Conjugate Gradient for Domain Wall Fermions with 4-d EO preconditioning

Andrew Pochinsky

September 14, 2006

Abstract

This document presents an implementation of a conjugate gradient solver for the Domain Wall Fermion Dirac operator. The code targets SciDAC machines that implement QMP protocol.

This version is suboptimal in overlapping communication and computation because of defficiencies of the QMP implementation.

1   ⟨*Version* 1⟩≡
```
static const char *version = "Version 1.3.3";
```
This code is used in chunk 14b.

Contents

1 INTRODUCTION

The code below interfaces with a Chroma-like upper level environment to provide file access and machine initialization and configuration. In fact, this file is an implementation of a level 3 routine for solving the Dirac equation. There are some restrictions on input parameters imposed by the algorithm and a particular way the vector hardware is used by the implementation. There are the following restrictions on the lattice geometry:

- All four-dimensional extends of the lattice should be even. This is required for even-odd decomposition used in the preconditioner.

- The fifth-dimension extend should be a multiple of 4 or 2 depending on the implementation. It is needed for efficient use of vector registers and simplification of vector code.

- The implementation supports up to four dimensional tori as a network topology.

Because of many issues involved in optimizing the code, it is advantageous to put together some definitions and outline here the optimization strategy used.

1.1 Definitions

Lattice extend is the total size of the lattice in a given dimension.

Network is the logical topology of the network presented by QMP to the apptication.

Node is a computing element in the network which runs an execution thread. For this implementation we assume that there is one compute node per network location. If an SMP is used, it is the responsibility of QMP to provide a proper abstraction to the application.

Sublattice is the part of the lattice that resides on a compute node.

Site is a point on the lattice.

1.2 Optimization Strategy

For this code we assume that scarcity of resources makes us run the inverter on a small number of nodes compared to the number of sites. This is based on the observation that physics needs grow faster than SciDAC budget and computer deployment plans. We also assume, that the current trend in computer industry persists, namely, that the processors grow faster while memory speed and latency continues to lag in relatice terms. We also want a solver whose performance would degrade gracefully when one moves out of the optimization domain. In particular, we impose no limitation on the size of sublattice. There is even no requirement that all sublattices should be of the same size.

For the optimization sweetspot, we assume that the typical problem is too large to fit into the cache hierarhy and mostly resides in main memory. This is true now for existing and proposed clusters and is like to remain true for the future, since large scale computations tend to use larger lattices most of the time.

2 PHYSICS

Here we give the fermion action and γ-matrix and other conventions.

2.1 Dirac Operator

The Domain Wall Fermion Dirac operator is

$$\chi_{s,x} = D\psi \quad = \quad M_0\psi_{s,x} + \sum_\mu \left((1+\gamma_\mu)U_{x,\mu}\psi_{s,x+\hat{\mu}} + (1-\gamma_\mu)U^\dagger_{x-\hat{\mu},\mu}\psi_{s,x-\hat{\mu}} \right)$$

$$+(1+\gamma_5)M_s^{(+)}\psi_{s+1,x} + (1-\gamma_5)M_s^{(-)}\psi_{s-1,x}$$

where

$$M_s^{(+)} = \left\{ \begin{array}{ll} 1, & \text{if} \quad s < N_s - 1 \\ -m_f, & \text{if} \quad s = N_s - 1 \end{array} \right.$$

and

$$M_s^{(-)} = \left\{ \begin{array}{ll} 1, & \text{if} \quad s > 0 \\ -m_f, & \text{if} \quad s = 0 \end{array} \right.$$

We also assume that $\psi_{N_s,x} = \psi_{0,x}$ and $\psi_{-1,x} = \psi_{N_s-1,x}$.

2.2 Gamma matrices

We use the same γ-matrix basis as Chroma to simplify conversion between two codes. The choice below could be changed with a few modifications to the rest of the code, if γ_5 is kept diagonal, and one of other γ-matrices has all nonzero entries equal to +1.

$$\gamma_0 = -\sigma_2 \otimes \sigma_1 = \begin{pmatrix} 0 & i\sigma_1 \\ -i\sigma_1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}$$

First, the projector and the reconstructor for $(1 + \gamma_0)$:

6a   ⟨*Build* $(1 + \gamma_0)$ *projection of* `*f` *in* `*g` 6a⟩≡

```
g->f[0][c].re = f->f[0][c].re - f->f[3][c].im;
g->f[0][c].im = f->f[0][c].im + f->f[3][c].re;
g->f[1][c].re = f->f[1][c].re - f->f[2][c].im;
g->f[1][c].im = f->f[1][c].im + f->f[2][c].re;
```
This code is used in chunks 67b, 69d, 70a, 72c, and 73a.

6b   ⟨*Unproject and accumulate* $(1 + \gamma_0)$ *link* 6b⟩≡

```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[3][c].im -= hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[3][c].re += hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[2][c].im -= hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[2][c].re += hh[k].f[1][c].im;
```
This code is used in chunks 70b, 73b, and 75b.

Now, same for $(1 - \gamma_0)$:

6c   ⟨*Build* $(1 - \gamma_0)$ *projection of* `*f` *in* `*g` 6c⟩≡

```
g->f[0][c].re = f->f[0][c].re + f->f[3][c].im;
g->f[0][c].im = f->f[0][c].im - f->f[3][c].re;
g->f[1][c].re = f->f[1][c].re + f->f[2][c].im;
g->f[1][c].im = f->f[1][c].im - f->f[2][c].re;
```
This code is used in chunks 67c, 69d, 70a, 72c, and 73a.

6d   ⟨*Unproject and accumulate* $(1 - \gamma_0)$ *link* 6d⟩≡

```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[3][c].im += hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[3][c].re -= hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[2][c].im += hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[2][c].re -= hh[k].f[1][c].im;
```
This code is used in chunks 70b, 73b, and 75b.

$$\gamma_1 = \sigma_2 \otimes \sigma_2 = \begin{pmatrix} 0 & -i\sigma_2 \\ i\sigma_2 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$$

First, the projector and the reconstructor for $(1 + \gamma_1)$:

6e   ⟨*Build* $(1 + \gamma_1)$ *projection of* `*f` *in* `*g` 6e⟩≡

```
g->f[0][c].re = f->f[0][c].re - f->f[3][c].re;
g->f[0][c].im = f->f[0][c].im - f->f[3][c].im;
g->f[1][c].re = f->f[1][c].re + f->f[2][c].re;
g->f[1][c].im = f->f[1][c].im + f->f[2][c].im;
```
This code is used in chunks 67d, 69d, 70a, 72c, and 73a.

6f   ⟨*Unproject and accumulate* $(1 + \gamma_1)$ *link* 6f⟩≡

```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[3][c].re -= hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[3][c].im -= hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[2][c].re += hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[2][c].im += hh[k].f[1][c].im;
```
This code is used in chunks 70b, 73b, and 75b.

Now, same for $(1 - \gamma_1)$:

6g   ⟨*Build* $(1 - \gamma_1)$ *projection of* `*f` *in* `*g` 6g⟩≡

```
g->f[0][c].re = f->f[0][c].re + f->f[3][c].re;
g->f[0][c].im = f->f[0][c].im + f->f[3][c].im;
g->f[1][c].re = f->f[1][c].re - f->f[2][c].re;
g->f[1][c].im = f->f[1][c].im - f->f[2][c].im;
```
This code is used in chunks 67e, 69d, 70a, 72c, and 73a.

7   ⟨*Unproject and accumulate* $(1 - \gamma_1)$ *link* 7⟩≡

```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[3][c].re += hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[3][c].im += hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[2][c].re -= hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[2][c].im -= hh[k].f[1][c].im;
```

This code is used in chunks 70b, 73b, and 75b.

$$\gamma_2 = -\sigma_2 \otimes \sigma_3 = \begin{pmatrix} 0 & i\sigma_3 \\ -i\sigma_3 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}$$

First, the projector and the reconstructor for $(1 + \gamma_2)$:

8a ⟨*Build* $(1 + \gamma_2)$ *projection of* `*f` *in* `*g` 8a⟩≡

```
g->f[0][c].re = f->f[0][c].re - f->f[2][c].im;
g->f[0][c].im = f->f[0][c].im + f->f[2][c].re;
g->f[1][c].re = f->f[1][c].re + f->f[3][c].im;
g->f[1][c].im = f->f[1][c].im - f->f[3][c].re;
```
This code is used in chunks 68–70, 72c, and 73a.

8b ⟨*Unproject and accumulate* $(1 + \gamma_2)$ *link* 8b⟩≡

```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[2][c].im -= hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[2][c].re += hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[3][c].im += hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[3][c].re -= hh[k].f[1][c].im;
```
This code is used in chunks 70b, 73b, and 75b.

Now, same for $(1 - \gamma_2)$:

8c ⟨*Build* $(1 - \gamma_2)$ *projection of* `*f` *in* `*g` 8c⟩≡

```
g->f[0][c].re = f->f[0][c].re + f->f[2][c].im;
g->f[0][c].im = f->f[0][c].im - f->f[2][c].re;
g->f[1][c].re = f->f[1][c].re - f->f[3][c].im;
g->f[1][c].im = f->f[1][c].im + f->f[3][c].re;
```
This code is used in chunks 68–70, 72c, and 73a.

8d ⟨*Unproject and accumulate* $(1 - \gamma_2)$ *link* 8d⟩≡

```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[2][c].im += hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[2][c].re -= hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[3][c].im -= hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[3][c].re += hh[k].f[1][c].im;
```
This code is used in chunks 70b, 73b, and 75b.

$$\gamma_3 = \sigma_1 \otimes 1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

First, the projector and the reconstructor for $(1 + \gamma_3)$:

8e ⟨*Build* $(1 + \gamma_3)$ *projection of* `*f` *in* `*g` 8e⟩≡

```
g->f[0][c].re = f->f[0][c].re + f->f[2][c].re;
g->f[0][c].im = f->f[0][c].im + f->f[2][c].im;
g->f[1][c].re = f->f[1][c].re + f->f[3][c].re;
g->f[1][c].im = f->f[1][c].im + f->f[3][c].im;
```
This code is used in chunks 68–70, 72c, and 73a.

8f ⟨*Unproject* $(1 + \gamma_3)$ *link* 8f⟩≡

```
qs->f[0][c].re = hh[k].f[0][c].re; qs->f[2][c].re = hh[k].f[0][c].re;
qs->f[0][c].im = hh[k].f[0][c].im; qs->f[2][c].im = hh[k].f[0][c].im;
qs->f[1][c].re = hh[k].f[1][c].re; qs->f[3][c].re = hh[k].f[1][c].re;
qs->f[1][c].im = hh[k].f[1][c].im; qs->f[3][c].im = hh[k].f[1][c].im;
```
This code is used in chunks 70b, 73b, and 75b.

Now, same for $(1 - \gamma_3)$:

9a $\langle$*Build* $(1 - \gamma_3)$ *projection of* `*f` *in* `*g` 9a$\rangle\equiv$

```
g->f[0][c].re = f->f[0][c].re - f->f[2][c].re;
g->f[0][c].im = f->f[0][c].im - f->f[2][c].im;
g->f[1][c].re = f->f[1][c].re - f->f[3][c].re;
g->f[1][c].im = f->f[1][c].im - f->f[3][c].im;
```

This code is used in chunks 68–70, 72c, and 73a.

9b $\langle$*Unproject and accumulate* $(1 - \gamma_3)$ *link* 9b$\rangle\equiv$

```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[2][c].re -= hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[2][c].im -= hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[3][c].re -= hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[3][c].im -= hh[k].f[1][c].im;
```

This code is used in chunks 70b, 73b, and 75b.

3 CONJUGATE GRADIENT

Here we develop the algorithm used in the solver.

3.1 Standard Algorithm

The basic conjugate gradient algorithm 1 is simple. Its only requirement is that matrix A is hermitian. Otherwise, it appears suited for DWF better than other iterative solvers.

Input: A, the matrix
Input: b, the right hand side of the linear equation
Input: x_0, an initial guess
Input: n, the maximum number of iterations
Input: ϵ, required precision
Output: x, approximate solution
Output: ρ, final residue
Output: k, number of iterations used
begin
    $x \leftarrow x_0$
    $p \leftarrow r \leftarrow b - Ax$
    $\rho \leftarrow \langle r, r \rangle$
    $k \leftarrow 0$
    while $\rho > \epsilon$ *or* $k < n$ **do**
        $q \leftarrow Ap$
        $\alpha \leftarrow \rho / \langle p, q \rangle$
        $r \leftarrow r - \alpha q$
        $x \leftarrow x + \alpha p$
        $\gamma \leftarrow \langle r, r \rangle$
        if $\gamma < \epsilon$ **then**
            $\rho \leftarrow \gamma$
            break
        end
        $\beta \leftarrow \gamma / \rho$
        $\rho \leftarrow \gamma$
        $p \leftarrow r + \beta p$
        $k \leftarrow k + 1$
    end
    return x, ρ, k.
end

Algorithm 1: Generic Conjugate Gradient Solver

3.2 Overlap Opportunities

Our approach to overlapping computations with communications is to break the sublattice into boundary and inside pieces. After that, we first compute $(1 \pm \gamma_\mu)$ projections on the boundary and start send and receive operations. While communications are in progress, everything is computed on the inside nodes of the sublattice. Once receive is complete, we compute the operator on the boundary sites. Such an approach helps to improve temporal locality (and, therefore, cache utilization) at the expence of losing the ability of overlap if one of the sublattice dimensions is 2. However, it is unlikely that we could afford a large enough cluster to be forced into this corner of the parameter space.

3.3 Non-hermitial Matrix

Hermiticity of M is the only obstacle in applying algorithm 1 directly to our problem $M\psi = \eta$. This issue can be easily resolved by multiplying both sides by $M^\dagger$. However, instead of using algorithm 1 with $A = M^\dagger M$, it is better to keep M and $M^\dagger$ separate—this makes it possible to hide one of the global sum computations, thus improving machine size scaling. Algorithm 2 is what we use in the solver.

Input: M, the matrix
Input: b, the right hand side of the linear equation
Input: x_0, an initial guess
Input: n, the maximum number of iterations
Input: ϵ, required precision
Output: x, approximate solution
Output: ρ, final residue
Output: k, number of iterations used
begin
    $x \leftarrow x_0$
    $p \leftarrow r \leftarrow b - M^\dagger M x$
    $\rho \leftarrow \langle r, r \rangle$
    $k \leftarrow 0$
    while $\rho > \epsilon$ *or* $k < n$ **do**
        $z \leftarrow M p$
        $q \leftarrow M^\dagger z$
        $\alpha \leftarrow \rho / \langle z, z \rangle$
        $r \leftarrow r - \alpha q$
        $x \leftarrow x + \alpha p$
        $\gamma \leftarrow \langle r, r \rangle$
        if $\gamma < \epsilon$ **then**
            $\rho \leftarrow \gamma$
            break
        end
        $\beta \leftarrow \gamma / \rho$
        $\rho \leftarrow \gamma$
        $p \leftarrow r + \beta p$
        $k \leftarrow k + 1$
    end
    return x, ρ, k.
end

Algorithm 2: DWF-ready Gradient Solver.

4 PRECONDITIONING

We use four dimensional preconditioner to improve convergence of the CG. Following Kostas Orginos, let us color the lattice sites according to the parity of $x_0 + x_1 + x + 2 + x_3$. Then we can rewrite $\chi = D\psi$ as follows:

$$\left(\begin{array}{c} \chi_e \\ \chi_o \end{array} \right) = D\psi = \left(\begin{array}{cc} Q_{ee} & Q_{eo} \\ Q_{oe} & Q_{oo} \end{array} \right) \left(\begin{array}{c} \psi_e \\ \psi_o \end{array} \right)$$

From the form of D it follows that all dependance on the gauge field is located in Q_{xy}, and that Q_{xx} does not depend on U. That, in turn, allows us to invert Q_{xx} easily. With this in mind, one writes:

$$\left(\begin{array}{cc} Q_{ee} & Q_{eo} \\ Q_{oe} & Q_{oo} \end{array} \right) = \left(\begin{array}{cc} Q_{ee} & 0 \\ Q_{oe} & Q_{oo} \end{array} \right) \left(\begin{array}{cc} 1 & Q_{ee}^{-1}Q_{eo} \\ 0 & 1 - Q_{oo}^{-1}Q_{oe}Q_{ee}^{-1}Q_{eo} \end{array} \right)$$

Now, to solve the equation

$$D\psi = \eta,$$

one needs to perform the following steps:

1. Compute
$$\phi_o = Q_{oo}^{-1}(\eta_o - Q_{oe}Q_{ee}^{-1}\eta_e)$$

2. Set $M = 1 - Q_{oo}^{-1}Q_{oe}Q_{ee}^{-1}Q_{eo}$ for the following.

3. Compute
$$\varphi_o = M^\dagger \phi_o$$

4. Solve for ψ_o the following equation using Algorithm 2
$$M^\dagger M \psi_o = \varphi_o$$

5. Compute
$$\psi_e = Q_{ee}^{-1}(\eta_e - Q_{eo}\psi_o)$$

Note, that $M^\dagger = 1 - (Q_{eo})^\dagger (Q_{ee}^{-1})^\dagger (Q_{oe})^\dagger (Q_{oo}^{-1})^\dagger = 1 - S_{oe}S_{ee}^{-1}S_{oe}S_{oo}^{-1}$, where

$$\begin{array}{rcl} S_{ee} &=& Q_{ee}[\gamma_5 \to -\gamma_5] \\ S_{oo} &=& Q_{oo}[\gamma_5 \to -\gamma_5] \\ S_{oe} &=& Q_{eo}[\gamma_\mu \to -\gamma_\mu] \\ S_{eo} &=& Q_{oe}[\gamma_\mu \to -\gamma_\mu] \end{array}$$

4.1 Q_{xx} inversion

The previous section is based on a tacit assumption that Q_{ee} and Q_{oo} are easy to invert. Here we show that it is so. Let us rewrite

$$\chi_{s,x} = (Q_{ee}\psi)_{s,x} = M_0 \psi_{s,x} + (1+\gamma_5)M_s^{(+)}\psi_{s+1,x} + (1+\gamma_5)M_s^{(-)}\psi_{s-1,x}$$

as follows:

$$(Q_{ee}\psi)_{s,x} = M_0 \left(\left(\frac{1+\gamma_5}{2} \right) \left(\psi_{s,x} + \frac{2M_s^{(+)}}{M_0}\psi_{s+1,x} \right) + \left(\frac{1-\gamma_5}{2} \right) \left(\psi_{s,x} + \frac{2M_s^{(-)}}{M_0}\psi_{s-1,x} \right) \right).$$

Thus,

$$Q_{ee} = \frac{1+\gamma_5}{2} \left(\begin{array}{ccccc} a & b & \cdots & 0 & 0 \\ 0 & a & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & a & b \\ c & 0 & \cdots & 0 & a \end{array} \right) + \frac{1-\gamma_5}{2} \left(\begin{array}{ccccc} a & 0 & \cdots & 0 & c \\ b & a & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & a & 0 \\ 0 & 0 & \cdots & b & a \end{array} \right) = P_+A + P_-B,$$

where $a = M_0$, $b = 2$, $c = -2m_f$. Computing $Q_{xx}\psi$ and $S_{xx}\psi$ is done below with the vector hardware. Here we compute constant values needed for effective use of vectors:

13a $\langle$*Compute constant values for Q_{xx} and S_{xx}* 13a$\rangle\equiv$

```
{
    double a = M_0;
    double b = 2.0;
    double c = -2 * m_f;

    va   = vmk_1(a);
    vb   = vmk_1(b);
    vcbn = vmk_1n(c, b);  /* {c, b, ...} */
    vbnc = vmk_n1(b, c);  /* {b, ..., c} */
}
```

This code is used in chunks 19c and 20a.

13b $\langle$*Global variables* 13b$\rangle\equiv$

```
static vReal vcbn;
static vReal vbnc;
static vReal vb;
static vReal va;
```

This definition is continued in chunks 14, 15, 19b, 25b, 27a, 31, 45–47, 53g, and 55a.
This code is used in chunk 85a.

Now, since $P_\pm$ comute with A and B, $Q_{ee}^{-1} = P_+ A^{-1} + P_- B^{-1}$. Computing A^{-1} and B^{-1} is done by decomposition $A = L_A R_A$, $B = L_B R_B$, where

$$R_A = \begin{pmatrix} a & b & \cdots & 0 & 0 \\ 0 & a & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & a & b \\ 0 & 0 & \cdots & 0 & a \end{pmatrix} \qquad R_B = \begin{pmatrix} a & 0 & \cdots & 0 & 0 \\ b & a & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & a & 0 \\ 0 & 0 & \cdots & b & a \end{pmatrix},$$

and

$$L_A = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & & 0 \\ \vdots & & & & & \vdots \\ c/a & -bc/a^2 & b^2c/a^3 & -b^3c/a^4 & \cdots & 1+(-b)^{n-1}c/a^n \end{pmatrix}$$

$$L_B = \begin{pmatrix} 1+(-b)^{n-1}c/a^n & (-b)^{n-2}c/a^{n-1} & \cdots & b^2c/a^3 & -bc/a^2 & c/a \\ 0 & 1 & & 0 & 0 & 0 \\ \vdots & & & & & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}.$$

In these terms,

$$Q_{ee}^{-1} = \frac{1+\gamma_5}{2} R_A^{-1} L_A^{-1} + \frac{1-\gamma_5}{2} R_B^{-1} L_B^{-1}.$$

We will also need

$$S_{ee}^{-1} = \frac{1-\gamma_5}{2} R_A^{-1} L_A^{-1} + \frac{1+\gamma_5}{2} R_B^{-1} L_B^{-1}.$$

For further reference,

$$\gamma_5 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

5 CODE

This section contains chunks of the source that go into `dwf.c` source file. We start with the interface functions and elaborate from there.

5.1 Interface Functions

We can not expect the user to call different parts of the interface in an appropriate order. Therefore, successful initialization allows the user to call other interface elements, as well as prevents repeated initializations.

14a ⟨*Global variables* 13b⟩+≡
```
static int inited_p = 0;
```

5.1.1 DWF Initializer

14b ⟨*Interface functions* 14b⟩≡
```
int
L3(DWF_init)(const int lattice[DIM+1],
             void *(*allocator)(size_t size),
             void (*deallocator)(void *))
{
    ⟨Version 1⟩

    if (inited_p)
        return 1; /* error: second init */

    ⟨Check lattice size 15d⟩
    ⟨Get network topology 26b⟩
    ⟨Setup heap management functions 15a⟩
    ⟨Initialize tables 30c⟩
    ⟨Allocate fields 45d⟩
    ⟨Initialize QMP 40e⟩
    ⟨Show DWF version 81c⟩
    inited_p = 1;
    DEBUG_DWF("finished init, lattice=[%d %d %d %d %d]\n",
              lattice[0], lattice[1], lattice[2], lattice[3], lattice[4])
    return 0;

    ⟨Handle init errors 14c⟩
}
```
This definition is continued in chunks 15–20 and 22.
This code is used in chunk 85a.

If any error occurs during initialization, we simply unroll state and fail:

14c ⟨*Handle init errors* 14c⟩≡
```
error:
    L3(DWF_fini)();
    return 1;
```
This code is used in chunk 14b.

Check if the user requested special allocation mechanisms:

15a ⟨*Setup heap management functions* 15a⟩≡
```
if (allocator)
    tmalloc = allocator;
else
    tmalloc = malloc;

if (deallocator)
    tfree = deallocator;
else
    tfree = free;
```

15b ⟨*Global variables* 13b⟩+≡
```
static void *(*tmalloc)(size_t size);
static void (*tfree)(void *ptr);
```

15c ⟨*Include files* 15c⟩≡
```
#include <string.h>
#include <stdlib.h>
```

For simplicity of s-slice operations, L_s should be a muplitple of the vector size.

15d ⟨*Check lattice size* 15d⟩≡
```
if (lattice[DIM] % Vs)
    goto error;
tlattice[DIM] = lattice[DIM];
```

Otherwise, lattice sizes must be even to allow us to do red/black preconditioning:

15e ⟨*Check lattice size* 15d⟩+≡
```
{
    int i;
    for (i = 0; i < DIM; i++) {
        if (lattice[i] & 1)
            goto error;
        tlattice[i] = lattice[i];
    }
}
```

15f ⟨*Global variables* 13b⟩+≡
```
static int tlattice[DIM+1];
```

5.1.2 DWF Clean Up

The cleanup routine may be called from partially initialized context, we should be able to do a partial cleanup.

15g ⟨*Interface functions* 14b⟩+≡
```
void
L3(DWF_fini)(void)
{
    ⟨Cleanup QMP 44b⟩
    ⟨Free fields 45e⟩
    ⟨Free tables 39a⟩
    inited_p = 0;
    DEBUG_DWF("fini done\n")
}
```

5.1.3 DWF Fermion Allocator

When one needs a DWF fermion, the allocator does the job. Remember, users are stupid enough to call this function in the uninitialized state. It is convenient to break all internal fermions into odd and even parts at this stage.

16a ⟨*Interface functions* 14b⟩+≡

```
L3(DWF_Fermion) *
L3(DWF_allocate_fermion)(void)
{
    L3(DWF_Fermion) *ptr;

    if (!inited_p)
        return 0;

    ptr = tmalloc(sizeof (*ptr));
    if (ptr == 0)
        return 0;

    ptr->even = allocate_even_fermion();
    if (ptr->even == 0)
        goto error1;

    ptr->odd  = allocate_odd_fermion();
    if (ptr->odd == 0)
        goto error2;

    return ptr;
  error2:
    free16(ptr->even);
  error1:
    tfree(ptr);
    return 0;
}
```

5.1.4 DWF Fermion Exporter

When we need to create a DWF field and populate it from an outer environment, we use the following procedure

16b ⟨*Interface functions* 14b⟩+≡

```
L3(DWF_Fermion) *
L3(DWF_load_fermion)(const void *OuterFermion,
                     void *env,
                     L3(DWF_fermion_reader) reader)
{
    L3(DWF_Fermion) *ptr = L3(DWF_allocate_fermion)();

    DEBUG_DWF("OuterFermion=%p, reader=%p\n", OuterFermion, reader)

    /* Handle both lack of memory and missing initialization */
    if (ptr == 0)
        return 0;

    ⟨Read fermion 28d⟩

    return ptr;
}
```

5.1.5 DWF Fermion Importer

For moving data back to the outer environment, the following importer is used:

17a   ⟨*Interface functions* 14b⟩+≡

```
void
L3(DWF_save_fermion)(void *OuterFermion,
                     void *env,
                     L3(DWF_fermion_writer) writer,
                     L3(DWF_Fermion) *CGfermion)
{
    if (!inited_p)
        return;

    DEBUG_DWF("Outer fermion=%p, writer=%p\n", OuterFermion, writer)

    ⟨Write fermion 29c⟩
}
```

5.1.6 DWF Fermion Deallocator

We free only pointers that we allocated. The magic is in `free16()`—it knows about all heap objects allocated by `alloc16()`.

17b   ⟨*Interface functions* 14b⟩+≡

```
void
L3(DWF_delete_fermion)(L3(DWF_Fermion) *ptr)
{
    if (!inited_p)
        return;

    free16(ptr->even);
    free16(ptr->odd);
    tfree(ptr);
}
```

5.1.7 DWF Gauge Exporter

Unlike fermions, gauge fields are 4-d in the solver. Though they are not loaded by vector memory operations, we still allocate 16-byte aligned memory for them (apparently for no good reason at all.)

18a ⟨*Interface functions* 14b⟩+≡

```
L3(DWF_Gauge) *
L3(DWF_load_gauge)(const void *OuterGauge_U,
                   const void *OuterGauge_V,
                   void *env,
                   L3(DWF_gauge_reader) reader)
{
    L3(DWF_Gauge) *g;

    if (!inited_p)
        return 0;

    DEBUG_DWF("U=%p, V=%p, reader=%p\n", OuterGauge_U, OuterGauge_V, reader)

    g = allocate_gauge_field();
    if (g == 0)
        return 0;

    ⟨Read gauge field 27b⟩
    return g;
}
```

Let us also define `L3(DWF_Gauge)` here. We do not need anything fancy for the gauge field:

18b ⟨*Data types* 18b⟩≡

```
struct L3(DWF_Gauge) {
    scalar_complex v[Nc][Nc];
};
```

This definition is continued in chunks 23–25 and 31d.
This code is used in chunk 85a.

5.1.8 DWF Gauge Deallocator

Gauge deallocator is very much like fermion deallocator. We only keep them separate to help the type system cope with a error making user.

18c ⟨*Interface functions* 14b⟩+≡

```
void
L3(DWF_delete_gauge)(L3(DWF_Gauge) *ptr)
{
    if (!inited_p)
        return;

    free16(ptr);
}
```

5.1.9 The Solver

Finally, the solver itself. Here we check if the system has been properly initialized and dispatch on the float size (but not now yet.)

19a   ⟨*Interface functions* 14b⟩+≡
```
int
L3(DWF_cg_solver)(L3(DWF_Fermion)        *psi,
                  double                 *out_eps,
                  int                    *out_iter,
                  const L3(DWF_Gauge)    *gauge,
                  double                  M,
                  double                  m_f,
                  const L3(DWF_Fermion)  *x0,
                  const L3(DWF_Fermion)  *eta,
                  double                  eps,
                  int                     min_iter,
                  int                     max_iter)
{
    int status;

    if (!inited_p)
        return 1;

    U = (SU3 *)gauge;
```
⟨*Compute constant values for Q_{xx}^{-1} and S_{xx}^{-1}* 80g⟩
⟨*Compute φ_o* 45b⟩
⟨*Solve $M^\dagger M \psi_o = \varphi_o$* 45f⟩
⟨*Compute ψ_e* 47c⟩
```
    return status;
}
```

Save one argument in many functions:

19b   ⟨*Global variables* 13b⟩+≡
```
static SU3 *U;
```

5.1.10 Dirac Operator

It is convenient to have the Dirac operator and its conjugate as separate functions.

$$\chi \leftarrow D_{DW}\psi.$$

19c   ⟨*Interface functions* 14b⟩+≡
```
void
L3(DWF_Dirac_Operator)(L3(DWF_Fermion)        *chi,
                       const L3(DWF_Gauge)    *gauge,
                       double                  M_0,
                       double                  m_f,
                       const L3(DWF_Fermion)  *psi)
{
    if (!inited_p)
        return;

    U = (SU3 *)gauge;
```
⟨*Compute constant values for Q_{xx} and S_{xx}* 13a⟩
```
    compute_Do(chi->odd,  psi->odd,  psi->even);
    compute_De(chi->even, psi->even, psi->odd);
}
```

19

$$\chi \leftarrow D_{DW}^\dagger \psi.$$

20a $\langle$*Interface functions* 14b$\rangle+\equiv$

```
void
L3(DWF_Dirac_Operator_conjugate)(L3(DWF_Fermion)       *chi,
                                 const L3(DWF_Gauge)    *gauge,
                                 double                  M_0,
                                 double                  m_f,
                                 const L3(DWF_Fermion)  *psi)
{
    if (!inited_p)
        return;

    U = (SU3 *)gauge;
```
    $\langle$*Compute constant values for Q_{xx} and S_{xx}* 13a$\rangle$
```
    compute_Dco(chi->odd, psi->odd, psi->even);
    compute_Dce(chi->even, psi->even, psi->odd);
}
```

5.1.11   Little Helpers

$$\psi \leftarrow \varphi + a\eta$$

20b $\langle$*Interface functions* 14b$\rangle+\equiv$

```
void
L3(DWF_Add_Fermion)(L3(DWF_Fermion)        *psi,
                    const L3(DWF_Fermion)   *phi,
                    double                   a,
                    const L3(DWF_Fermion)   *eta)
{
    collect_add(&psi->odd->f,  &phi->odd->f,  a, &eta->odd->f,  odd_even.size * Sv);
    collect_add(&psi->even->f, &phi->even->f, a, &eta->even->f, even_odd.size * Sv);
}
```

⟨*Static function prototypes* 21⟩≡

```
static inline void
collect_add(vFermion *r,
            const vFermion *x,
            double A,
            const vFermion *y,
            int n)
{
  int i;
  vReal a = vmk_1(A);

  for (i = 0; i < n; i++, r++, x++, y++) {
      r->f[0][0].re = x->f[0][0].re + a * y->f[0][0].re;
      r->f[0][0].im = x->f[0][0].im + a * y->f[0][0].im;
      r->f[0][1].re = x->f[0][1].re + a * y->f[0][1].re;
      r->f[0][1].im = x->f[0][1].im + a * y->f[0][1].im;
      r->f[0][2].re = x->f[0][2].re + a * y->f[0][2].re;
      r->f[0][2].im = x->f[0][2].im + a * y->f[0][2].im;

      r->f[1][0].re = x->f[1][0].re + a * y->f[1][0].re;
      r->f[1][0].im = x->f[1][0].im + a * y->f[1][0].im;
      r->f[1][1].re = x->f[1][1].re + a * y->f[1][1].re;
      r->f[1][1].im = x->f[1][1].im + a * y->f[1][1].im;
      r->f[1][2].re = x->f[1][2].re + a * y->f[1][2].re;
      r->f[1][2].im = x->f[1][2].im + a * y->f[1][2].im;

      r->f[2][0].re = x->f[2][0].re + a * y->f[2][0].re;
      r->f[2][0].im = x->f[2][0].im + a * y->f[2][0].im;
      r->f[2][1].re = x->f[2][1].re + a * y->f[2][1].re;
      r->f[2][1].im = x->f[2][1].im + a * y->f[2][1].im;
      r->f[2][2].re = x->f[2][2].re + a * y->f[2][2].re;
      r->f[2][2].im = x->f[2][2].im + a * y->f[2][2].im;

      r->f[3][0].re = x->f[3][0].re + a * y->f[3][0].re;
      r->f[3][0].im = x->f[3][0].im + a * y->f[3][0].im;
      r->f[3][1].re = x->f[3][1].re + a * y->f[3][1].re;
      r->f[3][1].im = x->f[3][1].im + a * y->f[3][1].im;
      r->f[3][2].re = x->f[3][2].re + a * y->f[3][2].re;
      r->f[3][2].im = x->f[3][2].im + a * y->f[3][2].im;
  }
}
```

This definition is continued in chunks 23a, 25d, 29b, 30b, 32, 35a, 36f, 38–44, 46–52, 61, 62a, 65, 66, 79c, 82, and 83d.
This code is used in chunk 85a.

$$r \leftarrow \langle \psi | \phi \rangle$$

22   ⟨*Interface functions* 14b⟩+≡

```
void
L3(DWF_Fermion_Dot_Product)(double                *r_re,
                            double                *r_im,
                            const L3(DWF_Fermion)  *psi,
                            const L3(DWF_Fermion)  *phi)
{
    *r_re = *r_im = 0;
    collect_dot(r_re, r_im, &psi->odd->f,  &phi->odd->f,  odd_even.size * Sv);
    collect_dot(r_re, r_im, &psi->even->f, &phi->even->f, even_odd.size * Sv);
    DEBUG_QMP("before sum re: %g\n", *r_re)
    QMP_sum_double(r_re);
    DEBUG_QMP("after sum re: %g\n", *r_re)
    DEBUG_QMP("before sum im: %g\n", *r_im)
    QMP_sum_double(r_im);
    DEBUG_QMP("after sum im: %g\n", *r_im)
}
```

Running the dot product on one parity of fermions.

23a ⟨*Static function prototypes* 21⟩+≡

```
static inline void
collect_dot(double *r_re,
            double *r_im,
            const vFermion *a,
            const vFermion *b,
            int n)
{
  int i;
  vReal c0_re, c1_re, c2_re;
  vReal c0_im, c1_im, c2_im;

  for (i = 0; i < n; i++, a++, b++) {
    c0_re = a->f[0][0].re*b->f[0][0].re; c0_re += a->f[0][0].im*b->f[0][0].im;
    c0_im = a->f[0][0].re*b->f[0][0].im; c0_im -= a->f[0][0].im*b->f[0][0].re;
    c1_re = a->f[0][1].re*b->f[0][1].re; c1_re += a->f[0][1].im*b->f[0][1].im;
    c1_im = a->f[0][1].re*b->f[0][1].im; c1_im -= a->f[0][1].im*b->f[0][1].re;
    c2_re = a->f[0][2].re*b->f[0][2].re; c2_re += a->f[0][2].im*b->f[0][2].im;
    c2_im = a->f[0][2].re*b->f[0][2].im; c2_im -= a->f[0][2].im*b->f[0][2].re;

    c0_re += a->f[1][0].re*b->f[1][0].re; c0_re += a->f[1][0].im*b->f[1][0].im;
    c0_im += a->f[1][0].re*b->f[1][0].im; c0_im -= a->f[1][0].im*b->f[1][0].re;
    c1_re += a->f[1][1].re*b->f[1][1].re; c1_re += a->f[1][1].im*b->f[1][1].im;
    c1_im += a->f[1][1].re*b->f[1][1].im; c1_im -= a->f[1][1].im*b->f[1][1].re;
    c2_re += a->f[1][2].re*b->f[1][2].re; c2_re += a->f[1][2].im*b->f[1][2].im;
    c2_im += a->f[1][2].re*b->f[1][2].im; c2_im -= a->f[1][2].im*b->f[1][2].re;

    c0_re += a->f[2][0].re*b->f[2][0].re; c0_re += a->f[2][0].im*b->f[2][0].im;
    c0_im += a->f[2][0].re*b->f[2][0].im; c0_im -= a->f[2][0].im*b->f[2][0].re;
    c1_re += a->f[2][1].re*b->f[2][1].re; c1_re += a->f[2][1].im*b->f[2][1].im;
    c1_im += a->f[2][1].re*b->f[2][1].im; c1_im -= a->f[2][1].im*b->f[2][1].re;
    c2_re += a->f[2][2].re*b->f[2][2].re; c2_re += a->f[2][2].im*b->f[2][2].im;
    c2_im += a->f[2][2].re*b->f[2][2].im; c2_im -= a->f[2][2].im*b->f[2][2].re;

    c0_re += a->f[3][0].re*b->f[3][0].re; c0_re += a->f[3][0].im*b->f[3][0].im;
    c0_im += a->f[3][0].re*b->f[3][0].im; c0_im -= a->f[3][0].im*b->f[3][0].re;
    c1_re += a->f[3][1].re*b->f[3][1].re; c1_re += a->f[3][1].im*b->f[3][1].im;
    c1_im += a->f[3][1].re*b->f[3][1].im; c1_im -= a->f[3][1].im*b->f[3][1].re;
    c2_re += a->f[3][2].re*b->f[3][2].re; c2_re += a->f[3][2].im*b->f[3][2].im;
    c2_im += a->f[3][2].re*b->f[3][2].im; c2_im -= a->f[3][2].im*b->f[3][2].re;

    *r_re += vsum(c0_re + c1_re + c2_re);
    *r_im += vsum(c0_im + c1_im + c2_im);
  }
}
```

5.2 Internal Data Types

Given `scalar_complex` and `vector_complex` types, define QCD related data types. A vector of full Dirac fermions:

23b ⟨*Data types* 18b⟩+≡

```
typedef struct {
    vector_complex f[Fd][Nc];
} vFermion;
```

The projected fermion vector. We do not distinguish different projections here:

24a   ⟨*Data types* 18b⟩+≡
```
typedef struct {
    vector_complex f[Fd/2][Nc];
} vHalfFermion;
```

Gauge field after conversion into DWF form:

24b   ⟨*Data types* 18b⟩+≡
```
typedef struct {
    scalar_complex v[Nc][Nc];
} SU3;
```

Small piece of the gauge field converted to a vector form:

24c   ⟨*Data types* 18b⟩+≡
```
typedef struct {
    vector_complex v[Nc][Nc];
} vSU3;
```

Even and odd sublattices of fermions:

24d   ⟨*Data types* 18b⟩+≡
```
typedef struct {
    vFermion f;
} vEvenFermion;

typedef struct {
    vFermion f;
} vOddFermion;
```

The interface fermion is a pair of even and odd sublattices:

24e   ⟨*Data types* 18b⟩+≡
```
struct L3(DWF_Fermion) {
    vEvenFermion *even;
    vOddFermion *odd;
};
```

5.3 Memory Allocation

Vector hardware does like properly aligned memory. While automatic variables are aligned by the compiler, extra care is needed when dealing with the heap. The code allocates all its own memory aligned on 16-byte boundary by calling `alloc16()`, and returns the memory through `free16()`.

24f   ⟨*Static functions* 24f⟩≡
```
static void *
alloc16(int size)
{
    int xsize = PAD16(size + sizeof (struct memblock));
    struct memblock *p = tmalloc(xsize);

    if (p == 0)
        return p;

    p->data = ALIGN16(&p[1]);
    p->size = size;
    p->next = memblock.next;
    p->prev = &memblock;
    p->next->prev = p;
    p->prev->next = p;

    return p->data;
}
```
This definition is continued in chunks 25a, 26a, 31a, 32c, 34d, 37a, 38b, 41–44, 46–52, 62–65, and 84.
This code is used in chunk 85a.

For deallocation we need to find an appropriate memory block:

25a   ⟨*Static functions* 24f⟩+≡

```
static void
free16(void *ptr)
{
    struct memblock *p;

    if (ptr == 0)
        return;

    for (p = memblock.next; p != &memblock; p = p->next) {
        if (p->data != ptr)
            continue;
        p->next->prev = p->prev;
        p->prev->next = p->next;
        tfree(p);
        return;
    }
    /* this is BAD: control should not reach here! */
}
```

The head of the memory list is stored in a static variable. Of course, such an implementation is not threadable, but let us worry about that when the time is right.

25b   ⟨*Global variables* 13b⟩+≡

```
static struct memblock memblock = {
    &memblock,
    &memblock,
    NULL,
    0
};
```

Finally, the datatype for the linked list:

25c   ⟨*Data types* 18b⟩+≡

```
struct memblock {
    struct memblock *next;
    struct memblock *prev;
    void *data;
    size_t size;
};
```

5.3.1   Field allocators

First, the prototypes:

25d   ⟨*Static function prototypes* 21⟩+≡

```
static vEvenFermion *allocate_even_fermion(void);
static vOddFermion *allocate_odd_fermion(void);
static L3(DWF_Gauge) *allocate_gauge_field(void);
```

The only difference between even and odd fermions is (possibly) their size:

26a   ⟨*Static functions* 24f⟩+≡

```
vEvenFermion *
allocate_even_fermion(void)
{
    return alloc16(even_odd.size * Sv * sizeof (vFermion));
}

vOddFermion *
allocate_odd_fermion(void)
{
    return alloc16(odd_even.size * Sv * sizeof (vFermion));
}

L3(DWF_Gauge) *
allocate_gauge_field(void)
{
    return alloc16(gauge_XYZT * sizeof (L3(DWF_Gauge)));
}
```

5.4   Probing Cluster Topology

There is no proper way to query QMP about lattice layout. We have to request the minimal meaningful information the library provides and try to repeat outer layer's partitioning of the lattice. There are good chances of success, but this is a potential danger spot.

Here we prepare compute where on the lattice this node is and to build up our understanding of neighbors. Maybe optimistically, we assume that once QMP is initialized, it reports logical dimensions and coordinates properly, so that we do not need to be paranoidal about errors here.

26b   ⟨*Get network topology* 26b⟩≡

```
{
    int i, dn;
    const int *xn, *xc;

    if (!QMP_logical_topology_is_declared())
        /* The user must have declared logical topology before */
        goto error;
    dn = QMP_get_logical_number_of_dimensions();
    if (dn > DIM)
        /* Too high dimension of the logical network */
        goto error;

    xn = QMP_get_logical_dimensions();
    xc = QMP_get_logical_coordinates();
    for (i = 0; i < dn; i++) {
        network[i] = xn[i];
        coord[i] = xc[i];
    }

    for (; i < DIM; i++) {
        network[i] = 1;
        coord[i] = 0;
    }
}
```

This code is used in chunk 14b.

Some global variables:

27a   ⟨*Global variables* 13b⟩+≡
```
static int network[DIM];
static int coord[DIM];
```

5.5 Moving Data

5.5.1 Reading the Gauge Field

Let us start with reading of the gauge field from the outer environment first. Here we assume that there is an address translation function to help us in talking to the outer layer.

27b   ⟨*Read gauge field* 27b⟩≡
```
{
    int x[DIM], i, d, a, b, p1;

    ⟨Start DIM-d sublattice scan 27d⟩
        ⟨Load DIM gauge links from U  at x 27c⟩
    ⟨Advance DIM-d index for a sublattice scan 27e⟩

    for (d = 0; d < DIM; d++)
        ⟨Load gauge boundary in direction d 28b⟩
}
```
This code is used in chunk 18a.

At a given site, load `DIM` gauge elements:

27c   ⟨*Load* DIM *gauge links from U at* x 27c⟩≡
```
p1 = to_Ulinear(x, &bounds, -1);
for (d = 0; d < DIM; d++) {
    for (a = 0; a < Nc; a++) {
        for (b = 0; b < Nc; b++) {
            g[p1 + d].v[a][b].re = reader(OuterGauge_U, env, x, d, a, b, 0);
            g[p1 + d].v[a][b].im = reader(OuterGauge_U, env, x, d, a, b, 1);
        }
    }
}
```
This code is used in chunk 27b.

To start a scan over the lattice, initialize `x` and start the loop:

27d   ⟨*Start* DIM-*d sublattice scan* 27d⟩≡
```
for (i = 0; i < DIM; i++)
    x[i] = bounds.lo[i];
for (i = 0; i < DIM;) {
```
This code is used in chunks 27–29 and 35c.

Once all is done with the site `x`, we are ready to advance the index:

27e   ⟨*Advance* DIM-*d index for a sublattice scan* 27e⟩≡
```
    for (i = 0; i < DIM; i++) {
        ⟨Advance x at i 28a⟩
    }
}
```
This code is used in chunks 27–29 and 35c.

Since we are going to use a `DIM-1` dimensional scan as well, let us write it down here:

27f   ⟨*Advance* DIM-*d index for* DIM-1-*d scan* 27f⟩≡
```
    for (i = 0; i < DIM; i++) {
        if (i == d)
            continue;
        ⟨Advance x at i 28a⟩
    }
}
```
This code is used in chunk 28b.

Now we can scan `DIM`-dimensional indices:

28a ⟨*Advance* `x` *at* `i` 28a⟩≡
```
if (++x[i] == bounds.hi[i])
    x[i] = bounds.lo[i];
else
    break;
```
This code is used in chunk 27.

DWF Dirac operator needs backward gauge links. We get them from `OuterGauge_V`. Here we only read the boundary links.

28b ⟨*Load gauge boundary in direction* `d` 28b⟩≡
```
{
    if (network[d] == 1)
        continue;

    ⟨Start DIM-d sublattice scan 27d⟩
        ⟨Load a d gauge link from V  at x 28c⟩
    ⟨Advance DIM-d index for DIM-1-d scan 27f⟩
}
```
This code is used in chunk 27b.

Now we read a boundary element:

28c ⟨*Load a* `d` *gauge link from V at* `x` 28c⟩≡
```
p1 = to_Ulinear(x, &bounds, d);
for (a = 0; a < Nc; a++) {
    for (b = 0; b < Nc; b++) {
        g[p1].v[a][b].re = reader(OuterGauge_V, env, x, d, a, b, 0);
        g[p1].v[a][b].im = reader(OuterGauge_V, env, x, d, a, b, 1);
    }
}
```
This code is used in chunk 28b.

5.5.2   Reading a Fermion

There are but two complications in reading the domain wall fermion. First, this is a good time to break the fermion into red and black pieces. In addition, here we construct DWF fermions.

28d ⟨*Read fermion* 28d⟩≡
```
{
    int x[DIM+1], i;

    ⟨Start DIM-d sublattice scan 27d⟩
        ⟨Load an s-line of fermion at x 29a⟩
    ⟨Advance DIM-d index for a sublattice scan 27e⟩
}
```
This code is used in chunk 16b.

Data conversion is inherently inefficient. We do not try to optimize it here:

29a   ⟨*Load an s-line of fermion at* x 29a⟩≡
```
{
    int p = parity(x);
    int p1 = Sv * to_HFlinear(x, &bounds, -1, 0); /* p is taken care of! */
    vFermion *f = p? &ptr->odd[p1].f: &ptr->even[p1].f;

    for (x[DIM] = 0; x[DIM] < tlattice[DIM]; x[DIM] += Vs, f++) {
        int d;
        for (d = 0; d < Fd; d++) {
            int c;
            for (c = 0; c < Nc; c++) {
                f->f[d][c].re = import_vector(OuterFermion, env, reader,
                                              x, c, d, 0);
                f->f[d][c].im = import_vector(OuterFermion, env, reader,
                                              x, c, d, 1);
            }
        }
    }
}
```
This code is used in chunk 28d.

A simple packer of Vs elements into a vector:

29b   ⟨*Static function prototypes* 21⟩+≡
```
static inline vReal
import_vector(const void *z, void *env, L3(DWF_fermion_reader) reader,
              int x[DIM+1], int c, int d, int re_im)
{
    vReal f;
    REAL *v = (REAL *)&f;
    int i, xs;

    for (xs = x[DIM], i = 0; i < Vs; i++, x[DIM]++) {
        *v++ = reader(z, env, x, c, d, re_im);
    }
    x[DIM] = xs;
    return f;
}
```

5.5.3   Writing a Fermion

Writing a fermion is not much different:

29c   ⟨*Write fermion* 29c⟩≡
```
{
    int x[DIM+1], i;

    ⟨Start DIM-d sublattice scan 27d⟩
        ⟨Save an s-line of fermion at x 30a⟩
    ⟨Advance DIM-d index for a sublattice scan 27e⟩
}
```
This code is used in chunk 17a.

⟨*Save an s-line of fermion at* x 30a⟩≡

```
{
    int p = parity(x);
    int p1 = Sv * to_HFlinear(x, &bounds, -1, 0); /* p is taken care of! */
    vFermion *f = p? &CGfermion->odd[p1].f: &CGfermion->even[p1].f;

    for (x[DIM] = 0; x[DIM] < tlattice[DIM]; x[DIM] += Vs, f++) {
        int d;
        for (d = 0; d < Fd; d++) {
            int c;
            for (c = 0; c < Nc; c++) {
                save_vector(OuterFermion, env, writer, x, c, d, 0,
                            &f->f[d][c].re);
                save_vector(OuterFermion, env, writer, x, c, d, 1,
                            &f->f[d][c].im);
            }
        }
    }
}
```

This code is used in chunk 29c.

Here's another little helper good only for writing back the fermion from DWF to the outer environement:

⟨*Static function prototypes* 21⟩+≡

```
static inline void
save_vector(void *z, void *env, L3(DWF_fermion_writer) writer,
            int x[DIM+1], int c, int d, int re_im, vReal *f)
{
    REAL *v = (REAL *)f;
    int i, xs;

    for (xs = x[DIM], i = 0; i < Vs; i++, x[DIM]++) {
        writer(z, env, x, c, d, re_im, *v++);
    }
    x[DIM] = xs;
}
```

5.6 Solver Initialization

Here are all pieces for setting up the structures needed to run the solver.

5.6.1 Constructing the neighbor tables

⟨*Initialize tables* 30c⟩≡

```
if (init_tables()) {
    /* Something went wrong in the table construction */
    goto error;
}
```

This code is used in chunk 14b.

The table initializer creates all tables necessary for communication and computation. Memory is allocated here for index arrays.

31a   ⟨*Static functions* 24f⟩+≡
```
static int
init_tables(void)
{
    struct neighbor tmp;
    int i, v;

    init_neighbor(&bounds, &neighbor);
    ⟨Compute init sizes 31c⟩
    tmp = neighbor;
    build_neighbor(&even_odd, 0, &tmp);
    build_neighbor(&odd_even, 1, &tmp);

    return 0;
}
```

First, we set global data:

31b   ⟨*Global variables* 13b⟩+≡
```
static struct bounds bounds;
static int gauge_XYZT;
static int Sv, Sv_1;
```

31c   ⟨*Compute init sizes* 31c⟩≡
```
Sv = tlattice[DIM] / Vs;
Sv_1 = Sv - 1;
for (v = 1, i = 0; i < DIM; i++) {
    v *= bounds.hi[i] - bounds.lo[i];
}
gauge_XYZT = DIM * v;
for (i = 0; i < DIM; i++) {
    if (network[i] < 2)
        continue;
    gauge_XYZT += v / (bounds.hi[i] - bounds.lo[i]);
}
```
This code is used in chunk 31a.

The `struct bounds` helps us to navigate through the local part of the lattice. It is used by the initialization code only.

31d   ⟨*Data types* 18b⟩+≡
```
struct bounds {
    int lo[DIM];
    int hi[DIM];
};
```

We keep two `struct neighbor`, one for computation on the even sublattice, another—on the odd. In addition to `even_odd` and `odd_even`, we need one more `struct neighbor` to keep the allocated pointers in.

31e   ⟨*Global variables* 13b⟩+≡
```
static struct neighbor neighbor;
#ifdef NO_DEBUG_QMP
static struct neighbor odd_even;
static struct neighbor even_odd;
#else
struct neighbor odd_even;
struct neighbor even_odd;
#endif
```

Let us start with computing the boundary of the sublattice

32a   ⟨*Static function prototypes* 21⟩+≡

```
static inline int
lattice_start(int lat, int net, int coord)
{
    int q = lat / net;
    int r = lat % net;

    return coord * q + ((coord < r)? coord: r);
}

static inline void
mk_sublattice(struct bounds *bounds,
              int coord[])
{
    int i;

    for (i = 0; i < DIM; i++) {
        bounds->lo[i] = lattice_start(tlattice[i], network[i], coord[i]);
        bounds->hi[i] = lattice_start(tlattice[i], network[i], coord[i] + 1);
    }
}
```

All dynamic data are allocated in `init_neighbor` and are stored in `neighbor`.

32b   ⟨*Static function prototypes* 21⟩+≡

```
static void
init_neighbor(struct bounds *bounds, struct neighbor *neighbor);
```

32c   ⟨*Static functions* 24f⟩+≡

```
static void
init_neighbor(struct bounds *bounds, struct neighbor *neighbor)
{
    int i;

    mk_sublattice(bounds, coord);
#ifndef NO_DEBUG_QMP
    for (i = 0; i < DIM; i++)
        DEBUG_QMP("local: bounds[%d]: lo %d, hi %d\n",
                  i, bounds->lo[i], bounds->hi[i])
#endif /* defined(NO_DEBUG_QMP) */
    neighbor->qmp_smask = 0;
    ⟨Compute inside_size and boundary_size 33a⟩
    ⟨Allocate inside table 33b⟩
    ⟨Allocate boundary table 33c⟩
    ⟨Compute send sizes and allocate index tables 33d⟩
}
```

33a ⟨*Compute* `inside_size` *and* `boundary_size` 33a⟩≡

```
for (neighbor->size = 1, neighbor->inside_size = 1, i = 0; i < DIM; i++) {
    int ext = bounds->hi[i] - bounds->lo[i];

    neighbor->size *= ext;
    if (network[i] > 1)
        neighbor->inside_size *= ext - 2;
    else
        neighbor->inside_size *= ext;
}
neighbor->boundary_size = neighbor->size - neighbor->inside_size;
neighbor->site = tmalloc(neighbor->size * sizeof (struct site));
#ifndef NO_DEBUG_QMP
memset(neighbor->site, -1, neighbor->size * sizeof (struct site));
#endif
```
This code is used in chunk 32c.

33b ⟨*Allocate* `inside` *table* 33b⟩≡

```
if (neighbor->inside_size)
    neighbor->inside = tmalloc(neighbor->inside_size * sizeof (int));
else
    neighbor->inside = 0;
```
This code is used in chunk 32c.

33c ⟨*Allocate* `boundary` *table* 33c⟩≡

```
if (neighbor->boundary_size)
    neighbor->boundary = tmalloc(neighbor->boundary_size * sizeof (struct boundary));
else
    neighbor->boundary = 0;
```
This code is used in chunk 32c.

33d ⟨*Compute send sizes and allocate index tables* 33d⟩≡

```
for (i = 0; i < 2 * DIM; i++) {
    int d = i / 2;

    if (network[d] > 1) {
        neighbor->snd_size[i] = neighbor->size / (bounds->hi[d] - bounds->lo[d]);
        neighbor->snd[i] = tmalloc(neighbor->snd_size[i] * sizeof (int));
#ifndef NO_DEBUG_QMP
        memset(neighbor->snd[i], -1, neighbor->snd_size[i] * sizeof (int));
#endif
    } else {
        neighbor->snd_size[i] = 0;
        neighbor->snd[i] = 0;
    }
    DEBUG_QMP("Compute send sizes... snd_size[%d]=%d\n",
            i, neighbor->snd_size[i])
}
```
This code is used in chunk 32c.

Here is the definition of the neighbor table we spent soo much time initializing:

34a ⟨*Neighbor tables* 34a⟩≡

```
struct neighbor {
    int               size;            /* size of site table */
    int               inside_size;     /* number of inside sites */
    int               boundary_size;   /* number of boundary sites */
    int               snd_size[2*DIM]; /* size of send buffers in 8 dirs */
    int               rcv_size[2*DIM]; /* size of receive buffers */
    int               *snd[2*DIM];     /* i->x translation for send buffers */
    int               *inside;         /* i->x translation for inside sites */
    struct boundary   *boundary;       /* i->x,mask translation for boundary */
    struct site       *site;           /* x->site translation for sites */
    vHalfFermion      *snd_buf[2*DIM]; /* Send buffers */
    vHalfFermion      *rcv_buf[2*DIM]; /* Receive buffers */

    int               qmp_size[4*DIM]; /* sizes of QMP buffers */
    void              *qmp_xbuf[4*DIM]; /* QMP snd/rcv buffer addresses */
    vHalfFermion      *qmp_buf[4*DIM]; /* send and receive buffers for QMP */
    QMP_msgmem_t      qmp_mm[4*DIM];   /* msgmem's for send and receive */
    int               Nx;              /* number of msegs */

    int               qmp_smask;       /* send flags for qmp_sh[] */
    QMP_msghandle_t   qmp_handle;      /* common send & receive handle */
};
```

This definition is continued in chunk 34.
This code is used in chunks 83c and 85a.

For boundary sites we only need 8 bits for the boundary indicators. However, allocating a whole `int` for `mask` is what the compiler does anyway.

34b ⟨*Neighbor tables* 34a⟩+≡

```
struct boundary {
    int   index;   /* x-index of this boundary site */
    int   mask;    /* bitmask of the borders */
};
```

In the following structure we keep information about links and neighboors of the site. Note, that there is one address for four forward links: they are packed in memory as defined in the comment.

34c ⟨*Neighbor tables* 34a⟩+≡

```
struct site {
    int Uup;            /* up-links are Uup, Uup+1, Uup+2, Uup+3 */
    int Udown[DIM];     /* four down-links */
    int F[2*DIM];       /* eight neighboring fermions on the other sublattice */
};
```

Now we can define `build_neighbor()`:

34d ⟨*Static functions* 24f⟩+≡

```
static void
build_neighbor(struct neighbor *out,
               int             par,
               struct neighbor *in)
{
    int i,d, s, p, m;
    int x[DIM];

    ⟨Initialize out 35b⟩
    ⟨Walk through sublattice 35c⟩
    ⟨Build outside indices 36e⟩
}
```

35a   ⟨*Static function prototypes* 21⟩+≡

```
static void build_neighbor(struct neighbor *out,
                           int             parity,
                           struct neighbor *in);
```

First part is easy: we start with copying `in` to `out`, and reset fields which will be computed shortly.

35b   ⟨*Initialize* out 35b⟩≡

```
*out = *in;
out->size = 0;
out->inside_size = 0;
out->boundary_size = 0;
for (d = 0; d < DIM; d++) {
  out->rcv_size[2*d] = out->snd_size[2*d] = 0;
  out->rcv_size[2*d+1] = out->snd_size[2*d+1] = 0;
}
```

This code is used in chunk 34d.

This is a good place to reuse our lattice walking chunks.

35c   ⟨*Walk through sublattice* 35c⟩≡

```
⟨Start DIM-d sublattice scan 27d⟩
    ⟨Select same parity 35e⟩
    ⟨Compute p and m 35f⟩
    DEBUG_QMP("A: x[%d %d %d %d], (s,par)=(%d,%d), p=%d\n",
              x[0], x[1], x[2], x[3], s, par, p)
    ⟨Setup boundary or inside 36a⟩
    ⟨Build local neighbors 36d⟩
    out->size++;
    in->site++;
  next:
⟨Advance DIM-d index for a sublattice scan 27e⟩
```

This code is used in chunk 34d.

The lattice is broken into a "same" and an "opposite" parity pieces. Here are the selectors:

35d   ⟨*Select opposite parity* 35d⟩≡

```
s = parity(x);
if (s == par)
    goto next;
```

This code is used in chunk 37a.

35e   ⟨*Select same parity* 35e⟩≡

```
s = parity(x);
if (s != par)
    goto next;
```

This code is used in chunks 35c and 38b.

For `p` we use a function to compute it from `x`. As for `m`, its eight low bits encode if there is a boundary nearby. Note, that even bits corresponds to *step down* and odd bits correspond to *step up*.

35f   ⟨*Compute* p *and* m 35f⟩≡

```
p = to_HFlinear(x, &bounds, -1, 0);
for (m = 0, d = 0; d < DIM; d++) {
    if (network[d] > 1) {
        if (x[d] == bounds.lo[d])
            m |= 1 << (2 * d);
        if (x[d] + 1 == bounds.hi[d])
            m |= 1 << (2 * d + 1);
    }
}
```

This code is used in chunk 35c.

If no boundary was found near `p`, we put it into inside. Otherwise, `p` belongs to the boundary.

36a ⟨*Setup* `boundary` *or* `inside` 36a⟩≡
```
if (m) {
    ⟨Setup boundary 36c⟩
} else {
    ⟨Setup inside 36b⟩
}
```
This code is used in chunk 35c.

For the inside, simply add `p` to the list of sites and advance pointers and counters:

36b ⟨*Setup* `inside` 36b⟩≡
```
*in->inside++ = p;
out->inside_size++;
```
This code is used in chunk 36a.

For the boundary, place `p` into `index` and `m` into `mask` and advance pointers. We also take the opprotunity to place `p` into send buffers where bits of `m` are set

36c ⟨*Setup* `boundary` 36c⟩≡
```
in->boundary->index = p;
in->boundary->mask = m;
in->boundary++;
out->boundary_size++;
```
This code is used in chunk 36a.

We are ready now to build local neighbors. All gauge fields are local, and we still have `m` to tell if the other sublattice neighbor is local or not.

36d ⟨*Build local neighbors* 36d⟩≡
```
in->site->Uup = to_Ulinear(x, &bounds, -1);
for (d = 0; d < DIM; d++) {
    in->site->Udown[d] = to_Ulinear(x, &bounds, d);
    if ((m & (1 << (2 * d))) == 0)
        in->site->F[2*d] = Sv * to_HFlinear(x, &bounds, d, -1);
    if ((m & (1 << (2 * d + 1))) == 0)
        in->site->F[2*d + 1] = Sv * to_HFlinear(x, &bounds, d, +1);
}
```
This code is used in chunk 35c.

The only piece left is the one dealing with outside indices. This is a tricky part, but we just happen to have almost enough machinery already to solve it:

36e ⟨*Build outside indices* 36e⟩≡
```
for (d = 0; d < DIM; d++) {
    if (network[d] < 2)
        continue;
    construct_rec(out, par, &bounds, d, +1);
    construct_snd(out, in, par, &bounds, d, +1);
    construct_rec(out, par, &bounds, d, -1);
    construct_snd(out, in, par, &bounds, d, -1);
}
```
This code is used in chunk 34d.

We also need a function that will walk through a boundary of a neighbor building the outside part of the `site[].F` indices.

36f ⟨*Static function prototypes* 21⟩+≡
```
static void construct_rec(struct neighbor *out,
                          int par,
                          struct bounds *bounds,
                          int dir,
                          int step);
```

37a   ⟨*Static functions* 24f⟩+≡

```
static void
construct_rec(struct neighbor *out,
              int par,
              struct bounds *bounds,
              int dir,
              int step)
{
    struct bounds xb;
    int xc[DIM], x[DIM];
    int s, d, p, k;
    int dz = dir * 2 + ((step>0)?1:0);

    ⟨Construct the neighbor's network coordinates xc and bounds xb 37b⟩
    ⟨Construct the initial point of the hypersurface 37c⟩
    ⟨Start the hyperserface scan 38c⟩
        ⟨Select opposite parity 35d⟩
        ⟨Translate x to target p 37d⟩
        DEBUG_QMP("B: x[%d %d %d %d], (s,par)=(%d,%d), p=%d, k=%d\n",
                  x[0], x[1], x[2], x[3], s, par, p, k)
        ⟨Insert k into site[p].F[dx]  37e⟩
    ⟨Advance the hypersurface point 38d⟩
    out->rcv_size[dz] = k;
}
```

Constucting the neighbor's network position is straightforward:

37b   ⟨*Construct the neighbor's network coordinates* xc *and bounds* xb 37b⟩≡

```
for (d = 0; d < DIM; d++) {
    int v = coord[d] + ((d==dir)?step:0);

    if (v < 0)
        v += network[d];
    if (v >= network[d])
        v -= network[d];
    xc[d] = v;
}
mk_sublattice(&xb, xc);
#ifndef NO_DEBUG_QMP
    DEBUG_QMP("par=%d, dir=%d, step=%d\n", par, dir, step)
    for (d = 0; d < DIM; d++)
        DEBUG_QMP("neighbor: xb[%d] lo %d, di %d\n", d, xb.lo[d], xb.hi[d])
#endif /* !defined(NO_DEBUG_QMP) */
```

This code is used in chunks 37a and 38b.

The initial point should be on the surface we are walking:

37c   ⟨*Construct the initial point of the hypersurface* 37c⟩≡

```
for (d = 0; d < DIM; d++)
    x[d] = ((d == dir) && (step < 0))? (xb.hi[d] - 1): xb.lo[d];
```

This code is used in chunks 37a and 38b.

37d   ⟨*Translate* x *to target* p 37d⟩≡

```
p = to_HFlinear(x, bounds, dir, -step);
```

This code is used in chunks 37a and 38b.

37e   ⟨*Insert* k *into* site[p].F[dx] 37e⟩≡

```
out->site[p].F[dz] = Sv * k++;
```

This code is used in chunk 37a.

Constructing send buffer indices must match `construct_rec()`. Here we walk the same parity sublattice on the neighbor and index our local *opposite* parity sublattice.

38a ⟨*Static function prototypes* 21⟩+≡
```
static void construct_snd(struct neighbor *out,
                          struct neighbor *in,
                          int par,
                          struct bounds *bounds,
                          int dir,
                          int step);
```

38b ⟨*Static functions* 24f⟩+≡
```
static void
construct_snd(struct neighbor *out,
              struct neighbor *in,
              int par,
              struct bounds *bounds,
              int dir,
              int step)
{
      struct bounds xb;
      int xc[DIM], x[DIM];
      int s, d, p, k;
      int dz = dir * 2 + ((step>0)?1:0);

      ⟨Construct the neighbor's network coordinates xc and bounds xb 37b⟩
      ⟨Construct the initial point of the hypersurface 37c⟩
      ⟨Start the hyperserface scan 38c⟩
            ⟨Select same parity 35e⟩
            ⟨Translate x to target p 37d⟩
            DEBUG_QMP("C: x[%d %d %d %d], (s,par)=(%d,%d), p=%d, k=%d\n",
                      x[0], x[1], x[2], x[3], s, par, p, k)
            *in->snd[dz]++ = p * Sv;
            k++;
      ⟨Advance the hypersurface point 38d⟩
      out->snd_size[dz] = k;
}
```

38c ⟨*Start the hyperserface scan* 38c⟩≡
```
for (k = 0, d = 0; d < DIM; ) {
```
This code is used in chunks 37a and 38b.

38d ⟨*Advance the hypersurface point* 38d⟩≡
```
    next:
      for (d = 0; d < DIM; d++) {
          if (d == dir)
              continue;
          if (++x[d] == xb.hi[d])
              x[d] = xb.lo[d];
          else
              break;
      }
    }
```
This code is used in chunks 37a and 38b.

Here we do the reverse, namely, free all memory allocated by `init_tables()`:

39a ⟨*Free tables* 39a⟩≡

```
{
    int i;

    if (neighbor.site) {
        tfree(neighbor.site);
        neighbor.site = 0;
    }

    if (neighbor.inside) {
        tfree(neighbor.inside);
        neighbor.inside = 0;
    }

    if (neighbor.boundary) {
        tfree(neighbor.boundary);
        neighbor.boundary = 0;
    }

    for (i = 2 * DIM; i--;) {
        if (neighbor.snd[i] == 0)
            continue;
        tfree(neighbor.snd[i]);
        neighbor.snd[i] = 0;
    }
}
```

This code is used in chunk 15g.

5.6.2 Address translation routines

Let us define a couple of functions for translating 4-d lattice positions into 1-d offsets.

Computing linear position on the sublattice is used often enough to be placed in a function. To avoid writing two very similar functions, we pass two arguments q, and z to specify that q-component of p should adjusted by z. If $q < 0$, q and z are ignored.

39b ⟨*Static function prototypes* 21⟩+≡

```
static int
to_HFlinear(int p[],
            struct bounds *b,
            int q,
            int z)
{
    int x, d;
    for (x = 0, d = DIM; d--;) {
        int v = p[d] + ((d == q)?z:0);
        int s = b->hi[d] - b->lo[d];
        if (v < 0)
            v += tlattice[d];
        if (v >= tlattice[d])
            v -= tlattice[d];
        x = x * s + v - b->lo[d];
    }
    return x / 2;
}
```

Computing the index of the gauge link is similar to `to_HFlinear`, except that the extra parameter `q` tells us which of `p` should be stepped down by one. If $q < 0$, we are computing forward link position.

40a ⟨*Static function prototypes* 21⟩+≡

```
static int
to_Ulinear(int p[],
           struct bounds *b,
           int q)
{
    int x, d;

    if ((q < 0) || (p[q] > b->lo[q]) || (network[q] < 2)) {
        ⟨Find index of a regular gauge link 40b⟩
    } else {
        ⟨Find index of a borrowed gauge link 40c⟩
    }
}
```

Regular gauge links sits four per site and their indices are easy to compute:

40b ⟨*Find index of a regular gauge link* 40b⟩≡

```
for (x = 0, d = DIM; d--;) {
    int s = b->hi[d] - b->lo[d];
    int v = p[d] - ((q == d)?1:0);
    if (v < 0)
        v += tlattice[d];
    x = x * s + v - b->lo[d];
}
return DIM * x + ((q < 0)?0:q);
```

This code is used in chunk 40a.

For borrowed links we need first to skip all regulars and previous faces and then count position on the borrowed 3-face:

40c ⟨*Find index of a borrowed gauge link* 40c⟩≡

```
int s0, v0;
for (d = 0, v0 = 1; d < DIM; d++)
    v0 *= b->hi[d] - b->lo[d];
for (d = 0, s0 = DIM * v0; d < q; d++) {
    if (network[d] < 2)
        continue;
    s0 += v0 / (b->hi[d] - b->lo[d]);
}
for (d = DIM, x = 0; d--;) {
    int s = b->hi[d] - b->lo[d];
    int v = p[d];

    if (d == q)
        continue;
    x = x * s + v - b->lo[d];
}
return s0 + x;
```

This code is used in chunk 40a.

5.7   QMP Initialization

40d ⟨*Include files* 15c⟩+≡

```
#include <qmp.h>
```

Once the tables and sizes are known, allocate all send and receive buffers and register them with QMP.

40e ⟨*Initialize QMP* 40e⟩≡

```
if (build_buffers(&even_odd)) goto error;
if (build_buffers(&odd_even)) goto error;
```

This code is used in chunk 14b.

There are three cases we need to consider when preparing the communication handles. Note: return 1 if there was trouble.

41a ⟨*Static function prototypes* 21⟩+≡

```
static int build_buffers(struct neighbor *nb);
```

41b ⟨*Static functions* 24f⟩+≡

```
static int
build_buffers(struct neighbor *nb)
{
    int i, k, Nh;
    QMP_msghandle_t SRh[4*DIM];

    DEBUG_QMP("------------------------------------------\n")
    DEBUG_QMP("build buffers [%s]\n", nb == &even_odd? "even": "odd")
#ifndef NO_DEBUG_QMP
    {
      int i;

      DEBUG_QMP("nb->size        = %d\n", nb->size)
      DEBUG_QMP("nb->inside_size = %d\n", nb->inside_size)
      DEBUG_QMP("nb->boundary_size = %d\n", nb->boundary_size)
      for (i = 0; i < 2 * DIM; i++)
         DEBUG_QMP("[%d]: snd=%d, rcv=%d\n",
                   i, nb->snd_size[i], nb->rcv_size[i])
    }
#endif /* !defined(NO_DEBUG_QMP) */
    Nh = nb->Nx = 0;
    for (i = 0; i < DIM; i++) {
        switch (network[i]) {
            case 1: break;
            case 2:
                ⟨Clump up and down directions 41c⟩
                break;
            default:
                /* Order here is important */
                ⟨Allocate down buffers 42b⟩
                ⟨Allocate up buffers 42a⟩
                break;
        }
    }
    ⟨Construct the collective handle 43d⟩
    return 0;
}
```

If there is only two nodes in a direction, we use only up link to communicate (becasuse there is only one wire between the nodes.)

41c ⟨*Clump up and down directions* 41c⟩≡

```
DEBUG_QMP("Allocate up and down buffers, i=%d\n", i)
k = make_buffer(nb, nb->snd_size[2*i] + nb->snd_size[2*i+1]);
nb->snd_buf[2*i] = nb->qmp_buf[k];
nb->snd_buf[2*i+1] = nb->snd_buf[2*i] + Sv * nb->snd_size[2*i];
Nh = make_send(nb, k, i, +1, SRh, Nh);

k = make_buffer(nb, nb->rcv_size[2*i] + nb->rcv_size[2*i+1]);
nb->rcv_buf[2*i+1] = nb->qmp_buf[k]; /* should be opposite to snd_buf[] */
nb->rcv_buf[2*i] = nb->rcv_buf[2*i+1] + Sv * nb->rcv_size[2*i+1];
Nh = make_receive(nb, k, i, -1, SRh, Nh); /* -1 fixes a bug in GigE QMP */
```

This code is used in chunk 41b.

On a large machine, up and down buffers are separate:

42a ⟨*Allocate up buffers* 42a⟩≡
```
DEBUG_QMP("Allocate up buffers, i=%d\n", i)
k = make_buffer(nb, nb->snd_size[2*i+1]);
nb->snd_buf[2*i+1] = nb->qmp_buf[k];
Nh = make_send(nb, k, i, +1, SRh, Nh);

k = make_buffer(nb, nb->rcv_size[2*i+1]);
nb->rcv_buf[2*i+1] = nb->qmp_buf[k];
Nh = make_receive(nb, k, i, +1, SRh, Nh);
```
This code is used in chunk 41b.

42b ⟨*Allocate down buffers* 42b⟩≡
```
DEBUG_QMP("Allocate down buffers, i=%d\n", i)
k = make_buffer(nb, nb->snd_size[2*i]);
nb->snd_buf[2*i] = nb->qmp_buf[k];
Nh = make_send(nb, k, i, -1, SRh, Nh);

k = make_buffer(nb, nb->rcv_size[2*i]);
nb->rcv_buf[2*i] = nb->qmp_buf[k];
Nh = make_receive(nb, k, i, -1, SRh, Nh);
```
This code is used in chunk 41b.

Allocate a buffer of `size vHalfFermion`'s fit for send and/or receive.

42c ⟨*Static function prototypes* 21⟩+≡
```
static int make_buffer(struct neighbor *nb, int size);
```

42d ⟨*Static functions* 24f⟩+≡
```
static int
make_buffer(struct neighbor *nb, int size)
{
    int bcount = size * Sv * sizeof (vHalfFermion);
    int N = nb->Nx;

    nb->qmp_size[N] = size;
    sse_aligned_buffer(nb, N, bcount);
    nb->qmp_mm[N] = QMP_declare_msgmem(nb->qmp_buf[N], bcount);
    nb->Nx = N + 1;
    DEBUG_QMP("declare_msgmem(%p,%d)=0x%x\n",
              nb->qmp_buf[N], bcount, (int)nb->qmp_mm[N])
    return N;
}
```

Construct a send handle. This function places a send handle into `SRh` for future construction of the superhandle and sets a bit in `qmp_smask`.

42e ⟨*Static function prototypes* 21⟩+≡
```
static int make_send(struct neighbor *nb, int k, int i, int d,
                     QMP_msghandle_t SRh[4*DIM], int Nsr);
```

43a   ⟨*Static functions* 24f⟩+≡
```
    static int
    make_send(struct neighbor *nb, int k, int i, int d,
              QMP_msghandle_t SRh[4*DIM], int Nsr)
    {
        int j = 2 * i + ((d < 0)? 0: 1);

        nb->qmp_smask |= (1 << j);
        SRh[Nsr] = QMP_declare_send_relative(nb->qmp_mm[k], i, d, 1);

        DEBUG_QMP("declare_send_relative(0x%x,%d,%d,1)=0x%x\n",
                  (int)nb->qmp_mm[k], i, d, (int)SRh[Nsr])

        return Nsr+1;
    }
```

Constructing a receive handle is similar. There is no need for a mask bit though.

43b   ⟨*Static function prototypes* 21⟩+≡
```
    static int make_receive(struct neighbor *nb, int k, int i, int d,
                            QMP_msghandle_t SRh[4*DIM], int Nsr);
```

43c   ⟨*Static functions* 24f⟩+≡
```
    static int
    make_receive(struct neighbor *nb, int k, int i, int d,
                 QMP_msghandle_t SRh[4*DIM], int Nsr)
    {
        SRh[Nsr] = QMP_declare_receive_relative(nb->qmp_mm[k], i, d, 1);

        DEBUG_QMP("declare_receive_relative(0x%x,%d,%d,1)=0x%x\n",
                  (int)nb->qmp_mm[k], i, d, (int)SRh[Nsr])

        return Nsr+1;
    }
```

Finally, aggregate all receive handles:

43d   ⟨*Construct the collective handle* 43d⟩≡
```
    if (nb->qmp_smask) {
        nb->qmp_handle = QMP_declare_multiple(SRh, Nh);
    #ifndef NO_DEBUG_QMP
        {
            int i;
            for (i = 0; i < Nh; i++) {
                DEBUG_QMP("declare_multiple([%d]=0x%x)\n", i, (int)SRh[i])
            }
            DEBUG_QMP("declare_multiple(..., %d)=0x%x\n", Nh, (int)nb->qmp_handle)
        }
    #endif
    }
```
This code is used in chunk 41b.

The Vector hardware likes its memory aligned at 16 bytes. We need to keep that in mind when asking for QMP memory. Note, that this function may be in violation of a strict interpretation of the QMP Specification, but on many SciDAC calls numerous assurances were given that such usage is permissable.

43e   ⟨*Static function prototypes* 21⟩+≡
```
    static void sse_aligned_buffer(struct neighbor *nb, int k, int size);
```

44a   ⟨*Static functions* 24f⟩+≡

```
      static void
      sse_aligned_buffer(struct neighbor *nb, int k, int size)
      {
      #ifdef USE_QMP2
        nb->qmp_xbuf[k] = QMP_allocate_aligned_memory(size, 128, 0);
        nb->qmp_buf[k] = QMP_get_memory_pointer(nb->qmp_xbuf[k]);
      #else
        int xcount = size + 15;
        char *ptr = QMP_allocate_aligned_memory(xcount);
        nb->qmp_buf[k] = (void *)(~15 & (15 + (unsigned long)(ptr)));
        nb->qmp_xbuf[k] = ptr;
      #endif
        DEBUG_QMP("(%p,%d,%d): allocate: 0x%x\n",
                  nb, k, size, (int)nb->qmp_xbuf[k])
        DEBUG_QMP("(%p,%d,%d): ptr: %p\n",
                  nb, k, size, (void *)nb->qmp_buf[k])
      }
```

Freeing QMP structure does the reverse of the allocator:

44b   ⟨*Cleanup QMP* 44b⟩≡

```
      free_buffers(&even_odd);
      free_buffers(&odd_even);
```

This code is used in chunk 15g.

There are some unsettling omissions in the QMP specification. What follows is based on the tribal wisdom which was not codified.

44c   ⟨*Static function prototypes* 21⟩+≡

```
      static void free_buffers(struct neighbor *nb);
```

44d   ⟨*Static functions* 24f⟩+≡

```
      static void
      free_buffers(struct neighbor *nb)
      {
         int i;

         ⟨Free common handle 44e⟩
         ⟨Free QMP buffers 45a⟩
      }
```

Here we assume that `QMP_free_msghandle()` knows what to do with a bad handle returned from `QMP_declare_send...()` and `QMP_declare_receive...()`.

The first common wisdom is that `QMP_declare_multiple()` invalidates individual handles. We only need to free one handle in `nb->qmp_handle`:

44e   ⟨*Free common handle* 44e⟩≡

```
      if (nb->qmp_handle) {
         QMP_free_msghandle(nb->qmp_handle);
         DEBUG_QMP("free_msghandle(0x%x) / common receive handle\n",
                   (int)nb->qmp_handle)
      }
```

This code is used in chunk 44d.

Two steps are needed to deallocate QMP memory:

45a 〈*Free QMP buffers* 45a〉≡
```
for (i = nb->Nx; i--;) {
        if (nb->qmp_mm[i])
            QMP_free_msgmem(nb->qmp_mm[i]);
        DEBUG_QMP("free_msgmem(0x%x)\n", (int)nb->qmp_mm[i]);
#ifdef USE_QMP2
        if (nb->qmp_xbuf[i])
            QMP_free_memory(nb->qmp_xbuf[i]);
#else /* QMP 1.3 */
        if (nb->qmp_xbug[i])
            QMP_free_aligned_memory(nb->qmp_xbuf[i]);
#endif
        DEBUG_QMP("free_memory(0x%x)\n", (int)nb->qmp_xbuf[i]);
}
```
This code is used in chunk 44d.

5.8   Parts of the Solver

Here are three principal parts of the solver. Firts, we compute the right hand side of the equation to be solved by the CG. Next, there is a solver of a hermitian matrix. Finally, the second half of the solution is computed.

5.8.1   Compute the RHS

Here we perform steps 1–3 of the outline above.

45b 〈*Compute* φ_o 45b〉≡
```
compute_Qee1(auxA_e, eta->even);
compute_Qoe(auxB_o, auxA_e);
compute_sum_o(auxA_o, eta->odd, -1, auxB_o);
compute_Qoo1(auxB_o, auxA_o);
compute_Mx(Phi_o, auxB_o);
```
This code is used in chunk 19a.

45c 〈*Global variables* 13b〉+≡
```
static vOddFermion *auxA_o, *auxB_o, *Phi_o;
static vEvenFermion *auxA_e;
```

45d 〈*Allocate fields* 45d〉≡
```
Phi_o  = allocate_odd_fermion();  if (Phi_o == 0) goto error;
auxA_o = allocate_odd_fermion();  if (auxA_o == 0) goto error;
auxB_o = allocate_odd_fermion();  if (auxB_o == 0) goto error;
auxA_e = allocate_even_fermion(); if (auxA_e == 0) goto error;
```
This definition is continued in chunk 47.
This code is used in chunk 14b.

45e 〈*Free fields* 45e〉≡
```
if (auxA_e) free16(auxA_e); auxA_e = 0;
if (auxB_o) free16(auxB_o); auxB_o = 0;
if (auxA_o) free16(auxA_o); auxA_o = 0;
if (Phi_o)  free16(Phi_o);  Phi_o = 0;
```
This definition is continued in chunk 47.
This code is used in chunk 15g.

5.9   Field Operations

Hermitian solver follows:

45f 〈*Solve* $M^\dagger M \psi_o = \varphi_o$ 45f〉≡
```
status = cg(psi->odd, Phi_o, x0->odd, eps, min_iter, max_iter, out_eps, out_iter);
```
This code is used in chunk 19a.

⟨*Static function prototypes* 21⟩+≡

```
static int cg(vOddFermion *psi,
              const vOddFermion *b,
              const vOddFermion *x0,
              double epsilon,
              int min_iter,
              int max_iter,
              double *out_eps,
              int *out_iter);
```

46b ⟨*Static functions* 24f⟩+≡

```
static int
cg(vOddFermion *x_o,
   const vOddFermion *b,
   const vOddFermion *x0,
   double epsilon,
   int N0,
   int N,
   double *out_eps,
   int *out_N)
{
    double rho, alpha, beta, gamma, norm_z;
    int status = 1;
    int k;

    copy_o(x_o, x0);
    compute_MxM(p_o, &norm_z, x_o);
    compute_sum_oN(r_o, &rho, b, -1, p_o);
    copy_o(p_o, r_o);
    ⟨Finalize ⟨r, r⟩ computation 80h⟩

    for (k = 0; (k < N0) || ((rho > epsilon) && (k < N)); k++) {
        compute_MxM(q_o, &norm_z, p_o);
        ⟨Finalize ⟨r, r⟩ computation 80h⟩
        alpha = rho / norm_z;
        compute_sum2_oN(r_o, &gamma, -alpha, q_o);
        compute_sum2_o(x_o, alpha, p_o);
        ⟨Finalize ⟨r, r⟩ computation 80h⟩
        DEBUG_CG("cg loop: k %d , rho %g , norm_z %g , alpha %g , gamma %g\n",
                 k, rho, norm_z, alpha, gamma)
        if (gamma <= epsilon) {
            rho = gamma;
            status = 0;
            break;
        }
        beta = gamma / rho;
        rho = gamma;
        compute_sum2x_o(p_o, r_o, beta);
    }
    *out_N = k;
    *out_eps = rho;

    return status;
}
```

Temporaries used by the CG

46c ⟨*Global variables* 13b⟩+≡

```
static vOddFermion *r_o, *p_o, *q_o;
```

47a   ⟨*Allocate fields* 45d⟩+≡

```
r_o = allocate_odd_fermion(); if (r_o == 0) goto error;
p_o = allocate_odd_fermion(); if (p_o == 0) goto error;
q_o = allocate_odd_fermion(); if (q_o == 0) goto error;
```

47b   ⟨*Free fields* 45e⟩+≡

```
if (r_o) free16(r_o); r_o = 0;
if (p_o) free16(p_o); p_o = 0;
if (q_o) free16(q_o); q_o = 0;
```

5.9.1   Computing the even part of the result

Again, this is simpling performing step 5 of the outline above:

47c   ⟨*Compute* ψ_e 47c⟩≡

```
compute_Qeo(auxA_e, psi->odd);
compute_sum_e(auxB_e, eta->even, -1, auxA_e);
compute_Qee1(psi->even, auxB_e);
```

This code is used in chunk 19a.

47d   ⟨*Global variables* 13b⟩+≡

```
static vEvenFermion *auxB_e;
```

47e   ⟨*Allocate fields* 45d⟩+≡

```
auxB_e = allocate_even_fermion();  if (auxB_e == 0) goto error;
```

47f   ⟨*Free fields* 45e⟩+≡

```
if (auxB_e) free16(auxB_e); auxB_e = 0;
```

5.9.2   copy_o(d, s) or $d \leftarrow s$

This is a copy, $d \leftarrow s$:

47g   ⟨*Static function prototypes* 21⟩+≡

```
static void copy_o(vOddFermion *dst, const vOddFermion *src);
```

47h   ⟨*Static functions* 24f⟩+≡

```
static void
copy_o(vOddFermion *dst, const vOddFermion *src)
{
  int size = odd_even.size * Sv * sizeof (vOddFermion);

  memcpy(dst, src, size);
}
```

5.9.3   compute_sum2_o(d,alpha,s), or $d \leftarrow d + \alpha s$

This is a function we can not speedup much: too many bytes are needed per operation. In principle, one can play with uncached loads and stores, but let us leave that for later.

47i   ⟨*Static function prototypes* 21⟩+≡

```
static void compute_sum2_o(vOddFermion *dst, double alpha, const vOddFermion *src);
```

⟨*Static functions* 24f⟩+≡
```
  static void
  compute_sum2_o(vOddFermion *dst, double alpha, const vOddFermion *src)
  {
    vReal a = vmk_1(alpha);
    int n = odd_even.size * Sv;
    int i;

#define OP(d,c,ri) dst->f.f[d][c].ri += a * src->f.f[d][c].ri;
    for(i = 0; i < n; i++, dst++, src++) {
        LOOP_DIRAC(LOOP_COLOR, LOOP_REIM, OP)
    }
#undef OP
  }
```

Handy definitions for unrolling the loops:

⟨*Definitions* 48b⟩≡
```
  #define LOOP_REIM(m,a ...)   m(a,re) m(a,im)
  #define LOOP_COLOR(m,a ...)  m(a,0) m(a,1) m(a,2)
  #define LOOP_DIRAC(m,a ...)  m(a,0) LOOP1_DIRAC(m,a)
  #define LOOP1_DIRAC(m,a ...) m(a,1) m(a,2) m(a,3)
```
This definition is continued in chunks 60, 61, 71c, and 83.
This code is used in chunk 85a.

5.9.4 compute_sum2x_o(d,s,alpha), or $d \leftarrow \alpha d + s$

Almost the same as the previous one, but scaling is applied to another summand.

⟨*Static function prototypes* 21⟩+≡
```
  static void compute_sum2x_o(vOddFermion *dst, const vOddFermion *src, double alpha);
```

⟨*Static functions* 24f⟩+≡
```
  static void
  compute_sum2x_o(vOddFermion *dst, const vOddFermion *src, double alpha)
  {
    vReal a = vmk_1(alpha);
    int n = odd_even.size * Sv;
    int i;

#define OP(d,c,ri) dst->f.f[d][c].ri = a * dst->f.f[d][c].ri + src->f.f[d][c].ri;
    for (i = 0; i < n; i++, dst++, src++) {
        LOOP_DIRAC(LOOP_COLOR, LOOP_REIM, OP)
    }
#undef OP
  }
```

5.9.5 compute_sum_x(d,x,alpha,y) or $q \leftarrow x + \alpha y$

Next are a pair of general sums with the destination distict from the sources. Do we really need separate functions for these?

⟨*Static function prototypes* 21⟩+≡
```
  static void compute_sum_e(vEvenFermion *d,
                            const vEvenFermion *x, double alpha, const vEvenFermion *y);
  static void compute_sum_o(vOddFermion *d,
                            const vOddFermion *x, double alpha, const vOddFermion *y);
```

49a   ⟨*Static functions* 24f⟩+≡
```
static void
compute_sum_e(vEvenFermion *d,
              const vEvenFermion *x, double alpha, const vEvenFermion *y)
{
  int n = even_odd.size * Sv;
  vReal a = vmk_1(alpha);
  int i;

#define OP(s,c,ri) d->f.f[s][c].ri = x->f.f[s][c].ri + a * y->f.f[s][c].ri;
  for (i = 0; i < n; i++, d++, x++, y++) {
     LOOP_DIRAC(LOOP_COLOR, LOOP_REIM, OP)
  }
#undef OP
}
```

49b   ⟨*Static functions* 24f⟩+≡
```
static void
compute_sum_o(vOddFermion *d,
              const vOddFermion *x, double alpha, const vOddFermion *y)
{
  int n = odd_even.size * Sv;
  vReal a = vmk_1(alpha);
  int i;

#define OP(s,c,ri) d->f.f[s][c].ri = x->f.f[s][c].ri + a * y->f.f[s][c].ri;
  for (i = 0; i < n; i++, d++, x++, y++) {
     LOOP_DIRAC(LOOP_COLOR, LOOP_REIM, OP)
  }
#undef OP
}
```

5.9.6   compute_sum_oN(d,norm,x,alpha,y), or $d \leftarrow x + \alpha y$ and $r \leftarrow \langle d, d \rangle$

There are two remaining sums which compute a sum of two fermions and the norm of the result at the same time.

49c   ⟨*Static function prototypes* 21⟩+≡
```
static void compute_sum_oN(vOddFermion *d, double *norm,
                           const vOddFermion *x, double alpha, const vOddFermion *y);
```

⟨*Static functions* 24f⟩+≡

```
static void
compute_sum_oN(vOddFermion *d, double *norm,
               const vOddFermion *x, double alpha, const vOddFermion *y)
{
  vFermion *D = &d->f;
  const vFermion *X = &x->f;
  const vFermion *Y = &y->f;
  int n = odd_even.size * Sv;
  vReal a = vmk_1(alpha);
#define DF(c,r) vReal q##c##r, s##c##r;
  LOOP_COLOR(LOOP_REIM, DF)
#undef DF
  int i;

  *norm = 0;
  for (i = 0; i < n; i++, X++, Y++, D++) {
#define OP(eq,d,c,r) \
  q##c##r=D->f[d][c].r=X->f[d][c].r+a*Y->f[d][c].r; s##c##r eq q##c##r*q##c##r;
    LOOP_COLOR(LOOP_REIM, OP, =, 0)
    LOOP1_DIRAC(LOOP_COLOR, LOOP_REIM, OP, +=)
#undef OP
    *norm += vsum(s0re + s0im + s1re + s1im + s2re + s2im);
  }
  ⟨Start ⟨r, r⟩ computation 81d⟩
}
```

⟨*Static function prototypes* 21⟩+≡

```
static void compute_sum2_oN(vOddFermion *d, double *norm,
                            double alpha, const vOddFermion *y);
```

⟨*Static functions* 24f⟩+≡

```
static void
compute_sum2_oN(vOddFermion *d, double *norm,
                double alpha, const vOddFermion *y)
{
  vFermion *D = &d->f;
  const vFermion *Y = &y->f;
  int n = odd_even.size * Sv;
  vReal a = vmk_1(alpha);
#define DF(c,r) vReal q##c##r, s##c##r;
  LOOP_COLOR(LOOP_REIM, DF)
#undef DF
  int i;

  *norm = 0;
  for (i = 0; i < n; i++, Y++, D++) {
#define OP(eq,d,c,r) \
  q##c##r=D->f[d][c].r+=a*Y->f[d][c].r; s##c##r eq q##c##r*q##c##r;
    LOOP_COLOR(LOOP_REIM, OP, =, 0)
    LOOP1_DIRAC(LOOP_COLOR, LOOP_REIM, OP, +=)
#undef OP
    *norm += vsum(s0re + s0im + s1re + s1im + s2re + s2im);
  }
  ⟨Start ⟨r, r⟩ computation 81d⟩
}
```

5.9.7 `compute_MxM(eta,norm,psi)`, or $\eta \leftarrow M^{\dagger}M\psi$ and friends

Last three easy pieces.

51a ⟨*Static function prototypes* 21⟩+≡
```
static void compute_MxM(vOddFermion *eta, double *norm,
                        const vOddFermion *psi);
static void compute_M(vOddFermion *eta, double *norm,
                      const vOddFermion *psi);
static void compute_Mx(vOddFermion *eta,
                       const vOddFermion *psi);
```

51b ⟨*Static functions* 24f⟩+≡
```
static void
compute_MxM(vOddFermion *eta, double *norm,
            const vOddFermion *psi)
{
    compute_M(auxB_o, norm, psi);
    compute_Mx(eta, auxB_o);
}
```

Computation of M starts the global sum which will be completed separately.

51c ⟨*Static functions* 24f⟩+≡
```
static void compute_M(vOddFermion *eta, double *norm,
                      const vOddFermion *psi)
{
    compute_Qee1Qeo(auxA_e, psi);
    compute_1Qoo1Qoe(eta, norm, psi, auxA_e);
}
```

For $M^{\dagger}$ the order of factors differs from optimal. For now we have to live with the inefficiency here.

51d ⟨*Static functions* 24f⟩+≡
```
static void compute_Mx(vOddFermion *eta,
                       const vOddFermion *psi)
{
    compute_Soo1(auxA_o, psi);
    compute_See1Seo(auxA_e, auxA_o);
    compute_1Soe(eta, psi, auxA_e);
}
```

5.9.8 `compute_Qee1(eta,psi)`, or $\eta \leftarrow Q_{ee}^{-1}\psi$

Some code savings are still possible, since `compute_Qee1()` may differ from `compute_Qoo1()` by the number of sites only.

51e ⟨*Static function prototypes* 21⟩+≡
```
static void compute_Qxx1(vFermion *eta, const vFermion *psi, int xyzt);
static void inline compute_Qee1(vEvenFermion *eta, const vEvenFermion *psi)
{
    compute_Qxx1(&eta->f, &psi->f, even_odd.size);
}
```

5.9.9 `compute_Qoo1(eta,psi)`, or $\eta \leftarrow Q_{oo}^{-1}\psi$

51f ⟨*Static function prototypes* 21⟩+≡
```
static void inline compute_Qoo1(vOddFermion *eta, const vOddFermion *psi)
{
    compute_Qxx1(&eta->f, &psi->f, odd_even.size);
}
```

5.9.10 compute_Qxx1(eta,psi), or $\eta \leftarrow Q_{xx}^{-1}\psi$

52a ⟨*Static functions* 24f⟩+≡
```
static void
compute_Qxx1(vFermion *chi, const vFermion *psi, int size)
{
    const vFermion *qs, *qx5;
    ⟨Q common locals 80a⟩
    ⟨Qxx locals 56a⟩

    for (i = 0; i < size; i++) {
        xyzt5 = i * Sv;
        ⟨Compute rx5 80e⟩
        ⟨Compute qx5 80f⟩
        ⟨Compute Q⁻¹xx part on the s-chain 53a⟩
    }
}
```

5.9.11 compute_Soo1(eta,psi), or $\eta \leftarrow S_{oo}^{-1}\psi$

52b ⟨*Static function prototypes* 21⟩+≡
```
static void compute_Soo1(vOddFermion *eta, const vOddFermion *psi);
```

52c ⟨*Static functions* 24f⟩+≡
```
static void
compute_Soo1(vOddFermion *Chi, const vOddFermion *Psi)
{
    vFermion *chi = &Chi->f;
    const vFermion *psi = &Psi->f;
    int size = odd_even.size;
    const vFermion *qs, *qx5;
    ⟨Q common locals 80a⟩
    ⟨Qxx locals 56a⟩

    for (i = 0; i < size; i++) {
        xyzt5 = i * Sv;
        ⟨Compute rx5 80e⟩
        ⟨Compute qx5 80f⟩
        ⟨Compute S⁻¹xx part on the s-chain 53b⟩
    }
}
```

5.9.12 Q_{xx}^{-1} and S_{xx}^{-1} on a single s-chain

Therefore,

53a $\quad\langle$*Compute Q_{xx}^{-1} part on the s-chain* 53a$\rangle\equiv$
$\qquad\langle$*Compute $A^{-1}\psi$ on the upper two components* 53c$\rangle$
$\qquad\langle$*Compute $B^{-1}\psi$ on the lower two components* 53f$\rangle$
This code is used in chunks 52a, 74, and 76a.

53b $\quad\langle$*Compute S_{xx}^{-1} part on the s-chain* 53b$\rangle\equiv$
$\qquad\langle$*Compute $A^{-1}\psi$ on the lower two components* 53d$\rangle$
$\qquad\langle$*Compute $B^{-1}\psi$ on the upper two components* 53e$\rangle$
This code is used in chunks 52c and 74.

And

53c $\quad\langle$*Compute $A^{-1}\psi$ on the upper two components* 53c$\rangle\equiv$
$\qquad\langle$*Compute L_A^{-1} on the upper components* 56b$\rangle$
$\qquad\langle$*Compute R_A^{-1} on the upper components* 59b$\rangle$
This code is used in chunk 53a.

53d $\quad\langle$*Compute $A^{-1}\psi$ on the lower two components* 53d$\rangle\equiv$
$\qquad\langle$*Compute L_A^{-1} on the lower components* 57a$\rangle$
$\qquad\langle$*Compute R_A^{-1} on the lower components* 59c$\rangle$
This code is used in chunk 53b.

53e $\quad\langle$*Compute $B^{-1}\psi$ on the upper two components* 53e$\rangle\equiv$
$\qquad\langle$*Compute L_B^{-1} on the upper components* 58a$\rangle$
$\qquad\langle$*Compute R_B^{-1} on the upper components* 59d$\rangle$
This code is used in chunk 53b.

53f $\quad\langle$*Compute $B^{-1}\psi$ on the lower two components* 53f$\rangle\equiv$
$\qquad\langle$*Compute L_B^{-1} on the lower components* 58b$\rangle$
$\qquad\langle$*Compute R_B^{-1} on the lower components* 59e$\rangle$
This code is used in chunk 53a.

For both Q_{xx}^{-1} and S_{xx}^{-1} we need to compute R_A and R_B. This can be done iteratively:

$$y_k^{(A)} = \begin{cases} \frac{1}{a}z_k, & \text{if } k = n-1 \\ \frac{1}{a}z_k - \frac{b}{a}y_{k+1}^{(A)}, & \text{otherwise} \end{cases}$$
$$y_k^{(B)} = \begin{cases} \frac{1}{a}z_0, & \text{if } k = 0 \\ \frac{1}{a}z_k - \frac{b}{a}y_{k-1}^{(B)}, & \text{otherwise} \end{cases}$$

It turns out, that these computations are faster on the regular FP instructions than on vectors. For this reason corresponding parts for L_X^{-1} depend on the memory layout of `vReal`.

Let us compute constant pieces first. Division is slow, so we compute $1/a$ and $-b/a$ once and for all:

53g $\quad\langle$*Global variables* 13b$\rangle+\equiv$
```
static REAL inv_a;
static REAL b_over_a;
```

53h $\quad\langle$*Compute values from a, b and c* 53h$\rangle\equiv$
```
inv_a = 1.0 / a;
b_over_a = -b * inv_a;
```
This definition is continued in chunk 55b.
This code is used in chunk 80g.

Computing $z^{(A)} = L_A^{-1} x$ and $z^{(B)} = L_B^{-1} x$ is easy:

$$z_k^{(A)} = \begin{cases} -\sum_{j=0}^{n-2} \frac{(-b)^j c/a^{j+1}}{1+(-b)^{n-1}c/a^n} x_j + \frac{1}{1+(-b)^{n-1}c/a^n} x_{n-1}, & \text{if } k = n-1 \\ x_k, & \text{otherwise} \end{cases}$$

$$z_k^{(B)} = \begin{cases} \frac{1}{1+(-b)^{n-1}c/a^n} x_0 - \sum_{j=1}^{n-1} \frac{(-b)^{n-1-j}c/a^{n-j}}{1+(-b)^{n-1}c/a^n} x_j, & \text{if } k = 0 \\ x_k, & \text{otherwise} \end{cases}$$

We need to rewrite these expressions in a form suitable for a vector processor.
If `Vs==4`, we can write

$$z_{n-1}^{(A)} = z_a^{(A)} + z_b^{(A)} + z_c^{(A)} + z_d^{(A)}$$
$$z_0^{(B)} = z_a^{(B)} + z_b^{(B)} + z_c^{(B)} + z_d^{(B)}$$

where

$$z_a^{(A)} = \sum_{j=0}^{n/4-1} \frac{-b^{4j}c/a^{4j+1}}{1+(-b)^{n-1}c/a^n} x_{4j}$$

$$z_b^{(A)} = \sum_{j=0}^{n/4-1} \frac{b^{4j+1}c/a^{4j+2}}{1+(-b)^{n-1}c/a^n} x_{4j+1}$$

$$z_c^{(A)} = \sum_{j=0}^{n/4-1} \frac{-b^{4j+2}c/a^{4j+3}}{1+(-b)^{n-1}c/a^n} x_{4j+2}$$

$$z_d^{(A)} = \sum_{j=0}^{n/4-2} \frac{b^{4j+3}c/a^{4j+4}}{1+(-b)^{n-1}c/a^n} x_{4j+3} + \frac{1}{1+(-b)^{n-1}c/a^n} x_{n-1}$$

$$z_a^{(B)} = \frac{1}{1+(-b)^{n-1}c/a^n} x_0 + \sum_{j=1}^{n/4-1} \frac{b^{n-1-4j}c/a^{n-4j}}{1+(-b)^{n-1}c/a^n} x_{4j}$$

$$z_b^{(B)} = \sum_{j=0}^{n/4-1} \frac{-b^{n-2-4j}c/a^{n-4j-1}}{1+(-b)^{n-1}c/a^n} x_{4j+1}$$

$$z_c^{(B)} = \sum_{j=0}^{n/4-1} \frac{b^{n-3-4j}c/a^{n-4j-2}}{1+(-b)^{n-1}c/a^n} x_{4j+2}$$

$$z_d^{(B)} = \sum_{j=0}^{n/4-1} \frac{-b^{n-4-4j}c/a^{n-4j-3}}{1+(-b)^{n-1}c/a^n} x_{4j+3}$$

For `Vs==2`, we can write

$$
\begin{aligned}
z_{n-1}^{(A)} &= z_e^{(A)} + z_f^{(A)} \\
z_0^{(B)} &= z_e^{(B)} + z_f^{(B)}
\end{aligned}
$$

where

$$
\begin{aligned}
z_e^{(A)} &= \sum_{j=0}^{n/2-1} \frac{-b^{2j}c/a^{2j+1}}{1+(-b)^{n-1}c/a^n} x_{2j} \\
z_f^{(A)} &= \sum_{j=0}^{n/2-2} \frac{b^{2j+1}c/a^{2j+2}}{1+(-b)^{n-1}c/a^n} x_{2j+1} + \frac{1}{1+(-b)^{n-1}c/a^n} x_{n-1} \\
z_e^{(B)} &= \frac{1}{1+(-b)^{n-1}c/a^n} x_0 + \sum_{j=1}^{n/2-1} \frac{b^{n-1-2j}c/a^{n-2j}}{1+(-b)^{n-1}c/a^n} x_{2j} \\
z_f^{(B)} &= \sum_{j=0}^{n/2-1} \frac{-b^{n-2-2j}c/a^{n-2j-1}}{1+(-b)^{n-1}c/a^n} x_{2j+1}
\end{aligned}
$$

These sums could be effectively computed with vector operations, because their structures match DWF memory layout. Here are constants needed to compute $z^{(A)}$ and $z^{(B)}$:

55a   ⟨*Global variables* 13b⟩+≡
```
static vReal vfx_A;
static vReal vfx_B;
static vReal vab;
static REAL c0;
```

55b   ⟨*Compute values from a, b and c* 53h⟩+≡
```
c0 = 1./(1.-c/b*d_pow(b/a, Sv*Vs));
vab = vmk_1(d_pow(b/a, Vs));
vfx_A = vmk_fn(-c0*c/a, -b/a);
vfx_B = vmk_bn(-b/a, -c0*c/a);
```

5.9.13   Compute L_A^{-1} and L_B^{-1}

There are two cases:

1. L_X^{-1} is computed as part of standalone diagonal piece. In this case, the computation is done from q to r and L_X^{-1} copies elements as needed.

2. Q_{xx}^{-1} is part of combined operator. In this case q is aliased to r, and no copy is performed.

Before spelling out the details, let us define a few handy macros:

55c   ⟨*Check xx-aliasing of q* 55c⟩≡
```
#if defined(qs)
#define QSETUP(s)
#define Q2R(d,pt)
#else
#define QSETUP(s) qs = &qx5[s];
#define Q2R(d,pt) rs->f[d][c].pt = qs->f[d][c].pt;
#endif
```
This code is used in chunks 56–58.

55d   ⟨*End xx-aliasing of q* 55d⟩≡
```
#undef QSETUP
#undef Q2R
```
This code is used in chunks 56–58.

For completeness, here are definitions of variables used in the pieces bellow:

56a   ⟨Q_{xx} locals 56a⟩≡
```
vReal fx;
vHalfFermion zV;
vector_complex zn;
scalar_complex zX[Fd/2][Nc];
```
This definition is continued in chunk 59f.
This code is used in chunks 52, 63, and 64a.

56b   ⟨Compute L_A^{-1} on the upper components 56b⟩≡
```
vhfzero(&zV);
fx = vfx_A;
```
⟨Check xx-aliasing of q 55c⟩
```
for (s = 0; s < Sv_1; s++, fx = fx * vab) {
    rs = &rx5[s];
    QSETUP(s)
```
⟨Compute $zV \leftarrow zV + fx * qs^{up}$ 56c⟩
```
}
rs = &rx5[Sv_1];
QSETUP(Sv_1)
fx = vput_n(fx, c0);
```
⟨Compute $zV \leftarrow zV + fx * qs^{up}$ 56c⟩
```
for (c = 0; c < Nc; c++) {
```
  ⟨Compute wall value in zX[c] 59a⟩

```
  zn.re = qs->f[0][c].re;                zn.im = qs->f[0][c].im;
  zn.re = vput_n(zn.re, zX[0][c].re);   zn.im = vput_n(zn.im, zX[0][c].im);
  rs->f[0][c].re = zn.re;                rs->f[0][c].im = zn.im;

  zn.re = qs->f[1][c].re;                zn.im = qs->f[1][c].im;
  zn.re = vput_n(zn.re, zX[1][c].re);   zn.im = vput_n(zn.im, zX[1][c].im);
  rs->f[1][c].re = zn.re;                rs->f[1][c].im = zn.im;
}
```
⟨End xx-aliasing of q 55d⟩
This code is used in chunk 53c.

This piece is used twice: once in the loop over L_s, and the second time after correcting s_3:

56c   ⟨Compute $zV \leftarrow zV + fx * qs^{up}$ 56c⟩≡
```
for (c = 0; c < Nc; c++) {
    zV.f[0][c].re += fx * qs->f[0][c].re; Q2R(0,re)
    zV.f[0][c].im += fx * qs->f[0][c].im; Q2R(0,im)
    zV.f[1][c].re += fx * qs->f[1][c].re; Q2R(1,re)
    zV.f[1][c].im += fx * qs->f[1][c].im; Q2R(1,im)
}
```
This code is used in chunks 56b and 58a.

The only difference between L_A^{-1} on lower components is the source of the data and the destination of the result. We have to repeat most of the above pieces though.

$\langle$*Compute L_A^{-1} on the lower components* 57a$\rangle\equiv$

```
vhfzero(&zV);
fx = vfx_A;
```
$\langle$*Check xx-aliasing of q* 55c$\rangle$
```
for (s = 0; s < Sv_1; s++, fx = fx * vab) {
    rs = &rx5[s];
    QSETUP(s)
```
    $\langle$*Compute $zV \leftarrow zV + fx * qs^{down}$* 57b$\rangle$
```
}
rs = &rx5[Sv_1];
QSETUP(Sv_1)
fx = vput_n(fx, c0);
```
$\langle$*Compute $zV \leftarrow zV + fx * qs^{down}$* 57b$\rangle$
```
for (c = 0; c < Nc; c++) {
```
  $\langle$*Compute wall value in* zX[c] 59a$\rangle$

```
  zn.re = qs->f[2][c].re;              zn.im = qs->f[2][c].im;
  zn.re = vput_n(zn.re, zX[0][c].re);  zn.im = vput_n(zn.im, zX[0][c].im);
  rs->f[2][c].re = zn.re;              rs->f[2][c].im = zn.im;

  zn.re = qs->f[3][c].re;              zn.im = qs->f[3][c].im;
  zn.re = vput_n(zn.re, zX[1][c].re);  zn.im = vput_n(zn.im, zX[1][c].im);
  rs->f[3][c].re = zn.re;              rs->f[3][c].im = zn.im;
}
```
$\langle$*End xx-aliasing of q* 55d$\rangle$

This code is used in chunk 53d.

$\langle$*Compute $zV \leftarrow zV + fx * qs^{down}$* 57b$\rangle\equiv$

```
for (c = 0; c < Nc; c++) {
    zV.f[0][c].re += fx * qs->f[2][c].re; Q2R(2,re)
    zV.f[0][c].im += fx * qs->f[2][c].im; Q2R(2,im)
    zV.f[1][c].re += fx * qs->f[3][c].re; Q2R(3,re)
    zV.f[1][c].im += fx * qs->f[3][c].im; Q2R(3,im)
}
```

This code is used in chunks 57a and 58b.

For L_B^{-1} the difference is in the direction of the sweep along the s-chain:

⟨*Compute L_B^{-1} on the upper components* 58a⟩≡

```
vhfzero(&zV);
fx = vfx_B;
⟨Check xx-aliasing of q 55c⟩
for (s = Sv; --s; fx = fx * vab) {
    rs = &rx5[s];
    QSETUP(s)
    ⟨Compute zV ← zV + fx * qs^up 56c⟩
}
rs = &rx5[0];
QSETUP(0)
fx = vput_0(fx, c0);
⟨Compute zV ← zV + fx * qs^up 56c⟩
for (c = 0; c < Nc; c++) {
  ⟨Compute wall value in zX[c] 59a⟩

  zn.re = qs->f[0][c].re;              zn.im = qs->f[0][c].im;
  zn.re = vput_0(zn.re, zX[0][c].re); zn.im = vput_0(zn.im, zX[0][c].im);
  rs->f[0][c].re = zn.re;             rs->f[0][c].im = zn.im;

  zn.re = qs->f[1][c].re;              zn.im = qs->f[1][c].im;
  zn.re = vput_0(zn.re, zX[1][c].re); zn.im = vput_0(zn.im, zX[1][c].im);
  rs->f[1][c].re = zn.re;             rs->f[1][c].im = zn.im;
}
⟨End xx-aliasing of q 55d⟩
```
This code is used in chunk 53e.

Again, some repetition is needed for the lower component case:

⟨*Compute L_B^{-1} on the lower components* 58b⟩≡

```
vhfzero(&zV);
fx = vfx_B;
⟨Check xx-aliasing of q 55c⟩
for (s = Sv; --s; fx = fx * vab) {
  rs = &rx5[s];
  QSETUP(s)
  ⟨Compute zV ← zV + fx * qs^down 57b⟩
}
rs = &rx5[0];
QSETUP(0)
fx = vput_0(fx, c0);
⟨Compute zV ← zV + fx * qs^down 57b⟩
for (c = 0; c < Nc; c++) {
  ⟨Compute wall value in zX[c] 59a⟩

  zn.re = qs->f[2][c].re;              zn.im = qs->f[2][c].im;
  zn.re = vput_0(zn.re, zX[0][c].re); zn.im = vput_0(zn.im, zX[0][c].im);
  rs->f[2][c].re = zn.re;             rs->f[2][c].im = zn.im;

  zn.re = qs->f[3][c].re;              zn.im = qs->f[3][c].im;
  zn.re = vput_0(zn.re, zX[1][c].re); zn.im = vput_0(zn.im, zX[1][c].im);
  rs->f[3][c].re = zn.re;             rs->f[3][c].im = zn.im;
}
⟨End xx-aliasing of q 55d⟩
```
This code is used in chunk 53f.

By now, we have four partial sums which must be combined into z_{n-1}:

59a   ⟨*Compute wall value in* `zX[c]` 59a⟩≡
```
zX[0][c].re = vsum(zV.f[0][c].re);
zX[0][c].im = vsum(zV.f[0][c].im);
zX[1][c].re = vsum(zV.f[1][c].re);
zX[1][c].im = vsum(zV.f[1][c].im);
```
This code is used in chunks 56–58.

5.9.14   Compute R_A^{-1} and R_B^{-1}

Since R_X^{-1} is always computed after L_X^{-1}, it takes its source from `rs` and places the result back into `rs`. For R_X^{-1}, again combinations of A and B and upper and lower parts require some cut, paste and edit.

59b   ⟨*Compute R_A^{-1} on the upper components* 59b⟩≡
```
⟨Init out of bound y 60a⟩
for (s = Sv; s--;) {
    rs = &rx5[s];
    ⟨Compute y_{k,[0]}^{(A)} 60b⟩
    ⟨Compute y_{k,[1]}^{(A)} 60c⟩
}
```
This code is used in chunk 53c.

59c   ⟨*Compute R_A^{-1} on the lower components* 59c⟩≡
```
⟨Init out of bound y 60a⟩
for (s = Sv; s--;) {
    rs = &rx5[s];
    ⟨Compute y_{k,[2]}^{(A)} 60d⟩
    ⟨Compute y_{k,[3]}^{(A)} 60e⟩
}
```
This code is used in chunk 53d.

59d   ⟨*Compute R_B^{-1} on the upper components* 59d⟩≡
```
⟨Init out of bound y 60a⟩
for (s = 0; s < Sv; s++) {
    rs = &rx5[s];
    ⟨Compute y_{k,[0]}^{(B)} 60f⟩
    ⟨Compute y_{k,[1]}^{(B)} 60g⟩
}
```
This code is used in chunk 53e.

59e   ⟨*Compute R_B^{-1} on the lower components* 59e⟩≡
```
⟨Init out of bound y 60a⟩
for (s = 0; s < Sv; s++) {
    rs = &rx5[s];
    ⟨Compute y_{k,[2]}^{(B)} 60h⟩
    ⟨Compute y_{k,[3]}^{(B)} 60i⟩
}
```
This code is used in chunk 53f.

We do not handle boundary cases in a special way. Instead, the previos value of y is stored in `yOut`:

59f   ⟨Q_{xx} *locals* 56a⟩+≡
```
scalar_complex yOut[Fd/2][Nc];
```

$\langle$*Init out of bound y* 60a$\rangle\equiv$
```
yOut[0][0].re = yOut[0][0].im = 0;
yOut[0][1].re = yOut[0][1].im = 0;
yOut[0][2].re = yOut[0][2].im = 0;
yOut[1][0].re = yOut[1][0].im = 0;
yOut[1][1].re = yOut[1][1].im = 0;
yOut[1][2].re = yOut[1][2].im = 0;
```
This code is used in chunk 59.

Now, the magic of copy paste. These are some of many places where we assume `Nc==3`:

60b $\langle$*Compute* $y_{k,[0]}^{(A)}$ 60b$\rangle\equiv$
```
COMPUTE_YA(0,0,0) COMPUTE_YA(0,0,1) COMPUTE_YA(0,0,2)
```
This code is used in chunk 59b.

60c $\langle$*Compute* $y_{k,[1]}^{(A)}$ 60c$\rangle\equiv$
```
COMPUTE_YA(1,1,0) COMPUTE_YA(1,1,1) COMPUTE_YA(1,1,2)
```
This code is used in chunk 59b.

60d $\langle$*Compute* $y_{k,[2]}^{(A)}$ 60d$\rangle\equiv$
```
COMPUTE_YA(2,0,0) COMPUTE_YA(2,0,1) COMPUTE_YA(2,0,2)
```
This code is used in chunk 59c.

60e $\langle$*Compute* $y_{k,[3]}^{(A)}$ 60e$\rangle\equiv$
```
COMPUTE_YA(3,1,0) COMPUTE_YA(3,1,1) COMPUTE_YA(3,1,2)
```
This code is used in chunk 59c.

60f $\langle$*Compute* $y_{k,[0]}^{(B)}$ 60f$\rangle\equiv$
```
COMPUTE_YB(0,0,0) COMPUTE_YB(0,0,1) COMPUTE_YB(0,0,2)
```
This code is used in chunk 59d.

60g $\langle$*Compute* $y_{k,[1]}^{(B)}$ 60g$\rangle\equiv$
```
COMPUTE_YB(1,1,0) COMPUTE_YB(1,1,1) COMPUTE_YB(1,1,2)
```
This code is used in chunk 59d.

60h $\langle$*Compute* $y_{k,[2]}^{(B)}$ 60h$\rangle\equiv$
```
COMPUTE_YB(2,0,0) COMPUTE_YB(2,0,1) COMPUTE_YB(2,0,2)
```
This code is used in chunk 59e.

60i $\langle$*Compute* $y_{k,[3]}^{(B)}$ 60i$\rangle\equiv$
```
COMPUTE_YB(3,1,0) COMPUTE_YB(3,1,1) COMPUTE_YB(3,1,2)
```
This code is used in chunk 59e.

Now, the computations for a single color and given in/out- and temp- spinor indices:

60j $\langle$*Definitions* 48b$\rangle+\equiv$
```
#define COMPUTE_YA(j,t,c) {                \
    REAL *v_re = (REAL *)&rs->f[j][c].re;   \
    REAL *v_im = (REAL *)&rs->f[j][c].im;   \
    BLOCKOF_YA(j,t,c,re)                     \
    BLOCKOF_YA(j,t,c,im) }
```

60k $\langle$*Definitions* 48b$\rangle+\equiv$
```
#define COMPUTE_YB(j,t,c) {                \
    REAL *v_re = (REAL *)&rs->f[j][c].re;   \
    REAL *v_im = (REAL *)&rs->f[j][c].im;   \
    BLOCKOF_YB(j,t,c,re)                     \
    BLOCKOF_YB(j,t,c,im) }
```

Next, let us implement `BLOCKOF_*` for `Vs==2` and `Vs==4`. The target definitions of vector operations should agree with the layout here, of course.

If `Vs==4` then

61a   ⟨*Definitions* 48b⟩+≡
```
#define BLOCKOF2_YA(j,t,c,ri)                                          \
    v_##ri[1] = inv_a * v_##ri[1] + b_over_a * yOut[t][c].ri;          \
    yOut[t][c].ri = v_##ri[0] = inv_a * v_##ri[0] + b_over_a * v_##ri[1];
```

61b   ⟨*Definitions* 48b⟩+≡
```
#define BLOCKOF4_YA(j,t,c,ri)                                          \
    v_##ri[3] = inv_a * v_##ri[3] + b_over_a * yOut[t][c].ri;          \
    v_##ri[2] = inv_a * v_##ri[2] + b_over_a * v_##ri[3];              \
    v_##ri[1] = inv_a * v_##ri[1] + b_over_a * v_##ri[2];              \
    yOut[t][c].ri = v_##ri[0] = inv_a * v_##ri[0] + b_over_a * v_##ri[1];
```

61c   ⟨*Blocks for* `YA` *and* `YB` *of length four* 61c⟩≡
```
#define BLOCKOF_YA(j,t,c,ri) BLOCKOF4_YA(j,t,c,ri)
#define BLOCKOF_YB(j,t,c,ri) BLOCKOF4_YB(j,t,c,ri)
```
This code is used in chunks 86a and 91a.

Else if `Vs==2` then

61d   ⟨*Definitions* 48b⟩+≡
```
#define BLOCKOF2_YB(j,t,c,ri)                                          \
    v_##ri[0] = inv_a * v_##ri[0] + b_over_a * yOut[t][c].ri;          \
    yOut[t][c].ri = v_##ri[1] = inv_a * v_##ri[1] + b_over_a * v_##ri[0];
```

61e   ⟨*Definitions* 48b⟩+≡
```
#define BLOCKOF4_YB(j,t,c,ri)                                          \
    v_##ri[0] = inv_a * v_##ri[0] + b_over_a * yOut[t][c].ri;          \
    v_##ri[1] = inv_a * v_##ri[1] + b_over_a * v_##ri[0];              \
    v_##ri[2] = inv_a * v_##ri[2] + b_over_a * v_##ri[1];              \
    yOut[t][c].ri = v_##ri[3] = inv_a * v_##ri[3] + b_over_a * v_##ri[2];
```

61f   ⟨*Blocks for* `YA` *and* `YB` *of length two* 61f⟩≡
```
#define BLOCKOF_YA(j,t,c,ri) BLOCKOF2_YA(j,t,c,ri)
#define BLOCKOF_YB(j,t,c,ri) BLOCKOF2_YB(j,t,c,ri)
```
This code is used in chunks 88c, 93b, and 95a.

Otherwise we will get an error. That is what we want.

5.9.15   Standalone off-diagonal pieces

First, simple off-diagonal parts.

61g   ⟨*Static function prototypes* 21⟩+≡
```
static void compute_Qxy(vFermion *d, const vFermion *s, struct neighbor *nb);
```

5.9.16   compute_Qoe(d,s) or $d \leftarrow Q_{eo}s$

61h   ⟨*Static function prototypes* 21⟩+≡
```
static void inline compute_Qoe(vOddFermion *d, const vEvenFermion *s)
{
    compute_Qxy(&d->f, &s->f, &odd_even);
}
```

5.9.17   compute_Qeo(d,s) or $d \leftarrow Q_{oe}s$

61i   ⟨*Static function prototypes* 21⟩+≡
```
static void inline compute_Qeo(vEvenFermion *d, const vOddFermion *s)
{
    compute_Qxy(&d->f, &s->f, &even_odd);
}
```

5.9.18 `compute_1Soe(d,q,s)` or $d \leftarrow q - S_{eo}s$

62a   ⟨*Static function prototypes* 21⟩+≡

```
static void compute_1Sxy(vFermion *d,
                         const vFermion *q,
                         const vFermion *s,
                         struct neighbor *nb);
static void inline compute_1Soe(vOddFermion *d,
                                const vOddFermion *q,
                                const vEvenFermion *s)
{
    compute_1Sxy(&d->f, &q->f, &s->f, &odd_even);
}
```

5.9.19 `compute_Qxy(chi,psi)`, or $\chi \leftarrow Q_{xy}\psi$

62b   ⟨*Static functions* 24f⟩+≡

```
static void
compute_Qxy(vFermion *chi,
            const vFermion *psi,
            struct neighbor *nb)
{
```
      ⟨*Q common locals* 80a⟩
      ⟨*Q_{xy} locals* 80b⟩

      ⟨*Setup xy-aliasing of q* 65b⟩
      ⟨*Compute projections for Q send* 66c⟩
      ⟨*Start sends and receives* 81a⟩
      ⟨*Compute inside part for Q_{xy}* 68e⟩
      ⟨*Finish sends and receives* 81b⟩
      ⟨*Compute boundary part for Q_{xy}* 69a⟩
      ⟨*Finish xy-aliasing of q* 65c⟩
```
}
```

5.9.20 `compute_1Sxy(chi,eta,psi)`, or $\chi \leftarrow \eta - S_{xy}\psi$

For other functions, little need to be changes at this granularity. E.g., here is the final part of $M^{\dagger}$:

62c   ⟨*Static functions* 24f⟩+≡

```
static void
compute_1Sxy(vFermion *chi,
             const vFermion *eta,
             const vFermion *psi,
             struct neighbor *nb)
{
```
      ⟨*Q common locals* 80a⟩
      ⟨*Q_{xy} locals* 80b⟩

      ⟨*Setup xy-aliasing of q* 65b⟩
      ⟨*Compute projections for S send* 67a⟩
      ⟨*Start sends and receives* 81a⟩
      ⟨*Compute inside part for $1 - S_{xy}$* 71d⟩
      ⟨*Finish sends and receives* 81b⟩
      ⟨*Compute boundary part for $1 - S_{xy}$* 71e⟩
      ⟨*Finish xy-aliasing of q* 65c⟩
```
}
```

5.9.21   `compute_Qxx1Qxy(chi,psi)`, or $\chi \leftarrow Q_{xx}^{-1} Q_{xy} \psi$

  ⟨*Static functions* 24f⟩+≡

```
static void
compute_Qxx1Qxy(vFermion *chi,
                const vFermion *psi,
                struct neighbor *nb)
{
```
     ⟨*Q common locals* 80a⟩
     ⟨*Q_{xy} locals* 80b⟩
     ⟨*Q_{xx} locals* 56a⟩

     ⟨*Setup xy-aliasing of q* 65b⟩
     ⟨*Compute projections for Q send* 66c⟩
     ⟨*Start sends and receives* 81a⟩
     ⟨*Compute inside part for $Q_{xx}^{-1} Q_{xy}$* 74a⟩
     ⟨*Finish sends and receives* 81b⟩
     ⟨*Compute boundary part for $Q_{xx}^{-1} Q_{xy}$* 74b⟩
     ⟨*Finish xy-aliasing of q* 65c⟩
```
}
```

5.9.22   `compute_Sxx1Sxy(chi,psi)`, or $\chi \leftarrow S_{xx}^{-1} S_{xy} \psi$

  ⟨*Static functions* 24f⟩+≡

```
static void
compute_Sxx1Sxy(vFermion *chi,
                const vFermion *psi,
                struct neighbor *nb)
{
```
     ⟨*Q common locals* 80a⟩
     ⟨*Q_{xy} locals* 80b⟩
     ⟨*Q_{xx} locals* 56a⟩

     ⟨*Setup xy-aliasing of q* 65b⟩
     ⟨*Compute projections for S send* 67a⟩
     ⟨*Start sends and receives* 81a⟩
     ⟨*Compute inside part for $S_{xx}^{-1} S_{xy}$* 74c⟩
     ⟨*Finish sends and receives* 81b⟩
     ⟨*Compute boundary part for $S_{xx}^{-1} S_{xy}$* 74d⟩
     ⟨*Finish xy-aliasing of q* 65c⟩
```
}
```

5.9.23   `compute_1Qxx1Qxy(chi,norm,eta,psi)`, or $\chi \leftarrow \eta - Q_{xx}^{-1}Q_{xy}\psi$ and $r \leftarrow \langle\chi,\chi\rangle$

64a   ⟨*Static functions* 24f⟩+≡

```
static void
compute_1Qxx1Qxy(vFermion *chi,
                 double *norm,
                 const vFermion *eta,
                 const vFermion *psi,
                 struct neighbor *nb)
{
```
      ⟨*Q common locals* 80a⟩
      ⟨Q_{xy} *locals* 80b⟩
      ⟨Q_{xx} *locals* 56a⟩
```
      vReal vv;
      vReal nv;
      *norm = 0;
```
      ⟨*Setup xy-aliasing of q* 65b⟩
      ⟨*Compute projections for Q send* 66c⟩
      ⟨*Start sends and receives* 81a⟩
      ⟨*Compute inside part for* $1 - Q_{xx}^{-1}Q_{xy}$ 75c⟩
      ⟨*Finish sends and receives* 81b⟩
      ⟨*Compute boundary part for* $1 - Q_{xx}^{-1}Q_{xy}$ 75d⟩
      ⟨*Start* $\langle r,r \rangle$ *computation* 81d⟩
      ⟨*Finish xy-aliasing of q* 65c⟩
```
}
```

5.9.24   `compute_Dx(chi,eta,psi)`, or $\chi_x \leftarrow Q_{xx}\eta_x + Q_{xy}\psi_y$

64b   ⟨*Static functions* 24f⟩+≡

```
static void
compute_Dx(vFermion *chi,
           const vFermion *eta,
           const vFermion *psi,
           struct neighbor *nb)
{
```
      ⟨*Q common locals* 80a⟩
      ⟨Q_{xy} *locals* 80b⟩
      ⟨D_{xx} *locals* 78d⟩

      ⟨*Setup xy-aliasing of q* 65b⟩
      ⟨*Compute projections for Q send* 66c⟩
      ⟨*Start sends and receives* 81a⟩
      ⟨*Compute inside part for* $Q_{xx}\eta + Q_{xy}\psi$ 76c⟩
      ⟨*Finish sends and receives* 81b⟩
      ⟨*Compute boundary part for* $Q_{xx}\eta + Q_{xy}\psi$ 76d⟩
      ⟨*Finish xy-aliasing of q* 65c⟩
```
}
```

5.9.25   compute_Dcx(chi,eta,psi), or $\chi_x \leftarrow S_{xx}\eta_x + S_{xy}\psi_y$

65a   ⟨*Static functions* 24f⟩+≡

```
static void
compute_Dcx(vFermion *chi,
            const vFermion *eta,
            const vFermion *psi,
            struct neighbor *nb)
{
```

    ⟨*Q common locals* 80a⟩
    ⟨Q_{xy} *locals* 80b⟩
    ⟨D_{xx} *locals* 78d⟩

    ⟨*Setup xy-aliasing of q* 65b⟩
    ⟨*Compute projections for S send* 67a⟩
    ⟨*Start sends and receives* 81a⟩
    ⟨*Compute inside part for* $S_{xx}\eta + S_{xy}\psi$ 77a⟩
    ⟨*Finish sends and receives* 81b⟩
    ⟨*Compute boundary part for* $S_{xx}\eta + S_{xy}\psi$ 77b⟩
    ⟨*Finish xy-aliasing of q* 65c⟩

```
}
```

5.9.26   Aliasing macros

Remember, that Z_{xy} always puts result into `q`. For standalone diagonal pieces a couple of `define`'s help to manage `__restrict__` pointers properly.

65b   ⟨*Setup xy-aliasing of q* 65b⟩≡

```
#define qx5 rx5
#define qs rs
```

This code is used in chunks 62–65.

65c   ⟨*Finish xy-aliasing of q* 65c⟩≡

```
#undef qs
#undef qx5
```

This code is used in chunks 62–65.

5.9.27   compute_De(chi,eta,psi), or $\chi \leftarrow Q_{ee}\eta + Q_{eo}\psi$

65d   ⟨*Static function prototypes* 21⟩+≡

```
static void compute_Dx(vFermion *chi,
                       const vFermion *eta,
                       const vFermion *psi,
                       struct neighbor *nb);
static void inline compute_De(vEvenFermion *chi,
                              const vEvenFermion *eta,
                              const vOddFermion *psi)
{
    compute_Dx(&chi->f, &eta->f, &psi->f, &even_odd);
}
```

5.9.28   compute_Do(chi,eta,psi), or $\chi \leftarrow Q_{oo}\eta + Q_{oe}\psi$

65e   ⟨*Static function prototypes* 21⟩+≡

```
static void inline compute_Do(vOddFermion *chi,
                              const vOddFermion *eta,
                              const vEvenFermion *psi)
{
    compute_Dx(&chi->f, &eta->f, &psi->f, &odd_even);
}
```

5.9.29 `compute_Dce(chi,eta,psi)`, or $\chi \leftarrow S_{ee}\eta + S_{eo}\psi$

⟨*Static function prototypes* 21⟩+≡

```
static void compute_Dcx(vFermion *chi,
                        const vFermion *eta,
                        const vFermion *psi,
                        struct neighbor *nb);
static void inline compute_Dce(vEvenFermion *chi,
                               const vEvenFermion *eta,
                               const vOddFermion *psi)
{
    compute_Dcx(&chi->f, &eta->f, &psi->f, &even_odd);
}
```

5.9.30 `compute_Dco(chi,eta,psi)`, or $\chi \leftarrow S_{oo}\eta + S_{oe}\psi$

⟨*Static function prototypes* 21⟩+≡

```
static void inline compute_Dco(vOddFermion *chi,
                               const vOddFermion *eta,
                               const vEvenFermion *psi)
{
    compute_Dcx(&chi->f, &eta->f, &psi->f, &odd_even);
}
```

5.9.31 Projections to be sent

Next we compute $(1 \pm \gamma_\mu)$ projections to be sent to our neighbors. There are two cases here, one of Q_{xy} and another for S_{xy}. In principle, we might have handled both of them with some jungling of the `struct neighbor` tables, but let us go a simple if extensive way for now.

Notice that projections are done in the opposite direction than for inside sites. This is because the receiving node looks backward where we send forward.

⟨*Compute projections for Q send* 66c⟩≡

```
{
    int k, i, s, c, *src;
    const vFermion *f;
    vHalfFermion *g;

    k = 0; ⟨Construct (1 + γ₀) send k-buffer 67b⟩
    k = 1; ⟨Construct (1 − γ₀) send k-buffer 67c⟩
    k = 2; ⟨Construct (1 + γ₁) send k-buffer 67d⟩
    k = 3; ⟨Construct (1 − γ₁) send k-buffer 67e⟩
    k = 4; ⟨Construct (1 + γ₂) send k-buffer 68a⟩
    k = 5; ⟨Construct (1 − γ₂) send k-buffer 68b⟩
    k = 6; ⟨Construct (1 + γ₃) send k-buffer 68c⟩
    k = 7; ⟨Construct (1 − γ₃) send k-buffer 68d⟩
}
```

This code is used in chunks 62–64.

67a ⟨*Compute projections for S send* 67a⟩≡

```
{
    int k, i, s, c, *src;
    const vFermion *f;
    vHalfFermion *g;

    k = 0; ⟨Construct (1 − γ₀) send k-buffer 67c⟩
    k = 1; ⟨Construct (1 + γ₀) send k-buffer 67b⟩
    k = 2; ⟨Construct (1 − γ₁) send k-buffer 67e⟩
    k = 3; ⟨Construct (1 + γ₁) send k-buffer 67d⟩
    k = 4; ⟨Construct (1 − γ₂) send k-buffer 68b⟩
    k = 5; ⟨Construct (1 + γ₂) send k-buffer 68a⟩
    k = 6; ⟨Construct (1 − γ₃) send k-buffer 68d⟩
    k = 7; ⟨Construct (1 + γ₃) send k-buffer 68c⟩
}
```

This code is used in chunks 62c, 63b, and 65a.

67b ⟨*Construct* $(1 + \gamma_0)$ *send k-buffer* 67b⟩≡

```
for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
    for (s = Sv, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < Nc; c++) {
            ⟨Build (1 + γ₀) projection of *f in *g 6a⟩
        }
    }
}
```

This code is used in chunks 66c and 67a.

67c ⟨*Construct* $(1 − \gamma_0)$ *send k-buffer* 67c⟩≡

```
for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
    for (s = Sv, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < Nc; c++) {
            ⟨Build (1 − γ₀) projection of *f in *g 6c⟩
        }
    }
}
```

This code is used in chunks 66c and 67a.

67d ⟨*Construct* $(1 + \gamma_1)$ *send k-buffer* 67d⟩≡

```
for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
    for (s = Sv, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < Nc; c++) {
            ⟨Build (1 + γ₁) projection of *f in *g 6e⟩
        }
    }
}
```

This code is used in chunks 66c and 67a.

67e ⟨*Construct* $(1 − \gamma_1)$ *send k-buffer* 67e⟩≡

```
for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
    for (s = Sv, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < Nc; c++) {
            ⟨Build (1 − γ₁) projection of *f in *g 6g⟩
        }
    }
}
```

This code is used in chunks 66c and 67a.

68a   ⟨*Construct* $(1 + \gamma_2)$ *send k-buffer* 68a⟩≡

```
for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
    for (s = Sv, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < Nc; c++) {
            ⟨Build (1 + γ₂) projection of *f in *g 8a⟩
        }
    }
}
```

This code is used in chunks 66c and 67a.

68b   ⟨*Construct* $(1 - \gamma_2)$ *send k-buffer* 68b⟩≡

```
for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
     for (s = Sv, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < Nc; c++) {
            ⟨Build (1 - γ₂) projection of *f in *g 8c⟩
        }
    }
}
```

This code is used in chunks 66c and 67a.

68c   ⟨*Construct* $(1 + \gamma_3)$ *send k-buffer* 68c⟩≡

```
for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
    for (s = Sv, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < Nc; c++) {
            ⟨Build (1 + γ₃) projection of *f in *g 8e⟩
        }
    }
}
```

This code is used in chunks 66c and 67a.

68d   ⟨*Construct* $(1 - \gamma_3)$ *send k-buffer* 68d⟩≡

```
for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
    for (s = Sv, f = &psi[*src]; s--; g++, f++) {
        for (c = 0; c < Nc; c++) {
            ⟨Build (1 - γ₃) projection of *f in *g 9a⟩
        }
    }
}
```

This code is used in chunks 66c and 67a.

5.9.32   Parts of $Q_{xy}\psi$

Let us start with the simplest of the five operators we need.

68e   ⟨*Compute inside part for* Q_{xy} 68e⟩≡

```
for (i = 0; i < nb->inside_size; i++) {
    xyzt = nb->inside[i];
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute Qxy part on the inside s-chain 69b⟩
}
```

This code is used in chunk 62b.

69a ⟨*Compute boundary part for* Q_{xy} 69a⟩≡

```
for (i = 0; i < nb->boundary_size; i++) {
    int m = nb->boundary[i].mask;

    xyzt = nb->boundary[i].index;
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute Qxy part on the boundary s-chain 69c⟩
}
```

This code is used in chunk 62b.

69b ⟨*Compute* Q_{xy} *part on the inside s-chain* 69b⟩≡

```
for (s = 0; s < Sv; s++) {
    ⟨Compute Q inside γ-projections 69d⟩
    ⟨Inside multiply by V s 70c⟩
    ⟨Compute Q γ-unprojections and sum the results 70b⟩
}
```

This code is used in chunks 68e and 74–76.

69c ⟨*Compute* Q_{xy} *part on the boundary s-chain* 69c⟩≡

```
for (s = 0; s < Sv; s++) {
    ⟨Compute Q boundary γ-projections 70a⟩
    ⟨Boundary multiply by V s 71a⟩
    ⟨Compute Q γ-unprojections and sum the results 70b⟩
}
```

This code is used in chunks 69a and 74–76.

69d ⟨*Compute Q inside* γ-*projections* 69d⟩≡

```
⟨Construct neighbor pointers 79b⟩
for (c = 0; c < Nc; c++) {
    k=0; f=&psi[ps[0]]; g=&gg[0]; ⟨Build (1 − γ0) projection of *f in *g 6c⟩
    k=1; f=&psi[ps[1]]; g=&gg[1]; ⟨Build (1 + γ0) projection of *f in *g 6a⟩
    k=2; f=&psi[ps[2]]; g=&gg[2]; ⟨Build (1 − γ1) projection of *f in *g 6g⟩
    k=3; f=&psi[ps[3]]; g=&gg[3]; ⟨Build (1 + γ1) projection of *f in *g 6e⟩
    k=4; f=&psi[ps[4]]; g=&gg[4]; ⟨Build (1 − γ2) projection of *f in *g 8c⟩
    k=5; f=&psi[ps[5]]; g=&gg[5]; ⟨Build (1 + γ2) projection of *f in *g 8a⟩
    k=6; f=&psi[ps[6]]; g=&gg[6]; ⟨Build (1 − γ3) projection of *f in *g 9a⟩
    k=7; f=&psi[ps[7]]; g=&gg[7]; ⟨Build (1 + γ3) projection of *f in *g 8e⟩
}
```

This code is used in chunk 69b.

70a   ⟨*Compute Q boundary γ-projections* 70a⟩≡
   ⟨*Construct neighbor pointers* 79b⟩

```
for (c = 0; c < 3; c++) {
    if ((m & 0x01) == 0) {
        k=0; f=&psi[ps[0]]; g=&gg[0]; ⟨Build (1 − γ0) projection of *f in *g 6c⟩
    }
    if ((m & 0x02) == 0) {
        k=1; f=&psi[ps[1]]; g=&gg[1]; ⟨Build (1 + γ0) projection of *f in *g 6a⟩
    }
    if ((m & 0x04) == 0) {
        k=2; f=&psi[ps[2]]; g=&gg[2]; ⟨Build (1 − γ1) projection of *f in *g 6g⟩
    }
    if ((m & 0x08) == 0) {
        k=3; f=&psi[ps[3]]; g=&gg[3]; ⟨Build (1 + γ1) projection of *f in *g 6e⟩
    }
    if ((m & 0x10) == 0) {
        k=4; f=&psi[ps[4]]; g=&gg[4]; ⟨Build (1 − γ2) projection of *f in *g 8c⟩
    }
    if ((m & 0x20) == 0) {
        k=5; f=&psi[ps[5]]; g=&gg[5]; ⟨Build (1 + γ2) projection of *f in *g 8a⟩
    }
    if ((m & 0x40) == 0) {
        k=6; f=&psi[ps[6]]; g=&gg[6]; ⟨Build (1 − γ3) projection of *f in *g 9a⟩
    }
    if ((m & 0x80) == 0) {
        k=7; f=&psi[ps[7]]; g=&gg[7]; ⟨Build (1 + γ3) projection of *f in *g 8e⟩
    }
}
```

This code is used in chunk 69c.

70b   ⟨*Compute Q γ-unprojections and sum the results* 70b⟩≡

```
rs = &rx5[s];
for (c = 0; c < Nc; c++) {
    k = 7; ⟨Unproject (1 + γ3) link 8f⟩
    k = 6; ⟨Unproject and accumulate (1 − γ3) link 9b⟩
    k = 3; ⟨Unproject and accumulate (1 + γ1) link 6f⟩
    k = 2; ⟨Unproject and accumulate (1 − γ1) link 7⟩
    k = 1; ⟨Unproject and accumulate (1 + γ0) link 6b⟩
    k = 0; ⟨Unproject and accumulate (1 − γ0) link 6d⟩
    k = 5; ⟨Unproject and accumulate (1 + γ2) link 8b⟩
    k = 4; ⟨Unproject and accumulate (1 − γ2) link 8d⟩
}
```

This code is used in chunk 69.

Now we have everything we need to compute $U(1 \pm \gamma_\mu)\psi$ pieces:

70c   ⟨*Inside multiply by V s* 70c⟩≡

```
for (d = 0; d < 2*DIM; d++) {
    vHalfFermion * __restrict__ h = &hh[d];
    vSU3 *u = &V[d];
    g = &gg[d];
    ⟨Multiply *u by *g and store the result in *h 71b⟩
}
```

This code is used in chunks 69b, 72a, and 74e.

If the neighbor is on another node, it is in the receive buffer by now.

71a ⟨*Boundary multiply by V s* 71a⟩≡
```
for (d = 0; d < 2*DIM; d++) {
    vHalfFermion * __restrict__ h = &hh[d];
    vSU3 *u = &V[d];
    g = (m & (1 << d))? &nb->rcv_buf[d][ps[d]]: &gg[d];
    ⟨Multiply *u by *g and store the result in *h 71b⟩
}
```
This code is used in chunks 69c, 72b, and 75a.

71b ⟨*Multiply* *u *by* *g *and store the result in* *h 71b⟩≡
```
#define OP(d,c) h->f[d][c].re=u->v[c][0].re*g->f[d][0].re-u->v[c][0].im*g->f[d][0].im \
                        +u->v[c][1].re*g->f[d][1].re-u->v[c][1].im*g->f[d][1].im \
                        +u->v[c][2].re*g->f[d][2].re-u->v[c][2].im*g->f[d][2].im;\
                h->f[d][c].im=u->v[c][0].im*g->f[d][0].re+u->v[c][0].re*g->f[d][0].im \
                        +u->v[c][1].im*g->f[d][1].re+u->v[c][1].re*g->f[d][1].im \
                        +u->v[c][2].im*g->f[d][2].re+u->v[c][2].re*g->f[d][2].im;
            LOOP_HALF(LOOP_COLOR, OP)
#undef OP
```
This code is used in chunks 70c and 71a.

Run through the half fermion indices:

71c ⟨*Definitions* 48b⟩+≡
```
#define LOOP_HALF(m,a ...) m(a,0) m(a,1)
```

5.9.33   Parts of $\eta - S_{xy}\psi$

71d ⟨*Compute inside part for* $1 - S_{xy}$ 71d⟩≡
```
for (i = 0; i < nb->inside_size; i++) {
    const vFermion *ex5, *es;

    xyzt = nb->inside[i];
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ex5 = &eta[xyzt5];
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute 1 − Sxy part on the inside s-chain 72a⟩
}
```
This code is used in chunk 62c.

71e ⟨*Compute boundary part for* $1 - S_{xy}$ 71e⟩≡
```
for (i = 0; i < nb->boundary_size; i++) {
    const vFermion *ex5, *es;
    int m = nb->boundary[i].mask;

    xyzt = nb->boundary[i].index;
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ex5 = &eta[xyzt5];
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute 1 − Sxy part on the boundary s-chain 72b⟩
}
```
This code is used in chunk 62c.

72a   ⟨*Compute* $1 - S_{xy}$ *part on the inside s-chain* 72a⟩≡
```
for (s = 0; s < Sv; s++) {
      ⟨Compute S inside γ-projections 72c⟩
      ⟨Inside multiply by V s 70c⟩
      ⟨Compute 1 − S γ-unprojections and sum the results 73b⟩
}
```
This code is used in chunk 71d.

72b   ⟨*Compute* $1 - S_{xy}$ *part on the boundary s-chain* 72b⟩≡
```
for (s = 0; s < Sv; s++) {
      ⟨Compute S boundary γ-projections 73a⟩
      ⟨Boundary multiply by V s 71a⟩
      ⟨Compute 1 − S γ-unprojections and sum the results 73b⟩
}
```
This code is used in chunk 71e.

72c   ⟨*Compute S inside* γ*-projections* 72c⟩≡
```
⟨Construct neighbor pointers 79b⟩
for (c = 0; c < Nc; c++) {
      k=0; f=&psi[ps[0]]; g=&gg[0]; ⟨Build (1 + γ₀) projection of *f in *g 6a⟩
      k=1; f=&psi[ps[1]]; g=&gg[1]; ⟨Build (1 − γ₀) projection of *f in *g 6c⟩
      k=2; f=&psi[ps[2]]; g=&gg[2]; ⟨Build (1 + γ₁) projection of *f in *g 6e⟩
      k=3; f=&psi[ps[3]]; g=&gg[3]; ⟨Build (1 − γ₁) projection of *f in *g 6g⟩
      k=4; f=&psi[ps[4]]; g=&gg[4]; ⟨Build (1 + γ₂) projection of *f in *g 8a⟩
      k=5; f=&psi[ps[5]]; g=&gg[5]; ⟨Build (1 − γ₂) projection of *f in *g 8c⟩
      k=6; f=&psi[ps[6]]; g=&gg[6]; ⟨Build (1 + γ₃) projection of *f in *g 8e⟩
      k=7; f=&psi[ps[7]]; g=&gg[7]; ⟨Build (1 − γ₃) projection of *f in *g 9a⟩
}
```
This code is used in chunks 72a and 74e.

73a $\langle$*Compute S boundary γ-projections* 73a$\rangle\equiv$

```
    ⟨Construct neighbor pointers 79b⟩
    for (c = 0; c < Nc; c++) {
        if ((m & 0x01) == 0) {
            k=0; f=&psi[ps[0]]; g=&gg[0]; ⟨Build (1 + γ₀) projection of *f in *g 6a⟩
        }
        if ((m & 0x02) == 0) {
            k=1; f=&psi[ps[1]]; g=&gg[1]; ⟨Build (1 − γ₀) projection of *f in *g 6c⟩
        }
        if ((m & 0x04) == 0) {
            k=2; f=&psi[ps[2]]; g=&gg[2]; ⟨Build (1 + γ₁) projection of *f in *g 6e⟩
        }
        if ((m & 0x08) == 0) {
            k=3; f=&psi[ps[3]]; g=&gg[3]; ⟨Build (1 − γ₁) projection of *f in *g 6g⟩
        }
        if ((m & 0x10) == 0) {
            k=4; f=&psi[ps[4]]; g=&gg[4]; ⟨Build (1 + γ₂) projection of *f in *g 8a⟩
        }
        if ((m & 0x20) == 0) {
            k=5; f=&psi[ps[5]]; g=&gg[5]; ⟨Build (1 − γ₂) projection of *f in *g 8c⟩
        }
        if ((m & 0x40) == 0) {
            k=6; f=&psi[ps[6]]; g=&gg[6]; ⟨Build (1 + γ₃) projection of *f in *g 8e⟩
        }
        if ((m & 0x80) == 0) {
            k=7; f=&psi[ps[7]]; g=&gg[7]; ⟨Build (1 − γ₃) projection of *f in *g 9a⟩
        }
    }
```

This code is used in chunks 72b and 75a.

73b $\langle$*Compute $1 - S$ γ-unprojections and sum the results* 73b$\rangle\equiv$

```
    rs = &rx5[s];
    es = &ex5[s];
    for (c = 0; c < Nc; c++) {
        k = 6; ⟨Unproject (1 + γ₃) link 8f⟩
        k = 7; ⟨Unproject and accumulate (1 − γ₃) link 9b⟩
        k = 2; ⟨Unproject and accumulate (1 + γ₁) link 6f⟩
        k = 3; ⟨Unproject and accumulate (1 − γ₁) link 7⟩
        k = 1; ⟨Unproject and accumulate (1 − γ₀) link 6d⟩
        k = 0; ⟨Unproject and accumulate (1 + γ₀) link 6b⟩
        k = 5; ⟨Unproject and accumulate (1 − γ₂) link 8d⟩
        k = 4; ⟨Unproject and accumulate (1 + γ₂) link 8b⟩
        ⟨Compute (∗rs) ← η − (∗rs) for color c 73c⟩
    }
```

This code is used in chunk 72.

73c $\langle$*Compute $(\ast rs) \leftarrow \eta - (\ast rs)$ for color c* 73c$\rangle\equiv$

```
    rs->f[0][c].re = es->f[0][c].re - rs->f[0][c].re;
    rs->f[0][c].im = es->f[0][c].im - rs->f[0][c].im;
    rs->f[1][c].re = es->f[1][c].re - rs->f[1][c].re;
    rs->f[1][c].im = es->f[1][c].im - rs->f[1][c].im;
    rs->f[2][c].re = es->f[2][c].re - rs->f[2][c].re;
    rs->f[2][c].im = es->f[2][c].im - rs->f[2][c].im;
    rs->f[3][c].re = es->f[3][c].re - rs->f[3][c].re;
    rs->f[3][c].im = es->f[3][c].im - rs->f[3][c].im;
```

This code is used in chunk 73b.

5.9.34 Parts of $Q_{xx}^{-1}Q_{xy}\psi$

74a　⟨*Compute inside part for $Q_{xx}^{-1}Q_{xy}$* 74a⟩≡

```
for (i = 0; i < nb->inside_size; i++) {
    xyzt = nb->inside[i];
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute Q_xy part on the inside s-chain 69b⟩
    ⟨Compute Q_xx^-1 part on the s-chain 53a⟩
}
```

This code is used in chunk 63a.

74b　⟨*Compute boundary part for $Q_{xx}^{-1}Q_{xy}$* 74b⟩≡

```
for (i = 0; i < nb->boundary_size; i++) {
    int m = nb->boundary[i].mask;

    xyzt = nb->boundary[i].index;
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute Q_xy part on the boundary s-chain 69c⟩
    ⟨Compute Q_xx^-1 part on the s-chain 53a⟩
}
```

This code is used in chunk 63a.

5.9.35 Parts of $S_{xx}^{-1}S_{xy}\psi$

74c　⟨*Compute inside part for $S_{xx}^{-1}S_{xy}$* 74c⟩≡

```
for (i = 0; i < nb->inside_size; i++) {
    xyzt = nb->inside[i];
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute S_xy part on the inside s-chain 74e⟩
    ⟨Compute S_xx^-1 part on the s-chain 53b⟩
}
```

This code is used in chunk 63b.

74d　⟨*Compute boundary part for $S_{xx}^{-1}S_{xy}$* 74d⟩≡

```
for (i = 0; i < nb->boundary_size; i++) {
    int m = nb->boundary[i].mask;

    xyzt = nb->boundary[i].index;
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute S_xy part on the boundary s-chain 75a⟩
    ⟨Compute S_xx^-1 part on the s-chain 53b⟩
}
```

This code is used in chunk 63b.

74e　⟨*Compute S_{xy} part on the inside s-chain* 74e⟩≡

```
for (s = 0; s < Sv; s++) {
    ⟨Compute S inside γ-projections 72c⟩
    ⟨Inside multiply by V s 70c⟩
    ⟨Compute S γ-unprojections and sum the results 75b⟩
}
```

This code is used in chunks 74c and 77a.

75a $\langle$*Compute S_{xy} part on the boundary s-chain* 75a$\rangle\equiv$
```
for (s = 0; s < Sv; s++) {
    ⟨Compute S boundary γ-projections 73a⟩
    ⟨Boundary multiply by V s 71a⟩
    ⟨Compute S γ-unprojections and sum the results 75b⟩
}
```
This code is used in chunks 74d and 77b.

75b $\langle$*Compute S γ-unprojections and sum the results* 75b$\rangle\equiv$
```
rs = &rx5[s];
for (c = 0; c < Nc; c++) {
    k = 6; ⟨Unproject (1 + γ₃) link 8f⟩
    k = 7; ⟨Unproject and accumulate (1 − γ₃) link 9b⟩
    k = 2; ⟨Unproject and accumulate (1 + γ₁) link 6f⟩
    k = 3; ⟨Unproject and accumulate (1 − γ₁) link 7⟩
    k = 1; ⟨Unproject and accumulate (1 − γ₀) link 6d⟩
    k = 0; ⟨Unproject and accumulate (1 + γ₀) link 6b⟩
    k = 5; ⟨Unproject and accumulate (1 − γ₂) link 8d⟩
    k = 4; ⟨Unproject and accumulate (1 + γ₂) link 8b⟩
}
```
This code is used in chunks 74e and 75a.

5.9.36 Parts of $\eta - Q_{xx}^{-1}Q_{xy}\psi$

75c $\langle$*Compute inside part for $1 - Q_{xx}^{-1}Q_{xy}$* 75c$\rangle\equiv$
```
for (i = 0; i < nb->inside_size; i++) {
    const vFermion *ex5, *es;

    xyzt = nb->inside[i];
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ex5 = &eta[xyzt5];
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute Q_xy part on the inside s-chain 69b⟩
    ⟨Compute 1 − Q_xx⁻¹ part on the s-chain 76a⟩
}
```
This code is used in chunk 64a.

75d $\langle$*Compute boundary part for $1 - Q_{xx}^{-1}Q_{xy}$* 75d$\rangle\equiv$
```
for (i = 0; i < nb->boundary_size; i++) {
    const vFermion *ex5, *es;
    int m = nb->boundary[i].mask;

    xyzt = nb->boundary[i].index;
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ex5 = &eta[xyzt5];
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute Q_xy part on the boundary s-chain 69c⟩
    ⟨Compute 1 − Q_xx⁻¹ part on the s-chain 76a⟩
}
```
This code is used in chunk 64a.

76a ⟨*Compute* $1 - Q_{xx}^{-1}$ *part on the s-chain* 76a⟩≡
        ⟨*Compute* Q_{xx}^{-1} *part on the s-chain* 53a⟩
        ```
        for (s = 0; s < Sv; s++) {
            rs = &rx5[s];
            es = &ex5[s];
            nv = vmk_1(0.0);
            for (c = 0; c < Nc; c++) {
                ⟨Compute (*rs) ← η − (*rs) and collect ⟨r, r⟩ 76b⟩
            }
            *norm += vsum(nv);
        }
        ```
        This code is used in chunk 75.

76b ⟨*Compute* $(*rs) \leftarrow \eta - (*rs)$ *and collect* $\langle r, r \rangle$ 76b⟩≡
        ```
        vv = es->f[0][c].re - rs->f[0][c].re; rs->f[0][c].re = vv; nv += vv * vv;
        vv = es->f[0][c].im - rs->f[0][c].im; rs->f[0][c].im = vv; nv += vv * vv;
        vv = es->f[1][c].re - rs->f[1][c].re; rs->f[1][c].re = vv; nv += vv * vv;
        vv = es->f[1][c].im - rs->f[1][c].im; rs->f[1][c].im = vv; nv += vv * vv;
        vv = es->f[2][c].re - rs->f[2][c].re; rs->f[2][c].re = vv; nv += vv * vv;
        vv = es->f[2][c].im - rs->f[2][c].im; rs->f[2][c].im = vv; nv += vv * vv;
        vv = es->f[3][c].re - rs->f[3][c].re; rs->f[3][c].re = vv; nv += vv * vv;
        vv = es->f[3][c].im - rs->f[3][c].im; rs->f[3][c].im = vv; nv += vv * vv;
        ```
        This code is used in chunk 76a.

5.9.37 Parts of $Q_{xx}\eta + Q_{xy}\psi$

76c ⟨*Compute inside part for* $Q_{xx}\eta + Q_{xy}\psi$ 76c⟩≡
        ```
        for (i = 0; i < nb->inside_size; i++) {
            const vFermion *ex5, *es;

            xyzt = nb->inside[i];
            xyzt5 = xyzt * Sv;
            ⟨Extract 1-d addresses 80d⟩
            ex5 = &eta[xyzt5];
            ⟨Build vector SU(3) objects 79a⟩
            ⟨Compute Qxy part on the inside s-chain 69b⟩
            ⟨Compute Qxxη + χ part on the s-chain 76e⟩
        }
        ```
        This code is used in chunk 64b.

76d ⟨*Compute boundary part for* $Q_{xx}\eta + Q_{xy}\psi$ 76d⟩≡
        ```
        for (i = 0; i < nb->boundary_size; i++) {
            const vFermion *ex5, *es;
            int m = nb->boundary[i].mask;

            xyzt = nb->boundary[i].index;
            xyzt5 = xyzt * Sv;
            ⟨Extract 1-d addresses 80d⟩
            ex5 = &eta[xyzt5];
            ⟨Build vector SU(3) objects 79a⟩
            ⟨Compute Qxy part on the boundary s-chain 69c⟩
            ⟨Compute Qxxη + χ part on the s-chain 76e⟩
        }
        ```
        This code is used in chunk 64b.

76e ⟨*Compute* $Q_{xx}\eta + \chi$ *part on the s-chain* 76e⟩≡
        ⟨*Compute* $\chi + A\eta$ *on the upper components* 77d⟩
        ⟨*Compute* $\chi + B\eta$ *on the lower components* 78c⟩
        This code is used in chunk 76.

5.9.38 Parts of $S_{xx}\eta + S_{xy}\psi$

77a ⟨*Compute inside part for $S_{xx}\eta + S_{xy}\psi$ 77a*⟩≡

```
for (i = 0; i < nb->inside_size; i++) {
    const vFermion *ex5, *es;

    xyzt = nb->inside[i];
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ex5 = &eta[xyzt5];
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute Sxy part on the inside s-chain 74e⟩
    ⟨Compute Sxxη + χ part on the s-chain 77c⟩
}
```

This code is used in chunk 65a.

77b ⟨*Compute boundary part for $S_{xx}\eta + S_{xy}\psi$ 77b*⟩≡

```
for (i = 0; i < nb->boundary_size; i++) {
    const vFermion *ex5, *es;
    int m = nb->boundary[i].mask;

    xyzt = nb->boundary[i].index;
    xyzt5 = xyzt * Sv;
    ⟨Extract 1-d addresses 80d⟩
    ex5 = &eta[xyzt5];
    ⟨Build vector SU(3) objects 79a⟩
    ⟨Compute Sxy part on the boundary s-chain 75a⟩
    ⟨Compute Sxxη + χ part on the s-chain 77c⟩
}
```

This code is used in chunk 65a.

77c ⟨*Compute $S_{xx}\eta + \chi$ part on the s-chain 77c*⟩≡

```
⟨Compute χ + Bη on the upper components 78b⟩
⟨Compute χ + Aη on the lower components 78a⟩
```

This code is used in chunk 77.

5.9.39 Computing A and B

77d ⟨*Compute $\chi + A\eta$ on the upper components 77d*⟩≡

```
for (s = Sv, vbc = vbnc, es1 = &ex5[0]; s--; vbc = vb) {
    es = &ex5[s];
    rs = &rx5[s];

#define QXX(d,c,r) rs->f[d][c].r += va * es->f[d][c].r \
                        + vbc * shift_up1(es->f[d][c].r, es1->f[d][c].r)
    QXX(0,0,re); QXX(0,0,im);
    QXX(0,1,re); QXX(0,1,im);
    QXX(0,2,re); QXX(0,2,im);
    QXX(1,0,re); QXX(1,0,im);
    QXX(1,1,re); QXX(1,1,im);
    QXX(1,2,re); QXX(1,2,im);
#undef QXX
    es1 = es;
}
```

This code is used in chunk 76e.

78a   ⟨*Compute* χ + Aη *on the lower components* 78a⟩≡

```
for (s = Sv, vbc = vbnc, es1 = &ex5[0]; s--; vbc = vb) {
    es = &ex5[s];
    rs = &rx5[s];

#define QXX(d,c,r) rs->f[d][c].r += va * es->f[d][c].r \
                        + vbc * shift_up1(es->f[d][c].r, es1->f[d][c].r)
    QXX(2,0,re); QXX(2,0,im);
    QXX(2,1,re); QXX(2,1,im);
    QXX(2,2,re); QXX(2,2,im);
    QXX(3,0,re); QXX(3,0,im);
    QXX(3,1,re); QXX(3,1,im);
    QXX(3,2,re); QXX(3,2,im);
#undef QXX
    es1 = es;
}
```

This code is used in chunk 77c.

78b   ⟨*Compute* χ + Bη *on the upper components* 78b⟩≡

```
for (s = 0, vbc = vcbn, es1 = &ex5[Sv_1]; s < Sv; s++, vbc = vb) {
    es = &ex5[s];
    rs = &rx5[s];

#define QXX(d,c,r) rs->f[d][c].r += va * es->f[d][c].r \
                        + vbc * shift_upN(es1->f[d][c].r, es->f[d][c].r)
    QXX(0,0,re); QXX(0,0,im);
    QXX(0,1,re); QXX(0,1,im);
    QXX(0,2,re); QXX(0,2,im);
    QXX(1,0,re); QXX(1,0,im);
    QXX(1,1,re); QXX(1,1,im);
    QXX(1,2,re); QXX(1,2,im);
#undef QXX
    es1 = es;
}
```

This code is used in chunk 77c.

78c   ⟨*Compute* χ + Bη *on the lower components* 78c⟩≡

```
for (s = 0, vbc = vcbn, es1 = &ex5[Sv_1]; s < Sv; s++, vbc = vb) {
    es = &ex5[s];
    rs = &rx5[s];

#define QXX(d,c,r) rs->f[d][c].r += va * es->f[d][c].r \
                        + vbc * shift_upN(es1->f[d][c].r, es->f[d][c].r)
    QXX(2,0,re); QXX(2,0,im);
    QXX(2,1,re); QXX(2,1,im);
    QXX(2,2,re); QXX(2,2,im);
    QXX(3,0,re); QXX(3,0,im);
    QXX(3,1,re); QXX(3,1,im);
    QXX(3,2,re); QXX(3,2,im);
#undef QXX
    es1 = es;
}
```

This code is used in chunk 76e.

78d   ⟨D_{xx} *locals* 78d⟩≡

```
const vFermion *es1;
vReal vbc;
```

This code is used in chunks 64b and 65a.

5.9.40 Miscallenious

We also need to uplift the gauge fields

79a ⟨*Build vector SU*(3) *objects* 79a⟩≡

```
Uup = &U[nb->site[xyzt].Uup];
for (d = 0; d < DIM; d++, Uup++) {
    Udown = &U[nb->site[xyzt].Udown[d]];
    for (c1 = 0; c1 < Nc; c1++) {
        for (c2 = 0; c2 < Nc; c2++) {
            /* conjugate down-link */
            V[d*2+0].v[c1][c2].re = vmk_1( Udown->v[c2][c1].re);
            V[d*2+0].v[c1][c2].im = vmk_1(-Udown->v[c2][c1].im);
            /* normal up-link */
            V[d*2+1].v[c1][c2].re = vmk_1(Uup->v[c1][c2].re);
            V[d*2+1].v[c1][c2].im = vmk_1(Uup->v[c1][c2].im);
        }
    }
}
```

This code is used in chunks 68e, 69a, 71, and 74–77.

We want to keep code small, so computing the neighbors is done in a loop:

79b ⟨*Construct neighbor pointers* 79b⟩≡

```
for (d = 0; d < 2*DIM; d++) {
  ps[d] = p5[d] + s;
}
```

This code is used in chunks 69d, 70a, 72c, and 73a.

5.9.41 Combined pieces

In these cases, Q_{xx}^{-1} is applied to the result of Q_{xy}

79c ⟨*Static function prototypes* 21⟩+≡

```
static void compute_Qxx1Qxy(vFermion *d,
                            const vFermion *s,
                            struct neighbor *nb);
static void inline compute_Qee1Qeo(vEvenFermion *d, const vOddFermion *s)
{
  compute_Qxx1Qxy(&d->f, &s->f, &even_odd);
}

static void compute_Sxx1Sxy(vFermion *d,
                            const vFermion *s,
                            struct neighbor *nb);
static void inline compute_See1Seo(vEvenFermion *d, const vOddFermion *s)
{
  compute_Sxx1Sxy(&d->f, &s->f, &even_odd);
}

static void compute_1Qxx1Qxy(vFermion *d,
                             double *norm,
                             const vFermion *q,
                             const vFermion *s,
                             struct neighbor *nb);
static void inline compute_1Qoo1Qoe(vOddFermion *d,
                                    double *norm,
                                    const vOddFermion *q,
                                    const vEvenFermion *s)
{
  compute_1Qxx1Qxy(&d->f, norm, &q->f, &s->f, &odd_even);
}
```

5.9.42 Common locals

Some local bindings are used by all parts above. Let us collect them together.

80a ⟨*Q common locals* 80a⟩≡
```
int i, xyzt5, s, c;
vFermion * __restrict__ rx5, * __restrict__ rs;
```
This code is used in chunks 52 and 62–65.

Others are used only in Z_{xy} parts:

80b ⟨*Q_{xy} locals* 80b⟩≡
```
int xyzt, k, d;
const vFermion *f;
vHalfFermion *g;
vHalfFermion gg[2*DIM], hh[2*DIM];
vSU3 V[2*DIM];
int ps[2*DIM], p5[2*DIM];
```
This definition is continued in chunk 80c.
This code is used in chunks 62–65.

80c ⟨*Q_{xy} locals* 80b⟩+≡
```
const SU3 *Uup, *Udown;
int c1, c2;
```

For the inside sites, compute the *s*-chain address of the neighbor. For the boundary sites, the address of the *s*-chain in the receive buffer is used instead:

80d ⟨*Extract 1-d addresses* 80d⟩≡
```
for (d = 0; d < 2*DIM; d++)
    p5[d] = nb->site[xyzt].F[d];
```
⟨*Compute* rx5 80e⟩
This code is used in chunks 68e, 69a, 71, and 74–77.

80e ⟨*Compute* rx5 80e⟩≡
```
rx5 = &chi[xyzt5];
```
This code is used in chunks 52 and 80d.

80f ⟨*Compute* qx5 80f⟩≡
```
qx5 = &psi[xyzt5];
```
This code is used in chunk 52.

5.9.43 Common globals

Some of these values depend of `m_f` and `M`. Here we compute their values:

80g ⟨*Compute constant values for Q_{xx}^{-1} and S_{xx}^{-1}* 80g⟩≡
```
{
    double a = M;
    double b = 2.;
    double c = -2*m_f;

    ⟨Compute values from a, b and c 53h⟩
}
```
This code is used in chunk 19a.

5.10 QMP Pieces

Here are miscalenious piece of QMP:
We are ready to use the result of the global sum. Check that it has been computed.

80h ⟨*Finalize ⟨r, r⟩ computation* 80h⟩≡
```
/* relax, QMP does not support split reductions yet. */
```
This code is used in chunk 46b.

Because there is no confirming QMP implementation, we have to deal with esoteric restrictions that are imposed rhyme or reason and are not documented either. Otherwise, they would be not esoteric, ain't they?

Restriction one says that deadlocks happen if there more8 than one active handle at a time. We had already packed all communication into a single superhandle, now it is a time to use it. To be extra paranoic, we check if there is a handle before using it.

81a ⟨*Start sends and receives* 81a⟩≡

```
if (nb->qmp_smask) {
#ifndef NO_DEBUG_QMP
    cleanup_receivers(nb);
    dump_buffers("start", nb);
    DEBUG_QMP("start sends and receives (0x%x)\n", (int)nb->qmp_handle)
#endif
    QMP_start(nb->qmp_handle);
}
```

This code is used in chunks 62–65.

When it is a time to use the received data, we wait for the sends as well. It is a waste, but we have to work with software that we have.

81b ⟨*Finish sends and receives* 81b⟩≡

```
if (nb->qmp_smask) {
    QMP_wait(nb->qmp_handle);
#ifndef NO_DEBUG_QMP
    DEBUG_QMP("waiting for sends and receives (0x%x)\n",
                (int)nb->qmp_handle)
    dump_buffers("wait", nb);
    cleanup_senders(nb);
#endif
}
```

This code is used in chunks 62–65.

We need to print the version of the code from node 0.

81c ⟨*Show DWF version* 81c⟩≡

```
{
    if (QMP_get_node_number() == 0) {
        QMP_printf("DWF init: %s (" MACHINE ")\n", version);
    }
}
```

This code is used in chunk 14b.

5.10.1   Global sums

Until the split global sums are implemented in QMP, everything is done at the beginning, when `*norm` contains the local part of the sum. Start the global operation which will distribute the pieces, compute the sum, and provide the result to each node.

81d ⟨*Start* $\langle r|r \rangle$ *computation* 81d⟩≡

```
DEBUG_QMP("sum_double(%p): before <r|r>: %g\n", norm, *norm)
QMP_sum_double(norm);
DEBUG_QMP("after <r|r>: %g\n", *norm)
```

This code is used in chunks 50 and 64a.

5.11 Generally Useful Functions

Here is a collection of simple functions that are useful throughout the code:

82a ⟨*Static function prototypes* 21⟩+≡

```
static inline int
parity(const int x[DIM])
{
    int i, v;
    for (i = v = 0; i < DIM; i++)
        v += x[i];
    return v & 1;
}
```

We need some powers, but `pow()` is too generic. Here's a simple version

82b ⟨*Static function prototypes* 21⟩+≡

```
static double
d_pow(double x, unsigned int n)
{
    double v = 1;

    while (n) {
        if (n & 1)
            v *= x;
        x *= x;
        n /= 2;
    }
    return v;
}
```

Clear a half fermion:

82c ⟨*Static function prototypes* 21⟩+≡

```
static inline void
vhfzero(vHalfFermion *v)
{
    vReal z = vmk_1(0.0);

    v->f[0][0].re = v->f[0][0].im =
    v->f[0][1].re = v->f[0][1].im =
    v->f[0][2].re = v->f[0][2].im =
    v->f[1][0].re = v->f[1][0].im =
    v->f[1][1].re = v->f[1][1].im =
    v->f[1][2].re = v->f[1][2].im = z;
}
```

5.12 Debug Aids

Here are macros for debugging QMP.

83a ⟨*Definitions* 48b⟩+≡
```
#ifdef DEBUG_CG
#define DEBUG_CG(msg,a ...) do \
      printf("[%05d] %s:%d:QMP/%s(): " msg, QMP_get_node_number(), \
                              __FILE__,__LINE__,__FUNCTION__,##a); \
   while(0);
#else /* !defined(DEBUG_CG) */
#define DEBUG_CG(msg,a ...)
#define NO_DEBUG_CG
#endif /* defined(DEBUG_CG) */


#ifdef DEBUG_QMP
#undef DEBUG_QMP
#define DEBUG_QMP(msg,a ...) do \
      printf("[%05d] %s:%d:QMP/%s(): " msg, QMP_get_node_number(), \
                              __FILE__,__LINE__,__FUNCTION__,##a); \
   while(0);
#else /* !defined(DEBUG_QMP) */
#define DEBUG_QMP(msg,a ...)
#define NO_DEBUG_QMP
#endif /* defined(DEBUG_QMP) */
```

Another debug aid for other pieces of the code:

83b ⟨*Definitions* 48b⟩+≡
```
#ifdef DEBUG_DWF
#undef DEBUG_DWF
#define DEBUG_DWF(msg,a ...) do \
      printf("[%05d] %s: %d: DWF/%s(): " msg, QMP_get_node_number(), \
                              __FILE__,__LINE__,__FUNCTION__,##a); \
   while(0);
#else /* !defined(DEBUG_DWF) */
#define DEBUG_DWF(msg, a ...)
#define NO_DEBUG_DWF
#endif /* defined(DEBUG_DWF) */
```

5.12.1 Neighbor table debugging

83c ⟨*dwf-tables.h* 83c⟩≡
```
   ⟨Neighbor tables 34a⟩
   extern struct neighbor even_odd;
   extern struct neighbor odd_even;
```
Root chunk (not used in this document).

5.12.2 Communication debugging

Let us start with prototypes.

83d ⟨*Static function prototypes* 21⟩+≡
```
#ifndef NO_DEBUG_QMP
static void cleanup_receivers(struct neighbor *nb);
static void cleanup_senders(struct neighbor *nb);
static void dump_buffers(const char *name, struct neighbor *nb);
#endif /* !defined(NO_DEBUG_QMP) */
```

And their implementations. The cleanups fills buffers with `0x1q` for sends and `0x2q` for receives, where `q` is the direction.

84 ⟨*Static functions* 24f⟩+≡

```
#ifndef NO_DEBUG_QMP
static void
cleanup_receivers(struct neighbor *nb)
{
    int i;

    for (i = 0; i < 2 * DIM; i++) {
        if (nb->rcv_size[i])
            memset(nb->rcv_buf[i],
                    0x20 + i,
                    nb->rcv_size[i] * Sv * sizeof (vHalfFermion));
    }
}

static void
cleanup_senders(struct neighbor *nb)
{
    int i;

    for (i = 0; i < 2 * DIM; i++) {
        if (nb->snd_size[i])
            memset(nb->snd_buf[i],
                    0x10 + i,
                    nb->snd_size[i] * Sv * sizeof (vHalfFermion));
    }
}

static void
dump_buffer(const char *name,
            const char *type,
            int        num,
            void       *ptr,
            int        size)
{
    unsigned char *p = ptr;
    int count = size * Sv * sizeof (vHalfFermion);
    int i;
    char buffer[200];
    int node = QMP_get_node_number();

    if (count == 0) {
        printf("[%05d] %s %s[%d]: empty\n", node, name, type, num);
        return;
    }
    for (i = 0; i < count;) {
        int j = i;
        int k = 0;
        for (buffer[0] = 0, k = 0; (k < 16) && (i < count); k++, i++, p++) {
            char v[10];
            sprintf(v, " %02x", *p);
            strcat(buffer, v);
        }
        printf("[%05d] %s %s[%d]: %08x %s\n", node, name, type, num, j, buffer);
    }
}
```

```
      static void
      dump_buffers(const char *name, struct neighbor *nb)
      {
          int i;

          for (i = 0; i < 2 * DIM; i++)
              dump_buffer(name, "s", i, nb->snd_buf[i], nb->snd_size[i]);
          for (i = 0; i < 2 * DIM; i++)
              dump_buffer(name, "r", i, nb->rcv_buf[i], nb->rcv_size[i]);
      }
      #endif /* !defined(NO_DEBUG_QMP) */
```

5.13 Source Files

The main body of the code resides in a single file:

85a ⟨*dwf.c* 85a⟩≡
   ⟨*Include files* 15c⟩
   ⟨*Definitions* 48b⟩
   ⟨*Data types* 18b⟩
   ⟨*Neighbor tables* 34a⟩
   ⟨*Global variables* 13b⟩
   ⟨*Static function prototypes* 21⟩
   ⟨*Static functions* 24f⟩
   ⟨*Interface functions* 14b⟩
Root chunk (not used in this document).

This file is included into target-specific files to produce an object file. For each target we build its own C file in corresponding subsections below.

For some constants it is better to have symbolic names even if one can not easily change their values.

85b ⟨*Macro definitions* 85b⟩≡
```
      #define Nc  3      /* Number of colors */
      #define DIM 4      /* number of dimensions */
      #define Fd  4      /* Fermion representation dimension */
```
This code is used in chunks 86a, 88c, 91a, 93b, and 95a.

Strictly speaking, there is no need to have separate types for even/odd sublattices. But, while writing the CG, the compiler caught quite a few logic errors because of these two tiny structures.

When complex types are constructed in a trivial way from real types, this chunk is handy:

85c ⟨*Usual complex types* 85c⟩≡
```
      typedef struct {
          float re, im;
      } scalar_complex;

      typedef struct {
          vReal re, im;
      } vector_complex;
```
This code is used in chunks 86b, 88d, 91b, 93c, and 95b.

5.13.1 Single Precision SSE version

⟨*dwf-ssef.c* 86a⟩≡

    ⟨*Macro definitions* 85b⟩
    ⟨*SSE single precision types* 86b⟩
    ⟨*SSE single precision functions* 86c⟩
    #include "dwf-ssef.h"
    #define MACHINE "sse float"
    #define L3(n) MIT_ssef_##n
    #define PAD16(size) (15+(size))
    #define ALIGN16(addr) ((void *)(~15 & (15 + (size_t)(addr))))
    ⟨*Blocks for* YA *and* YB *of length four* 61c⟩
    #include "dwf.c"

Root chunk (not used in this document).

For single precision SSE, vectors are of length 4. Special gcc construct is needed to access SSE units.

⟨*SSE single precision types* 86b⟩≡

    #define Vs 4
    typedef float REAL;
    typedef REAL vReal __attribute__((mode(V4SF),aligned(16)));
    ⟨*Usual complex types* 85c⟩

This code is used in chunk 86a.

Ten functions dealing with vectors follow. First, propagate a scalar value into all four components of the SSE vector.

⟨*SSE single precision functions* 86c⟩≡

    static inline vReal
    vmk_1(double a)   /* return (a a ... a) */
    {
        float b = a;
        vReal v = __builtin_ia32_loadss((float *)&b);
        asm("shufps\t$0,%0,%0" : "+x" (v));
        return v;
    }

This definition is continued in chunks 86–88.
This code is used in chunk 86a.

Next, packing Vs-1 copies of a and one b:

⟨*SSE single precision functions* 86c⟩+≡

    static inline vReal
    vmk_n1(double a, double b)   /* return (a ... a b) */
    {
      vReal v;
      REAL *r = (REAL *)&v;
      r[0] = a; r[1] = a; r[2] = a; r[3] = b;
      return v;
    }

Also, packing one a and Vs-1 copies of b:

⟨*SSE single precision functions* 86c⟩+≡

    static inline vReal
    vmk_1n(double a, double b)   /* return (a b ... b) */
    {
      vReal v;
      REAL *r = (REAL *)&v;
      r[0] = a; r[1] = b; r[2] = b; r[3] = b;
      return v;
    }

And two more constructors:

⟨*SSE single precision functions* 86c⟩+≡

```
static inline vReal
vmk_fn(double a, double b)                    /* return (a a*b ... a*b^(Vs-1)) */
{
  vReal v;
  REAL *r = (REAL *)&v;
  r[0] = a; r[1] = a*b; r[2] = a*b*b; r[3] = a*b*b*b;
  return v;
}
```

⟨*SSE single precision functions* 86c⟩+≡

```
static inline vReal
vmk_bn(double a, double b)                    /* return (a^(Vs-1)*b ... a*b b) */
{
  vReal v;
  REAL *r = (REAL *)&v;
  r[0] = a*a*a*b; r[1] = a*a*b; r[2] = a*b; r[3] = b;
  return v;
}
```

Add all components of the vector together:

⟨*SSE single precision functions* 86c⟩+≡

```
static inline double
vsum(vReal v)                                 /* return sum(i, [i]v) */
{
  REAL *r = (REAL *)&v;
  return r[0] + r[1] + r[2] + r[3];
}
```

We need also to change the first element of the vector.

⟨*SSE single precision functions* 86c⟩+≡

```
static inline vReal
vput_0(vReal a, double b)                     /* return (b [1]a ... [Vs-1]a) */
{
  REAL *v = (REAL *)&a;
  v[0] = b;
  return a;
}
```

and the last element:

⟨*SSE single precision functions* 86c⟩+≡

```
static inline vReal
vput_n(vReal a, double b)                      /* return ([0]a ... [Vs-2]a b) */
{
  REAL *v = (REAL *)&a;
  v[3] = b;
  return a;
}
```

Shift to the left:

88a ⟨*SSE single precision functions* 86c⟩+≡
```
static inline vReal
shift_up1(vReal a, vReal b)                /* return ([1]a ... [Vs-1]a [0]b) */
{
   vReal z;
   REAL *X = (REAL *)&a;
   REAL *Y = (REAL *)&b;
   REAL *Z = (REAL *)&z;

   Z[0] = X[1]; Z[1] = X[2]; Z[2] = X[3]; Z[3] = Y[0];
   return z;
}
```

And to the right:

88b ⟨*SSE single precision functions* 86c⟩+≡
```
static inline vReal
shift_upN(vReal a, vReal b)                /* return ([Vs-1]a [0]b ... [Vs-2]b) */
{
   vReal z;
   REAL *X = (REAL *)&a;
   REAL *Y = (REAL *)&b;
   REAL *Z = (REAL *)&z;

   Z[0] = X[3]; Z[1] = Y[0]; Z[2] = Y[1]; Z[3] = Y[2];
   return z;
}
```

5.13.2 Double Precision SSE version

88c ⟨*dwf-ssed.c* 88c⟩≡
```
   ⟨Macro definitions 85b⟩
   ⟨SSE double precision types 88d⟩
   ⟨SSE double precision functions 89a⟩
   #include "dwf-ssed.h"
   #define MACHINE "sse double"
   #define L3(n) MIT_ssed_##n
   #define PAD16(size) (15+(size))
   #define ALIGN16(addr) ((void *)(~15 & (15 + (size_t)(addr))))
   ⟨Blocks for YA and YB of length two 61f⟩
   #include "dwf.c"
```
Root chunk (not used in this document).

For double precision SSE, vectors are of length 4. Special gcc construct is needed to access SSE units.

88d ⟨*SSE double precision types* 88d⟩≡
```
   #define Vs 2
   typedef double REAL;
   typedef double __attribute__((mode(V2DF),aligned(16))) vReal;
   ⟨Usual complex types 85c⟩
```
This code is used in chunk 88c.

Ten functions dealing with vectors follow. First, propagate a scalar value into all two components of the SSE vector.

89a ⟨*SSE double precision functions* 89a⟩≡
```
static inline vReal
vmk_1(double a)                                    /* return (a a ... a) */
{
    vReal v;
    REAL *w = (REAL *)&v;
    w[0] = w[1] = a;
    return v;
}
```
This definition is continued in chunks 89 and 90.
This code is used in chunk 88c.

Next, packing `Vs-1` copies of `a` and one `b`:

89b ⟨*SSE double precision functions* 89a⟩+≡
```
static inline vReal
vmk_n1(double a, double b)                          /* return (a ... a b) */
{
  vReal v;
  REAL *r = (REAL *)&v;
  r[0] = a; r[1] = b;
  return v;
}
```

Also, packing one `a` and `Vs-1` copies of `b`:

89c ⟨*SSE double precision functions* 89a⟩+≡
```
static inline vReal
vmk_1n(double a, double b)                          /* return (a b ... b) */
{
  vReal v;
  REAL *r = (REAL *)&v;
  r[0] = a; r[1] = b;
  return v;
}
```

And two more constructors:

89d ⟨*SSE double precision functions* 89a⟩+≡
```
static inline vReal
vmk_fn(double a, double b)                   /* return (a a*b ... a*b^(Vs-1)) */
{
  vReal v;
  REAL *r = (REAL *)&v;
  r[0] = a; r[1] = a*b;
  return v;
}
```

89e ⟨*SSE double precision functions* 89a⟩+≡
```
static inline vReal
vmk_bn(double a, double b)                   /* return (a^(Vs-1)*b ... a*b b) */
{
  vReal v;
  REAL *r = (REAL *)&v;
  r[0] = a*b; r[1] = b;
  return v;
}
```

Add all components of the vector together:

90a   ⟨*SSE double precision functions* 89a⟩+≡

```
static inline double
vsum(vReal v)                                    /* return sum(i, [i]v) */
{
   REAL *r = (REAL *)&v;
   return r[0] + r[1];
}
```

We need also to change the first element of the vector.

90b   ⟨*SSE double precision functions* 89a⟩+≡

```
static inline vReal
vput_0(vReal a, double b)                 /* return (b [1]a ... [Vs-1]a) */
{
   REAL *v = (REAL *)&a;
   v[0] = b;
   return a;
}
```

and the last element:

90c   ⟨*SSE double precision functions* 89a⟩+≡

```
static inline vReal
vput_n(vReal a, double b)                 /* return ([0]a ... [Vs-2]a b) */
{
   REAL *v = (REAL *)&a;
   v[1] = b;
   return a;
}
```

Shift to the left:

90d   ⟨*SSE double precision functions* 89a⟩+≡

```
static inline vReal
shift_up1(vReal a, vReal b)               /* return ([1]a ... [Vs-1]a [0]b) */
{
   vReal r;
   REAL *x = (REAL *)&a;
   REAL *y = (REAL *)&b;
   REAL *z = (REAL *)&r;
   z[0] = x[1];
   z[1] = y[0];

   return r;
}
```

And to the right:

90e   ⟨*SSE double precision functions* 89a⟩+≡

```
static inline vReal
shift_upN(vReal a, vReal b)               /* return ([Vs-1]a [0]b ... [Vs-2]b) */
{
   vReal r;
   REAL *x = (REAL *)&a;
   REAL *y = (REAL *)&b;
   REAL *z = (REAL *)&r;
   z[0] = x[1];
   z[1] = y[0];

   return r;
}
```

5.13.3 Single Precision AltiVec version

⟨*dwf-altivecf.c* 91a⟩≡

```
   ⟨Macro definitions 85b⟩
   ⟨AltiVec single precision types 91b⟩
   ⟨AltiVec single precision functions 91c⟩
   #include "dwf-altivecf.h"
   #define MACHINE "altivec float"
   #define L3(n) MIT_altivecf_##n
   #define PAD16(size) (15+(size))
   #define ALIGN16(addr) ((void *)(~15 & (15 + (size_t)(addr))))
   ⟨Blocks for YA and YB of length four 61c⟩
   #include "dwf.c"
```

Root chunk (not used in this document).

The AltiVec extension of the PPC architecture is quite similar to SSE. It requires the `-maltivec` flag for gcc.

⟨*AltiVec single precision types* 91b⟩≡

```
   #include <altivec.h>
   typedef float REAL;
   typedef vector float vReal;
   #define Vs 4
   ⟨Usual complex types 85c⟩
```

This code is used in chunk 91a.

Now, the ten vector functions.
First, propagate a scalar value into all four components of the SSE vector.

⟨*AltiVec single precision functions* 91c⟩≡

```
   static inline vReal
   vmk_1(double a)                                   /* return (a a ... a) */
   {
     vReal v;
     REAL *r = (REAL *)&v;
     r[0] = a; r[1] = a; r[2] = a; r[3] = a;
     return v;
   }
```

This definition is continued in chunks 91–93.
This code is used in chunk 91a.

Next, packing `Vs-1` copies of `a` and one `b`:

⟨*AltiVec single precision functions* 91c⟩+≡

```
   static inline vReal
   vmk_n1(double a, double b)                        /* return (a ... a b) */
   {
     vReal v;
     REAL *r = (REAL *)&v;
     r[0] = a; r[1] = a; r[2] = a; r[3] = b;
     return v;
   }
```

Also, packing one `a` and `Vs-1` copies of `b`:

⟨*AltiVec single precision functions* 91c⟩+≡

```
   static inline vReal
   vmk_1n(double a, double b)                        /* return (a b ... b) */
   {
     vReal v;
     REAL *r = (REAL *)&v;
     r[0] = a; r[1] = b; r[2] = b; r[3] = b;
     return v;
   }
```

And two more constructors:

92a   ⟨*AltiVec single precision functions* 91c⟩+≡
```
static inline vReal
vmk_fn(double a, double b)                    /* return (a a*b ... a*b^(Vs-1)) */
{
  vReal v;
  REAL *r = (REAL *)&v;
  r[0] = a; r[1] = a*b; r[2] = a*b*b; r[3] = a*b*b*b;
  return v;
}
```

92b   ⟨*AltiVec single precision functions* 91c⟩+≡
```
static inline vReal
vmk_bn(double a, double b)                    /* return (a^(Vs-1)*b ... a*b b) */
{
  vReal v;
  REAL *r = (REAL *)&v;
  r[0] = a*a*a*b; r[1] = a*a*b; r[2] = a*b; r[3] = b;
  return v;
}
```

Add all components of the vector together:

92c   ⟨*AltiVec single precision functions* 91c⟩+≡
```
static inline double
vsum(vReal v)                                 /* return sum(i, [i]v) */
{
  REAL *r = (REAL *)&v;
  return r[0] + r[1] + r[2] + r[3];
}
```

We need also to change the first element of the vector.

92d   ⟨*AltiVec single precision functions* 91c⟩+≡
```
static inline vReal
vput_0(vReal a, double b)                     /* return (b [1]a ... [Vs-1]a) */
{
  REAL *v = (REAL *)&a;
  v[0] = b;
  return a;
}
```

and the last element:

92e   ⟨*AltiVec single precision functions* 91c⟩+≡
```
static inline vReal
vput_n(vReal a, double b)                     /* return ([0]a ... [Vs-2]a b) */
{
  REAL *v = (REAL *)&a;
  v[3] = b;
  return a;
}
```

Shift to the left:

92f   ⟨*AltiVec single precision functions* 91c⟩+≡
```
static inline vReal
shift_up1(vReal a, vReal b)                   /* return ([1]a ... [Vs-1]a [0]b) */
{
  return vec_sld(a, b, 4);
}
```

And to the right:

⟨*AltiVec single precision functions* 91c⟩+≡

```
static inline vReal
shift_upN(vReal a, vReal b)                /* return ([Vs-1]a [0]b ... [Vs-2]b) */
{
    return vec_sld(a, b, 12);
}
```

5.13.4   Single Precision BlueLight version

⟨*dwf-bluelightf.c* 93b⟩≡

```
⟨Macro definitions 85b⟩
⟨BlueLight single precision types 93c⟩
⟨BlueLight functions 93d⟩
#include "dwf-bluelightf.h"
#define MACHINE "bluelight float"
#define L3(n) MIT_bluelightf_##n
#define PAD16(size) (15+(size))
#define ALIGN16(addr) ((void *)(~15 & (15 + (size_t)(addr))))
⟨Blocks for YA and YB of length two 61f⟩
#include "dwf.c"
```

Root chunk (not used in this document).

The BlueLight extension of the PPC architecture is for the BlueGene/L. It requires the `-mbluelight` flag for gcc.

⟨*BlueLight single precision types* 93c⟩≡

```
#include <bluelight.h>
typedef float REAL;
typedef vector float vReal;
#define Vs 2
⟨Usual complex types 85c⟩
```

This code is used in chunk 93b.

Now, the ten vector functions.
First, propagate a scalar value into all four components of the SSE vector.

⟨*BlueLight functions* 93d⟩≡

```
static inline vReal
vmk_1(double a)                              /* return (a a ... a) */
{
    return (vReal)(vec_mk2d(a, a));
}
```

This definition is continued in chunks 93 and 94.
This code is used in chunks 93b and 95a.

Next, packing `Vs-1` copies of `a` and one `b`:

⟨*BlueLight functions* 93d⟩+≡

```
static inline vReal
vmk_n1(double a, double b)                    /* return (a ... a b) */
{
  return (vReal)(vec_mk2d(a, b));
}
```

Also, packing one `a` and `Vs-1` copies of `b`:

⟨*BlueLight functions* 93d⟩+≡

```
static inline vReal
vmk_1n(double a, double b)                    /* return (a b ... b) */
{
  return (vReal)(vec_mk2d(a, b));
}
```

And two more constructors:

94a   ⟨*BlueLight functions* 93d⟩+≡
```
static inline vReal
vmk_fn(double a, double b)                    /* return (a a*b ... a*b^(Vs-1)) */
{
  return (vReal)(vec_mk2d(a, a*b));
}
```

94b   ⟨*BlueLight functions* 93d⟩+≡
```
static inline vReal
vmk_bn(double a, double b)                    /* return (a^(Vs-1)*b ... a*b b) */
{
  return (vReal)(vec_mk2d(a*b, b));
}
```

Add all components of the vector together:

94c   ⟨*BlueLight functions* 93d⟩+≡
```
static inline double
vsum(vReal v)                                      /* return sum(i, [i]v) */
{
    return vec_get0((vector double)v) + vec_get1((vector double)v);
}
```

We need also to change the first element of the vector.

94d   ⟨*BlueLight functions* 93d⟩+≡
```
static inline vReal
vput_0(vReal a, double b)                     /* return (b [1]a ... [Vs-1]a) */
{
    return (vReal)(vec_mk2d(b, vec_get1((vector double)a)));
}
```

and the last element:

94e   ⟨*BlueLight functions* 93d⟩+≡
```
static inline vReal
vput_n(vReal a, double b)                     /* return ([0]a ... [Vs-2]a b) */
{
    return (vReal)(vec_mk2d(vec_get0((vector double)a), b));
}
```

Shift to the left:

94f   ⟨*BlueLight functions* 93d⟩+≡
```
static inline vReal
shift_up1(vReal a, vReal b)                /* return ([1]a ... [Vs-1]a [0]b) */
{
    return (vReal)(vec_mk10((vector double)a, (vector double)b));
}
```

And to the right:

94g   ⟨*BlueLight functions* 93d⟩+≡
```
static inline vReal
shift_upN(vReal a, vReal b)             /* return ([Vs-1]a [0]b ... [Vs-2]b) */
{
    return (vReal)(vec_mk10((vector double)a, (vector double)b));
}
```

5.13.5 Double Precision BlueLight version

There is very little difference between single and double precision BlueLight implementations.

95a ⟨*dwf-bluelightd.c* 95a⟩≡
```
⟨Macro definitions 85b⟩
⟨BlueLight double precision types 95b⟩
⟨BlueLight functions 93d⟩
#include "dwf-bluelightd.h"
#define MACHINE "bluelight double"
#define L3(n) MIT_bluelightd_##n
#define PAD16(size) (15+(size))
#define ALIGN16(addr) ((void *)(~15 & (15 + (size_t)(addr))))
⟨Blocks for YA and YB of length two 61f⟩
#include "dwf.c"
```
Root chunk (not used in this document).

The BlueLight extension requires the `-mbluelight` flag for gcc.

95b ⟨*BlueLight double precision types* 95b⟩≡
```
#include <bluelight.h>
typedef double REAL;
typedef vector double vReal;
#define Vs 2
⟨Usual complex types 85c⟩
```
This code is used in chunk 95a.

6 CHUNKS