \tag{C.21}$$

so that

$$\begin{aligned} C(x) &= -(\gamma_5)_{\alpha\beta}\psi^{ij}_{\beta\gamma}(x;0)(\gamma_5)_{\gamma\delta}(\gamma_5)_{\delta\rho}\psi^{\dagger ji}_{\rho\sigma}(0;x)(\gamma_5)_{\sigma\alpha} \\ &= Tr\left[\gamma_5\psi(x;0)\gamma_5\gamma_5\psi^\dagger(0;x)\gamma_5\right] \\ &= Tr\left[\psi(x;0)\psi^\dagger(0;x)\right] \\ &= |\psi^{ij}_{\alpha\beta}(x;0)|^2 \end{aligned} \tag{C.22}$$

What is done in practice is to calculate the modulus squared of the quark propagator for a particular source spin (β) and colour (j), then sum over sink spin (α), colour (i) and spatial indices ($\vec{x}$) to get a value for the time-slice, this is a real number. We finally sum over the source spin and colour to obtain a single real value for each time-slice.

$$\boxed{\text{pion prop}(t) = \sum_j\sum_\beta\sum_{\vec{x}}\sum_i\sum_\alpha |\psi^{ij}_{\alpha\beta}(\vec{x}, t)|^2} \tag{C.23}$$

C.3.1 Free field pion

When performing calculations in unit gauge (or gauge transformed unit gauge) *e.g.* test code, the full pion (with source summed over all spin and colour) can be constructed from just one spin and colour using the periodic boundary conditions and spin symmetry.

C.4 Rotations in the Clover action

As shown in section 1.1.3, we need to rotate the fermion fields according to

$$\begin{aligned} \psi \quad &\rightarrow \quad \psi' = (1 - \frac{1}{2}\overrightarrow{\not{D}})\psi = R_l\psi \\ \text{and } \bar{\psi} \quad &\rightarrow \quad \bar{\psi}' = \bar{\psi}(1 + \frac{1}{2}\overleftarrow{\not{D}}) = \bar{\psi}R_r \end{aligned} \tag{C.24}$$

where the lattice covariant derivatives are defined by

$$\begin{aligned} (\overrightarrow{D}_\mu f)(x) &= \frac{1}{2}(U_\mu(x)f(x+\hat{\mu}) - U_\mu^\dagger(x-\hat{\mu})f(x-\hat{\mu})) \\ \text{and } (f \overleftarrow{D}_\mu)(x) &= \frac{1}{2}(f(x+\hat{\mu})U_\mu^\dagger(x) - f(x-\hat{\mu})U_\mu(x-\hat{\mu})) \end{aligned} \tag{C.25}$$

If we write these derivatives as full matrices with f a column vector and f^T the corresponding row vector, then

$$\begin{aligned} (\overrightarrow{D}_\mu f)(x) &= \sum_y \overrightarrow{D}_\mu (x,y)f(y) \\ \text{and } (f^T \overleftarrow{D}_\mu)(x) &= \sum_y f(y) \overleftarrow{D}_\mu (y,x) \end{aligned} \tag{C.26}$$

so that we can write

$$\begin{aligned} \overrightarrow{D}_\mu (x,y) &= \frac{1}{2}(U_\mu(x)\delta_{y,x+\hat{\mu}} - U_\mu^\dagger(x-\hat{\mu})\delta_{y,x-\hat{\mu}}) \\ \text{similarly } \overleftarrow{D}_\mu (y,x) &= \frac{1}{2}(U_\mu^\dagger(x)\delta_{y,x+\hat{\mu}} - U_\mu(x-\hat{\mu})\delta_{y,x-\hat{\mu}}) \\ \text{or } \overleftarrow{D}_\mu (x,y) &= \frac{1}{2}(-U_\mu(x)\delta_{y,x+\hat{\mu}} + U_\mu^\dagger(x-\hat{\mu})\delta_{y,x-\hat{\mu}}) \\ &= -\overrightarrow{D}_\mu (x,y) \end{aligned} \tag{C.27}$$

i.e. when viewed as full matrices which can act in either direction, we have that

$$\overleftarrow{D}_\mu = -\overrightarrow{D}_\mu \tag{C.28}$$

which implies that

$$\begin{aligned} R_l &= (1 - \frac{1}{2}\overrightarrow{D}) \\ &= (1 - \frac{1}{2}(-\overleftarrow{D})) \\ &= (1 + \frac{1}{2}\overleftarrow{D}) \\ &= R_r \equiv R \end{aligned} \tag{C.29}$$

i.e. the two rotations are exactly the same matrix.

The Clover action is invariant under the rotation to terms of order a^2, hence

$$\begin{aligned} \overline{q}(y) M_{\text{clover}}(y,x) q(x) &\rightarrow \overline{q}(y) R R^{-1} M_{\text{clover}} R^{-1} R q(x) + \mathcal{O}(a^2) \\ &= \overline{q}_R(x) M_I q_R(x) + \mathcal{O}(a^2) \end{aligned} \tag{C.30}$$

So the improved quark propagator is given by

$$< q_R(x) \overline{q}_R(y) >= M_I^{-1}(x,y) \tag{C.31}$$

We therefore need to compute

$$M_I^{-1}(x,y) = R M_{\text{clover}}^{-1}(x,y) R \tag{C.32}$$

We cannot compute the whole propagator matrix since it is too large, so we fix point y at the origin and restrict ourselves to calculating a single column of the propagator. So if we let $\eta(y) = \delta_{y,0}$ be the usual source vector, we can apply the rotation R to η and solve

$$\sum_x M_{\text{clover}}(y,x) \phi(x) = R\eta(y) \tag{C.33}$$

for ϕ, *i.e.*

$$\phi(x) = \sum_y M_{\text{clover}}^{-1}(x,y) R\eta(y) \tag{C.34}$$

If we apply the rotation again to the solution ϕ, we obtain the required quark propagator field

$$\begin{aligned} \psi(x) &= R\phi(x) \\ &= \sum_y RM^{-1}_{\text{clover}}(x,y)R\eta(y) \\ &= RM^{-1}_{\text{clover}}(x,0)R \end{aligned} \tag{C.35}$$

which is the required result.

In summary, the procedure used is

1. Apply the rotation R to the source $\eta \rightarrow \eta' = R\eta$.
2. Solve $M_{\text{clover}}\phi = \eta'$ for ϕ.
3. Compute $\psi = R\phi$.

Appendix D

F90 and HPF: Important language features

D.1 Introduction

In this appendix we explain the most important features of Fortran 90, High Performance Fortran (HPF) and Connection Machine Fortran (CMF). These features are related to array handling in Fortran. Only the features used in this project have been documented here (a very small subset).

An early description of migration from CM Fortran to HPF is described in [127]. This has several omissions which have been inserted in this discourse.

D.2 Relevant Fortran 90 features

In this section we present not only the Fortran 90 features which are used extensively, but also those which have not been used because their use may be misleading, or lead to problems. For further details of these features, refer to [60]

D.2.1 Array declarations

Arrays are declared in the following way

```
      Cfpoint, DIMENSION (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1) ::
     $
     $  gauge_xevn, gauge_xodd

      Cfpoint single_gauge (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)
```

which shows the declaration of multiple arrays of the same type (first statement) or a single array (second statement). The single declaration can be put in the same form as the multiple with only a single variable name after the '`::`' if desired. Both of the arrays are of type `Cfpoint`, single precision complex. Note that CMF allows the use of `ARRAY` as well as `DIMENSION`, but HPF does not; the latter should always be used.

D.2.2 Arithmetic operations on arrays and array sections

When arrays are of the same type, arithmetic operations can be performed on whole arrays with a simple statement *e.g.*

```
INTEGER, DIMENSION (0:4,0:4) :: a, b, c

a = b + c
a = a * 6
```

adds the elements of `b` to those of `c` and places the result in `a`, and then multiplies *all* elements of `a` by a scalar. If you wish to only do the first element in the first dimension but all the elements in the second dimension, the '`:`' notation can be used *e.g.*

```
INTEGER, DIMENSION (0:4,0:4) :: a, b, c

a(0,:) = b(0,:) + c(0,:)
```

this notation should not be used for local indices, only distributed, as CMF cannot implement it properly.

Although HPF allows you to pass out sections of arrays to functions, *e.g.*

```
INTEGER, DIMENSION (0:9) :: pass_out

CALL operate(pass_out(0:8:2))
```

which would pass out elements 0, 2, 4, 6, 8 of the array `pass_out`, CMF does not; this feature must not be used.

D.2.3 Index ordering and 'fastest index'

Fortran 90 does **not define** which index moves fastest *i.e.* the layout in memory of the array. In Fortran 77 it is defined to be the leftmost index. This means that passing an array element to a function (as is done in Fortran 77) to use as a 'pointer' to the part of the array you are interested in **cannot be done**. Subroutines must either be given the whole array and select a section of it, or a section must be copied into workspace and passed to the subroutine. The Fortran 77 code

```
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "precision.h"

      INTEGER big (0:Ncolour-1,0:Ncolour-1,0:Npar-1)

      CALL junk(big(0,0,1))
```

which passes the second parity of the array `big` to the subroutine `junk`, would have to be coded in Fortran 90 as

```
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "precision.h"

      INTEGER big (0:Ncolour-1,0:Ncolour-1,0:Npar-1)
      INTEGER temp (0:Ncolour-1,0:Ncolour-1)

      INTEGER row,col
      DO row=0,Ncolour-1
        DO col=0,Ncolour-1
          temp (row,col) = big (row,col,1)
        END DO
      END DO

      CALL junk(temp)
```

As we often need to operate on single parities of the fields, we use a separate array for each parity in order to reduce the amount of copying to temporary arrays.

D.2.4 Number of indices allowed

Only 7 indices are allowed in HPF to ensure backwards compatibility with Fortran 77. For this reason a set of gauge fields has to have a separate array for each direction and parity.

D.2.5 Array intrinsics used in MPP codes

These are operations on arrays. Some result in a derived quantity such as a sum, others transform the array *e.g.* `CSHIFT`. The following intrinsics operate in the same way on arrays as they do on scalars

☐ `ABS` — take the absolute value (modulus) of an array of numbers.
☐ `REAL` — take the real part of an array of complex numbers.
☐ `AIMAG` — take the imaginary part of an array of complex numbers.
☐ `CONJG` – take the complex conjugate of an array of complex numbers.

D.2.5.1 The `SUM` intrinsic

This intrinsic sums the elements of an array. It can have options specifying a mask and the dimension to sum over, but we only use a very simple form

```
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "precision.h"

      Cfpoint tmp (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)
      Cfpoint sum_tmp
```

```
      sum_tmp=SUM(tmp)
```

which sums every element of `tmp`.

D.2.5.2 The `ANY` intrinsic

This intrinsic returns a `LOGICAL` value, depending on a test *e.g.*

```
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "precision.h"

      Fpoint tmp (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)

      IF (ANY(tmp .NE. 0.0)) THEN
        WRITE (*,*) 'Non-zero element found'
      END IF
```

which displays a message if any element of `tmp` contains a non-zero value.

D.2.5.3 The `CSHIFT` intrinsic

This intrinsic is used for moving data in an array by Cyclic **SHIFT**ing, exactly the operation needed to implement periodic boundary conditions. The syntax is

`<dest> = CSHIFT (<source>, SHIFT=<dir>, DIM=<dim>)`

where `<dest>` is the resultant array, `<source>` is the array to be operated on, `<dir>` is the direction and amount to shift the array and `<dim>` is the dimension of the array to shift. Note that the CM Fortran version of `CSHIFT` reverses the last two arguments if the `SHIFT` and `DIM` keywords are not put in. To be portable you **must specify these keywords**. The direction and amount `<dir>` is only ever set to $+1$ or -1.

As an example of its use consider the array

$$source = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

All arrays are addressed throughout our software as (row,col) so the command

```
source = CSHIFT (source, SHIFT=-1, DIM=2)
```

changes `source` to

$$\begin{pmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \end{pmatrix}$$

i.e. the matrix has been shifted in the increasing column direction. Note that `<dim>` starts counting from 1, not 0 as our array indices do.

D.2.5.4 Masks

A mask is simply an array of logicals which indicates where a conditional operation is to take place. For example with the matrix `source` above, the matrix `source_mask`

$$\texttt{source\_mask} = \begin{pmatrix} \texttt{.TRUE.} & \texttt{.FALSE.} & \texttt{.TRUE.} \\ \texttt{.FALSE.} & \texttt{.TRUE.} & \texttt{.FALSE.} \end{pmatrix}$$

can be used to enable operations on `source` only where the mask is `.TRUE.`. See the sections on `MERGE` and `WHERE` for details of use.

D.2.5.5 The `MERGE` intrinsic

The `MERGE` intrinsic merges together two arrays depending on the value held in a mask. The syntax is

```
<dest>=MERGE(<tsource>,<fsource>,<mask>)
```

Consider the following example of its use with the matrices `tarray`, `farray`, `tfmask`.

$$\texttt{tarray} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

$$\texttt{farray} = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$$

$$\texttt{tfmask} = \begin{pmatrix} \texttt{.TRUE.} & \texttt{.FALSE.} & \texttt{.TRUE.} \\ \texttt{.FALSE.} & \texttt{.TRUE.} & \texttt{.FALSE.} \end{pmatrix}$$

with the following command

```
result = MERGE (tarray, farray, tfmask)
```

The matrix `result` would be set to

$$\texttt{tarray} = \begin{pmatrix} 1 & 8 & 3 \\ 10 & 5 & 12 \end{pmatrix}$$

D.2.5.6 The `WHERE` statement

The `WHERE` statement can operate on arrays according to the value of the mask. It can be used in one of two forms

1. The `WHERE` *statement*.

```
WHERE (<mask>) <statement>
```

2. The `WHERE...ELSEWHERE...ENDWHERE` *construct*.

```
WHERE (<mask>)
  <statement1>
ELSEWHERE
  <statement2>
ENDWHERE
```

The statements enclosed must operate on arrays of the same shape and size as `<mask>`. They may not call subroutines or functions (except intrinsics).

Note that on the Connection Machine `MERGE` is used more often as it is faster (by a factor of 2!). This may not be true for all compilers and should be investigated on the machine in question. The following two statements are equivalent

```
result = MERGE (tarray,farray,tfmask)
```

and

```
WHERE (tfmask)
  result = tarray
ELSEWHERE
  result = farray
ENDWHERE
```

If all matrix shifting is placed in a single subroutine for each matrix type, switching between `MERGE` and `WHERE` is simplified.

D.2.6 Features not used, or not allowed

D.2.6.1 The `INTERFACE` block

These blocks (which declare the interface to a subroutine or function and allow greater type checking within the compiler) are not used. This is because the source files become extremely long and the compilers on the Connection Machine cannot cope. An additional pain is that the interface blocks would have to be put in header files; if a header file is included but the relevant routine is not called a whole slew of warning and error messages are generated by the compiler.

D.2.6.2 The `CASE` statement

This is disallowed by the $\subset$HPF, on the grounds that it is non-essential. The construct `IF...THEN...ELSE IF...` must be used.

D.2.6.3 The `SYSTEM_CLOCK` intrinsic

This intrinsic is not used to perform timing because of the ambiguity in what is being timed. When time-sharing on machines the system clock on the front-end bears no relation to the processor time on the machine. Machine-dependent timing is expected.

D.2.6.4 The `RANDOM_NUMBER` and `RANDOM_SEED` intrinsics

The use of these intrinsics is not forced so that users have greater flexibility in choosing a random number generator. The RNG selected by these defaults is not a standard across compilers or machines — the performance is an unknown — so use of these intrinsics is also not encouraged.

D.3 Subset High Performance Fortran

The use of $\subset$HPF requires additions to the Fortran 90 features discussed above, not changes.

D.3.1 HPF directives

HPF directives start with

```
!HPF$ <directive> ...
```

The only directives we use are

☐ `PROCESSORS`

☐ `DISTRIBUTE`

which are explained below.

D.3.1.1 The `PROCESSORS` directive

This directive specifies the mapping of arrays elements (*abstract* or *virtual* processors) onto the *physical* processors (or *processing elements*). Since all of our arrays are distributed on a 4-D lattice, the usage is

```
!HPF$ PROCESSORS <name> (<x>,<y>,<z>,<t>)
```

which specifies that the mapping, called `<name>`, uses (`<x>`,`<y>`,`<z>`,`<t>`) processors along each direction. In practice this should be defined in a header file which is included by all subroutines.

D.3.1.2 The `DISTRIBUTE` directive

This directive specifies how the array is laid out on the machine. The usage is

```
!HPF$ DISTRIBUTE <name> (<layout>,...) ONTO <mapping>
```

for single arrays, or

```
!HPF$ DISTRIBUTE (<layout>,...) ONTO <mapping> ::
!HPF$$ <name1>,...
```

for multiple arrays.

An example of use is

```
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "processors.h"
#include "precision.h"

      Cfpoint, DIMENSION (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1) ::
     $
     $  gauge_evn, gauge_odd
!HPF$ DISTRIBUTE (*,*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS ::
!HPF$$ gauge_evn, gauge_odd
```

which will cause all colour components to live on the *same* abstract processor (because of the '*'), and the x, y, z and t components to be distributed over different abstract processors. Each space-time point will have a separate $SU(3)$ matrix on a separate abstract processor in effect.

D.3.2 The `FORALL` statement

This allows you to perform several loops simultaneously. The syntax used is

```
      FORALL (<loopvar>=<lowlim>:<highlim>,...) <statement>
```

e.g.

```
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "precision.h"
#include "processors.h"

      INTEGER y, z, t
      LOGICAL spin2_mask (0:Ncolour-1,0:Nspin4-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1)
!HPF$ DISTRIBUTE (*,*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS ::
!HPF$$ spin2_mask

      FORALL (y=0:Ny-1, z=0:Nz-1, t=0:Nt-1)
     $  spin2_mask (:,:,:,y,z,t) = MOD(y+z+t,2) .EQ. 0
```

D.4 Connection Machine Fortran

The only differences from the features discussed earlier are those of layout and common block location. The following code fragment illustrates the differences (the addition of `CMF$` directives). The use of `FEONLY` specifies that the common block lives on the front end; if the common block is intended to be on the MPP machine, omit this line.

```
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "processors.h"
#include "precision.h"

      INTEGER common_var
      COMMON /var_common/ common_var
CMF$  COMMON FEONLY /var_common/
      Cfpoint, DIMENSION (0:Ncolour-1,0:Ncolour-1,
     $  0:Nxby2-1,0:Ny-1,0:Nz-1,0:Nt-1) ::
     $
     $  gauge_evn, gauge_odd
CMF$  LAYOUT gauge_evn (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT gauge_evn (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
!HPF$ DISTRIBUTE (*,*,BLOCK,BLOCK,BLOCK,BLOCK) ONTO QCDPROCS ::
!HPF$$ gauge_evn, gauge_odd
```

Note that there is no way of declaring multiple variables in a single CMF directive. Since directives are legal Fortran comments, the MPP codes have both sets of directives left in place.

D.5 The x-direction with regard to parity

The layout of the arrays is particularly important when considering which array elements are on each virtual processor. Note that the important concept in this section is the virtual processor (VP) - *not* the processing element (PE). Array elements from two arrays situated on the same VP involve *no inter-processor* *communications* by definition. This speeds up the execution of such elements by an order or so.

This is most useful when considering the splitting up of the lattice into two sub-lattices labelled by parity. A point on each lattice with the same *logical* coordinate is on the same VP.

The x-direction is a special case. Neighbouring points in the x-direction may or may not need communication *e.g.* case (1) in figure(D.1) requires communication whereas case (2) does not. For this reason, all matrix shifting should be performed using subroutines which check the direction and parity.

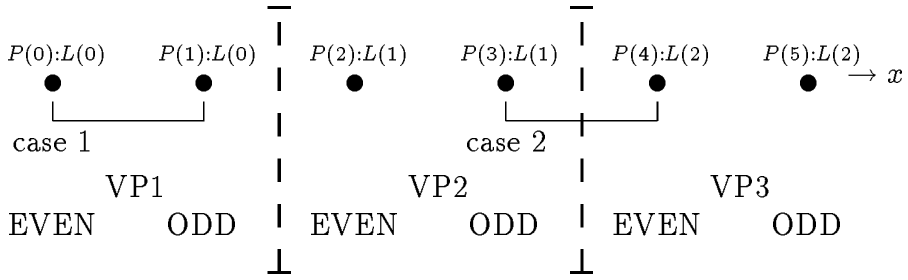


Figure D.1: The siting of points on the same or different VPs determines whether communication is required. Case 1 potentially requires communication whereas case 2 does not.

Appendix E

Message passing packages: important features

The code presented in this appendix was written by Stephen Booth. We assume that the reader has some prior knowledge of PVM. We do not aim to explain all of the structures underlying the code; it is merely included as an example of implementation of the message-passing layer in PVM.

E.1 Header files

There are several options which can be implemented throughout the message-passing code if required. To ease the selection of required options, all relevant build-time flags are defined in a single file, '`pvm_options.h`' as shown below.

```
C options file for the PVM versions of the comms routines.
C

C
C leave data in place.
C
#undef INPLACE

C put in barriers at the start and end
C of all boundary communications
#undef BOUND_BARRIER
#ifdef INPLACE
C we need the barriers if we are doing in-place pvm
#define BOUND_BARRIER
#endif
C


C Use broadcast or multicast.
C
#undef BCAST

C use the binary tree to return gsum results rather than
C using the set functions
#undef TREE_SEND
```

```
C
C complete boundary comms in the start call
C instead of waiting for the end call
C
#undef NO_OVERLAP
```

All constants, common MP variables and data sizes are declared in the header file 'pvmcomms.h'. This file is included by all communications routines.

```
C
C Include file for PVM version of the comms
C

C first we need the standard pvm file
#include <fpvm3.h>

C position of parameters packed in the initialisation block.
      INTEGER Block_size
      INTEGER X_size, Y_size, Z_size, T_size
      INTEGER X_pos, Y_pos, Z_pos, T_pos
      INTEGER Boss_pid, My_pid

      PARAMETER(Block_size = 10,
     $   X_size = 1, Y_size = 2,
     $   Z_size = 3, T_size = 4,
     $   X_pos = 5, Y_pos = 6,
     $   Z_pos = 7, T_pos = 8,
     $   My_pid=9, Boss_pid=10)

C     parameters for the comms.
      INTEGER Nbranch, Encoding, Bound_encode
      PARAMETER(Nbranch=2, Encoding=PVMRAW)
#ifdef INPLACE
      PARAMETER(Bound_encode=PVMINPLACE)
#else
      PARAMETER(Bound_encode=PVMRAW)
#endif

      CHARACTER*(*) g_group
      PARAMETER(g_group = 'ggrp')

C parameters for message tags
      INTEGER Fgsum_tag, Dgsum_tag, Igsum_tag
      INTEGER Fgset_tag, Dgset_tag, Igset_tag
      INTEGER Pio_tag, Pio_req_tag
      INTEGER Bound_base, Init_tag

      PARAMETER(Bound_base = 10,
     $     Init_tag = 1,
```

```
     $     Fgsum_tag = 2,
     $     Dgsum_tag = 3,
     $     Igsum_tag = 4,
     $     Fgset_tag = 5,
     $     Dgset_tag = 6,
     $     Igset_tag = 7,
     $     Pio_tag = 8,
     $     Pio_req_tag=9)

      INTEGER tidtable, neighbours, n_proc, myid, dir_tag
      LOGICAL send_ok

      COMMON /PVMTAB/
     $   tidtable(0:Max_proc-1),
     $   neighbours(0:Ndir-1),
     $   n_proc, myid,
     $   dir_tag(0:Ndir-1),
     $   send_ok(0:Ndir-1)


#ifndef PVMTYPE
#define PVMTYPE
C declare FTYPE DTYPE ITYPE to match precision.h
#if (Fsize == 4)
#define FTYPE REAL4
#endif
#if (Fsize == 8)
#define FTYPE REAL8
#endif
#if (Dsize == 4)
#define DTYPE REAL4
#endif
#if (Dsize == 8)
#define DTYPE REAL8
#endif
#if (Isize == 4)
#define ITYPE INTEGER4
#endif
#if (Isize == 8)
#define ITYPE INTEGER8
#endif
#endif
```

E.2 Loader program: `pvmgrid`

To get the application running on the nodes of the MPP we need to spawn the individual processes in a 4-D torus. We use the model where the host process plays no part in the calculation so this program only has to fire up the node programs and tell them their position in the processor array.

```
C host has to have the same integer format as nodes
#include "precision.h"

      PROGRAM pvmgrid
#include "implicit.h"
#include "build_constants.h"
#include "build_size.h"
#include "pvmcomms.h"

      INTEGER Max_node
      PARAMETER(Max_node = Max_proc)
      CHARACTER*80 name
      INTEGER size(0:3)
      INTEGER mytid
      INTEGER count
      INTEGER i,x,y,z,t,bufid,status
      INTEGER info(Block_size)
C functions
      INTEGER proc_pos

C macro defs

C check the task ID
      CALL pvmfmytid(mytid)
      IF (mytid .LT. 0)THEN
        CALL pvmfperror('pvmgrid',status)
        CALL pvmfexit(info)
        STOP
      END IF

C set automatic error printing to ON
      CALL pvmfserror(1,status)

      name = SLAVE_PROG
#ifdef AUTO_SIZE
C AUTO_SIZE can be set in the pre-processor flags. If
C set, the user must hard-wire in the processor grid size
C and lattice size.
      size(X_index) = X_proc
      size(Y_index) = Y_proc
      size(Z_index) = Z_proc
      size(T_index) = T_proc
#else
C Otherwise we must read in the grid from the user
C at run-time; more flexible, but less efficient in
C some cases.
      WRITE(*,*) 'grid size ?'
```

```
      READ(*,*) (size(i), i=0,Ndim-1)
#endif
      WRITE(*,*) 'loading program ' , name
      WRITE(*,2) (size(i), i=0,Ndim-1)
2     FORMAT('onto ',I2,' * 'I2,' * 'I2,' * 'I2)
      DO 5, i=0,3
        IF(size(i) .LT. 1)THEN
          WRITE(*,*) 'illegal grid size, size(',i,') = ',
     $      size(i)
          STOP
        END IF
5     CONTINUE
      n_proc = size(0) * size(1) * size(2) * size(3)
      IF ((n_proc .LT. 1) .OR. (n_proc .GT. Max_node))THEN
        WRITE(*,*) 'illegal grid size',size
        STOP
      END IF
C start the `n_proc' new processes on any machine (PVMDEFAULT),
C the task ID's are returned in `tidtable'. `Count' is the number
C of actual processes started.
      CALL pvmfspawn(name,PVMDEFAULT,'',n_proc,tidtable,count)
      IF(n_proc .NE. count)THEN
C couldn't start enough processes.... die !
        WRITE(*,*) ' error loading program'
        DO 6, i=0,n_proc
          WRITE(*,*) tidtable(i)
          IF (tidtable(i) .LT. 0)THEN
            CALL pvmfperror('pvmgrid',tidtable(i))
          END IF
6       CONTINUE
C leave gracefully after printing error messages...
        CALL pvmfexit(status)
        STOP
      END IF

      WRITE(*,*) 'load ok'

C  now send id information to each processor.
C  ----------------------------------------

C  grid size
      info(X_size) = size(X_index)
      info(Y_size) = size(Y_index)
      info(Z_size) = size(Z_index)
      info(T_size) = size(T_index)

C Tell the slave processes who's boss (proc. 0)
      info(Boss_pid) = 0

C Generate a unique Processor ID for each grid position,
C info(My_pid). See subsection on `Processor layout' for
```

```
C algorithm.
      DO 70, t=0,size(T_index)-1
        DO 65, z=0,size(Z_index)-1
          DO 60, y=0,size(Y_index)-1
            DO 55, x=0,size(X_index)-1
              info(X_pos) = x
              info(Y_pos) = y
              info(Z_pos) = z
              info(T_pos) = t
              i = proc_pos(x,y,z,t,size)
              info(My_pid) = i

C Send info block and task ID's for all processors to
C each processor. Uses the ‘raw’ transfer, i.e. no
C encoding.
              WRITE(*,*) 'starting proc ',i
              CALL pvmfinitsend(Encoding,bufid)
              IF (bufid .LT. 0)THEN
                CALL pvmfperror('pvmgrid',bufid)
                CALL pvmfexit(bufid)
                STOP
              END IF
              CALL pvmfpack(ITYPE,info,Block_size,1,status)
              IF (status .LT. 0) THEN
                CALL pvmfperror('pvmgrid',status)
                CALL pvmfexit(status)
                STOP
              END IF
              CALL pvmfpack(ITYPE,tidtable,n_proc,1,status)
              IF (status .LT. 0) THEN
                CALL pvmfperror('pvmgrid',status)
                CALL pvmfexit(status)
                STOP
              END IF
              WRITE(*,*) 'sending to ',tidtable(i)
              CALL pvmfsend(tidtable(i),Init_tag,status)
              IF (status .LT. 0) THEN
                CALL pvmfperror('pvmgrid',status)
                CALL pvmfexit(status)
                STOP
              END IF
55          CONTINUE
60        CONTINUE
65      CONTINUE
70    CONTINUE

      WRITE(*,*) 'pvmgrid exiting'
      CALL pvmfexit(status)

      STOP
      END
```

E.2.1 Processor layout

We need to tell the processors how they are arranged in the 4-D torus. This function, 'proc_pos' does this using a simple algorithm.

```
      INTEGER FUNCTION proc_pos(x, y, z, t, size)
#include "implicit.h"
#include "build_constants.h"
      INTEGER size(0:Ndim-1), x,y,z,t
      INTEGER pos(0:Ndim-1)

      pos(X_index) = MOD(size(X_index) + x , size(X_index))
      pos(Y_index) = MOD(size(Y_index) + y , size(Y_index))
      pos(Z_index) = MOD(size(Z_index) + z , size(Z_index))
      pos(T_index) = MOD(size(T_index) + t , size(T_index))

      proc_pos = pos(X_index) +
     $   (size(X_index) * (pos(Y_index) +
     $   (size(Y_index) * (pos(Z_index) +
     $   (size(Z_index) * pos(T_index))))))

      RETURN
      END
```

E.3 Initialising the communications system

As described in section 3.1 we need to initialise the communications system and find out where we are using the information sent by the loader program.

```
      SUBROUTINE init_comms(grid_size, grid_pos,
     $  proc_id, boss_proc)
#include "implicit.h"
#include "build_constants.h"
#include "build_size.h"
#include "pvmcomms.h"

      INTEGER grid_size(0:Ndim-1)
      INTEGER grid_pos(0:Ndim-1)
      INTEGER proc_id, boss_proc
      INTEGER mess(Block_size)
      INTEGER mytid, parent, bufid, stat, myinst
      INTEGER count,i, t,z,y,x
      INTEGER proc_pos

      CALL pvmfmytid(mytid)
```

```
C If we were using a machine which did not utilise a
C loader program ‘pvmgrid’ to spawn the processes, we could
C whether there was a parent process here, and spawn any others
C needed in the same way as ‘pvmgrid’ does.

C recieve the data from the parent process.

        CALL pvmfrecv(parent,Init_tag,bufid)
        IF (bufid .LT. 0) THEN
          CALL pvmfperror('init_comms',bufid)
          CALL pvmfexit(bufid)
          STOP
        END IF
        CALL pvmfunpack(ITYPE,mess,Block_size,1,stat)
        IF (stat .LT. 0) THEN
          CALL pvmfperror('init_comms',stat)
          CALL pvmfexit(stat)
          STOP
        END IF

C check where we are, who we are, and who the boss is.
        proc_id = mess(My_pid)
        myid = proc_id
        boss_proc = mess(Boss_pid)
        grid_size(X_index) = mess(X_size)
        grid_size(Y_index) = mess(Y_size)
        grid_size(Z_index) = mess(Z_size)
        grid_size(T_index) = mess(T_size)
        grid_pos(X_index) = mess(X_pos)
        grid_pos(Y_index) = mess(Y_pos)
        grid_pos(Z_index) = mess(Z_pos)
        grid_pos(T_index) = mess(T_pos)
        n_proc=grid_size(X_index) * grid_size(Y_index) *
     $      grid_size(Z_index) * grid_size(T_index)
        CALL pvmfunpack(ITYPE,tidtable,n_proc,1,stat)
        IF (stat .LT. 0) THEN
          CALL pvmfperror('init_comms',stat)
          CALL pvmfexit(stat)
          STOP
        END IF

C set up the neighbour table (see next code fragment)
      CALL init_bound(proc_id,grid_pos,grid_size)

C wait for everyone to catch up.
      CALL pvmfsetopt(PVMFASTBARR,1,stat)
      CALL pvmfbarrier(g_group,n_proc,stat)
      RETURN
      END
C}}}
```

The neighbour tables are set up so that the torus is implemented in software.

```
      SUBROUTINE init_bound(proc_id, grid_pos, grid_size)
#include "implicit.h"
#include "build_constants.h"
#include "build_size.h"
#include "pvmcomms.h"

      INTEGER proc_id
      INTEGER grid_pos(0:Ndim-1)
      INTEGER grid_size(0:Ndim-1)
      INTEGER i, n
C     functions
      INTEGER proc_pos

      DO 15, i=0,Ndir-1
        send_ok(i) = .TRUE.
        dir_tag(i) = 0
15    CONTINUE

      DO 20, i=0,Ndim-1
C define the neighbour in the positive direction.
        grid_pos(i) = grid_pos(i) + 1
        n = proc_pos(grid_pos(X_index), grid_pos(Y_index),
     $               grid_pos(Z_index), grid_pos(T_index),
     $               grid_size)
        neighbours(i) = tidtable(n)
C define the neighbour in the negative direction.
        grid_pos(i) = grid_pos(i) - 2
        n = proc_pos(grid_pos(X_index), grid_pos(Y_index),
     $               grid_pos(Z_index), grid_pos(T_index),
     $               grid_size)
        neighbours(i+Ndim) = tidtable(n)
        grid_pos(i) = grid_pos(i) + 1
20    CONTINUE

      RETURN
      END
```

E.4 Global sum

```
C{{{  SUBROUTINE g_sum(rval)
C
C ~g_sum  single precision global sum
C
      SUBROUTINE g_sum(rval)
#include "implicit.h"
#include "messages.h"
```

```
#include "build_constants.h"
#include "build_size.h"
#include "pvmcomms.h"

      Fpoint rval
      Fpoint sum, tmp
      INTEGER i, parent, son, bufid,stat

C If there's only one process, we don't have much to do !
      IF(n_proc .EQ. 1) RETURN

      sum = rval
C{{{ get from sons - we are using a binary tree,
C (Nbranch=2)
      DO 10, i=1,Nbranch
        son= (Nbranch*myid) + i
        IF (son .LT. n_proc) THEN
C we do have a son, receive message into tmp
          CALL pvmfrecv(tidtable(son),Fgsum_tag,bufid)
          IF (bufid .LT. 0) THEN
            CALL pvmfperror('g_sum',bufid)
            CALL pvmfexit(bufid)
            STOP
          END IF
          CALL pvmfunpack(FTYPE,tmp,1,1,stat)
          IF (stat .LT. 0) THEN
            CALL pvmfperror('g_sum',stat)
            CALL pvmfexit(stat)
            STOP
          END IF
C add to cumulative sum
          sum = sum + tmp
        END IF
10    CONTINUE
C}}}

C check we are not at the top of the tree, then
C send cumulative sum to the parent.
      IF (myid .NE. 0) THEN
        parent=(myid-1)/Nbranch
C{{{   send to parent
        CALL pvmfinitsend(Encoding,bufid)
        IF (bufid .LT. 0) THEN
          CALL pvmfperror('g_sum',bufid)
          CALL pvmfexit(bufid)
          STOP
        END IF
        CALL pvmfpack(FTYPE,sum,1,1,stat)
        IF (stat .LT. 0) THEN
          CALL pvmfperror('g_sum',stat)
          CALL pvmfexit(stat)
```

```
          STOP
        END IF
        CALL pvmfsend(tidtable(parent),Fgsum_tag,stat)
        IF (stat .LT. 0) THEN
          CALL pvmfperror('g_sum',stat)
          CALL pvmfexit(stat)
          STOP
        END IF
C}}}

#ifdef TREE_SEND
C we want to send the final result back down the tree.
C{{{   receive from parent, only executes this bit if
C we are not process 0.
        CALL pvmfrecv(tidtable(parent),Fgsum_tag,bufid)
        IF (bufid .LT. 0) THEN
          CALL pvmfperror('g_sum',bufid)
          CALL pvmfexit(bufid)
          STOP
        END IF
        CALL pvmfunpack(FTYPE,sum,1,1,stat)
        IF (stat .LT. 0) THEN
          CALL pvmfperror('g_sum',stat)
          CALL pvmfexit(stat)
          STOP
        END IF
C}}}
      END IF
C{{{ send back down tree
      DO 20, i=1,Nbranch
        son= (Nbranch*myid) + i
        IF (son .LT. n_proc) THEN
C         send message
          CALL pvmfinitsend(Encoding,bufid)
          CALL pvmfpack(FTYPE,sum,1,1,stat)
          IF (stat .LT. 0) THEN
            CALL pvmfperror('g_sum',stat)
             CALL pvmfexit(stat)
            STOP
          END IF
          CALL pvmfsend(tidtable(son),Fgsum_tag,stat)
          IF (stat .LT. 0) THEN
            CALL pvmfperror('g_sum',stat)
            CALL pvmfexit(stat)
            STOP
          END IF
        END IF
20    CONTINUE
C}}}
#else
C otherwise use the global set routine to broadcast the
```

```
C result.
      END IF
      CALL g_set(sum)
#endif

      rval = sum
      RETURN
      END
C}}}
```

E.5 Global set

We often want to set a variable on a processors to the value on the boss processor, e.g. a global sum result. This is implemented with a suite of functions, as shown in section 3.1, one of which is 'g_set' which operates on a single-precision real number.

```
      SUBROUTINE g_set(rval)
#include "implicit.h"
#include "messages.h"
#include "build_constants.h"
#include "build_size.h"
#include "grid_def.h"
#include "pvmcomms.h"
      Fpoint rval
      INTEGER bufid, stat

C only one processor, so not much to do!
      IF(n_proc .EQ. 1) RETURN

C     send message
      IF(myid .EQ. 0)THEN
        CALL pvmfinitsend(Encoding,bufid)
        CALL pvmfpack(FTYPE,rval,1,1,stat)
        IF (stat .LT. 0) THEN
          CALL pvmfperror('g_set',stat)
          CALL pvmfexit(stat)
          STOP
        END IF
#ifdef BCAST
        CALL pvmfbcast(g_group,Fgset_tag,stat)
#else
        CALL pvmfmcast(n_proc,tidtable,Fgset_tag,stat)
#endif
        IF (stat .LT. 0) THEN
          CALL pvmfperror('g_set',stat)
```

```
          CALL pvmfexit(stat)
          STOP
        END IF
      ELSE
C receive message
        CALL pvmfrecv(tidtable(boss_id),Fgset_tag,bufid)
        IF (bufid .LT. 0) THEN
          CALL pvmfperror('g_set',bufid)
          CALL pvmfexit(bufid)
          STOP
        END IF
        CALL pvmfunpack(FTYPE,rval,1,1,stat)
        IF (stat .LT. 0) THEN
          CALL pvmfperror('g_set',stat)
          CALL pvmfexit(stat)
          STOP
        END IF
      END IF
      RETURN
      END
```

E.6 Boundary communications

Most array shifting is implemented through the '`start_com`' and '`end_com`' routines described in section 3.1. They are implemented in PVM as follows (only the single-precision real version is shown).

```
      SUBROUTINE fstart_com(length, idir, icmp, ocmp,
     $                       ilen, input, olen, output)
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "messages.h"
#include "pvmcomms.h"
      INTEGER length, idir, icmp, ocmp, ilen, olen
      Fpoint input(0:(icmp*ilen)-1,0:ocmp-1)
      Fpoint output(0:(icmp*olen)-1,0:ocmp-1)
      INTEGER i, bufid, stat

#ifdef BOUND_BARRIER
      CALL pvmfbarrier(g_group,n_proc,stat)
#endif
      IF(.NOT. send_ok(idir))THEN
        CALL error_message('outstanding comms not finished',
     $         'fstart_com',Mess_local,Err_logic_error)
      END IF
```

```
      IF(length .GT. 0)THEN
        CALL pvmfinitsend(Bound_encode,bufid)
        IF (bufid .LT. 0) THEN
          CALL pvmfperror('fstart_com',bufid)
          CALL pvmfexit(bufid)
          STOP
        END IF
        DO 10, i=0,ocmp-1
          CALL pvmfpack(FTYPE,input(0,i),(length*icmp),1,stat)
          IF (stat .LT. 0) THEN
            CALL pvmfperror('fstart_com',stat)
            CALL pvmfexit(stat)
            STOP
          END IF
10      CONTINUE
        dir_tag(idir)=dir_tag(idir)+1
        CALL pvmfsend(neighbours(idir),
     $    dir_tag(idir)+Bound_base,stat)
        IF (stat .LT. 0) THEN
          CALL pvmfperror('fstart_com',stat)
          CALL pvmfexit(stat)
          STOP
        END IF
      END IF
#ifdef BOUND_BARRIER
      CALL pvmfbarrier(g_group,n_proc,stat)
#endif

#ifdef NO_OVERLAP
C complete communications in the start call, do not
C wait for the end_com.
      CALL real_fend_com(length, idir, icmp, ocmp,
     $                     ilen, input, olen, output)
#endif
      RETURN
      END

      SUBROUTINE fend_com(length, idir, icmp, ocmp,
     $                     ilen, input, olen, output)
#ifdef NO_OVERLAP
C complete communications in the start call
C instead of waiting for a separate end call.
#include "implicit.h"

      INTEGER length, idir, icmp, ocmp, ilen, olen
      Fpoint input(0:(icmp*ilen)-1,0:ocmp-1)
      Fpoint output(0:(icmp*olen)-1,0:ocmp-1)

      RETURN
      END
```

```
      SUBROUTINE real_fend_com(length, idir, icmp, ocmp,
     $                       ilen, input, olen, output)
#endif
C}}}
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "messages.h"
#include "pvmcomms.h"

      INTEGER length, idir, icmp, ocmp, ilen, olen
      INTEGER bufid, stat
      Fpoint input(0:(icmp*ilen)-1,0:ocmp-1)
      Fpoint output(0:(icmp*olen)-1,0:ocmp-1)
      INTEGER i, rdir

#ifdef BOUND_BARRIER
      CALL pvmfbarrier(g_group,n_proc,stat)
#endif
      IF(length .GT. 0)THEN
        rdir = MOD(idir + Ndim, Ndir)
        CALL pvmfrecv(neighbours(rdir),
     $    dir_tag(idir)+Bound_base,bufid)
        IF (bufid .LT. 0) THEN
          CALL pvmfperror('fend_com',bufid)
          CALL pvmfexit(bufid)
          STOP
        END IF
        dir_tag(idir) = dir_tag(idir) - 1
        DO 10, i=0,ocmp-1
          CALL pvmfunpack(FTYPE,output(0,i),
     $      (icmp*length),1,stat)
          IF (stat .LT. 0) THEN
            CALL pvmfperror('fend_com',stat)
            CALL pvmfexit(stat)
            STOP
          END IF
10      CONTINUE
      END IF
C     once recv is started must flush all outstanding data.
      send_ok(idir) = (dir_tag(idir) .EQ. 0)

#ifdef BOUND_BARRIER
      CALL pvmfbarrier(g_group,n_proc,stat)
#endif
      RETURN
      END
```

E.7 Parallel file I/O support routines

When we perform parallel file I/O we have to move blocks of data to the boss processor, which performs all I/O. The routines to do are '`block_push`' and '`block_pull`'.

```
      SUBROUTINE block_push(pos,size,buff)
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "messages.h"
#include "pvmcomms.h"
      INTEGER pos, size
      Fpoint buff(0:size-1)

      INTEGER bufid, stat, rsize

C wait for a request for data.
      CALL pvmfrecv(tidtable(pos),Pio_req_tag,bufid)
      IF (bufid .LT. 0) THEN
        CALL pvmfperror('block_push',bufid)
        CALL pvmfexit(bufid)
        STOP
      END IF
      CALL pvmfunpack(ITYPE,rsize,1,1,stat)
      IF (stat .LT. 0) THEN
        CALL pvmfperror('block_pull',stat)
        CALL pvmfexit(stat)
        STOP
      END IF
      IF(rsize .NE. size)THEN
        CALL error_message('wrong size message requested',
     $         'block_push',Mess_local,Err_logic_error)
      END IF
      CALL pvmfinitsend(Encoding,bufid)
      IF (bufid .LT. 0) THEN
        CALL pvmfperror('block_push',bufid)
        CALL pvmfexit(bufid)
        STOP
      END IF
      CALL pvmfpack(FTYPE,buff,size,1,stat)
      IF (stat .LT. 0) THEN
        CALL pvmfperror('block_push',stat)
        CALL pvmfexit(stat)
        STOP
      END IF
      CALL pvmfsend(tidtable(pos),Pio_tag,stat)
      IF (stat .LT. 0) THEN
```

```
        CALL pvmfperror('fstart_com',stat)
        CALL pvmfexit(stat)
        STOP
      END IF

      RETURN
      END

      SUBROUTINE block_pull(pos,size,buff)
#include "implicit.h"
#include "build_size.h"
#include "build_constants.h"
#include "messages.h"
#include "pvmcomms.h"
      INTEGER pos, size
      Fpoint buff(0:size-1)

      INTEGER bufid, stat

C  send a request for data
      CALL pvmfinitsend(Encoding,bufid)
      IF (bufid .LT. 0) THEN
        CALL pvmfperror('block_pull',bufid)
        CALL pvmfexit(bufid)
        STOP
      END IF
      CALL pvmfpack(ITYPE,size,1,1,stat)
      IF (stat .LT. 0) THEN
        CALL pvmfperror('block_push',stat)
        CALL pvmfexit(stat)
        STOP
      END IF
      CALL pvmfsend(tidtable(pos),Pio_req_tag,stat)
      IF (stat .LT. 0) THEN
        CALL pvmfperror('fstart_com',stat)
        CALL pvmfexit(stat)
        STOP
      END IF
C     receive the data
      CALL pvmfrecv(tidtable(pos),Pio_tag,bufid)
      IF (bufid .LT. 0) THEN
        CALL pvmfperror('block_pull',bufid)
        CALL pvmfexit(bufid)
        STOP
      END IF
      CALL pvmfunpack(FTYPE,buff,size,1,stat)
      IF (stat .LT. 0) THEN
        CALL pvmfperror('block_pull',stat)
        CALL pvmfexit(stat)
        STOP
      END IF
```

```
      END
```

E.8 Finishing up after the end of the program

When we have finished, we must tidy up cleanly.

```
      SUBROUTINE finish_comms()
#include "implicit.h"
#include "build_constants.h"
#include "build_size.h"
#include "pvmcomms.h"

      INTEGER code

      CALL pvmfexit(code)

      RETURN
      END
```

References and bibliography

[1] R. D. Kenway. Non-perturbative calculations in the standard model. *Rep. Prog. Phys.*, 52, 1989.

[2] A. S. Kronfeld. Lattice QCD. Technical Report Fermilab-conf-92/040-T, Fermilab preprint, 1992. Introductory lectures given at TASI Summer School.

[3] H. J. Rothe. *Lattice Gauge Theories : an introduction.* World Scientific, 1992.

[4] C. Rebbi, editor. *Lattice Gauge Theories and Monte Carlo Simulations.* World Scientific, 1992.

[5] Ta-Pei Cheng and Ling-Fong Li. *Gauge theory of elementary particle physics.* Oxford University Press, 1984.

[6] K. G. Wilson. Confinement of quarks. *Phys. Rev.* , D10:2445, 1974.

[7] B. Sheikholeslami and R. Wohlert. Improved continuum limit lattice action for QCD with Wilson fermions. *Nucl. Phys.*, B259:572–596, 1985.

[8] G. Heatlie et al. The improvement of hadronic matrix elements in Lattice QCD. *Nucl. Phys.*, B352:266–288, 1991.

[9] M. Lüscher and P. Weisz. On-shell Improved Lattice Gauge Theories. *Commun. Math. Phys.*, 97:59–77, 1985.

[10] G. P. Lepage and P. B. Mackenzie. On the viability of lattice perturbation theory. *Phys. Rev.*, D48:2250–2264, 1993. HEP-LAT 9209022, Fermilab-Pub-91/355-T Revised, NSF-ITP-90-227.

[11] K. Binder. Introduction: Theory and “Technical” aspects of Monte Carlo Simulations. In K. Binder, editor, *Monte Carlo Methods in Statistical Physics*, chapter 1, pages 1–45. Springer-Verlag, 1979.

[12] F. James. Monte Carlo in theory and practice. *Rep. Prog. Phys.*, 43:1145–

1189, 1980.

[13] N. Metropolis et al. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087, 1953.

[14] Gyan Bhanot. The Metropolis algorithm. *Rep. Prog. Phys.*, 51:429–457, 1988.

[15] A. D. Kennedy. Progress in Lattice field Theory Algorithms. In *Proceedings of the 1992 Symposium on Lattice Field Theory.* Nuclear Physics B Proc. Supp., 1992.

[16] A. Simpson. *Algorithms for lattice QCD.* PhD thesis, Physics Department, University of Edinburgh, 1991.

[17] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid Monte Carlo. *Phys. Lett.*, 195(2), 1987.

[18] R. Gupta et al. QCD with dynamical fermions. *Phys. Rev.*, D40:2072, 1989.

[19] C. Bernard, A. Soni, and K. Yee. Introduction to lattice gauge fixing and effective quark and gluon masses. Technical Report LSUHEP002-1992, LSU, 1992. Presented at Workshop on QCD Vacuum Structure, Paris, France.

[20] H. Suman and K. Schilling. A comparative study of gauge fixing procedures on the connection machines CM2 and CM5. *Par. Comp.*, 20:975–990, 1994.

[21] C. R. Allton et al. Gauge invariant smearing and matrix correlators using Wilson fermions at $\beta = 6.2$. *Phys. Rev.*, D47:5128–5137, 1993.

[22] S. Collins. Gauge invariant smearing and the extraction of excited state masses using Wilson fermions at $\beta = 6.2$. *Nucl. Phys.*, B (Proc. Suppl.) 30:393–396, 1993.

[23] C. Michael and M. Teper. The glueball spectrum in SU(3). *Nucl. Phys.*, B314:347, 1989.

[24] S. Güsken. A study of smearing techniques for hadron correlation functions. *Nucl. Phys.*, B (Proc. Suppl.) 17:301, 1990.

[25] H. P. Shanahan. *Lattice calculations in heavy hadron physics.* PhD thesis, Department of Physics and Astronomy, University of Edinburgh, 1994.

[26] Michael J. Flynn. Some computer organisations and their effectiveness. *IEEE Transactions on computers*, C-21:948–960, 1972.

[27] M. Kumar, Y. Baransky, and M. Denneau. The GF11 parallel computer. *Parallel Computing*, 19:1393–1412, 1993.

[28] A. Bartolini et al. A hardware implementation of the APE100 architecture. *Int. Jour. Mod. Phys.*, C4(5):969, 1993.

[29] Thinking Machines Corporation, Cambridge, Massachusetts. *CM User's Guide*, version 6.1 edition, October 1991.

[30] N. Christ and A. Terrano. A very fast parallel processor. *IEEE Trans. Comput.*, 33:344, 1984.

[31] F. Butler. Status of the Columbia Parallel Processor. *Nucl. Phys.*, B (Proc. Suppl.) 9:557, 1989.

[32] Y. Oyanagi. New parallel computer project in Japan dedicated to computational physics. *Nucl. Phys.*, B (Proc. Suppl.) 30:299, 1993.

[33] I. Arsenin, D. Chen, N. Christ, R. Edwards, A. Gara, S. Hansen, A. Kennedy, R. Mawhinney, J. Parsons, and J. Sexton. A 0.5 teraflops machine optimised for lattice QCD. *Nucl. Phys.*, B (Proc. Suppl.) 34:820–822, 1994.

[34] J. W. Negele. QCD teraflops computer. *Nucl. Phys.*, B (Proc. Suppl.) 30:295, 1993.

[35] Y. Iwasaki et al. Status of QCDPAX. *Nucl. Phys.*, B (Proc. Suppl.) 17:259, 1990.

[36] Y. Iwasaki et al. QCDPAX: Present status and first physical results. *Nucl. Phys.*, B (Proc. Suppl.) 20:141, 1991.

[37] M. Fischler et al. The Fermilab lattice supercomputer project. *Nucl. Phys.*, B (Proc. Suppl.) 9:571, 1989.

[38] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, January 1992.

[39] Wilfred Oed. *The Cray Research Massively Parallel Processor System CRAY T3D.* Cray Research GmbH, November 1993.

[40] Intel. Paragon user's guide. WWW URL http://www.ccsf.caltech.edu/-paragon/man.html.

[41] K. Akemi et al. QCD on the highly parallel computer AP1000. *Nucl. Phys.*, B (Proc. Suppl.) 26:644, 1992.

[42] S. Ohta. Towards lattice QCD simulation on AP1000. *Nucl. Phys.*, B (Proc. Suppl.) 26:647, 1992.

[43] James Rothnie. Overview of the KSR1 computer system. Technical Report TR9202001, Kendall Square Research, March 1992.

[44] Norman H. Christ. QCD machines. *Nucl. Phys.*, B (Proc. Suppl.) 9:549, 1989.

[45] Norman H. Christ. QCD machines — present and future. *Nucl. Phys.*, B (Proc. Suppl.) 20:129, 1991.

[46] D. Weingarten. Parallel QCD machines. *Nucl. Phys.*, B (Proc. Suppl.) 26:126, 1992.

[47] E. Marinari. A review talk about computers and theoretic physics. *Nucl. Phys.*, B (Proc. Suppl.) 30:122, 1993.

[48] Y. Iwasaki. Computers for lattice field theories. *Nucl. Phys.*, B (Proc. Suppl.) 34:78, 1994.

[49] A. Trew and G. Wilson, editors. *Past, Present, Parallel — A survey of available parallel computing systems.* Springer-Verlag, 1991.

[50] Almasi and Gottlieb. *Highly Parallel Computing.* Benjamin-Cummings, 2nd edition, 1994.

[51] Reinhold P. Weicher. A detailed look at some popular benchmarks. *Parallel Computing*, 17:1153–1172, 1991. Special issue: Benchmarking of High Performance Supercomputers.

[52] M. Berry, G. Cybenko, and J. Larson. Scientific benchmark characterizations. *Parallel Computing*, 17:1173–1194, 1991. Special issue: Benchmarking of High Performance Supercomputers.

[53] David Levine, David Callahan, and Jack Dongarra. A comparative study of automatic vectorising compilers. *Parallel Computing*, 17:1223–1244, 1991. Special issue: Benchmarking of High Performance Supercomputers.

[54] A. J. G. Hey. The Genesis distributed memory benchmarks. *Parallel Computing*, 17:1275–1283, 1991. Special issue: Benchmarking of High Performance Supercomputers.

[55] D. Bailey et al. The NAS parallel benchmarks. WWW URL http://-www.nas.nasa.gov/RNR/Parallel/NPB/NPBindex.html.

[56] R. W. Hockney and C. R. Jesshope. *Parallel Computers 2: Architecture, Programming and Algorithms.* Adam Hilger, Bristol and Philadelphia, 1988.

[57] A. Bartolini et al. The software of the APE100 processor. *Int. Jour. Mod. Phys.*, C4(5):955–967, 1993.

[58] Thinking Machines Corporation, Cambridge, Mass. *CM Fortran Programmers Guide.* Version 1.1.

[59] Thinking Machines Corporation, Cambridge, Mass. *CM Fortran Reference Manual.* Version 1.0.

[60] Brainerd, Goldberg, and Adams. *Programmer's guide to Fortran 90.* McGraw-Hill/Intertext, 1990.

[61] B. J. N. Wylie and M. G. Norman. High Performance Fortran: A perspective. Technical Report EPCC-TN92-05, Edinburgh Parallel Computing Centre, 1992.

[62] HPF Forum. *High Performance Fortran Language Specification.* Rice University, Houston Texas, 1993. Version 1.0, May 3. Available by anonymous ftp from "titan.cs.rice.edu".

[63] C. Koelbel and P. Mehrota. An overview of High Performance Fortran. *Fortran Forum*, 11(4), December 1992.

[64] David B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1), February 1993.

[65] High Performance Fortran Forum. High Performance Fortran Specification. *Fortran Forum*, 12(4), December 1993.

[66] High Performance Fortran Forum. WWW home page. WWW URL http://www.erc.msstate.edu/hpff/home.html.

[67] O. A. McBryan. An overview of message passing environments. *Parallel Computing*, 20(4):417, April 1994.

[68] R. Hempel, H.-C. Hoppe, and A. Supalov. A proposal for a PARMACS library interface. GMD, Postfach 1316, D-5205 Sankt Augustin 1, Germany, October 1992.

[69] R. Hempel. *The ANL/GMD Macros (PARMACS) in FORTRAN for Portable Parallel Programming using the Message Passing Programming Model — User's Guide and Reference Manual*, 1991. Version 5.1.

[70] R. Calkin et al. Portable programming with the PARMACS message-passing library. *Parallel Computing*, 20(4):615, April 1994.

[71] A. Beguelin et al. A user's guide to PVM parallel virtual machine. Technical Report TM-11826, Oak Ridge National Laboratory, July 1991.

[72] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Mancheck. The PVM concurrent computing system: evolution, experiences and trends. *Parallel Computing*, 20(4):531, April 1994.

[73] D. W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657,

April 1994.

[74] MPI Forum. *Document for a Standard Message-Passing Interface*, November 1993. DRAFT. Available by sending message "send index from mpi" to "netlib@ornl.gov".

[75] MPI Forum. Document for a Standard Message-Passing Interface. WWW URL http://www.mcs.anl.gov/mpi/mpi-report.html.

[76] Argonne National Laboratory. ANL MPI implementation. WWW URL http://www.mcs.anl.gov/mpi/index.html.

[77] Edinburgh Parallel Computing Centre. MPI implementation for CHIMP v2.1. Available by anonymous ftp from host: ftp.epcc.ed.ac.uk, directory: /pub/chimp/release, file: chimp.tar.Z.

[78] Ohio Supercomputer Centre. MPI implementation for LAM. Available by anonymous ftp from host: tbag.osc.edu, directory: /pub/lam.

[79] R. Lusk, B. Gropp, and A. Skjellum. *Using MPI.* MIT Press. To be released.

[80] U. Gärtel et al. Two strategies in parallel computing: porting existing software versus developing new parallel algorithms — two examples. *Future Generation Computer Systems (FGCS)*, 10:257–262, 1994.

[81] D. M. Pase, T. MacDonald, and A. Meltzer. *MPP Fortran Programming Model.* Cray Research Inc., October 1993. Available by anonymous ftp from ftp.cray.com, directory: /product-info/mpp.

[82] L. W. Tucker and A. Mainwaring. CMMD: active messages on the CM-5. *Parallel Computing*, 20(4):481, April 1994.

[83] S. Booth. Parallel file access. Internal report, June 1994.

[84] N. Stanford, H. Hoeber, and N. Hazel. *Pure Gauge Configuration Generation on the CM-200*, 1992. Version 1.

[85] Stewart C. Loken. Software Engineering: What do experiments need? In C. Verkerk and W. Wojcik, editors, *Proceedings of the International*

Conference on Computing in High Energy Physics, pages 87–89. CERN, 1992. CERN 92-07.

[86] I. Sommerville. *Software Engineering.* Addison-Wesley, fourth edition, 1992.

[87] J. A. McDermid. *Software Engineer's Reference Book.* Butterworth-Heinemann, 1991.

[88] E. Yourdon. *Modern Structured Analysis.* Prentice Hall International, 1989.

[89] W. W. Royce. Managing the development of large software systems. In *WESTCON*, 1970.

[90] UKQCD. *Massively Parallel Processor (MPP) Project, Volume 1: General Documentation.* University of Edinburgh, 1994.

[91] UKQCD. *Massively Parallel Processor (MPP) Project, Volume 2: Message Passing version.* University of Edinburgh, 1994.

[92] UKQCD. *Massively Parallel Processor (MPP) Project, Volume 3: High Performance Fortran (HPF) version.* University of Edinburgh, 1994.

[93] UKQCD. *Massively Parallel Processor (MPP) Project, Volume 4: C Workstation codes.* University of Edinburgh, 1994.

[94] US DoD. Military Standard, Supplement to the American National Standard X3.9-1978. Technical Report MIL-STD-1753, US Department of Defense, 1978.

[95] C. A. Addison et al. The GENESIS distributed-memory benchmarks. *Computer Benchmarks*, 1993.

[96] R. Hockney. A framework for benchmark performance analysis. *Supercomputer*, 48:9–22, 1992.

[97] G. Marsaglia. A current view of random number generators. In *Computer Science and Statistics 16th Symposium on the Interface*, 1984.

[98] I. Vattulainen, K. Kankaala, J. Saarinen, and T. Ala-Nissila. A Com-

parative Study of Some Pseudorandom Number Generators. Technical Report HU-TFT-93-22, HEP-LAT 9304008, Research Institute for Theoretical Physics, University of Helsinki, 1993.

[99] P. D. Coddington. Analysis of random number generators using Monte Carlo simulation. Technical Report SCCS-526, COND-MAT 930917, Northeast Parallel Architectures Centre, Syracuse University, September 1993.

[100] I. Vattulainen, T. Ala-Nissila, and K. Kankaala. Physical tests for random numbers in simulations. Technical Report HU-TFT-94-2, COND-MAT 9406054, Research Institute for Theoretical Physics, University of Helsinki, 1994.

[101] Stephen Booth. Private Communication.

[102] Peer Ueberholtz. Private Communication.

[103] G. Marsaglia and A. Zaman. Towards a universal random number generator. *Ann. Appl. Prob.*, 1:462, 1991.

[104] Press et al. *Numerical Recipes in C : the art of scientific computing.* Cambridge University Press, second edition, 1992.

[105] D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, 2nd edition, 1981.

[106] G. Marsaglia and L. Tsay. Matrices and the Structure of Random Number Sequences. *Linear Algebra and its Applications*, 67:147–158, 1985.

[107] F. James. A review of pseudorandom number generators. *Comp. Phys. Comm.*, 60, 1990.

[108] M. Lüscher. A portable high-quality random number generator for lattice field theory simulations. *Comput. Phys. Commun.*, 79(DESY 93-133, hep-lat 9309020):100–110, 1994.

[109] M. Lüscher. A random number generator for the ape-100 parallel computer. Internal report. Available by anonymous ftp from 141.108.16.27

in directory "pub/random"., June 1993.

[110] Cray Research, Inc. *CRAY RANF manual page.* SR-2138, version 1.0.

[111] K. Kankaala, T. Ala-Nissila, and I. Vattulainen. Bit Level Correlations in Some Pseudorandom Number Generators. Technical Report HU-TFT-93-41, HEP-LAT 9308018, Research Institute for Theoretical Physics, University of Helsinki, 1993.

[112] I. Vattulainen, K. Kankaala, J. Saarinen, and T. Ala-Nissila. Influence of Implementation on the Properties of Pseudorandom Number Generators with a Carry Bit. Technical Report HU-TFT-93-33, HEP-LAT 9306008, Research Institue for Theoretical Physics, University of Helsinki, 1993.

[113] Alan M. Ferrenberg, D. P. Landau, and Y. Joanna Wong. Monte carlo simulations: hidden errors from "good" random number generators. *Phys. Rev. Lett.*, 69(23), December 1992.

[114] S. L. Anderson. Random number generators on vector supercomputers and other advanced architectures. *SIAM Rev.*, 32(2):221–251, June 1990.

[115] I. Deak. Uniform random number generators for parallel computers. *Parallel Computing*, 15, 1990.

[116] S. Aluru, G. M. Prabhu, and J. Gustafson. A random number generator for parallel computers. *Parallel Computing*, 18:839, 1992.

[117] R. C. Edgar, L. McCrossen, and K. J. M. Moriarty. The specific heat of SU(3) lattice gauge theory. *J. Phys.*, G: Nucl. Phys. 7:L85–L88, 1981.

[118] R. Balian, J. M. Drouffe, and C. Itzykson. Gauge fields on a lattice. III. strong-coupling expansions and transition points. *Phys. Rev.*, D11(8):2104–2119, April 1975.

[119] R. Balian, J. M. Drouffe, and C. Itzykson. Erratum: gauge fields on a lattice. III. strong-coupling expansions and transition points. *Phys. Rev.*, D19(8):2514–2515, April 1979.

[120] D. B. Carpenter and C. F. Baillie. Free fermion propagators and lattice

finite size effects. *Nucl. Phys.*, B260, 1985.

[121] Craig McNeile and Rob Baxter. Private Communication.

[122] R. Hockney and M. Berry. Public international benchmarks for parallel computers: report 1. Technical report, PARKBENCH Committee, February 1994. Available by rcp from “anon@netlib2.cs.utk.edu:-parkbench/parkbench.ps”.

[123] I. J. R. Aitchison and A. J. G. Hey. *Gauge Theories in Particle Physics.* Adam Hilger, 2nd edition, 1989.

[124] N. Cabibbo and E. Marinari. A new method for updating $SU(N)$ matrices in computer simulations of gauge theories. *Phys. Lett.*, B119, 1982.

[125] A. D. Kennedy and B. J. Pendleton. Improved heatbath method for Monte Carlo calculations in lattice gauge theory. *Phys. Lett.*, B156, 1985.

[126] J. H. Wilkinson and C. Reinsch. *Linear Algebra*, volume II. Springer-Verlag, 1971.

[127] Patrick D. Surrey and B. J. N. Wylie. High Performance Fortran Migration (HPF and $\subset$HPF) via CM-Fortran. Technical Report EPCC-TN-93-01, Edinburgh Parallel Computing Centre, January 1993. An early report, has several omissions fixed by private correspondance.